A Dynamic Multi-contextual GPU-based Particle System using Vector Fields for Particle Propagation

Michael J. Smith[+*] Roger V. Hoang[+*] Matthew R. Sgambati[+*]
Sergiu M. Dascalu[+] Frederick C. Harris, Jr.[+*]

Department of Computer Science and Engineering[+]
University of Nevada, Reno
Reno, NV 89557

CAVCaM[*]
Desert Research Institute
Reno, NV 89512

{mjs, hoangr, sgambati, dascalus, Fred.Harris}@cse.unr.edu

Abstract

Particle systems have long been used in scientific visualizations. The advancement of graphics technology has allowed for the computations of these systems to be performed on the graphics processing units (GPUs). The parallelism offered by these devices allows for a greater number of particles to be updated in real-time. Scientists can immerse themselves in these systems through the use of virtual reality. However, the use of such environments with multiple screens and multiple rendering contexts presents data synchronization problems with respect to dynamic GPU data. We present a solution to this problem and apply it to the visualization of brownouts caused by helicopter downwash.

Keywords: Virtual Reality, GPU, Multi-Contextual

1 Introduction

Due to the loose sand, desert terrains are difficult areas for helicopters to land. The downwash generated by rotors can cause the loose sand particles to be lifted up into the air resulting in reduced visibility, engine clogging, and unsafe flying conditions. Visualizing these brownouts using virtual reality allows scientists to better understand and study this phenomenon and allows pilots to better prepare for such situations.

One such way to visualize this phenonmenon is to use a particle system. By storing the forces generated by the downwash in a vector field, we can visualize the flow of sand particles through the air by subjecting each particle to the forces of the wind field. While particle systems have tradionally been computed on the CPU, the advent of the programmable shader and the application of GPUs as massively data-parallel general purpose processors have resulted in the migration of these calculations onto the GPU, allowing for a much larger number of particles to be visualized while maintaining real-time framerates.

Not only has the increase in computing power allowed real-time particle systems to grow in complexity, it has also allowed these visualizations to be used in virtual reality (VR) environments. The use of VR and 3D user interfaces allows a user to more intuitively examine and manipulate 3D data such as vector fields; additionally, realistic visualizations of these dust simulations can be used to prepare helicopter pilots to respond to brownout conditions.

While both GPU-based particle systems and virtual reality technology can contribute to the effectiveness of dust simulations and visualizations, integration of the two can be non-trivial. A VR system such as the a CAVE consists of multiple screens that must be synchronized. Each screen is usually rendered to by a separate GPU with its own separate video memory. Due to the random nature of particle systems, the particle data in video memory must be kept synchronized between all GPUs to ensure that particles traveling from one screen to another remain consistent. With a large number of particles or a large number of GPUs, the bandwidth costs of transferring these computations between components may outweigh any benefit to be gained from using these systems.

In this paper, we present a solution to this problem by replicating all GPU-based computations using the same random variables. This ensures that all of the results will be consistent across all rendering contexts. The remainder of this paper is structured as follows: Section 2 discusses the background and related work of GPU particle systems; Section 3 outlines our approach to the problem; Section 4 details the implementation of our prototype; Section 5 presents our results and Section 6 closes with conclusions and future work.

2 Background

2.1 Virtual Reality

The ability to immerse oneself in an application or simulation has interested both scientists and engineers alike. Virtual reality works towards this goal. In addition to depth information normally provided by visualizations flatly displayed on desktop monitors, the visual aspects of virtual reality also present the user with stereoscopic depth cues by rendering a slightly different image to each eye [13].

Within the realm of virtual reality systems, multi-screen environments, such as CAVE (see Figure 1) and large wall displays, are typically connected to more than one graphics pipe. Pipes can be connected to different graphics cards and hence different rendering contexts. In order to ensure that an image remains consistent across all rendering contexts, the data within these contexts must be synchronized.

Figure 1: Four-sided FLEX CAVE.

2.2 GPU Offloading

With games and visualization driving the evolution of graphics processors, the fixed functionality of the rendering pipeline once offered has been steadily replaced by the introduction of programmable pipeline components called shaders. These shaders not only allow the GPU to be used for more elaborate graphical effects but also allow it to be used for more general purpose computations. By storing general data as texture data, user-programmed vertex and fragment shaders can transform the GPU into a highly data-parallel multiprocessor [10]. Figure 2 illustrates the flow of graphics data from the CPU to the GPU in a typical graphics application and highlights areas of user-programmability.

