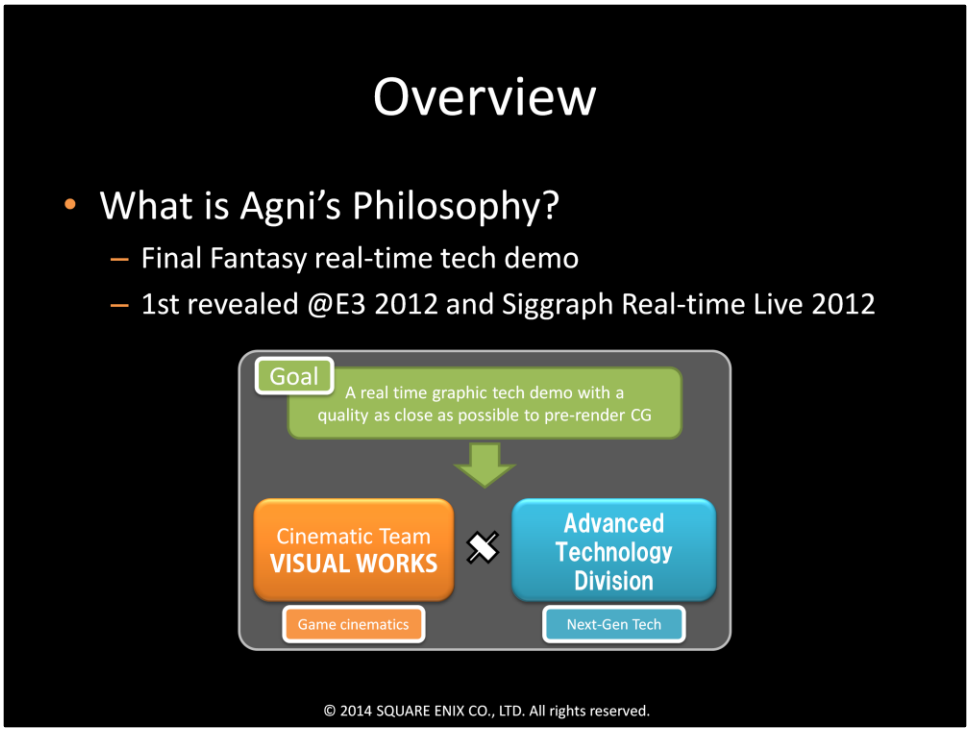
Practical Applications of Compute for Simulation in

AGNI'S PHILOSOPHY

FINAL FANTASY REALTIME TECH DEMO

About me

- Name : Napaporn Metaaphanon (Noi)
- Nationality : Thai
- 2008 → came to Japan for my Master degree
- 2010 → joined SQEX
- 2011 → Agni's Philosophy project
  (particle FX, glass rendering)
- 2012 → FFXIV
  (cloth simulation)
- 2013 → FFXV
  (GPU related VFX)



Agni's Philosophy is a collaboration between VISUAL WORKS, which is our Cinematic Team, and our R&D department, the Advanced Technology Division.
The goal of this project is to create a real time graphics tech demo with a quality as close as possible to pre-rendered CG.
It was first revealed at E3 and Siggraph Real-time Live in 2012

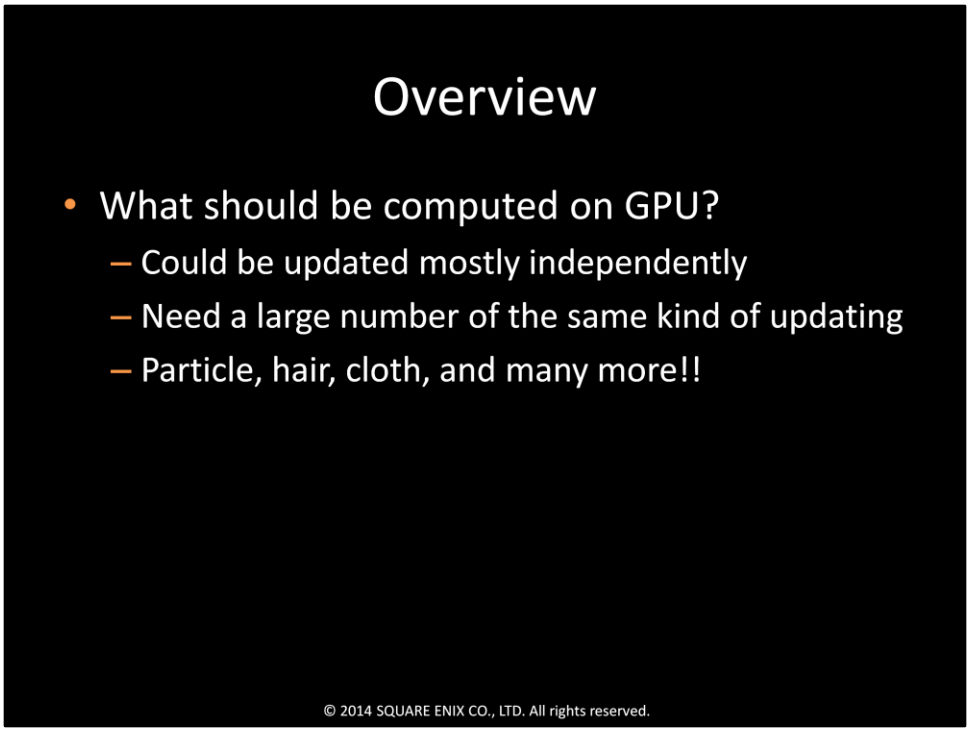
Overview

- www.agniphilosophy.com

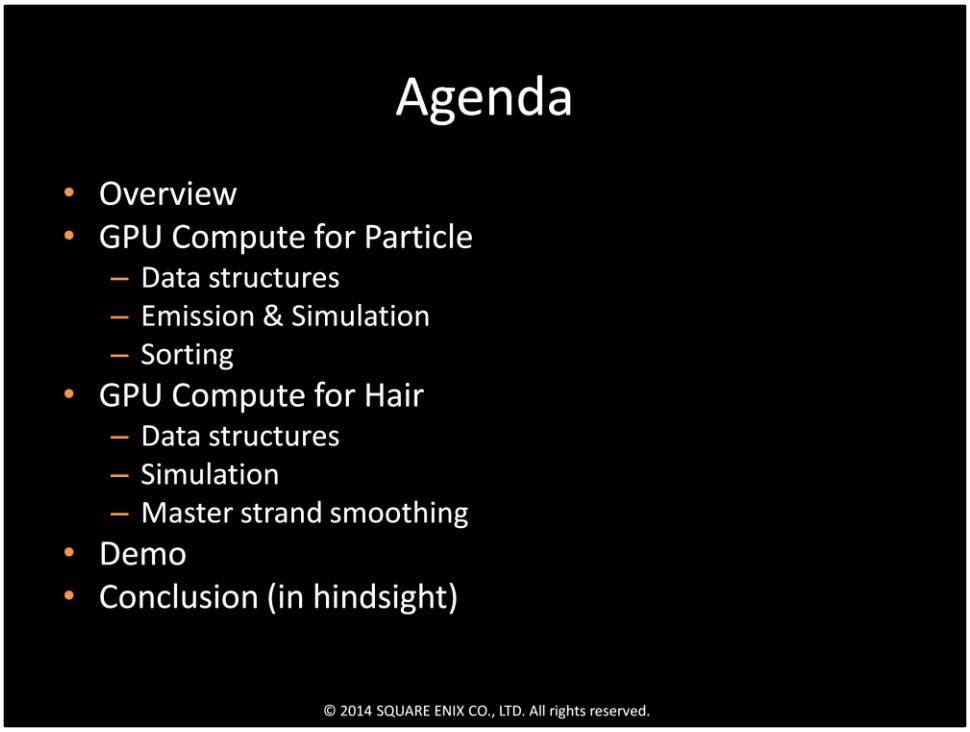
Overview

- Why did we use GPU?
  - We were making FF tech demo!!
    → needed tons of VFXs
    → too heavy on CPU
    → had to move some operations that could be calculated in parallel to GPU

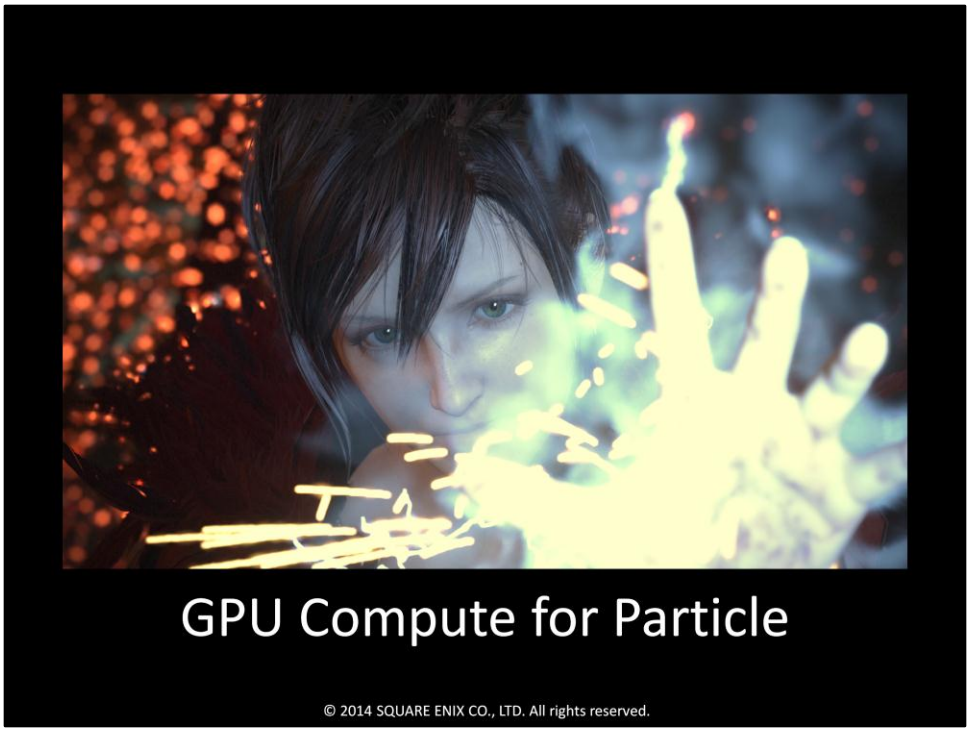
The FF series always use tons of VFXs, and since we were making a tech demo for FF, we also need them.
And since it was very heavy to do everything on the CPU, we had to move some operations that could be calculated in parallel to the GPU



What kind of operations could be moved to the GPU?
First, we should be able to update them independently or mostly independently.
And, second, we need a large number of them. The first thing that came to our mind was particles.
Without interaction among particles themselves, they can move freely, and we also need a huge number of them for FF.
Then, as an extension for particle, we also thought of hair, cloth and many more things.

