ISSN 2464-4617 (print)
ISSN 2464-4625 (online)
Computer Science Research Notes - CSRN
http://www.wscg.eu
WSCG 2025 Proceedings

Real-time volumetric explosion special effects for games

Marta Rašeta, Žiga Lesar, and Ciril Bohak
University of Ljubljana, Faculty of Computer and Information Science
Večna pot 113
1000, Ljubljana, Slovenia
mr4571@student.uni-lj.si, {ziga.lesar, ciril.bohak}@fri.uni-lj.si

Abstract

Video game special effects are usually created using sprites or simplistic triangle meshes with added textures, which is a good compromise between performance and quality. This paper presents an implementation of a real-time volumetric explosion effect that aims to enhance realism in video games while seamlessly integrating the volume into a 3D scene, allowing it to interact with the environment. The effect is rendered in a web environment using JavaScript and WebGPU. Our implementation uses ray marching for volume rendering, integrating a transfer function for color mapping, noise functions for adding natural variation, and bloom post-processing to enhance fire illumination. The contribution of this paper is to demonstrate the rendering possibilities and growing support of volumetric special effects for more interactive and immersive web applications and video games.

Keywords

real-time rendering, volumetric effects, WebGPU, ray marching, game development

1 INTRODUCTION

One of the most significant challenges in computer graphics remains the creation of realistic real-time volumetric effects, particularly in interactive applications and video games. Despite advancements in graphics technology, volumetric effects are computationally intensive and demand more system resources. Practical methods like polygon meshes and billboards have great performance but lack volumetric depth, light scattering, and natural interaction with the environment. This compromise limits the realism and immersion of special effects.

Recent advancements in GPUs and graphics APIs have expanded the possibilities for volumetric effects, enabling more realistic and efficient rendering techniques. WebGPU [3], a new standard for web-based graphics, is currently the only web-based API that supports compute shaders, making it easier to implement volumetric effects. This paper explores the potential of WebGPU for rendering realistic volumetric effects. It was chosen for its cross-platform compatibility and accessibility through web browsers. By developing a real-time volumetric explosion effect, this work aims to demonstrate how it is possible to achieve both realism and computational efficiency. The full implementation is available at `https://github.com/makieew/webgpu-volumetric-explosion`.

The main contributions of this paper are:

- a WebGPU-based real-time implementation of animated volumetric explosion effects using pre-simulated volumetric data, enhanced with procedural noise and post-processing;
- evaluation and analysis of performance across different hardware configurations;
- user study on perceived realism of the implemented effects.

The rest of the paper is structured as follows: we first present the related work in Section 2; next we present our approach in Section 3, followed by Section 4 where we present the performance analysis and user study and their outcomes. We discuss the results and present the limitations of our approach in Section 5, and finally conclude the paper with Section 6.

2 RELATED WORK

The field of volume rendering has evolved significantly over the past decades following the advancements in hardware and algorithm development. Initial approaches to volume rendering were limited by the lack of computational power and frameworks to model light interactions within the

medium. However, the advancement of more powerful GPUs, optimized algorithms, and efficient memory management enabled the development of more detailed and eventually real-time volumetric effects.

The work of Kajiya and Von Herzen [7] set a foundation for volume rendering. They mathematically formalized light transport within volumes, using ray tracing as a method to simulate light absorption, emission, and scattering. Because of hardware limitations at the time, their approach was extremely slow.

Later, Levoy [10] proposed an innovative method for rendering surfaces from volumetric datasets. Around the same time, volumetric data was displayed by converting it to geometric primitives using the marching cubes algorithm [12]. Levoy introduced a new technique where the volumetric data is rendered directly using ray casting, sampling the data at regular intervals along a ray and accumulating the samples. His work also provided new methods of data classification, using transfer functions to emphasize specific features within the data. A limitation of the method was its support for only emission and absorption without light scattering. Even so, Levoy's work was successful enough to be used in various scientific domains for the next few decades.

Guthe et al. [6] demonstrated efficient compression techniques for large-volume datasets while using interactive rendering on a commodity desktop PC. Their work highlighted the potential for achieving complex rendering even on commercial hardware.

Enhancing the realism of computer-generated effects is also a well-researched topic, especially in simulating natural phenomena like smoke, fire, and fluids. Perlin [15] developed the Perlin noise algorithm, which is a type of gradient noise that has many uses, often in the generation of procedural content. It is used to create effects with natural patterns and textures. Based on this development, Worley [17] provided a method, Worley noise, to simulate cellular structures widely used in organic and natural phenomena. In addition, curl noise, created by Bridson et al. [2], helped in making turbulent flow fields, which are essential for realistic fluid dynamics.

Fedkiw et al. [5] made a major contribution by developing a realistic smoke simulation that renders in real time. Their approach used inviscid Euler equations for improved efficiency and a physically consistent vorticity confinement term to capture the characteristics of smoke. Additionally, their model correctly handles the interaction of smoke with moving objects. This set a new standard for special effects in interactive applications and games.

