2.4

Massively Parallel Particle Systems on the GPU

Lutz Latta

Introduction

Reality is full of motion, full of chaos, and full of fuzzy objects. Physically correct particle systems (PS) are designed to add these essential properties to the virtual world. Over the last decades they have been established as a valuable technique for a variety of volumetric effects, both in real-time applications and in pre-rendered visual effects of motion pictures and commercials.

Particle systems have a long history in video games and computer graphics. Very early video games in the 1960s already used 2D pixel clouds to simulate explosions. The first publication about the use of dynamic PS in computer graphics was written after the completion of the visual effects for the motion picture *Star Trek II* at Lucasfilm [Reeves83]. Reeves describes basic motion operations and basic data representing a particle—both have not been altered much since. An implementation on parallel processors of a super computer has been done by [Sims90]. He and [McAllister00] also describe many of the velocity and position operations of the motion simulation that are used below. The latest description of CPU-based PS for use in video games has been done by [Burg00].

Real-time PS are often limited by the fill rate or the CPU-to-graphics hardware (GPU) communication. The fill rate, the number of pixels the GPU can draw for each frame, is often a limiting factor when there is a high overdraw, i.e., single particles are relatively large and a lot of them overlap each other. Since the realism of a particle system simulation increases when smaller particles are used, the fill rate limitation loses importance. The second limitation, the transfer bandwidth of particle data from the simulation on the CPU to the rendering on the GPU, now dominates the system. Sharing the graphics bus with many other rendering tasks allows CPU-based PS to achieve only up to 10,000 particles per frame in typical game applications. Therefore, it is desirable to minimize the amount of communication of particle data. This can be achieved by integrating both parts—simulation and rendering—of this visualization problem on the GPU.

To simulate particles on a GPU you can use stateless or state-preserving PS. Stateless PS require a particle's data to be computed from its birth to its death by a closed form function which is defined only by a set of start values and the current time. State-preserving PS allows using numerical, iterative integration methods to compute

the particle data from previous values and a changing environmental description (e.g., moving collider objects). Both simulation methods have their areas of applications and should be chosen based on the requirements of the desired effect.

Stateless PS have been introduced on the first generation of programmable PC GPUs [NVIDIA01]) and are described in the next section. The state-preserving simulation has recently become possible on floating-point graphics hardware and is described in the following sections. This particle physics simulation on a GPU can be a flexible combination of a multitude of motion and position operations, e.g., gravity, local forces, and collision with primitive geometry shapes or texture-based height fields. Additionally, a parallel sorting algorithm can be used to perform a distance-based sorting of the particles for correct alpha-blended rendering (see page 127).

Stateless Particle Systems

Some PS have been implemented with vertex shaders (also called vertex programs) on programmable GPUs [NVIDIA2001]. These PS are, however, stateless, i.e., they do not store the current positions and other attributes of the particles. To determine a particle's position you need to find a closed form function for computing the current position only from initial values and the current time. As a consequence, such PS can hardly react to a dynamic environment.

Particles that are not meant to collide with the environment and that are only influenced by a global gravity acceleration $\mathbf{g}$ can be simulated quite easily with a simple function:

$$\mathbf{p} = \mathbf{p}_0 + \mathbf{v}_0 t + \tfrac{1}{2}\mathbf{g}t^2 \tag{2.4.1}$$

where $\mathbf{p}$ is the computed particle position, $\mathbf{p}_0$ the initial position, $\mathbf{v}_0$ the initial velocity, and t the current time. Adding simple collisions or forces with local influence, however, lead to much more complex functions.

Particle attributes besides velocity and position (e.g., the particle's orientation, size, and texture coordinates) have generally much simpler computation rules than the position. It is often sufficient to calculate them from a start value and a constant factor of change over time, which makes them ideal for a stateless simulation. This holds true even if the position is determined with the state-preserving simulation in the next section.

The strengths of the stateless PS make it ideal for simulating small and simple effects without influence from the local environment. In action video games these might be a weapon impact, splash, or the sparks of a collision. Larger effects that require interaction with the environment are less suitable for the technique.

Particle Simulation on Graphics Hardware

The following sections describe the algorithm of a state-preserving particle system on a GPU in detail. After a brief overview of the algorithm, the storage, and then the processing of particles is described.

Algorithm Overview

The state-preserving particle system stores the velocities and positions of all particles in textures. These textures are also render targets. In one rendering pass the texture with particle velocities is updated using the previous velocities as input. The update performs a one-time step of an iterative integration which applies acceleration forces and collision reactions. Another rendering pass updates the position textures in a similar way, using the just computed velocities for the position integration. Depending on the integration method, it is possible to skip the velocity update pass, and directly integrate the position from accelerations.

