Particle effect system for the needs of a modern video game using the GPU

Wojciech Zeler, Paweł Rohleder
Techland Sp. z o.o.
ul. Żółkiewskiego 3, 63-400 Ostrów Wlkp, Poland
http://techland.pl

Abstract.

A new system of creation and management of particle effects created for the needs of the future productions of Techland Co. Ltd. is presented. By a proper organisation of memory buffers it provides for maximum data density in the memory. This makes it possible to simplify the calculations and to use a smaller number of threads and less memory readings.

Key words: GPU, particles, FX, compute, shader, real time, graphics

1. Introduction

A particle effect is any physical phenomenon to replicate and visualize which it is necessary to simulate the behavior and interaction of a number of unrelated, so called, *particles*, i.e. individual entities subject to certain behaviors specified by the artist. Depending on the intended result, the particles can be e.g. sparks, dust particles, clouds of smoke, individual insects, flames, liquid streams, etc. (see Fig. 1 for an example).

In the discussed implementation it has been tried to fully use the benefits offered by today's graphics cards and game consoles. In order to operate, the equipment compatible with the Shader Model 5.0 [3] is necessary.

2. Review of competing solutions

Prior to the implementation, a review of competitive solutions was conducted. It helped us to create list of features that our new system must have. Among them there were:

- particle effects editor in the Unity engine,
- particle effects editor in the Unreal 4 engine,
- particle effects editor in the CryEngine.

The comparative analysis was made in the following aspects:

- editor shape and appearance, artist's work process with the editor – window appearance, integration with the rest of the editor;
- ways to control the particles, such as e.g. curves, gradients;

Fig. 1. Example of the particle effect with fire and smoke.

- additional options, such as e.g. sub-emissions (i.e. particle emission by other particles), and depth buffer collisions.

The comparison revealed that none of the existing systems had all the features we required. These features will be considered in the context of the methodology developed within the present paper.

3. Tasks of the system

The main tasks to be executed by every particle effect system are as follows.

New particle emission Each individual particle is created by a, so called, *emitter*. It defines in what way the particles created in it will act throughout their lifespan. The operation of emitting particles is quite complicated and should take into account, e.g. the necessity to emit more than ten thousand particles from one emitter during one animation frame.

Individual particle movement simulation Each particle has certain physical properties, such as instantaneous velocity, size, acceleration, and orientation. Each of those properties can be further modified by a set of modifiers freely set by the artist (in case of e.g. acceleration, it is the force of gravity).

Particle sorting Because of the translucency of most of the particles the order in which

they are painted on the scene becomes important. A lack of sorting would result in a lot of artifacts in the generated image.

Taking into account different types of particles Translucent particles are the most popular. However, the possibility of using particle effects at any stage of rendering of frame animation is necessary because it gives access to many potentially interesting effects, such as e.g. refraction.

Drawing particles This is the last stage of the system's operation. It is at this time that the particles are placed on the scene in a way visible to the user (up till now they were only the compressed structures in the graphic card's memory).

4. Algorithm

The algorithm was designed in such a way as to strive for storing the maximum density of data in the smallest area of memory. This makes it possible to keep the calculations simple and to use a smaller number of threads and make less memory readings.

The discussed implementation works in two stages.

In the first stage basic calculations for the continuous operation of the entire system are performed, i.e., calculations of state changes of all particles, emission of new particles and deleting the old ones.

The purpose of the second stage is to determine a subset of particles visible from every camera known to the game, and then to perform calculations associated with their presence and to prepare data needed for instantiating.

First stage The order of operations included in the first stage is shown in the chart in Fig. 2.

First, the information on the amount of particles in the main buffer is read. This information allows to determine the amount of particles at the start of the animation frame, as well as how many thread groups must be used to perform the simulation.

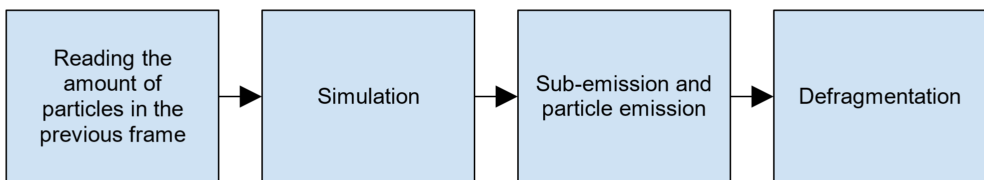

Fig. 2. Chart of operations of the stage one of the algorithm.

