# Spherical Billboards for Rendering Volumetric Data

**Umenhoffer Tamás, László Szirmay-Kalos, and Gábor Szíjártó**
**Budapest University of Technology, Hungary**

# 1. Introduction

Many types of natural phenomena belong to participating media, including fire, smoke, explosions, fog, cloud, etc., which can be simulated by animating and rendering a population of individual particles forming a particle system. A particle is an animation and rendering primitive, whose geometric extent is small, but which possess certain properties such as position, speed, color, and opacity. Formally, a particle system is a discretization of a continuous medium, where a particle represents a spherical, locally homogeneous neighborhood. This allows us to replace differentials of the equations governing the motion and light scattering by finite differences during simulation and rendering. In this article we focus on real-time rendering such systems, which means the solution of the volumetric rendering equation.

Let us consider a ray going through a particle sphere and examine how the radiance is modified by the particle. Denoting the length of the ray segment intersecting the sphere of particle $j$ by $\Delta s_j$, and the *density*, *albedo*, and *phase function* of this particle by $t_j, a_j, P_j$, respectively, the discretization of the volumetric rendering equation leads to the following equation expressing *outgoing radiance* $L(j,\vec{w})$ of particle $j$ at direction $\vec{w}$:

$$L(j,\vec{w}) = I(j,\vec{w}) \cdot (1-a_j) + a_j \cdot a_j \cdot C_j + a_j \cdot (1-a_j) \cdot L_j^e(\vec{w}), \tag{1}$$

where $I(j,\vec{w})$ is the *incoming radiance* from direction $\vec{w}$,

$$a_j = 1 - e^{-\int_{\Delta s_j} t(s)\,ds} = 1 - e^{-t_j \Delta s_j} \tag{2}$$

is the *opacity* that expresses the decrease of radiance caused by this particle due to *extinction*, $L_j^e$ is the emission radiance, and

$$C_j = \int_{\Omega'} I(j,\vec{w}') \cdot P_j(\vec{w}',\vec{w})\, d\vec{w}'$$

is the total contribution from *in-scattering*.

Having approximated the in-scattering term somehow, the volume can efficiently be rendered from the camera using splatting and alpha blending. Splatting substitutes particles with semi-transparent, camera-aligned rectangles, called *billboards*. When a billboard is rasterized, the in-scattering term is attenuated according to the total opacity of the particles that are between the camera and this particle. This requires particles to be sorted in the view direction before sending them to the frame buffer in back to front order. At a given particle, the evolving image is decreased according to the opacity of the particle and increased by its in-scattering and emission terms (equation 1).

