Comparison Between Particle Rendering Techniques in DirectX 11

Jämförelse Mellan Tekniker för Partikelrendering i DirectX 11

Robin Andersson
Simon Johansson
Comparison Between Particle Rendering Techniques in DirectX 11

Robin Andersson
paul.robin.andersson@gmail.com
Game Development Program
Computer Science and Media Technology
Malmö University
Sweden

Simon Johansson
simon.johansson@fjelie.com
Game Development Program
Computer Science and Media Technology
Malmö University
Sweden

Abstract—Comparison between GPU and CPU particle systems. Computer games have used particle systems for visual effects such as weather, smoke, fog, etc and it’s relatively low cost to implement. There are a mixture of different techniques used to render particle systems, usually CPU heavy implementations or a GPU based one.

In this study we developed two rendering techniques for a particle system, instancing and stream-out, and compare their performance. As almost as expected we saw that the stream-out technique that is GPU based outperformed instancing when we tested using non-intractable particle systems. We have furthered discussed both advantages and disadvantages in other scenarios.

Keywords—DirectX, stream output, instancing, particle system, comparison, benchmark

I. INTRODUCTION

Particle systems are widely used in digital games. It has been used since 1962 in the game “Spacewar” as the effect for explosions but was not referred as a particle system until 1983 when the term “particle system” first introduced and was used in the movie “Star Trek II: The Wrath of Kahn” for the genesis effect [1].

Figure 1. 1983 Star Trek 2

Visual effects such as rain, snow, smoke, fog, etc are some of the effects a particle system can generate [1]. It’s relatively low cost to implement and has a high visual impact on digital games. There are different techniques for rendering a particle system. Some are computed entirely on the Graphical Processing Unit (GPU), others use both GPU and Computer Processing Unit (CPU) to compute.

This thesis serves as a general benchmark comparison between different techniques for rendering particle systems. The evaluation could decide whether or not to use a special rendering technique. It should be noted that the problem is much more wider than just particle systems. Rendering similar geometric objects can occur with trees, foliage etc [2].

A. Particle System

Particle system is defined by an emitter and a set of particles. An emitter is a point or an area in space that acts as a generator of particles.

A particle can be a single point, line, triangle, quad or a model although you should keep the number of vertices at a minimum. This is to reduce the cost of rendering each particle and lets you increase the number of particles in the scene. Characteristics of a particle can be a position, velocity, direction, color, etc. By manipulating these values, the particle system will generate different effects. Giving the particles a rising effect with red color that fades to gray the particle system will generate smoke. By giving them a falling effect the particle system will generate snow or rain [3] effects.

Collision detection may be implemented in a particle system but it will increase the performance cost depending on the collision detection technique as demonstrated by Kolb et al. in their study with hardware-based simulation and collision detection [4]. This is to get a more realistic environmental particle effect by example excluding the particles from entering a house or creating a waterfall effect as Wong simulated in his thesis [5].

The update cycle for the particles can be implemented differently ranging between computational complexity of $O(n)$ to $O(n^2)$ depending on how you implement your PS. Collision detection and other expensive computation will drastically increase your time complexity. The least intensive computation is easily implemented by just moving the particles and no interaction with the surrounding objects. These are referred to as Non-interacting particle systems (NIPS) as described by Knowles [1], and has a time complexity of $O(n)$.

for every particle i do
  move i
end for

B. Billboarding

Billboarding is an easy and cheap way to make a flat polygon to always show its face towards the camera. It works by orienting a textured polygon on the view direction and
the polygon is a billboard. As the view changes, the polygon is oriented to face the view of the camera. This technique is much cheaper in performance because a quad has less vertices to process than a 3D model [3].

C. Order-independent transparency

![Macroscopic snow particles in Tom Clancy’s The Division](image)

Alpha blending is a technique that makes an object or a part of the object transparent. Figure 2 is an example on how it looks when implemented correctly. In addition of red, green and blue which are the primary color channels a forth can be added which is known as the alpha channel that represents the level of transparency.