In the field of volume rendering, Lesar et al. [9] presented a volumetric path tracing framework that uses web technologies to visualize volumetric data, making it more accessible and efficient. General web-based rendering frameworks have also shaped browser-based visualization. Babylon.js[1] and Three.js[2] are widely adopted frameworks that simplify the creation of interactive 3D web applications. Building on WebGL and WebGPU technologies, Bohak et al. [1] introduced RenderCore, a lightweight framework that supports deferred rendering. Together, these frameworks make it possible to visualize volumes interactively across different platforms.

Recent work, such as Nubis volumetric cloud system[3], has advanced the real-time rendering of voxel-based clouds. Krüger et al. [8] introduce techniques for rendering real-time volumetric phenomena like smoke, fire, and explosions, focusing on optimization for interactive applications. The fast vortex particle method for fluid-character interaction [14] presents an efficient approach for simulating dynamic fluid interactions with characters and environments. NVIDIA's FlameWorks[4] demonstrates a real-time fire simulation system that relies on particle-based techniques to generate convincing fire effects.

2.1 Volumetric Effects in Games

Volumetric effects simulate the interaction with light, scattering, emission, and absorption in a 3D medium. They are used to create realistic and atmospheric effects such as smoke, fire, fog, clouds, or explosions. Adding these effects significantly contributes to the game's ambient and mood, making it more alive.

Several modern games have pushed the boundaries of volumetric effects, showcasing their potential to enhance realism and immersion. For instance, *Horizon Zero Dawn*[5] has a volumetric cloud system[6] that creates dynamic skies that change with the time of day and weather conditions. These clouds are rendered in real time, interacting with the game's lighting system to produce soft shadows and atmospheric scattering, adding to the game's

[1] https://babylonjs.com
[2] https://threejs.org
[3] https://www.guerrilla-games.com/read/nubis-cubed
[4] https://developer.nvidia.com/flameworks
[5] Horizon Zero Dawn, Guerrilla Games, 2017
[6] https://advances.realtimerendering.com/s2015/

beautiful landscape. It uses ray marching with two levels of detail: a low-frequency cloud base and a high-frequency noise (Perlin, Worley, and curl). The lighting model incorporates Beer's Law for absorption, the Henyey-Greenstein phase function for scattering, and a "powdered sugar" effect for simulating the scattering of light within the cloud. To optimize performance, the shader reduces sample counts and reuses previous frame data via reprojection. This allows real-time rendering at 2 ms per frame by using a quarter-resolution buffer with upscaling techniques.

Similarly, *Red Dead Redemption 2*[7] features volumetric fog and clouds[8]. Fog rolls through valleys and reacts to light, creating stunning sunrises and sunsets. Like *Horizon Zero Dawn*, it uses ray marching and multi-scale noise (Perlin and Worley) to shape clouds, but with a focus on physically based lighting. To optimize performance, effects near the camera are stored in a voxel grid, while distant fog and clouds use ray marching with large step sizes. Shadows are handled with shadow maps and approximated transmittance for light scattering, while temporal reconstruction helps smooth out sampling artifacts and improve efficiency.

Unlike previous games with atmospheric realism, *Counter-Strike 2*'s[9] volumetric smoke is a gameplay feature. Throwing a smoke grenade creates a 3D smoke effect that can be manipulated, bullets and explosions create holes that reveal what is behind the smoke. While the exact implementation details are not publicly available, the effect likely relies on a voxel grid for simulation and ray marching for rendering. A flood-fill algorithm or similar approach may be used to propagate changes in the smoke's shape.

These games show the growing trend of using volumetric special effects. However, their implementation often requires significant optimization, skill, and time, making them a challenge for developers.

3 METHODS

3.1 System Overview

The voxel data is initially generated in Blender and then pre-processed in Python to align, pad, and export it in a format suitable for rendering. We reused an existing engine[10] that renders basic objects through a node system as the foundation for this implementation. A renderer is created that supports both solid objects and the volumetric explosion. Additional features such as noise, bloom, and temperature-based color mapping are applied to enhance the visual effect. These techniques work together to produce a dynamic and visually compelling explosion effect. Figure 1 illustrates the steps of the technique, from voxel data generation to rendering.

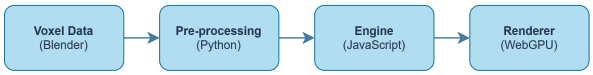


Figure 1: Diagram illustrating the process from voxel data generation in Blender to rendering the volumetric explosion using WebGPU, with intermediate steps including Python pre-processing and JavaScript based engine.