## *1.1 Billboard clipping and popping artifacts*

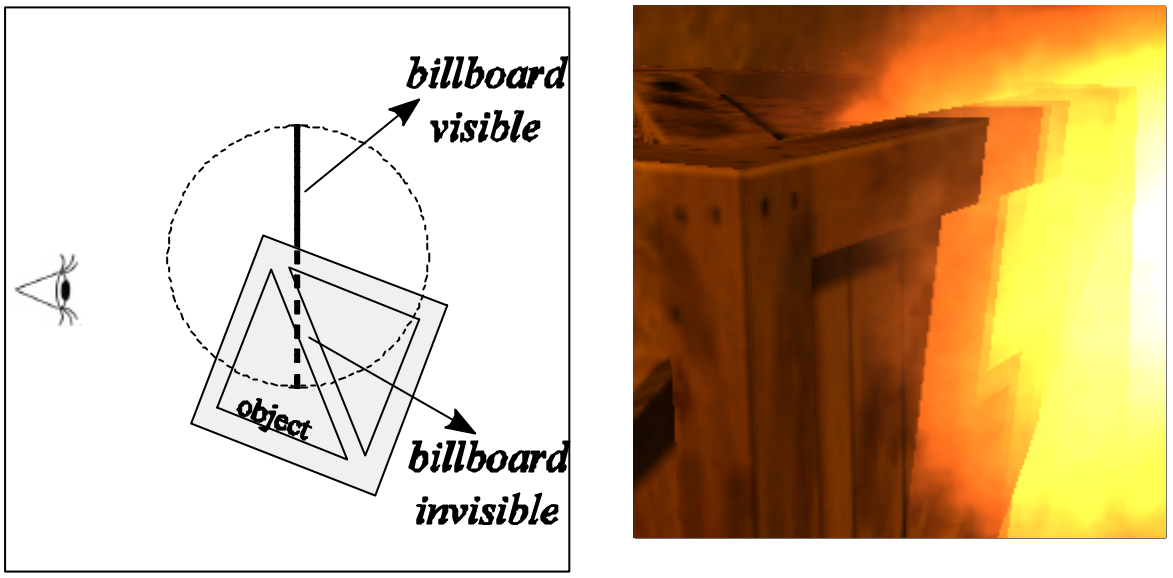


Figure 1. Billboard clipping artifact. When the billboard rectangle intersects an opaque object, transparency becomes spatially discontinuous.

Billboards are planar rectangles having no extension along one dimension. This can cause artifacts when billboards intersect opaque objects making the intersection of the billboard plane and the object clearly noticeable (figure 1). The core of this *clipping artifact* is that a billboard fades those objects that are behind it according to its transparency as if the object were fully behind the sphere of the particle. However, those objects that are in front of the billboard plane are not faded at all, thus transparency changes abruptly at the object–billboard intersection.

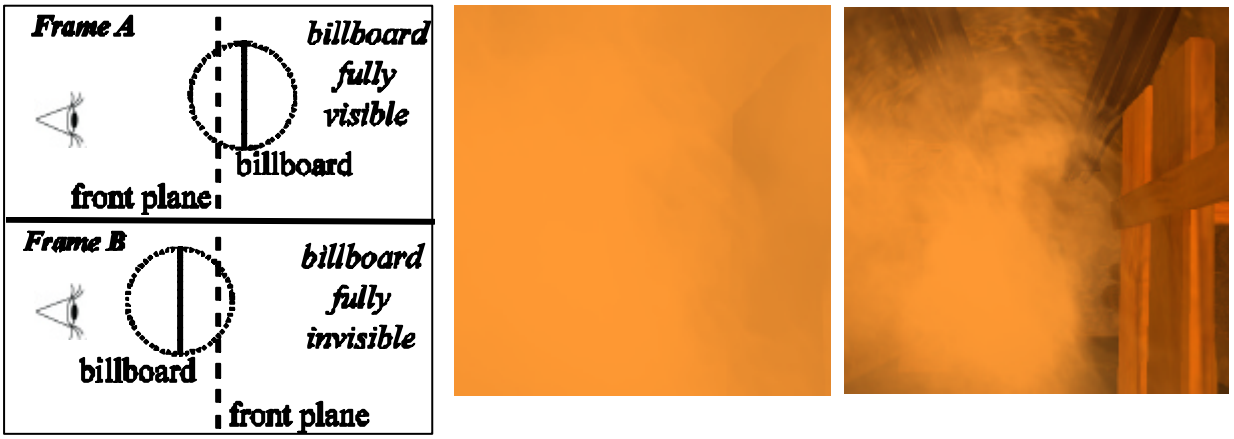


Figure 2. Billboard popping artifact. Where the billboard gets to the other side of the front clipping plane, the transparency is discontinuous in time (the figure shows two adjacent frames in an animation).

On the other hand, when the camera moves into the media, billboards also cause *popping artifacts.* In this case, the billboard is either behind or in front of the front clipping plane, and the transition between the two stages is instantaneous. The former case corresponds to a fully visible, while the latter to a fully invisible particle, which results in an abrupt change during animation (figure 2).

To solve the problems of clipping and popping we introduce spherical billboards in section 2. The proposed idea is used then to render realistic explosions in section 3.

# 2. Spherical billboards

Billboard clipping artifacts are solved by calculating the real path length a light ray travels inside a given particle since this length determines the opacity value to be used during rendering. The traveled distance is obtained from the spherical geometry of particles instead of assuming that a particle can be represented by a planar rectangle. However, in order to keep the implementation simple and fast, we still send the particles through the rendering pipeline as quadrilateral primitives, and take into account the spherical shape only during fragment processing.

