Large Scale Dynamics Visualization On GPU

Lei Kang¹, JianZhe Liang²
¹Xi'an Shiyou University, Xi'an, Shaanxi Province. P.R. China
²Tsinghua University, Beijing, P. R. China
kangl@xsyu.edu.cn

Abstract. Simulating and visualizing dynamics of systems consisting of large amount of particles on GPU can be generally difficult due to limitation on data size and programmability. We propose a scheme for developing such algorithms when there exists certain kind of independence and similarity between subsystems. We have instantiated the scheme into algorithms and have integrated them into a real-world game engine. Empirical results show that these algorithms incur little overhead with tens of thousands of particles.

1. Introduction
One of the main goals of computer graphics is to compute and demonstrate natural processes to create immersive scenes. A widely adopted approach is to create, simulate and visualize various physical models following certain dynamics [1]. Challenges, however, can occur when one considers large scale dynamics visualization where a system consisting of tremendous amount of objects has to be simulated. Carrying out such computation solely on CPU would be unfeasible even if one considers the motion of different components to be independent since the time required grows linearly as object amount increases.

A natural approach would be utilizing accelerators to enable high parallelism. Among all candidates, GPU is the most promising one. First proposed in 1980s [2], these integrated circuits enable high parallelism for rendering tasks including transform, light, clipping, etc. Modern GPU even provide interfaces for running custom programs for individual geometry primitives and even for individual pixels, enabling development of algorithms with a nature of parallelism. Though a GPU may possess hundreds of cores, each single core allows only limited operations compared with a CPU core. There can be no communication among different cores when performing render-related computation. Besides, all of the data required must be allocated in GPU’s own memory before rendering, posing limitation of the size of data used for simulation [3].

In this paper we propose a scheme for visualizing large scale dynamics on GPU. Our key observation is that when a system consists of many independent subsystems following the same analytical form of dynamics, we can structure system parameter in a levelled fashion to achieve data efficiency while performing simulation computation directly on GPU and visualize the system's dynamics in real time.

The rest of the paper is structured as follows. In section 2 we review related work on particle system and instanced rendering which provide inspiration upon our scheme. Section 3 starts with a rigorous while flexible formal framework of our scheme followed by two instantiated algorithms to simulate rain. We have integrated these algorithms into an existing game engine and have measured...
their performance in terms of time cost. Section 4 provides experiment details. The conclusions are summarized in section 5.

2. Related Work
Particle systems contain several related objects to simulate a certain amount of particles’ motion under customized external forces in a user defined manner. In Qt’s implementation [4], such a system contains an emitter to emit particles at a certain rate within some location, a group of particles to compute logical trajectory where each particle exists for a short period of time, an affector to add optional effects to the simulation and a render to visualize particles. Such systems have been widely used to simulate phenomena like fire, smoke, meteor, etc. with chaotic dynamics. In many cases, however, there are systems with much simpler dynamics, yet one may want to simulate and visualize tens of thousands of them simultaneously. It would be inevitable to utilize modern accelerator like GPU to enable parallelism on rendering. The flexible yet complex nature of particle maintenance and on-the-fly trajectory computation makes particle systems unfeasible to meet such requirement. In our scheme we also assume that the finest grain of objects having physical properties is particle. Instead of repeatedly creating and destroying particles, however, each particle in our implementation moves in a periodic manner. Note that although the whole trajectory may seem to be artificial, local motion of each particle still follows physical restrictions. Setting initial time to different instant within the motion cycle also helps to create a visual effect of randomization.

Instanced rendering is an interface intended to separate models’ geometry data from attributes, enabling rendering of multiple models from a single call. Such an interface reduces the overhead of CPU-GPU communication on rendering data and instruction [5]. In our problem, trajectory computation of a single particle require plenty of parameters including boundary dynamic conditions like initial velocity, motion limitations like the lowest y value a particle can reach, initial time instant, etc. The situation gets worse when one uses a triangle or squire to visualize a particle so as to create perspective effects since multiple vertices may share the same particle parameters. A naive approach which assigns each geometry vertex a same copy of particle parameter leads to severe redundancy of rendering data which, apart from increasing CPU-GPU communication overhead, quickly eats up GPU memory. Instead of aggregating data at particle level, we perform aggregation on different levels to reduce redundancy. We rely on instanced rendering which, at lowest level, updates vertex data at different interval to achieve such separation.

3. Implementation

3.1 Formal Description
In this section we present a rigorous yet flexible scheme for computing and visualizing particle dynamics. Such a scheme \( S \) is a five tuple: \( S = (P_{\text{geometry}}, P_{\text{particle}}, P_{\text{group}}, f, T) \) where

- \( P_{\text{geometry}} \) is a set of vertex data describing the geometry construction for visualization a single particle. In most cases, one may wish to use a simple 3D object to represent a particle so as to create a perspective effect which cannot be provided by drawing points or lines. Vertices’ position in \( P_{\text{geometry}} \) can be displacement relative to the particle's position. All of the particles in a scheme share the same geometry construction.