Optionally, the particle positions can be sorted depending on the viewer distance to avoid rendering artifacts later on. The sorting performs several additional rendering passes on textures that contain the particle distance and a reference to the particle itself.

Then the particle positions are transferred from the position texture to a vertex buffer. Finally this geometry data is rendered to the screen in a traditional way—as point sprites, primitive triangles, or quads.

These six basic steps of the algorithm are described in the following sections:

1. Process birth and death
2. Update velocities
3. Update positions
4. Sort for alpha blending (optional)
5. Transfer texture data to vertex data
6. Render particles

Particle Data Storage

The most important attributes of a particle are its position and velocity. The positions of all active particles are stored in a floating point texture with three color components that will be treated as x, y, and z coordinates. Each texture is conceptually treated as a one-dimensional array; texture coordinates representing the array index. The actual textures however, need to be two-dimensional due to the size restrictions of current hardware. The texture itself is also a render target, so it can be updated with the computed positions. By drawing a full-screen rectangle, the GPU is instructed to call a pixel shader once for each pixel in the render target. The pixel shader also needs to read the previous position values from the texture. As a texture cannot be used for reading and rendering at the same time, we use a pair of these textures and a double buffering technique to compute new data from the previous values (see Figure 2.4.1).

A pair of velocity textures can be created in the same way as the position textures. Due to their reduced precision requirements it is usually sufficient to store the velocity coordinates in a 16-bit floating point format. Depending on the integration algorithm there is no need to store the velocity explicitly (see page 126). If the velocity is not stored in textures, you need a third position texture, which basically forms a triple buffer.

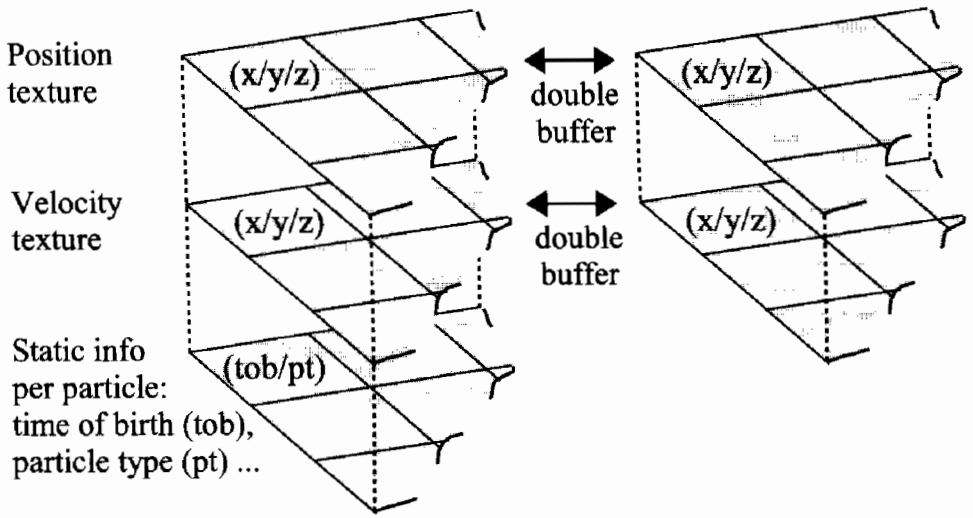

FIGURE 2.4.1 *Particle data storage in multiple textures.*

If other particle attributes (like orientation, size, color, and opacity) were to be simulated with the iterative integration method, they would need double buffer textures as well. However, since these attributes typically follow simple computation rules or are even static, we can take a simpler approach. An algorithm similar to the stateless particle system (see page 120) can be used to compute these values only from the relative age of the particle and a function description, e.g., initial and differential values or a set of keyframes. To be able to evaluate this function, we need to store two static values for each particle: its time of birth and a reference to a set of attribute parameters for its particle type. They are stored in a further texture, but the double buffer approach is not necessary.

We assume that the particles can be grouped by a particle type in order to minimize the amount of static attribute parameters that need to be uploaded during the final rendering of the particles. This particle type can either be directly coupled in a one-to-one relationship to the particle emitter or a group of emitters emits all particles of the same type.

The mass of a particle needs to be known to calculate accelerations from forces. Possible approaches are: treating all particles as having equal mass, uploading a mass value or function as particle-type parameters, or storing the mass of each particle in the static data texture described above.

To sum up: a single particle consists of data values spread between several textures, but placed at equal texture coordinates in all those textures. According to demand, particle-type parameters also allow the computation of further particle values.

Process Birth and Death

The particles in a system can either exist permanently or only for a limited time. A static number of permanently existing particles represents the simplest case for the

simulation, as it only requires uploading all initial particle data to the particle attributes textures once. As this case is rather rare, we assume a varying number of short-living particles for the rest of the discussion. The particle system must then process the birth of a new particle, i.e., its allocation and the death of a particle—its deallocation.

