Soft Edges and Burning Things: Enhanced Real-Time Rendering of Particle Systems

Tommi Ilmonen
Helsinki Univ. of Technology
Telecommunications Software and Multimedia Laboratory
Tommi.Ilmonen@hut.fi

Tapio Takala
Helsinki Univ. of Technology
Telecommunications Software and Multimedia Laboratory
tta@cs.hut.fi

Juha Laitinen
Helsinki Univ. of Technology
Telecommunications Software and Multimedia Laboratory
Juha.Laitinen@tml.hut.fi

ABSTRACT

This paper describes two methods that can be used to enhance the looks of particle systems. These methods fit applications that use modern graphics hardware for rendering. The first method removes clipping artifacts. These artifacts often appear when a fuzzy particle texture intersects solid geometry, resulting in a visible, undesireable edge in the rendered graphics. This problem can be overcome by softening the edges with proper shading algorithms.

The second method of this paper presents the use of a five-component color model. The parameters of the model are: red, green, blue, alpha and burn. The first four color components have their usual meaning while the "burn" parameter is used to control the additivity of the color. This color model allows particles' alpha blending to range from pure additive (useful for rendering flames) to normal smoke-style rendering. This method can be implemented on most graphics hardware, even without shader support.

Keywords
Particle systems, OpenGL, Shaders

1 INTRODUCTION

Particle systems are typically used to render gaseous phenomena. In real-time applications the particles are rendered as polygons that have a fuzzy texture map. This paper presents two methods that can be used to make such rendering more attractive. Part of these effects can be implemented with any graphics processing unit (GPU) while some methods need shader suport in the GPU.

According to our industry contacts these methods are known in the industry and among particle system developers, but there are few if any publications about them. This article is intended to give detailed information on how these techniques can be integrated into real-world applications.

SHORT COMMUNICATION proceedings
ISBN 80-86943-05-4
WSCG'2006, January 30-February 3, 2006
Plzen, Czech Republic.
Copyright UNION Agency – Science Press

This paper is organized as follows. First we cover available publications on particle systems, then how to render fuzzy particles that intersect solid objects and finally how to achieve a useful blending mode for explosions and fire.

2 BACKGROUND

Particle systems were first published by Reeves [Ree83]. Sims has later described in more detail how to implement the dynamics of particle systems [Sim90]. The author introduced the second order particle system that included moving force fields [Ilm03].

Few publications address the rendering of particle systems. Reeves has published methods to optimize off-line rendering of complex particle systems [Ree85]. Some particle system papers also discuss the rendering of particles, e.g. Burg's introductory article on particle systems gives an excellent overview on how to render particles on graphics hardware [Bur00] (see also McAllister's article on the same topic [McA00]). Burg covers basic topics, such as texturing, alpha blending and texture animation that are widely used throughout the industry.

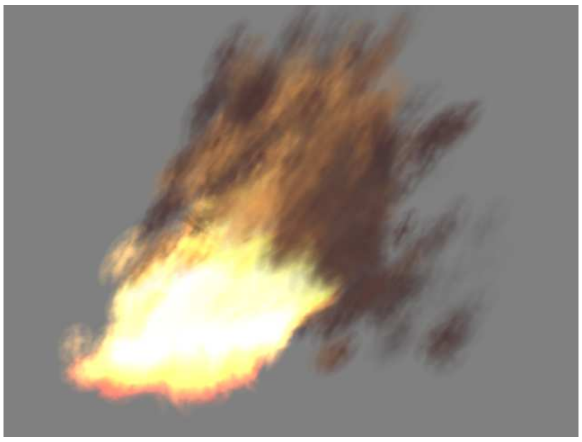


Figure 1: A fire effect. Individual fire and smoke particles are visible.

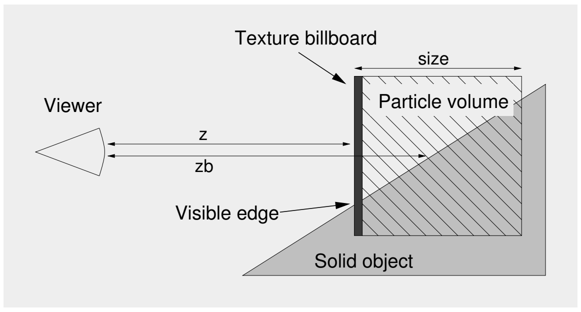


Figure 2: A particle is clipped against a solid object. The volumetric particle is rendered as a single textured billboard.