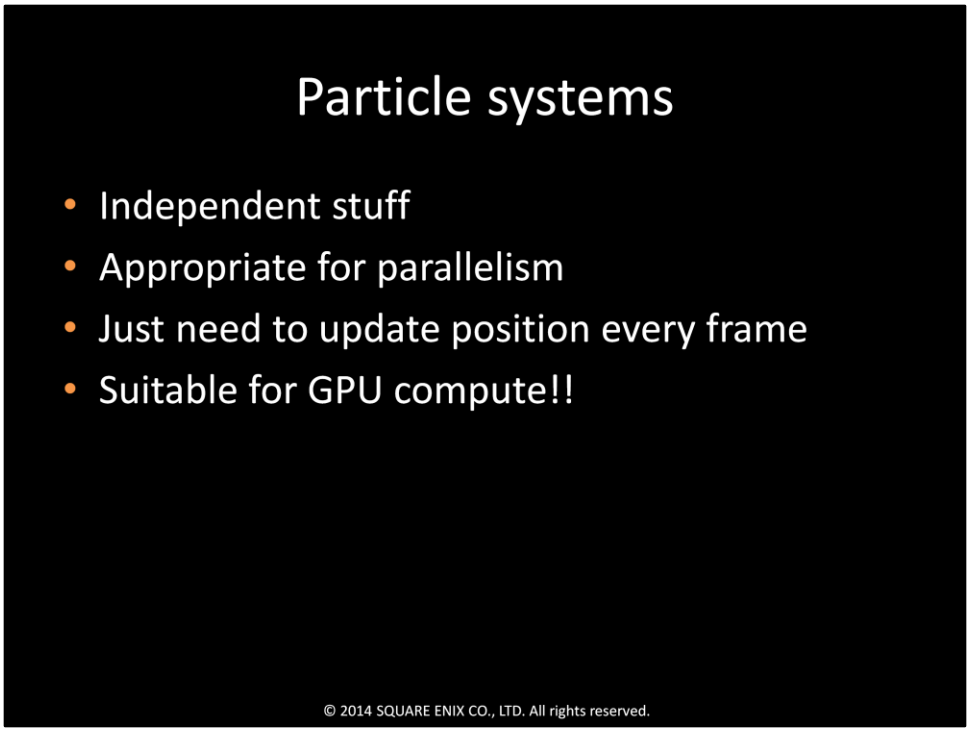


I have to mention here that, at that time, it was the first time we touched DX11 and also GPU compute.
What I'm going to tell you now is just what we have done and found, not the best solution or anything like that.
So, in the conclusion part, I'll also let you know what we might have done if we had had more time.

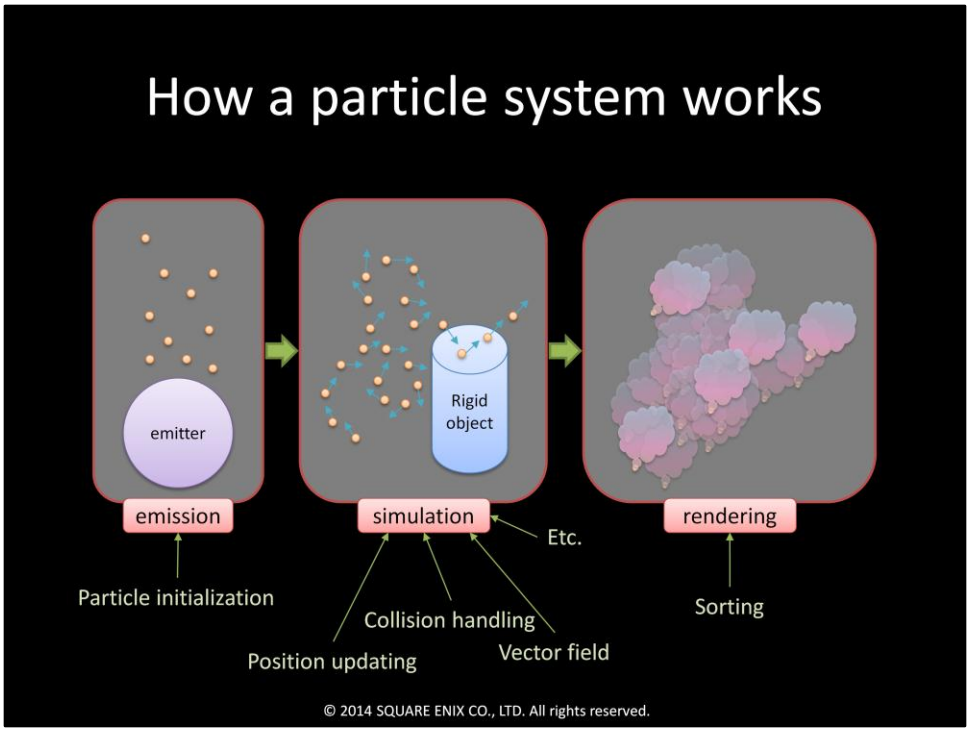


A video that I'm going to show you is an extraction from the real demo, which includes some selected shots with many types of particles.

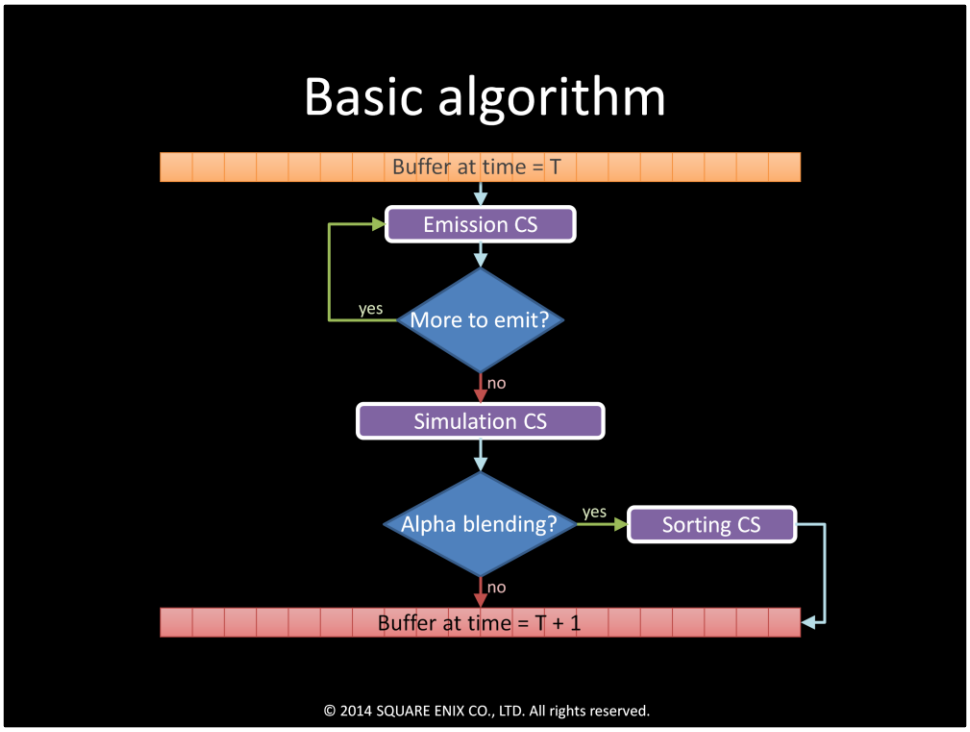
First, let's see how it looks!!



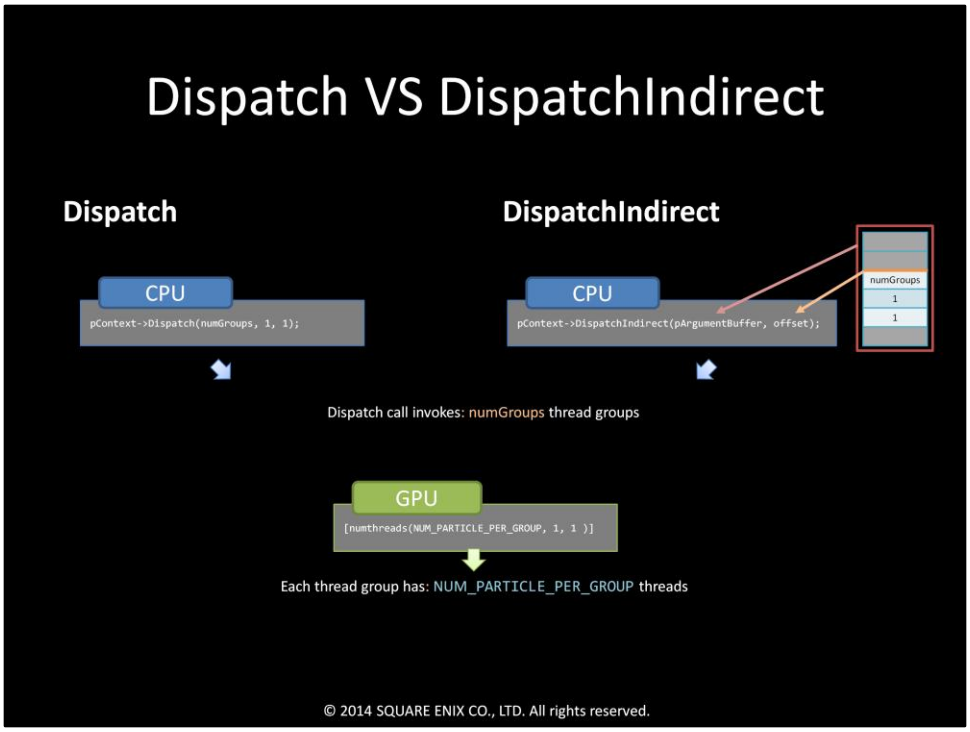
As you can see from the video, each type of particle we used can move independently. So, position, velocity or other properties of each particle can be updated independently in parallel.
And, that's the reason we pick this as the first choice to be implemented with GPU compute.