To find out where opaque objects are during particle rendering, first these objects are drawn and the resulting depth buffer storing camera space *z* coordinates is saved in a texture. Having rendered the opaque objects, particle systems are processed. The particles are rendered as quads perpendicular to axis *z* of the camera coordinate system. These quads are placed at the farthest point of the particle sphere to avoid unwanted front plane clipping. Disabling depth test is also needed to eliminate incorrect object–billboard clipping.

When rendering a fragment of the particle, we compute the interval the ray travels inside the particle sphere in camera space. This interval is obtained considering the saved depth values of opaque objects and the camera's front clipping plane distance. From the resulting interval we can compute the opacity for each fragment in such a way that both fully and partially visible particles are displayed correctly, giving the illusion of a volumetric medium. During opacity computation we assume that the density is uniform inside a particle sphere.

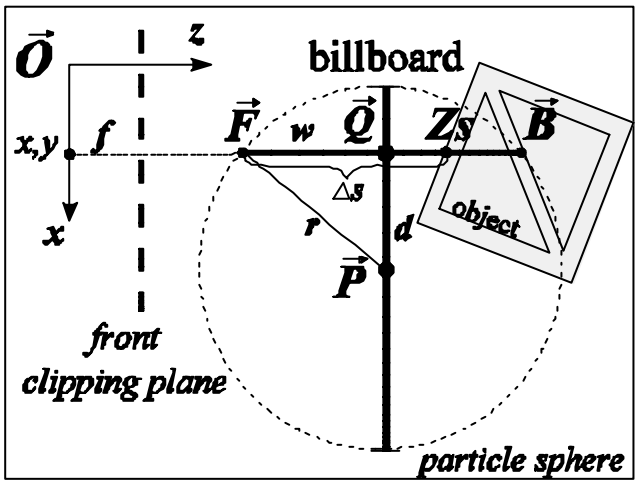


Figure 3. Computation of length $\Delta s$ the ray segment travels inside a particle sphere in camera space.

Let us use the notations of figure 3. We consider the fragment processing of a particle of center $P = (x_p, y_p, z_p)$, which is rendered as a quad perpendicular to axis *z*. Suppose that the current fragment corresponds to the visibility ray cast through point $Q = (x_q, y_q, z_q)$ of the quadrilateral. Although visibility rays start in the origin of the camera coordinate system and thus form a perspective bundle, for the sake of simplicity, we consider them as being parallel

with axis *z*. This approximation is acceptable if the perspective distortion is not too strong. If the visibility ray is parallel with axis *z*, then the *z* coordinates of $\vec{P}$ and $\vec{Q}$ are identical, i.e. $z_p = z_q$.

The radius of the particle sphere is denoted by *r*. The distance between the ray and the particle center is $d = \sqrt{(x - x_p)^2 + (y - y_p)^2}$. The closest and the farthest points of the particle sphere on the ray from the camera are $\vec{F}$ and $\vec{B}$, respectively. The distances of these points from the camera can be obtained as

$$\left|\vec{F}\right| \approx z_p - w$$

where $w = \sqrt{r^2 - d^2}$. The ray travels inside the particle in interval $\left[\left|\vec{F}\right|, \left|\vec{B}\right|\right]$.

Taking into account the front clipping plane and the depth values of opaque objects, these distances may be modified. First to eliminate popping artifacts, we should ensure that $\left|\vec{F}\right|$ is not smaller than the front clipping plane distance *f*, thus the distance the ray travels in the particle before reaching the front plane is not be included. Secondly we should also ensure that $\left|\vec{B}\right|$ is not greater than $Z_s$ which is the stored object depth at the given pixel, thus the distance traveled inside the object is not considered.