3 SOFT EDGES

In this section we deal with particles that represent fuzzy objects. Typically such particles are rendered as texture-mapped polygons. In real-time applications the particles are usually so large that individual ones can be identified. This is the case in for example figure 1.In these cases one can often see a sharp edge where the particle texture is clipped against the rigid object.

Figure 2 shows an overview of the situation. The particle is a volumetric, fuzzy object that is shown as a billboard to the user. Where the billboard intersects a solid object a visible edge can be seen. This rendering artifact usually makes a particle system look much less volumetric than intended. It also reveals the volumetric particles are in fact 2D textures. One can counter this problem by using a greater number of smaller particles. Unfortunately this approach is computationally heavy and seldom fit for real-time applications.

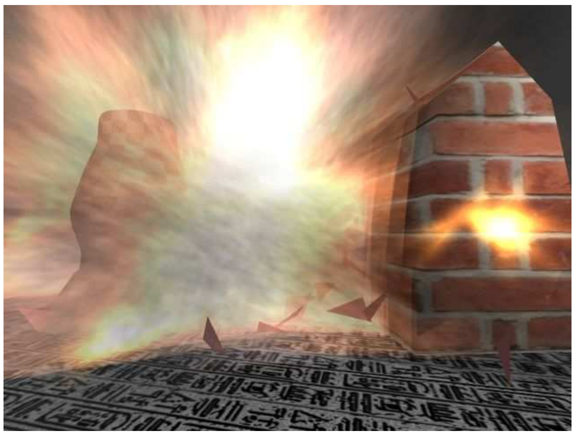
A more robust method to make clipping unobtrusive is to hide the edges with progressive alpha-blending. For example in figure 2 the lower part of the particle billboard should have an alpha value that goes linearly from one to zero. This can be done with the following pixel shader pseudo-code. In this example *pcol* is the user-defined color of the particle, *tcol* the color of the billboard texture at the pixel location, z the depth value of the billboard at the pixel locations, *zb* the depth in frame buffer, *size* represents the diameter of the particle volume, *ascale* is the scaling coefficient used to make pixels more transparent when *z* is close to *zb* and *final_color* is the color of the billboard at the pixel location (see figure 2). All colors have three color components and an alpha value that ranges from zero (fully transparent) to one (fully opaque).

```
vector4 pcol = particle_color();
vector4 tcol = texture_color();
float z = pixel_depth();
float zb = frame_buf_depth();
float size = particle_size();
float ascale = (zb - z) / size;
if(ascale > 1)
  ascale = 1;
else if(ascale < 0)
  ascale = 0;
vector4 final_color =
  pcol * tcol;
final_color.alpha *= ascale;
```

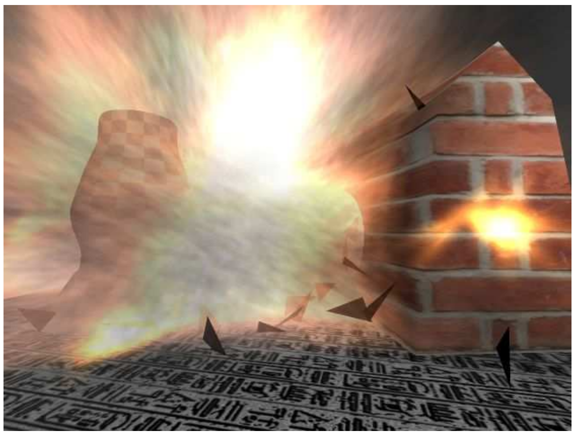
The above shader cannot be implemented directly in current PC hardware since in the modern GPU the pixel shaders cannot read depth values from the frame buffer. To counter this problem we use multi-pass rendering:

1. Render all solid objects in the scene.
2. Copy the depth values from the frame buffer to a depth texture.
3. Render particles back to front with a pixel shader that reads the depth values from the depth texture. The shader lowers the fragment's alpha as necessary value to achieve soft clipping.

