Particle Explosions

Charles Emerson

Abstract

What is an explosion? An explosion is a rapid increase of volume in an extreme manner. Within the context of Computer-Generated Imagery (CGI), explosions appear in special effects, game development, visual simulation, virtual reality and more [1]. As computer performance improves, the problems of interest become even larger and more complex. Efficient algorithms for well-known design concepts, like explosions, becomes paramount to representing bigger, more complex, and more realistic situations. In this paper, the challenges and considerations in maintaining and representing a many-particle explosion are analyzed.

I. Introduction

Particle systems are central to CGI special effects. Specifically, many CGI special effects are composed of particles obeying some rule set and iteratively applying that rule set to each particle. Consider an explosion. Some gas particles at the origin of the explosion will be given some initial velocity. The particles will then travel away from the origin while slowing down as a function of time. The rule set for this particle system would be to "slow down" each particle as a function of time until a particle has "stopped" in which case it is removed from the system (non-moving particles are typically of no interest). This system excludes turbulent flows, the random noise associated to actual gas particles, particle collisions and even gravity. These may be inconsequential considerations or they could be realistic requirements for an accurate simulation.

There are several concerns when implementing a particle system. The first concern is to determine the goal of implementing a particle system. The goal is typically to represent some natural or fantastical phenomenon. From there, choosing a rule set to apply to fulfill that goal is important. Lesser concerns include how efficiently the particle system is updated or how the particle system will be illustrated. If the goal includes illustrating a phenomenon in real-time then these two concerns may become significant.

For this paper, the goal of implementing a particle system is to illustrate a firework-like explosion. Different rule sets will be discussed and analyzed. The efficiency of updating each particle in the system and several different representations of an explosion will also be discussed. In addition, the goal for this paper is to illustrate the firework explosion in real-time.

II. Previous Research

While there were many examples of CGI particle systems, very few papers are interested in simple particle systems in real-time. The explosion-related papers are typically focused on representing realistic explosions of incompressible gases and the smoke resulting from such explosions. One such paper, uses simplex noise to simulate real-time smoke [1]. Another paper focuses on animating suspended particle explosions but is not a real-time simulation [2]. Yet another paper focuses on using turbulence for creating "interesting" rolling explosions [3]. The focus of this paper does not include using noise, rendering realistic explosions or even rendering turbulent explosions. This paper focuses on the challenges of rendering many particles in real-time. Similar to the problem of rendering many luminous particles in real-time [4].

The biggest problem with rendering many particles is typically how to apply their rule sets quickly so their vertex representations can be sent to the GPU for rendering. The solution proposed by the paper for rendering luminous particles is clustering. Specifically, they clustered by random sampling until they had $\frac{2}{3}c$ clusters and then split the largest clusters until there were c clusters. Their reasoning is that clustering by random sampling is very fast but is not perfect and can generate a few very large clusters so the splitting helps undermine this discrepancy.

While clustering could be used to speed up some of the calculations for nearby particles in an explosion, this paper does not make use of the technique. Rather, this paper is concerned with the consequences of using a more naive approach: individually applying a rule set to many particles.

III. Methods and Implementation

One concern with an appropriate particle system model is updating the many particles quickly. However, the update speed depends on the rule set of the system. Even a simple rule set can be slow when applied per particle. For example, an obvious approach for an explosion model is to create a list of particles at the origin of the explosion, loop through each particle and decelerate each particle as a function of time. In other words, each particle gets its current velocity $\vec{v}$ added with a change in velocity $\Delta\vec{v} = -a * \Delta t * \hat{v}$ where acceleration a is constant for all particles and in the direction opposite the particle's velocity. Additionally, every particle will adhere to basic motion; a particles' velocity will be applied to its position as a function of time ($\vec{x_f} = \vec{x_o} + \Delta t * \vec{v}$). With the deceleration and the translation as functions of time (and not per iteration), the simulation will be less susceptible to visual

defects from software interrupts. Algorithm 1 makes use of this rule set as an update loop.