While a large number of fragment shader instances can be executed on the GPU in parallel, each instance can only output a single fragment of data. Newer GPUs insert another programmable component between the vertex and fragment stages called a geometry shader [8]. This type of shader is capable of emitting a variable number of vertices. Along with the introduction of this shader type, these newer GPUs also introduce the ability to stream this variable size output to a vertex buffer object, a one-dimensional array of video memory [5][9].

2.3 GPU Particle Systems

Particle systems have long been used to model phenomena with no particularly well-defined surfaces [11]. A cloud of particles is instead used to model the volume that the phenomenon inhabits. In general, particle systems are implemented with five stages that are performed at each time step.

- New particles are created.
- Parameters for newly created particles are initialized.
- Particles that have existed past their assigned lifetimes are destroyed.
- Existing particles are transformed by some update function.
- Particles are rendered onto the screen.

Depending on the type of phenomenon being modeled, a high degree of parallelization can be achieved. Creation, updating, and destruction of particles all essentially operate on large arrays of independent data that can be acted upon by multiple processes or threads simultaneously. Rendering is inherently parallel due to the nature of graphics processor design.

Even if every stage of a particle system can be parallelized, the cost of communication can become the new bottleneck in the system. As the rendering phase consists of sending graphics data from main memory to a video card, the graphics bus can quickly limit the amount of particles that can visualized per frame. To ameliorate this situation, recent developments in particle systems have seen a shift of all computation to the GPU. This shift not only relieves the load on the graphics bus but also exploits the highly data-parallel nature of the GPU.

One such GPU particle system is the UberFlow system described in [2]. Particle data is stored as 2D textures and updated by rendering these data textures over another set of data textures, overwriting them in the process. The textures are then swapped for the

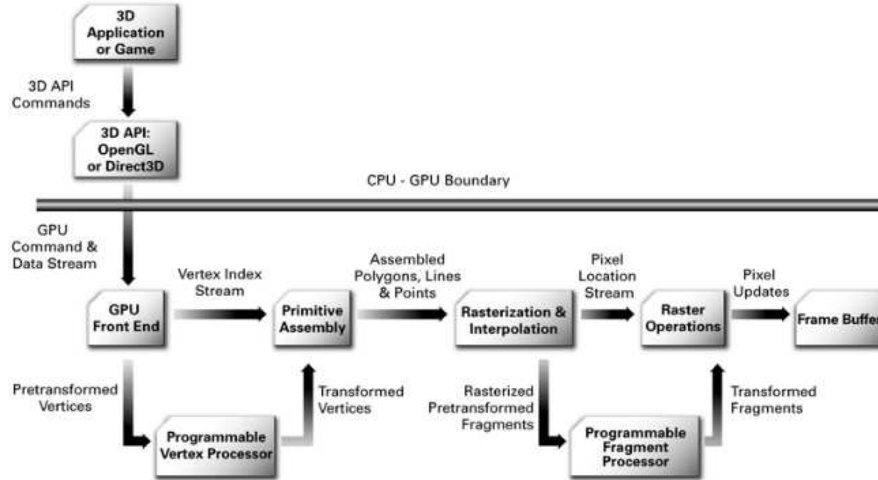

Figure 2: The graphics pipeline [7].

next update cycle. The number of particles is statically limited by the size of the texture with the particle emission and death rates dictated by this limit in conjunction with particle lifetimes. Particles that die simply reset themselves. Particles are rendered by essentially recasting the texture data as a vertex array and streaming this array through a render program.

A similar approach is described in [3]. Particle data is again stored in textures and double-buffered for updating. Particle creation and destruction, however, is done in a different manner. Indices to every texel not inhabited by a living particle are maintained by the CPU. Particle destruction is monitored by both the CPU and GPU; upon death, the CPU adds the texel index back to its pool of free indices. Particles are created on the CPU and sent down to the GPU by rendering points at these free locations. Rendering is performed by asynchronously copying the texture data into a vertex array.

While [2] forces the creation of a new particle upon the death of another and [3] must use the CPU to maintain particle creation and destruction, a technique described in [4] allows for emitters to emit independently while moving all computation to be performed on the GPU. Particle data along with emitter data is double-buffered in vertex buffer objects. Data is updated by streaming the vertex buffers through a geometry shader. Any data that will persist past the update is streamed back out with updated values. Emitters that emit particles during an update stream an extra vertex out containing data for the newly created particle. The output of the geometry shader is streamed into another vertex buffer object that can be directly used to render the particles and/or used as input for the next update cycle.

