Efficient usage of compute shaders on Xbox One and PS4

Alexis Vaisse
Lead Programmer – Ubisoft Montpellier
Motion Cloth

• Cloth simulation developed by Ubisoft

• Used in:

  - Child of Light
  - Assassins's Creed
  - Far Cry 4
  - Rainbow Six Siege
  - The Division
Agenda

• What is this talk about?
• Why porting a cloth simulation to the GPU?
• The first attempts – A new approach
• The shader – Easy parts – Complex parts
• Optimizing the shader
• The PS4 version
• What you can do & cannot do in compute shader
• Tips & tricks
What is this talk about?

- Cloth simulation ported to the GPU
- For PC DirectX 11, Xbox One and PS4
What is this talk about?

• Cloth simulation ported to the GPU
• For PC DirectX 11, Xbox One and PS4

• This talk is about all that we have learned during this adventure
• What is this talk about?
• Why porting a cloth simulation to the GPU?
• The first attempts
• A new approach
• The shader – Easy parts – Complex parts
• Optimizing the shader
• The PS4 version
• What you can do & cannot do in compute shader
• Tips & tricks
Why porting a cloth simulation to the GPU?

5 ms of CPU time

| # of dancers | Xbox360 | 34 |
Why porting a cloth simulation to the GPU?

5 ms of CPU time

<table>
<thead>
<tr>
<th></th>
<th># of dancers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xbox360</td>
<td>34</td>
</tr>
<tr>
<td>PS3</td>
<td>105</td>
</tr>
</tbody>
</table>

SPUs rock!
Why porting a cloth simulation to the GPU?

5 ms of CPU time

<table>
<thead>
<tr>
<th></th>
<th># of dancers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xbox360</td>
<td>34</td>
</tr>
<tr>
<td>PS3</td>
<td>105</td>
</tr>
</tbody>
</table>

Now let’s switch to next gen!
Why porting a cloth simulation to the GPU?

5 ms of CPU time

<table>
<thead>
<tr>
<th></th>
<th># of dancers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xbox360</td>
<td>34</td>
</tr>
<tr>
<td>PS3</td>
<td>105</td>
</tr>
<tr>
<td>PS4</td>
<td>98</td>
</tr>
</tbody>
</table>
Why porting a cloth simulation to the GPU?

5 ms of CPU time

<table>
<thead>
<tr>
<th># of dancers</th>
<th>Xbox360</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS3</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>PS4</td>
<td>98</td>
<td></td>
</tr>
</tbody>
</table>

5 SPUs @ 3.2 GHz

6 cores @ 1.6 GHz
Why porting a cloth simulation to the GPU?

5 ms of CPU time

<table>
<thead>
<tr>
<th># of dancers</th>
<th>Xbox360</th>
<th>PS3</th>
<th>PS4</th>
<th>Xbox One</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xbox360</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PS3</td>
<td>105</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PS4</td>
<td>98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xbox One</td>
<td>113</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why porting a cloth simulation to the GPU?

Next gen doesn’t look sexy!

![Bar chart comparing Xbox360, PS3, PS4 CPU, and Xbox One CPU performance.](chart.png)
What is the solution?
Why porting a cloth simulation to the GPU?

Peak power: \textbf{Xbox One} \quad \textbf{PS4}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{chart.png}
\end{figure}
• What is this talk about?
• Why porting a cloth simulation to the GPU?
• The first attempts
• A new approach
• The shader – Easy parts – Complex parts
• Optimizing the shader
• The PS4 version
• What you can do & cannot do in compute shader
• Tips & tricks
The first attempts

Easy to use

Not available on all platforms
The first attempts

+ Easy to use
- Not available on all platforms

+ Close to C++
- Black box: no possibility to know what’s going on

DirectCompute
The first attempts

Integrate velocity

Resolve some constraints

Resolve collisions

Resolve some more constraints

Do some other funny stuffs

...
The first attempts

Integrate velocity
Resolve some constraints
Resolve collisions
Resolve some more constraints
Do some other funny stuffs

...
The first attempts

Too many “Dispatch”
The first attempts

Too many “Dispatch”

Bottleneck = CPU
The first attempts

Merge several cloth items to get better performance

All cloth items must have the same properties
• What is this talk about?
• Why porting a cloth simulation to the GPU?
• The first attempts
• A new approach
  • The shader – Easy parts – Complex parts
  • Optimizing the shader
  • The PS4 version
• What you can do & cannot do in compute shader
• Tips & tricks
A new approach

- A single huge compute shader to simulate the entire cloth
- Synchronization points inside the shader
- A single “Dispatch” instead of 50+
A new approach