Transparent images can cause problems in rendering, specifically in the rasterizer stage.

Order-independent transparency (OIT) is a technique used for rendering transparency in a 3D scene. OIT works by sorting alpha blended geometry by depth and something that Carpenter has delved deeper in with explanation and why it’s so significant [7]. Although this adds more complexity when rendering the scene, it’s necessary if you don’t want to get unwanted artifact in the rendered image of the scene.

D. Pipeline Overview

The graphical pipeline consists of several stages illustrated in figure 3. The blue rectangles represents fixed-function stages that cannot be modified programmatically. Green represents programmable stages. Not all programmable stages need to be initialized. Arrows represents the flow of data from each stage. The yellow block on the right side represents the GPU’s memory storage that both the CPU or GPU can read and write to. The green block represents multiple stages that includes hull shader, Tessellator, and domain shader stage which like the green stages are also programmable. The orange stage will not be utilized for the artifact.

![DirectX 11 rendering pipeline](image)

1) Input-Assembler: Input-Assembler assembles index and vertex data specified by the program for the graphics hardware. The data is then sent to the Vertex Shader to be processed.

2) Vertex Shader: Vertex Shader (VS) is a programmable stage, it processes vertices. It operates per vertex and get a vertex as input and outputs a vertex. VS usually perform transformations from model-space\(^1\) to clip-space\(^2\) in preparation for rendering the 2D representation of the world. Our particles will have a VS that utilizes a billboard effect that turns each quad to face the camera eliminating the effect of paperthin and disappearing snowflakes.

3) Geometry Shader: Geometry Shader (GS) is an optional programmable stage that has the ability to manipulate and increase the number of vertices given from the VS. E.g. possibility to extend the geometry with more vertices. Normally the GS sends the vertices to the rasterizer stage for rendering but GS also has the option to save the information in GPU memory by utilizing the stream output stage.

\(^1\)The position for all the vertices before any transformation for rendering

\(^2\)Clip space is the space entered after the projection matrix was applied.
4) **Razterizer Stage:** Razterizer Stage is a fixed-function stage. Clip primitives into the view frustum, primitive culling and interpolate the faces of the vertices and sends the interpolated values to the pixel shader.

5) **Pixel Shader:** Pixel Shader is a programmable stage, it operates per pixel. It outputs a color represented as red, green, blue, and alpha that the pixel will show on the screen. Texture sampling is one of many operations the pixel shader support.

**E. Instancing**

Instancing lets you render multiple copies of the same model at once. The instancing draw method essentially use a single draw call to submit a model to the pipeline. This model is then ‘instanced’ multiple times, but with a subset of the vertex data specified at a per-instance level, instead of a per-vertex level [8]. Each instance have differentiating parameters such as color and position.

**F. Stream-out**

Stream-out is a technique that is heavily GPU dependent, differentiating itself by reusing many of the pipeline stages [9]. It’s called Stream-out because it utilize the stream output stage. In short the technique uses the pipeline twice in an update cycle. The first pass updates the geometry and saves them in GPU memory using the stream output stage. We disable both the depth stencil and the pixel shader at this stage because there is nothing to render. The second pass renders the geometry.

Because the stream-out technique stores all its information on the GPU it becomes harder when implementing other techniques such as collision detection with objects in the scene that are stored on the CPU. Needing to send information from the CPU to the GPU may reduces the performance and defeat the purpose of using the stream-out technique.

**G. Related Work**

Previous research shows comparisons with particle systems in terms of different collision detection algorithms. Knowles made the tests with the use of old hardware and outdated software [1]. Our study will focus more on rendering techniques rather than collision detection algorithms.

Rendering million instances of a model has been conducted earlier in studies by Fan at el. with grass blades [2] or when simulating a waterfall by Wong [5]. They were both conducted and based purely on the GPU and shows performance limitations that would be lifted if they were operated on the GPU instead. A solution might be a modification for what Tariq and Bavoil shows by simulating realistic hair with a GPU solution including the stream output stage [10].