3 Design

As discussed, the use of GPU-based particle systems provides considerable advantages for creating real-time immersive simulations and visualizations. However, the use of these particle systems in conjunction with VR systems presents a unique problem. The multi-contextual nature, that is, the fact that there are multiple sets of video memory of VR systems combined with the random nature of particle systems requires that either the computations for all particles are done on a single machine and distributed out, or the computations are replicated for each rendering context. Our solution to the problem takes the latter approach.

To achieve such a system all contexts are updated in lock-step. That is, all contexts must be updated to a particular time step before continuing to the next step. Within each time step, each context is updated using the exact same delta time and random variables; as a result, all particle data should be exactly the same for all contexts.

4 Implementation

To allow the emission computations to also be performed on the GPU, the particle system was implemented using geometry shaders and vertex streamout.

Particle and emitter data is stored into a vertex buffer object, then streamed through a geometry shader. The geometry shader determines wheter a particular "vertex" is a particle or an emitter and updates it accordingly. All live objects are steamed out from the geometry shader and into another vertex buffer object. Particles that have expired after this update do not get streamed out again; additionally, emitters stream out not only themselves but also any particles that they emit.

When particles are emitted they are given a random initial position, velocity and lifetime (see Figure 3). This randomness must be controlled in order to keep the particle data consistent between contexts. To do so, we generate a large set of random numbers at initialization time. This same set of numbers is stored in a texture for each context. Also during initialization, every emitter is given a set of texture coordinates that it uses to fetch values from the random texture. During each update step, each context is given the same randomly generated texture transformation matrix to shift all of the coordinates of the random texture resulting in a new set of random values being fetched by the geometry shader.

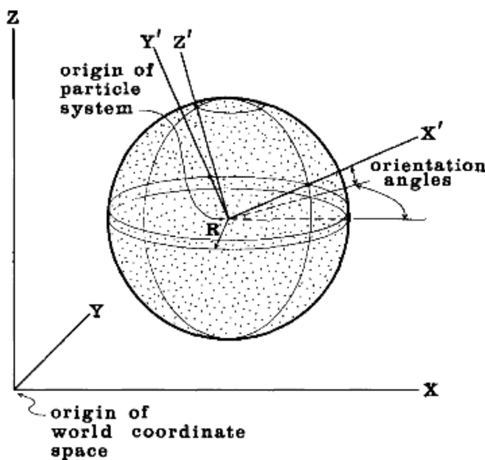

Figure 3: Particle emission [11].

Particle motion is dictated by a 3D user-generated vector field [1]. The vector field is maintained in video memory as a 3D texture. When the geometry shader updates a particle, it uses the particle's current position to look up the appropriate voxel in the 3D texture. It then applies the force stored in that voxel to the particle. Hardware-accelerated interpolation can be used to obtain a more realistic force vector.

Particles are rendered by streaming the particle data into another geometry shader. If the data is determined to be a particle then the geometry shader constructs a textured billboard and streams it out to be rendered; otherwise, nothing is streamed out. In order to create more realistic looking dust without overusing an expensive pixel shader to do so, deferred shading is used instead [6]. Rather than rendering a colored image of a particle to the visible framebuffer, data such as the density and normals of each particle is accumulated in a separate framebuffer. This data is then used to composite a dust layer over the rendered scene. The pixel shader that composites this image uses this data to perform lighting and blending computations.

5 Results

A prototype was implemented using OpenGL and its shader language GLSL. The VR environment was handled through the use of FreeVR, a virtual reality library [12]. The prototype was run on the four screen CAVE-like system at the Desert Research Institute. The system driving the virtual environment consists of four quad-core Xeon processors, 48 GB of RAM, and a NVIDIA Quadroplex FX 5600 Model IV with two GPUs rendering to four screens.

Figure 4 shows the result of the prototype running on multiple screens with a rendering context per screen. The yellow line highlights the edge where the two displays meet. As evidenced by the image, particles that move between the two displays form a consistent image to the user.