The birth of a particle requires associating new data with an available index in the attribute textures. Since allocation problems are serial by nature, this cannot be done efficiently with a data-parallel algorithm on the GPU. Therefore an available index is determined on the CPU via traditional fast-allocation schemes. The simplest allocation method uses a stack filled with all available indices. A more complex allocator uses a heap data structure that is optimized to always return the smallest available index. Its advantage is that if a particle system has a highly varying number of particles, the particles remain packed in the first portion of the system. Thus the following simulation and rendering steps only need to update that portion of data. After the index has been determined, the new particle's data is rendered as single pixel into the attribute textures. This initial particle data is determined on the CPU and can use complex algorithms, e.g., various probability distributions for initial starting positions and directions etc. (see [McAllister00]).

A particle's death is processed independently on the CPU and GPU. The CPU registers the death of a particle and adds the freed index to the allocator. The GPU does an extra pass over the particle data: the death of a particle is determined by the time of birth and the computed age. The dead particle's position is simply moved to invisible areas, e.g., infinity. As particles at the end of their lifetime usually fade out or fall out of visible areas anyway, the extra pass rarely really needs to be done. It is basically a clean-up step to increase rendering efficiency.

Update Velocities

The first part of the simulation updates the particles' velocity. The actual program code for the velocity simulation is a pixel shader which is executed for each pixel of the render target by rendering a screen-sized quad. The current render target is set to one of the double buffer velocity textures. The other texture of the double buffer is read by the pixel shader and contains the velocities from the previous time step. Other particle data, either from inside the attribute textures or as general constants, is set before the shader is executed.

There are several velocity operations that can be combined as desired (see [Sims90] and [McAllister00]): global forces (e.g., gravity, wind), local forces (attraction, repulsion), velocity dampening, and collision responses. For our GPU-based particle system these operations need to be parameterized via pixel shader constants. Their dynamic combination is a typical problem of real-time graphics. It is comparable to the problem of light sources and material combinations and can be solved in similar ways. Typical operation combinations are to be prepared in several variations beforehand. Other operations can be applied in separate passes, as all operations are completely independent.

Global forces, e.g., gravity, influence particles regardless of their position with a constant acceleration in a specific direction. The influence of local forces however depends on the particle's position which is read from the position texture. A magnet attracting or repelling a particle has a local acceleration toward a point. This force can fall off with the inverse square of the distance or it can stay constant up to a maximum distance (see [McAllister00]). A particle can also be accelerated toward its closest point on a line, leading to a vortexlike streaming.

A more complex local force can be extracted from a flow field texture. Since texture look-ups are very cheap on the GPU, it is quite efficient to map the particle position into a 2D or 3D texture containing flow velocity vectors. This sampled flow vector $\mathbf{v}_{\mathrm{fl}}$ can be used with Stoke's law of a drag force $\mathbf{F}_{\mathrm{d}}$ on a sphere:

$$\mathbf{F}_{\mathrm{d}} = \underbrace{6\pi\eta r}_{c}\left(\bar{\mathbf{v}} - \mathbf{v}_{\mathrm{fl}}\right) \tag{2.4.2}$$

where η is the flow viscosity, r the radius of the sphere (in our case the particle) and $\bar{\mathbf{v}}$ the particle's velocity. The constants can all be combined to a single constant c, for efficient computation and for simpler manual adjustment.

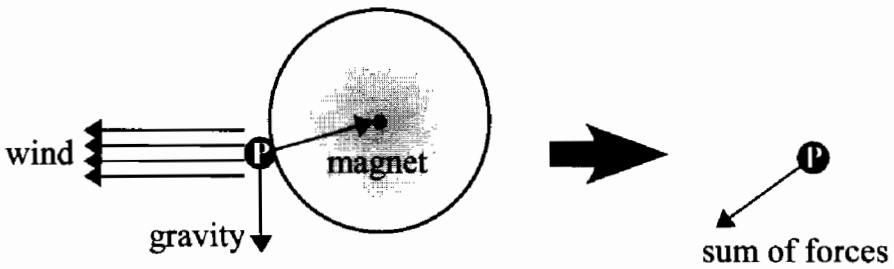

FIGURE 2.4.2 *Adding various forces to one force vector.*

Global and local forces are accumulated into a single force vector, as in the example in Figure 2.4.2. The acceleration can then be calculated with Newtonian physics:

$$\mathbf{a} = \frac{\mathbf{F}}{m} \tag{2.4.3}$$

where $\mathbf{a}$ is the acceleration vector, F the accumulated force and m the particle's mass. If all particles have unit mass, forces have the same value as accelerations and can be used without further computation.