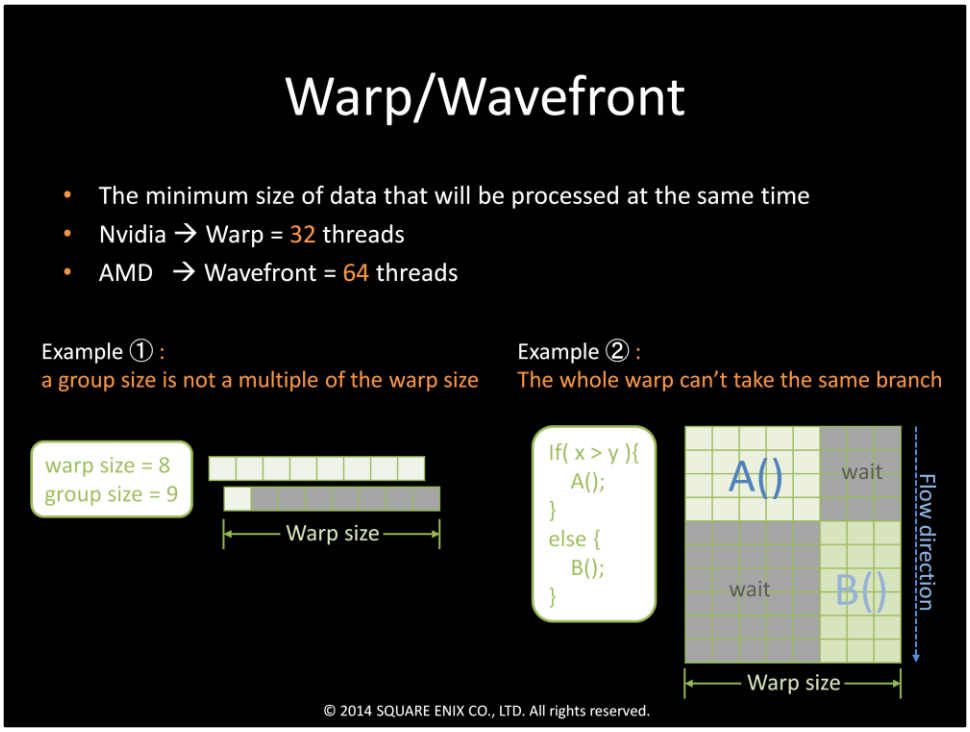
So, how does a basic particle system work?
The first thing is we have to create some particles using the emitter, then we simulate and render them on the screen.
In our system, we use 3 CSs for these 3 parts.
The emission CS initializes particles and adds them to a buffer.
The simulation CS does time integration, collision handling and some other processes, and then updates positions and velocities.
For the rendering part, we use a CS for sorting.



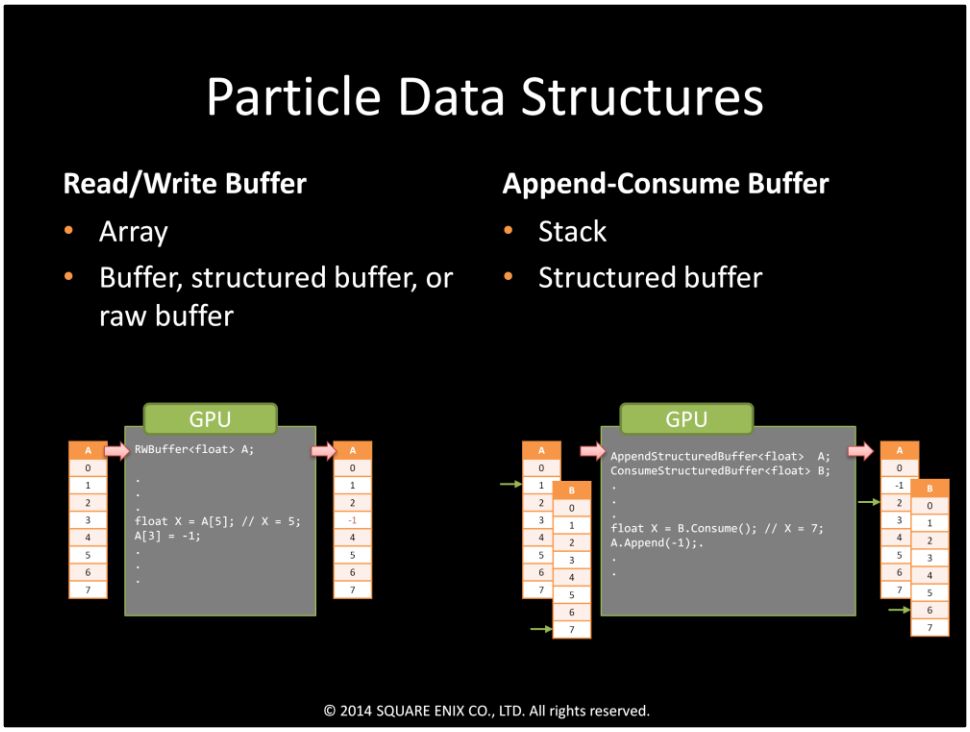
Here is the flow of our data.
Starting at time T, we will send the buffer to the emission CS; 1 dispatch call per emitter.
Then we will pass the buffer to the simulation CS.
After that, if alpha blending is needed, we will send the buffer to sorting CS, and get the result buffer for time T+1.



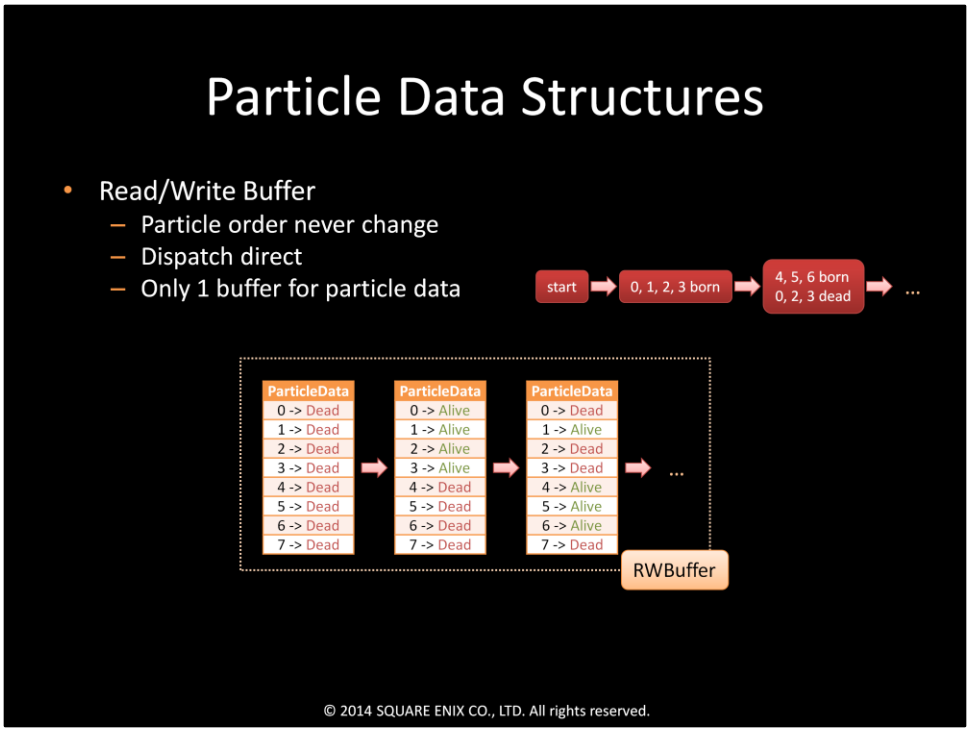
To execute commands in a CS we have to call dispatch with the number of thread groups we want to use.
The number of total threads to be run is the number of thread groups multiplied by the number of threads in the groups.
There're 2 types of dispatch call: direct and indirect.
The only difference between these two is, for direct dispatch, we have to pass the number of thread groups we want to run directly through the dispatch function parameters.
But, for indirect dispatch, it will be passed through a GPU buffer. The shader code will all be the same.



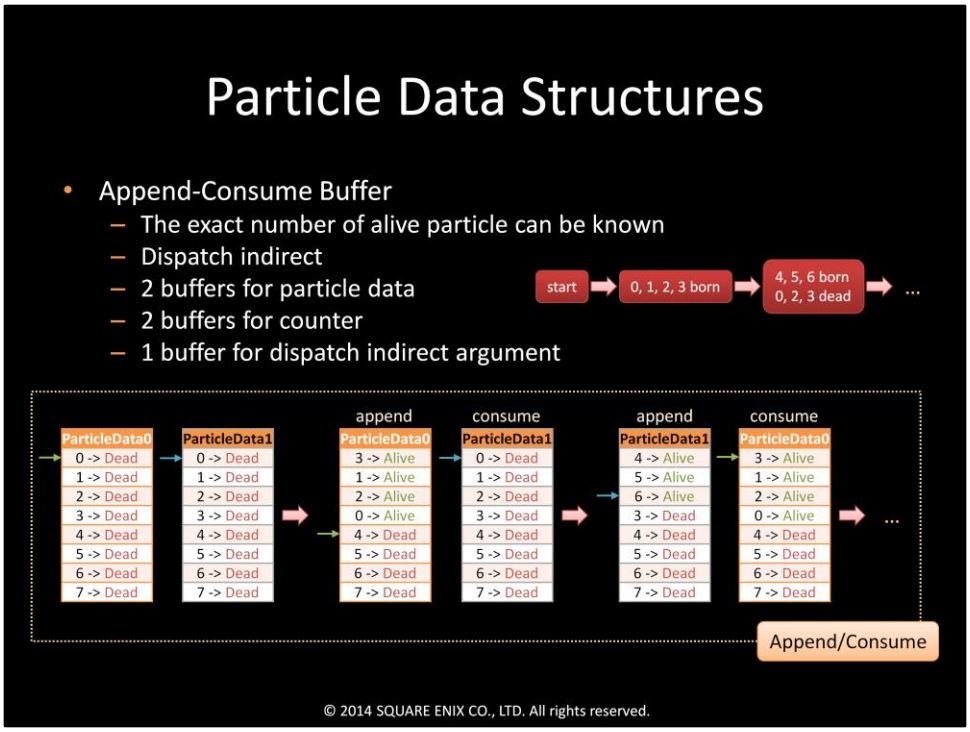
Currently, on NVIDIA HW, we call it a "warp", which consists of 32 threads, and, on AMD HW, we call it a "wavefront", which consists of 64 threads.
If our code does anything that causes the threads in a warp to be unable to execute the same instruction, then some threads in the warp will be diverged during the execution of that instruction.
There are 2 obvious cases that cause a divergence:
When a group size is not a multiple of the warp size.
Or when there's a branching where the whole warp is unable to take the same branch.
Most of the time, we should avoid these kinds of situations.



We tried using 2 types of buffers: typical buffers and append-consume buffers.
A typical read/write buffer is basically an array, which could be a buffer, structured buffer or raw buffer.
We can specify an array index and access the data in the array directly.
However, an append-consume buffer is a stack that can only be a structured buffer.
When append is called, it will add new data to the next index of the stack, and when consume is called, it will pop data out from the stack.
The counter of the stack will be automatically updated using an atomic operation.

