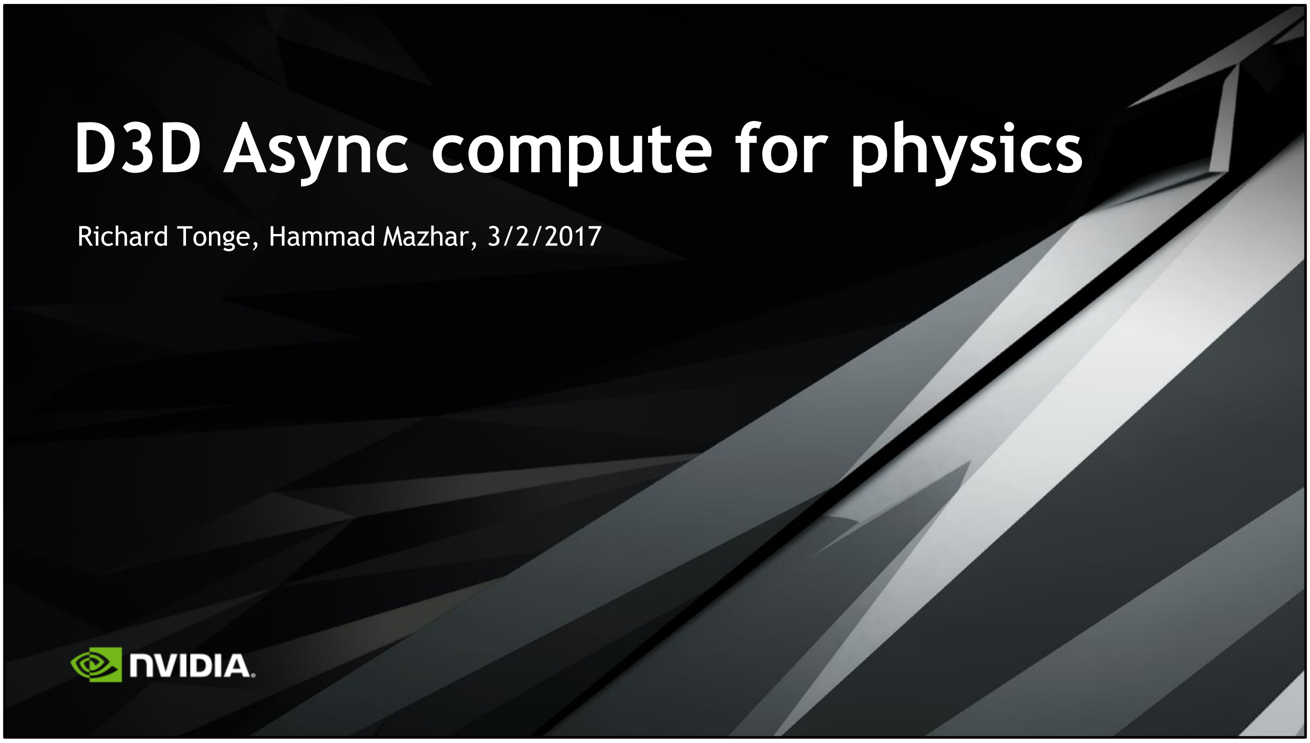


D3D Async compute for physics

Richard Tonge, Hammad Mazhar, 3/2/2017

Hello, my name is Richard, this is Hammad.

We’d like to share with you our experience at NVIDIA using D3D async compute to do inexpensive GPU particle simulation.

Our simulator is called Flex. You can use it for many purposes, from making the ground solid with respect to particles, to simulating rigid bodies, fluids or cloth.

The subtitle for this presentation is “Bullets, Bandages and Blood”, and the demo you are about to see shows how you can get these three effects from interacting particles.

Flex Demo

www.gameworks.nvidia.com

2

The first effect is bandages, which are strips of cloth.

Press key to display particle representation

Next is fluids, in this case blood.

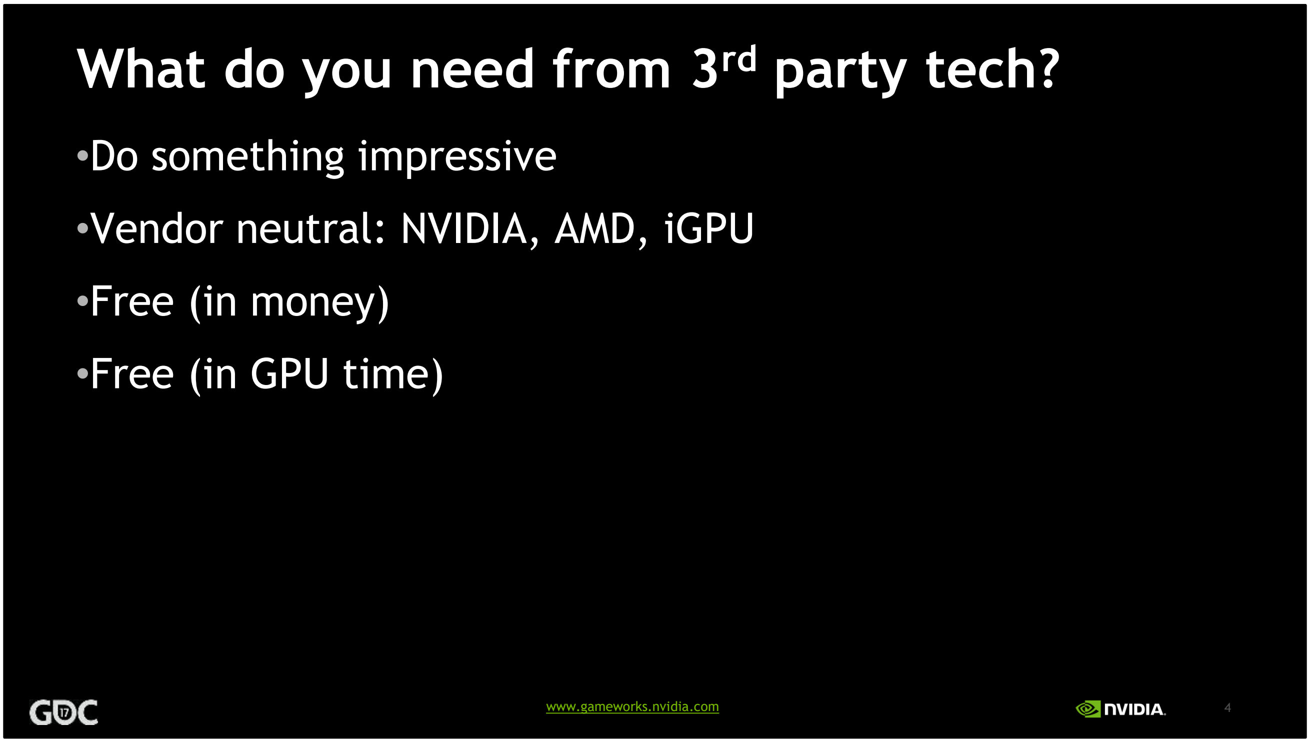
Again, you represent the fluid by particles, but you select a different particle-particle interaction rule that makes it look like a fluid.

Finally, rigid body bullets made from groups of 4 particles.

What do you need from 3rd party tech?

Flex is a 3rd party technology. It’s code that you didn’t write yourself, and that could have downsides.

What do you need from 3rd party technology to be comfortable about using it?



What do you need from 3rd party tech?

- Do something impressive
- Vendor neutral: NVIDIA, AMD, iGPU
- Free (in money)
- Free (in GPU time)

Impressive: If it doesn't do something new and interesting, then you'd just use your own code. Why risk it?

Vendor neutral:

As game developers you naturally want to maximize sales by releasing on many platforms, and provide a similar experience on each. On PC, this means you need tech to work (and be efficient) on NVIDIA, AMD and iGPU.

Free (in money): Things that cost money need producers etc. to agree to pay for them.

If something's a free download it's easier for you as a programmer to download and evaluate it.

Free (in GPU time): I know what you're thinking, you're thinking "Richard, even if you gave me all the things above, I don't care, GPUs are for graphics and our graphics guys want 100% of it for their stuff. There's nothing spare for GPU physics."

We've got something for you for this last problem, we'll look at that in a few slides time.

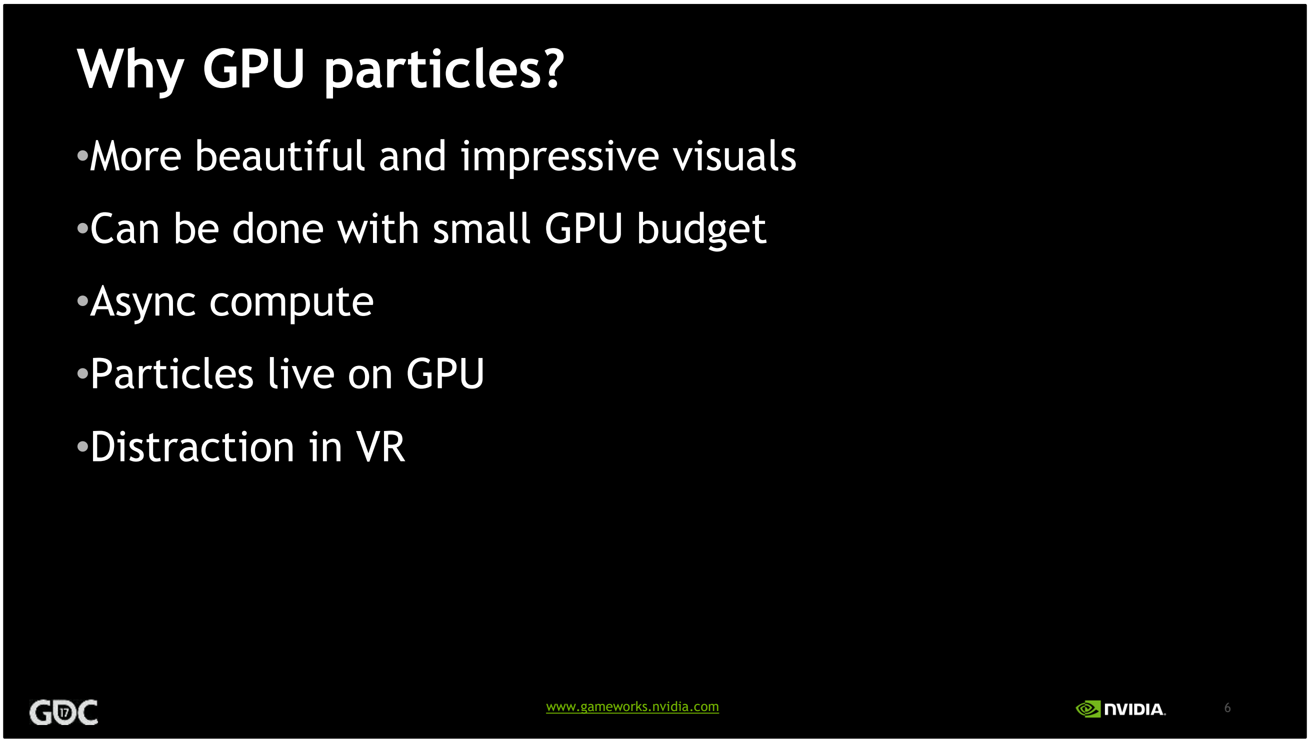
But first, let's do something about vendor neutrality….

Flex is now implemented in D3D Compute

www.gameworks.nvidia.com

5

Flex 1.1 is available today on D3D11 and D3D12, so now runs on all PC GPUs.



Why GPU particles?

- More beautiful and impressive visuals
- Can be done with small GPU budget
- Async compute
- Particles live on GPU
- Distraction in VR

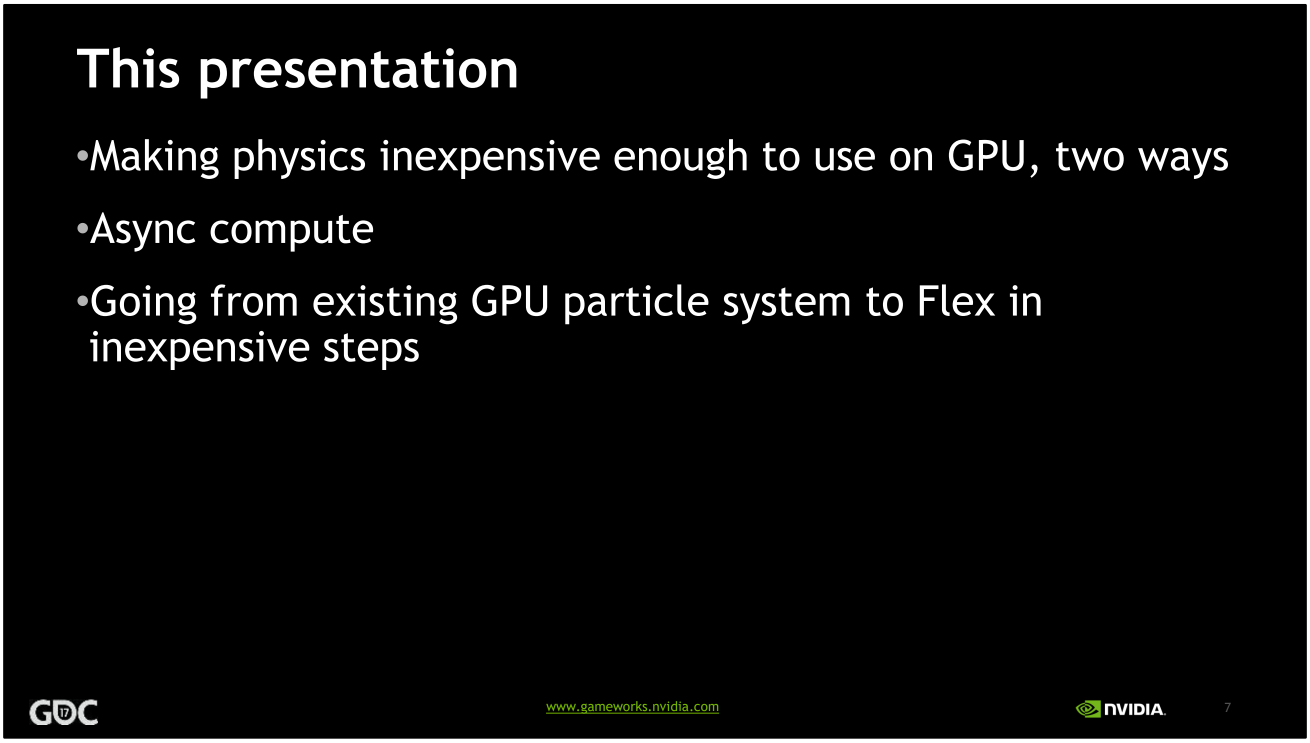
www.gameworks.nvidia.com 6

Most of you already render particles on the GPU, and some of you are already doing particle collision detection on the GPU, as we'll see later.

Why improve your GPU particles to do collision, rigid bodies, cloth or fluids?

Effects like you saw in the demo undoubtedly make games more beautiful and impressive, but isn't that what CPUs are for?

- I'd like to convince you today that these effects can be done very inexpensively on GPU now that we have async compute.
- Particles naturally live on GPU, you've got to render them, and CPU->GPU transfers are not free. Low latency is increasingly important, and simulating particles immediately before rendering them is the lowest latency solution.
- You might say, "I'm happy that my existing GPU particles fade out or are killed when they hit the ground, I only need full collision detection for big rigid bodies". It's part of the visual language of (desktop) games that debris particles don't have to look solid. However, in VR the game is your reality, you have more time to look at things, and solid particles not behaving like solids breaks the illusion and is distracting.



This presentation

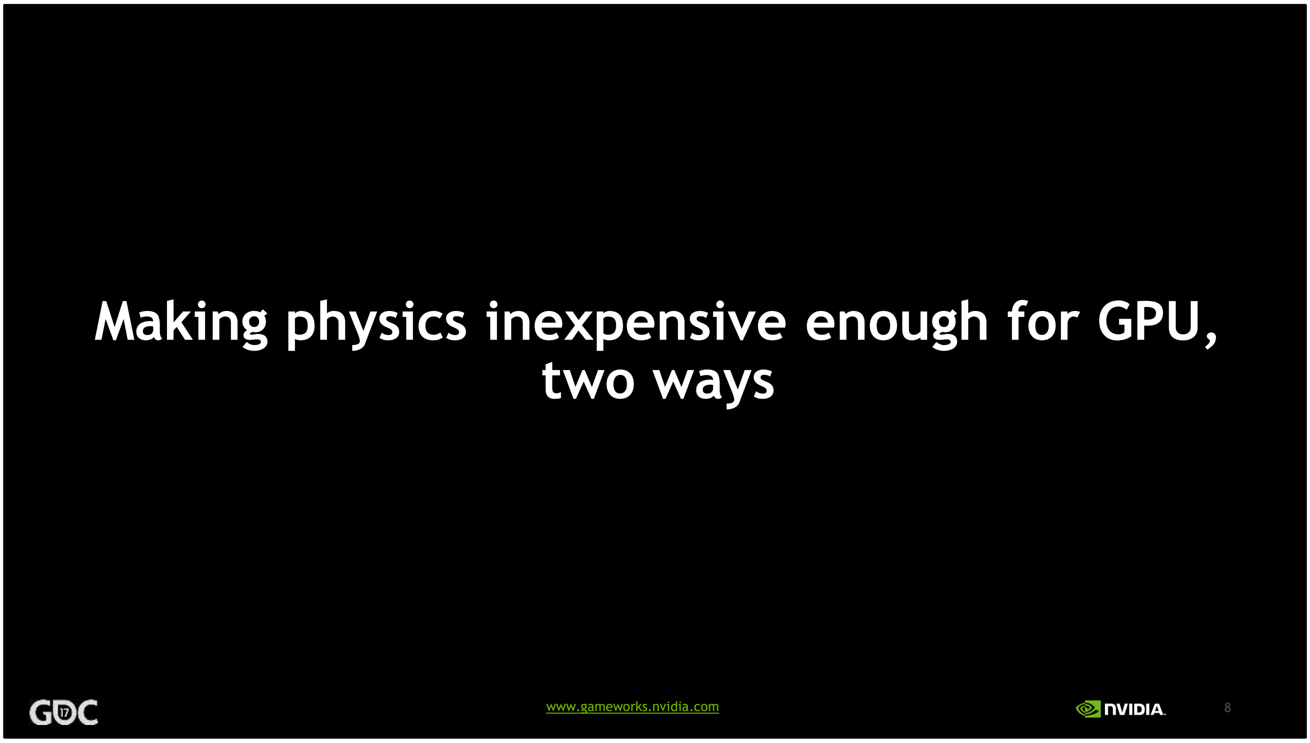
- Making physics inexpensive enough to use on GPU, two ways
- Async compute
- Going from existing GPU particle system to Flex in inexpensive steps

www.gameworks.nvidia.com

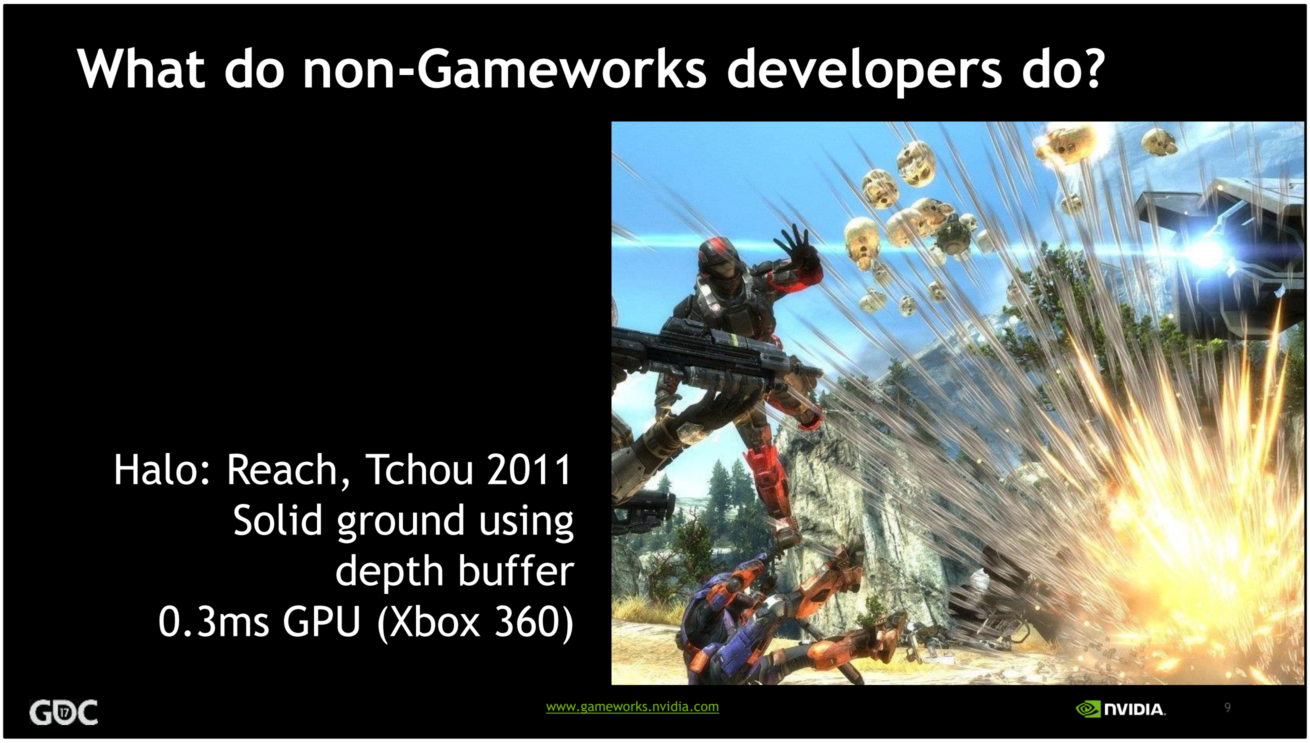
I'd like to convince you to shoehorn particle simulation onto your GPU.

I'm going to show you two things:

- Async compute
- Extending your existing GPU graphics particles towards doing full physics in inexpensive steps.



Making physics inexpensive enough for GPU, two ways



What do non-Gameworks developers do?

Halo: Reach, Tchou 2011
Solid ground using
depth buffer
0.3ms GPU (Xbox 360)

www.gameworks.nvidia.com

9

Before we talk about steps to improve particles, what sorts of GPU particle simulation are you already doing?