Algorithm 1 Applying Constant Negative Acceleration

Input: A list $particles$, whose entries have position $\vec{x}$ and velocity $\vec{v}$, and a constant negative acceleration $-a$ to apply to each entry given Δt time has occurred since the last update.

Output: The list $particles$ is updated and only contains particles still exploding outward.

1: **for** each p in $particles$ **do**
2: $p.\vec{x} = p.\vec{x} + p.\vec{v} * \Delta t$
3: let $\Delta\vec{v} = -a * \Delta t * p.\hat{v}$
4: $p.\vec{v} = p.\vec{v} + \Delta\vec{v}$
5: **if** $(p.\vec{v} \cdot \Delta\vec{v} \geq 0)$ **then**
6: Remove p from $particles$ ($p.\vec{v}$ is now in the opposite direction).
7: **end if**
8: *There might be other kill conditions (i.e. max distance traveled, minimum speed reached, etc.)*
9: **end for**
10: **return** $particles$

Algorithm 1 is a straightforward deceleration per particle implementation. However, this algorithm has a hidden inefficiency. The quantity $-a * \Delta t$ is constant (part of the Step 3 calculation) and is re-calculated each iteration. This can be resolved easily by pre-calculating before the for-loop. This is the single optimization available to this algorithm since $\vec{x}$ and $\vec{v}$ are different for every particle. However, what if there were a way of not calculating a directional acceleration per particle?

An alternative rule set is to apply a constant slow factor to velocity as a function of time. The idea is that velocity decreases by a fraction each iteration. This slow factor can be pre-calculated like the $-a * \Delta t$ quantity in Algorithm 1. Algorithm 2 illustrates the proposed method.

Algorithm 2 Applying Constant Slow Factor

Input: A list $particles$, whose entries have position $\vec{x}$ and velocity $\vec{v}$, and a constant slow factor S to apply to each entry given Δt time has occurred since the last update.

Output: The list $particles$ is updated and only contains particles still exploding outward with at least MIN_SPEED.

1: let $S' = S^{\Delta t}$
2: **for** each p in $particles$ **do**
3: $p.\vec{x} = p.\vec{x} + p.\vec{v} * \Delta t$
4: $p.\vec{v} = S' * p.\vec{v}$
5: **if** $(|p.\vec{v}| \leq$ MIN_SPEED) **then**
6: Remove p from $particles$ ($p.\vec{v}$ is small or zero).
7: **end if**
8: *There might be other kill conditions (i.e. max distance traveled, etc.)*
9: **end for**
10: **return** $particles$

Although Algorithm 2 may not seem like an improvement on Algorithm 1, there is one less calculation. The velocity direction $p.\hat{v}$ must be calculated per particle in Algorithm 1 in order to calculate the change in velocity $\Delta\vec{v}$. However, Algorithm 2's pre-calculated slow factor can be directly applied to a particle's velocity. When there are many particles to update, one less calculation per particle can be significant.

Another concern is that the particles are represented in an appropriate manner. WebGL has three representations which were considered: TRIANGLE_FAN, LINES, and POINTS. A WebGL TRIANGLE_FAN representation uses four vertices forming a square which varies in size depending on distance from the viewer. A WebGL LINES representation uses two vertices and does not vary in size. A WebGL POINTS representation uses one vertex and also does not vary in size. A TRIANGLE_FAN can show depth which is not possible with LINES or POINTS without having a special shader to show depth as color.

For the sake of simplicity, a particle explosion will be represented by an array of 10^n particles for a non-negative integer n where all particles start at the origin with the same initial speed and a random direction. The particles will be updated and maintained through Javascript and rendered using WebGL. Performance will be measured by frame rate or frames per second (FPS). Assuming the user interface (UI) is not updated every animation frame, the calculation of frame rate is the number of animation frames displayed divided by the time in seconds since the last UI update.