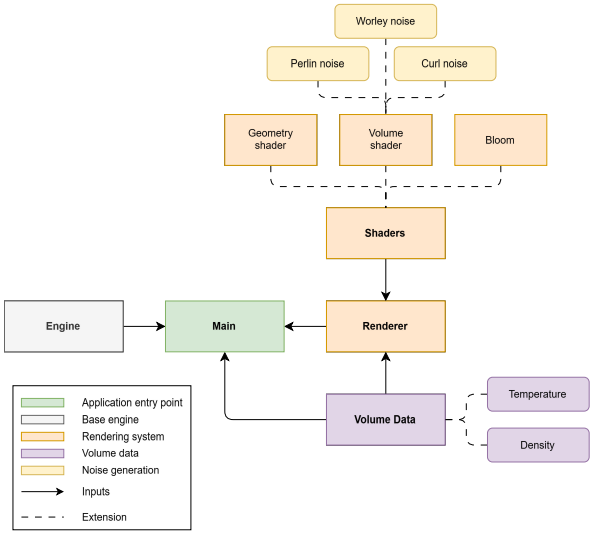


Figure 2: Overview of the system structure. Green represents the application entry point, light gray corresponds to the base engine, orange highlights components of the rendering system, purple indicates volume data, and yellow denotes noise generation components.

The overall structure of the system is shown in Figure 2, where the application begins by initializing the base engine and rendering system, followed by loading the volume data. The loaded data is then rendered in five distinct render passes. These steps are further explained in the following sections.

3.2 Data preparation

The 3D explosion effect begins with a low-resolution $32 \times 32 \times 32$ volumetric animation of an explosion created in Blender. Each voxel contains density and temperature data that define opacity and color, respectively, through a transfer function. This relatively small volume was chosen to ensure real-time performance and manageable memory usage, as increasing the resolution significantly raises

[7] Red Dead Redemption 2, Rockstar Games, 2018
[8] https://advances.realtimerendering.com/s2019/index.htm
[9] Counter-Strike 2, Valve, 2023
[10] https://github.com/UL-FRI-LGM/webgpu-examples

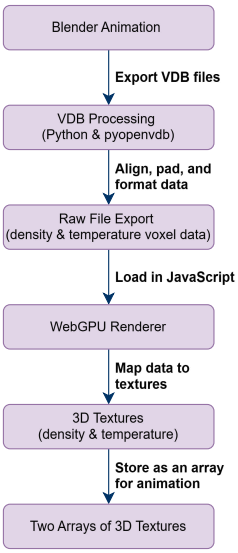


Figure 3: Data preparation workflow, from generating Blender animation data to processing it into two arrays of 3D textures for rendering.

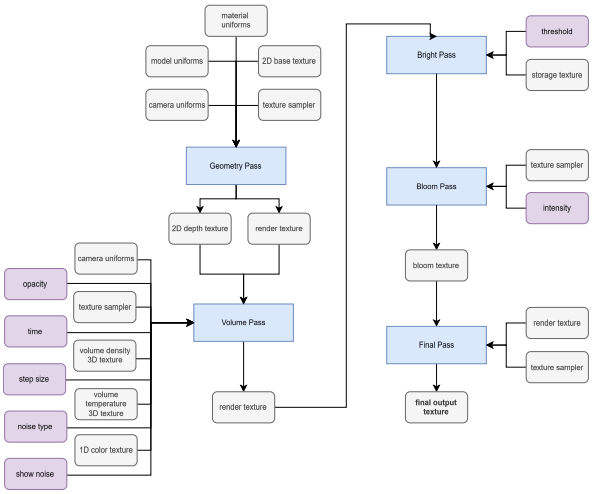


Figure 4: Render pass flow diagram. Render passes are shown in blue, fixed inputs/outputs are shown in gray, and user-configurable inputs are shown in purple.

the memory access cost due to the cubic growth of the voxel count.

The exported animation data is processed in Python using the pyopenvdb[11] library. While OpenVDB's infinite grid system allocates space dynamically for the expanding explosion, WebGPU requires fixed texture sizes. Therefore, all frames must be positioned correctly within a consistent simulation box to prevent misalignment during animation.

After alignment and padding, the data is exported as two separate raw files (density and temperature), each containing data for all frames of the $32 \times 32 \times 32$ volume. In JavaScript, these files are loaded to generate 3D textures for each frame. These textures are stored in two separate arrays for efficient access during animation, as frames can be quickly bound to the shader for sampling. The data preparation steps are shown in Figure 3.

3.3 Rendering Pipeline and Render Passes

The rendering pipeline transforms 3D scene data into the final 2D image through a sequence of GPU operations. Our WebGPU implementation uses shaders written in WGSL [4] to process geometry, determine pixel colors, and perform ray marching for volumetric rendering.

As illustrated in Figure 4, our implementation employs five distinct render passes: (1) geometry pass, (2) volume pass, (3) bright pass, (4) bloom pass, and (5) final pass. Each pass handles specific aspects of the volumetric explosion effect and generates textures that are used by subsequent passes.

The geometry pass establishes the foundation by rendering basic scene geometry. It produces color and depth textures that provide context for the volumetric effect. Standard vertex and fragment shaders handle the geometry transformation and basic texturing in this initial stage.

3.4 Animation

3D textures for each frame are stored in two separate arrays (density and temperature) for easy access during animation. To switch between the animation frames, the corresponding 3D textures and their respective samplers are bound to the shader with a dedicated bind group based on the current animation frame.