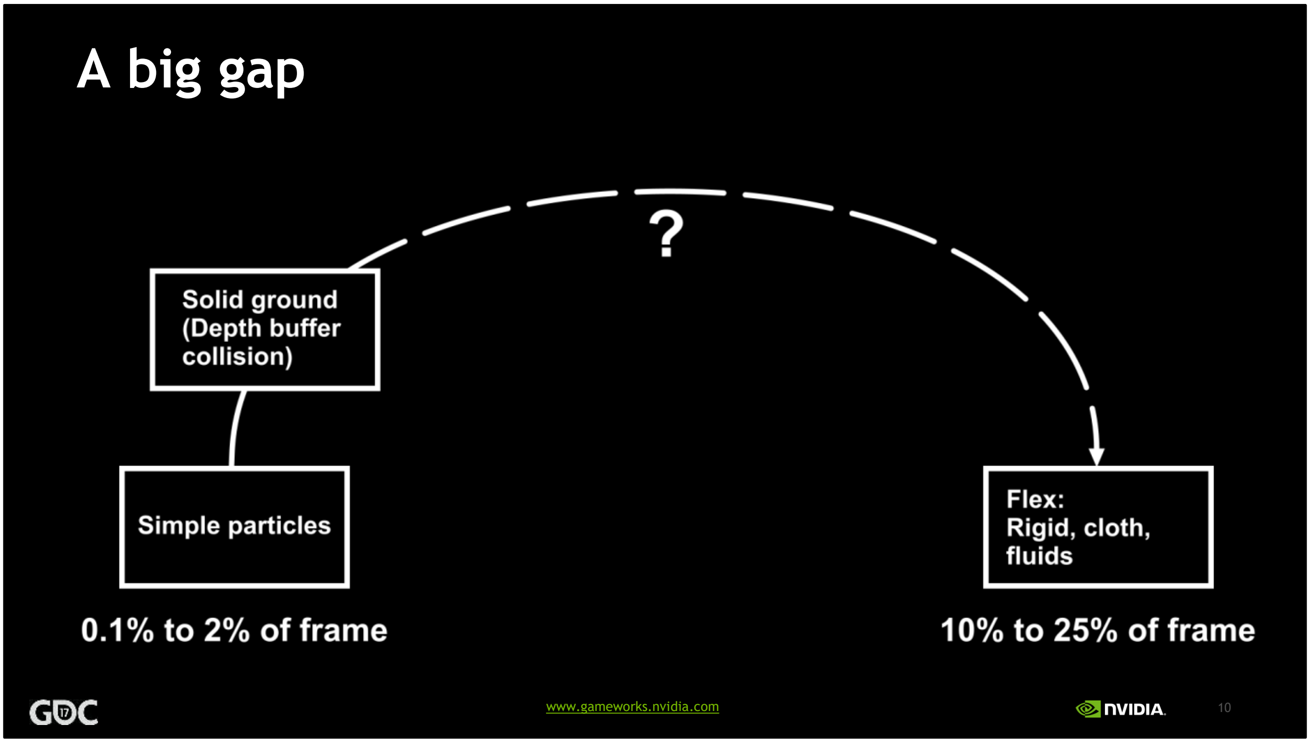
Of the games we analyze at NVIDIA, we often see the GPU colliding particles against the depth buffer. We'll describe it in a bit.

Chris Tchou used it in Halo Reach and did a great GDC2011 talk about it, I recommend that you look it up if you missed it.

He mentioned his GPU budget was 0.3ms for simulation and rendering on Xbox 360, so it's a cheap and fairly good step up from particles that fall through the floor.

Notes:

Picture used with permission of Chris Tchou



A big gap

Solid ground (Depth buffer collision)

Simple particles

?

Flex: Rigid, cloth, fluids

0.1% to 2% of frame

10% to 25% of frame

www.gameworks.nvidia.com

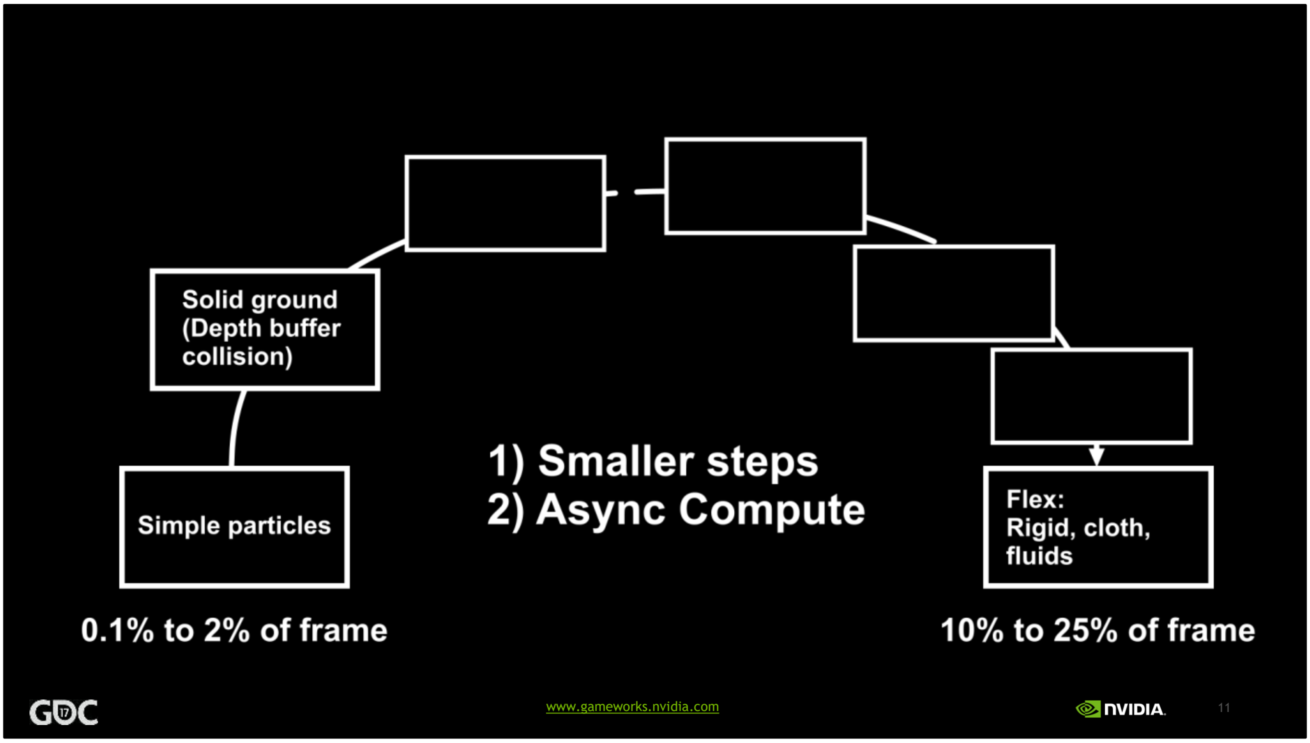
10

On the left you can see non-colliding particles and the previous slide’s depth buffer method.

They typically take 0.1% to 2% of the GPU frame, a budget which almost all games are happy with.

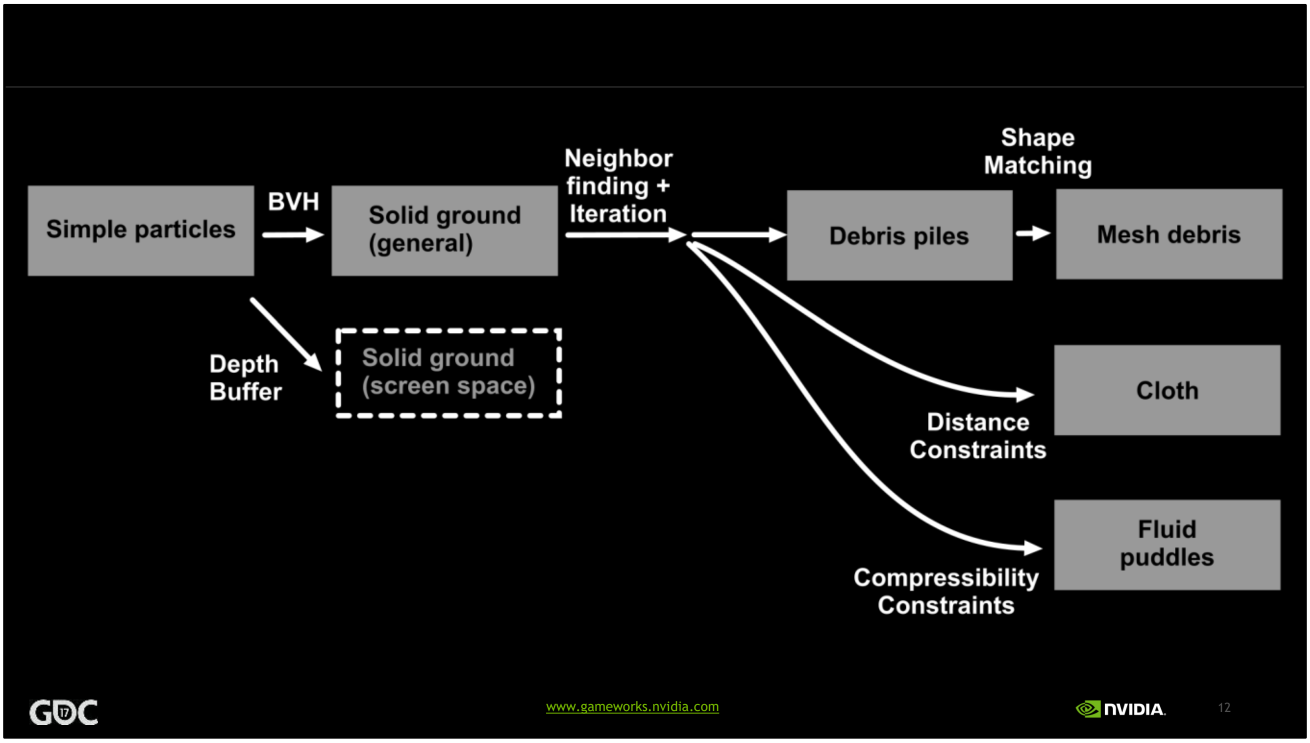
On the right is a full physics engine running on the GPU, which could take 10% to 25% of the GPU, a budget which few (GPU graphics) programmers would be happy with.

This presentation is about how to bridge this gap in two ways…



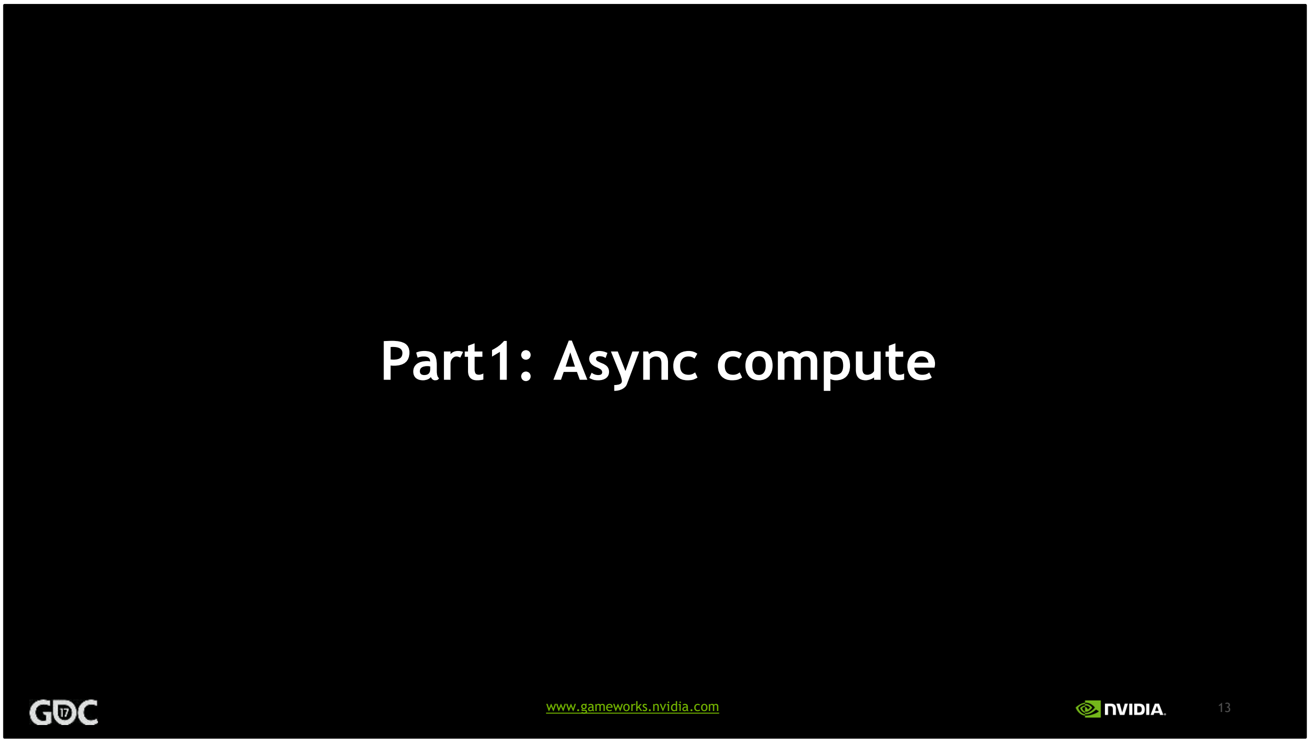
First is showing you some intermediate steps whose cost is closer to what you're already doing on GPU.

Second is making all the numbers on this slide smaller using async compute.



There are some intermediate steps we're going to show you later.

For each step we'll show you the incremental cost in GPU time, memory and code complexity.

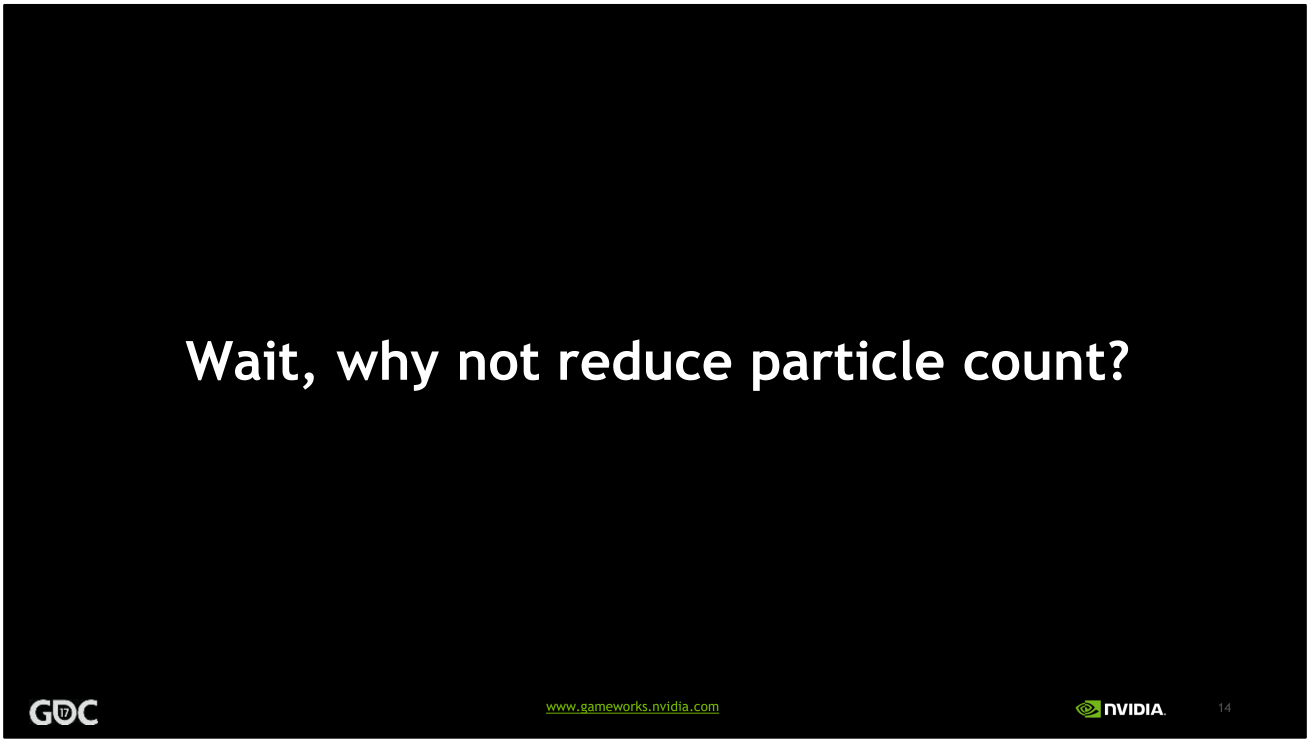


Part1: Async compute

www.gameworks.nvidia.com

13

But first, async compute.



Wait, why not reduce particle count?

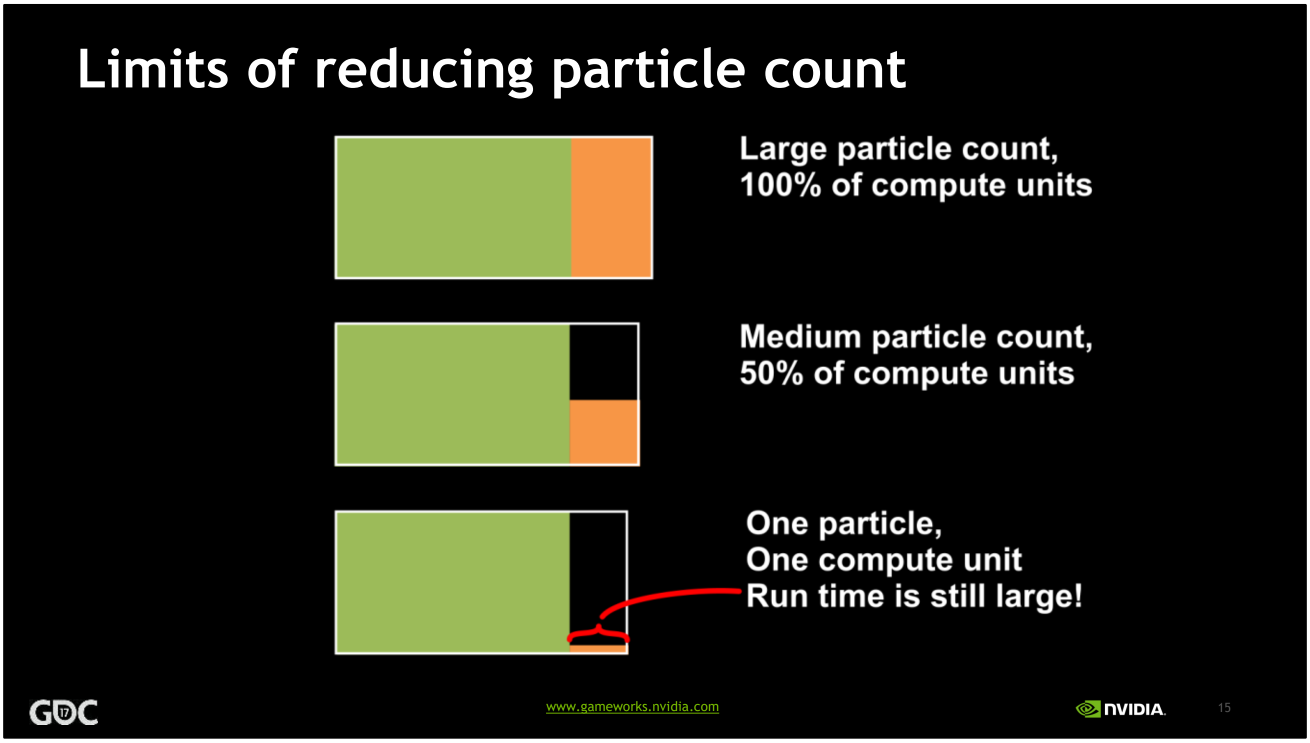
www.gameworks.nvidia.com

14

Wait, why is async compute necessary?

Can’t you just take a GPU particle engine like Flex and turn down the number of particles until any arbitrarily small GPU budget is met?

I’ll show you why that doesn’t work in the next slide.



Limits of reducing particle count

Large particle count,
100% of compute units

Medium particle count,
50% of compute units

One particle,
One compute unit
Run time is still large!

GDC

www.gameworks.nvidia.com

15

Engines like Flex compute particles in parallel on the GPU’s compute units.

In all the diagrams I’m going to show you:

X axis is time,

Y axis is GPU compute unit utilization,

Green is **gr**aphics,

Orange is compute.

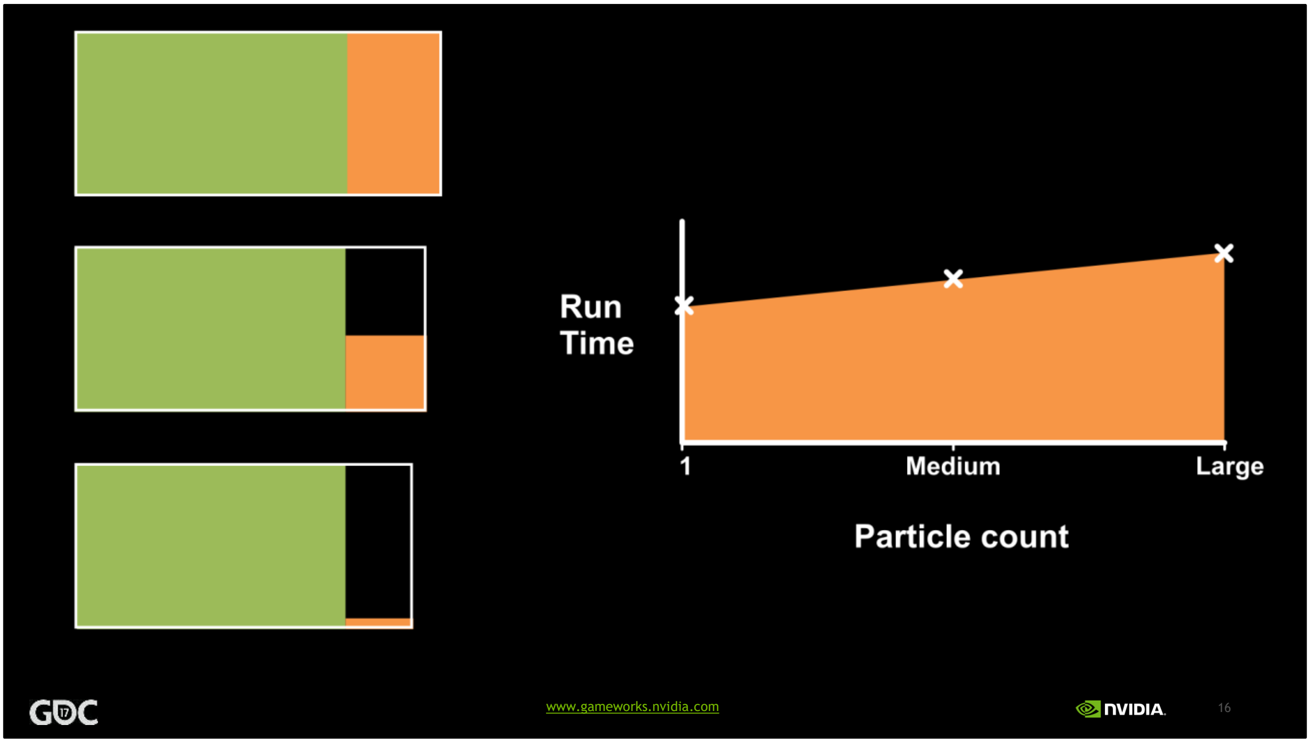
The top diagram shows the case where you’ve got just enough particles to fill the parallelism of the GPU.