The particles' initial velocity will be determined by a random direction using spherical coordinates with the same initial speed for every particle. The reason for spherical coordinates is that a unit vector in any direction can be generated with two random numbers for θ and ϕ. The unit vector will then be multiplied by the initial speed to get the initial velocity of the particle.

IV. Results and Analysis

Maintaining and representing up to 10^4 particles was possible in real-time for the systems under test: a Sixth Generation ThinkPad Carbon X1 (laptop), a First Generation iPad Pro (tablet), and a Samsung Galaxy J7 V (mobile phone). Note the mobile phone was capable of creating and maintaining 10^4 particles in real-time until it was time to remove each particle from the system where there was a noticeable pause (thus failing to be real-time). Specifically, the mobile phone was able to maintain 10 FPS until the particles were "too slow" and were then to be removed. The noticeable pause is the result of splicing by 1 from an array. The issue can be resolved by changing the data structure containing the list of live particles from an array to a doubly-linked list. Note Javascript does not include a linked list in its base library. An appropriate implementation of a linked list would have to designed which is beyond the scope of this paper.

Regarding the choice of representation, using the WebGL LINES representation resulted in better performance for the 10^4 particles case. However, none of the representations had significantly better performance. This suggests to not worry about the particle explosion's representation beyond being aesthetically pleasing. Representing an explosion with a defined spherical shape required 10^3 particles, using less particles looked unnatural in some cases because significantly more particles would be on one side. In the 10^3 particles case, the sheer amount of particles makes the probability of an unnatural-looking explosion unlikely.

Figure 1 is an illustration of an explosion of 10^3 particles using a TRIANGLE_FAN representation. Note the larger squares are closer to the viewer and smaller squares are further away. The depth difference is accentuated while the particles are moving and is not properly illustrated by the following still-frame.

Figure 2 is an illustration of the same explosion using the POINTS representation. Note there is no visual difference for near or far particles. The animated explosion appears 2-dimensional with its particles moving at different speeds.

Figure 3 is an illustration of the same explosion using the LINES representation. Again, there is no visual difference between near or far particles but the origin of the explosion is clear. The LINES representation would benefit from each particle being a different color, when the LINES are the same color the result includes regions of solid color. The explosion is reminiscent of a Koosh ball (a rubber ball with 2000 rubber filaments surrounding the surface).

The reader may notice the higher particle density around the top and bottom of each of the representations. This is an artifact of the pseudo-random number generator used to generate the particle's initial velocity direction (e.g. JavaScript $Math.random()$).

Recall Algorithm 1 applied a constant negative acceleration and Algorithm 2 applied a constant slow factor. Comparing the explosions made by using Algorithm 1 and 2, Algorithm 2's explosions have better defined shape as the particles survive longer near the edges. Algorithm 1 might be used in a "realistic" rendering of an explosion but Algorithm 2 is superior for aesthetic purposes. The non-linear movement of the particles is more interesting and the particle explosion forms a defined shape at its edges.

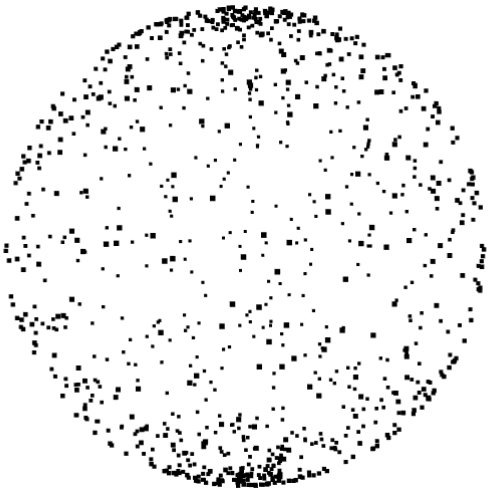

Fig. 1. A WebGL TRIANGLE_FAN representation of a particle explosion using 10^3 particles.