Let us introduce the following denotations:

P – set of all particles,

P_{prev} – number of particles in the previous frame (equal to the index of the last live particle in the set),

P_{curr} – number of particles in the current frame (at the beginning of the frame it is $P_{\mathrm{curr}} := P_{\mathrm{prev}}$),

Th_{opt} – optimal number of threads in one group. For contemporary GPUs it is most commonly $Th_{\mathrm{opt}} = 64$,

P_{prev} – number of particles in the previous frame (equal to the index of the last live particle in the set),

G_{sim} – number of thread groups needed to perform the simulation.

Therefore

$$G_{\mathrm{sim}} = \mathrm{ceil}\left(\frac{P_{\mathrm{prev}}}{Th_{\mathrm{opt}}}\right) . \tag{1}$$

During simulation, excluding velocity, position, acceleration, orientation and any other physical property change, particles may be assigned to the following sets:

set D containing particles whose lifetime came to an end in the current animation frame;

set S containing particles capable of emitting other particles.

The sets S and D are reset with the beginning of each consecutive animation frame.

The next step is the, so called, *sub-emission*. It is limited from above to a maximum volume, due to the easy to cause exponential increase in particle quantity visible on the screen (which may result in a sudden drop in the animation speed, and even the loss of system stability). The principle of *first come, first served* is in force.

Immediately afterwards, a classic particle emission occurs, just like in the older particle effect systems based on calculations performed by the CPU.

Both emissions work according to the following algorithm:

- If the set D is not empty, take the index of that dead particle.
  - Otherwise increase P_{curr} and take its previous value – this is the new index of the particle.
- Fill a particle, with this index, with data from the emitter.

The last stage of the first step is the, so called, *defragmentation*, i.e. transferring a given number of particles, whose indices are close to P_{curr}, to places marked with indices of particles in the set D. This operation is rather light because only up to a few thousand particles are being transferred per animation frame and it takes about 100 microseconds on an XBox One console. Its goal is to decrease the P_{curr} value, so that in the next animation frame it would be possible to create a smaller G_{sim} quantity. Without this procedure, the buffer, even for a small quantity of particles (< 1000), can very quickly fill in the whole prepared pool ($\sim 0.5 \times 10^6$), thus forcing a maximally high G_{sim} value or looking for free spaces in the memory for new particles each time.

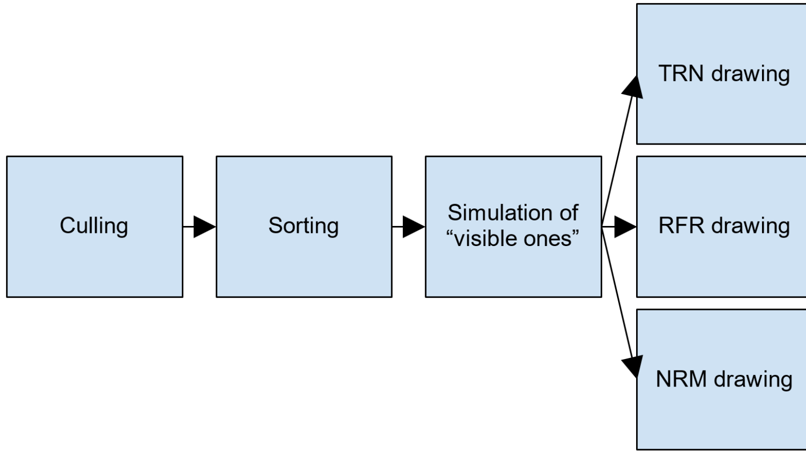

Fig. 3. Chart of operations of the stage two of the algorithm.

Second stage After performing a simulation of all the particles and emitting new ones or culling old ones, the second stage takes place, in which calculations for individual cameras are performed. They are directly connected with instantiating, being the culmination of the entire process. The stage is comprised of the steps shown in Fig. 3.