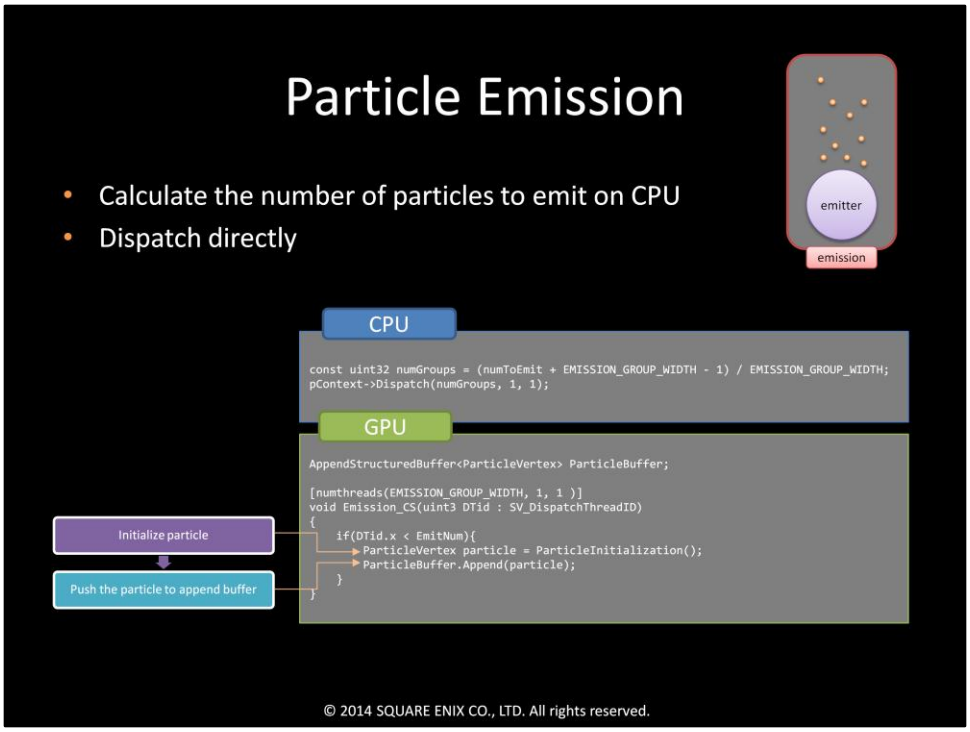


For read/write buffer cases, only one buffer is needed for particle data storage.
For each particle, we can simply read and write back to the same index of the buffer.
However, it is quite difficult to know the exact number of living particles and also which indices they are in.
What we tried was tracking the maximum number of particles we had emitted, not the number of alive particles, and then used that number to calculate how many thread groups we should dispatch.
As a result, we had to dispatch and draw with the number of particles we had emitted, and that was a waste.
Also, since the same particle always uses the same slot in the buffer, it is still left there when it is dead.
So, we can have both alive and dead particles in the buffer.
When we update or render particles, we need to check if each particle is still alive, and just skip the dead ones.
That also means there is a high possibility that some wavefronts will be diverged.

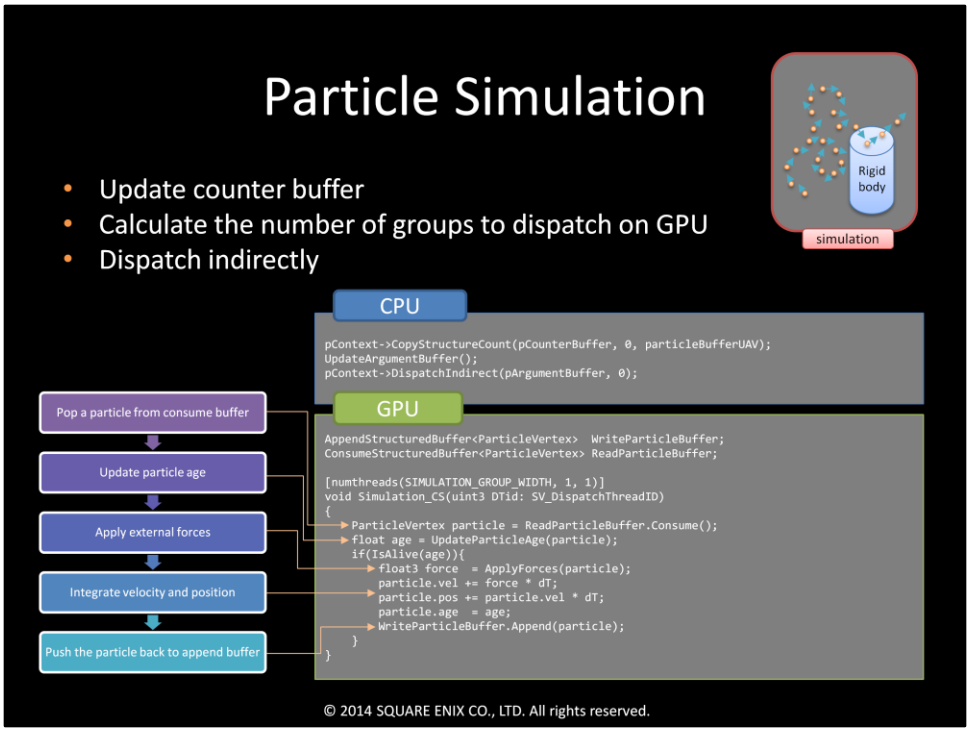


To solve these problems, we tried using an append-consume buffer instead. We used two buffers to store particle data: one for AppendStructuredBuffer, and the other for ConsumeStructuredBuffer. Everytime we update, we pop a particle from the consume buffer, and if the particle is not dead, we will push it back to the append buffer.
To track the number of alive particles and dispatch indirectly, we also need some more buffers to store a counter and also a dispatch indirect argument
And, since the order of particles in the buffers could change every time we update, if we don't sort the buffer; the translucent particles will flicker. (with read/write buffer, it won't flicker, but the result might be incorrect).
Anyway, to produce a correct alpha-blending result, we always need sorting.
So it didn't sound so bad, and we decided to go with append-consume buffers.

The explanation from now on will be based on append-consume buffers.



Emission is very simple. What we need to do is just initialize particle data (in our case, position, velocity, life and id) and push it into our append buffer.
Because we know the number of particles we want to emit, we can dispatch it directly.
Since the number of total threads here is (numGroups * EMISSION_GROUP_WIDTH), we might have more threads than the number of particles we want to emit. That is why we have to add a check inside our CS to only initialize and append a particle to the buffer when the thread id is less than the number of particles we want to emit.

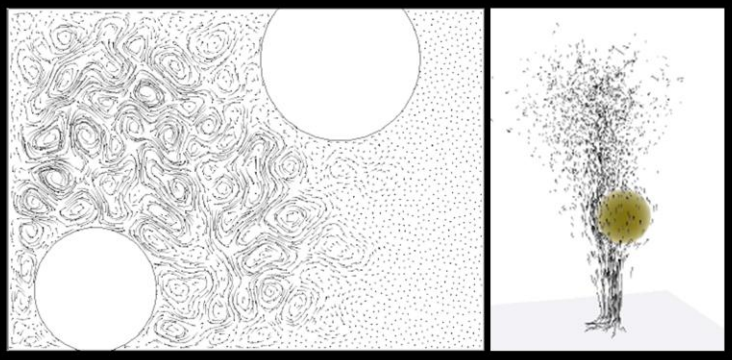


Here is just a simple case where each particle does not interact with any others, so we can update them separately on each thread.
In this step, since we do not know the exact number of particles in the buffer, we will dispatch it indirectly.
First, we have to update the counter buffer, and then update the argument buffer and pass the argument buffer as a parameter to dispatch function.

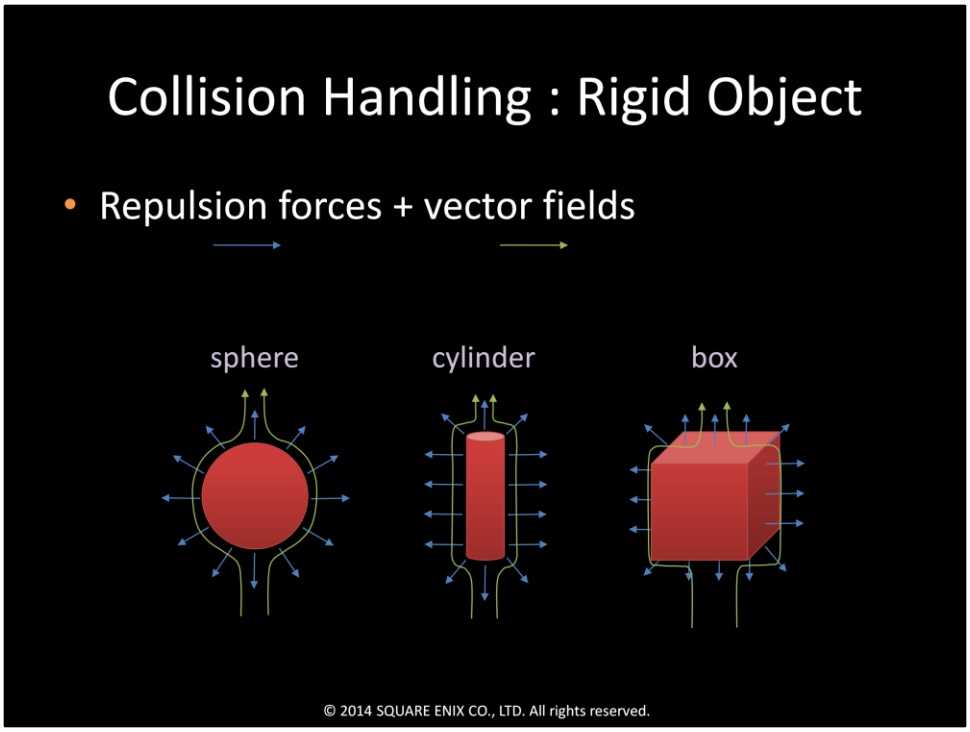
On the shader side, we will pop a particle out from a consume buffer, process, then push it back to the append buffer.

Curl Noise

- "Curl-Noise for Procedural Fluid Flow" [Bridson *et al.* 2007]



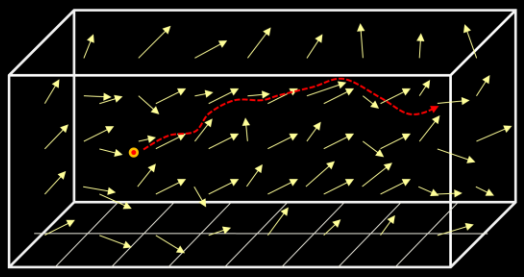
We also added some more features to create more complex movement of particles: for example, curl noise.