From these modified distances we can obtain the real length the ray travels in the particle:

$$\Delta s = \min(Z_s, \left|\vec{B}\right|) - \max(f, \left|\vec{F}\right|).$$

Assuming that the density is homogeneous inside a particle and using equation 2 to obtain the respective opacity value correspond to the piece-wise constant finite-element approximation. While constant finite-elements might be acceptable from the point of view of numerical precision, their application results in annoying visual artifacts. The problem is that although the accumulated opacity at the contour of the particle sphere becomes zero, but it does not diminish smoothly, which makes the contour of the particle sphere clearly visible for the human observer. This artifact can be eliminated if we use piece-wise linear rather than piece-wise constant finite-elements, that is, the density is supposed to be linearly decreasing with the distance from the particle center. The effects of piece-wise linear finite elements can be approximated by modulating the accumulated density by this linear function. It means that instead of equation 2 we use the following formula to obtain the opacity of particle *j*:

$$a_j \approx 1 - e^{-t_j (1 - d / r_j) \Delta s_j}$$

where *d* is the distance between the ray and the particle center, and $r_j$ is the radius of this particle. Note that this approach is very similar to the trick applied in radiosity methods. While computing the patch radiosities piece-wise constant finite-elements are used, but the final image is presented with Gouraud shading, which corresponds to linear filtering.

## *2.1 GPU Implementation of spherical billboards*

The evaluation of the length the ray travels in a particle sphere and the computation of the corresponding opacity can be efficiently executed by a custom fragment shader program. The fragment program gets some of its inputs from the vertex shader: the particle position in camera space (P), the shaded billboard point in camera space (Q), the particle radius (r), and

the screen coordinates of the shaded point (`screen`). The fragment program also gets uniform parameters, including the texture of the depth values of opaque objects (`Depth`), the density (`tau`), and the camera's front clipping plane distance (`f`). The fragment program calls the following function to compute the particle opacity at the shaded fragment:

```
float Opacity(float3 P, float3 Q, float r, float2 screen) {
  float alpha = 0;
  float d = length(P.xy - Q.xy);
  if(d < r) {                       // if ray intersects the particle sphere
    float w  = sqrt(r*r - d*d);
    float F  = P.z - w;                    // entry
    float B  = P.z + w;                    // exit
    float Zs = tex2D(Depth, screen);       // get depth of opaque objects
    float ds = min(Zs, B) - max(f, F);     // traveled length
    alpha = 1 - exp(-tau * (1-d/r) * ds); // accumulated opacity
  }
  return alpha;
}
```

With this simple calculation the shader program obtains the real ray segment length (`ds`) and computes opacity `alpha` of the given particle that controls blending of the particle into the frame buffer. The consideration of the spherical geometry during fragment processing eliminates clipping and popping artifacts (see figure 6).