- A single huge compute shader to simulate the entire cloth
- Synchronization points inside the shader
- A single “Dispatch” instead of 50+
- Simulate several cloth items (up to 32) using a single “Dispatch”
• What is this talk about?
• Why porting a cloth simulation to the GPU?
• The first attempts
• A new approach
• The shader – Easy parts – Complex parts
• Optimizing the shader
• The PS4 version
• What you can do & cannot do in compute shader
• Tips & tricks
The shader

- 41 .hlsl files
- 3,100 lines of code
  (+ 800 lines for unit tests & benchmarks)
- Compiled shader code size = 69 KB
The shader – Easy parts

• Thread group:

  0 1 2 3 4 5  63

• We do the same operation on 64 vertices at a time

There must be no dependency between the threads
The shader – Easy parts

Read some global properties to apply (ex: gravity, wind)

- Read position of vertex 0
- Read position of vertex 1
- ... Read position of vertex 63
The shader – Easy parts

Read some global properties to apply (ex: gravity, wind)

Read position of vertex 0

Read position of vertex 1

... Read position of vertex 63

Compute

Compute

... Compute

Write position of vertex 0

Write position of vertex 1

... Write position of vertex 63
The shader – Easy parts

Read some global properties to apply (ex: gravity, wind)

Read position of vertex 64

Read position of vertex 65

Read position of vertex 127

Compute

Compute

Compute

Write position of vertex 64

Write position of vertex 65

Write position of vertex 127
The shader – Easy parts

- Read position of vertex 0
- Read property for vertex 0

- Read position of vertex 1
- Read property for vertex 1

- Read position of vertex 63
- Read property for vertex 63
The shader – Easy parts

- Read position of vertex 0
- Read property for vertex 0
- Compute
- Write position of vertex 0
- Read position of vertex 1
- Read property for vertex 1
- Compute
- Write position of vertex 1
- ...
The shader – Easy parts

Read property for vertex 0  Read property for vertex 1  ...  Read property for vertex 63

Ensure contiguous reads to get good performance
The shader – Easy parts

Read property for vertex 0
Read property for vertex 1
... Read property for vertex 63

Ensure contiguous reads to get good performance

Coalescing = 1 read instead of 16
i.e. use Structure of Arrays (SoA) instead of Array of Structures (AoS)
The shader – Complex parts

• Binary constraints:

Constraint

Vertex A  Vertex B
The shader – Complex parts

• Binary constraints:
The shader – Complex parts

- Binary constraints:
The shader – Complex parts

- Binary constraints:
The shader – Complex parts

• Binary constraints:
The shader – Complex parts

• Binary constraints: Group 1
The shader – Complex parts

• Binary constraints:

Group 1

Group 2
The shader – Complex parts

• Binary constraints:
The shader – Complex parts

• Binary constraints:

Group 1
GroupMemoryBarrierWithGroupSync()

Group 2
GroupMemoryBarrierWithGroupSync()

Group 3
GroupMemoryBarrierWithGroupSync()

Group 4
The shader – Complex parts

• Collisions: Easy or not?
  • Collisions with vertices Easy
The shader – Complex parts

- Collisions: Easy or not?
  - Collisions with vertices: Easy
  - Collisions with triangles
    - Each thread will modify the position of 3 vertices
    - You have to create groups and add synchronization
• What is this talk about?
• Why porting a cloth simulation to the GPU?
• The first attempts
• A new approach
• The shader – Easy parts – Complex parts

• Optimizing the shader
• The PS4 version
• What you can do & cannot do in compute shader
• Tips & tricks
Optimizing the shader

• General rule:

\[ \text{Bottleneck} = \text{memory bandwidth} \]

• Data compression:

<table>
<thead>
<tr>
<th>CPU</th>
<th>128 bits (4 floats)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td></td>
</tr>
</tbody>
</table>
Optimizing the shader

• General rule:

Bottleneck = memory bandwidth

• Data compression:

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>128 bits (4 floats)</td>
</tr>
<tr>
<td>Normal</td>
<td>128 bits (4 floats)</td>
</tr>
</tbody>
</table>
Optimizing the shader

- General rule:

  Bottleneck = memory bandwidth

- Data compression:

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>128 bits (4 floats)</td>
<td>64 bits (21:21:21:1)</td>
</tr>
<tr>
<td>Normal</td>
<td>128 bits (4 floats)</td>
<td></td>
</tr>
</tbody>
</table>
Optimizing the shader

• General rule:

Bottleneck = memory bandwidth

• Data compression:

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>128 bits (4 floats)</td>
<td>64 bits (21:21:21:1)</td>
</tr>
<tr>
<td>Normal</td>
<td>128 bits (4 floats)</td>
<td>32 bits (10:10:10)</td>
</tr>
</tbody>
</table>
Optimizing the shader

- Use Local Data Storage (aka Local Shared Memory)

Compute Unit
(12 on Xbox One, 18 on PS4)
Optimizing the shader

- Store vertices in Local Data Storage

Copy vertices from VRAM to LDS
Optimizing the shader

- Store vertices in Local Data Storage
  
  Copy vertices from VRAM to LDS
  
  Step 1 – Update vertices
  
  Step 2 – Update vertices
  
  ... 
  