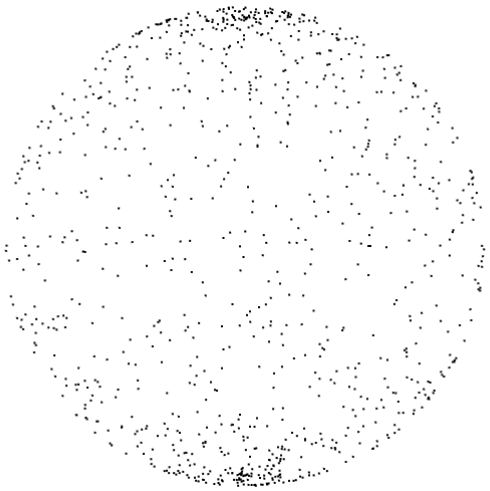

Fig. 2. A WebGL POINTS representation of a particle explosion using 10^3 particles.

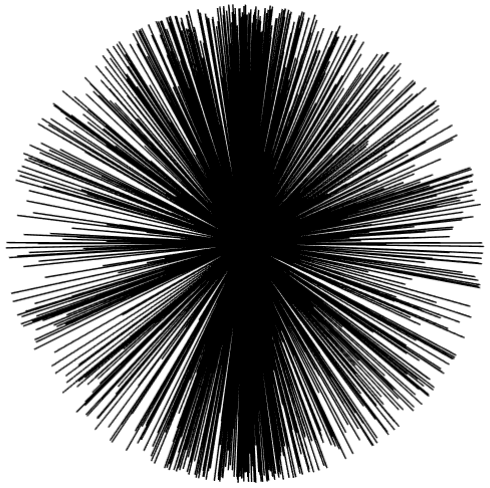

Fig. 3. A WebGL LINES representation of a particle explosion using 10^3 particles.

Comparing the performance of using the constant negative acceleration (Algorithm 1) and the constant slow factor (Algorithm 2), using a constant slow factor had slightly better average case performance. Since the slow factor was also more aesthetically pleasing than the constant deceleration, a constant slow factor is better for representing aesthetically pleasing firework explosions.

The current Javascript implementation allows for continuous explosions; every update call, a similar explosion is generated at the origin. An artifact of the implementation is that a continuous particle explosion has visible layers. Assuming the frame rate is relatively consistent and since all of the particles start with the same speed, there is a sphere around the origin where the particles will be after 1 time step and a larger sphere after 2 time steps, etc. Specifically, if the explosion is continuous then there will be particles at each layer. This is an easily noticeable pattern and may detract from the viewer's enjoyment of the firework explosion. This is more noticeable if particles created at the same time are the same color and each "wave" of particles is a different color.

When there are many particles (approximately $10,000$ on a laptop and tablet), particles start living significantly longer than expected (at least 10% longer). The discrepancy gets even worse the more particles there are. This is the result of predictive branching. The computer has to guess whether or not the particles are still alive in order to fulfill timing requirements and consistently guesses that each particle is still alive. Under the current implementation, this causes drops in frame rate when the computer finally realizes its misprediction. This could be reduced by removing all of the particles (created at the same time under the same initial conditions) from the list of particles or by having a better data structure allowing quick removal.

One of necessary optimizations for drawing many particles is how the particles' vertex representations are sent to the GPU. Rendering each particle with a separate draw call was inefficient. Instead, all of the particles' vertices were added to a buffer and a draw call was made using the buffer. This vastly improves performance and allows for thousands of particles as opposed to a few hundred. Note this requires the vertex buffer is reconstructed every animation frame. A method to avoid the buffer reconstruction would be to scale an initial buffer's particle vertices based on total distance traveled (passed via a uniform float or something similar). The method would work for a POINTS or LINES representation but would scale up the TRIANGLE_FAN representation as time passed (the representing squares would be tiny at the origin and large at the boundary).