Another reason for which it is worthwhile to pursue the reduction of the P_{prev} value is the frustum culling step. It must be performed for all P_{curr} particles (including the possibility that dead particles are inside that block). After performing this operation we get the set denoted A_0 on the output, in which the indices of all particles visible to a given camera, denoted as C_0, are located.

With the information about which particles are necessary to properly draw a scene (their indices are known), as well as knowing their distance from a given camera, it becomes possible to sort them according to distance to ensure a proper order of drawing. To achieve this we used the bitonic sort algorithm [1] and we have based its implementation on AMD's publication [4] presenting a very minimalistic particle effects system.

The particles in the memory are sorted not only by distance to the camera, but also according to the technique they will be drawn with. Thanks to that, after sorting, the buffer contains a few continuous blocks with particles ready to be drawn with the use of various techniques.

Example For example: the buffer containing the set A_0 (i.e. containing particles which passed the visibility tests) before sorting looks like shown in Tab. 1, where I_x – particle

Tab. 1. The buffer with set A_0 before sorting.

$I_0 = 123$ $D_0 = 12.3$ $T_0 = 0$	$I_1 = 345$ $D_1 = 4$ $T_1 = 0$	$I_2 = 234$ $D_2 = 3$ $T_2 = 1$	$I_3 = 4$ $D_3 = 59$ $T_3 = 1$	$I_4 = 3$ $D_4 = 22.4$ $T_4 = 2$	$I_5 = 45$ $D_5 = 1.2$ $T_5 = 0$	$I_6 = 1222$ $D_6 = 23$ $T_6 = 0$

Tab. 2. The buffer with set A_0 after sorting.

$I_0 = 123$ $D_0 = 12.3$ $T_0 = 0$	$I_1 = 345$ $D_1 = 4$ $T_1 = 0$	$I_2 = 234$ $D_2 = 3$ $T_2 = 1$	$I_3 = 4$ $D_3 = 59$ $T_3 = 1$	$I_4 = 3$ $D_4 = 22.4$ $T_4 = 2$	$I_5 = 45$ $D_5 = 1.2$ $T_5 = 0$	$I_6 = 1222$ $D_6 = 23$ $T_6 = 0$

index in the set P, D_x – particle distance from the camera in which it is drawn, T_x – technique with which the particle will be drawn (to be explained further).

After sorting, the buffer will look as shown in Tab. 2.

Final steps The next step is *simulating visible particles*, that is, calculating all variables needed to properly draw each particle. Up until now we have used a representation containing only data realted to movement, position, lifespan, etc. Information on textures, color, gradients, and other features related solely to drawing the particles was not needed and additionally would increase the amount of transferred data. After this step, the data representation optimized for drawing in various techniques alone becomes correct. It lacks e.g. information on the particle's velocity or the forces acting on it. Such an approach makes it possible to decrease the amount of data needed in a given moment from over 40 MB to around 20. This is critical in the case of using XBox One and its 32 MB of ESRAM.

The last stage, which crowns the whole process, is drawing. Various techniques are used in the rendering engine for this task. Each of them is dedicated to drawing different type of objects. The technique used to draw translucent objects is called TRN, the one used to draw refractive objects is called RFR, opaque objects are drawn with OPQ technique, etc.

As it was mentioned before – it takes place in a few stages of scene rendering, because particles can have different purposes. The most popular ones are translucent and are an ideal fit for drawing, e.g. fire, smoke, dust, snow, etc. The TRN technique fills this need. The screenshot shown in Fig. 4 presents fire and smoke generated with this technique. Background image was used by our artists to show refraction of light.

Another part of particles in our system consists of ones responsible for the refraction of light. As it was said before – they are drawn with the RFR technique – see fig. 5

The combination of the two effects above allows for a quite realistically looking fire.

Fig. 4. Example of the particle effect with fire and smoke.

Fig. 5. Example of the light refraction.

Thanks to the data being sorted in the above described way, it is very easy to implement particle drawing in each of the techniques. Each time the input data are exactly the same – the buffer with data created after the *simulation of visible particles* and the buffer with a number of particles for each technique. Knowing the order of techniques in the buffer, it is easy to determine parameters for further drawing operations (using the DrawInstancedIndirect method [2]).