3.5 Ray marching

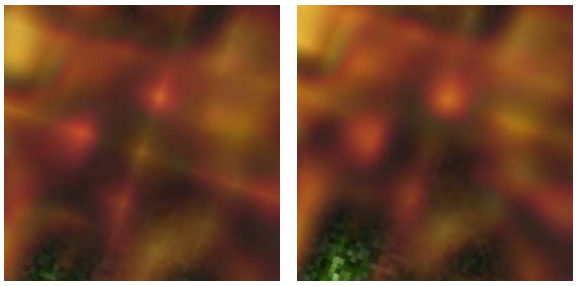
Ray marching visualizes volumes by casting rays from the camera and sampling at discrete steps along each ray's path. Our implementation traverses the voxel grid and samples transfer functions, which map raw values to color and opacity.

The fragment shader derives ray origin and direction from camera matrices and calculates ray-volume bounding box intersections to determine entry and exit points. Ray marching employs a configurable step count (16, 32, 64, 128, or 256), with 128-256 steps providing a good balance between performance and visual quality.

At each step, the ray position is updated as:

$$p(t) = o + td,$$

[11] https://www.openvdb.org/documentation/doxygen/



(a) linear sampling (b) quasi-cubic sampling

Figure 5: Sampling artifacts comparison.

Figure 6: Explosion color palette.

where $p(t)$ is the current position, o is the ray origin, and d is the normalized direction. Step size Δt is calculated as:

$$\Delta t = \frac{t_{max} - t_{min}}{N},$$

with t_{min} and t_{max} representing entry and exit points, and N being the step count.

We sample density and temperature textures using quasi-cubic sampling[12] to enhance interpolation quality, as shown in Listing 1.

Listing 1: Pseudocode for quasi-cubic sampling.

```
// u is the input texture coordinate
vec4f quasiCubicSampling(texture_3d volume,
      sampler s, vec3f u)
{
  R = textureDimensions(volume);
  U = u * R + 0.5;
  F = fract(U);
  U = floor(U) + F * F * (3.0 - 2.0 * F);
  C = (U - vec3f(0.5)) / R;
  return textureSample(volume, s, C);
}
```

This approach reduces artifacts by pre-distorting the texture sampling coordinates using a smoothstep function for smoother interpolation[13]. Figure 5 demonstrates the improvement over linear sampling.

Temperature values are mapped to explosion colors using a transfer function that samples from a color palette (Figure 6) encoded as a 1D texture.

Alpha blending [16] combines voxel contributions along the ray path:

$$\alpha_{acc} := 1 - (1 - \alpha_{acc}) \cdot (1 - \alpha_{current}),$$
$$C_{acc} := C_{acc} + (1 - \alpha_{acc}) \cdot C_{current} \cdot \alpha_{current},$$

where α_{acc} and C_{acc} are accumulated opacity and color. $\alpha_{current}$ and $C_{current}$ are the opacity and color of the current voxel. The ray march continues until the ray exits the volume or reaches an opacity threshold.

3.6 Depth testing

For proper occlusion between volumetric effects and solid objects, we implement manual depth testing within the ray marching loop. This approach uses depth values from the geometry pass to determine visibility.

To compare volume samples against the depth buffer, we transform each sample's world position to clip space. We compare this depth against the stored depth buffer value with a small bias ϵ to avoid precision issues:

$$d_w := \begin{cases} 1, & \text{if } z \leq (z_b + \epsilon) \\ 0, & \text{otherwise} \end{cases},$$

where d_w is the depth weight, z is the screen depth and z_b is the stored depth buffer value.

For visible samples, we apply the depth weight to the density $w := \rho \cdot d_w$, where w is the weighted density and ρ is the density value of the volume sample.

The accumulated color and opacity are updated as:

$$\alpha_{acc} := \alpha_{acc} + (1 - \alpha_{acc}) \cdot \frac{w \cdot o}{N},$$
$$C_{acc} := C_{acc} + (1 - \alpha_{acc}) \cdot \frac{C_{current} \cdot w \cdot o}{N},$$

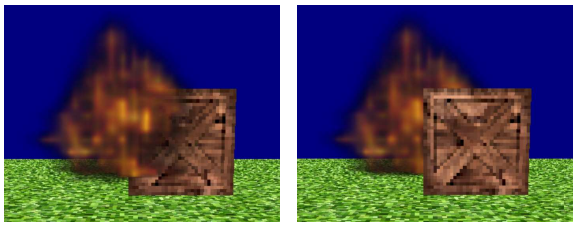
where o is a user-adjustable opacity factor. This accumulation formula adapts classical alpha blending following the approach described by Max [13].

If the volume sample's depth is greater than the stored depth, it lies behind an opaque object and is discarded. Otherwise, it contributes to the final color accumulation, and it blends with the scene. Figure 7 demonstrates the effect of depth testing, while Figure 8 shows the importance of adding bias for precision.

3.7 Noise

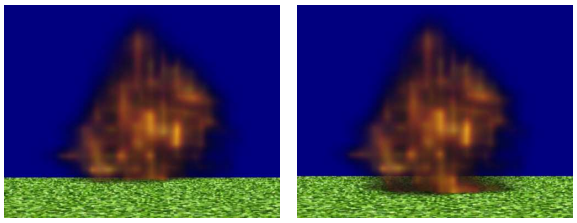
To enhance realism in our volumetric explosion, we implemented three types of noise: Perlin noise, Worley noise, and a combination of curl noise with Worley noise. Each implementation can be individually selected to evaluate its distinct contribution. This was done by linearly blending the original sample value with a noise-modified version, using a weight that controls how strongly the noise affects the result.