  Step n – Update vertices
  
  Copy vertices from LDS to VRAM

![Bar chart showing VRAM vs. LDS]
Optimizing the shader

- Use bigger thread groups

0 1 2 3 4 5 6 3 ...

Load
Wait
Compute
Optimizing the shader

- Use bigger thread groups

0 1 2 3 4 5 6 3

Load
Wait
Compute
Load
Wait
Compute
Optimizing the shader

• Use bigger thread groups
Optimizing the shader

• Use bigger thread groups

With 256 or 512 threads, we hide most of the latency!
Optimizing the shader

Dummy vertices
Optimizing the shader

Dummy vertices = Useless work!
Optimizing the shader
Optimizing the shader

0 1 2 3 4 5  63 64  127 128  191 192  255
Optimizing the shader

Performance (higher = better)

Cloth's vertices

- 64
- 128
- 256
- 512
Optimizing the shader

Performance (higher = better)

Cloth’s vertices

64-thread group

128-thread group

64
128
256
512

64
128
256
512
Optimizing the shader

![Graph showing performance vs. number of threads and cloth vertices.]
• What is this talk about?
• Why porting a cloth simulation to the GPU?
• The first attempts
• A new approach
• The shader – Easy parts – Complex parts
• Optimizing the shader
• The PS4 version
• What you can do & cannot do in compute shader
• Tips & tricks
The PS4 version

- Port from HLSL to PSSL

```c
#ifdef __PSSL__
    #define numthreads NUM_THREADS
    #define SV_GroupIndex S_GROUP_INDEX
    #define SV_GroupID S_GROUP_ID
    #define StructuredBuffer RegularBuffer
    #define RWStructuredBuffer RW_RegularBuffer
    #define ByteAddressBuffer ByteBuffer
    #define RWByteAddressBuffer RW_ByteBuffer
    #define GroupMemoryBarrierWithGroupSync ThreadGroupMemoryBarrierSync
    #define groupshared thread_group_memory
#endif
```
The PS4 version

- On DirectX 11:
The PS4 version

- On DirectX 11:

  1. Compute shader
  2. Compute shader
  3. CopyResource

Synchronization
The PS4 version

• On PS4:

  - No implicit synchronization, no implicit buffer duplication
  - You have to manage everything by yourself
  - Potentially better performance because you know when you have to sync or not
The PS4 version

- We use labels to know if a buffer is still in use by the GPU

- Still used → Automatically allocate a new buffer

- “Used” means used by a compute shader or a copy

- We also use labels to know when a compute shader has finished, to copy the results
• What is this talk about?
• Why porting a cloth simulation to the GPU?
• The first attempts
• A new approach
• The shader – Easy parts – Complex parts
• Optimizing the shader
• The PS4 version
• What you can do & cannot do in compute shader
• Tips & tricks
What you can do in compute shader

Peak power: **Xbox One**

![Graph comparing CPU and GPU performance on Xbox One](image1)

**Gflops**

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Peak power:</strong></td>
<td>15x</td>
<td></td>
</tr>
</tbody>
</table>

**PS4**

![Graph comparing CPU and GPU performance on PS4](image2)

**Gflops**

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Peak power:</strong></td>
<td>23x</td>
<td></td>
</tr>
</tbody>
</table>
What you can do in compute shader

+ Using DirectCompute, you can do almost everything in compute shader

- The difficulty is to get good performance
What you can do in compute shader

- Efficient code = you work on 64+ data at a time

```c
if (threadIndex < 32)
{
  ...
};
```

```c
if (threadIndex == 0)
{
  ...
};
```
What you can do in compute shader

- Efficient code = you work on 64+ data at a time

```cpp
if (threadIndex < 32) {
    ...
};

if (threadIndex == 0) {
    ...
};
```

// Read the same data on all threads
...