Suppose the width of the orange bar exceeds your GPU time budget for particles, let’s try halving the particle count.

As you can see in the middle diagram, the run time does decrease, but not significantly.

Let’s take it to the extreme, one particle running on one compute unit, shown on the bottom diagram.

Again, the time decreases a tiny bit, but probably not by enough.



www.gameworks.nvidia.com

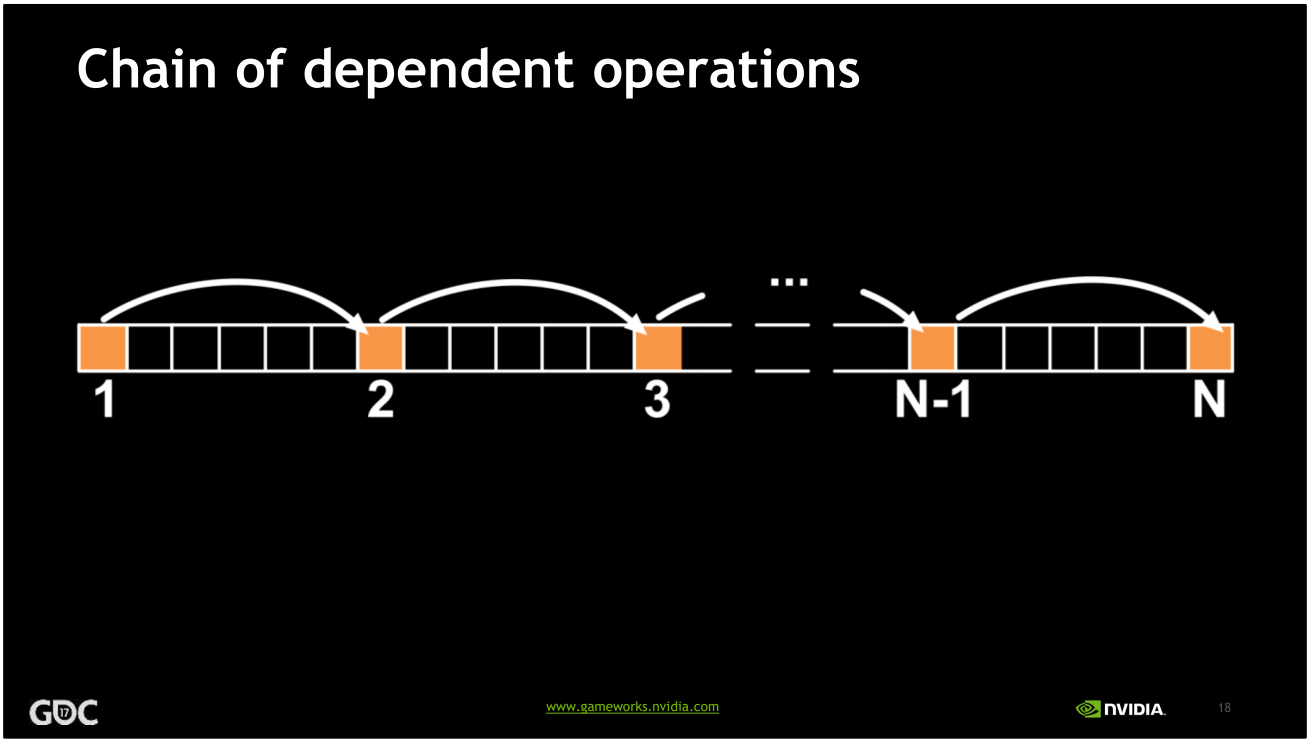
If you graph GPU run time versus particle count, it looks like this.

The key thing is that (unlike on a CPU) the line doesn’t go through the origin, there’s a large fixed cost.

Why is minimum cost high on GPU?

www.gameworks.nvidia.com

But why?



Chain of dependent operations

GDC 17

www.gameworks.nvidia.com

NVIDIA 18

In this diagram, the orange boxes represent instructions, and the arrows represent dependencies between the instructions.

This is a simple program where, every instruction depends on the previous one, but all programs have critical paths that look like this.

CPUs and GPUs are pipelined, meaning that the results of instructions are not available right away.

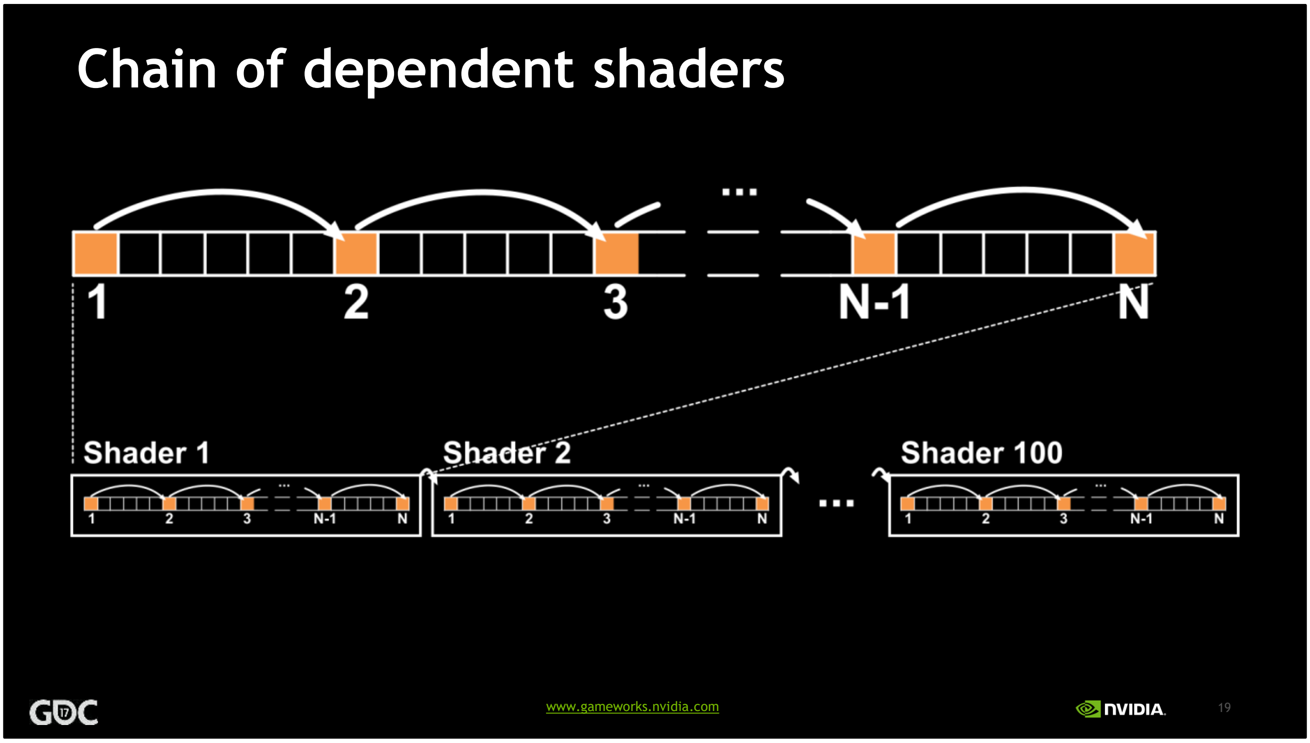
That’s why there’s unused cycles between the instructions in this diagram, the number of unused cycles is the instruction latency.

To make this diagram readable, I’ve shown an instruction latency of 6 cycles, but GPUs have very high instruction latencies, hundreds of cycles for cache misses.

So when the critical path of your algorithm is a long chain, and then you run it on a GPU with high instruction latency, the result is very long run time.

GPUs get around this problem by multithreading, after executing instruction 1 they can cheaply context switch to another thread.

So in this example if you had 6 threads you would have no wasted cycles.



Chain of dependent shaders

www.gameworks.nvidia.com

19

The last slide showed a single shader.

In physics engines it is common to have long chains of dependent shaders.

It depends on solver iteration count, but Flex can have a chain of 100 dependent shaders.

So the long latency of a single shader gets multiplied by the number of dependent shaders.

This latency is the large fixed cost you saw on the graph a few slides ago.

3 Opportunities for async compute

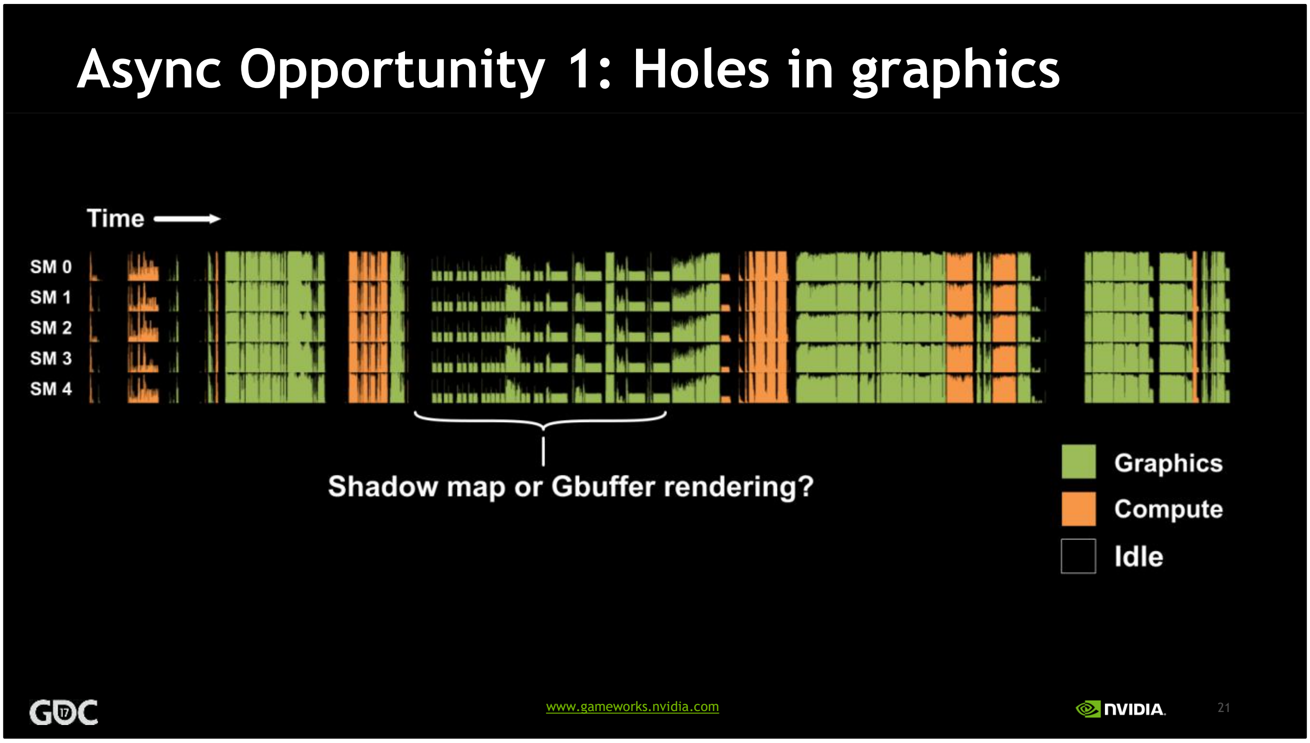
www.gameworks.nvidia.com

I’m going to show you three opportunities for using async compute.

The first is the one many of you use already, filling holes in graphics with compute.

The second two are about reducing the fixed cost of compute, by filling holes in compute with graphics.

Note that these last two methods work even if your graphics code is perfect and has no holes.



Async Opportunity 1: Holes in graphics

Shadow map or Gbuffer rendering?

🟩 Graphics

🟧 Compute

□ Idle

This graph is from an internal tool we use to identify optimization opportunities in games.

The X axis is time, and the Y axis has one row per SM. Within each row, the height of the bar gives utilization.

Again, GReen is for Graphics, orange is compute.

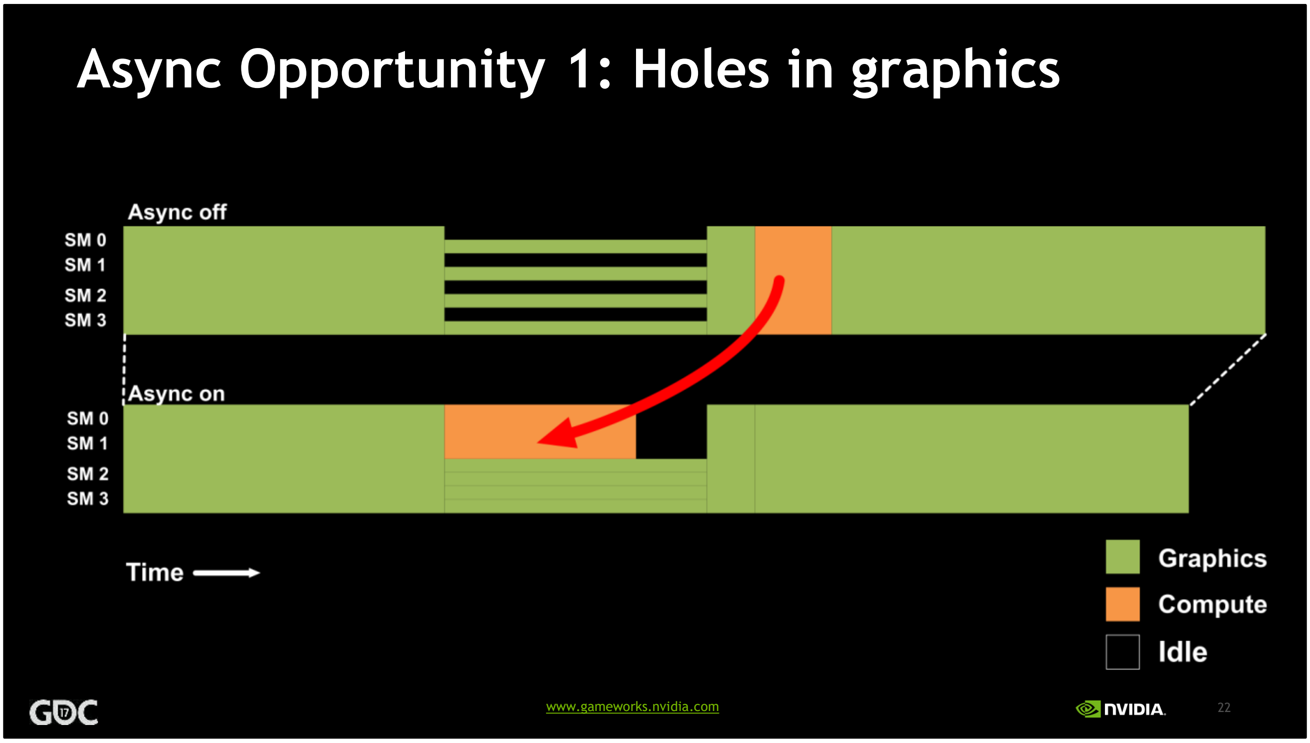
Black is idle time that we want to reuse for running physics compute.

What you’re looking at here is a single frame of an application.

These graphs look different from game to game.

But what they usually have in common is a region of Z-only rendering for shadow maps where the fixed function units are busy.

Previously, most of the shader units were idle during this time, and async compute allows us to reclaim this waste.



Async Opportunity 1: Holes in graphics

Async off

SM 0
SM 1
SM 2
SM 3

Async on

SM 0
SM 1
SM 2
SM 3

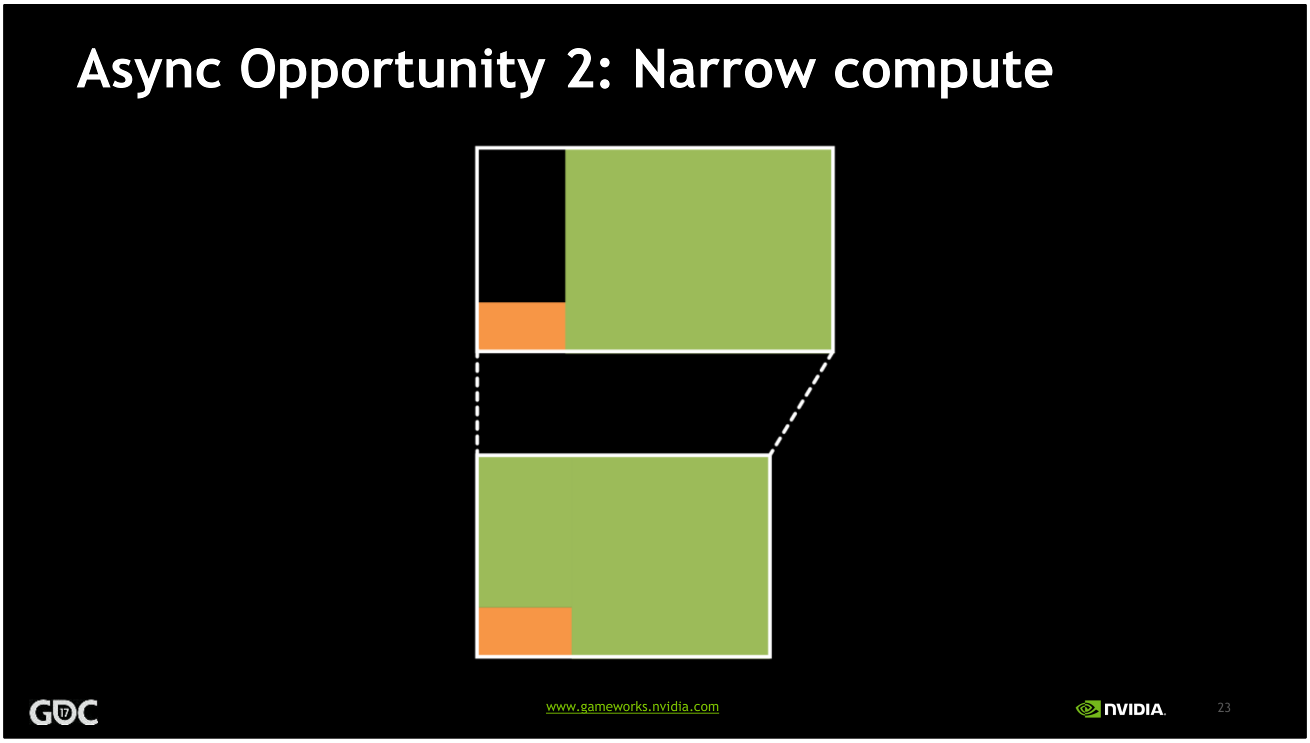
Time →

🟩 Graphics
🟧 Compute
□ Idle

Here’s an abstract diagram of how running compute during Z-only rendering looks.
Top is async-off, bottom is async on.
The dashed line shows how much the frame gets shorter by.

I know that many of you (especially those working on console games) are already doing this.

But… it doesn’t matter, the next two async compute opportunities work even if you’ve already filled all your graphics holes with your own compute.



Async Opportunity 2: Narrow compute

GDC

www.gameworks.nvidia.com

NVIDIA

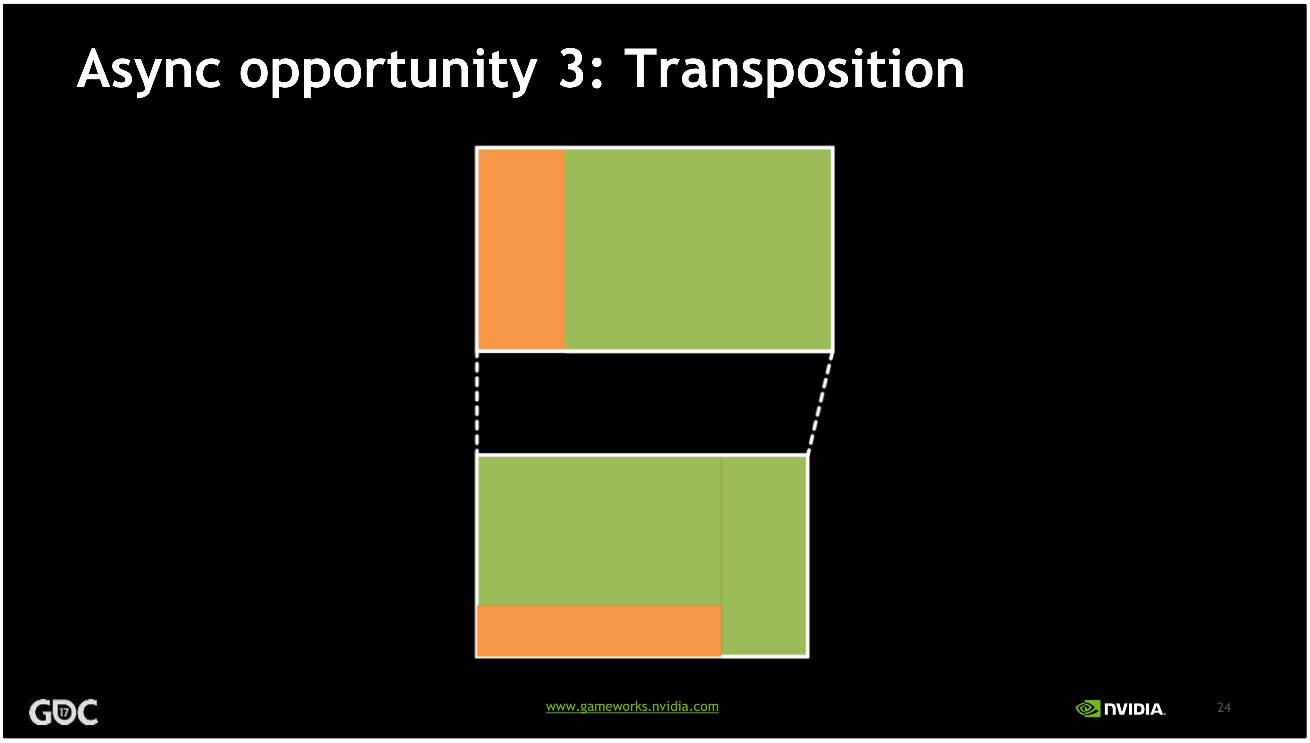
23

Suppose you need, say, 1000 particles.