To fairly compare these two techniques we will use methods that Olofsson utilized when comparing their data so that we will be able to more optimally read and discuss our results [11].

**H. Research Question**

**RQ1:** What are the principal advantages and disadvantages of the Stream-out and Instancing techniques.

**RQ2:** How do they scale performance-wise with the number of particles.

**I. Contribution**

This article will only compare two different techniques by using them in a particle engine for rendering but they can be applied to other scenarios with foliage, grass, etc. Thus game developers can gain insight on when to choose the optimal rendering technique to render many similar geometric objects.

Source code is available at github [12].

**II. Method**

Our Method will follow A Design Science Research Methodology (DSRM) [13]. The study concludes that there are 6 optional steps in DSRM: problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation, and communication. The model is well known and serves as a foundation for creating a successful artifact for evaluation in our study.

![DSRM Process Model](image)

**A. Method description**

The underlying motivation is that there is insufficient data on performance comparisons between rendering techniques that uses particle systems (PS). The data you are able to find is either another type of PS or was gathered on old hardware. A company named Valve has conducted surveys on what GPU’s their users are running, many of which use outdated GPU’s [14] but shows a steady growth of faster hardware. Our objective is to develop an artifact that will benchmark PS rendering techniques. We will run simulations that will gather data. The data will be compared and analysed giving us a structure that other game developers can learn from in making their own particle system.

Both instancing and stream-out were chosen because we want to measure techniques that take advantage of different hardware, GPU and CPU. While at the same time can be used to render a particle system.

We will be creating our artifact using DirectX 11 which is an application programming interface (API) created by Microsoft for communication between the application and the graphics card. The different methods that will be evaluated are techniques called instancing and stream-out that are present when using DirectX 11. The collected data will be represented.
with the number of particles in the world and the time each frame takes to render the scene, represented as milliseconds. When simulating we want to get the same expected result each time we start over the test. With instancing we will be utilizing a constant seed number that we feed our random generator to get a scene that will always generate the same result. The Stream-out depend on a texture as the seed to get the same random numbers for each simulation.

The results we gathered will be represented with graphs. The metrics that we will be using are measurement of time between each frame and the number of particles in the scene. Even though one technique might be better we will discuss other limitations for the method which might be the biggest factor when choosing a technique.

B. Limitations

The scene will contain prop models that will be rendered with both PS. This is to make the comparison between the PS fair and will not impact the data that will be gathered. Likewise we will not be implementing difficult methods that will drastically reduce the performance such as collision detection.

Other concerning factors are that the PS techniques will have to be implemented as closely the same as possible. That is taking advantage of pipeline stages that are useful for both of the PS techniques. E.g. each particle represent a point and both particle systems will utilize the geometry shader to expand the point into a quad.

When the computer is gathering data it is of importance that the benchmarking program is the only application running on the computer. Otherwise the data will be invalid and cannot be compared equally with other tests.

The environment such as the camera view and resolution needs to be identical in all test-cases in order to fairly test and compare the techniques.

III. IMPLEMENTATION

We began working on a basic framework that was built on top of DirectX 11. This framework is the base for both our test-cases so that they have a fair comparison later in the testing phase.

Our goal was to implement a particle system that simulated somewhat of a realistic snow effect without external forces such as vector fields. This effect was achieved with multiple sinus curves at different intervals. The sinus curve was used to create the floating effect of snow. Each particle had an offset on the sinus curve to randomize the floating effect, eliminating synchronization when floating downwards.

After each iteration we improved the framework to support new features that is required for our test-cases.

The first particle system was very basic. We used Rastertek Particle System as a template, then later modifying it to our needs [15]. Each particle was its own object. This is a worst case scenario because each particle will call its own draw call from the CPU to the GPU. This will take a long time because submitting an object to the GPU takes longer than the GPU transforming and rendering the object. Especially if there are thousands of particles in the scene. Our first goal was to use the instancing technique, thus we improved our basic particle system to support it.