This is likely to be the bottleneck
What you can do in compute shader

- Example: collisions
- On the CPU:

  Compute a bounding volume (ex: Axis-Aligned Bounding Box)

  Use it for an early rejection test
What you can do in compute shader

- Example: collisions
- On the CPU:

  - Compute a bounding volume (ex: Axis-Aligned Bounding Box)
  - Use it for an early rejection test
  - Use an acceleration structure (ex: AABB Tree) to improve performance
What you can do in compute shader

- Example: collisions
- On the GPU:

  Compute a bounding volume
  (ex: Axis-Aligned Bounding Box)

  Just doing this can be more costly than computing the collision with all vertices!!!
What you can do in compute shader

- Compute 64 sub-AABoxes
What you can do in compute shader

- Compute 64 sub-AABoxes
What you can do in compute shader

- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes

We use only 32 threads for that
What you can do in compute shader

- Compute 64 sub-AABBoxes
- Reduce down to 32 sub-AABBoxes
- Reduce down to 16 sub-AABBoxes

We use only 16 threads for that
What you can do in compute shader

- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes
- Reduce down to 8 sub-AABoxes

We use only 8 threads for that
What you can do in compute shader

- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes
- Reduce down to 8 sub-AABoxes
- Reduce down to 4 sub-AABoxes

We use only 4 threads for that
What you can do in compute shader

- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes
- Reduce down to 8 sub-AABoxes
- Reduce down to 4 sub-AABoxes
- Reduce down to 2 sub-AABoxes

We use only 2 threads for that
What you can do in compute shader

- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes
- Reduce down to 8 sub-AABoxes
- Reduce down to 4 sub-AABoxes
- Reduce down to 2 sub-AABoxes
- Reduce down to 1 AABox

We use a single thread for that
What you can do in compute shader

- Compute 64 sub-AABoxes
- Reduce down to 32 sub-AABoxes
- Reduce down to 16 sub-AABoxes
- Reduce down to 8 sub-AABoxes
- Reduce down to 4 sub-AABoxes
- Reduce down to 2 sub-AABoxes
- Reduce down to 1 AABox

This is ~ as costly as computing the collision with 7 x 64 = 448 vertices!!
What you can do in compute shader

- Atomic functions are available
- You can write lock-free thread-safe containers
- Too costly in practice
What you can do in compute shader

- Atomic functions are available
  - You can write lock-free thread-safe containers

- Too costly in practice

The brute-force approach is almost always the fastest one
What you can do in compute shader

- Atomic functions are available
- You can write lock-free thread-safe containers
- Too costly in practice

The brute-force approach is almost always the fastest one

- Bandwidth usage
- Data compression
- Memory coalescing
- LDS usage
What you can do in compute shader

Port an algorithm to the GPU only if you find a way to handle 64+ data at a time 95+% of the time
• What is this talk about?
• Why porting a cloth simulation to the GPU?
• The first attempts
• A new approach
• The shader – Easy parts – Complex parts
• Optimizing the shader
• The PS4 version
• What you can do & cannot do in compute shader

• Tips & tricks
Sharing code between C++ & hlsl

```c
#if defined(_WIN32) || defined(_WIN64)
|| defined(_DURANGO) || defined(__ORBIS__)
typedef unsigned long uint;
struct float2 { float x, y; }
struct float3 { float x, y, z; }
struct float4 { float x, y, z, w; }
struct uint2 { uint x, y; }
struct uint3 { uint x, y, w; }
struct uint4 { uint x, y, z, w; }
#endif
```
Debug buffer

```c
struct DebugBuffer {
    ...
};
```
Debug buffer

```c
struct DebugBuffer {
    float3 m_Velocity;
    float  m_Weight;
};

// Uncomment the following line
// to use the debug buffer
#define USE_DEBUG_BUFFER

#if USE_DEBUG_BUFFER
    RWStructuredBuffer<DebugBuffer> g_DebugBuffer : register(u1);
#endif
```
Debug buffer

```cpp
struct DebugBuffer {
    float3 m_Velocity;
    float m_Weight;
};

// Uncomment the following line
// to use the debug buffer
#define USE_DEBUG_BUFFER

#if defined USE_DEBUG_BUFFER
    RWStructuredBuffer<DebugBuffer> g_DebugBuffer : register(u1);
#endif

WRITE_IN_DEBUG_BUFFER(m_Velocity, threadIdx, value);

DebugBuffer *debugBuffer = GetDebugBuffer();
```
What to put in LDS?

LDS → Yes → Random access?
What to put in LDS?