[12] https://iquilezles.org/articles/texture/
[13] https://iquilezles.org/articles/smoothsteps/



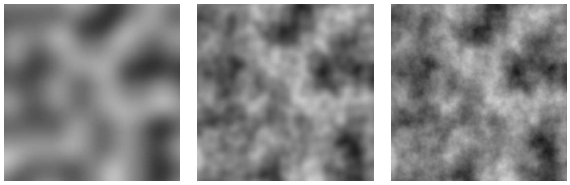
(a) without depth testing (b) with depth testing

Figure 7: The effect of depth testing results in incorrect rendering in (a) and the correct version in (b).



(a) without bias ϵ (b) with bias $\epsilon = 0.004$

Figure 8: Depth testing without bias (a) results in incorrect rendering. Adding a small bias accounts for numerical errors (b).



(a) 1 octave (b) 3 octaves (c) 6 octaves

Figure 9: Perlin noise with varying octave counts.

3.7.1 Perlin noise

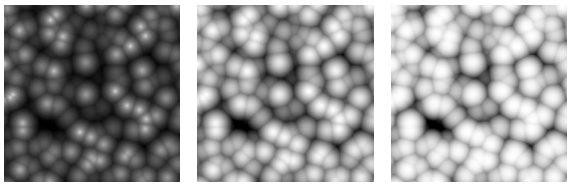
Our 3D Perlin noise implementation uses layered octaves for added detail. The function takes parameters p (3D coordinate), f (base frequency), o (octave count), λ (persistence), l (lacunarity), and s (seed). The noise is computed as:

$$P(p) = \sum_{i=0}^{o} \text{perlinNoise}(p \cdot f_i, s_i) \cdot a_i,$$

where s_i is a derived seed, $a_i = \lambda^i$, and $f_i = f \cdot l^i$. The base function maps position p to a grid cell, assigns gradient vectors to corners, and computes dot products between gradients and relative positions. A smoothing function $F(t) = 6t^5 - 15t^4 + 10t^3$ ensures smooth transitions between grid values. Figure 9 demonstrates the effect of varying octave counts.

3.7.2 Worley noise

Worley noise generates cellular patterns by partitioning space into a grid with pseudo-random feature points. The feature point placement uses a hashing function:



(a) 0.5 power (b) 1.5 power (c) 2.5 power

Figure 10: Worley noise with different power factor.

$$\begin{aligned} p_3 &= \text{fract}(p \cdot (0.1031, 0.1030, 0.0973)), \\ p_3 &= p_3 + \text{dot}(p_3, p_3.yxz + 33.33), \\ \text{hash}(p) &= \text{fract}((p_3.xxy + p_3.yxx) \cdot p_3.zyx). \end{aligned}$$

The noise value is determined by the minimum distance to the nearest feature point, adjusted by power factor α:

$$D' = \min_i ||p - f_i||^\alpha,$$

where $\alpha < 1$ creates smoother transitions and $\alpha > 1$ produces sharper edges, as shown in Figure 10. The final noise is inverted to make high-density areas correspond to feature points.

3.7.3 Worley and curl noise

Curl noise introduces turbulent motion when combined with Worley noise, creating a balance between structured smoke and chaotic movement. Derived from Perlin noise gradients, curl noise is calculated using the curl of the vector field:

$$\nabla \times G = \left(\frac{\partial G_z}{\partial_y} - \frac{\partial G_y}{\partial_z}, \frac{\partial G_x}{\partial_z} - \frac{\partial G_z}{\partial_x}, \frac{\partial G_y}{\partial_x} - \frac{\partial G_x}{\partial_y} \right).$$

We approximate derivatives using finite differences with step size δ:

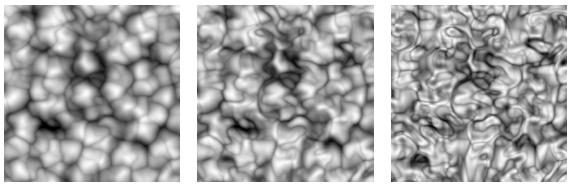
$$\begin{aligned} c_x &= G(y + \delta, z + \delta) - G(y - \delta, z - \delta), \\ c_y &= G(x + \delta, z + \delta) - G(x - \delta, z - \delta), \\ c_z &= G(x + \delta, y + \delta) - G(x - \delta, y - \delta). \end{aligned}$$

The resulting vector $\vec{c}$ represents local swirling motion. Figure 11 shows the effect of varying δ values.

3.8 Post-Processing: Bloom

Bloom enhances the perception of bright regions by simulating light scattering. Our implementation follows a multi-stage process based on techniques described[14].