That’s not enough to fill the GPU, so in this case you can use async compute to flow the graphics around the compute.

In this diagram, I’m assuming that the graphics has no holes, the green area is the same.

The dashed line again shows how much shorter the frame gets.



Async opportunity 3: Transposition

www.gameworks.nvidia.com

24

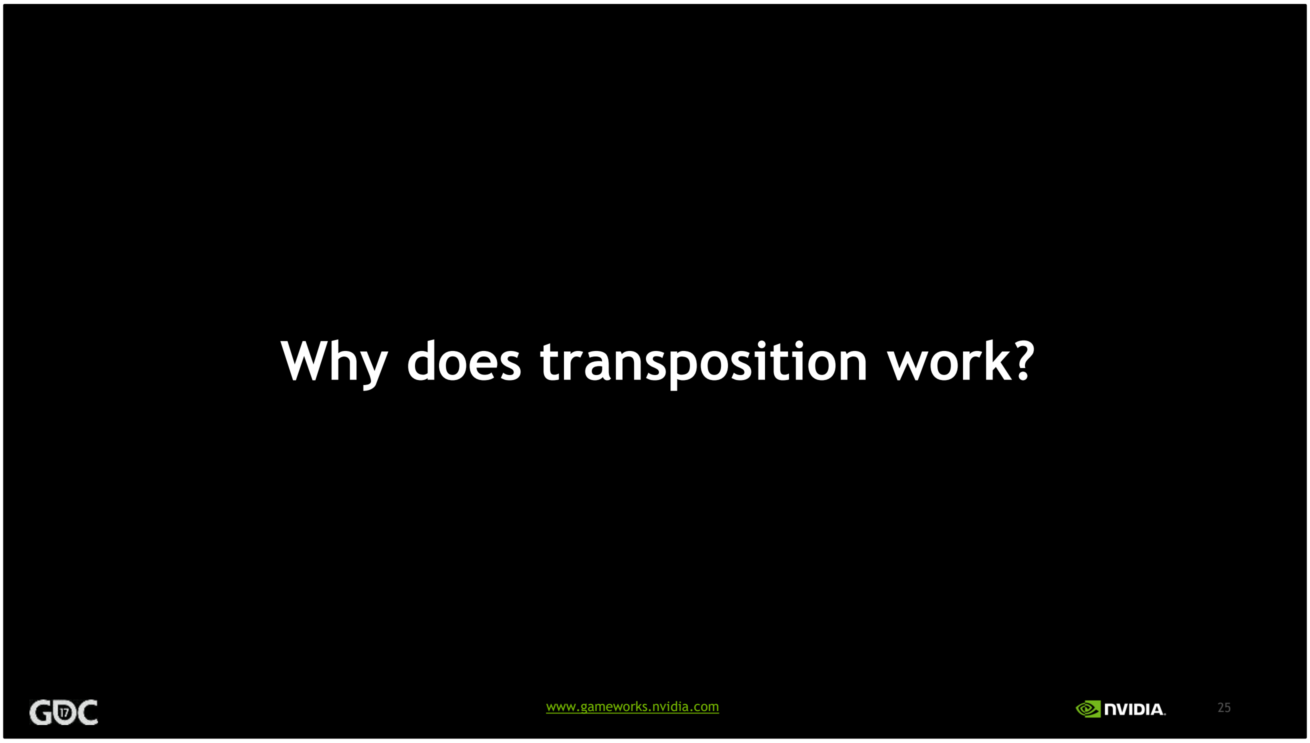
1000 particles is probably not your common case though, so it’s this third opportunity that gives you the huge benefits in physics engines like Flex.

The top diagram shows a case where you are using all the compute units of the GPU.

Suppose that you deliberately restrict the compute to run on ¼ of the compute units, shown in the bottom diagram. We call this transposition.

Again we’re assuming that graphics has no holes so the green areas are equal.

You might be surprised to see that the area of the orange rectangle has reduced, leading to a decrease in frame time (the dashed line).

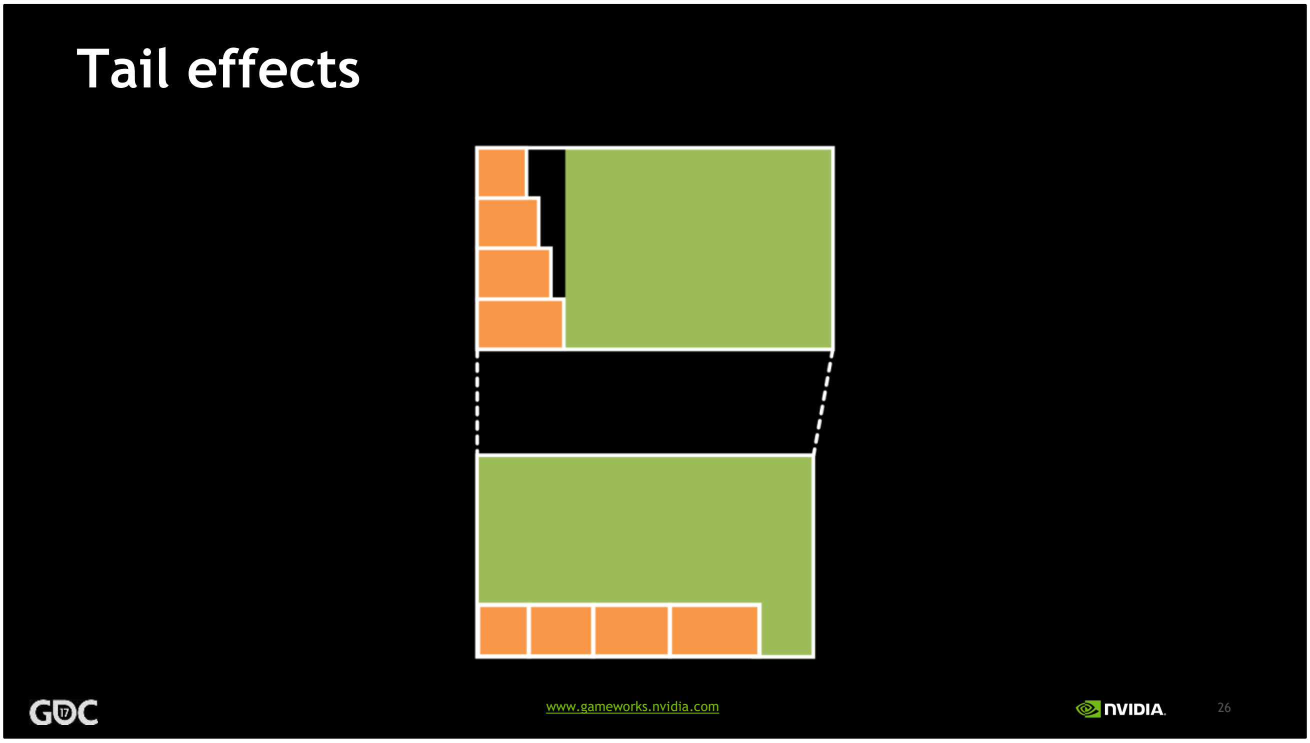


Why does transposition work?

Why?

The simplest reason is that it’s better to concentrate threads on as few compute units as possible to fill the holes in the chains of dependent operations I showed you earlier.

I’ll show you two further reasons.



Tail effects

www.gameworks.nvidia.com

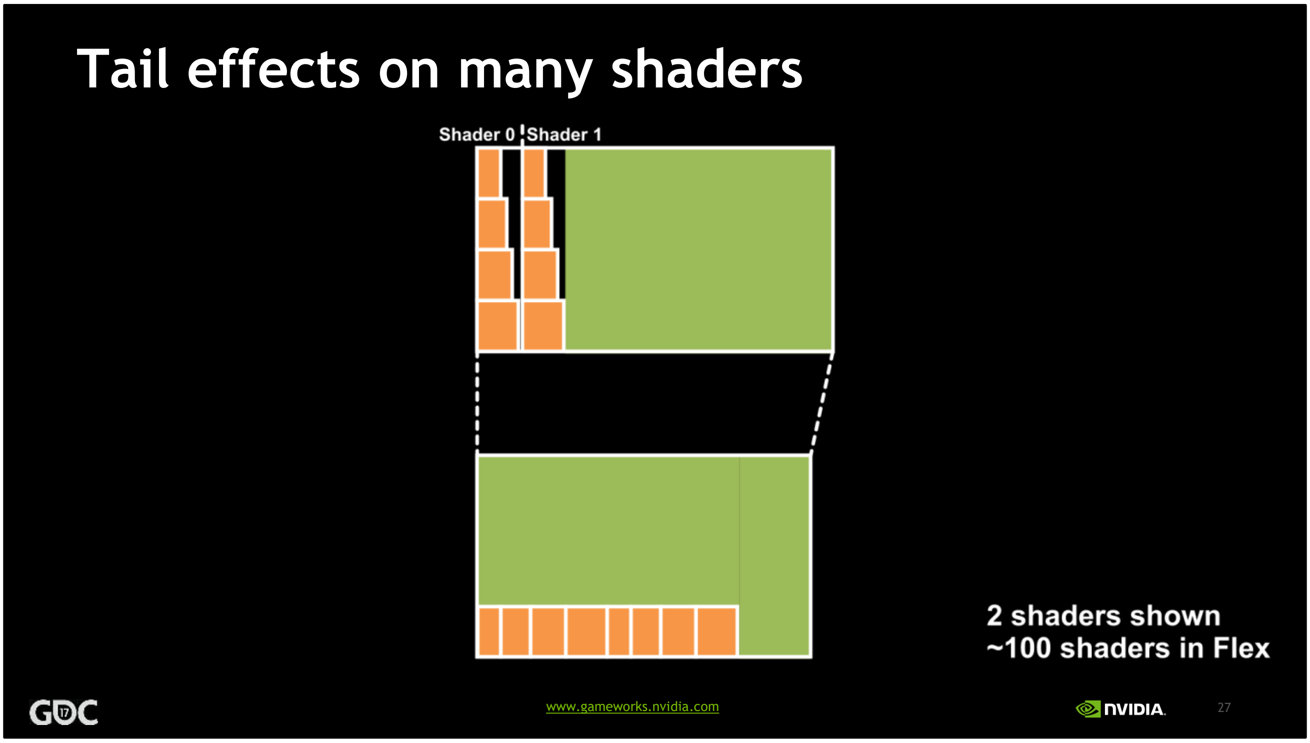
26

The first reason is tail effects.

Each particle can have a different number of neighbors, and so threads tend to finish at different times.

All the threads have to wait until the slowest one is finished. We call this a tail effect.

As you can see, async compute allows us to reclaim this waste.

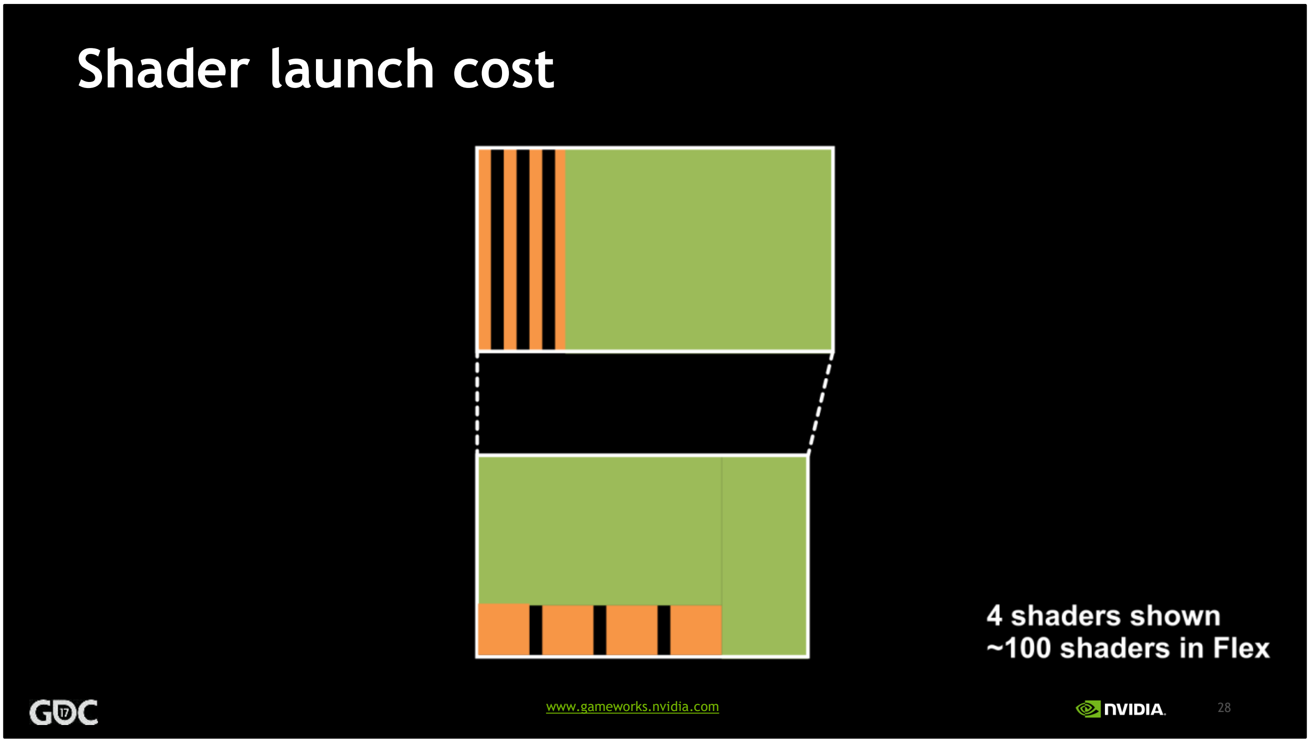


Tail effects on many shaders

2 shaders shown
~100 shaders in Flex

Tail effects occur in almost every shader in Flex, and so each shader has waste we can reclaim.

On this slide you can see the waste from two shaders, but real cases like Flex can have 100 dependent shaders, leading to a lot of waste.



Shader launch cost

4 shaders shown
~100 shaders in Flex

www.gameworks.nvidia.com

28

The second of the two reasons transposition works is shader launch cost.

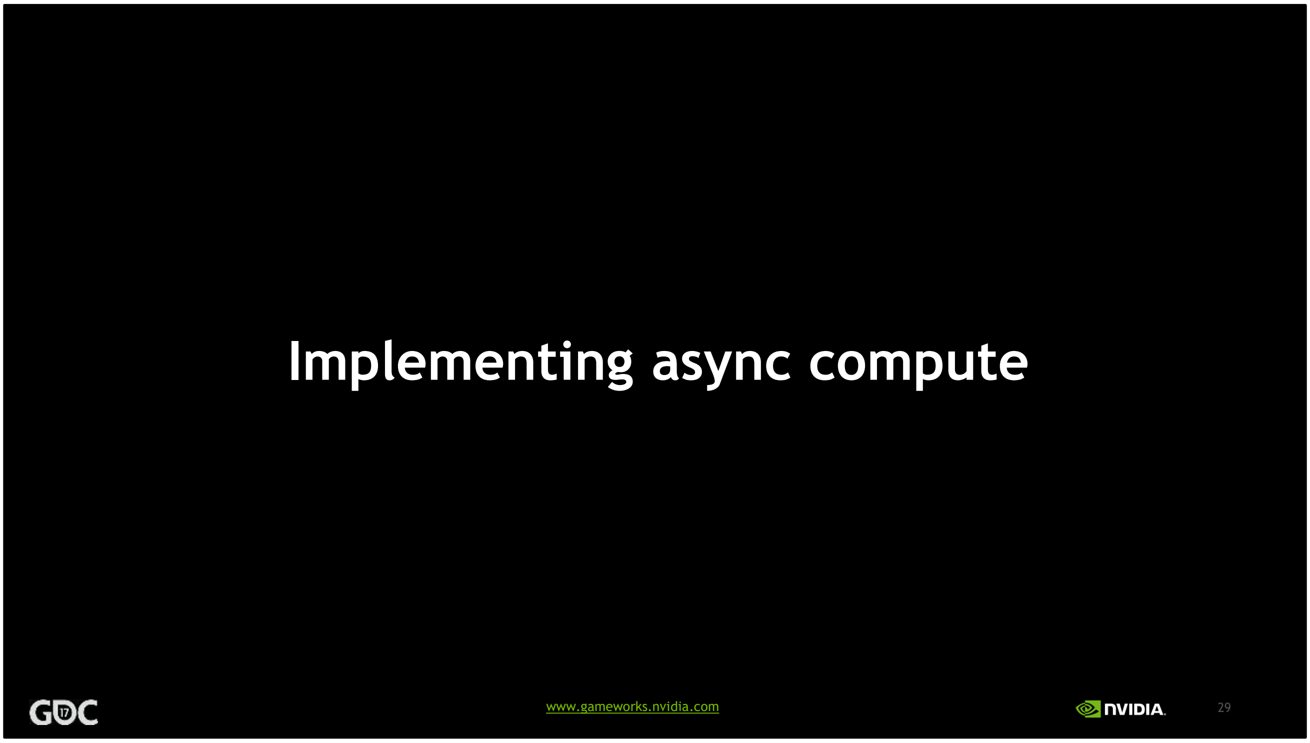
In the top diagram you can see four dependent shaders, the four orange bars.

The three black bars are the shader launch cost.

In this exaggerated diagram, the run time of the shader is equal to the shader launch cost.

In the bottom diagram you can see the benefit of transposition, you only pay the shader launch cost on compute units that you actually use.

The work takes four times as long in the bottom diagram, but the ratio of work to waste is 4 times better.

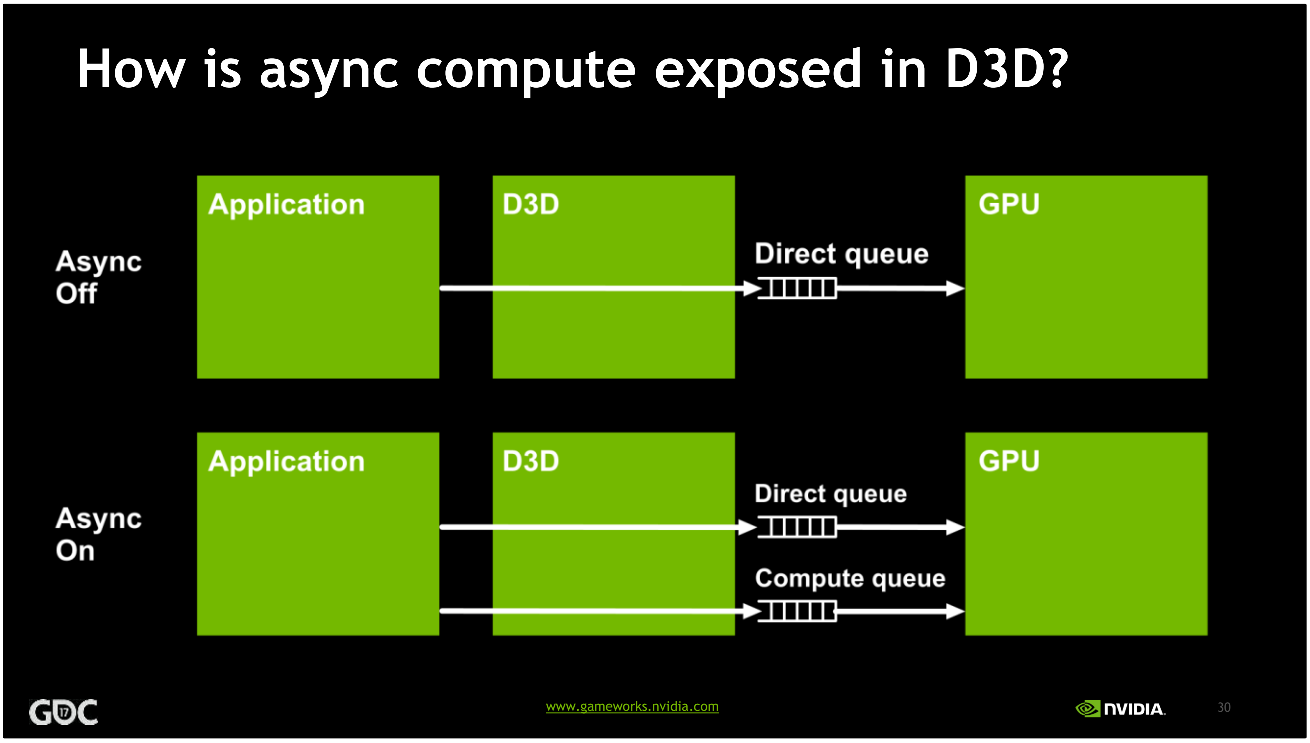


Implementing async compute

www.gameworks.nvidia.com

29

I hope by now that I’ve convinced you to try out async compute.



How is async compute exposed in D3D?

Async Off

Application

D3D

Direct queue

GPU

Async On

Application

D3D

Direct queue

Compute queue

GPU

www.gameworks.nvidia.com

30

How is async compute exposed in D3D?

You need some way to tell the GPU which compute work can be run in parallel with graphics.

Not all compute work can be asynchronous, sometimes compute and graphics feed into each other.

D3D12

D3D12 has a concept of queues. A queue can be either direct, compute, or copy.

Direct queues can do compute, graphics and copy, compute can only do compute and copy.

So if your app only has a single queue, it is probably a direct queue.

The way you add async compute to such an app is to add a compute queue.

By default, everything in the compute queue can run in parallel with anything in the direct queue.

