3D Smoke Simulation and Rendering on a Custom GPU Pipeline with Godot

Alexis Cordelle, Antonin Granados, Pierre Gueguen, Baptiste Girardin, Livio Personne
Telecom Paris, Palaiseau, France
{alexis.cordelle, antonin.granados, pierre.gueguen, baptiste.girardin,
livio.personne}@telecom-paris.fr

Figure 1: Examples of the result of our simulation and rendering pipeline running in a Godot scene in real-time.

***Abstract* — In this report, we present our interactive 3D smoke simulation developed with the Godot game engine using a custom compute pipeline. Our pipeline implements the Lattice-Boltzmann Method (LBM) to achieve a stable and physically accurate simulation. We leverage 3D textures to store smoke density and the voxelized scene directly in GPU memory; these textures are updated using compute shaders and rendered through a fragment shader. This voxel-based approach enables the use of a ray marching algorithm to display the smoke in real time, with optimizations that provide realistic lighting while minimizing performance overhead. We describe the overall architecture of our pipeline, its integration into Godot, and the stages responsible for voxelization, simulation, and rendering. Our work demonstrates the feasibility of integrating advanced volumetric effects into Godot through GPU-based techniques.**

Keyword* — *Godot Engine, Lattice-Boltzmann Method, GPU Pipeline, Smoke Simulation, Real-time Rendering

1. Introduction

Real-time smoke simulation plays an important role in modern visual effects, either in animations, interactive applications, or video games. However, implementing such a simulation in a general-purpose engine like Godot presents several challenges: accessing low-level graphics APIs (such as Vulkan), and achieving the performance required for real-time or interactive rendering.

In this project, we aim to simulate dynamic smoke within the Godot game engine using a fully custom GPU pipeline based on compute and fragment shaders, along with 3D voxel data. Rather than relying on traditional particle systems, we represent the smoke as a 3D texture (voxel grid) and simulate its evolution entirely on the GPU. This allows us to implement the Lattice-Boltzmann Method (LBM), which enables stable and physically meaningful modeling of turbulence and diffusion. The same voxel data is also used for rendering the smoke using a ray marching algorithm.

To support this simulation, we developed two Vulkan compute pipelines integrated into Godot. These pipelines handle the voxelization and simulation stages, updating the volumetric data every frame. The resulting data is then passed to a fragment shader that performs ray marching through a bounding box enclosing the smoke volume. This approach improves performance by limiting computation to relevant screen regions and allows for convincing light–smoke interactions.

This report presents the design and implementation of our system. We detail our compute pipeline architecture, describe the voxelization, simulation, and rendering stages, and evaluate the visual results and performance. Our work demonstrates the viability of integrating advanced GPU-based volumetric effects into Godot for interactive real-time applications.

2. Related Work

i. Smoke Simulation

The Lattice–Boltzmann Method (LBM) is well-suited for GPU due to its data-parallel nature. Academic implementations have shown strong results: for instance, efficient LBM schemes optimized for GPU can deliver high frame rates and very high physical accuracy [1].

ii. Compute Pipeline in Game Engines

Mainstream engines like Unity and Unreal often simulate volumetric effects with particles or screen-space fog. Full GPU-driven LBM fluid simulation is rare in real-time contexts due to computational cost.

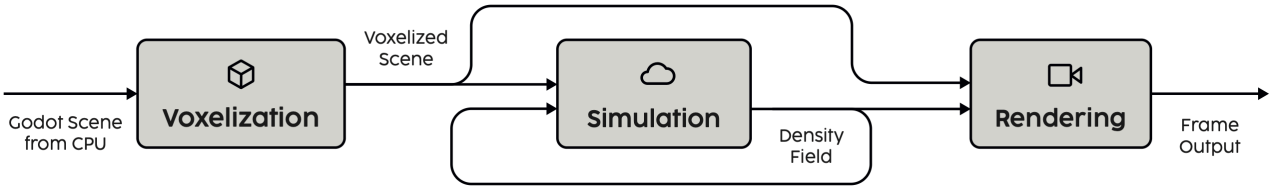

Figure 2: There are 3 stages in our pipeline: voxelization, simulation, and rendering. Each one has its own shader and run entirely on the GPU. All the data passed between the stages is stored on the GPU memory and accessed without any CPU intervention.

In Godot, volumetric fluid simulation is not available natively. Community approaches rely on fragment shaders and 3D textures for simple ray marched volumes [2], but lack full compute-based physics.

iii. Volume ray marching

Volumetric ray marching iteratively samples density along a ray through a volume. Techniques like constant-step ray marching, ray termination, and bounding volumes improve performance. Using a bounding box to constrain ray marching to a smoke volume reduces shading cost and enables more efficient screen-space evaluation. Shaders can also integrate lighting interactions via nested ray marching and Beer–Lambert law for realistic volumetric illumination [3], [4].

iv. Other Simulation Tools

EmberGen [5] is a standalone tool that simulates fire, smoke, and explosions in real time, offering GPU-accelerated hybrid voxel/particle systems. It can export simulation results for game engines and could potentially work with our rendering shader. In contrast, Blender supports offline volumetric effects with Cycles and Eevee, but it is not optimized for interactive use. Volumetric smoke can be slow and unstable in real-time previews, making it better suited for rendered animations .