[14] https://developer.nvidia.com/gpugems/gpugems/part-iv-image-processing/chapter-21-real-time-glow



(a) 0.005 δ step (b) 0.01 δ step (c) 0.02 δ step

Figure 11: Worley noise plus curl vector with varying δ step size.

3.8.1 Bright pass

The bright pass isolates high-intensity pixels using a luminance function derived from the CIE XYZ color space:

$$L = 0.2126R + 0.7152G + 0.0722B,$$

where R, G, and B are color channels weighted by human visual sensitivity. Pixels exceeding a threshold T are retained:

$$C_{bright} = \begin{cases} C, & \text{if } L > T \\ 0, & \text{otherwise} \end{cases},$$

where C is the original pixel color.

3.8.2 Downsampling

We downsample the bright regions across multiple mip levels, with each level at half the resolution of the previous. The number of mip levels is determined by:

$$M = \lceil \log_2(max(W,H)) \rceil.$$

Each mip level i is generated by bilinear filtering from the previous level:

$$C_i(x,y) = \frac{1}{4}\sum_{j=1}^{4} C_{i-1}(x+\Delta x_j, y+\Delta y_j),$$

where Δx_j, Δy_j are offsets of neighboring texels.

3.8.3 Upsampling

We reconstruct the bloom effect by progressively upsampling and blending from the smallest mip level:

$$C_i(x,y) = C_i(x,y) + \alpha \cdot U(C_{i+1}, x, y),$$

where α is the bloom intensity factor. Figure 12 illustrates this process.

The final pass combines the main rendered image with the processed bloom texture to produce the final output.

Figure 12: Stages of the bloom effect. From left to right: original image, bright pass image, and raw bloom effect (blurred highlights before combining with the original image).

3.9 Resolution Scaling

To reduce the rendering time of the volumetric effect, we use resolution scaling, where the scene is rendered at a smaller resolution and then scaled up. There are three resolution options: full, halved, and quartered. The full resolution renders at the native canvas resolution, offering the best visual fidelity at the cost of performance. On the other hand, the quartered option reduces both width and height to $\frac{1}{4}$ of their original size, meaning that it renders at only $\frac{1}{16}$ of the original pixel count. When the resolution is quartered, it has the highest performance boost, though at the expense of some visual plausibility.

4 RESULTS

This section presents the results of the benchmark and visual quality assessment for the real-time explosion effect. The core rendering settings included volume density (20.0), bloom intensity (0.8), and bloom threshold (1.0). The noise was generated using curl noise (delta 0.01), Worley noise (power 1.5) when combined with curl noise scaled by a factor of 5, and Perlin noise (frequency 4, octave count 2, persistence 0.5, lacunarity 2). 30 % of the noise influenced the color distribution, while 50 % influenced the density distribution.

4.1 Benchmark

Performance was evaluated on three hardware configurations with varying ray marching steps (Nsteps), resolutions (Q: quartered – 480×270 and upscaled to full screen, H: halved = 960×540 and upscaled, F: full - 1920×1080), and noise types (P: Perlin, W: Worley, WC: Worley+curl). Real-time performance results ($\leq$16.67 ms) are bolded.

Laptop (ASUS TUF Gaming FX505DT) specifications:

- **CPU:** AMD Ryzen 3550H
- **GPU:** NVIDIA GeForce GTX 1650
- **Memory:** 16 GB RAM

Results are shown in Table 1:

Table 1: Laptop render times

Nsteps	Resolution	Noise	Time (ms)
16	Q	P	**7.21**
256	F	WC	5412.35
32	Q	P	**9.37**
32	H	P	33.64
64	Q	WC	83.85
64	H	WC	320.51
128	Q	P	38.52
128	H	P	142.50
128	Q	WC	180.47
128	H	WC	675.69

Desktop PC specifications:

- **CPU:** Intel Core i5-9400F
- **GPU:** Nvidia GTX 1050 Ti
- **Memory:** 16 GB RAM

Results are shown in Table 2:

Table 2: Desktop PC render times

Nsteps	Resolution	Noise	Time (ms)
16	Q	P	**3.54**
256	F	WC	1867.12
32	Q	P	**5.11**
32	H	P	**11.19**
64	Q	WC	34.54
64	H	WC	127.27
128	Q	P	**12.78**
128	H	P	48.23
128	Q	WC	68.88
128	H	WC	252.12

Workstation specifications:

- **CPUs:** Dual Intel(R) Xeon(R) Gold 6140
- **GPUs:** three Nvidia RTX A4000 (Ada generation) 20 GB RAM (one GPU used)
- **Memory:** 256 GB RAM

Results are shown in Table 3:

Table 3: Workstation render times

Nsteps	Resolution	Noise	Time (ms)
16	Q	P	**1.77**
256	F	WC	159.30
32	Q	P	**3.41**
32	H	P	**8.32**
64	Q	WC	**10.56**
64	H	WC	**10.68**
128	Q	P	**9.24**
128	H	P	**12.78**
128	Q	WC	**9.50**
128	H	WC	21.17

4.2 Visual Quality Assessment

A survey compared the real-time implementation against a Blender-rendered effect using four animation frames (10, 30, 50, 70). The real-time effect used 128 steps, quartered resolution, and Worley+curl noise. For each frame, participants answered multiple comparison and evaluation questions structured as follows:

- Perceived similarity between the real-time and Blender explosions (5-point Likert scale [11], from not similar at all to almost identical)
- Realism rating of the real-time explosion (5-point Likert scale, from very unrealistic to very realistic)
- Realism rating of the Blender explosion (5-point Likert scale, from very unrealistic to very realistic)
- Perceived realism preference, where participants selected which version appeared more realistic (real-time or Blender)

At the end of the survey, participants were given the render times of one frame for both effects. They were asked to pick the effect they think is more suitable for use in games, considering their render times.