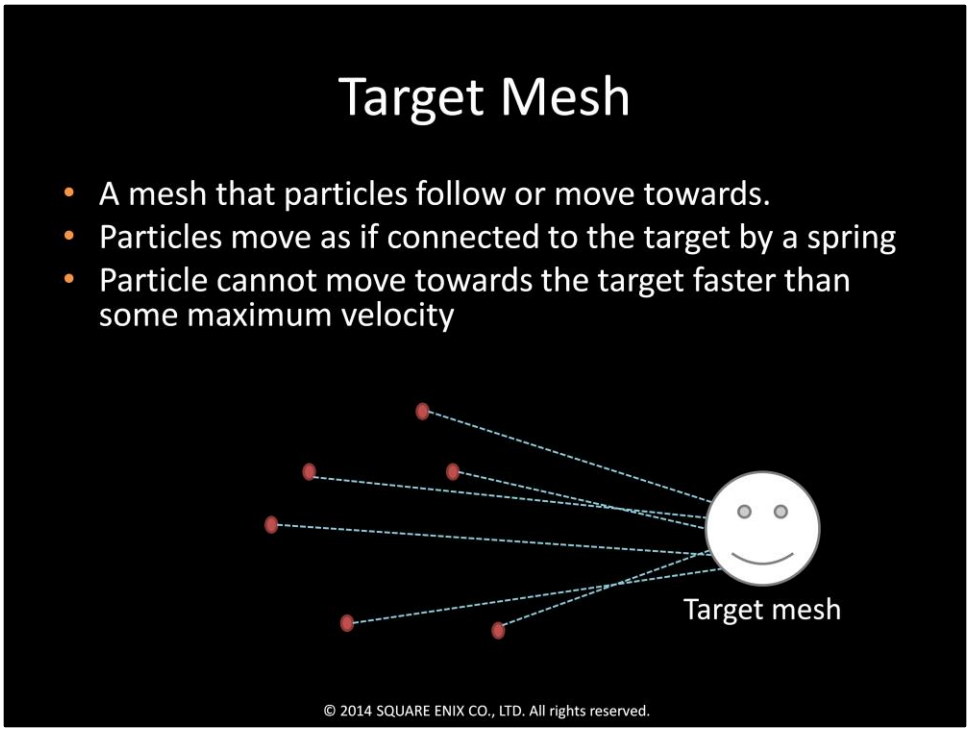
We also added a feature for collision handling. We allow particles to interact with some simple rigid objects by using repulsion forces and also a simple vector field.

Vector Field (3D Texture)

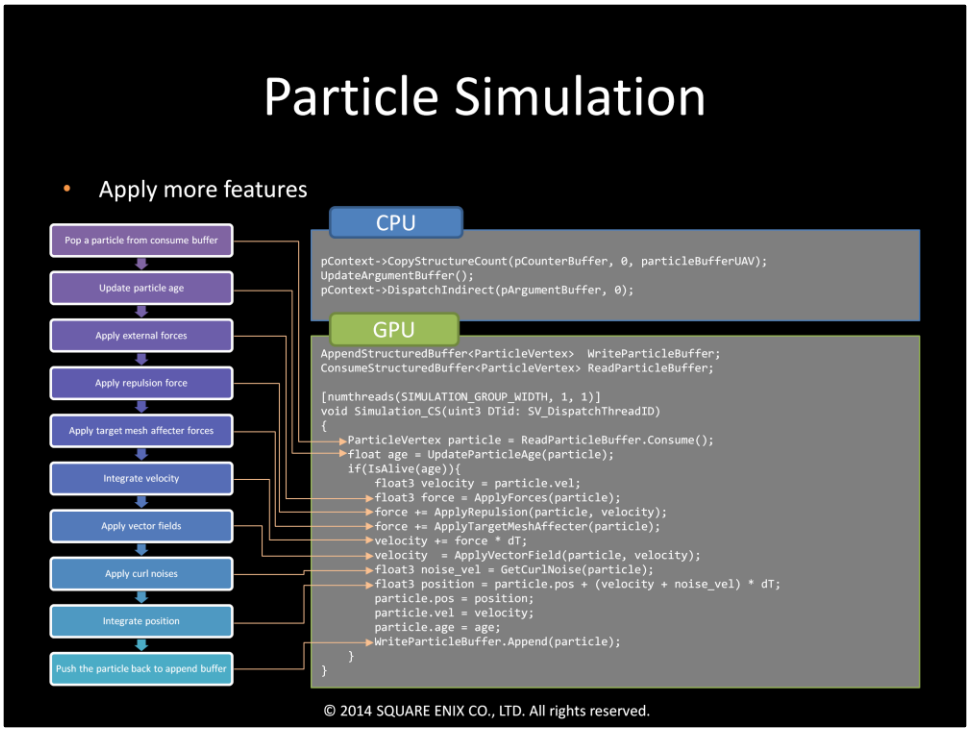
- Add extra movement to dynamic systems



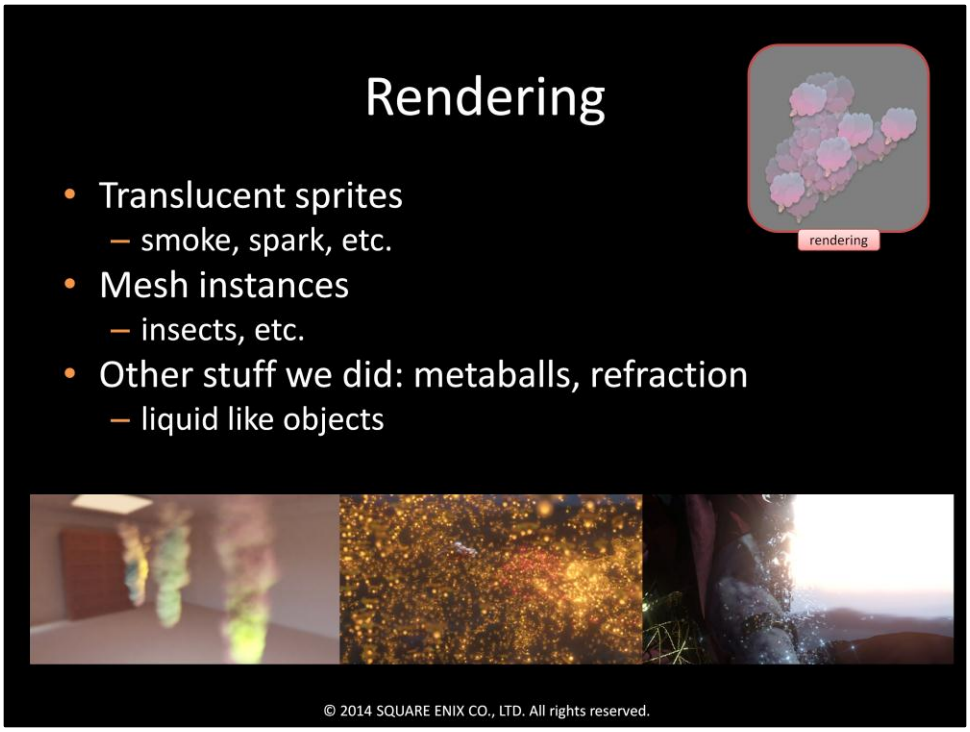
We also let the artists add some artistic movement using 3D Texture.



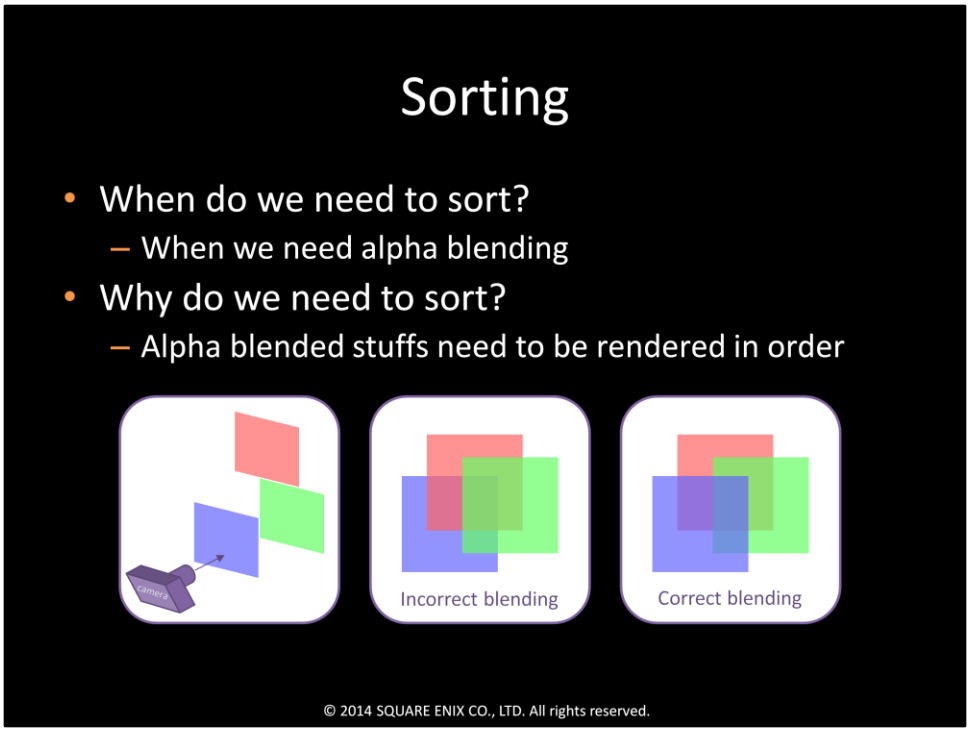
In addition, we also added a target mesh feature to make particles follow some specific mesh.
What we did is very simple, just adding some spring force between each particle and some position on the specified mesh.
However, if a particle is very far from its target, its spring force will be very strong, and it will make the particle move too fast.
To solve this problem, we added a parameter to let the artists set the maximum velocity that a particle can move.



When we put everything together by adding each feature to the shader sequentially, then the simulation shader will be more complex like this.

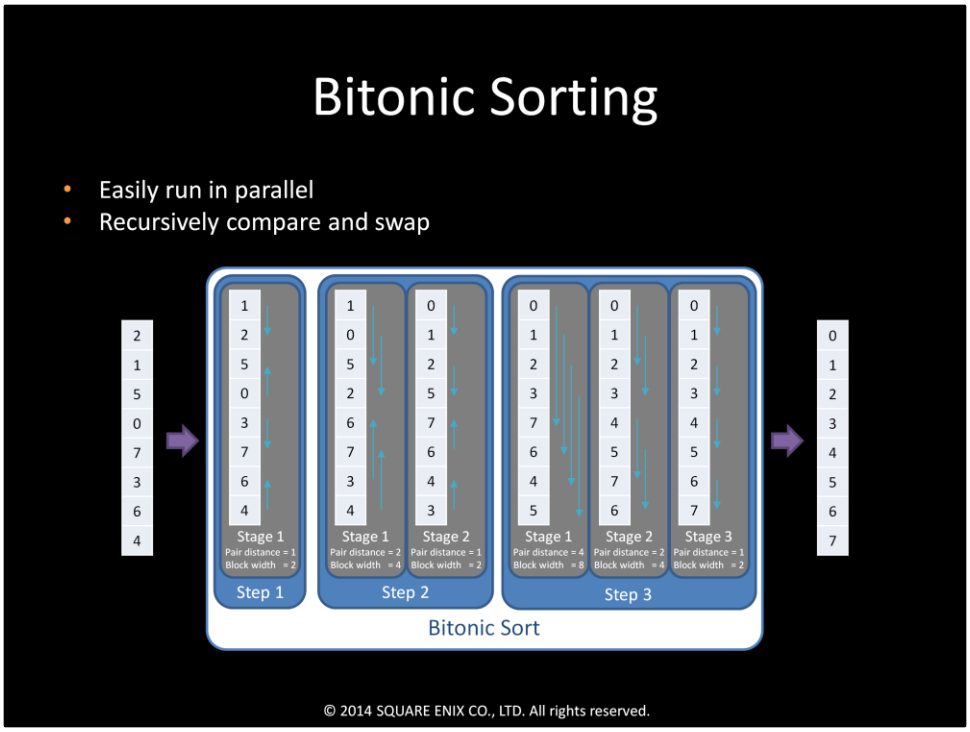


In Agni's Philosophy, we had 3 main types of particle rendering:
The first one is translucent sprites for smoke, sparks and some other effects.
The second one is mesh instance for insects, and the third one is metaballs with refraction for liquid like objects.