The velocity is then updated from the acceleration with a simple Euler integration in the form:

$$\mathbf{v} = \bar{\mathbf{v}} + \mathbf{a} \cdot \Delta t \tag{2.4.4}$$

where $\mathbf{v}$ is the current velocity, $\bar{\mathbf{v}}$ the previous velocity and Δt the time step.

Another simple velocity operation is dampening, i.e., a scaling of the velocity vector, which imitates viscous materials or air resistance. This is basically a special case of equation 2.4.2 with a flow velocity of zero. The reverse operation, an un-dampening, can be used to imitate self-propelled objects, e.g., a bee swarm.

A more important operation is collision. Collisions with a plane or bounding spheres are simple and rather cheap. The real strength of collision on the GPU, however, is collision against texture-based height fields that are typically used to model terrain. By sampling the height field three times a normal can be computed which is then used for calculating the reflection vector. The normal can also be stored in the height field, basically making it a scaled normal map. Note that the height field can also be computed dynamically, by rendering the depth values of an object into a texture, similar to shadow mapping algorithms. With this approach it is also possible to perform collisions with complex geometries, that can be approximated with one or more height fields. This technique is described in detail in [Kolb04].

If a collision has been detected, the collision reaction, i.e., the velocity after the collision, has to be computed (see [Sims1990]). First the current velocity has to be split into a normal and a tangential component. If $\mathbf{n}$ is the normal of the collider at the collision point, these can be computed as:

$$\begin{aligned} \mathbf{v}_n &= (\mathbf{v}_{bc} \cdot \mathbf{n})\mathbf{v}_{bc} \\ \mathbf{v}_t &= \mathbf{v}_{bc} - \mathbf{v}_n \end{aligned} \tag{2.4.5}$$

where $\mathbf{v}_{bc}$ is the velocity computed so far, i.e., before the collision occurs, $\mathbf{v}_n$ is the normal component of the velocity, and $\mathbf{v}_t$ the tangential one. The velocity after the collision can now be computed with two further parameters describing material properties. Dynamic friction μ reduces the tangential component, and resilience ε scales the reflected normal component. The new velocity is computed as:

$$\mathbf{v} = (1-\mu)\mathbf{v}_t - \varepsilon \mathbf{v}_n \tag{2.4.6}$$

This default handling of the collision however has two problems which cause visual artifacts. The slow-down effect of the dynamic friction will lead to situations where the velocity is (very close to) zero. Since in our case the collision is processed after acceleration forces like gravity, this might lead to particles hanging in the air. They virtually seem to be attached to the side of a collider, e.g., at the equator of a sphere collider with respect to the global gravity. Therefore the friction slow-down should not be applied if the overall velocity is smaller than a given threshold.

The second problem is caused by particles getting caught inside a collider. A collider with sharp edges, e.g., a height field, or two colliders close to each other might push particles into a collider. This can be avoided by trying to push a caught particle out of the collider. Normally, the collision detection is done with the expected next particle position to avoid the particle from entering an object for the time of one integration step (see Figure 2.4.3a). The expected particle position $\mathbf{p}_{bc}$ is computed as:

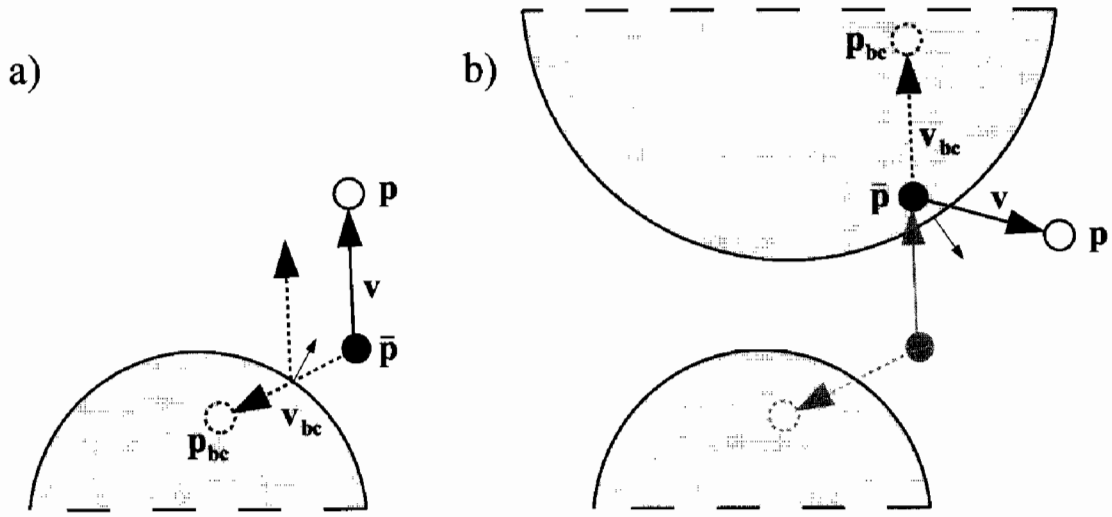

FIGURE 2.4.3 *Particle collision: a) Normal collision reaction before penetration b) Double collision with danger of the particle getting caught inside a collider.*