- \( P_{\text{particle}} \) is a closely related group of particles. Such relation may be sharing of a certain set of parameters such as a common displacement of a local coordinate origin relative to world coordinate system when calculating trajectory, or involving interactions between particles. Note our scheme does not disable interaction, but restrict such interaction to be performed only within the same group of particles. Particles from different groups move independently. Each item in \( P_{\text{particle}} \) contains a single particle's boundary dynamics such as initial position and velocity.

- \( P_{\text{group}} \) stores parameters that varies in different particle groups. Such parameters may not only be dynamics-related such as various kinds of factors when computing trajectories, they can also include transformations applied to geometry constructions mentioned above. For instance, if one use squares to
visualize particles then each group may have its own randomized rotation to be applied on squares so one can see such squares from any perspective. Each group may also contain a common bias to be added on $t$ when calculating the motion function $f$ (see below). Randomizing the bias enables visualizing different state of motion simultaneously, like playing several copy of a gif file starting from different frames.

- $f(p, g, t) : P_{\text{particle}} \times P_{\text{group}} \times [0, T] \to \mathbb{R}^3$ is a function that calculates a particle's position within world coordinate given its initial state (possibly including its id when taking interaction into consideration), group configuration and current time instant. Although in certain situations $f$ may also return a transformation applied to a particle's geometry representation (such as an element in the $SE_3$ transformation group), in such cases, however, a particle no longer represents a mass point which physically has only position and mass information but more likely a rigid body. We restrict ourselves to the physical representation of mass point for particles for the sake of simplicity. The exact form of $f$ may be an analytical expression or a series of precomputed key points (to perform interpolation). The latter form can occur when exact solution requires integration or taking particle interaction into consideration.

- $T$ is the length of all of the particle's motion period.

While rendering, geometry representation for a particle is treated as an instance, entry pointer towards $P_{\text{geometry}}$ is updated every vertex. Pointer to $P_{\text{particle}}$ is updated every instance while to $P_{\text{group}}$ every $|P_{\text{particle}}|$ instances.

### 3.2 Examples

In this section we instantiate the scheme formulated above and integrate the generated algorithms into a game engine to give an example of use case. We chose an open-source Minecraft engine [6]. This project is developed in C and utilizes modern OpenGL's shader language to perform 3D rendering. We have forked this project and have added the feature of raining effect. Our simulation of rain includes both dropping and splashing effect. We will show how to instantiate the scheme to organize different parameters defined above, given a time instant $t \in [0, T]$, the position of particle $j$ within group $i$ is calculated by the following formula

$$
\begin{align*}
\mathbf{p}_j &= \mathbf{p}_i + \mathbf{b}_i + \mathbf{f}(\mathbf{p}_i, \mathbf{g}_i, t) \\
\mathbf{p}_i &= \mathbf{p}_0 + t \mathbf{v}_i,
\end{align*}
$$

where $\mathbf{p}_i$ and $\mathbf{v}_i$ represent the initial position and velocity of the particle respectively. $\mathbf{f}(\mathbf{p}_i, \mathbf{g}_i, t)$ is the motion function calculated as above. The geometry representation for a particle is treated as an instance, entry pointer towards $P_{\text{geometry}}$ is updated every vertex. Pointer to $P_{\text{particle}}$ is updated every instance while to $P_{\text{group}}$ every $|P_{\text{particle}}|$ instances.
The absolute position of vertex $k$ within the drop's representation rectangle would be

$$p_{i,j} = p_{i,j} + R_i d_k$$

where $d_k$ is the vertex's displacement vector to particle $j$.

In splashing simulation, each particle is visualized as a square, with mass still concentrating on its geometry center so we may reuse $P_{\text{geometry}}$ of dropping simulation with $l = w$. Each particle is emitted from a certain position in world coordinate with varying initial velocity. The particle then moves in a parabola trajectory under the action of gravity only. We group certain amount of particles and assign them the same initial position which is the origin of the group's local coordinate system (see figure 1 (b)). Thus $P_{\text{particle}}$ contains only initial velocity of different particles in local coordinate system: $P_{\text{particle}} = \{v_i\}$. In our implementation, we distribute direction of $v_i$'s unit projection on xz plane uniformly while adding noises on their angle with y axis and $|v_i|$. Each group contains the position of its local origin in world coordinate, a time bias as well as a rotation along y axis as mentioned above, thus

$$P_{\text{group}} = \{(p_i, t_b, R_i)\}$$

Another important parameter here is the span of motion denoted by $T_{\text{move}}$. As we cannot perform complex collision detection on GPU, drops splashed continues to move downward after they hit a block's surface. Thus one have to control the distance a drop can fall so as not to see rain penetrating his/her roof. Similarly, we have to control the distance a particle flies horizontally so as to keep the rain outside houses. We use $T_{\text{move}}$ to control the time a particle moves and is visualized within a period. After $T_{\text{move}}$, it is gone (not being visualized). We do not use $T_{\text{move}}$ as $T$ so we can utilize $T$ to control splashing frequency without worrying the problems mentioned above so long as $T > T_{\text{move}}$.