3. Pipeline Architecture

A. Motivation

We chose to use the LBM algorithm because it is able to simulate a wide range of fluids, but this algorithm can become computationally expensive for large simulations. However, it is made to use a voxel grid (3D grid) where each voxel is computed only using the previous frame, avoiding the need to use multiple shader stages per update — unlike the stable fluid algorithm [6] that is used in other solvers. Thus, we can leverage GPU parallelism by dispatching a compute thread per voxel.

Additionally, rendering directly in GPU memory minimizes costly CPU–GPU data transfers and maximizes real-time performance.

B. General Structure

Our simulation and rendering pipeline is composed of three different stages (Fig. 2), each one as its own shader (LBM allows us to only use one for the simulation). We use two compute shaders for the per-voxel stages that are dispatched on the physics thread and one fragment shader for the per-pixel stage that is running on the main thread.

Here we will describe briefly the role of each stage (and go in more detail in the following sections) and their interactions with the pipeline.

i. Voxelization

This stage converts the scene into a voxel grid thus being one of our compute stages. Our scenes are made of multiple meshes made of triangles, and this shader will fill all the voxels that are inside an object. It will also determine if a voxel is a collider or a source and its speed based on the mesh properties and movement. For efficiency we store all the vertices and triangle indices on the GPU at the start of the simulation, and while the simulation is running we only need to pass the transformation matrix and the speed/rotation speed of each object. It outputs a 3D texture where the RGB components represent the speed while the alpha represents the type (empty, collider or source).

ii. Simulation

This is our second compute stage and it does one simulation step when called. We need to store a lot of data for LBM to work, and because we used the D3Q27 version, some fields have 27 dimensions, so we can't store them in 3D textures (limited to 4 fields). For those, we used multiple Shader Storage Buffer Objects (SSBOs). We started by using only one, but we were quickly limited by the maximum size set by Vulkan, so we switched to one SSBO per field. A few 3D textures were still used, for the velocity field and the smoke, for instance. It will use the 3D texture computed by the voxelization stage as well as the previous frame's smoke texture.

iii. Rendering

This is our fragment shader, and as the name suggests it is responsible for drawing the smoke on the screen. As opposed to what ray marcher implementations usually do, we apply this shader to the bounding box of the smoke to process only the necessary fragments. The stage uses the 3D smoke texture to compute the light for each voxel and the 3D voxelized scene texture to help with the rendering of shadows. The output is the frame displayed on the screen.

C. Integration with Godot

The Godot Engine gives us tools to interact directly with the GPU through Vulkan. The Godot API `RenderingDevice` is built on top of Vulkan and give us an OpenGL-like interface to the GPU. The only Vulkan-specific structures we had to fill were the formats for all the uniforms.

We have to keep track of the different textures' `RID` (Resource ID); it is how we pass data between shaders. Before submitting a stage, we have to set the current `RID` from the output of one stage to the uniform input of the next.

Also, Godot provides us with `PhysicsProcess` and `Process` functions that run on two separate threads. The physics thread is executed at a fixed rate (50fps) in our case so we can perform

the simulation on it. When running the main thread, we gather the latest data computed on the physics thread and use it to render.

4. Voxelization

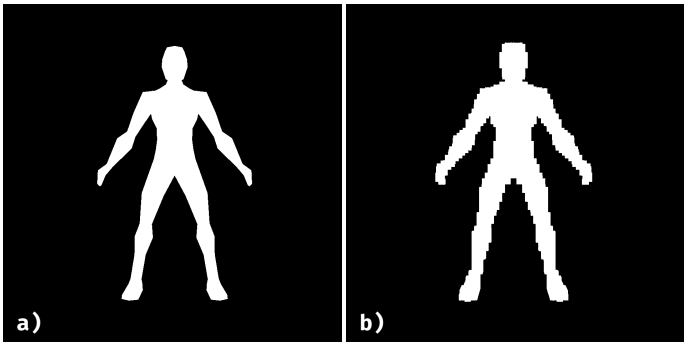

Figure 3: Unshaded scene composed of the low-poly mesh of a man — a) unshaded mesh, b) voxelized mesh

A. Overview

The goal of the first stage in our pipeline is to convert a Godot scene to a voxel grid (Fig. 3) so that the smoke can interact with it. For this to work, we have to fill every voxel that is contained in one of our objects (triangular meshes) and encode information that will be used by the next stages.