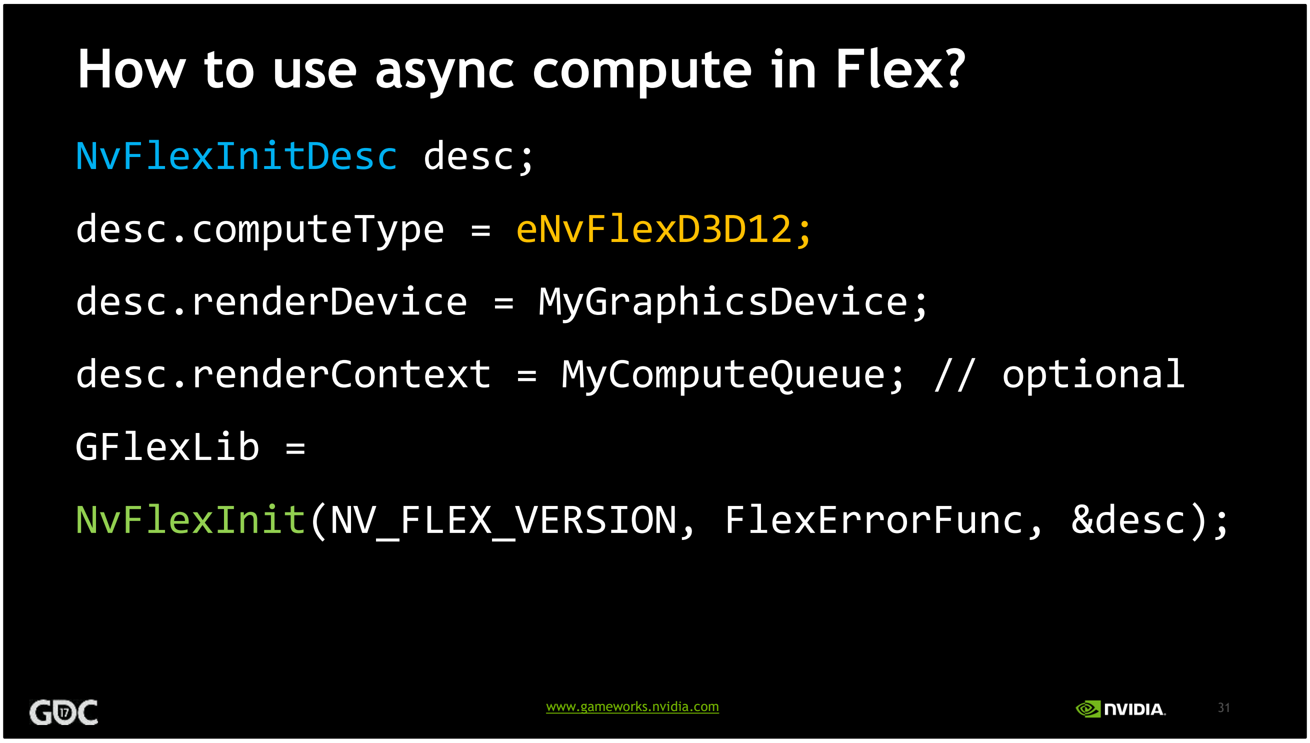
So if you want dependencies, you have to add them yourself using fences.

D3D11

D3D11 out of the box has no way to express that compute can be run in parallel, there are no queues in the API.

If you are interested in a backdoor for using async compute in D3D11, email me.

I can’t advertise the other vendor’s D3D11 async compute solution, but you could easily find it.



How to use async compute in Flex?

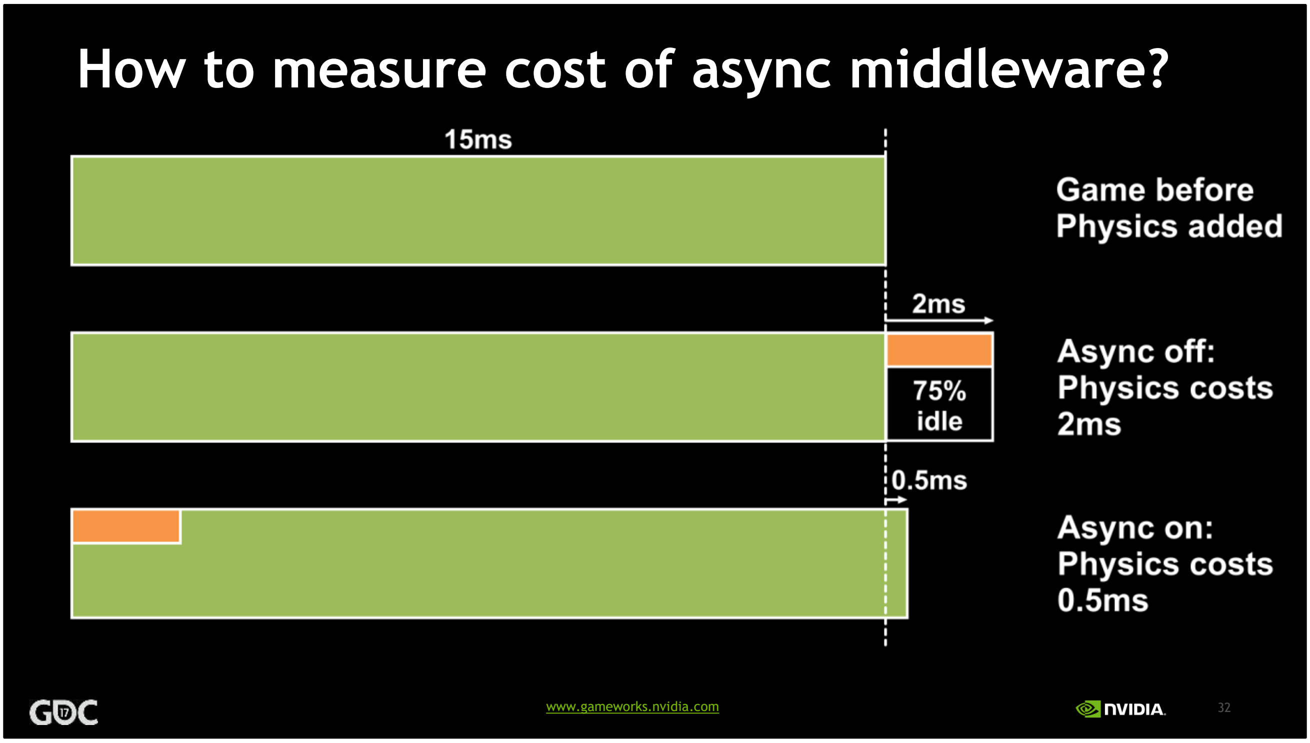
```
NvFlexInitDesc desc;
desc.computeType = eNvFlexD3D12;
desc.renderDevice = MyGraphicsDevice;
desc.renderContext = MyComputeQueue; // optional
GFlexLib =
NvFlexInit(NV_FLEX_VERSION, FlexErrorFunc, &desc);
```

Everything I’ve said about async compute so far probably applies to all GPU physics engines.

If you’d like to try D3D12 async compute in Flex, here’s how to do it.

If you have a compute queue that you’re using for your own async-compute, you can pass it to us and we’ll use that.

Otherwise we’ll create one for you.



How to measure cost of async middleware?

Game before
Physics added

Async off:
Physics costs
2ms

Async on:
Physics costs
0.5ms

www.gameworks.nvidia.com

32

It’s important to measure how much GPU time each effect in your game is taking, especially if it is from a third party library like Flex.

In the past you could do did this by using D3D events to record a GPU timestamp before and after each effect.

However, if you do this with async compute, you are going to get “billed” for all the graphics work that’s running at the same time.

In this picture the height of the bars represents the compute units of the GPU, the x axis is time. We’ve got 15ms of graphics, and 2ms of compute. But, for the reasons you saw earlier we’re effectively only using 25% of the compute units during this 2ms.

Without async compute, you’re wasting 75% of the GPU for 2ms. Adding the compute made the frame 2ms longer, so 2ms is the cost.

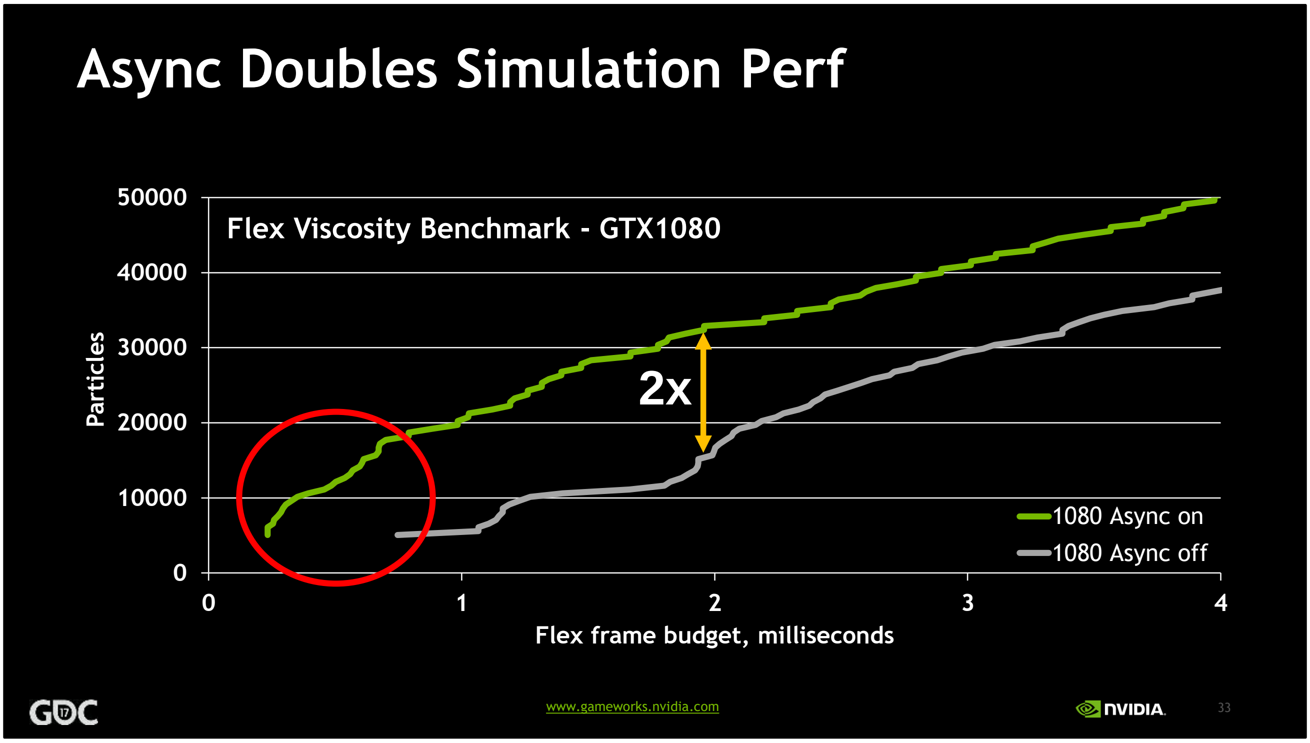
With async compute we move the compute to the start of the frame and run graphics in parallel with it.

If you put D3D timers around the compute they will still say 2ms, even though the frame only got 0.5ms longer.

So, don’t use D3D timers to time Flex or any other async compute effect.

Run your game twice, once with Flex, once without.

The difference in frame time is the cost of Flex.



Async Doubles Simulation Perf

www.gameworks.nvidia.com

33

All the diagrams I've shown you so far were cartoons to illustrate the concepts.

Now I'm going to show you some real data to show that it works.

In the earlier cartoon graph right at the start I showed cost on the Y-axis and particle count on the X-axis.

Before async compute, the problem was that the graph didn't go through the origin, there was a large fixed cost.

Your artists might want to know how many particles they can use for a given GPU budget. So on this slide I've flipped the axes.

Budget and cost are as defined in the previous slide.

The gray line shows async off. If you extrapolate, you can see there's a fixed cost of roughly 0.8ms.

The green line shows async-on, you can see that it's a lot closer to going through the origin.

This benefits you in two ways, for a given budget you can do more particles. At a 2ms budget you can do 30K particles instead of 15K, and the benefit increases for budgets smaller than 2ms.

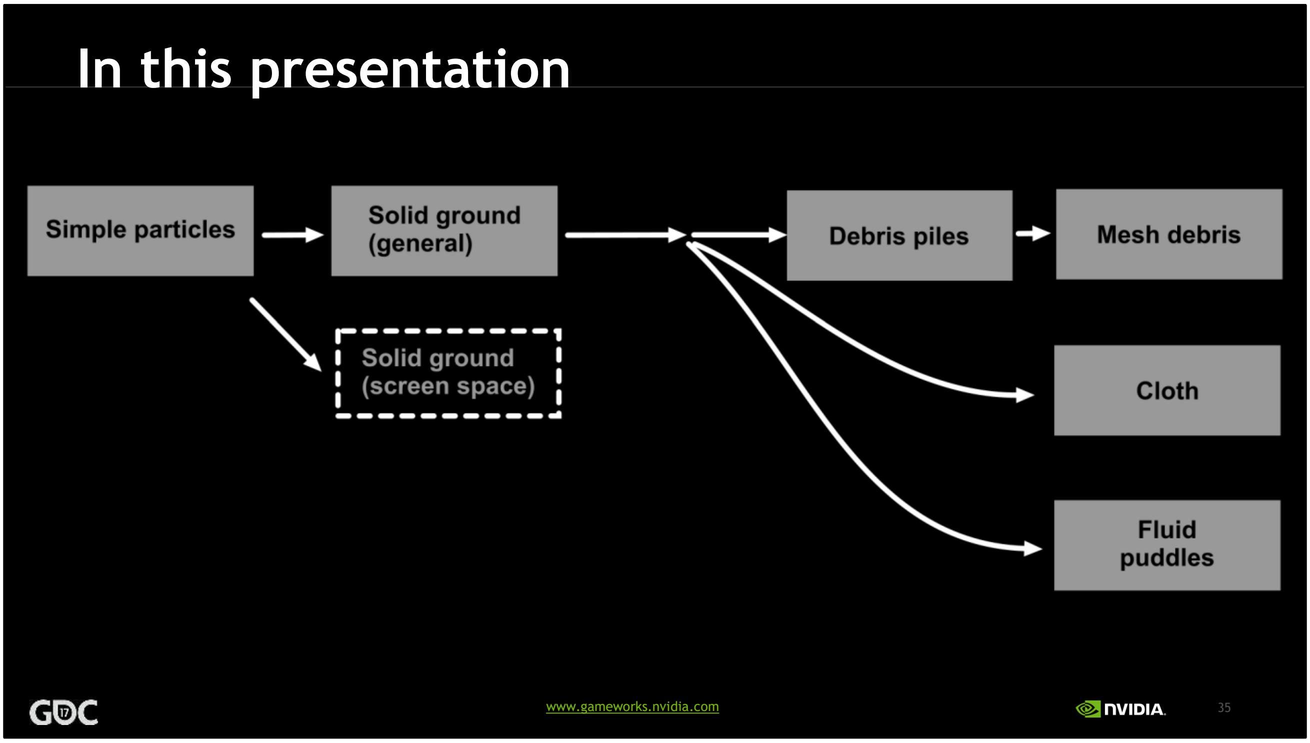
Look at the red circle. Before async compute, if your budget was less than 0.8ms I had nothing for you.

But with async compute, you can use Flex with smaller budgets than 0.8ms, with fairly decent particle counts. (The demo earlier was 10K particles).

Part 2: Smaller steps

Ok, that was async compute, it made everything cheaper even if graphics had no holes.

Now for part two of two, taking smaller steps.



In this presentation

Simple particles

Solid ground (general)

Solid ground (screen space)

Debris piles

Mesh debris

Cloth

Fluid puddles

Earlier you saw a gap between the kinds of GPU physics devs are already doing on their own, and what we've traditionally done with Flex.

Let's look at some steps to bridge that gap.

On the left you see "simple particles", this is old-school particles falling under gravity, falling through the floor. Cheap and cheerful.

The simplest thing you might want to do is make the ground solid, the dotted box is the depth buffer method mentioned already. The box is dotted because it's not included in Flex.

Screen space collision has some limitations, so I'm going to show you a more general solution that (fingers crossed) will be not too much more expensive than simple particles.

Making just the ground solid is fine if your happy for the objects to bounce away from the impact point. What if you want to make a pile instead, you need particle-particle interaction. You'll see how much extra code and GPU time it costs to add that.

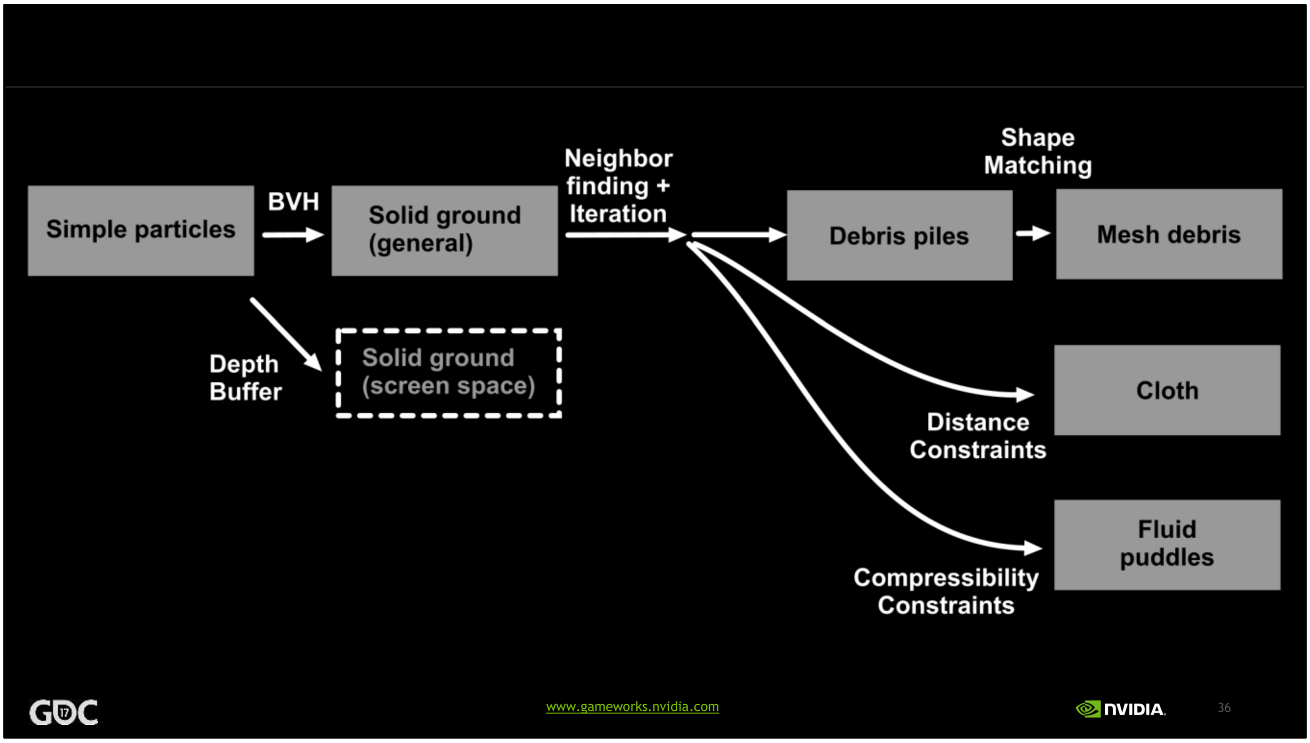
Piles of particles fine if your debris is roughly spherical, a common use case is roughly spherical rocks. Also its fine for non-spherical objects when you can fake orientation. In the past we did a particle system for a couple of games where the particles snapped to the orientation of the ground when they contacted. Cheap, but not ideal.

The far right on is full mesh debris with orientation, the most expensive option.

In between making the solid ground and making particles solid we're going to have to add a way for particles to find nearby particles and correct their positions. Once we have the code to do that, we can add different neighbor updating code to get cloth and fluids.

Now, I know some of you have used the simpler methods on the left for diffuse fluids like smoke or fine splashy liquids. What you get for the extra expense of the fluids on the right are puddles. Later you'll see for yourself how much extra that costs and what the visual difference is.

In the next slide you'll see the algorithms we use in Flex to implement each improvement.



For cases where the depth buffer collision artifacts are a problem, we'll show you how to do a BVH compute shader, and how cheaply that can be done.

As I just mentioned, to get piles of debris, particles need to be aware of and prevent penetration with each other. So we'll show you neighbor finding in shader(s).

Next we'll show you shape matching, which allows small groups of particles to represent orientable debris.

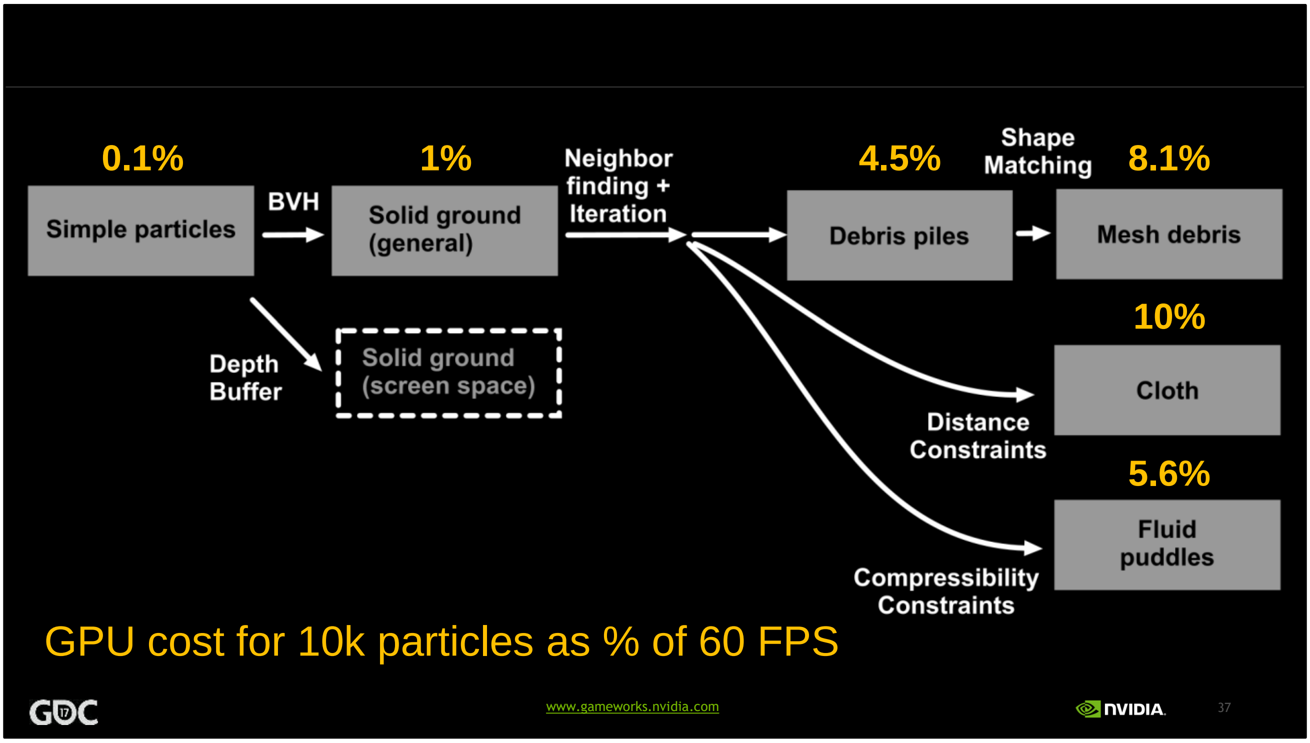
Once particles can access other particles, you can add distance constraints to make cloth meshes.

