Volume 0 (1981), Number 0 pp. 1–4 COMPUTER GRAPHICS *forum*

Volume-based special effects in video games

T. Mlakar[1]

[1]Fakulteta za računalništvo in informatiko, Univerza v Ljubljani

Abstract

This research paper explores an implementation of volume-based smoke simulation for video games. The introduction touches upon the advantages of volume-based rendering of special effects over using traditional methods. The second section touches upon smoke propagation types and properties as a research for better understanding of real life smoke propagation. Implementation section describes the tools and techniques used for achieving a voxel grid representation of objects and smoke volume, as well as ray marching algorithm and different aspects of light interaction with the participating medium. The paper finishes off with evaluation of the implementation and some final, concluding thoughts.

CCS Concepts

•***Computing methodologies*** → ***Volumetric models; Physical simulation;***

1. Introduction

With each passing year, the gaming industry is rapidly expanding and developers must always be on the hunt for improved and fresh ways to deliver immersive experiences. A big part of that are visual effects. Creating realistic visual effects such as smoke, fire, explosions, clouds and fog in video games involves a combination of techniques and technologies. Most common used technique for rendering these types of visuals is based on sprites, 2D images, which are often managed through particle systems [smo20]. While effective and efficient for rendering smoke and other special effects in video games, sprites have certain limitations that can affect their perceived depth and overall realism.

In recent years, the appearance of volume-based effects in video games marked a significant advancement in realism and complexity of visual effects in games. We've witnessed this with realistic cloud simulation in games such as Horizon Zero Dawn [hzd17] and No Man's sky [nms18] and realistic smoke grenade propagation in CS2 [cs223]. Volume-based rendering is a step beyond traditional 2D sprite-based methods, since the particles can dynamically interact with light sources, allowing for realistic lighting and shadowing effects, whereas sprites can have simple transparency and basic shading, but they cannot accurately represent how light diffuses and scatters within a volume.

Rendering any kind of natural phenomena (fog, smoke, fire, ..) in a realistic manner requires a deep understanding of its physical properties and behaviours.

2. Smoke and types of propagation

Smoke grenades are pyrotechnic devices designed to emit a dense cloud of smoke when activated. They can be used in number of ways. In military operations, they are most commonly used for signaling, camouflage and obscuring visibility, but also to mark targets or areas. They are also no strangers to various sporting events and celebrations, since they add a touch of excitement. The smoke can take on various shapes, such as clouds or cones, depending on the design and the conditions in which the grenade is used. The initial burst of smoke is often very rapid, followed by a slower, more sustained release as the chemical reaction continues. The dissipation of the produced smoke typically starts around a few minutes after the initial ignition. Complete dissipation of the particles and clearing of the area can, however, take significantly longer. The particles settle through several natural forces such as gravity or winds and naturally spread out to fill the available space, disperse over a wider area, thin out and decrease in concentration over time. They might also react with other substances in the environment, which breaks them down into less visible forms [wik20, his23].

3. Implementation

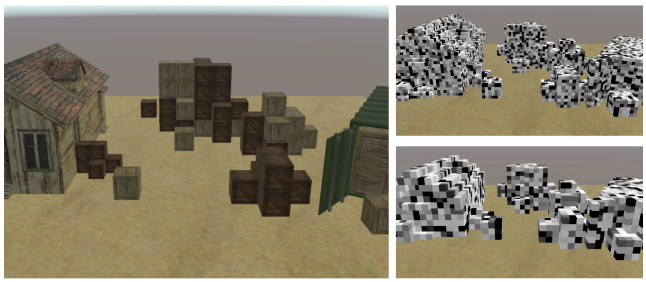
3.1. Voxelization

In our pursuit of rendering volumetric special effects, grenade smoke to be exact, we delve into the realm of voxels.

After creating a simple scene within our Unity project, our first step involved partitioning the scene's dimensions into a three-dimensional regular grid of voxels. To efficiently manage this spatial data, we opted for a 1D compute buffer of which indices range

from zero to total number of voxels in the grid. This allows for number of ways of manipulation through a compute shader, one of which was uniformly filling the buffer with a value of 1. This signified voxel occupancy, a crucial aspect we later utilised in rendering.

We typically find a lot of small and bigger sized objects within game and real world environments. Each usually has some way of interaction with a special effect / real life phenomena. Hence we embarked on the task of voxelizing the meshes of static scene objects. Computation of the scene voxels can, given the static nature of the environment throughout gameplay, occur only once, in the build stage of the environment processing. Therefore we initialised a separate 1D compute buffer of the voxel grid for storing the computed static voxels. We first sent it off to compute shader to uniformly fill the buffer with a value of 0, which in translation states there is no voxels occupied by static objects in the voxel grid. From there we utilised the Separating Axis Theorem (SAT) method to calculate a possible overlapping between each triangle of the static object mesh and each voxel of the voxel grid. SAT algorithm checks 13 potential separating axes: by projecting both shapes onto these axes, it determines if a separation exists. If not, they intersect. [aab, gpu21]. If they intersect, we fill the voxel with index of the current position to 1, which indicates a presence of a static object.