$$\mathbf{p}_{bc} = \bar{\mathbf{p}} + \mathbf{v}_{bc} \cdot \Delta t \tag{2.4.7}$$

where $\bar{\mathbf{p}}$ is the previous position. Doing the collision detection twice, once with the previous and once with the expected position, allows differentiating between particles that are about to collide and those having already penetrated (see Figure 2.4.3b). The latter can then be pushed out of the collider, either immediately or by applying the collision velocity without any slow-down. The direction of the shortest way out of the collider can be guessed from the normal component of the velocity:

$$\mathbf{v} = \begin{cases} \mathbf{v}_{bc} & \mathbf{v}_{bc} \cdot \mathbf{n} \geq 0 \\ \mathbf{v}_t - \mathbf{v}_n & \mathbf{v}_{bc} \cdot \mathbf{n} < 0 \end{cases} \tag{2.4.8}$$

Update Positions

The second part of the particle system simulation updates the position of all particles. Here we are going to discuss the possible integration methods in detail that have been mentioned earlier (see page 121). For the integration of large data sets on the GPU in real-time only simple integration algorithms can be used. Two good candidates for particle simulation are Euler and Verlet integration.

Euler integration has already been used in the previous section to integrate the velocity by using the acceleration. The computed velocity can be applied to all particles in just the same way. This leads to:

$$\mathbf{p} = \bar{\mathbf{p}} + \mathbf{v} \cdot \Delta t \tag{2.4.9}$$

where $\mathbf{p}$ is the current position and $\bar{\mathbf{p}}$ the previous position.

In some ways even simpler than Euler integration is Verlet integration (see [Verlet67]). Verlet integration for a particle system (see [Jakobsen01]) does not store the velocity explicitly. Instead, the velocity is implicitly deduced by comparing the previous position to the one before. The great advantage of this handling for the particle simulation is that it reduces memory consumption and removes the velocity update rendering pass.

If we assume the time step is constant, the combination of the velocity and position update rules from the Euler integration can be combined to a position update rule based only on the acceleration:

$$\mathbf{p} = 2\bar{\mathbf{p}} - \bar{\bar{\mathbf{p}}} + \mathbf{a} \cdot \Delta t^2 \tag{2.4.10}$$

where $\bar{\bar{\mathbf{p}}}$ is the position two time steps ago. Verlet integration handles simple (global or local) acceleration forces quite efficiently. However, complex velocity operations, like the collision reaction discussed above, require position manipulations to implicitly change the velocity in the following frames. Alternatively, collision can be handled more efficiently with position constraints that simply move the position out of the collider. Due to the deduction of velocity based on this constraint movement, an implicit reflection velocity is set. Other constraints can be used to limit the distance between a pair or group of particles, which is useful for particle simulation of cloth or hair (see [Jakobsen01] for more details).

Sort for Alpha Blending

If particles are alpha blended, a distance-based sorting should be applied or else particles in the front will not be blended correctly with particles behind them. This error might be intolerable depending on the size and amount of the partially transparent pixels of each particle. Due to the cost of sorting, first examine whether these particles can instead be rendered with a commutative blending function, e.g., additive or multiplicative.

A particle system on the GPU can be sorted quite efficiently with the parallel sorting algorithm "odd-even merge sort" [Batcher68]. It is independent of the data's sortedness, i.e., it always has a constant number of iterations for a data set of a given size. This is important as the parallel hardware execution makes it inefficient to check whether all data is already in sequence. This algorithm also guarantees that with each iteration the sortedness never decreases. If you assume a high frame-to-frame coherence of the sort order, this property allows you to distribute the whole sorting sequence over 20–50 frames. This will of course lead to some visual errors, especially when newly added particles need to be moved to their correct location in the sort order. However, these errors are usually hard to notice except when two particles with very different appearance swap their order. Game applications often know the maximum velocity with which the viewer and the particle objects can move, so assumptions about an acceptable number of frames to distribute the sorting over can be made.