The survey was anonymous and distributed online. In total, 48 responses were collected from participants with varying levels of experience in computer graphics, based on the authors' knowledge of the distribution channels. The goal was to evaluate subjective visual quality by comparing our real-time implementation against the offline-rendered simulation, using the same input data to ensure a fair comparison. The images used in the survey are available in the supplemental material folder of the code repository. The left image represents the real-time render, while the right image shows the offline Blender render.

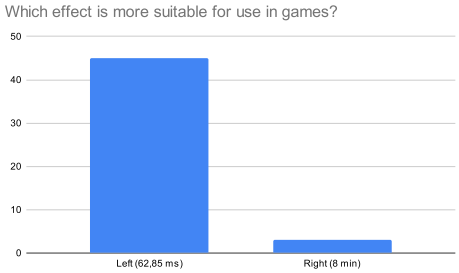


Figure 13: Survey results on the perceived suitability for games: "Left (62.85 ms)" represents the real-time effect, while "Right (8 min)" represents the Blender rendered effect, with their respective render times.

Key Findings:

- Similarity: The real-time explosion was rated "slightly similar" to the Blender version, with Frame 30 being the most similar (39.6 % chose "very similar").
- Individual Realism: Real-time explosion was commonly rated "somewhat unrealistic" (35.4 % across all frames).
- Comparative Realism: Real-time explosion was preferred in Frames 10 (83.3 %) and 70 (70.8 %), while Blender was preferred in Frame 30 (62.5 %).
- Suitability for Games: 93.8% selected the real-time explosion (62.85 ms render time) over the Blender version (8 minutes) for game applications, as shown in Figure 13.

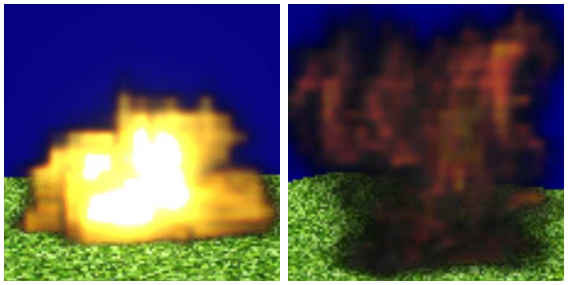
4.3 Rendered Images

The images in Figure 14 demonstrate the final results of the rendered volumetric explosion with 128 steps.

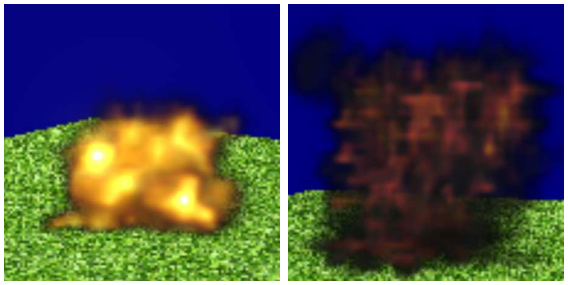
5 DISCUSSION

The explosion effect scales well across hardware, but performance degrades significantly at higher step counts, increased resolution, and more complex noise functions. It is important to use a performant GPU, as the lowest-performing device in the benchmark is a gaming laptop.

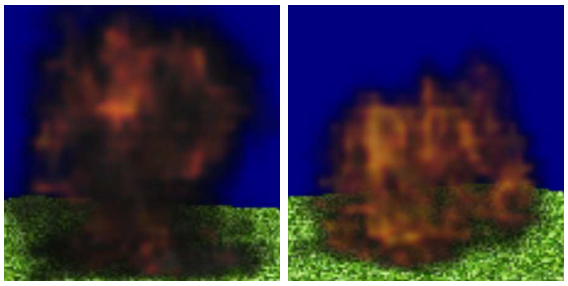
In the benchmark, we used the halved and full resolutions for comparison; further on, the quartered resolution is considered default and implied. To achieve real-time performance on lower-end hardware, the maximum number of steps is 32, with Perlin or Worley noise. On a mid-range desktop PC, 128 steps are achievable in real time, but without Worley+curl. On a high-performance workstation optimized for graphics workloads, real-time is



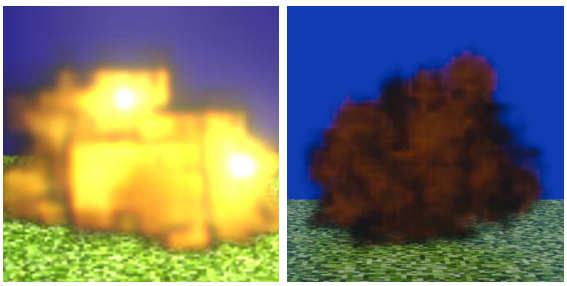
(a) Perlin noise



(b) Worley noise



(c) Worley+curl noise



(d) Worley noise with intense bloom

(e) Blender rendered explosion

Figure 14: Rendered images with different noise types and comparison with Blender-rendered output image.

maintained across almost all settings. The only exception is in the more extreme cases, such as at 128 steps, Worley+curl noise, and halved resolution.