Finally, to get puddles we need a way of making the fluid as close to incompressible as we can.

A word I've not mentioned yet that's on the slide is iteration. Some of the phenomena on the right (especially rigids, cloth and fluids) can be quite stiff in real life.
Stiffness happens when sound waves travel quickly through an object. You're going to see a diagram of this later, but if a particle can only affect its nearest neighbor then that limits the distance a wave can travel in one frame to its nearest neighbor. We can fix that by running multiple substeps or iterations per frame. This adds to the cost of the effects on the right.

So doing huge rivers of incompressible particle fluids in a small amount of a GPU frame is still impossible, but all these effects are just steps on a journey. If you get the earlier cheap steps in place now, it'll be easier to switch on the more expensive effects as new GPUs and consoles come along. Also the benefit of particle based dynamics is when you do add say fluids, it automatically interacts with your cloth and your debris and everything else you added previously.

Before showing you the algorithms, in the next slide I'm going to jump right to the important bit. How much does this all cost?



0.1%

Simple particles

BVH

1%

Solid ground (general)

Depth Buffer

Solid ground (screen space)

Neighbor finding + Iteration

4.5%

Debris piles

Shape Matching

8.1%

Mesh debris

10%

Cloth

Distance Constraints

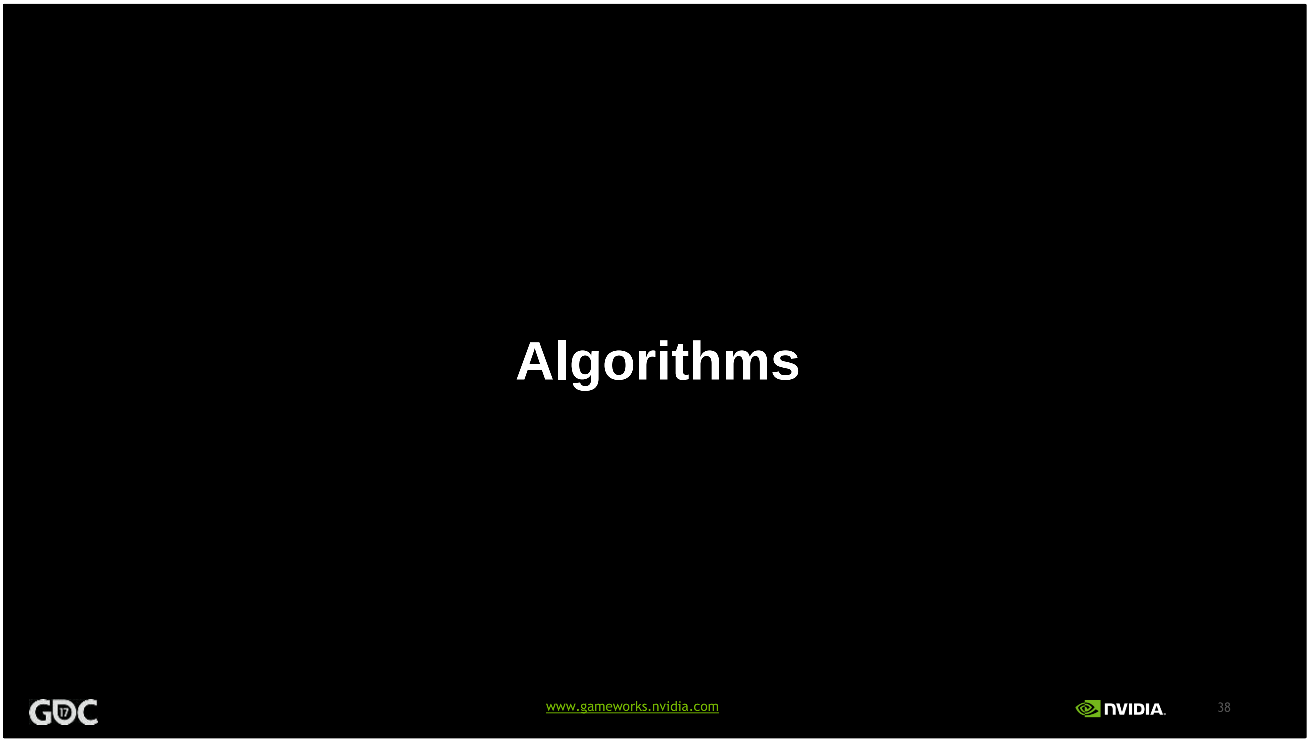
5.6%

Fluid puddles

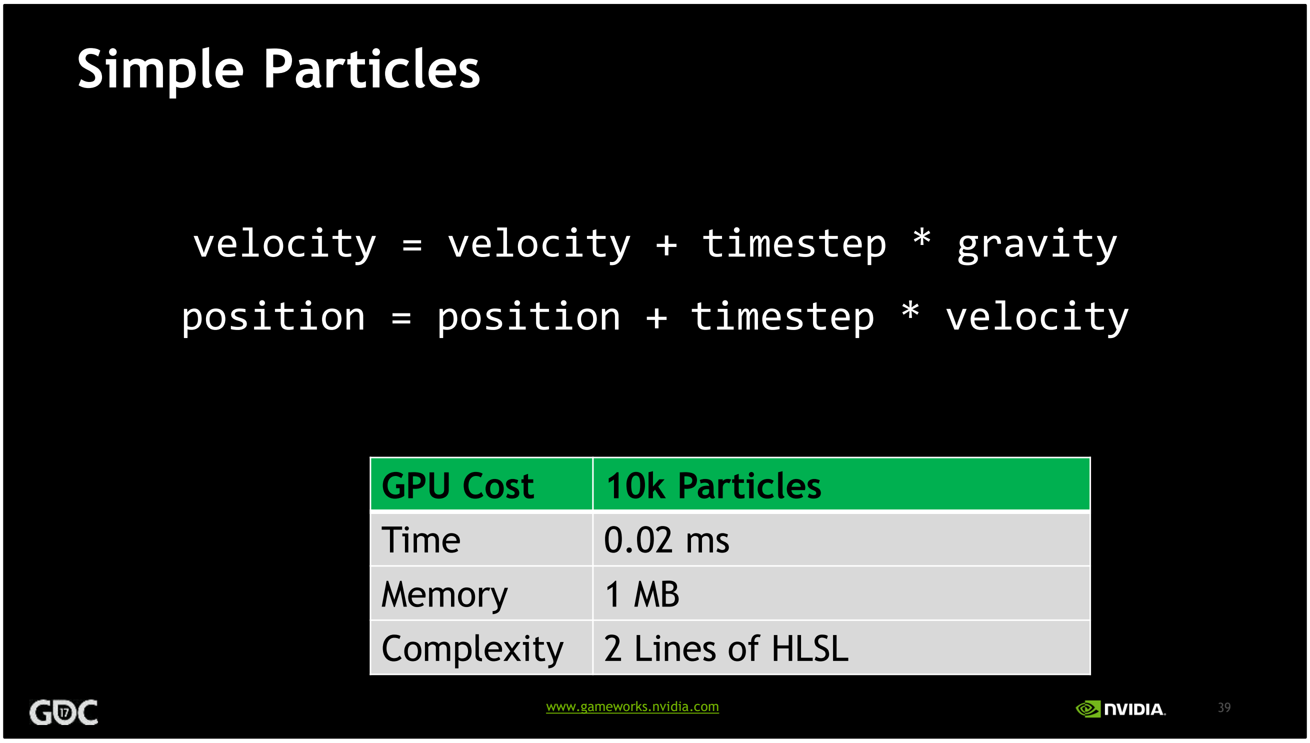
Compressibility Constraints

GPU cost for 10k particles as % of 60 FPS

All the costs on this slide are % of a 60FPS frame, measured on a GTX 1080



Algorithms



Simple Particles

```
velocity = velocity + timestep * gravity
position = position + timestep * velocity
```

GPU Cost	10k Particles
Time	0.02 ms
Memory	1 MB
Complexity	2 Lines of HLSL

This is the simplest particle update, adding gravity and stepping velocity and position.

Velocity += mass*gravity*frametime
Position += velocity*frametime

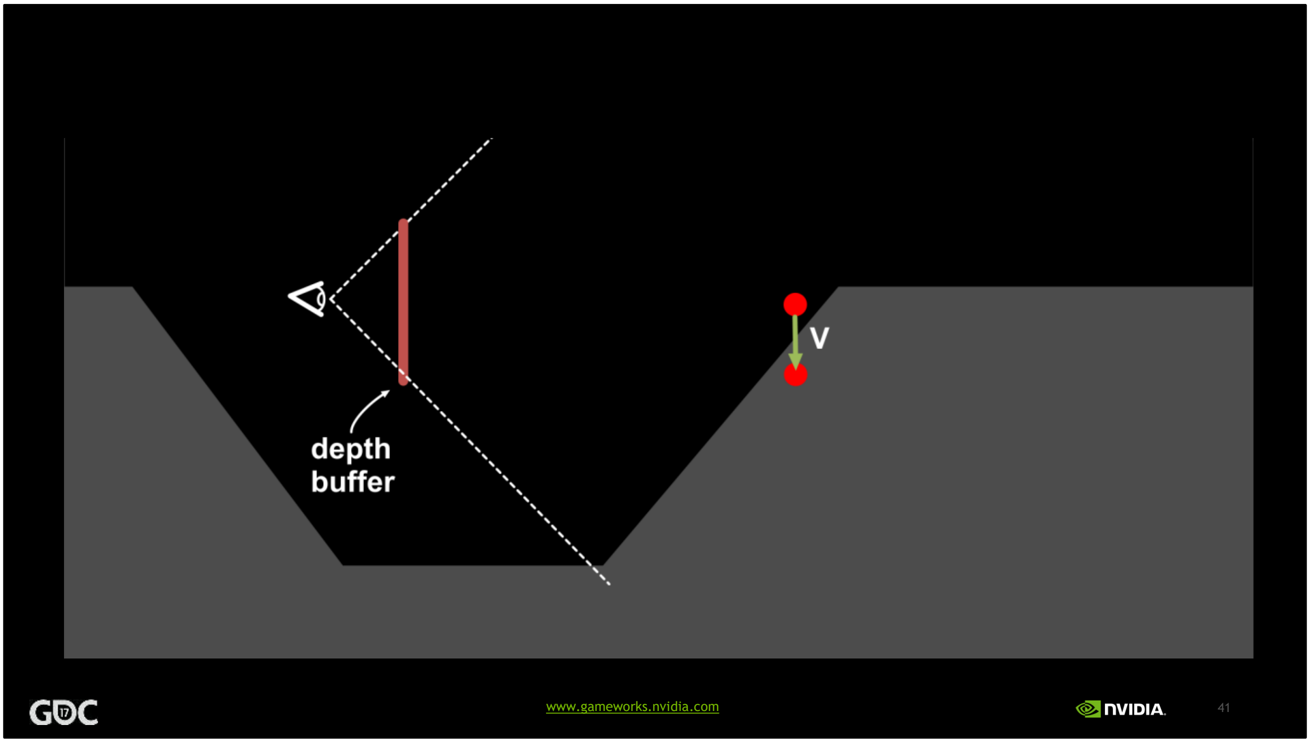
Solid ground (approximate)

www.gameworks.nvidia.com

40

Earlier I mentioned the depth buffer collision detection method that some of you are using.

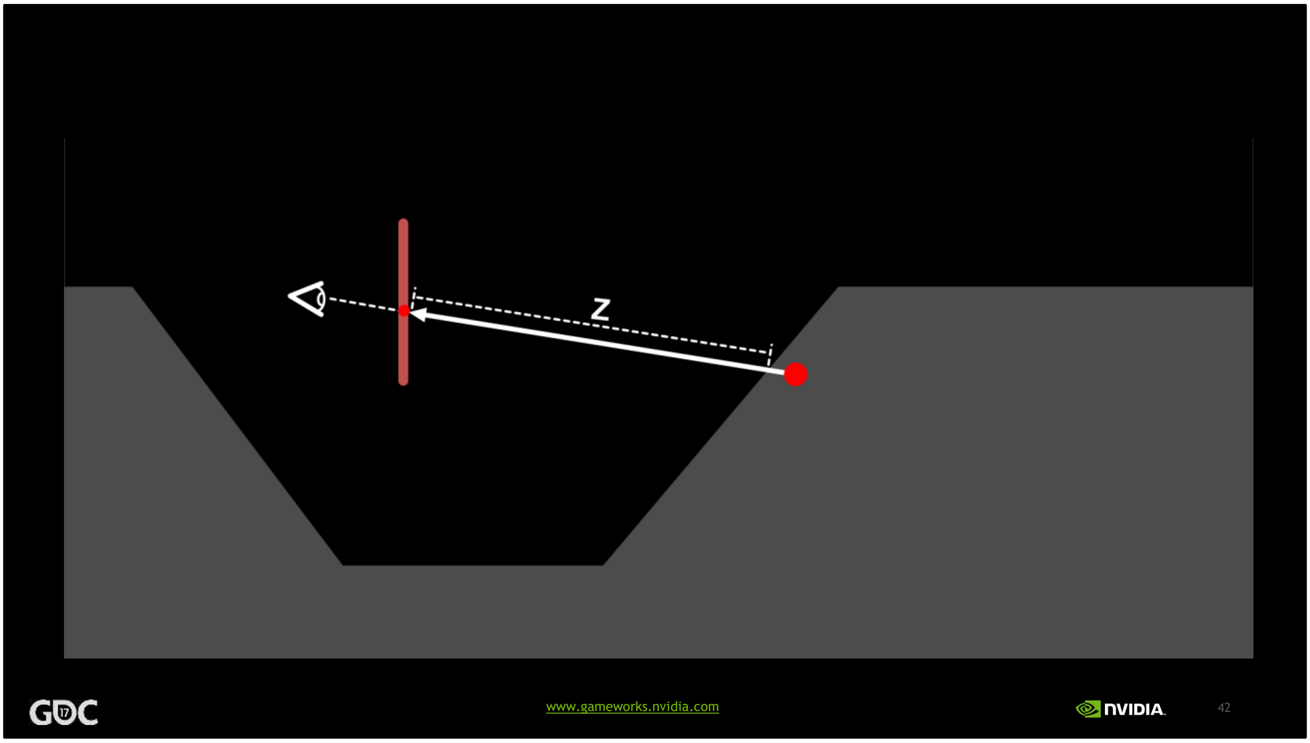
I’d like to quickly recap how it works.



The top red circle is the initial position of the particle.

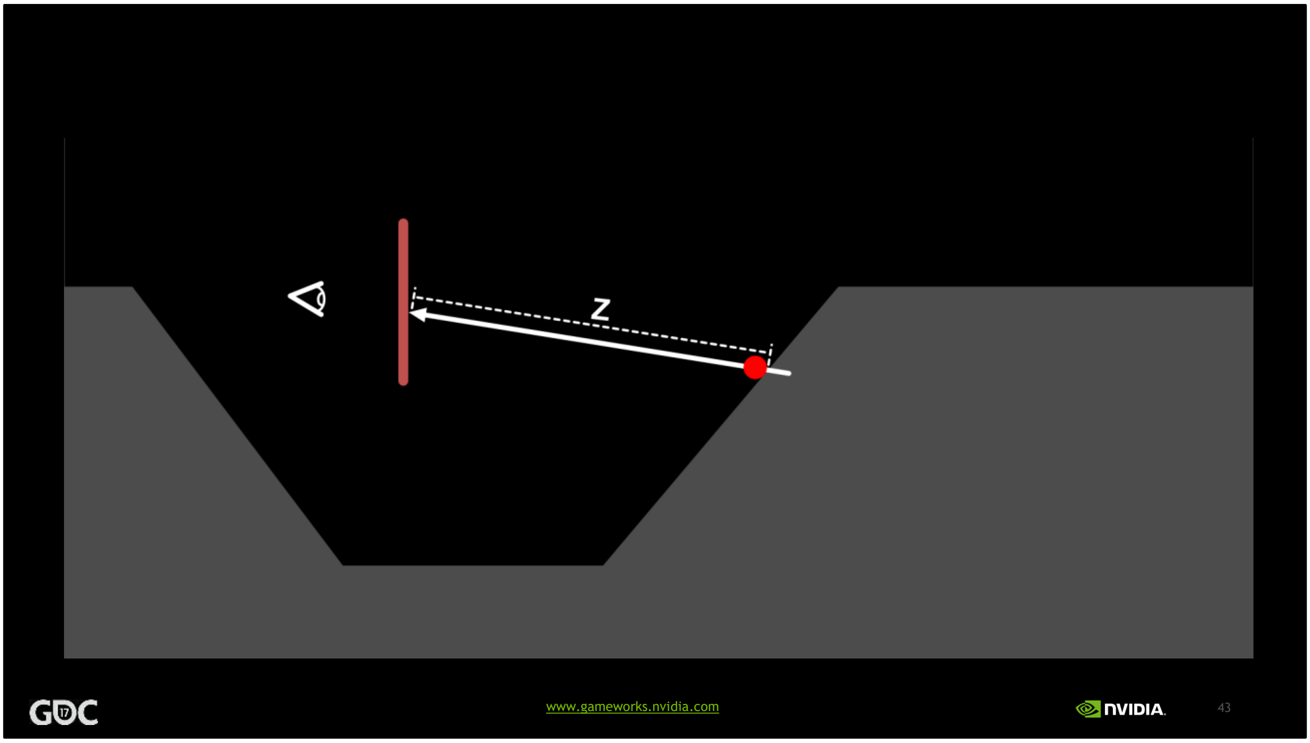
We step time to give the new position, the bottom red circle. In the new position the particle is penetrating the ground.

On the left you can see the camera, and the projection plane.

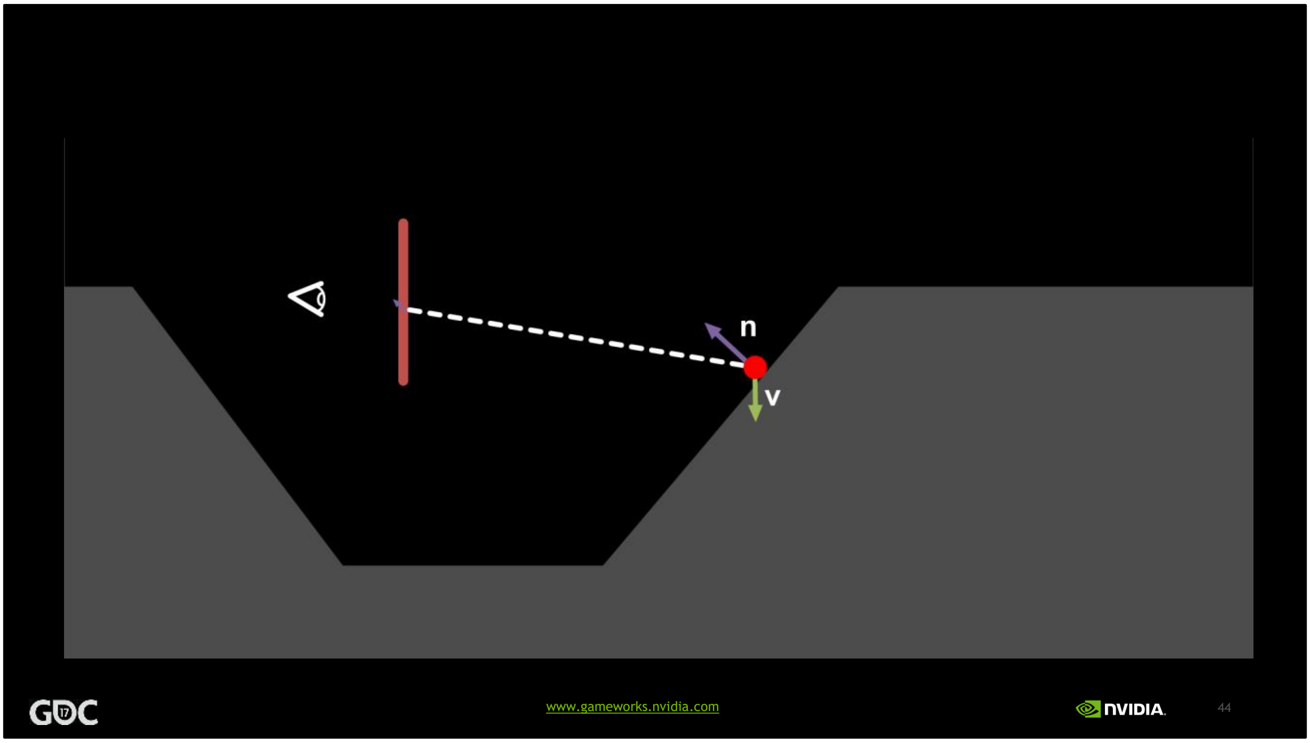


Your goal is to move the particle so it is no longer penetrating the ground.

In this simple case, the depth buffer tells you how far away the ground is for every point on the screen.