For moving objects, we have to update the voxel grid every frame, so we need to be as efficient as possible; otherwise, it would become a bottleneck for our simulation.

B. Data Structures

We use four SSBOs to store the scene data, and the rest of the meshes' information is passed through push constants. Our SSBOs are composed of the following data:

- *Vertex Buffer Object* (VBO) : which stores a list of `vec3` for each object; it is commonly used in a graphics pipeline to render an object.
- *Index Buffer Object* (IBO) : which stores a list of `int` for each object; it is also used in graphics pipeline.
- *Object Buffer* : which keeps track of the number of objects and for each object, the number of vertices/indices.
- *Bounding Box Buffer* : which as the name suggests, stores the Axis-Aligned Bounding Box (AABB) for each object to speed up the voxelization process.

We use one push constant per object because our voxelization iterates through all the objects one by one (one shader dispatch per object). We have to do it that way because we hit the Vulkan push constant size limit with only one object. In each push constant, we store the following data:

- `transform` (`mat4`) : the transformation matrix of the object.
- `lin_speed` (`vec3`) : the linear speed of the object.
- `rot_speed` (`vec3`) : the rotation speed of the object.
- `object_idx` (`int`) : the index of the object.
- `object_type` (`uint`) : the type of the object (collider/source).

Which is represented by 79 bytes and the Vulkan limit is set to 128 bytes.

Our implementation fills a 3D texture using the format `r32g32b32a32Sfloat` (4 floating point numbers with 32 bits precision), it is easier to keep track of this type of data in the Godot Engine. We can then pass the texture `RID` to the next stages.

C. Implementation

Our voxelization works in two parts:

i. Preprocessing

At the start of the application, we fill our SSBOs with the scene data. The VBO and IBO are filled directly using the Godot data. We then use the vertices to compute the AABBs and fill the Bounding Box Buffer. And the last one, Object Buffer, is pretty straightforward.

ii. Computation

This part is executed in each frame for moving objects. We iterate through all the objects one by one and compute their linear and angular speeds. We then fill the push constant with the computed data and dispatch the shader.

The shader starts by computing the true bounding box — the object might have rotated or translated and the precomputed AABB might not be aligned anymore. It transforms the AABB using the transformation matrix and uses that to compute the true bounding box.

For each voxel inside of the true AABB, we cast a ray from the voxel's center and count the number of triangle intersections using the Möller-Trumbore algorithm [7]. If the count is even, the voxel is outside the object; otherwise, it is inside.

We then compute the RGB component of our texture by adding the object's linear velocity to the angular velocity at the voxel's center. The alpha component is set to 1.0 for a collider object and to 2.0 for a source object (0.0 by default).

D. Performance Considerations

The voxelization is still very computationally expensive. That's why we decided to only execute it once at the start of the application when we only use static objects.

If the object is moving, we can still run it each frame, but it is more efficient to only have the part that will interact with the smoke set to be voxelized. For instance in the dragon scene (Fig. 1), only the head of the dragon is voxelized. Furthermore, using a low resolution representation of the mesh is a lot more efficient and the mesh size is in any case limited by the maximum size of an SSBOs.

5. Simulation

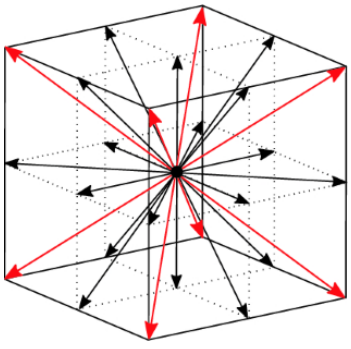

Figure 4: Representation of the D3Q27 directions as given in [8]

A. LBM Overview

LBM is a voxel-based and forward-iterative process method. It uses distributions to statistically approximate the proportion of particles going in different directions within a voxel. We use the D3Q27 scheme (Fig. 4) meaning that we have 27 discrete directions $i \in [\![0, 26]\!]$. Each direction has a corresponding vector $\boldsymbol{c_i} \in \{-1, 0, 1\}^3$ and a weight w_i from this scheme :

- Center : $w_{i\in\{13\}} = \frac{8}{27}$
- Faces : $w_{i\in\{4,10,12,14,16,22\}} = \frac{2}{27}$
- Edges : $w_{i\in\{1,3,5,7,9,11,15,17,19,21,23,25\}} = \frac{1}{54}$
- Corners : $w_{i\in\{0,2,6,8,18,20,24,26\}} = \frac{1}{216}$

The two main distributions we use are f_i for the air particles and g_i for the smoke particles. From these mesoscopic quantities we can compute the following macroscopic quantities :

- air density : $\rho = \sum_{i=1}^{27} f_i$
- air velocity : $\boldsymbol{u} = \frac{1}{\rho}\sum_{i=1}^{27} f_i \boldsymbol{c_i}$
- smoke density : $\varphi = \sum_{i=1}^{27} g_i$

To ensure stability, we have to make sure that at every frame $\boldsymbol{u}$ respects Mach's condition ($\frac{\|\boldsymbol{u}\|}{c_s} < 0.3$) [9].