For translucent sprites, to get the correct blending, we need to sort particles from back to front.

Why we need sorting?
The purple sprite is closest to the camera, then the green and the red sprite.
However, if we render these sprites in random order, for example, purple, then red and green, the result from alpha blending will make us lose the perception of which object is closer to the camera than others.
So, it will look like the purple one is at the rear.
But, if we render them in order from back to front, that is red, green and then purple, we'll get the correct perception.



For sorting, we chose bitonic sort, since it can be easily run in parallel, and also is not very difficult to implement.
Bitonic sort is a comparison-based algorithm.
The idea of bitonic sort is to recursively sort 2 subsequences by comparing and swapping, then merge the results to get a sorted sequence.
The size of buffer should be a power of 2.
Anyway, since the worst case complexity of bitonic sorting is O(n * log(n) * log(n)), when the size of array is very huge, it will take a lot of time to sort.
In our demo, we didn't have a lot of translucent objects. Only smoke needed alpha blending. So, bitonic sorting was enough.

Bitonic Sorting

CPU

```
uint32 max_num = GetClosetPowerOfTwo(numParticles);
// calculate the number of stages we need
uint32 numSteps = 0;
for(LmUint32 temp = max_num; temp > 1; temp >>= 1){
    ++numSteps;
}

// calculate the number of thread groups we need
// Bitonic sort does N/2 comparisons and swaps per step
const uint32 n_to_dispatch = max_num / 2;
uint32 numGroups = (n_to_dispatch + GROUP_WIDTH - 1) / GROUP_WIDTH;

for(uint32_t step = 0; step < numSteps; ++step) {
    for(uint32 stage = 0; stage < step + 1; ++stage) {
        UpdateAndBindSortParameter(step, stage);
        pContext->Dispatch(numGroups, 1, 1);
    }
}
```

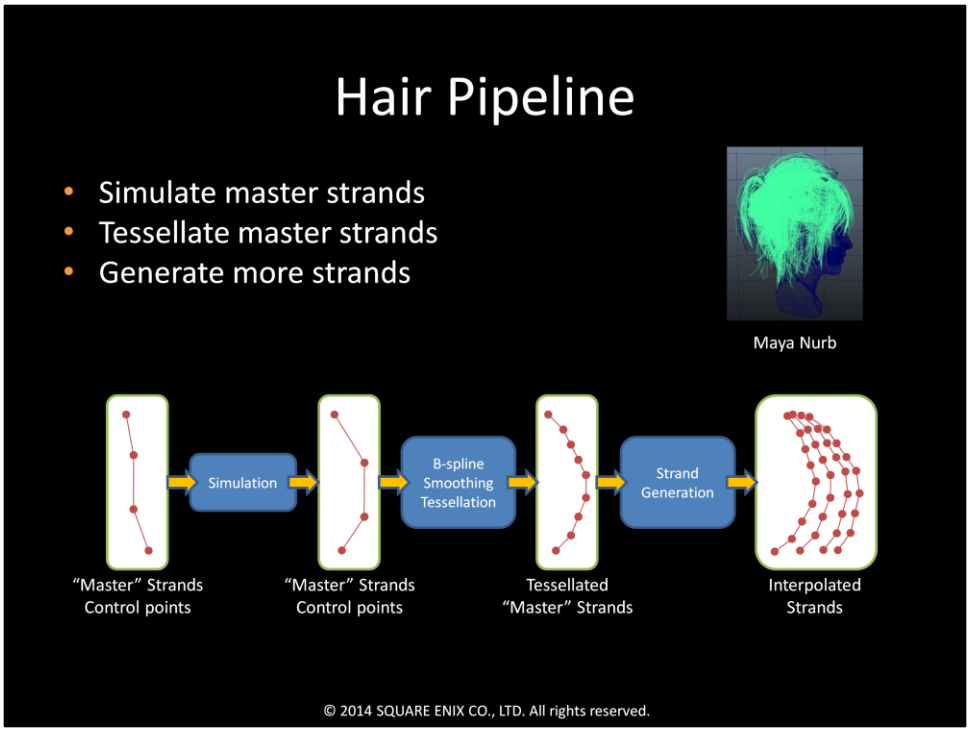
GPU

```
RWStructuredBuffer<SortData>  SortBuffer;

[numthreads(GROUP_WIDTH, 1, 1)]
void Sort_CS(uint3 DTid: SV_DispatchThreadID)
{
    uint thread_id = DTid.x;

    uint block_offset = (thread_id / PairDistance) * BlockWidth;
    uint left_id  = block_offset + (thread_id % PairDistance);
    uint right_id = left_id + PairDistance;

    SortData left_data  = SortBuffer[left_id];
    SortData right_data = SortBuffer[right_id];

    uint same_dir_block_width = 1 << Step;
    uint increasing_order = (thread_id / same_dir_block_width) % 2);

    CompareAndSwap(increasing_order, left_data, right_data);

    SortBuffer[left_id]  = left_data;
    SortBuffer[right_id] = right_data;
}
```


And, here is our pioneer source code for bitonic sorting, without any optimization. Actually, we also used this version in the demo you've seen.

GPU Compute for Hair

Let’s see the result!!



The pipeline of our hair is, first, we use Maya to create master hair strands as nurb curves.
Then, we export those control points from Maya, and import them to our run-time, and call them master strands.
Then we simulate these master strands, and tessellate them to create smoother strands.
After that, we generate some more strands and make them camera-aligned ribbons before rendering them.

Hair Data Structures

Control point positions

p0	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11	p12

Control point velocities

v0	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12

Rest lengths

l0	l1	l2	0	l4	l5	l6	l7	0	l9	l10	l11	0

Coordinate frames

c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12

Angular stiffness

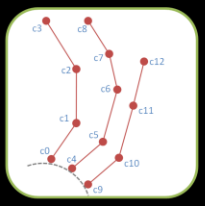
s0	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12

Rest vector in coordinate frames

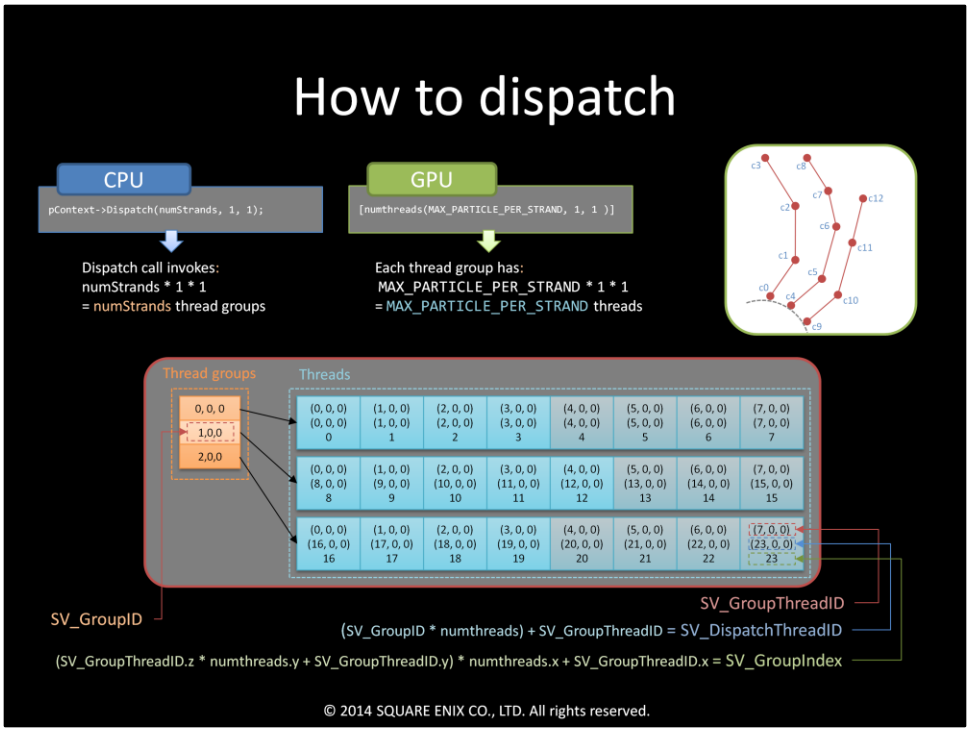
r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12

Number of control points in each strand

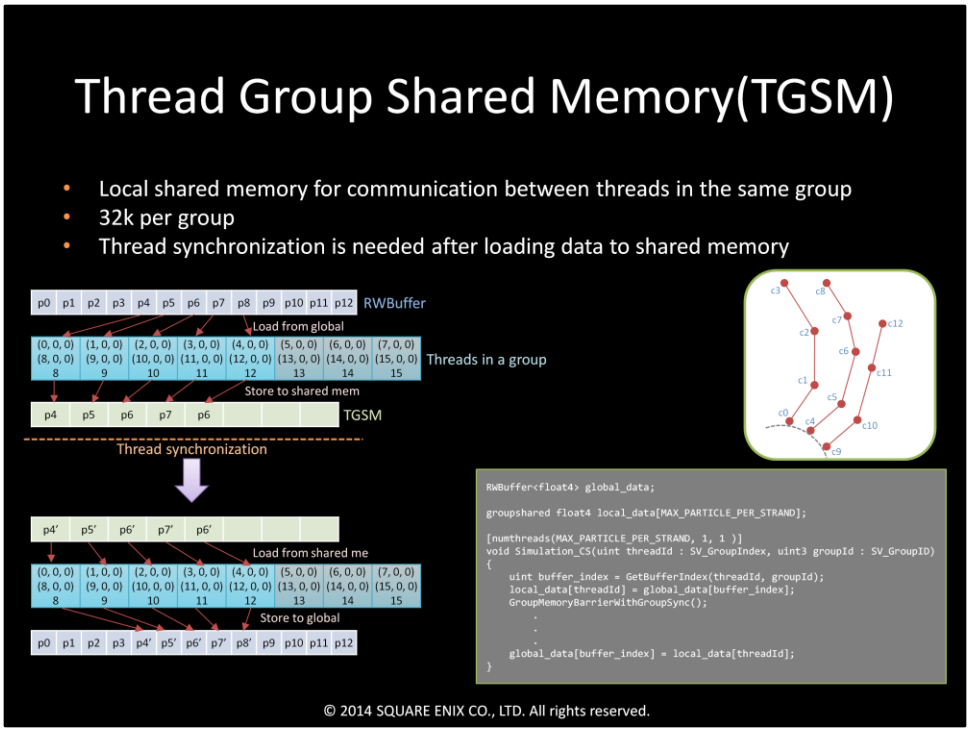
4	5	4



Unlike particles, where we use array of structure to store the data, for hair we decided to store each kind of hair data in different buffers.
Doing so gave us more flexibility to add or remove some types of data.
And, actually, at that time we kept adding and removing some types of data to get a good simulation result.
Here is the list of data that we end up with.
Most of the buffers have the size of the number of vertices.
And, since the number of vertices in each strand can be varied, we also need to store the number of vertices in a buffer.