Overall, achieving a visually appealing explosion effect at around 128 steps is feasible on mid-range commercial hardware by using Perlin and Worley noise to stay within real-time performance. Higher-end setups can afford more complex noise functions and higher resolutions.

The survey results highlight a trade-off between realism and performance. While the real-time explosion effect is generally well received and deemed suitable for games, it falls short in terms of achieving complete visual plausibility, yet is perceived as more realistic compared to the pre-rendered simulation.

6 CONCLUSION

This paper presented the implementation of a real-time volumetric explosion effect using WebGPU and JavaScript, focusing on efficient rendering of volumetric data while maintaining real-time performance. Explosion data was generated in Blender, processed in Python for alignment and padding, and imported as 3D textures. Ray marching was used for volume traversal, evaluating density and temperature to compute color and opacity, with quasi-cubic sampling ensuring smooth interpolation.

Noise functions like Perlin, Worley, and curl noise introduced natural variation and turbulence, while a bloom post-processing effect emphasized high-intensity regions. Performance improvements included adjustable step counts, resolution scaling, and noise configuration, achieving real-time frame rates on mid-range hardware. User surveys indicated the real-time effect, though less realistic than the Blender reference, was preferred for its speed and suitability for games.

6.1 Future Work

Improvements could include integrating physically based lighting models, 4D noise functions, and dynamic transfer functions that evolve over time. Adaptive ray marching and early ray termination could further optimize performance. Real-time volumetric effects have the potential to greatly enhance immersion through more convincing environmental interactions. As hardware continues to evolve, we expect volumetric effects to become an essential component in modern gaming. Beyond games, these techniques have applications in scientific visualization, simulation, and interactive media, paving the way for future advancements in real-time rendering.

7 REFERENCES

[1] Ciril Bohak et al. "RenderCore–a new WebGPU-based rendering engine for ROOT-EVE". In: *EPJ Web of Conferences*. Vol. 295. EDP Sciences. 2024, p. 03035.

[2] Robert Bridson, Jim Houriham, and Marcus Nordenstam. "Curl-noise for procedural fluid flow". In: *ACM Transactions on Graphics (ToG)* 26.3 (2007), 46–es.

[3] World Wide Web Consortium. *WebGPU*. 2025. URL: https://www.w3.org/TR/webgpu/.

[4] World Wide Web Consortium. *WebGPU Shading Language*. 2025. URL: https://www.w3.org/TR/WGSL/.

[5] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. "Visual simulation of smoke". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, pp. 15–22.

[6] Stefan Guthe et al. "Interactive rendering of large volume data sets". In: *IEEE Visualization, 2002. VIS 2002.* IEEE. 2002, pp. 53–60.

[7] James T Kajiya and Brian P Von Herzen. "Ray tracing volume densities". In: *ACM SIGGRAPH computer graphics* 18.3 (1984), pp. 165–174.

[8] Jens H Krüger and Rüdiger Westermann. "GPU simulation and rendering of volumetric effects for computer games and virtual environments". In: *Computer Graphics Forum*. Vol. 24. 3. Amsterdam: North Holland, 1982-. 2005, pp. 685–694.

[9] Žiga Lesar, Ciril Bohak, and Matija Marolt. "Real-time interactive platform-agnostic volumetric path tracing in WebGL 2.0". In: *Web3D 2018 : proceedings*. 2018, pp. 1–7.

[10] M. Levoy. "Display of surfaces from volume data". In: *IEEE Computer Graphics and Applications* 8.3 (1988), pp. 29–37. DOI: 10.1109/38.511.

[11] Rensis Likert. "A technique for the measurement of attitudes." In: *Archives of psychology* (1932).

[12] William E Lorensen and Harvey E Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *Seminal graphics: pioneering efforts that shaped the field*. 1998, pp. 347–353.

[13] N. Max. "Optical models for direct volume rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 1.2 (1995), pp. 99–108. DOI: 10.1109/2945.468400.

[14] Asger Meldgaard, Sune Darkner, and Kenny Erleben. "Fast vortex particle method for fluid-character interaction". In: *Graphics Interface 2022*. 2022.

[15] Ken Perlin. "An image synthesizer". In: *ACM Siggraph Computer Graphics* 19.3 (1985), pp. 287–296.

[16] Thomas Porter and Tom Duff. "Compositing digital images". In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. 1984, pp. 253–259.

[17] Steven Worley. "A cellular texture basis function". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, pp. 291–294.