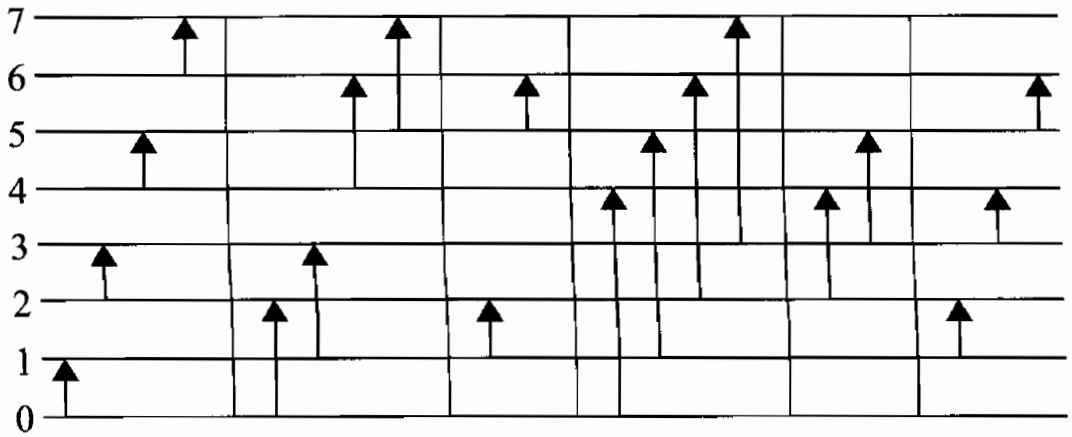

FIGURE 2.4.4 *Odd-even merge sorting network for eight values. y-axis: elements to sort, x-axis: sorting steps.*

The basic principle of odd-even merge sort is to divide the data into two halves, to sort these, and then to merge the two halves (for further details see [Lang03]). The algorithm is commonly written recursively and in a serial way, but a closer look at the resulting sorting network shows its parallel nature. Figure 2.4.4 shows the sorting network for eight values, an arrow marking a comparison pair. The first arrow indicates that element 0 is compared to element 1 and possibly swapped in case the order is not fulfilled. You can see that several consecutive comparisons are independent of each other. They can be grouped for parallel execution, which is indicated here by the vertical lines.

The Cg (see [Mark03]) code for odd-even merge sorting a one-dimensional texture contains two entry points:

```
float4 mergeSort1DEnd(float _Current : TEXCOORD0,
    uniform int _Step) : COLOR
{
  float currentSample = (float)texRECT(_SortData, (float2)_Current);
  float direction = (fmod(_Current / _Step, 2.0) < 1.0 ? 1.0 : -1.0);
  float otherSample = (float)texRECT(_SortData,
    (float2)(_Current + direction * _Step));
  if (direction >= 0)
    return max(currentSample, otherSample);
  else
    return min(currentSample, otherSample);
}

float4 mergeSort1DRecursion(float _Current : TEXCOORD0,
    uniform int _Step, uniform int _Count) : COLOR
{
  float currentSample = (float)texRECT(_SortData, (float2)_Current);
  int modulus = fmod(_Current / _Step, (float)_Count);
  if (modulus >= 1 && modulus < _Count - 1)
  {
```

```
    if (fmod((float)modulus, 2.0) > 1.0)
      return max(currentSample,
        (float)texRECT(_SortData, (float2)(_Current + _Step)));
    else
      return min(currentSample,
        (float)texRECT(_SortData, (float2)(_Current - _Step)));
  }
  else
    return currentSample;
}
```

This Cg code can be ported to Microsoft HLSL (see [Microsoft02]) by simply mapping the `texRECT` instruction onto a `tex2D` instruction. The code is slightly simplified for readability and sorts only a one dimensional texture. To enhance the shader for sorting two dimensional textures the texel index needs to be split into u and v coordinates before look-up. The sorting has two alternative sort shaders, a "recursion" and an "end" step. The pseudo code to trigger the sort passes on the CPU is shown here:

```
MergeSort(int _Count) :
  if (_Count > 1)
    MergeSort(_Count / 2)
    Merge(_Count, 1)

Merge(int _Count, int _Step) :
  if (_Count > 2)
    Merge(_Count / 2, _Step * 2)
    Render with mergeSortRecursion shader
  else
    Render with mergeSortEnd shader
```

The sorting requires $\frac{1}{2}\log_2^2 n + \frac{1}{2}\log_2 n$ passes, where n is the number of elements to sort. For a 1024×1024 texture this leads to 210 rendering passes. Just like in the particle simulation, they always render the full texture into a render target using a double buffer. As mentioned earlier, running all 210 passes each frame is far too expensive on current hardware, but spreading the whole sorting sequence over 50 frames, i.e., 1–2 seconds, reduces the workload to only four passes each frame, which results in acceptable performance.

This general sorting algorithm is applied to the particle simulation in the following way: the sorting data textures contain the particle-viewer distance and the index of the particle. The distance in this texture is updated after the position simulation. After sorting, the rendering step (see the next two sections) looks up the particle attributes via the index in this texture. Note that this indirection will cause a rather random access to the texture and can therefore be quite cache unfriendly.