A. Properties of a particle

In our test case the particles contains the value of

- Initial spawn position
- current age of the particle
- offset of the particle

Building the Update cycle for both particle system is crucial with regards of reducing unnecessary code structure that increases frame time. These properties are needed for the effect of falling snow.

B. Instancing Particle System

We used Rasterteks article on Instancing [16] to implement instancing in our particle system. This led to just initializing
one particle that would later be 'instanced' several times. Instead of representing a particle with a quad we minimized it to a point. The point would then be expanded into a quad in the Geometry Shader before rendering. The CPU has control over the whole updating phase, including emitting, killing and updating particles which makes this very dynamic, in terms of overall CPU and GPU computations.

After some iterations on the stream-out programming, we quickly saw how much more efficient the GPU was able to utilize 100 % of its processor than the CPU could with only one thread. To maximize the performance we wanted to utilize all available threads on the CPU. That is two threads per core due to Intel Hyper-threading Technology. This became mostly an afterthought and was not implemented optimally with the use of a threadpool, but still gave massively better performance.

A problem that occurred during implementation with our particle system was that transparency on the particles miss-matched and created unwanted artifacts on the rendered scene. The most basic way to solve the problem is to sort all the particles by depth, this is called ordered-independent transparency. The DirectX API has support for this issue, but major changes had to be made to support it. As of any sorting algorithm we realized this would drastically reduce our performance, so we solved it by minimizing the particles to the size that they are no longer able to create unwanted artifacts.

C. Stream-out Particle System

We took advantage of the book written by Luna [9] when implementing our Stream-out particle system. With the help of the PS stream-out section of the book we implemented a foreground PS which we later modified to suit our needs. During the initialising phase, it creates one particle which becomes our emitter and is later sent to the GPU, releasing it from the CPU which makes it no longer modifiable. The GPU then is able to create, store, and draw all the particles (represented as points). The first pass emits new particles and updates ‘living’ particles. The second pass expands the particle point into a quad which later gets rendered to the screen.

D. Benchmark

Next iteration we began working on a tool that would help us run all the tests and gather data. Olofsson [11] created a tool that sampled frame time by calculating the elapsed time with timestamps and used this to further develop our benchmarking tool.

There is 30 rounds of test, each round gathers data on both particle systems. Each round have a defined value of particles set to both particle systems. Each round has two phases, warmup phase which lets the particle system emit particles for 30 seconds making it ready to move on to phase two (sampling) which samples data for 30 seconds. Data that are sampled are frame time, how long it takes to render a frame, and number of particles. After each round we estimate the average frame time for the chosen particle system.

IV. Performance Tests

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i7-2600K 4.40GHz (4 cores/8 threads)</td>
</tr>
<tr>
<td>GPU1</td>
<td>NVIDIA GeForce GTX 1080 (Driver Version 378.78), Released 2016</td>
</tr>
<tr>
<td>GPU2</td>
<td>AMD Radeon HD 6950 (Driver Version 17.5.1), Released 2010</td>
</tr>
<tr>
<td>RAM</td>
<td>Corsair Vengeance DDR3 1600MHz 8GB</td>
</tr>
<tr>
<td>OS</td>
<td>Microsoft Windows 10 Build 10.0.10586</td>
</tr>
<tr>
<td>DirectX</td>
<td>Microsoft DirectX 11.3</td>
</tr>
<tr>
<td>Development</td>
<td>Microsoft Visual Studio Community 2013 with Update 3</td>
</tr>
</tbody>
</table>

Our test will run with the computer specifications listed in table I. Vertical synchronization will be disabled because we don’t want to be locked in at the same frame rate as the monitors refresh rate. E.g. many monitors have 60 Hz refresh rate, this would lock us in at maximum 60 fps.

We want to push our particle systems to the limit going past the reasonable frame time of 16 ms. Our benchmarks will therefor range between 0 to 500k particles with increments of 20k particles between each sampling state. All test cases will not be able to produce the limit of 500k particles and so we will terminate the test that go past a certain point.