We have done this with a pixel shader, using the OpenGL shading language (GLSL) [Ros04]. Since soft clipping requires more computation it is inevitably slower than normal rendering. The performance impact depends on multiple factors — the display resolution, the number of particles and their size. Thus any benchmark results are application-specific. In our tests the frame rates dropped typically to one third when soft edges were used with a heavy particle system (as in figures 3 and 5). More optimized shader implementation might improve this situation. The test computer had a 1,5 GHz AMD Athlon processor and an NVidia 5700 graphics card.



(a) Particles rendered with the default OpenGL pipeline. The intersection edges between particles and the brick wall are clearly visible (80 fps).



(b) Particles rendered with custom shading for soft clipping (28 fps).

Figure 3: Hard and soft clipping. The scene and particle locations are identical in both pictures. The frame rate of the soft clipping rendering is significantly lower.

4 ENHANCED BLENDING

The semi-transparent particle billboards need to be blended into the background. In modern GPUs the blending is controlled by blending functions that are a non-programmable part of the graphics pipeline. The two most common blend functions are additive blending and transparency blending. These functions are defined by the following formulas, where the *bg* is the RGB (red/green/blue) color of the frame buffer, *fg* is the RGB color of the particle, *alpha* represents the transparency of the particle (set by the user) and *c* is the final color of the frame buffer after updating its color value.

Additive blending adds luminance to the frame buffer:

$$c = fg * alpha + bg \tag{1}$$

Transparency blending changes the colors towards the particle color:

$$c = fg * alpha + bg * (1 - alpha) \tag{2}$$

The effect of these blending modes can be seen in figure 4. Both blend functions are useful in particle systems — the additive blend mode is widely used in fire effects and the transparency blending is useful for rendering smoke or fog.

We have developed a new blending function that can be used to create both additive and transparency blending. This is called ”controlled additive” blending and it can be used to interpolate smoothly between the additive and the transparency functions.

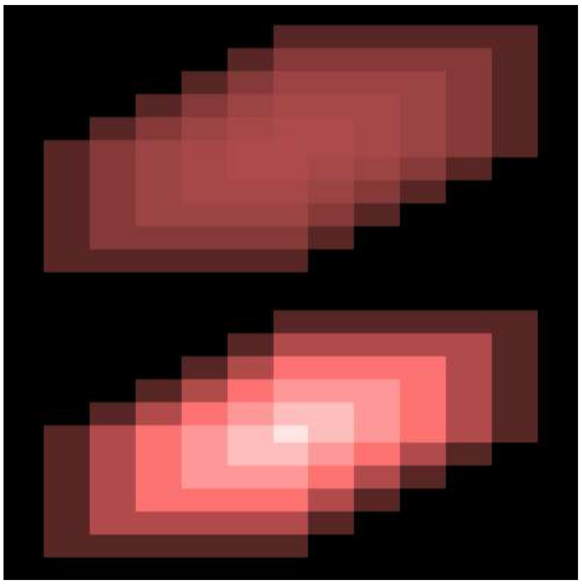


Figure 4: Different blending modes, above transparency blending of several red squares and below additive blending with same colors and geometry.

Interpolation of these two blending modes offers interesting new possibilites. The first is that one can make smooth transitions between additive and transparency blending. For example in the case of fire we can create particles that begin as additive fire particles, but later become transparency particles.

The second use of the controlled additive function is to create particle colors that are slightly additive. That is, large collection of slightly additive particles looks