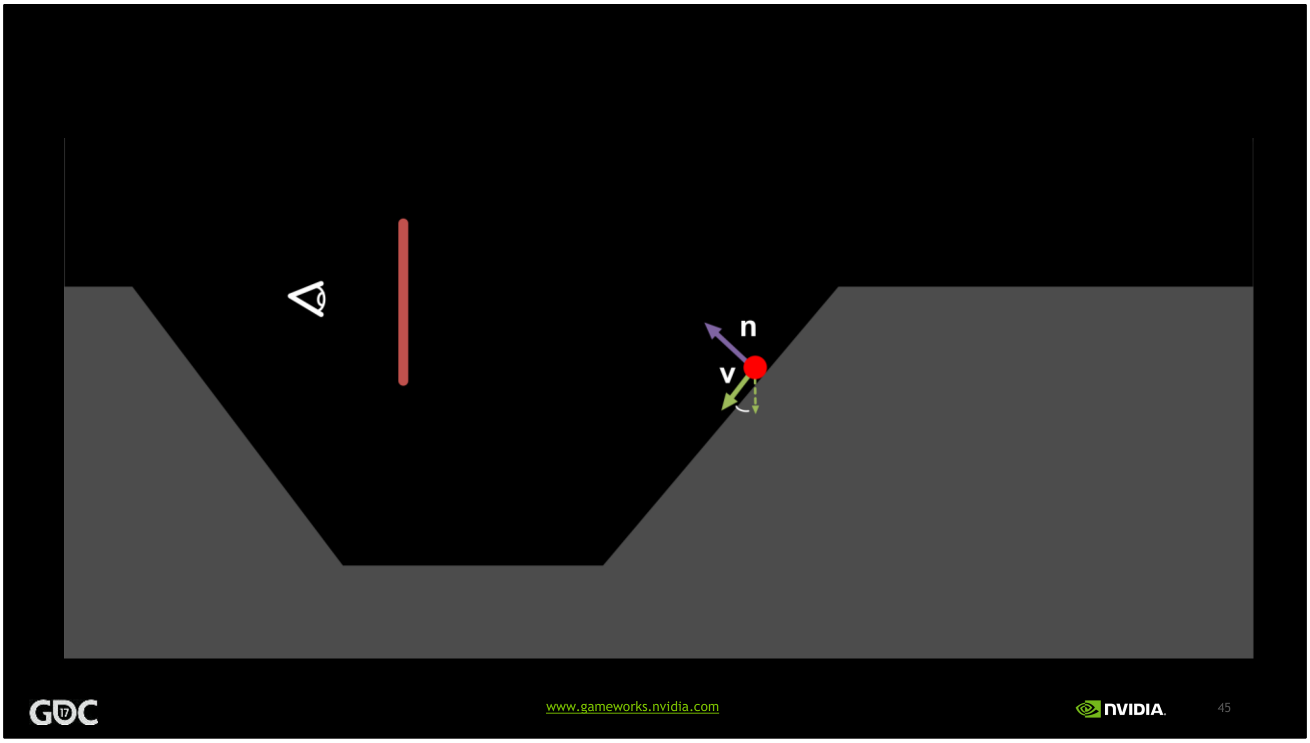


So you can project the particle into screen space, look up the correct distance, z, and move the particle along the ray so that it is at this distance.

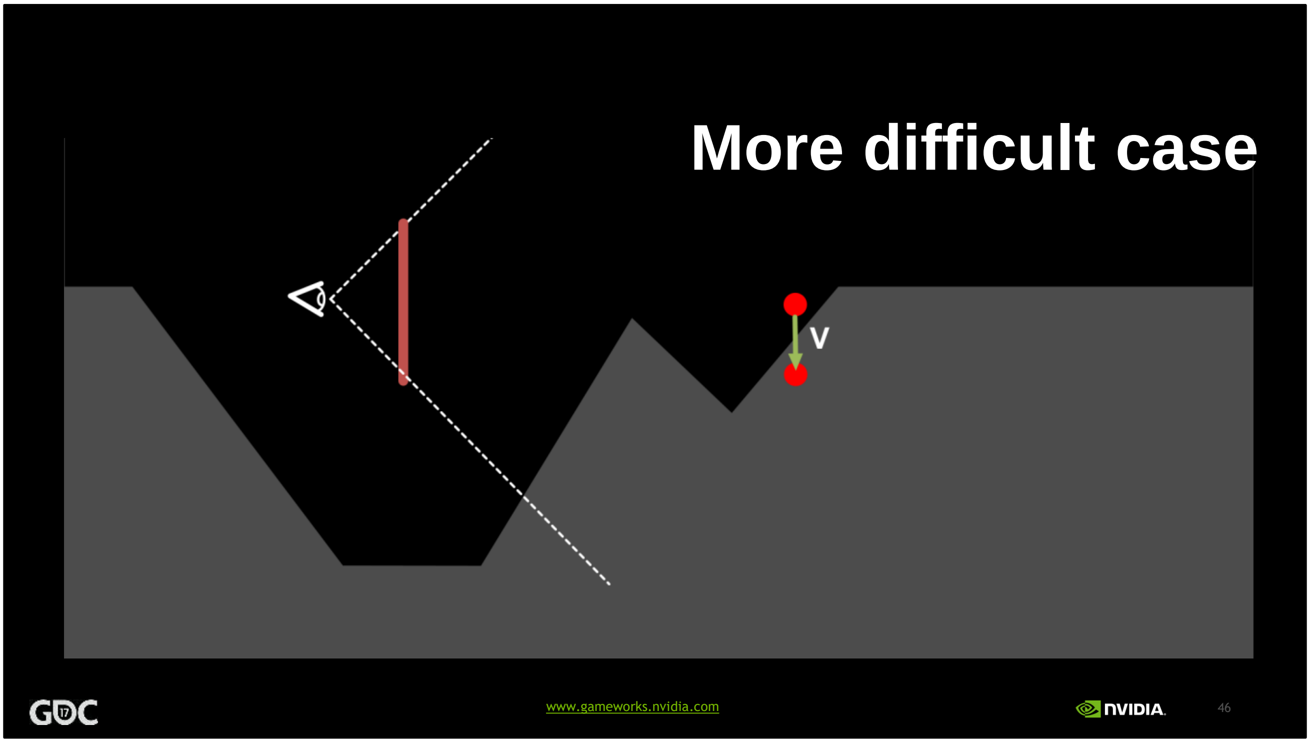


Next you need to fix up the velocity to either make the particle bounce, or slide down the hill.

For this you need an approximation of the normal, which you can also get from the depth buffer.



Here you can see the new velocity.

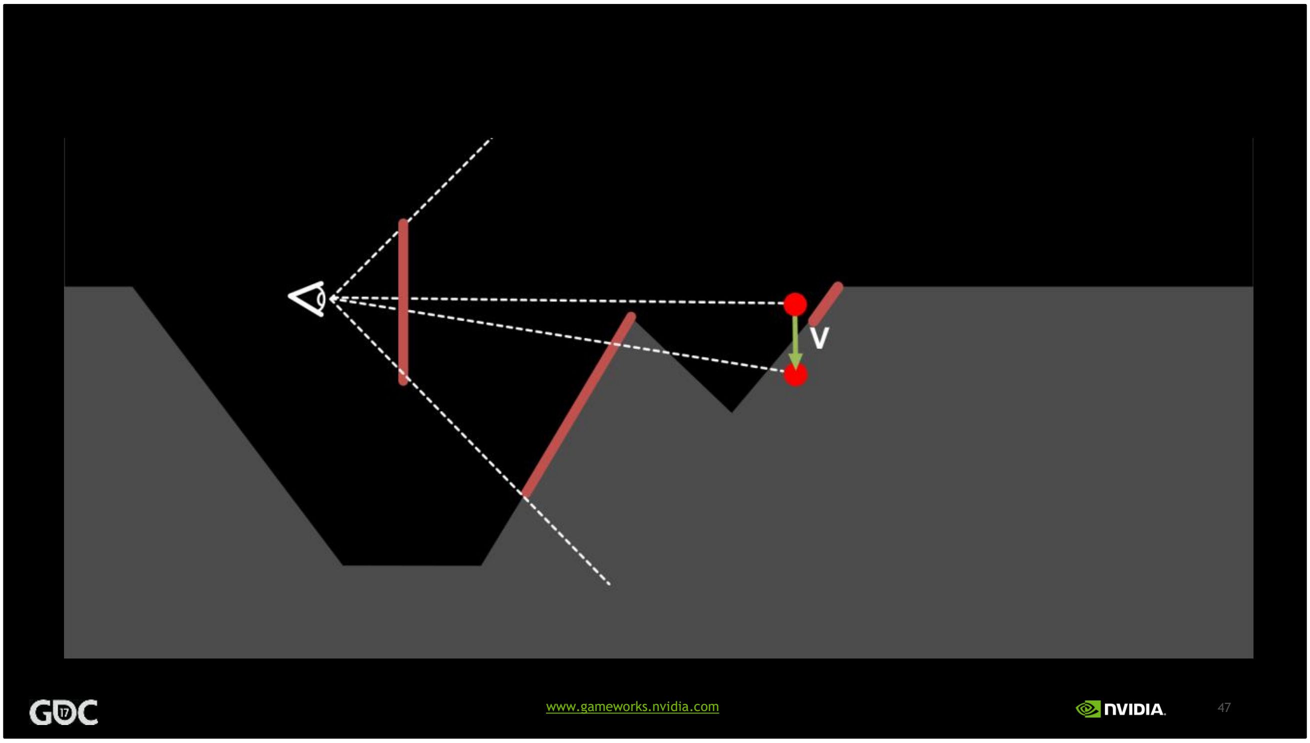


More difficult case

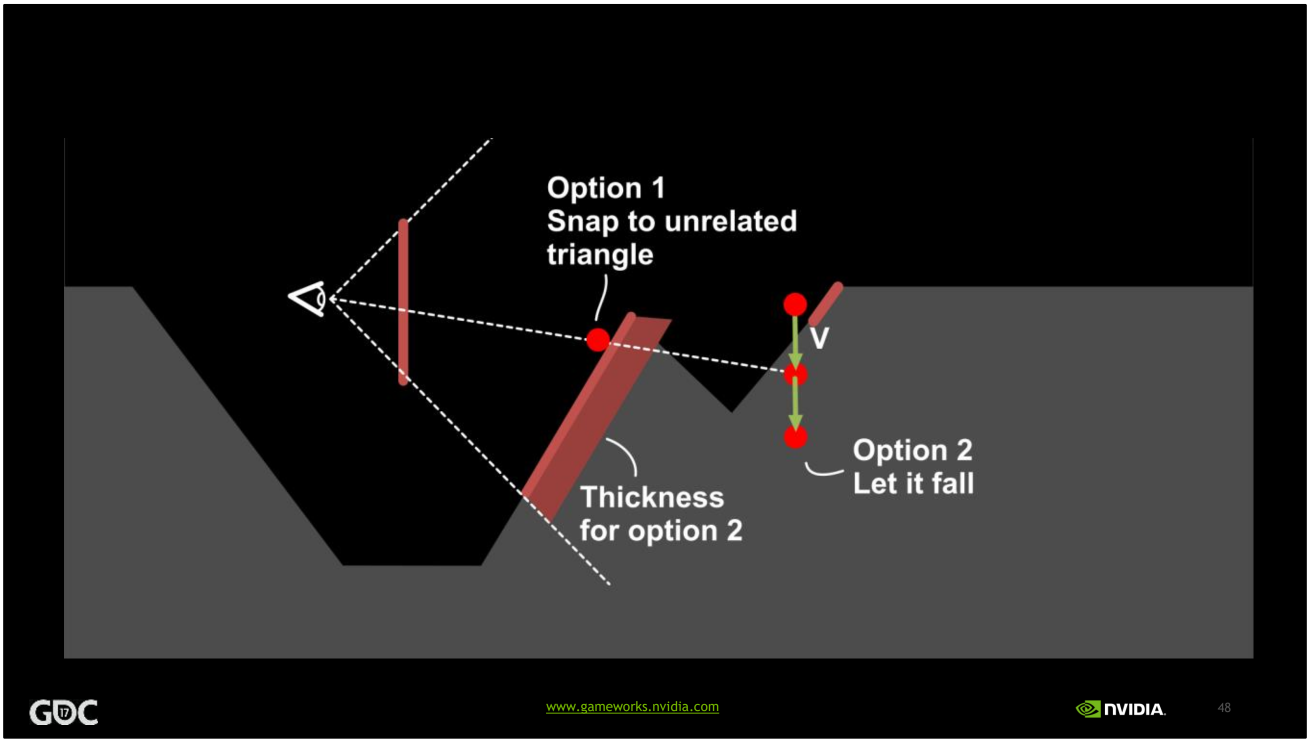
www.gameworks.nvidia.com

46

Here you can see a more difficult case, the particle is behind the geometry that is closest to the camera.



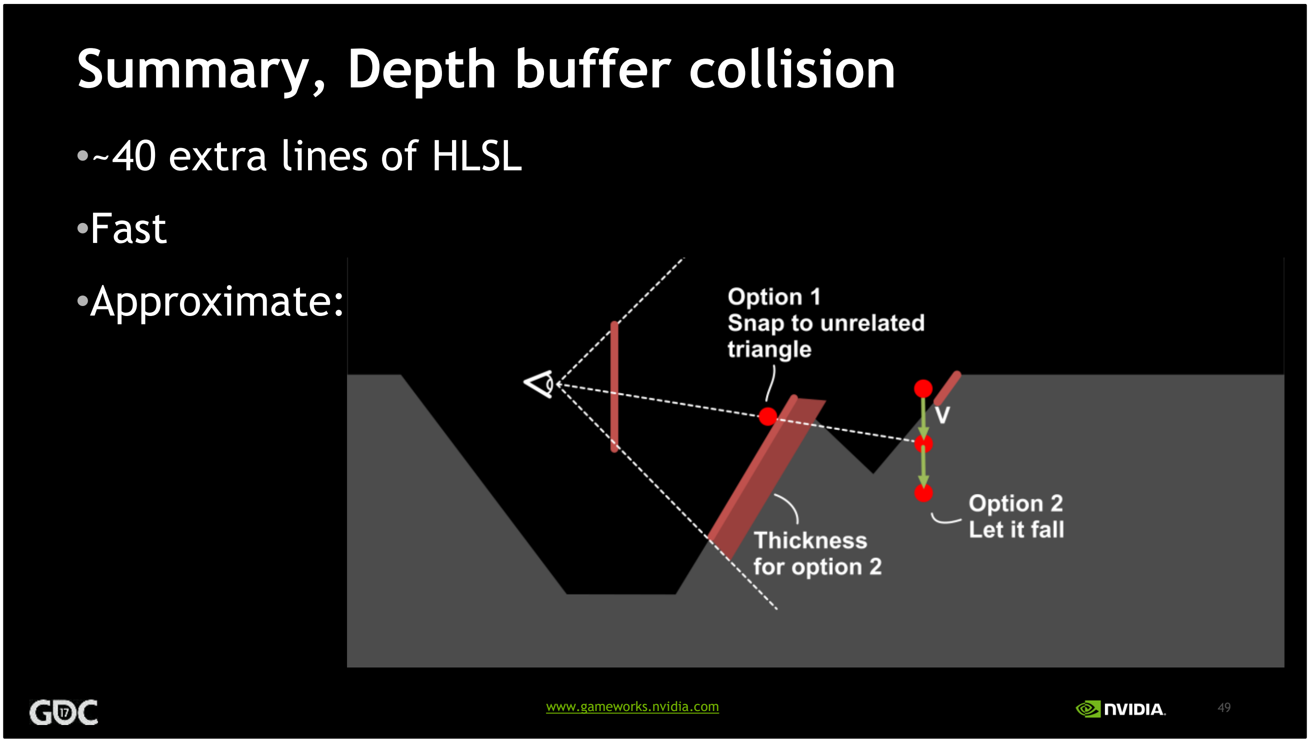
The red lines show what the triangles the depth buffer sees.



If you ignore this problem, then the particle will snap to the triangle closest to the camera, and that will look really weird.

The other option is to let the particle fall behind the triangle, and then perhaps kill it.

To decide whether the particle has really fallen behind or not, you'll need to give the geometry a thickness, shown as the red polygon.



Summary, Depth buffer collision

- ~40 extra lines of HLSL
- Fast
- Approximate:

This method is very simple to implement in HLSL, and is very fast, which is why games use it.

But, you just saw the downside.

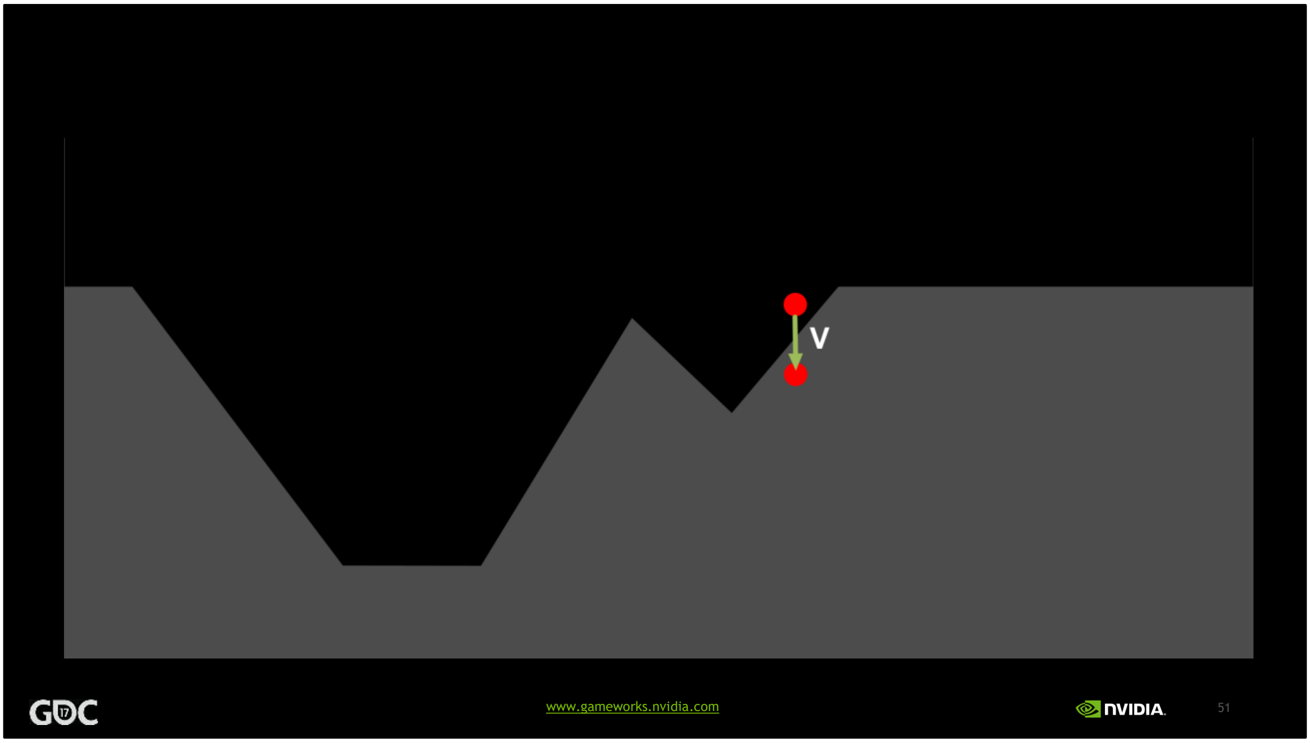
Solid ground (general)

www.gameworks.nvidia.com

50

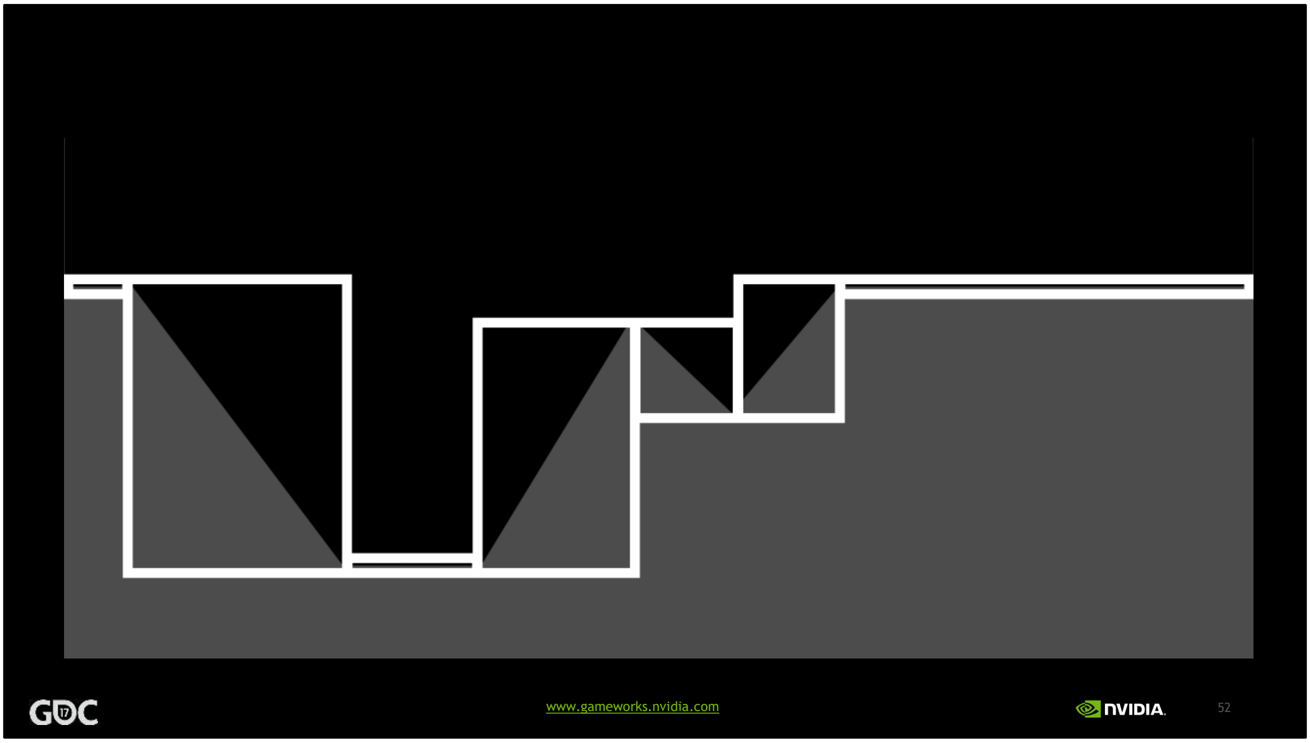
Next I’m going to try to convince you that for not much extra GPU cost you can do ground collision in 3D instead using a BVH tree.

This is what we do in Flex.