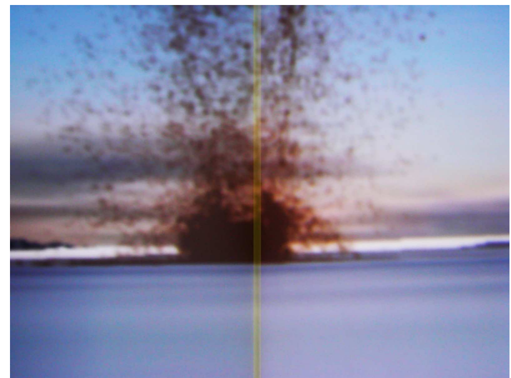

Figure 4: Particles moving across two contexts.

Figures 5 and 6 visualizes the vector field stored in a 3D texture. Larger arrows represent forces of greater magnitude. Figure 7 shows the paths of various particles as they are propagated through the vector field.

Figure 8 shows the final prototype with a helicopter model and a user-generated vector field. The user uses a tracked wand device to maneuver the helicopter by tilting the wand to control the speed of the rotor blades. Dust emission rates increase as the helicopter

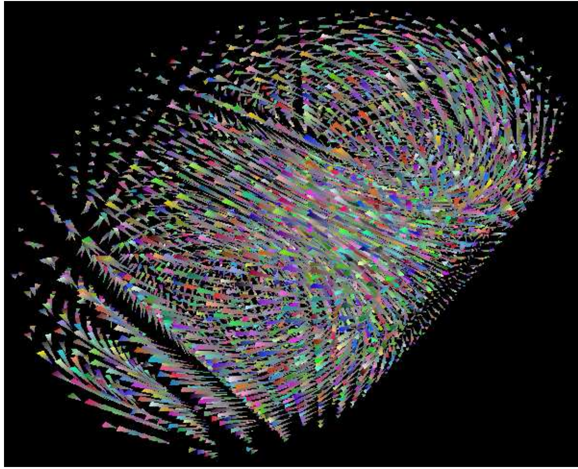

Figure 5: A vector field.

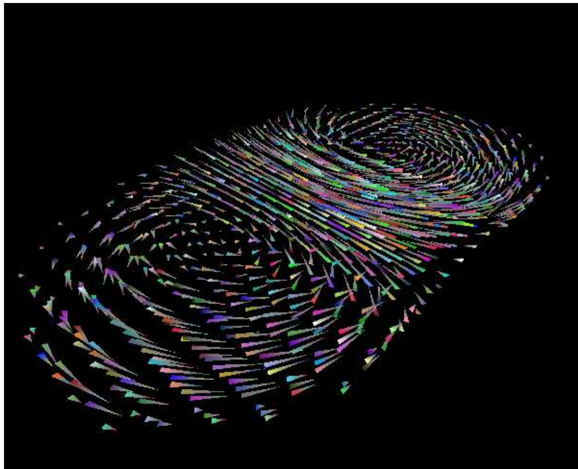

Figure 6: Several slices of the vector field.

approaches the ground, as do the forces in the vector field.

In terms of performance, the system was able to render a particle system consisting of over 300,000 particles. When viewed from afar, the system was able to run at 65 FPS. When immersed in the particle system, the framerate drops to between 15 and 20 FPS due to the system becoming fill-rate limited. It would be expected that the framerate would increase drastically on a machine where one GPU were dedicated to each rendering context in order to avoid potentially expensive context switches.

6 Conclusions and Future Work

We have presented a method for simulating and visualizing GPU-based particle systems in a multi-contextual environment. By replicating computations

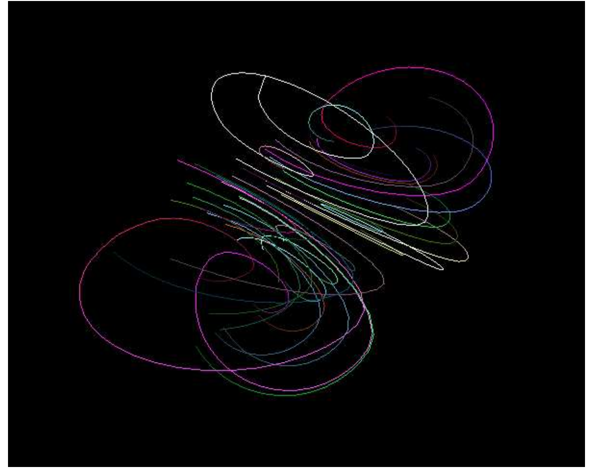

Figure 7: Particle paths through the vector field.

in lock step for each rendering context, a consistent system was visualized across multiple screens. Through the use of the GPU, the number of particles that could be simulated and visualized at interactive framerates was significantly increased.