LBM uses two main steps to simulate fluid physics, collision and streaming.

$$f_i^*(\boldsymbol{x}) \leftarrow f_i(\boldsymbol{x}) + \Omega(f_i(\boldsymbol{x})) + f_i^{\mathrm{F}}$$
$$f_i(\boldsymbol{x} + \boldsymbol{c_i}) \leftarrow f_i^*(\boldsymbol{x})$$

Ω is a collision operator which tends to push the different f_i proportions towards an equilibrium f_i^{eq} at a speed which depends on the relaxation time τ_ρ. f_i^{F} is the term responsible for applying external forces to our fluid like gravity :

$$\Omega = \frac{1}{\tau_\rho}(f_i^{\mathrm{eq}} - f_i)$$
$$f_i^{\mathrm{eq}} = w_i \rho \left(1 + \frac{\boldsymbol{c_i}\cdot\boldsymbol{u}}{c_s^2} + \frac{(\boldsymbol{c_i}\cdot\boldsymbol{u})^2}{2c_s^2} - \frac{\boldsymbol{u}\cdot\boldsymbol{u}}{2c_s}\right)$$
$$f_i^{\mathrm{F}} = w_i \frac{\boldsymbol{c_i}\cdot\boldsymbol{F}}{c_s^2}$$

For the g_i distribution the equations are essentially the same. The only differences are that there is no force term, τ_ρ is replaced by τ_φ and f_i is replaced by g_i. Both distributions use the same air velocity $\boldsymbol{u}$ for the streaming.

B. Boundaries & Sources

The voxelization shader hands to the simulation shader the type of each voxel. This allows us to treat voxels differently if they are smoke sources, colliders or in free space.

i. Colliders

For colliders, the collision step is skipped meaning particles don't interact and the streaming step is reversed.

We define $\bar{i} = 26 - i$ as the opposite direction of i.

$$f_{\bar{i}}(\boldsymbol{x} + \boldsymbol{c_{\bar{i}}}) \leftarrow f_i(\boldsymbol{x})$$

This effectively creates a bounce back condition on the border of collider objects. Even if this technique seems quite straightforward, it causes issues with moving objects.

ii. Smoke Sources

For smoke sources, we bypass the calculations for ρ and φ and force them to arbitrary ρ_0 and φ_0. u is also forced to u_{inlet} in the direction we want to set the source to create smoke. After that f_i^{eq} and g_i^{eq} can be calculated as normal in free space.

iii. Boundary Conditions

During the streaming step, we potentially need to access voxels outside the simulation zone. For this reason we implemented periodic boundary conditions. All voxel positions are considered modulo the size of the zone.

6. Rendering

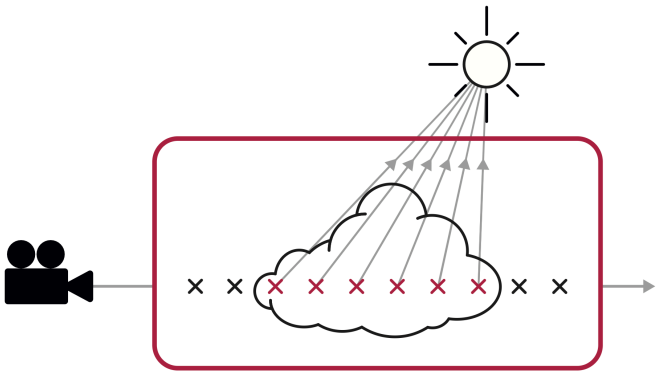

Figure 5: The ray marching algorithm with lighting represented in 2D

A. Ray Marching

The ray marching algorithm is a technique that we use to sample a volume's density in order to calculate the light's behavior accordingly. It shoots a ray from each fragment of the volume in the direction of the camera view through the volume. The ray marches a fixed step size before sampling the volume's density and repeating the process until it exits the volume. In Godot, we implement it on a spatial fragment shader running on the smoke's bounding box (Fig. 5).