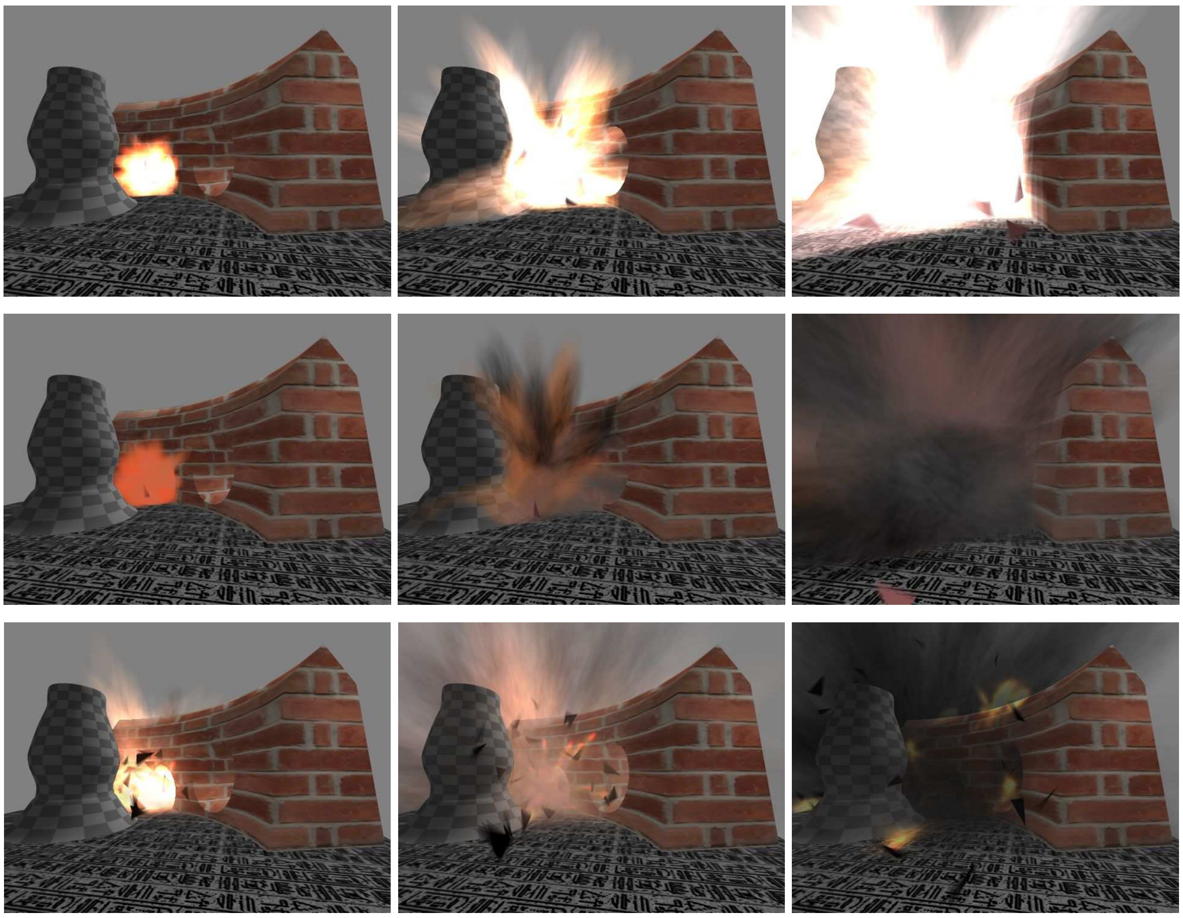


Figure 5: An explosion rendered with additive blending (top), normal transparency blending (middle) and controlled additive blending (bottom).

brighter than any single particle, but they will not result in white areas like with plain additive blending.

The controlled additivity is defined by the following blend function

$$c = fg * alpha + bg * (1 - alpha * (1 - burn)) \quad (3)$$

Here the *burn* variable is used to control the additivity of the function. The variable ranges from zero (transparency blending) to one (additive blending). This gives us a five-component color model composed of red, green, blue, alpha and burn values. Since graphics hardware only deals with four-component colors and fixed blending functions we need special methods to make the hardware render the particles with this model.

First of all we set the GPU's blending unit to use the following blend equation:

$$c = fgc + bg * (1 - alphac) \quad (4)$$

This equation gives the correct color values when

$$fgc = fg * alpha \quad (5)$$

and

$$alphac = alpha * (1 - burn) \quad (6)$$

Equations 5 and 6 need to be evaluated outside the blend unit due to hardware limitations. They can be easily calculated either in a vertex shader or in the application code. The first alternative may be slightly faster, but it also requires hardware with shader support.