Since the number of strands and also the number of particles in each strand won't change, the number of thread groups we dispatch will be the same every time.
For simplicity, we will use MAX_PARTICLE_PER_STRAND = 8 here. But in the actual shader, it should be multiple of warp/wavefront size.
Without hair-hair interaction, each strand can be updated individually. However, we can't update each vertex individually since the movement of its neighbors will affect its behavior.
The easiest way is to use one thread group for one strand. The data could be shared inside a group, so we will still be able to access their neighbors' data.
On the CPU side, we call dispatch with the number of strands we have, and on the GPU side, we make the number of threads per group equal to a maximum number of vertices per strand.
So, the number of threads generated per dispatch call is the number of strands * the maximum number of vertices per strand.

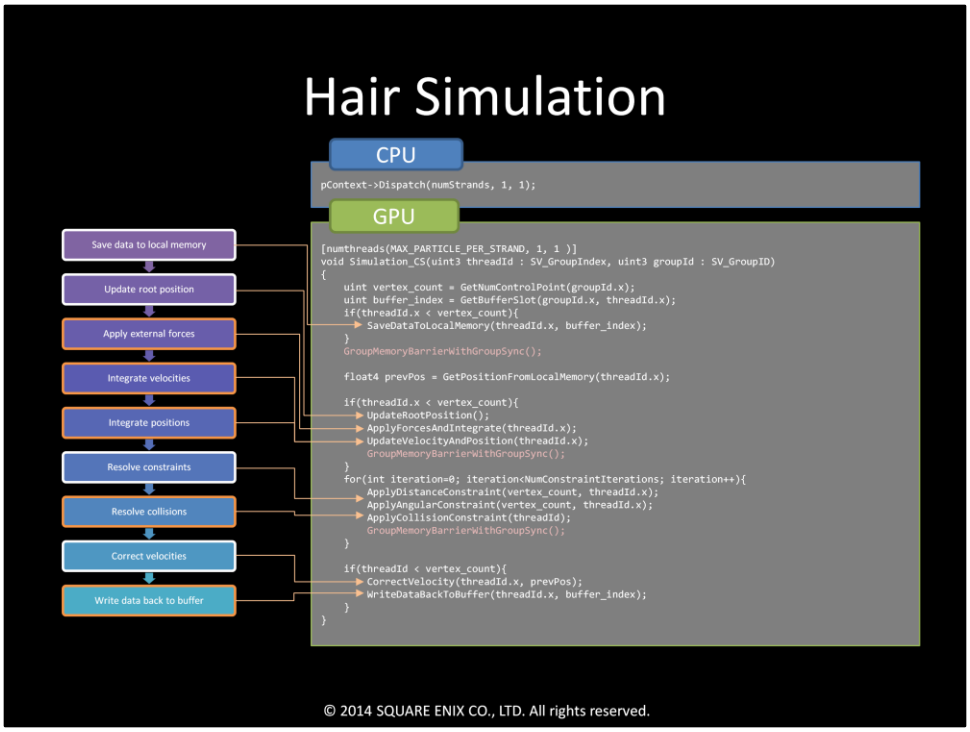


To update a hair strand, we need not only the data of a vertex, but also its neighbor's. Accessing global memory is pretty slow, so we shouldn't access it every time we want to read or update the data among the process.

A better way is to load the data we need to local memory and then process the data using local memory.

After we finish all the updating processes, we will load the data from local memory and store them back to global.

Don't forget to sync the threads every time we load or update data to local memory; otherwise, some threads might get the wrong data.



The hair simulation shader will look like this. The orange steps, i.e., apply external forces, integrate velocities and positions, resolve collisions and write data back to buffer, are the same as particle simulation.
We added a few more steps to make it compatible with hair.
For example, since the hair strands are attached to a scalp, we have to update the root positions accordingly.
Also, since particles in the strands should keep the same distance or maybe angle to its neighbors, we have to apply some constraints to them.

Distance Constraint Resolving

Sequential

- From root to tip
- Less stretch, less stable

Parallel

- Split into batches
- More stretch, more stable


```
void ApplyDistanceConstraint(uint numVertices, uint threadId)
{
    // since it is sequential,
    // we  will let only one thread do all the task
    if(threadId == 0){
        for(uint i=0; i<numVertices - 1; ++i){
            ApplyDistanceConstraint(i);
        }
    }
    GroupMemoryBarrierWithGroupSync();
}
```

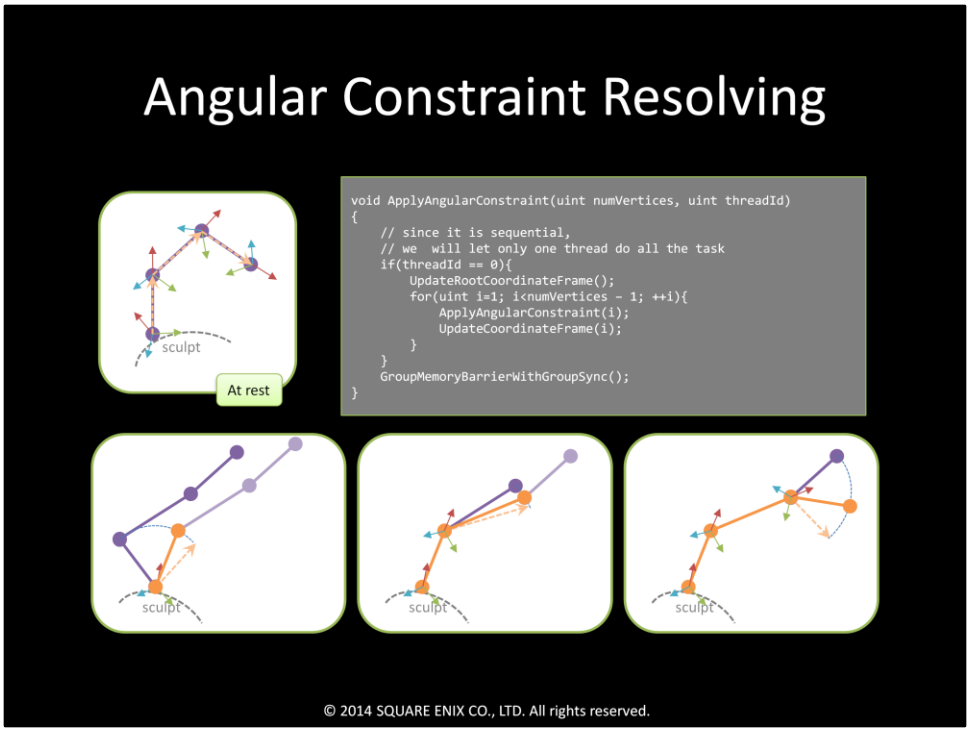
```
void ApplyDistanceConstraint(uint numVertices, uint threadId)
{
    uint first_half = numVertices >> 1;
    uint second_half = (numVertices - 1) >> 1;
    // 1st batch
    if(threadId < first_half){
        ApplyDistanceConstraint(threadId * 2);
    }
    GroupMemoryBarrierWithGroupSync();
    // 2nd batch
    if(threadId < second_half){
        ApplyDistanceConstraint(threadId * 2 + 1);
    }
    GroupMemoryBarrierWithGroupSync();
}
```

*Note: This is based on the time we implemented. However, there's a paper from Muller to resolve the stability problem of sequential method (a bit after we released our Agni). Try checking "Fast Simulation of Inextensible Hair and Fur" [Muller *et al.* 2012].

To keep the length of the strand stable, we apply some distance constraints to the particles.
We tried both sequential and parallel resolving methods.
For the sequential method, we resolve the constraint from root to tip. The strand won't stretch with this type of updating.
However, since instead of moving vertices at both ends of the link equally, only the vertex at the end further from the root is moving at any given time.
Vertices further from the root will have more and more kinetic energy as the distance increases and will swing around. So, it didn't look so stable.

With the parallel method, because we can't read and write to the same position at the same time, we had to separate the updating into 2 sets or batches.
The strand is now stable, but now it could stretch a lot even after updating, so we needed a few more iterations to make the result less stretchy.

Anyway, after we finished the project, there was new research published about how to solve the stability problem using the sequential method. I put a note down here. You can check it later if you're interested.

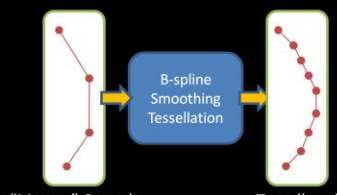


To control the bending of hair strands, we use angular constraint.

Coordinate frame is a set of local axes at each vertex.
The x axis of the vertex points along the tangent direction of its parent.
In the case of the root vertex, the x axis will point along the normal vector of the sculpt where the hair strand grows.
The y and z directions are generated once before the simulation starts, then we will update them every frame based on the x axis direction and its parent's y and z directions.
The coordinate frame at the root vertices always stay the same relative to the sculpt.
To control the bending, we calculate the rest local position relative to coordinate frame for each vertex, and try to move them back.
These coordinate frames are also used when we create wisps around master strands for rendering.

Master Strand Smoothing

- Done with a Compute Shader
  - Could certainly use tessellation
- We use B-Splines
  - Do not interpolate end points



"Master" Strands
Control points

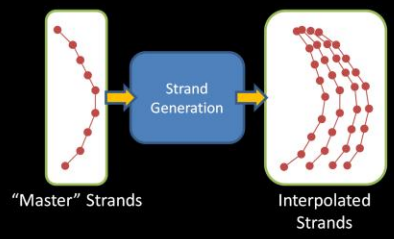
Tessellated
"Master" Strands

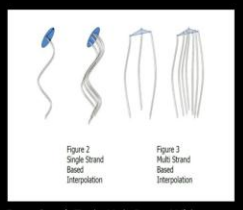
$$x(t) = \frac{1}{6}\begin{bmatrix} P_0 & P_1 & P_2 & P_3 \end{bmatrix}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

After we get the result from the simulation, we use a CS to interpolate the master strand using a b-spline interpolator, so that we can have smooth strands that are ready to be rendered.
This step can also be done using tessellation in hull and domain shaders, but in Agni's Philosophy we used a compute shader.