Transfer Texture Data to Vertex Data

The copying of position data from a texture to vertex data is a hardware feature that is only just coming up in PC GPUs. Currently there are two approaches to this problem:

DirectX and OpenGL offer vertex textures with the vertex shader (VS) version 3.0 model (see [Microsoft02]) and the `ARB_vertex_shader` extension (see [OpenGL03]). To render the particles with vertex textures, one large static vertex buffer is necessary, that is, filled with texture coordinates of all pixels in the particle position texture. When this buffer is rendered the vertex shader uses these texture coordinates to read the particle position from the texture (see Figure 2.4.5). Currently reading from a vertex texture has quite a high latency, i.e., the time between sending of a texture read command in the shader until that data is available is rather high. Fortunately the shader can process other arithmetic operations that do not depend on the texture read while waiting for its result. The particle system vertex shader contains many operations independent of the particle position (see page 131), and is able to hide the latency of the texture read.

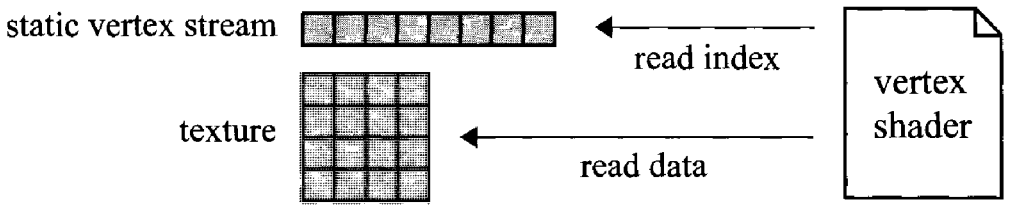

FIGURE 2.4.5 *Accessing a texture from the vertex shader to render particles.*

The alternative solution to vertex textures are "über-buffers" (also called super buffers; see [Mace04]) which basically are a data-agnostic storage of vertex or pixel data in a buffer. This concept is available in hardware since the first generation of floating-point GPUs, but up to now, it is only supported by the OpenGL API. The current implementation uses OpenGL with the `EXT_pixel_buffer_object` extension (see [NVIDIA04]), which offers accelerated asynchronous copying of data inside the GPU memory. Since the data can be copied from pixel to vertex memory (as illustrated in Figure 2.4.6), this is a basic implementation of the über-buffer concept.

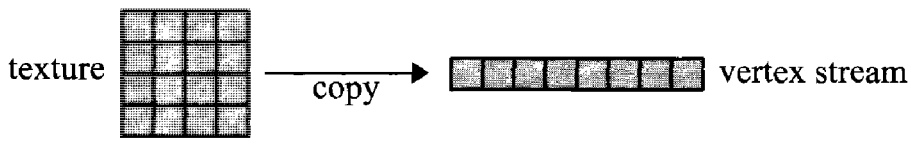

FIGURE 2.4.6 *Transferring pixel to vertex data with the über-buffer concept.*

The particles can be rendered as point sprites, triangles, or quads (see page 131). If vertex textures are used for the data transfer, it does not matter which type of base geometry is used. If multiple vertices per particle are needed, the static vertex buffer can simply contain the texture coordinates replicated three or four times for each particle. Alternatively, the so-called "vertex stream frequency" is introduced on hardware

supporting the VS3.0 version. This feature allows reducing the update frequency of vertex input data to a vertex shader. Basically, one entry in a vertex stream can be used for a range of several consecutive vertices, whereas other vertex streams change at a different rate. So, for the particle data, the vertex buffer does not need to contain replicated data, and instead uses an update frequency of three to render triangles or four to render quads.

On hardware that does not support vertex textures or the vertex stream frequency feature, using the über-buffer concept for data transferal requires manual replication of the particle position before rendering. In a further rendering step, the positions need to be rendered three or four times into pixels lying next to each other in a texture. To avoid this overhead, the current implementation uses point sprites which need only one vertex per particle.

Render Particles

Finally, the transferred vertex positions are used to render primitives to the frame buffer in a traditional way. For the reasons mentioned in the previous section and in order to reduce the workload of the vertex unit, particles are currently rendered as point sprites. Compared to rendering single triangles or quads this reduces the number of vertices to a third or a quarter. The disadvantage though is that particles are always axis-aligned and do not have a 2D rotation about the screen space z axis. To overcome this limitation, a 2D rotation is applied to the texture coordinates inside the pixel shader. The decision about using point sprites or generating triangle/quad geometry is made depending on the distribution of workload between vertex and pixel units of a particular application.

During the rendering, other attributes (e.g., color and size) are computed inside the vertex shader from the particle-type parameters (see page 121). Some of them take into account the relative age of the particle or a pseudorandom function.