The textures that are used in the rendering need special processing. In general the particle billboard texture is represented as an RGBA texture. Opening equation 6 we see that the *alpha* value of the texture has an important role in the calculations:

$$fg = ucol * tcol * talpha \quad (7)$$

thus, combining equations 5 and 7:

$$fgc = ucol * tcol * talpha * alpha \quad (8)$$

In these equations *ucol* is the RGB color set by the user, *tcol* is the RGB color in the texture and *talpha* is the alpha value of the texture. The default OpenGL graphics pipeline cannot calculate equation 8 directly. The evaluation can be done either in the pixel shader or by pre-processing the texture. We have used the latter approach since it does not require programmable pixel shaders and the work-load of the pixel units is lower, resulting in potentially better performance. To do so we pre-multiply the alpha values in the particle texture to the RGB values. Thus

$$fgc = ucol * tcola * alpha \tag{9}$$

where

$$tcola = tcol * talpha \tag{10}$$

To clarify the use of the above equations we present a pseudo-code that calculates the colors that are transmitted to the graphics hardware. In the following code *rgb* is the RGB color of particle, *alpha* represents opacity, *burn* controls additivity, *fgc* is the RGB color sent to the GPU and *alphac* is alpha value sent to the GPU. All parameters are assumed be in the range [0-1]. If the particle has a texture map, then the alpha value of the texture should be pre-multiplied to its RGB values.

```
vector3 rgb = get_rgb();
float alpha = get_alpha();
float burn  = get_burn();
vector3 fgc = rgb * alpha;
float alphac = alpha * (1-burn);
```

If one wants to use these parameters with OpenGL, then the blending mode needs to be set to match equation 7 with the following function calls:

```
glBlendEquation(GL_FUNC_ADD);
glBlendFunc(GL_ONE,
  GL_ONE_MINUS_SRC_ALPHA);
```

Once this is done the particle color can be sent to the graphics card as a normal RGBA color:

```
glColor3f(fgc[0], fgc[1],
          fgc[2], alphac);
```

So far we have assumed that particle bitmap is a four-component RGBA texture. In this case the burn parameter is assumed to be constant for the whole billboard. With this parameterization the addition of the burn parameter causes minimal performance impact. The number of texture lookups is the same as for the more traditional blending modes. In fact the pixel units of the GPU are not aware of the blending, since all of the work is done in the blend unit. The only extra calculations are the calculation of proper color and alpha parameters that need to be performed once for each particle. We have been unable to measure any performance loss due to these calculations — compared to all other work required for particle system dynemics and redering these color calculations are insignificant. We have done this calculation in the application code, but it could be moved to the vertex shader for potentially better performance.

It is also possible to add the burn parameter to the texture, creating a five-component texture. This might be useful for situations where one does not want the same burn value to apply to the whole particle. This would probably call for a separate texture with only the burn values in it as graphics hardware cannot deal with more than four color components per texture. A pixel shader would be needed to carry out the calculations. The additional texture lookups would also lower the rendering speed.

5 CONCLUSIONS

This paper has presented two techniques for rendering particle systems on modern graphics hardware. The soft-edge rendering method removes artifacts that appear when particles are rendered close to solid objects. This results in greater flexibility in application programming as developers do not need to hide or minimize the artifacts.

The controlled additive blending adds a new parameter to color definitions. The new burn parameter is useful when one needs to adjust the additivity of a particle. This parameter can be used on all GPUs that support the selection of blend equations with minimal performance impact.

Both presented methods are easy to implement and they can be used together. While these methods have been presented in the domain of rendering particle systems they may be useful in other kinds of applications where a particular special effect is desired.

6 ACKNOWLEDGEMENTS

This work has been funded by the Academy of Finland.

References

[Bur00] van der Burg, J. Building an Advanced Particle System. In Game Developer, March 2000

[Ilm03] Ilmonen, T. and Kontkanen, J. The Second Order Particle System, In The Journal of WSCG, volume 11(2), pages 240-247, 2003

[McA00] McAllister, D. The Design of an API for Particle Systems. Technical report, University of North Carolina, January 2000

[Ree83] Reeves, W. Particle Systems — a Technique for Modeling a Class of Fuzzy Objects. In Proceedings of the 10th annual conference on Computer graphics and interactive techniques, pages 359–375, 1983

[Ree85] Reeves, W. and Blau, R. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In Proceedings of the 12th annual conference on Computer graphics and interactive techniques, pages 313–322, 1985

[Ros04] Rost, R. OpenGL(R) Shading Language, Addison-Wesley, 2004

[Sim90] Sims, K. Particle animation and rendering using data parallel computation. In SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques, pages 405–413, 1990