Referring to the previously given example with sorting let us observe the following:

- DrawInstancedIndirect for the TRN technique (i.e., $T = 0$) will draw the first 4 particles;
- DrawInstancedIndirect for the RFR (i.e., $T = 1$) will draw 2 further ones (without first 4 TRN particles);
- DrawInstancedIndirect for the NRM technique (those responsible for modifying normal vectors in GBuffer) (i.e., $T = 2$, without first 6 TRN and RFR particles).

Of course, the above operations will not take place one after the other, but only when each of them is required. All techniques are rendered in specific order.

5. Data storage

From the point of view of the HLSL code, the particle, during the first stage of simulation, is represented by the quite highly compressed structure shown below, constituting the description of its physical properties:

```
struct GpuFx_ComputeParticle
  {
    float3  position;
    uint    sizeXY;
    uint2   flags_emitterId_ttl_seed;
    uint2   velocity_orientation;
    uint2   acceleration_ttlPhase;
    uint2   orbitOffsetXYZ_parentId;
  };
```

Floating-point numbers are used only for storing the position in the world in view of precision. All the other variables are stored as a half type. These types of measures aim to decrease the size of an individual particle stored in the memory. It should be remembered that one of the main objectives of the designed system told of supporting up to 0.5 million particles in real time with the use of current generation consoles, i.e. Playstation 4 and XBox One.

The stage of detecting particles in the frustum and sorting allows for using an even smaller structure:

```
struct GpuFx_ParticleInstance
  {
    uint   technique_index;  // 8  bytes - technique (NRM/RFR/OPQ...)
                             // 24 bytes - index
    float  distance;
  };
```

Only the data prepared in this way make it possible to prepare a buffer which will be the input for the drawing operation. It takes place in the so called *simulation of visible particles* stage and works well for filling the frustum composed only of some tens of bytes of data that will be read accordingly and interpreted by the vertex shader for one of the techniques. Each technique uses a different set of data, but all the particles share the same buffer.

6. Conclusion

The presented algorithm makes it possible to minimize the number of places in which data are loaded – further data are added only when they are actually needed.

The main course of the algorithm does not assume special cases. Using one buffer to store particles belonging to various techniques allows to maximally utilize the allocated memory without the need of establishing any smaller budgets. Everything takes place dynamically and is adjusted to production needs.

Drawing particles, thanks to the locality of data, favors the operations related to the data cache.

7. Further development

It is key to solve the problem of the low, so called, *fill-rate*, emerging while drawing large amounts of big particles close to the cameras, for example, by using smaller particles or hiding the invisible ones in a better way. Achieving this will require better co-operation with the team responsible for creating particle effects in order to work out an optimal way to design the effects.

Additionally, further work is planned, adding minor functionality such as support of vector fields and improvements in support of animated textures.

Also, the optimization is of priority here – in terms of both memory as well as performance.

The new system also includes a particle effect editor deliberately omitted in this paper. In further work it is expected that the emphasis will be put on ergonomics of its use and its stability.

Acknowledgement

The paper was created as the result of the project "Industrial Research on the Technology of Scaled City for the Needs of Computer Games" (original title in Polish: „Badania przemysłowe nad technologią skalowanego miasta na potrzeby gier komputerowych").

The project is co-financed by the European Regional Development Fund under the Innovative Economy Operational Programme.

References

[1] Vardit Cohen, Brian Hillman, Kyle Suarez, and Sesh Venugopal. Bitonic sort. `http://www.cs.rutgers.edu/~venugopa/parallel_summer2012/bitonic_overview.html` [Online; accessed Dec 2016].

[2] Microsoft Corporation. ID3D11DeviceContext::DrawInstancedIndirect method. `https://msdn.microsoft.com/en-us/library/windows/desktop/ff476413(v=vs.85).aspx` [Online; accessed Dec 2016].

[3] Microsoft Corporation. Shader Model 5. `https://msdn.microsoft.com/en-us/library/windows/desktop/ff471356(v=vs.85).aspx` [Online; accessed Dec 2016].

[4] Jason Stewart (jstewart-amd). AMD GPU Particles Sample. `https://github.com/GPUOpen-LibrariesAndSDKs/GPUParticles11` [Online; accessed Dec 2016].