Voxelization of static scene objects using two different voxel sizes. Smaller size gives more accurate representation of static scene objects.

Figure 1: *Voxelization of the scene.*

With our static voxels established and filled, it was time for smoke propagation. Smoke grenades typically generate a cloud-like or cone-like (plume) shape. The dispersion is rapid initially, creating a dense effect, but it gradually dissipates over time. We achieved that by implementing an easing function. Easing functions are typically used in non-linear animations to improve the quality of transitions, making them smoother compared to basic linear transformations.

In order for there to be interaction between smoke voxels and voxels that represent static scene, we need to take into account how smoke propagates through and around obstacles in real life. If the obstacle is small enough, the smoke will completely absorb its shape. On the other hand, if the obstacle is large enough to act as a barrier, the smoke will not propagate fully through. To replicate this we used the logic behind an algorithm for filling in connected areas of a multidimensional array - flood fill algorithm [Kar23], which assigns current voxel the highest value of all 6 neighbouring voxels in the three dimensional grid. This way, in terms of cloud-like smoke shape, the highest values will be closest to the origin of the smoke and smallest at the smoke edges. If any of the neighbouring voxels turn out to be static, the algorithm will assign it a small value. By assigning a small value to voxels neighbouring static objects, we effectively mark these voxels as part of the obstacle, which prevents smoke from propagating through and all the way around it [Ace23].

Figure 2: *Smoke propagation using voxels.*

3.2. Ray marching for volume rendering

Once we had a representation of smoke volume in voxels, it was time to utilize ray marching to render our voxel grid. In ray marching, we don't check for intersection with geometry like in traditional ray tracing methods. Instead we are stepping along each ray projected from camera through the pixels onto the scene to accumulate the properties of the materials encountered along the way. We could imagine it as adding a layer of semi-transparent paint to a canvas, which adds to the overall color and opacity.

To determine how much the particles in the voxel volume obscure the ability too see what's on the other side, we need to calculate light absorption while marching through the volume. At each iteration of ray marching algorithm we took a step forward and sampled density of the voxel volume at that point by converting world positions to voxel grid coordinates to check current value of voxel grid. To sample the current voxel value we used pre-existing values we obtained during the flood fill algorithm of smoke propagation. This approach mimics the denser concentration of smoke at its origin, with density gradually decreasing towards the edges of the volume. We used the accumulated density to calculate transmittance, a measure of how transparent the medium is, with Beer's law to produce and alpha mask that we later on blended into the final render. When traveling along the ray we casted, we continuously had to verify that we haven't surpassed the scene's depth. If we had, calculating the alpha mask would become redundant as it wouldn't even contribute to the final render, so we skipped any further computation, as seen in Figure 3 [Ace23, vol].

After calculating an alpha mask, we had a presentation of how much light the volume is absorbing to obscure the objects lying on the other side.

As the ray passes through a medium, it may collide with particles and be scattered in different directions. This has two effects on the total light energy that the ray carries. We call them out-scattering and in-scattering. To get the shadows and details within the smoke

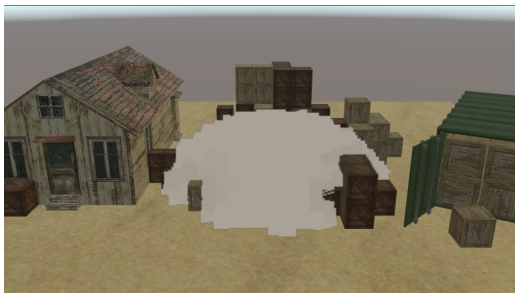


Figure 3: *Calculated alpha mask of smoke transmittance.*

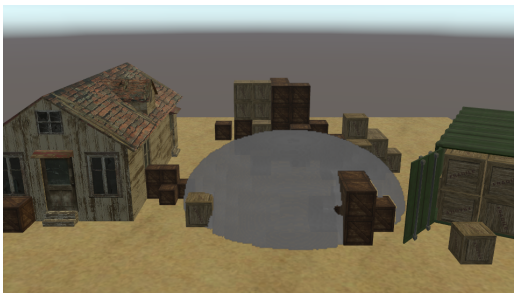


Figure 5: *Smoothing function to get rid of visible voxel edges.*

volume, we needed to calculate how much of the original light energy was lost once it reached the point that was being sampled. We did that by performing the logic behind the ray marching algorithm, but this time from the sample point to our light source. We used Beer's law to attenuate the light contribution due to in-scattering. We multiplied this energy loss by the color of the light source and our base transmittance and then added that to the color of the currently rendered pixel. While in-scattering redirects some of the light that was not initially meant for the observer towards him, out-scattering does the opposite. When a ray of light hits the particles in the participating media, some of the light that was originally meant to scatter in the direction of the observer also scatters elsewhere. We simulated this by using a Rayleigh phase function which models the behaviour of a particle scattering the light uniformly forwards and backwards. If we were simulating some different kind of participating media we could also use some of the other phase function models, which scatter the rays in some other manner, may it be an isotropic or an anisotropic scattering. This implementation can be seen in Figure 4 [Ace23, vol].