One of the tangential experiments included different rule sets. There were two notable rule sets resulting from these experiments: constant acceleration and constant speed factor (think the opposite of Algorithm 1 and 2, respectively). Constantly accelerating particles created an explosion that did not look significantly different from a constant deceleration. This is notable because it implies that a constant speed could be used to illustrate an explosion justs as well as Algorithm 1. While the constant speed factor did not look like any of its predecessors. The constant speed factor was reminiscent of the Star Wars Hyper-drive scene, every particle seemed to slowly stretch outward.

The current limitation is the data structure maintaining the particles. An array is not appropriate if the goal is to allow individually applied rule sets. One possible improvement is to remove all particles created at the same time (which are stored consecutively). However, this improvement requires all particles have the same initial conditions (same initial speed), the same rule set, and the same kill boundary (a minimum speed or maximum distance traveled, etc.). As such, an array is only appropriate when removing many particles at once; a linked list would be more appropriate for allowing individual particles with individual kill conditions.

V. Conclusion

In conclusion, up to 10^4 particles could be created and maintained in real-time, but removal by 1 proved to be a problem. The issue is every particle reaches its kill condition at the same time and removal by 1 from an array is slow. The result would benefit from a data structure that allowed for fast removal. In all cases, the choice of representation did not significantly affect performance.

VI. Post Mortem

The initial proposal was not well formed. Specifically, there were no project requirements beyond making an explosion. After making a basic explosion, much of the time spent on the project involved trying different rule sets with varying parameters (like initial speed) and drawing different representations with many colors.

A good project proposal would have answered the following questions:

- What is the definition of an explosion? What are the invariants of particles in an explosion? Is the explosion in 2 or 3 dimensions?
- How will the result be measured (frame rate, number of particles alive simultaneously, etc)?
- What systems will the result be tested on (laptop, tablet, mobile phone, etc)?
- How much error is inherent to the system? How much longer do particles live than expected?
- What will the result be compared to?

VII. Future Work

The most important next step is using a more appropriate data structure. Using a linked list for efficient removal would allow for individually complex particle rule sets. This paper was only able to focus on two relatively simple rule sets because of the penalty for removing particles from the system.

After implementing a better data structure, an appropriate following project would involve identifying the rule sets necessary to illustrate different types of explosions. Specifically, identify the rule sets governing particles in different types of fireworks like the Chrysanthemum, the Peony, the Palm, etc.

Illustrating different fireworks would require particle luminescence and time-varying shading options. Every type of firework has some defining aspect which needs predictable noise. Predictable noise could come from a texture map to govern initial conditions or particle motion. Some kinds of fireworks would benefit from clustering since clustered sections split up into many particles after a period of time. Additionally, some fireworks are turbulent and would need rolling motion.

In summary, further future work could involve:

- Using a data structure for managing the particles that allows for efficient removal.
- Adding luminescence to the particles.
- Shading the particles in interesting ways (i.e. based on speed, distance traveled, or time alive).
- Introducing predictable noise using a texture map for determining particle initial conditions or governing particle motion.
- Clustering the particles to update the system faster.
- Adding turbulence or gravity to motion of the particles.

Acknowledgments

The Fluxions Library, created by Jonathan B. Metzgar, was the foundation for the Javascript implementation of a particle explosion. The base HTML and CSS of the website where the implementation is available, "https://cjemerson.github.io/ParticleExplosion/", is also the work of Metzgar.

References

[1] Xihou Li, Hao Wang, and Hongyu Yang. Simulation of real-time explosion smoke based on simplex-noise. In *2010 International Conference on Information, Networking and Automation (ICINA)*, volume 2, pages V2–119–V2–123, Oct 2010.

[2] Bryan E. Feldman, James F. O'Brien, and Okan Arikan. Animating suspended particle explosions. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 708–715, New York, NY, USA, 2003. ACM.

[3] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. A vortex particle method for smoke, water and explosions. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 910–914, New York, NY, USA, 2005. ACM.

[4] Defang Deng and Shuangjiu Xiao. Clustering-based real-time lighting simulation for self-luminous particle system. In *Proceedings of the 13th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, VRCAI '14, pages 193–196, New York, NY, USA, 2014. ACM.