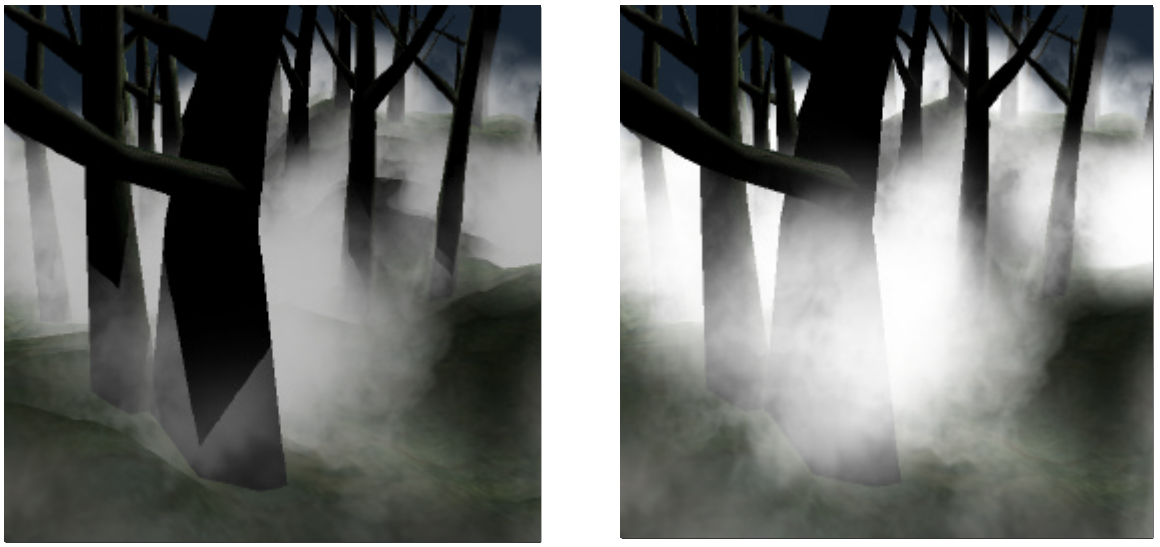

Figure 4. Particle system rendered with planar (left) and with spherical (right) billboards.

# 3. Rendering explosions

An explosion consists of dust, smoke, and fire, which are modeled by specific particle systems. Dust and smoke absorb light, fire emits light. These particle systems are rendered separately, and the final result is obtained by compositing their rendered images.

## *3.1. Dust and smoke*

Smoke particles are responsible for absorbing light in the fire. These particles typically have low albedo values $(a = 0.2, t = 0.4)$. High albedo $(a = 0.9, t = 0.4)$ dust that is swirling in the

air, on the other hand, is added to improve the realism of the explosion (figure 7).

When rendering dust and smoke we assume that these particles do not emit radiance so their emission term is zero. To calculate the in-scattering term, the length the light travels in the particle sphere, the albedo, the density, and the phase function (see equation 1) are needed. We use the Henyey-Greenstein phase function [?,?]:

$$P(\vec{w}',\vec{w}) = \frac{1}{4p} \cdot \frac{3(1-g^2)\cdot(1+(\vec{w}'\cdot\vec{w})^2)}{2(2+g^2)\cdot(1+g^2+2g(\vec{w}'\cdot\vec{w}))^{3/2}}$$

where $g \in (-1,1)$ is a material property describing how strongly the material scatters forward or backward. To speed up rendering, these function values are fetched from a prepared 2D texture addressed by $\cos q = \vec{w}'\cdot\vec{w}$ and $g$, respectively. We have found that setting $g$ to constant zero gives satisfactory results for dust and smoke.

The real length the light travels inside a smoke or dust particle is computed by the proposed spherical billboard method.

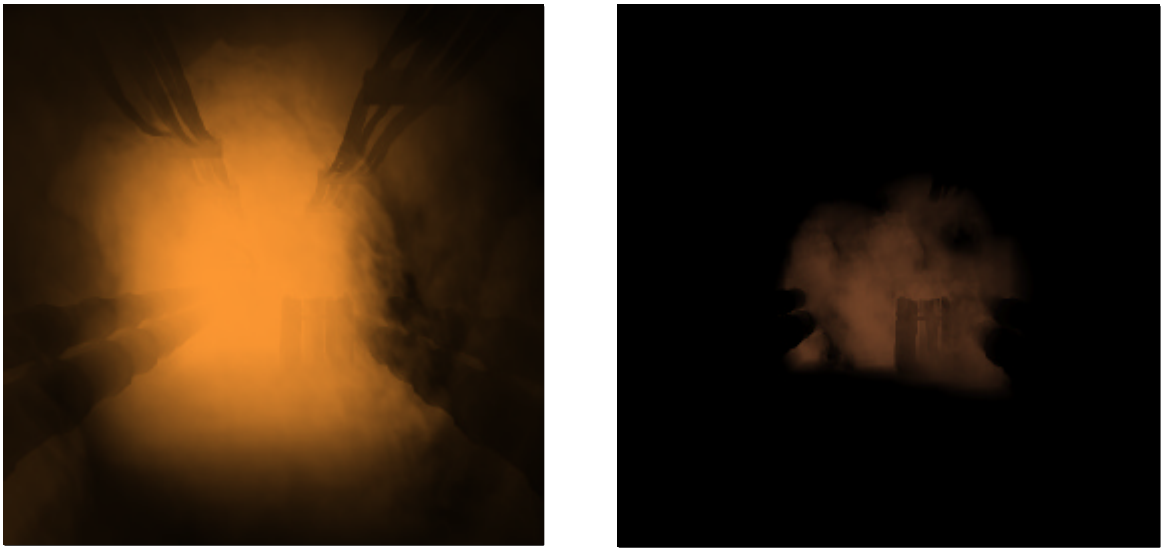

Figure 5. High albedo dust (left) and low albedo smoke (right).