To get a more accurate value on each increment we will take an average for the time it takes to render the scene within an interval of 30 seconds. They will be executed in window mode with a resolution of 1920x1080 (aspect ratio of 16:9).

A. Instancing

Figure 6 shows results while using instancing PS with 1, 2, 4 & 8 threads. If they exceeded over 24 ms the test were 3Also known as v-sync. Frame-rate is locked to the monitors refresh-rate.

Figure 6. These results were taken with the 1080 graphic card. Lower is better.

Table I

Computer Specifications
terminated. All tests shows that each test grow in a linear fashion. This is expected given that the particles are not included in any technique that increases exponentially. When the thread amount exceeds 1 thread it takes more rendering time to initialize threads. Thus the rendering time is higher at lower number of particles.

At 40k particles and lower it’s most efficient to use 1 thread. Between 40k and 80k particles 2 threads is the most efficient. Greater than 80k particles 8 threads is the most efficient.

B. Stream-out & multiple GPU’s

Figure 7 shows instancing (8 threads) on two different GPU’s. We used the Instancing technique with 8 threads because it is more relevant with its lower frame time at higher number of particles when comparing it with stream-out. The gap between the two instancing tests are small. Even if instancing is mostly only on CPU, the cost to send the information to the GPU increases which is the cause for the small gap. Stream-out on the other hand has a larger gap. They grow like instancing, linear, but with different angles meaning that when rendering particles past our test results the distance between the two GPU will grow and be even greater.

V. ANALYSIS & DISCUSSION

A. Instancing

Instancing utilize both CPU & GPU making it more dynamic e.g. the particle data isn’t only bound to the GPU such as stream-out. In this case it becomes easier to interact with other objects or entities in the game world. Although the trade of is that it’s slower performance wise and is something that Sheng et al. [2] faced when trying to render millions of grass blades with collision detection. The CPU has to transfer all the information to the GPU, thus the GPU has to wait until all the information is sent to render a frame.

In our test case we used a CPU with 4 cores (8 threads with hyper-threading technology) and implemented the instancing PS to support up to all of the 8 threads. Thus our code structure may not be efficient for CPU’s that has less than 8 threads. In figure 7 the instancing slope of the line seemed to increase at a higher rate during 0 to 160k particles for the older GPU. Afterwards it settles and becomes parallel with the newer GPU. This breakpoint may be due to the CPU taking so long that even the older GPU becomes idle. The gap after 160k particles can be due to the GPU architecture being different, seeing as we are using one AMD GPU and one NVIDIA GPU, or that the drivers for AMD has not been updated at some time.

All threads has to be initialized & synchronized. This has a big impact on performance at lower particle count. The threading was not implemented with the use of a treadpool and had to be initialized for every update which may be the results of the big gap shown in figure 6. If you don’t need to render many particles it’s not necessary to utilize multiple threads as seen in figure 6. It may even be better to just have one thread depending on particle count.

B. Stream-out

<table>
<thead>
<tr>
<th>Particles</th>
<th>NVIDIA GeForce GTX 1080 (ms)</th>
<th>AMD Radeon HD 6950 (ms)</th>
<th>Δ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100k</td>
<td>1.03</td>
<td>2.54</td>
<td>1.51</td>
</tr>
<tr>
<td>200k</td>
<td>1.87</td>
<td>3.79</td>
<td>1.92</td>
</tr>
<tr>
<td>300k</td>
<td>2.70</td>
<td>4.93</td>
<td>2.23</td>
</tr>
<tr>
<td>400k</td>
<td>3.53</td>
<td>6.05</td>
<td>2.52</td>
</tr>
<tr>
<td>500k</td>
<td>4.34</td>
<td>7.32</td>
<td>2.98</td>
</tr>
</tbody>
</table>

Table II shows rendering time for each graphics card represented in ms of an interval of 100k particles.

Stream-out is more efficient in rendering as seen in figure 7 due to it taking advantage of the graphical pipeline and splitting up the workload into two phases. Although this is faster performance wise the CPU can’t do any computation on the particle data that is bound to the GPU. E.g. the CPU has no control over the particle system at run-time.