B. Lighting & Shadowing

i. Transmittance

Once we can sample the density along the view, we can apply Beer-Lambert's law :

$$T(\boldsymbol{x}, \omega) = e^{-\int_{\mathcal{R}(\boldsymbol{x},\omega)} \sigma(\boldsymbol{p})} d\boldsymbol{p}$$

Where $\mathcal{R}(\boldsymbol{x}, \omega)$ is the ray starting from $\boldsymbol{x}$ in the direction ω, $\sigma(\boldsymbol{p})$ is the density at position $\boldsymbol{p}$, and $T(\boldsymbol{x}, \omega)$ is the transmittance of a smoke volume along $\mathcal{R}(\boldsymbol{x}, \omega)$. This can give us the transparency of the volume.

ii. Sunlight

To get the sunlight to play in the equation, we have to get the amount of light reaching each sample. To do that, we use the ray marching algorithm once again but this time from the sample to the sun. We then compute the color $C(\boldsymbol{x}, \omega_l)$ of the sample with the formula :

$$C(\boldsymbol{x}, \omega_l) = C_l e^{-\int_{\mathcal{R}(\boldsymbol{x},\omega_l)} \sigma(\boldsymbol{y}) d\boldsymbol{y}}$$

Where ω_l is the direction facing the light l, and C_l is its color. To enhance the illumination of the volume, we use the the Henyey-Greenstein (HG) phase scattering function [10] which enables us to simulate effects such as forward and backward scattering. We approximate this function for performance reasons with the function :

$$p_{HG}(\cos(\theta), g) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g\cos(\theta))^2}$$

Where g is between -1 and 1, ranging from backscattering through isotropic scattering to forward scattering [11]. We actually sum it multiple times with different values of g to get more realistic results :

$$\begin{aligned} p(\cos(\theta)) = 0.07 p_{HG}(\cos(\theta), 0.9) \\ + 0.4 p_{HG}(\cos(\theta), 0.2) \\ + 0.43 p_{HG}(\cos(\theta), -0.5) \end{aligned}$$

Finally, we compute the color of each fragment with the formula :

$$L(\boldsymbol{x}, \omega) = \int_{\mathcal{R}(\boldsymbol{x}, \omega)} T(\boldsymbol{p}, \omega)\sigma(\boldsymbol{p}) \sum_{l \in \mathcal{L}} p(\omega \cdot \omega_l) C(\boldsymbol{p}, \omega_l) d\boldsymbol{p}$$

where $\mathcal{L}$ is the set of all the lights in the scene.

iii. Shadowing

With the voxelized scene, we can easily detect any collision between a ray and an object. If the collision occurs, we stop the ray where it is. This enables us to achieve a pretty basic shadowing effet.

C. Debanding techniques

i. Ray randomization

Due to the ray marching algorithm sampling discretly, banding appears as shown in Fig. 6 a). To remove this effect, we randomize each ray's starting point over time. This gives very satisfying results but introduces some noise, see Fig. 6 b).

ii. Denoising

In order to hide the noise as much as possible, we use blue noise textures instead of white noise which was simulated with the randomization. The motivation for this is that when multiple blue noise textures are accumulated, they converge toward a uniform texture. This is very useful when used alongside some form of frame accumulation. For that, we use Godot's built in TAA algorithm.

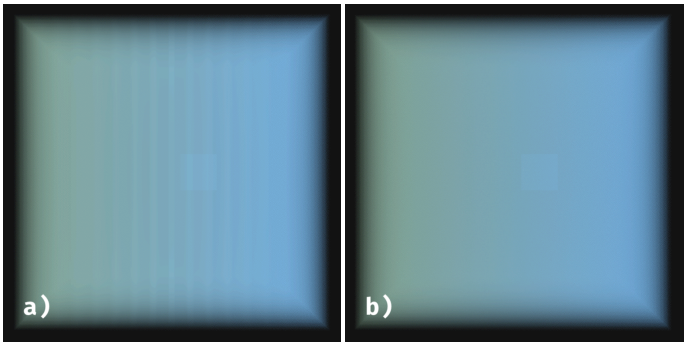

Figure 6: Entirely filled cube with smoke and illuminated by two lights — a) without debanding, b) with debanding

7. Results

A. Performance

Our voxelization stage remains the primary bottleneck. In the scene shown in Fig. 8, voxelization takes three times longer than rendering. For static scenes, we mitigate this by disabling voxelization after the initial frame (as in Fig. 8). For dynamic scenes, using low-poly versions of the meshes helps reduce the cost. A good enhancement would be to implement a BVH structure to accelerate this stage.

Rendering is pretty efficient, however, when the camera enters the smoke volume, the rendering behaves like a full-screen fragment shader, causing occasional lag spikes. Moreover, due to Godot's sync between the main thread and the physics thread, if the frame rate dips below 50fps, the simulation also slows down.

But on an Nvidia RTX 3060, we maintain a steady 60fps for scenes with up to ~2.1 million voxels, which is acceptable for most real-time applications.

B. Stability

Our simulation is generally stable, allowing a wide variety of smoke effects without noticeable artifacts.

Parameter control is robust, but some interdependencies make tuning sensitive. Choosing the wrong set of parameters can cause divergence, visible as abrupt density build-up or banding within only a few frames.

While minor visual artefacts can be mitigated during final rendering, fundamentally they remain simulation artifacts that cannot be fully corrected.