In order to maintain high frame rates, the number of particles should be limited, which may compromise high detail features. To cope with this problem, the number of particles is reduced while their radius is increased. The variety needed by the details is added by perturbing the opacity values computed by spherical billboards. Each particle has a unique, time dependent perturbation pattern. The perturbation is extracted from a grey scale texture, called *detail image*, which depicts real smoke or dust (figure 8). The perturbation pattern of a particle is taken from a randomly placed, small quad shaped part of this texture. This technique has been used for a longer time by off-line renderers of the motion picture industry [?]. As time advances this texture is dynamically updated to provide variety in the time domain as well. Such animated 2D textures can be obtained from real world videos and stored as a 3D texture since inter-frame interpolation and looping can automatically be provided by the graphics hardware's texture sampling unit [?].

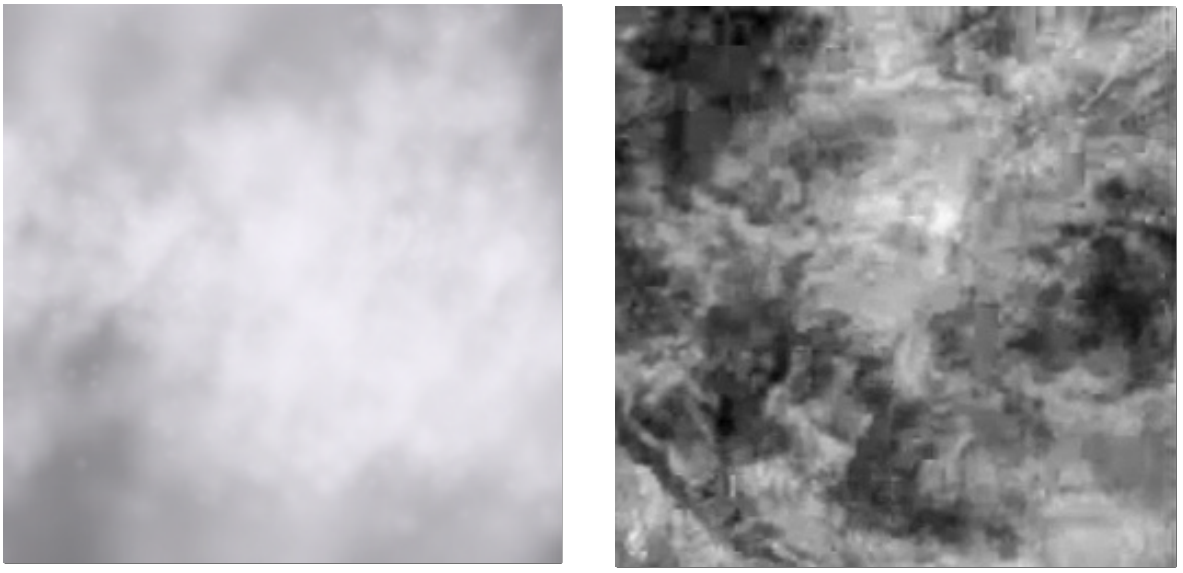

Figure 6. Images from real smoke and fire video clips, which are used to perturb the billboard fragment opacities and temperatures.

## *3.2. Fire*

Fire is modeled as a black-body radiator rather than participating medium, i.e. the albedo in equation 1 is zero, so only the emission term is needed. The color characteristics of fire particles are determined by the physics theory of black-body radiation. For wavelength $l$ , the emitted radiance of a black-body can be computed by Planck's formula:

$$L_{e,l}(x) = \frac{2C_1}{l^5(e^{C_2/(lT)} - 1)}$$

where $C_1 = 3.7418 \cdot 10^{-16} Wm^2$, $C_2 = 1.4388 \cdot 10^{-2} m^\circ K$ , and $T$ is the absolute temperature of the radiator [?]. For different temperature values, the RGB components can be obtained by integrating the spectrum multiplied by the color matching functions. These integrals can be precomputed and stored in a texture.