- LDS
  - Random access?
    - Yes
    - No
      - Contiguous access
  - Accessed several times?
    - Yes
    - No
      - VRAM
Memory consumption in LDS

- LDS = 64 KB per compute unit
- 1 thread group can access 32 KB
Memory consumption in LDS

- LDS = 64 KB per compute unit
- 1 thread group can access 32 KB

2 thread groups can run simultaneously on the same compute unit
Memory consumption in LDS

- LDS = 64 KB per compute unit
- 1 thread group can access 32 KB
- 2 thread groups can run simultaneously on the same compute unit
- Less memory used in LDS
- More thread groups can run in parallel
Memory consumption in LDS

- LDS = 64 KB per compute unit
- 1 thread group can access 32 KB
- 2 thread groups can run simultaneously on the same compute unit
- Less memory used in LDS
- More thread groups can run in parallel
Memory consumption in LDS

- LDS = 64 KB per compute unit
- 1 thread group can access 32 KB
- 2 thread groups can run simultaneously on the same compute unit
- Less memory used in LDS
- More thread groups can run in parallel
Optimizing bank access in LDS?

- LDS is divided into several banks (16 or 32)
- 2 threads accessing the same bank → Conflict
Optimizing bank access in LDS?

- LDS is divided into several banks (16 or 32)
- 2 threads accessing the same bank → Conflict

Visible impact on performance on older PC hardware

Negligible on Xbox One, PS4 and newer PC hardware
Beware the compiler

```c
CopyFromVRAMToLDS();
ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();
CopyFromLDSToVRAM();
```
Beware the compiler

```c
CopyFromVRAMToLDS();
ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

// CopyFromLDSToVRAM();
```
Beware the compiler

CopyFromVRAMToLDS();
ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();
CopyFromLDSToVRAM();

ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

The last copy takes all the time
This doesn’t make sense!
Beware the compiler

CopyFromVRAMToLDS();
ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();
ReadInputFromLDS();
DoSomeComputations();
WriteOutputToLDS();

//CopyFromLDSToVRAM();
float3 fanBlades[10];
for (uint i = 0; i < 10; ++i)
{
    Vertex fanVertex = GetVertexInLDS(neighborFan.m_VertexIndex[i]);
    fanBlades[i] = fanVertex.m_Position - fanCenter.m_Position;
}

float3 normalAccumulator = cross(fanBlades[0], fanBlades[1]);
for (uint j = 0; j < 8; ++j)
{
    float3 triangleNormal = cross(fanBlades[j+1], fanBlades[j+2]);
    uint isTriangleFilled = neighborFan.m_FilledFlags & (1 << j);
    if (isTriangleFilled) normalAccumulator += triangleNormal;
}
Optimizing compilation time

```cpp
float3 fanBlades[10];
for (uint i = 0; i < 10; ++i)
{
    Vertex fanVertex = GetVertexInLDS(neighborFan.m_VertexIndex[i]);
    fanBlades[i] = fanVertex.m_Position - fanCenter.m_Position;
}

float3 normalAccumulator = cross(fanBlades[0], fanBlades[1]);
for (uint j = 0; j < 8; ++j)
{
    float3 triangleNormal = cross(fanBlades[j+1], fanBlades[j+2]);
    uint isTriangleFilled = neighborFan.m_FilledFlags & (1 << j);
    if (isTriangleFilled) normalAccumulator += triangleNormal;
}
```
Optimizing compilation time

```cpp
float3 fanBlades[10];
for (uint i = 0; i < 10; ++i)
{
    Vertex fanVertex = GetVertexInLDS(neighborFan.m_VertexIndex[i]);
    fanBlades[i] = fanVertex.m_Position - fanCenter.m_Position;
}

float3 normalAccumulator = cross(fanBlades[0], fanBlades[1]);
for (uint j = 0; j < 8; ++j)
{
    float3 triangleNormal = cross(fanBlades[j+1], fanBlades[j+2]);
    uint isTriangleFilled = neighborFan.m_FilledFlags & (1 << j);
    if (isTriangleFilled) normalAccumulator += triangleNormal;
}
```

Shader compilation time

<table>
<thead>
<tr>
<th>Loop</th>
<th>19”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manually unrolled</td>
<td>6”</td>
</tr>
</tbody>
</table>
Iteration time

• It’s really hard to know which code will run the fastest.

• The “best” method:
  • Write 10 versions of your feature.
  • Test them.
  • Keep the fastest one.
Iteration time

• It’s really hard to know which code will run the fastest.

• The “best” method:
  • Write 10 versions of your feature.
  • Test them.
  • Keep the fastest one.

• A fast iteration time really helps
Bonus: final performance

Next gen can be sexy after all!
PS4 – 2 ms of GPU time – 640 dancers
Thank you!

Questions?