Thus given a time instant $t_c$, the position of a particle $j$ within group $i$ is

$$p_{i,j} = f(e_i^j e^i, t_c) = \begin{cases} y_j - v(t_i + t_c) & y_j - v(t_i + t_c) \geq y_{\text{min}} \\ y_j - v(t_i + t_c) - y_j + v(t_i + t_c) & y_j - v(t_i + t_c) < y_{\text{min}} \end{cases}$$

where

$$M_T(t) = \begin{cases} t & t \in [0, T) \\ t - T & t > T \end{cases}$$

Vertices' positions are calculated the same way as in dropping simulation. When it comes to the infinity case, instead of setting position vector's components to NaN, we set a Boolean flag shared between vertex shader and fragment shader. The fragment shader discards the pixel once such a flag is set.
4. Evaluation
We have evaluated the performance of our generated algorithms for rain simulation by measuring the
time cost of corresponding rendering calls. Our experiment has been carried out under Ubuntu 18.04
on a machine with intel CORE i7 CPU and NVIDIA GTX 1050 Ti. To ensure the rendering has
finished when the drawing function returns, we use glFinish [7] before and after the call being
measured to force previous render instructions to be completed before glFinish returns. Time is
measured utilizing Linux’s system call getrusage[8] to get accurate (in microseconds) elapse of user
space time.

![Figure 2 visual effect when σ=0 (a) and σ=1 (b)](image)

We have measured the time cost for dropping and splashing simulation separately with different
amount of particles. The exact amount of particles for each simulation process is determined by a
control variable \( \sigma \) varying from 0 to 1. In dropping simulation, we set \( \text{interval} = 40 - 30\sigma \), with
\( h = 200 \). Each \( 1 \times 1 \) square of a \( 32 \times 32 \) grid surrounding the user's camera contains a group of rain
drops' horizontal coordinate. Therefore the amount of particles in dropping simulation ranges from
1280 to 5120. In splashing simulation, the number of particles within a group (denoted by \( n_p \)) and the
number of groups in a \( 1 \times 1 \) square (denoted by \( n_g \)) are both determined by \( \sigma \) where
\( n_p = 10 + 20\sigma \) and \( n_g = 1 + 5\sigma \). Each group's local origin is still within a \( 32 \times 32 \) grid around user's camera. Thus
the total amount of particles in splashing simulation ranges from 2560 to 46080. An overall visual
effect when \( \sigma = 0 \) and \( \sigma = 1 \) can be seen in figure 2.

The average cost under different \( \sigma \) in two simulation is shown in figure 3. Note that time cost does
not increase as fast as the amount of particles does in both simulation due to high parallelism of GPU.
We have also measured the average time cost for block rendering with several thousands of squares in
the engine which is about 600μs. This shows that the time cost are still pretty low (180μs) even when
the amount of particles exceeds 40000.

5. Conclusions
We have proposed a data-efficient scheme for computing and visualizing dynamic systems containing
large amount of particles on GPU and have instantiated the scheme to generate algorithms that
simulate rain. We have integrated such algorithms into a game engine and have tested its performance
by means of measuring time cost on rendering call. The results show our algorithms scales up to tens
of thousands of particles with acceptable overhead.
Figure 3  time cost of dropping simulation (a) and splashing simulation (b)

References
[1] Physically-Based Modeling for Computer Graphics: A Structured Approach Barzel, R. and Barr, A.H. Elsevier Science 2013
[2] Advances in Computer Graphics II Hopgood, F.R.A. and Hubbold, R.J. and Duce, D. Springer Berlin Heidelberg 1986
[3] NVIDIA’s Next Generation CUDA Compute Architecture: Fermi NVIDIA 2009 https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
[4] http://doc.qt.io/qt-5/qtquick-effects-particles.html
[5] https://www.khronos.org/opengl/wiki/Vertex_Rendering#Instancing
[6] https://github.com/fogleman/Craft
[7] https://www.khronos.org/opengl/wiki/Synchronization
[8] http://man7.org/linux/man-pages/man2/getrusage.2.html