High detail features are added to fire particles similarly to smoke and dust particles. However, now not the opacity, but the emission radiance should be perturbed. We could use a color video and take the color samples directly from this video, but this approach would limit the freedom of controlling the temperature range and color of different explosions. Instead, we decided to store the temperature variations in the detail texture (figure 8), and its stored temperature values are scaled and are used for color computation on the fly. A randomly selected, small quadrilateral part of a frame in the detail video is assigned to a fire particle to control the temperature perturbation of the fragments of the particle billboard. The temperature is scaled and a bias is added if required. Then the resulting temperature is used to find the color of this fragment in the black-body radiation function.

The fragment program gets fire particle position in camera space (`P`), the shaded billboard point in camera space (`Q`), the particle radius (`r`), the screen coordinates of the shaded point (`screen`), and the position of the detail image in the texture (`detail`) and the starting time of the animation in `time`. The fragment program also gets uniform parameters, including the texture of the fire video (`FireVideo`), the black body radiation (`BBRad`), the depth values of opaque objects (`Depth`), the density (`tau`), the camera's front clipping plane distance (`f`),

and the temperature scale and bias in `T1` and `T0`, respectively. The fragment program calls the `Opacity` function and computes the color of the fire particle with the following Cg program:

```
float alpha = Opacity(P, Q, r, scr);            // opacity of the particle
float3 detuvw = detail + float3(0,0,time);      // texcoords of the detail
float T = T0 + T1 * tex3D(FireVideo, detuvw);   // temperature
return float4(tex1D(BBRad, T).rgb, 1) * alpha;  // emitted color
```

# 4. Layer composition

To combine the particle systems together and with the image of opaque objects, a layer composition method has been used. This way we should render the opaque objects and the particle systems into separate textures, and then compose them. This leads to three rendering passes: one pass for opaque objects, one pass for dust, fire, and smoke, and one final pass for composition. The first pass computes both the color and the depth of opaque objects.

One great advantage of rendering the participating medium into a texture is that we can use floating point blending. Another advantage is that this render pass may have smaller resolution rendering target than the final display resolution, which considerably speeds up rendering since blending needs a huge amount of pixel processing power to overdraw a pixel many times (see figure 11).

Figure 7. Particles rendered to render targets of screen (30 FPS), half (40 FPS), quarter (50 FPS), and and one eights (60 FPS) of the screen resolution.