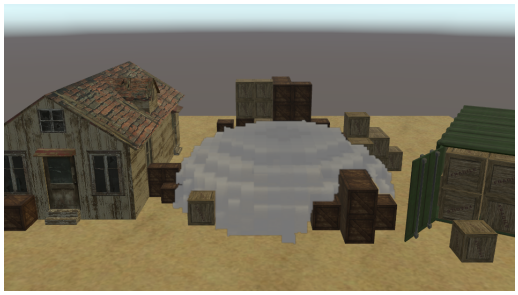


Figure 4: *Calculated in- and out-scattering of particles in particpating media.*

The overall shape of our smoke volume is determined by the underlying grid of smoke voxels. But because we ultimately want a smooth shape with some added texture variance, where voxel edges aren't as clearly visible as on Figure 4, we used a "smoothstep" function. The function ensured a smooth, gradual transition based on the distance from the smoke origin point. By combining this with the voxel value, we used the final result as our updated method of obtaining the density for the ray marching process. The result of this can be seen in Figure 5.

Participating media simulation usually uses one or more types of generated noise for creating detailed textures. So for the smoke to actually look like smoke we needed a noise texture for some extra added detail. We opted for the generation of Worley noise, since it's most frequently used for wide variety of textures like stones, water and clouds. To help us with visualisation of our generating process, we first implemented the generation of 2D noise which we rendered to a .png image, but the process both for 2D and 3D generally works the same.

We divided our texture resolution to a grid of uniformly sized cells, in which we randomly generated one feature point. For every pixel in the texture we calculated the distance to the feature point (or the closest distance between multiple of the feature points if we would generate multiple inside a single cell) and set the pixel to the value between 0 and 1, which corresponds to pixel being really far or really close to the feature point, respectfully. Just to be sure we calculated the correct shortest distance, we also checked the distances from pixel to feature points in the neighbouring cells. Once we inverted the result of the algorithm, we got a texture looking somewhat like this:

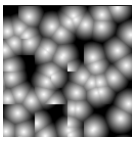


Figure 6: *Generated 128x128 Worley noise texture.*

We sadly ran out of time to finish the task of sampling the generated noise texture and its use for improving the final density distribution as well as adding some variance to our voxel volume.

4. Evaluation

It is important to note that our implementation is not perfect and probably wouldn't be even if we succeeded with adding the generated noise texture. It could use a lot of improvements regarding the overall interaction with the light source and some other aspects of the rendering. Using a specific combination of absorption and scattering coefficients, the overall density, specific phase function for light scattering as well as implementing other subtle changes can and will produce many kinds of participating media. To get a

better representation, we could, for one, use a different sampling of the density of the volume, like using trilinear interpolation. The rendered textures of alpha mask and overall pixel color were, to save on computational resources, sampled at quarter resolution and were later upscaled to the original resolution.

Due to some unfortunate circumstances with faulty computer equipment (the assignment sadly had to be done on a low-performance GPU) we had to compromise on some of the functionalities to save with computational resources so our volume rendering was still running at just around 60fps.

5. Conclusions

By leveraging the power of modern GPU's, volume-based effects significantly enhance the visual realism of video games with the help of techniques such as voxelization, ray marching and smart use of resources. While volume-based special effects have been the subject of considerable research and development over the years, there is still room for improvement and innovation in these areas.

References

[aab] Aabb - triangle · 3dcollisions. URL: https://gdbooks.gitbooks.io/3dcollisions/content/Chapter4/aabb-triangle.html. 2

[Ace23] ACEROLA: I tried recreating counter strike 2's smoke grenades, 05 2023. URL: https://www.youtube.com/watch?v=ryB8hT5TMSg. 2, 3

[cs223] Counter-strike 2: Official responsive smokes trailer, 03 2023. URL: https://www.youtube.com/watch?v=kDDnvAr6gGI. 1

[gpu21] Gpu mesh voxelizer part 2: Triangle / voxel intersection – bronson zgeb, 05 2021. URL: https://bronsonzgeb.com/index.php/2021/05/29/gpu-mesh-voxelizer-part-2/. 2

[his23] History of the smoke grenade and use cases for today, 04 2023. URL: https://shutterbombs.com/blogs/smoke-bombs-and-smoke-grenades/history-of-the-smoke-grenade. 1

[hzd17] Volumetric cloudscapes of horizon zero dawn, 02 2017. URL: https://www.youtube.com/watch?v=FhMni-atg6M. 1

[Kar23] KARA K.: Understanding and implementing flood fill algorithm, 06 2023. URL: https://medium.com/@koray.kara98.kk/understanding-and-implementing-flood-fill-algorithm-60ab81538d54. 2

[nms18] No man's sky next volumetric clouds, 07 2018. URL: https://www.youtube.com/watch?v=SAftOD8tUww. 1

[smo20] Smoke grenade - comparison in 50 different games, 03 2020. URL: https://www.youtube.com/watch?v=xY1fIXoP9zY&t=288s. 1

[vol] Volume rendering for developers: Foundations. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/volume-rendering-for-developers/intro-volume-rendering.html. 2, 3

[wik20] Smoke grenade, 06 2020. URL: https://en.wikipedia.org/wiki/Smoke_grenade. 1