C. Limitations

Because we store some fields for the simulation in SSBOs, we are limited by their inherent maximum size defined by the Vulkan API. Thus, we cannot have more than around 16 million voxels (a cube of side 256). But even if that wasn't the case, we would still have to deal with the drop in performance.

A way to solve this issue would be to split each field into multiple SSBOs, but this would require a significant amount of work to keep track of the data and fetch/write at the correct place.

8. Conclusion

Our implementation shows that real-time volumetric smoke simulation using LBM and ray marching is feasible within a modern game engine like Godot. Despite some limitations related to voxelization and memory constraints, the system runs efficiently and produces convincing visuals on a midrange GPU.

In the future, further optimizations could significantly improve scalability. This opens up exciting possibilities for integrating more advanced smoke simulations directly into interactive applications and games. Creating a full plugin would also make the user experience far better.

References

[1] W. Jinghuan and M. Huimin, *Real-time smoke simulation based on vorticity preserving lattice Boltzmann method*. 2018.

[2] F. Harits Nur, *Volumetric Rendering in Godot 4: Ray-Marching and Visualizing 3D Noise*. 2023.

[3] F. Häggström, *Real-time rendering of volumetric clouds*. 2018.

[4] Patapom and Bomb!, *Real-Time Volumetric Rendering*. 2013.

[5] JangaFX, *EmberGen Documentation*. JangaFX, 2025.

[6] H. Mark, *Fast Fluid Dynamics Simulation on the GPU*.

[7] M. Thomas and T. Ben, *Fast, Minimum Storage Ray-Triangle Intersection*. 2012. doi: https://doi.org/10.1080/10867651.1997.10487468.

[8] G. Martin, B. Amir, and R. Thomas, *Performance of Under-Resolved, Model-Free LBM Simulations in Turbulent Shear Flows*. Springer, Cham, 2019. doi: https://doi.org/10.1007/978-3-030-27607-2_1.

[9] H. Tobias, T. Hatem, V. Lucien, R. Denis, and L. Emmanuel, *Consistent time-step optimization in the lattice Boltzmann method*. 2022. doi: https://doi.org/10.1016/j.jcp.2022.111224.

[10] H. Louis and G. Jesse, *Diffuse radiation in the Galaxy*. 1941.

[11] H. Patrick, *The Henyey-Greenstein phase function*. 2021.

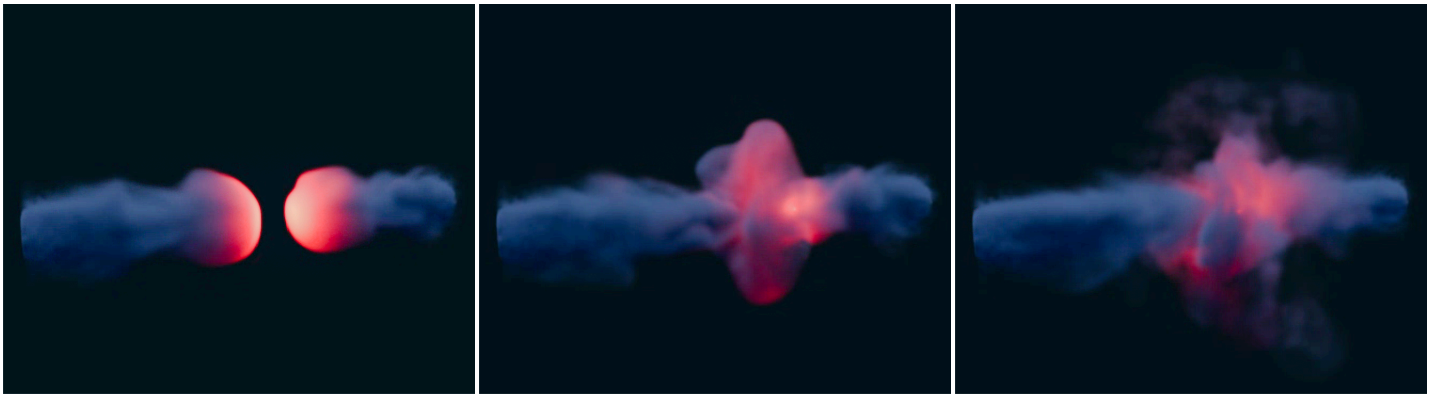

Figure 7: Evolution of two smoke plumes colliding over time with a point light in the middle