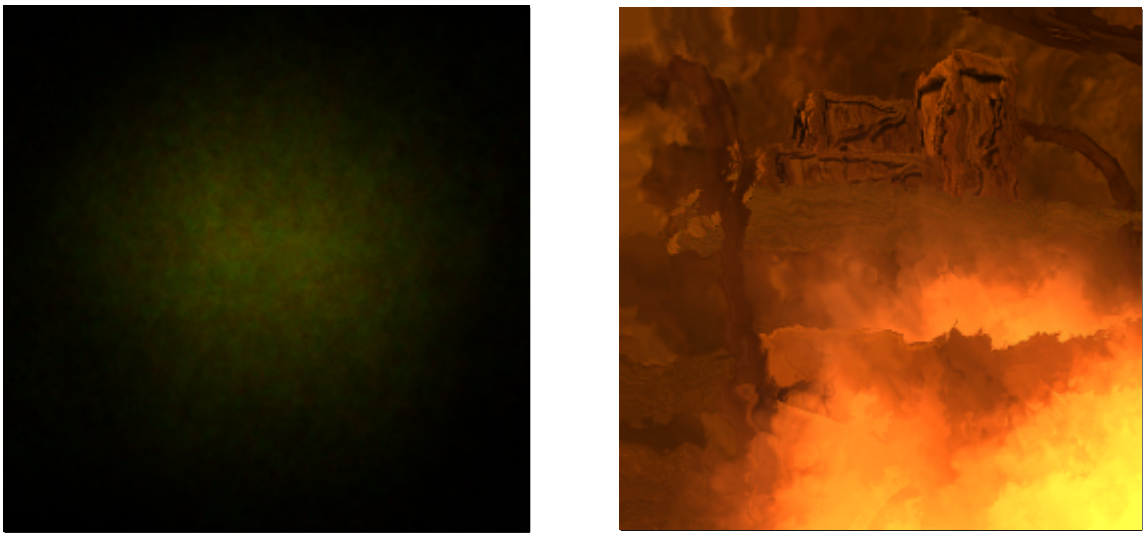

Figure 8. Heat noise texture and the final distorted image

To enhance realism, we also simulated heat shimmering that distorts the image [?]. This is done by rendering particles of a noisy texture. This noise is used in the final composition as *u*, *v* offset values to distort the image. With the help of multiple render targets, this pass can also be merged in the pass of rendering of fire particles.

The final effect that could be used through composition is motion blur, which can easily be done with blending, letting the new frame fade into previous frames. The complete rendering process is shown in figure 9.

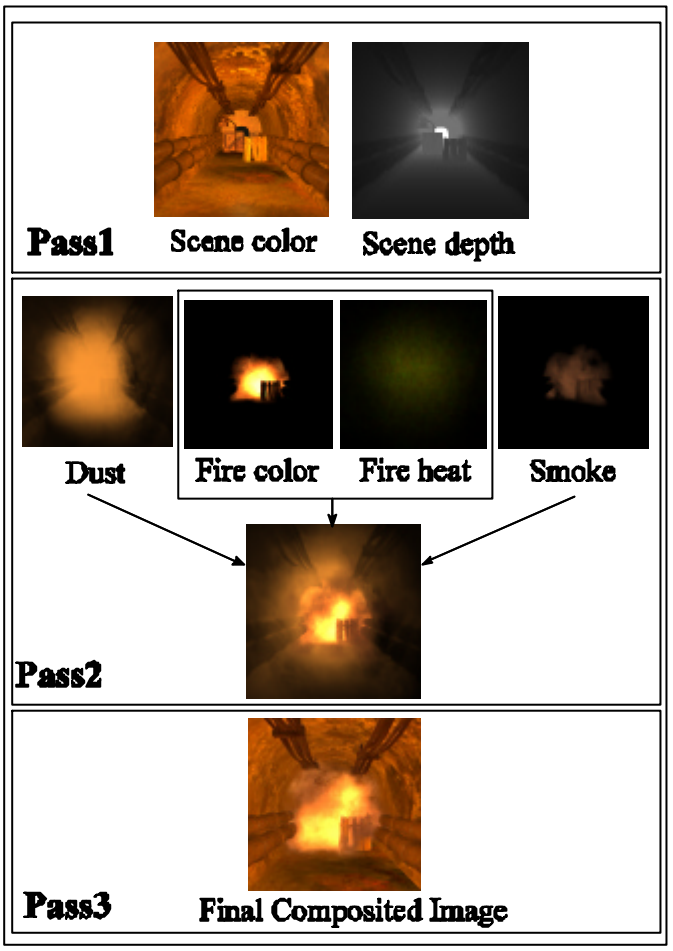


Figure 9. Rendering algorithm

# 5. Results

The presented algorithm has been implemented in OpenGL/Cg environment on an NV6800GT graphics card. The dust, the fire, and the smoke consist of 16, 115, and 28 animated particles, respectively. Note that small number of larger particles can be simulated very efficiently, but thanks to the opacity and temperature perturbation, the high frequency details are not compromised. The frame rate strongly depends on the number of overwritten pixels. For comparison, the scene without the particle system is rendered at 70 FPS, and the classic billboard rendering method would also run at about 40 FPS. This means that the performance lost due to the more complex spherical billboard calculations can be regained by decreasing the render target resolution during particle system drawing.

Figure 10. Rendered frames from the animation sequence.

# References