There is a practical limit on how many particles the graphical pipeline can produce at a certain moment. With the gathered data from figure 7 shown in table II the GPU NVIDIA GeForce GTX 1080, theoretically would have approx 1 888k particles at real-time. The same can be applied to GPU AMD Radeon HD 6950 giving 1 280k particles at real-time. This shows a great difference between the GPU’s at higher particle count. The particle update cycle is fairly lightweight for a GPU to calculate. So it becomes the rendering part of the particles that takes up time.

As said earlier the stream-out is only GPU dependent making the differences in rendering time huge when switching to an older GPU. The difference by changing GPU’s for instancing did not make as big of a difference in the gap given that the instancing technique uses both GPU & CPU.

C. Further Discussion

We decided to use NIPS in our PS because we wanted no interaction with both our techniques giving both a fair chance. Given our results we can see that they grow in a linear fashion
in every test and is something that Knowles also came to a conclusion with his results.

A problem with storing all information on the CPU is that you have to transfer the information later onto the GPU for rendering. With millions of particles in the scene it is more efficient to store and update the particles directly on the GPU. A downside with storing the particles on the GPU is that if you want to make more complex simulations like collision detection, then you will have to transfer the objects to be able to handle the detections. This defeats the purpose of not having to push information from the CPU to the GPU.

At lower particle count there is almost no difference between the techniques and mostly favors Instancing because of it can operate both on the CPU & GPU. Setting a breakpoint for when Stream-out becomes more enticing to use will differ for each developers purpose depending on how much workload needs to be calculated and/or if they need to interact with other game objects or entities.

Current technology advances in a rapid rate predicted by Moore and may increase the number of cores in the future which will drastically improve multi-threading performance.

The GPU’s that are used in the test cases are extremely new and old which includes a broad range of other GPU’s on the market in that spectrum. A Survey conducted by Steam on which computer specifications users are playing their games on shows that a majority of users have older hardware. Because both techniques differ from where most of their computation stems from, switching out the GPU or CPU may drastically alter the results and reduce or increase the gap at the higher particle count.

VI. FURTHER RESEARCH

Implementation of a particle system that utilizes compute shader and measure the performance would be ideal with providing the developer with more choice opportunity. This technique is GPU based but does not use the API the same way as stream-out. The instancing particle system could also be improved with the use of threadpool instead of single threads.

As we discussed when Knowles research uses outdated hardware, soon at some point in time our hardware will be also become outdated. Thus newer benchmarks would be ideal to run in the future.

Many games on the market uses DirectX 11 but is not the newest graphical API that is available. Changing API to Microsoft’s newest DirectX 12 or using the Vulcan API, which is developed by Khronos group, would be interesting to compare and see if there is a noticeable difference.

In our thesis we only saw how the graph changed when switching out the GPU. It would also be interesting to see how the curves behave if we changed only the CPU in the configuration and compare the results afterwards to see if the CPU gives more of a difference between the techniques than with the GPU.

VII. CONCLUSION

According to our results, with all information gathered and discussed we came to a conclusion that stream-out will always outperform instancing when it comes to NIPS. Beyond NIPS with low level numbers of particles and outside interact-able objects like collision detection or vector forces, using instancing seems like the best solution with account of simplicity of having every calculation on the CPU.

GLOSSARY

benchmark A program used to gather data and measure performance.. 1, 3–5, 7

hyper-threading Created by Intel that simulates 2 threads on each core e.g. 4 cores becomes 8 accessible threads.. 5, 6

prop models Models found in the scene that is static and does nothing but renders to the screen.. 4

real-time When a frame takes 16 ms or lower to render.. 6

screenshot A captured image of the screen that is currently rendered.. 4

threadpool A pool that stores and handles threads with tasks given from the program to be executed.. 5, 7

vertex A Point of the model that normally holds Position, Texture coordination, normals, etc.. 2, 3

REFERENCES