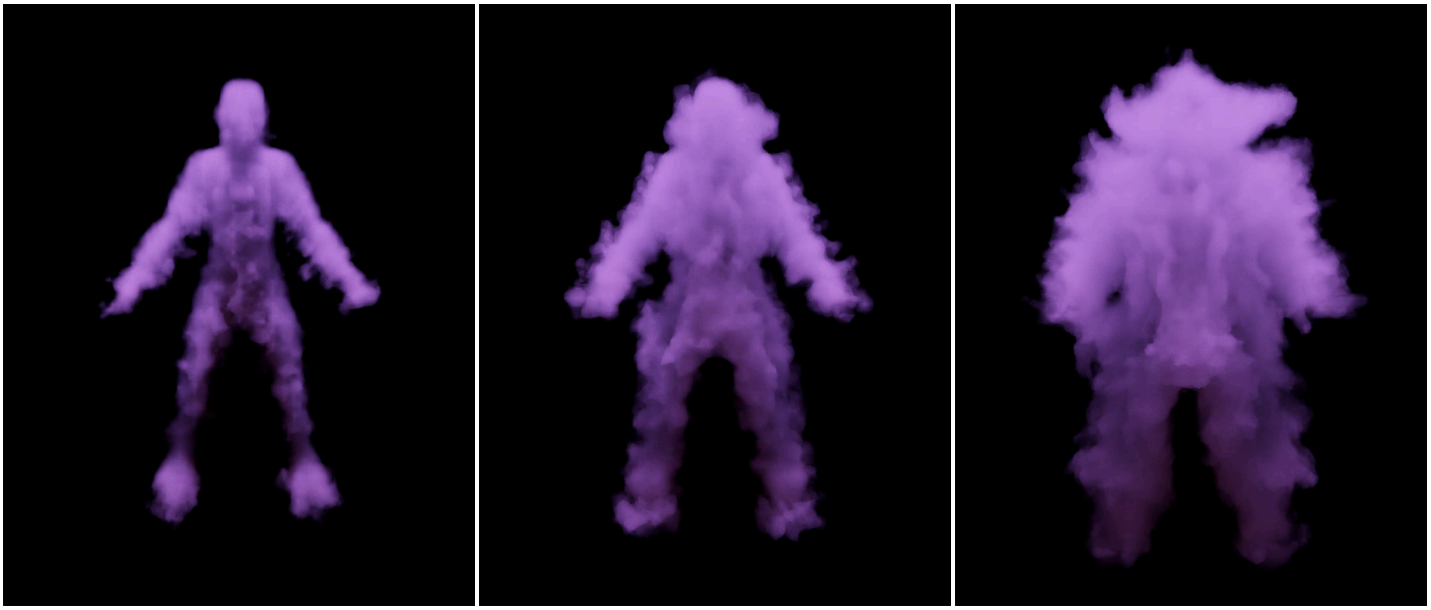

Figure 8: Evolution of ρ_0 with a human shaped source

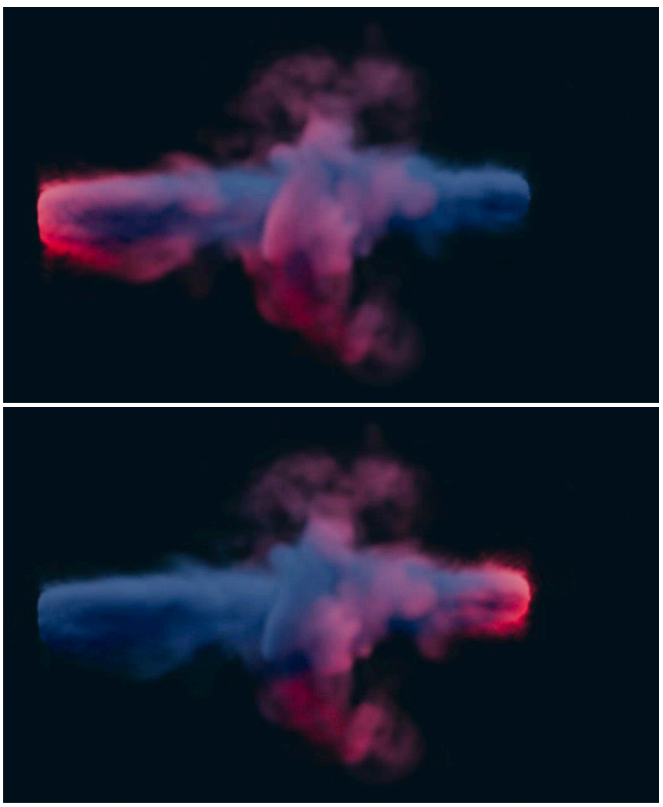

Figure 9: Point light interaction with two smoke plumes

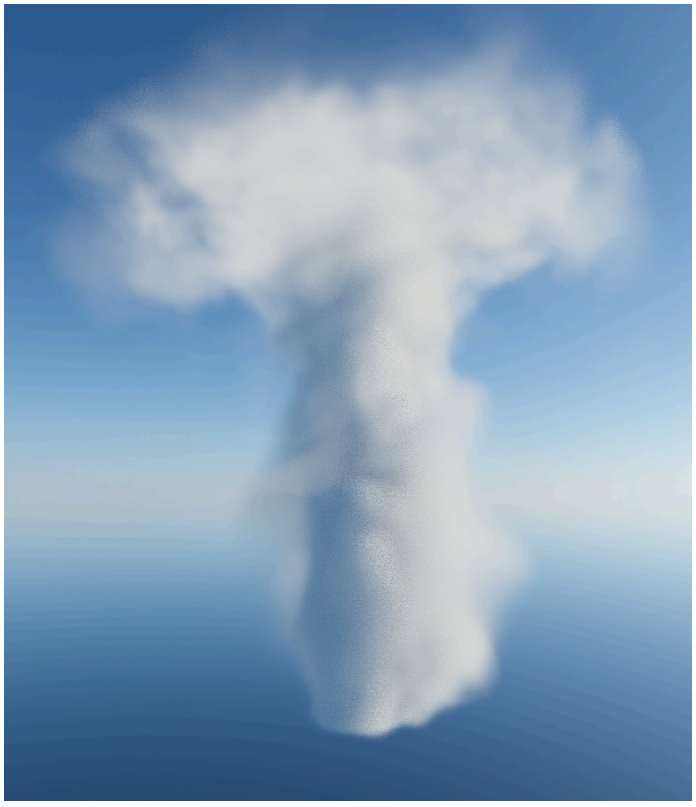