The current implementation uses the following computation rules for these particle attributes: the size of a particle is random within a range that is defined by the particle type. The initial orientation and a rotation velocity (2D in screen space) are also determined randomly within a defined type-based range.

Based on the relative age of the particle the color and opacity are interpolated from four keyframes. These keyframes are defined for each particle type and need not be equidistant. They define three linear function segments that are first converted into a form for efficient GPU evaluation and then uploaded with the particle type data.

It is not possible to switch textures on a per-particle basis while rendering. Thus it is necessary to combine different textures into tiles of a larger 2D texture. Each particle then modifies the texture coordinates appropriately. If point sprites are used, the texture coordinates will be generated automatically by the rasterizer in the range [0–1] × [0–1]. The sub-texture selection then needs to be done in the pixel shader. Fortunately, the texture coordinate transformation for the 2D rotation we described above

will do the sub-texture selection basically for free, as a transformation by a 2×2 or by a 3×2 matrix both need two vector instructions.

Conclusion

Of course processors are never fast enough, so the implementation on the recent generation of GPUs simulates and renders a 1024×1024 texture of particles in real-time only with few effects and without sorting. Sharing the GPU with other techniques and using the full feature set currently allows up to 512×512 particles. The performance is expected to improve significantly with the next generation of PC graphics hardware.

This article has shown how to design and implement a state-preserving physical particle simulation on current programmable graphics hardware. The simulation can use either Euler or Verlet integration to update the particle positions. Other particle attributes are simulated with less complex algorithms. Without permanent storage they are always evaluated on demand. Additionally, an efficient parallel sorting algorithm for particles has been described.

The main strength of GPU-based particle systems is the low cost of individual operations on the data set. Once a basic algorithm is implemented, endless ideas for manipulating velocity and position come up and are easily implemented in higher level shading languages.

How applicable to upcoming video game console hardware the introduced state-preserving particle simulation is, remains to be seen. But the trend toward increased multi-processing hardware is a good indication that the parallel computation of particle systems will grow in importance.

Acknowledgments

Thanks to Wolfgang Engel, Nicolas Thibieroz, my colleagues at Massive Development, especially Ingo Frick, Dr. Christoph Luerig, and Mark Novozhilov, and Prof. Andreas Kolb from the University of Siegen for the fruitful discussions and support in writing this article. Also thanks very much to Sieggi Fleder for never giving up on my Germanic English. Furthermore, much appreciation is owed to Matthias Wloka and his colleagues at NVIDIA, who helped the demo implementation to stay *on* the cutting edge of technology.

References

[Batcher68] Batcher, Kenneth E., "Sorting Networks and their Applications," Spring Joint Computer Conference, AFIPS Proceedings 1968.

[Burg00] van der Burg, John, "Building an Advanced Particle System," *Game Developer Magazine* (03/2000).

[Jakobsen01] Jakobsen, Thomas, "Advanced Character Physics," GDC Proceedings 2001.

[Kolb04] Kolb, Andreas; Latta, Lutz; Rezk-Salama, Christof, "Hardware-based Simulation and Collision Detection for Large Particle Systems," Graphics Hardware Proceedings 2004.

[Lang03] Lang, Hans W., "Odd-Even Merge Sort," available online at *http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/oemen.htm*, 2003.

[Mace04] Mace, Rob, "OpenGL ARB Superbuffers," available online at *http://www.ati.com/developer/gdc/SuperBuffers.pdf*, 2004.

[Mark03] Mark, William R.; Glanville, R. Steven; Akeley, Kurt; Kilgard, Mark J., "Cg: A System for Programming Graphics Hardware in a C-like Language," SIGGRAPH Proceedings 2003.

[McAllister00] McAllister, David K., "The Design of an API for Particle Systems," Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill, 2000.

[Microsoft02] Microsoft Corporation, "DirectX9 SDK," available online at *http://msdn.microsoft.com/directx/*, 2002–2004.

[NVIDIA01] NVIDIA Corporation, "NVIDIA SDK," available online at *http://developer.nvidia.com/*, 2001–2004.

[NVIDIA04] NVIDIA Corporation, "OpenGL Extension `EXT_pixel_buffer_object`," available online at *http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_pixel_buffer_object.txt*, 2004.

[OpenGL03] OpenGL ARB, "OpenGL Extension `ARB_vertex_shader`," available online at *http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_shader.txt*, 2003.

[Reeves83] Reeves, William T., "Particle Systems—Technique for Modeling a Class of Fuzzy Objects," SIGGRAPH Proceedings 1983.

[Sims90] Sims, Karl, "Particle Animation and Rendering Using Data Parallel Computation," SIGGRAPH Proceedings 1990.

[Verlet67] Verlet, Loup, "Computer Experiments on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules," *Physical Review* (159/1967).