Our prototype implements this solution and allows for a vector field to be visualized through the use of a particle system. While the vector field used in the prototype was a static user-generated one, the system is capable of using collected field data or simulated data to drive the system. In addition, although the prototype was intended to visualize a dust system, modifications to the rendering components of the system can be performed in order to visualize other phenomena, such as air flow and fluid simulations. Furthermore, our GPU-based solution allows for a much larger particle system. Even with greatly increased particle emission rates, there was a negligible decrease in system performance.

As our system is merely a prototype at this point in time, several improvments can be made. In order to increase the realism of the particle flow, collision detection must be implemented not only with static objects such as the terrain but also dynamic objects such as the helicopter itself and other particles. Additionally, a real-time air flow simulation can be implemented. As the particle computations and vector field already reside on the GPU, a fluid simulation should also be performed on the GPU, again, replicating all inputs to ensure that the resultant field is also consistent across all rendering contexts.

Several improvements can also be made to increase the visual fidelity and responsiveness of the system. As the particles are rendered using billboards, intersection edges with other objects in the scene are highly visible. Techniques such as using soft particles can

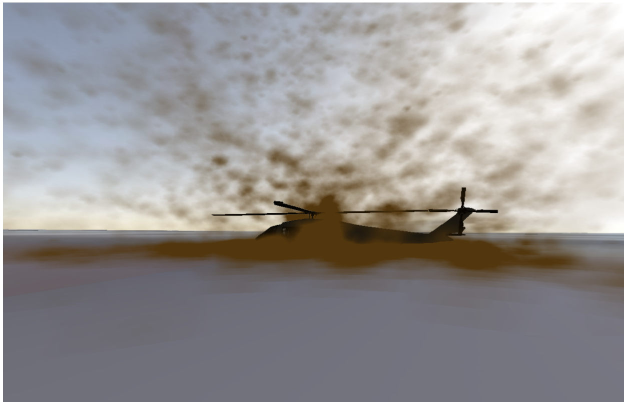

Figure 8: A helicopter-generated particle system.

ameliorate this problem. To increase the performance of the system, optimizations such as separating emitters and particles into separate data structures can be done in order to decrease the amount of redundant branching that is currently performed within the update geometry shader.

Acknowledgements

This work was funded by the STTC CAVE Project (ARO# N61339-04-C-0072) at the Desert Research Institute.

References

[1] Travis L. Hilton and Parris K. Egbert. Vector fields: an interactive tool for animation, modeling and simulation with physically based 3d particle systems and soft objects. In *Computer Graphics Forum*, pages 329–338, Aire-la-Ville, Switzerland, 1994. Eurographics.

[2] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM.

[3] Lutz Lata. Building a million particle system. In *Proceedings of the Game Developers Conference 2004*, 2004.

[4] Microsoft. ParticleGS Sample. `http://msdn2.microsoft.com/en-us/library/bb205329(VS.85).aspx` (Accessed April 23rd, 2008).

[5] Microsoft. Stream-Output Stage (Direct3D 10). `http://msdn2.microsoft.com/en-us/library/bb205121(VS.85).aspx` (Accessed April 23rd, 2008).

[6] Nicholas Francis, Over The Edge Entertainment. Deferred Particle Shading, Cooler Looking Smoke For Games. `http://unity3d.com/blogs/nf/` (Accessed April 23, 2008).

[7] NVIDIA. Cg toolkit user's manual. `http://developer.download.nvidia.com/cg/Cg_2.0/2.0.0015/CgUsersManual.pdf`, January 2004.

[8] Nvidia Corporation. OpenGL Geometry Shader 4 Extension. `http://developer.download.nvidia.com/opengl/specs/GL_EXT_geometry_shader4.txt` (Accessed April 23, 2008).

[9] Nvidia Corporation. OpenGL Transform Feedback Extension. `http://developer.download.nvidia.com/opengl/specs/GL_NV_transform_feedback.txt` (Accessed April 23rd, 2008).

[10] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[11] William T. Reeves. Particle systems–a technique for modeling a class of fuzzy objects. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 359–375, New York, NY, USA, 1983. ACM Press.

[12] William R. Sherman. Freevr. `http://www.freevr.org/(AccessedApril23,2008)`.

[13] William R. Sherman and Alan B. Craig. *Understanding Virtual Reality*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2003.