Figure 10: Rendering of a cloud-like smoke plume

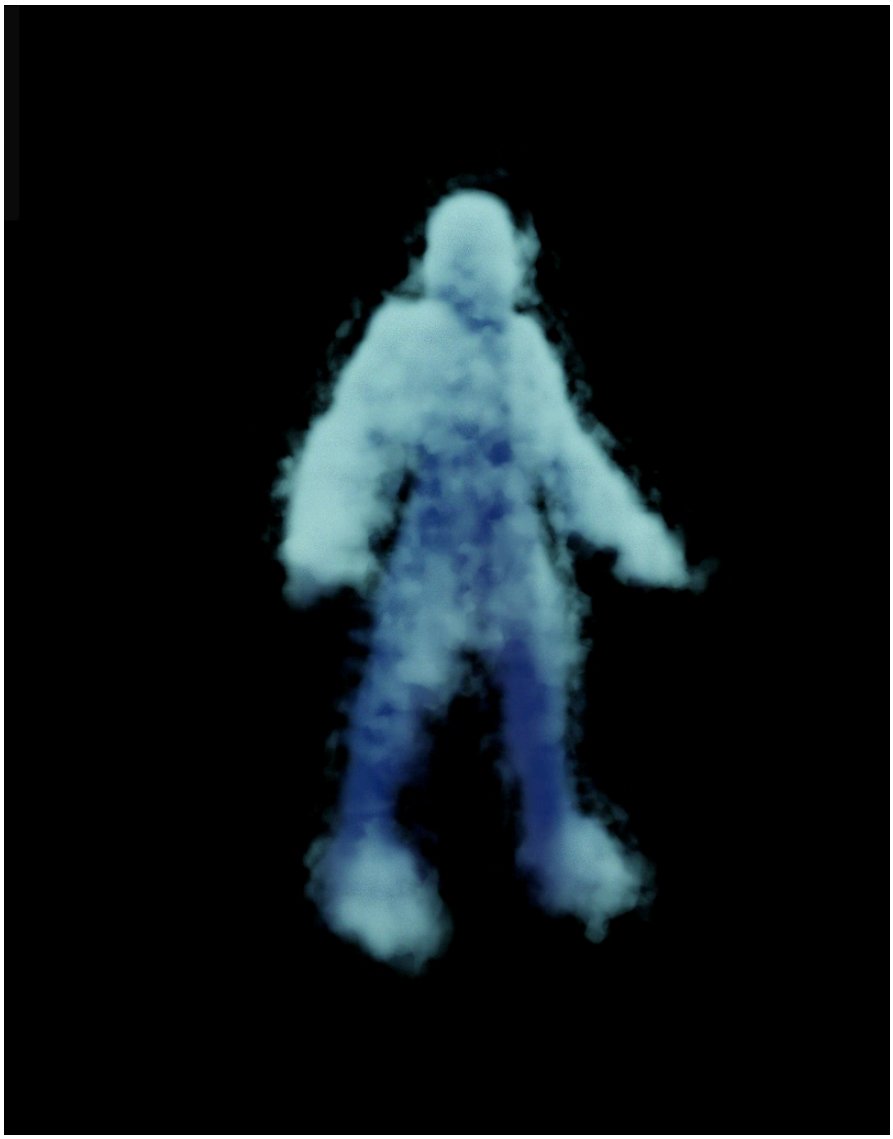

Figure 11: Human shaped source with blue smoke

Figure 12: Smoke coming out of the Stanford Dragon's mouth, with collision inside the dragon's mouth