Generating more strands

- Done with a Compute Shader (PS4 version)
  - Could certainly use tessellation
- Several complementary ways to create the strands *(see Sarah Tariq Hair Demo)*
  - Single strand interpolation
  - Multi-strand interpolation
- Also possible to jitter, twist, etc.



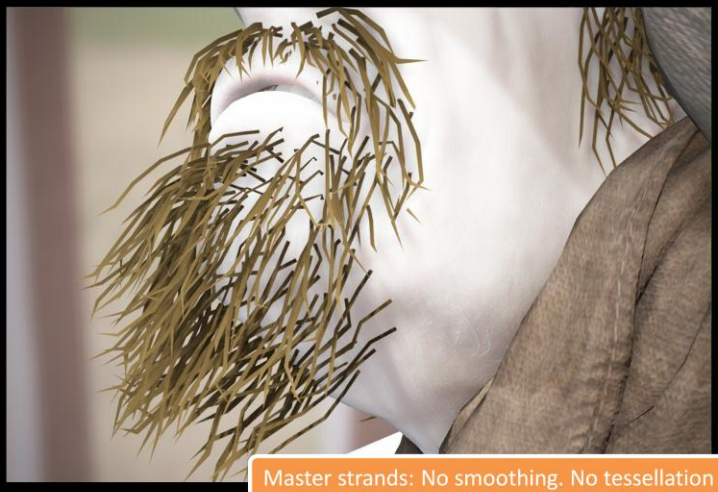


From Sarah Tariq Hair Demo Whitepaper

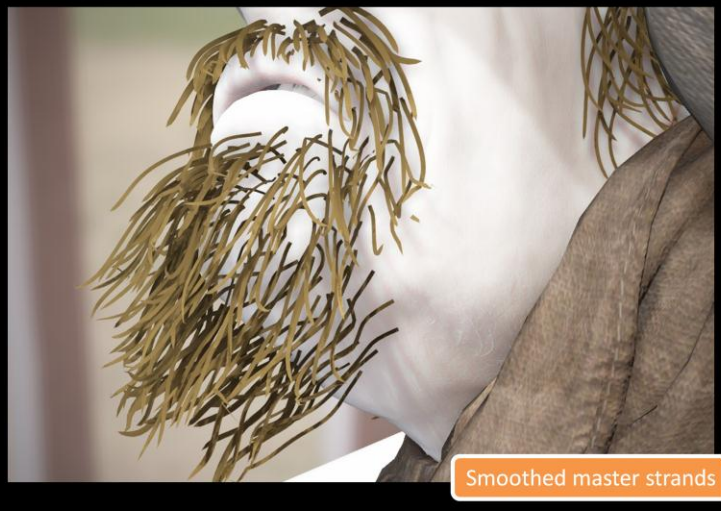
We also use CS to generate more strands to make the result look more realistic.
Same as the previous step, this step can be done using tessellation. Also this step could be swapped with the previous step.
In the case of Agni's Philosophy, we smooth master strands using CS, and do this step using tessellation on PC.
However, on PS4, tessellation was more expensive than we expected, so we switched to generating more strands using CS first, then smooth the generated strands using tessellation.

Strand Rendering

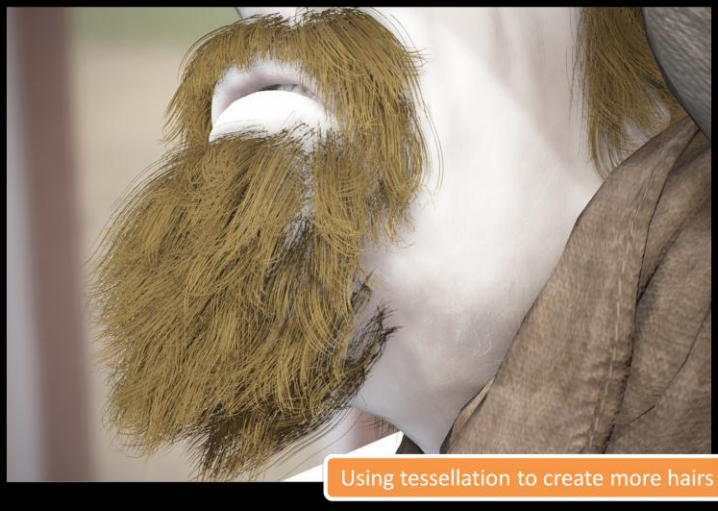
Strand Rendering



Strand Rendering

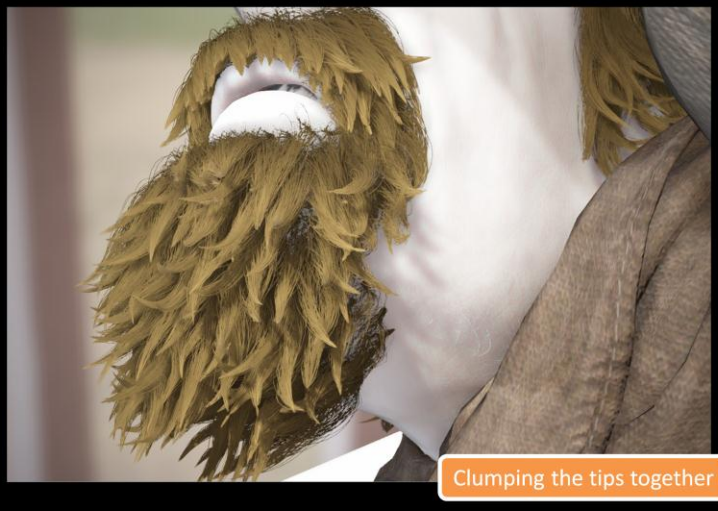


Strand Rendering



Strand Rendering

Strand Rendering



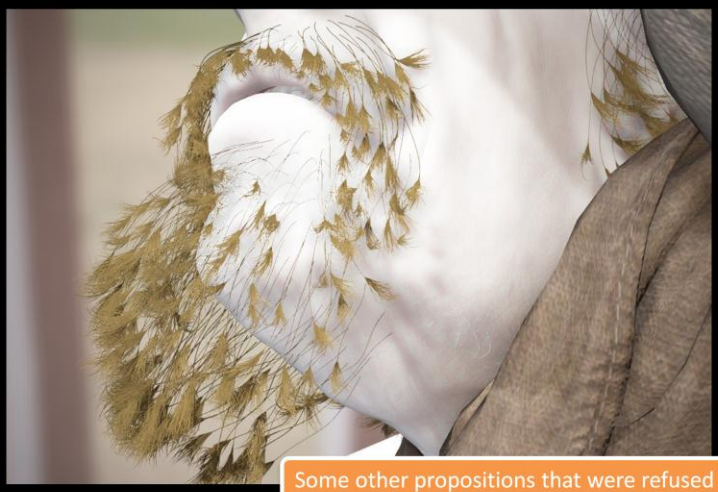
Strand Rendering

Strand Rendering

Strand Rendering

Some other propositions that were refused

Strand Rendering

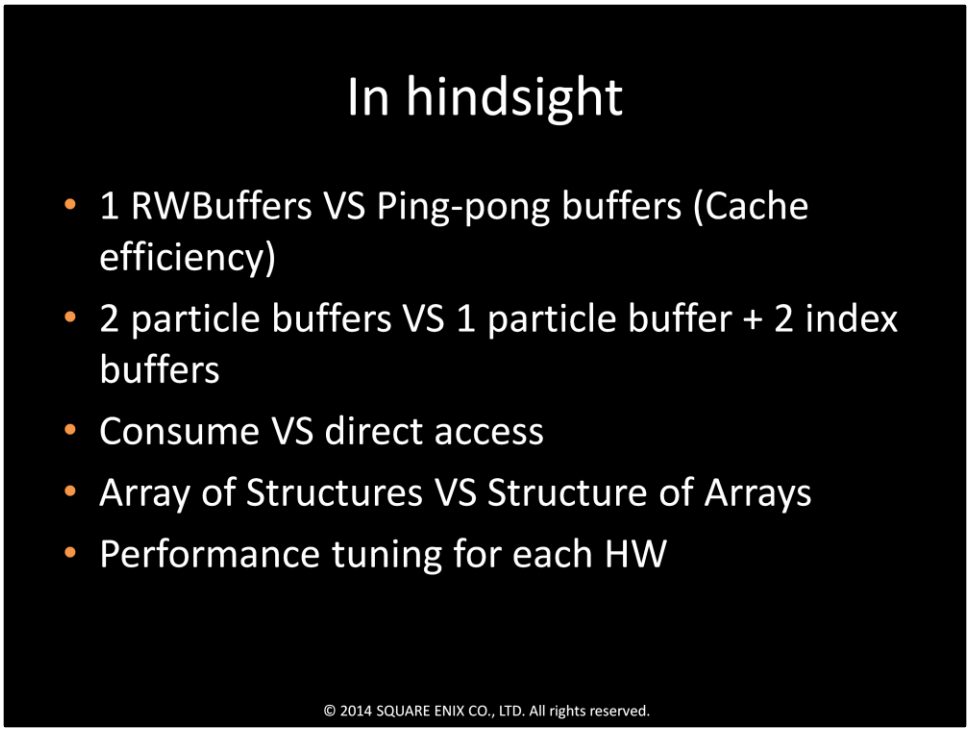


Some other propositions that were refused

Strand Rendering

Some other propositions that were refused

Demo



In hindsight, we didn't have time to try a lot of things. Here are some examples of them.
For sorting, we used only 1 RW buffer; however, my colleague suggested that using 2 buffers, one for read and one for write, then swapping them might get better efficiency.
For particle simulation, we used 2 structured buffers. Every time we updated, we popped and pushed the whole structure. If the structure is large, it would mean we have to access more memories. So, using index buffers, and doing push and pop on index buffers instead might get better performance. Anyway, using index buffers means we have to read an index from a buffer first, then we can read the correct particle data. So, it might be worth trying only when the structure size is large enough.
And, actually, consuming shouldn't be needed. We can declare a read-only structured buffer instead of a consume structured buffer in the shader code, and access the buffer directly by index instead of consuming the buffer.
Also, since we used structured buffers, each buffer was an array of structures. However, we thing that using an array layout structure might perform better because of more cache hits.
Anyway, the most important thing we have learnt is we need to profile everything. Different HW architectures might get different results. So, if we have to work on more than one platform, we might have to tune each platform differently.

Thank you very much for your attention!!