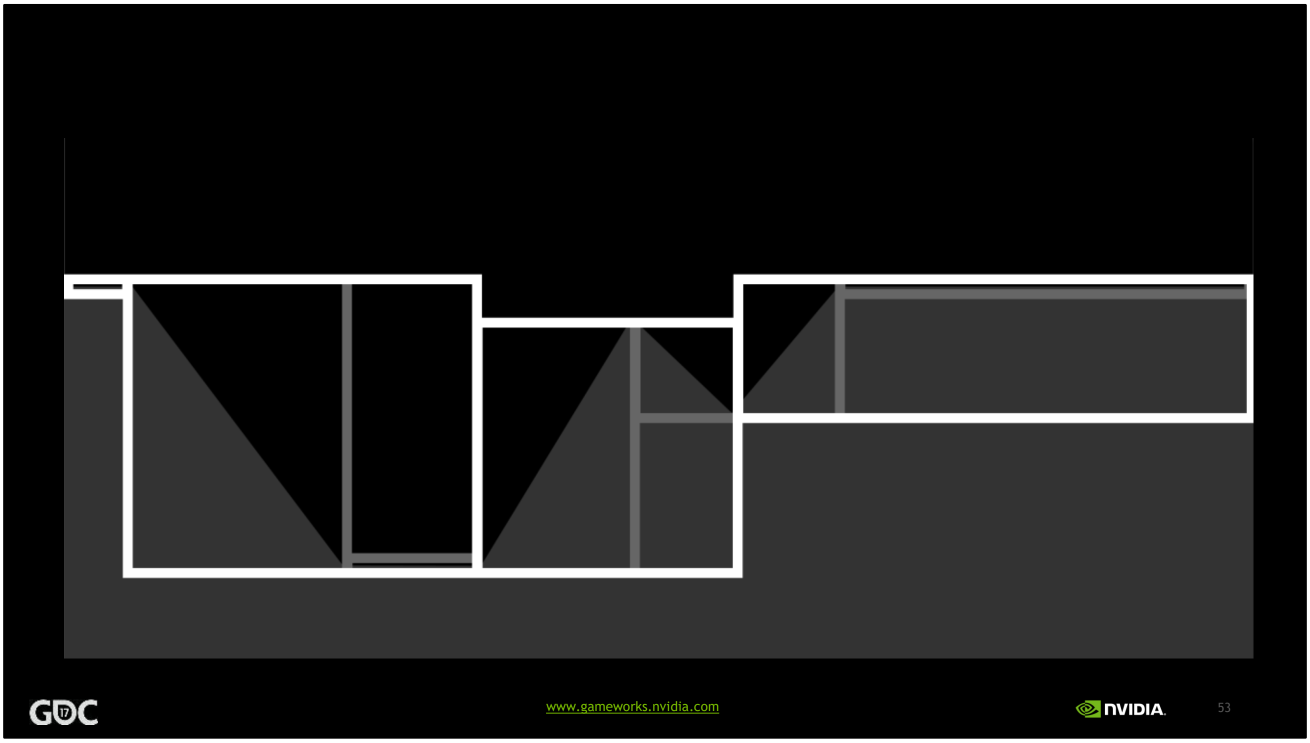


Same example as before.

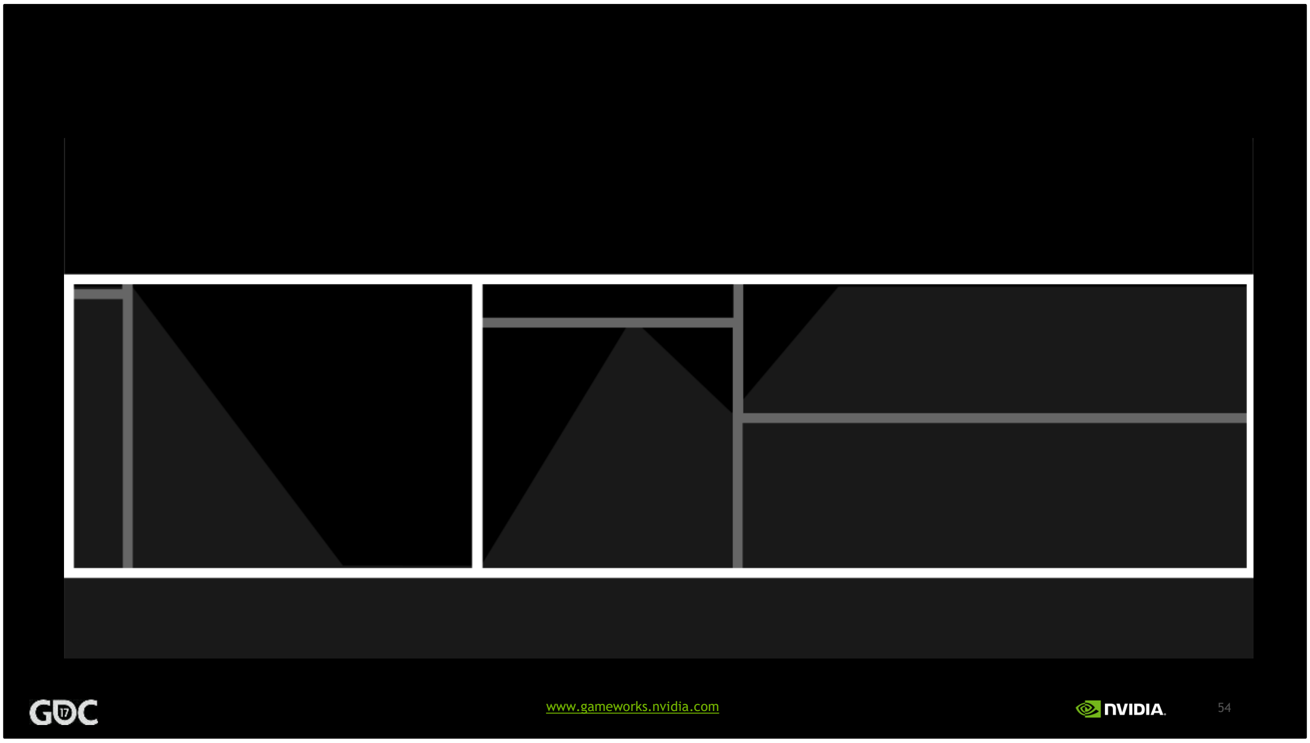
The problem before was the depth buffer knew only about the triangle closest to the camera, not the triangle closest the particle.



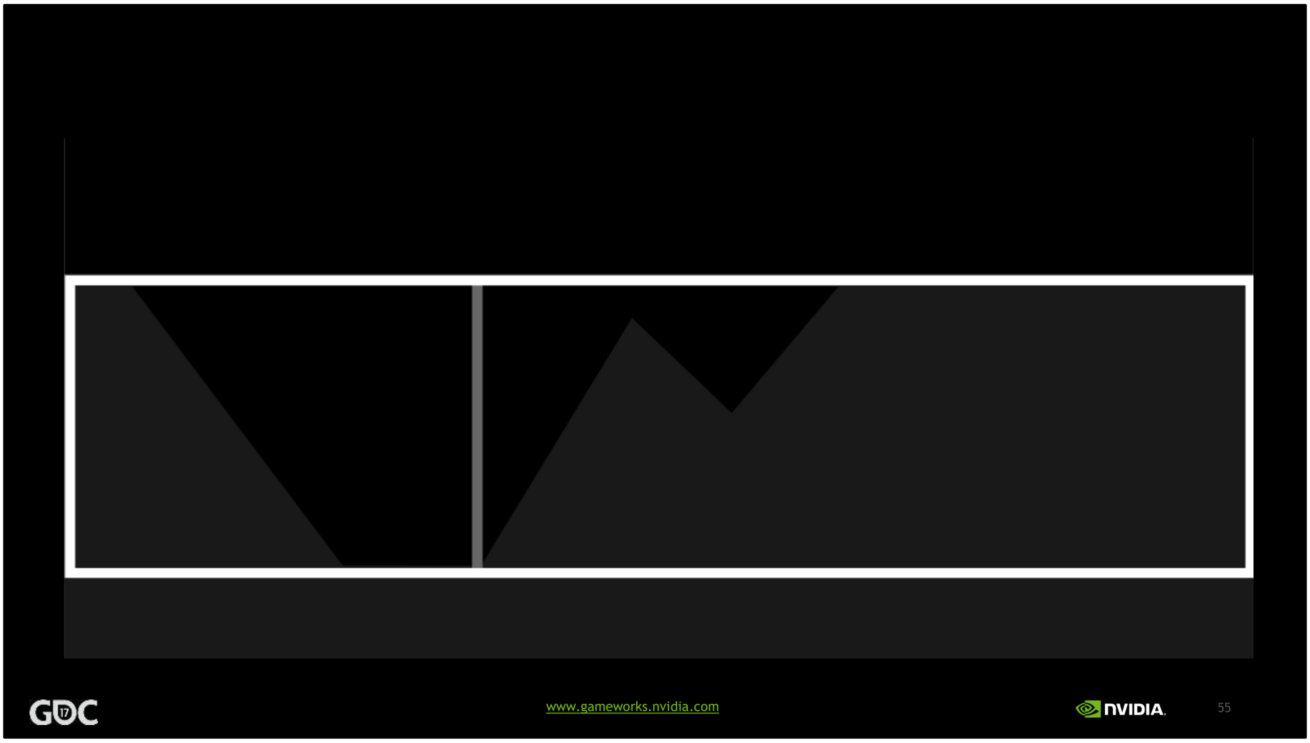
To find the triangle closest the particle, you first put axis aligned bounding boxes around each.



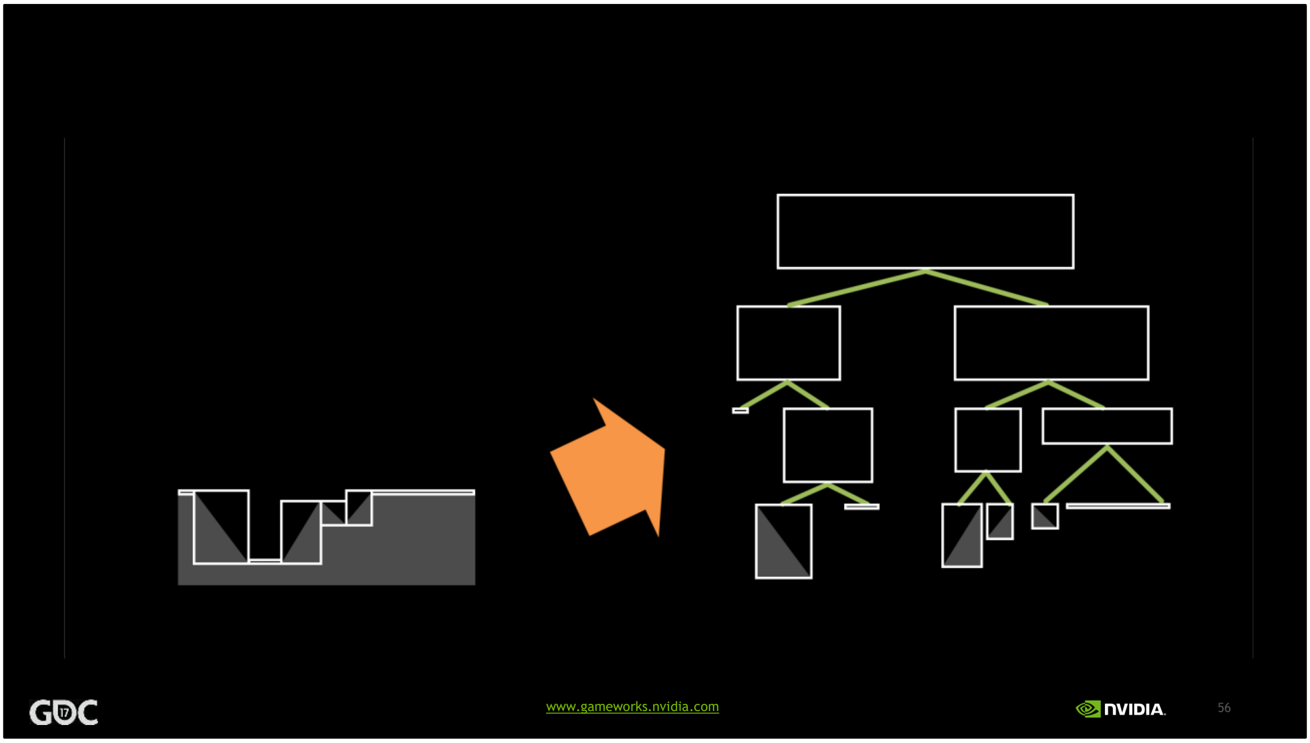
Then, merge adjacent pairs of bounding boxes.



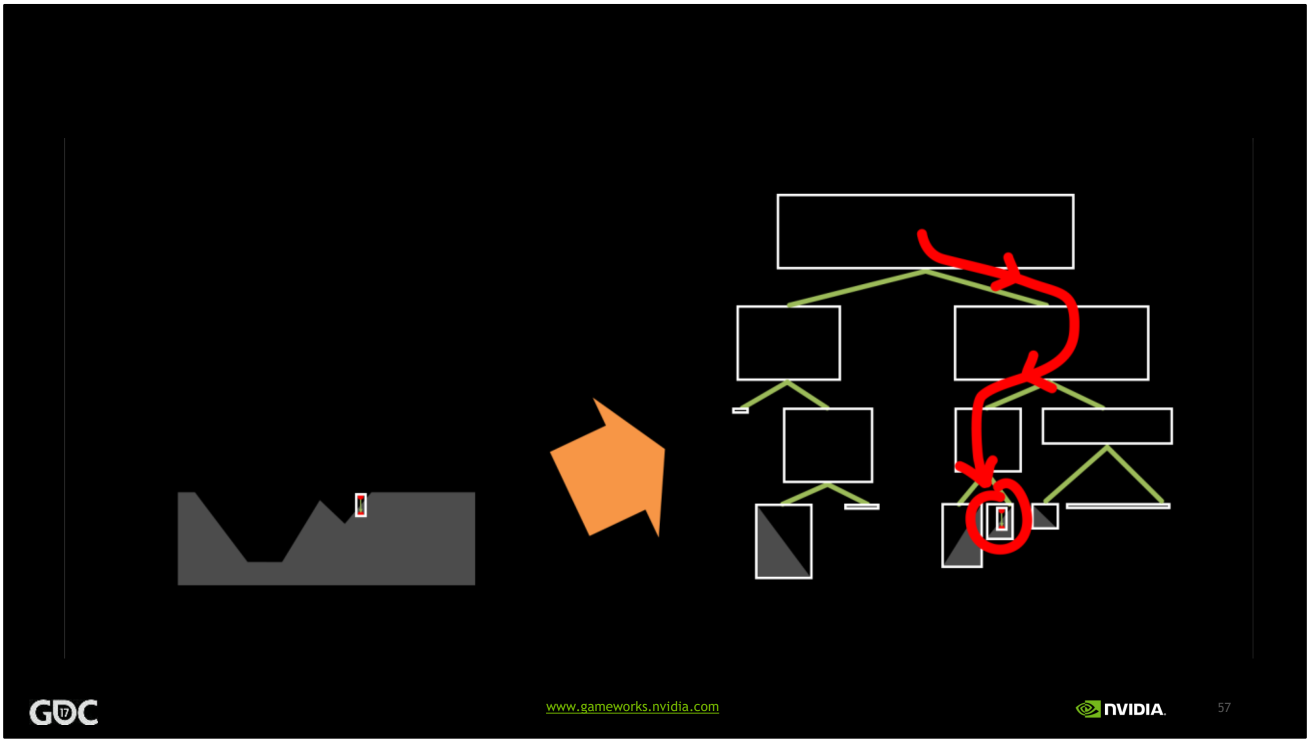
And you do this recursively.



Until you have a single bounding box covering the whole world.



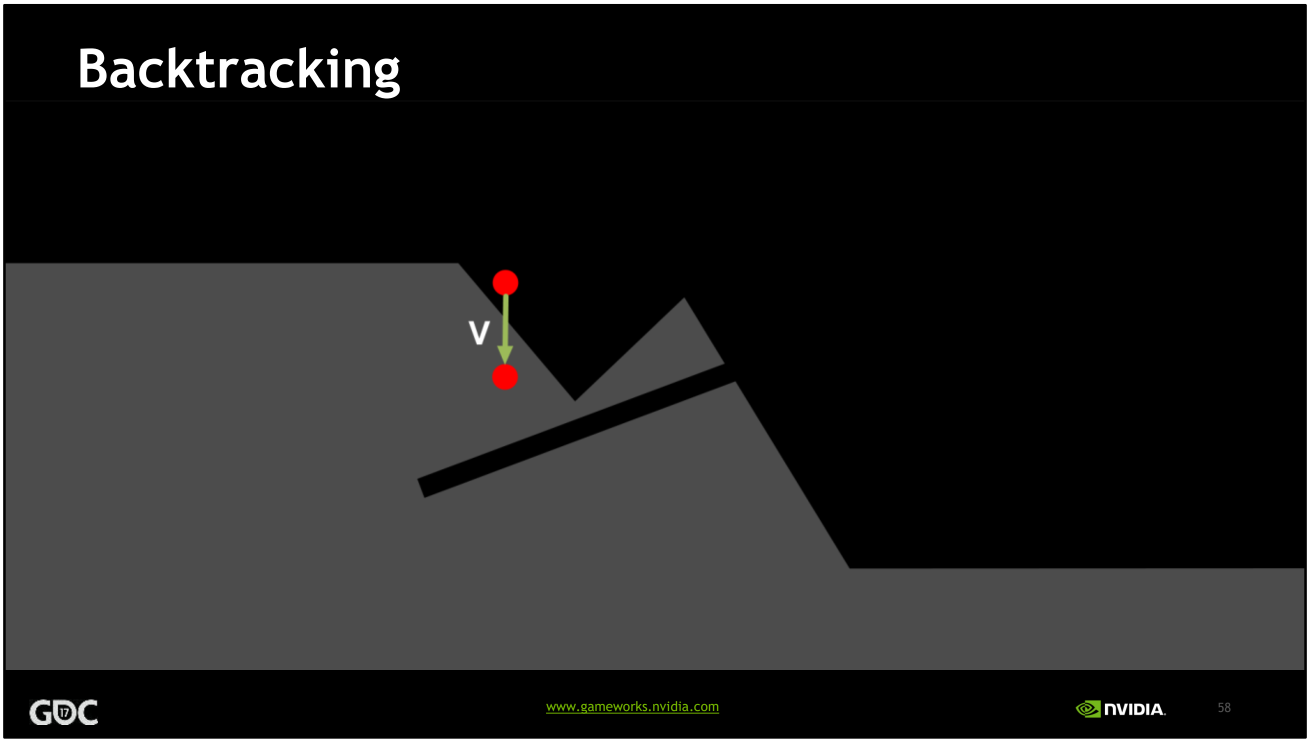
You then put these bounding boxes in a tree with the triangles at the leaves. This is called a Bounding Volume Hierarchy, or BVH.



To find the triangle nearest the particle, you first make a query bounding box that contains the new and old particle positions.

You then traverse the tree, at each node choosing the child whose bounding box overlaps the query bounding box.

When you get to the leaf, you do a line segment vs triangle test to find how far to move the particle to move it outside the triangle.



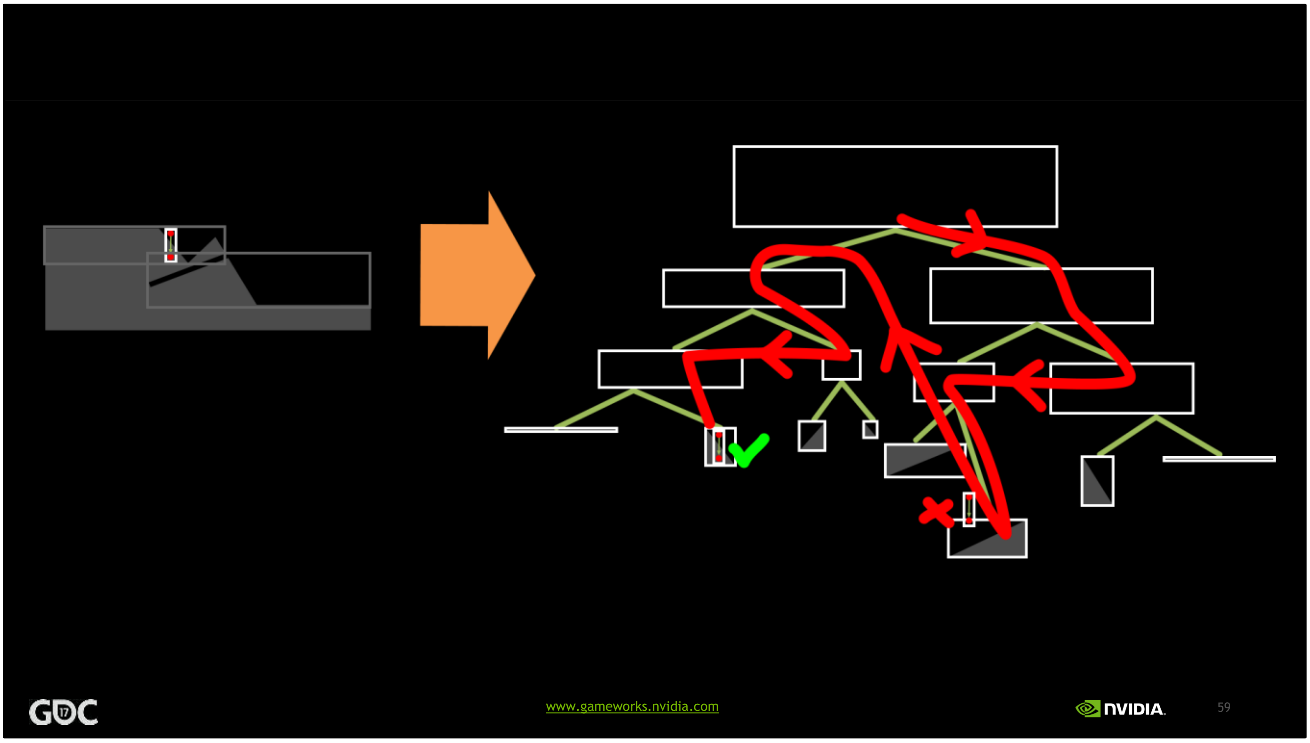
Backtracking

A property of BVHs that distinguish them from other acceleration structures like octrees or BSP trees is that they don’t partition space.

Specifically, the bounding boxes of each node’s children are allowed to overlap.

To illustrate this, I’ve made a contrived example…. Oh no, someone has cut a mine shaft into the mountain.

In the next slide you’ll see this causes the children of the root node to overlap.



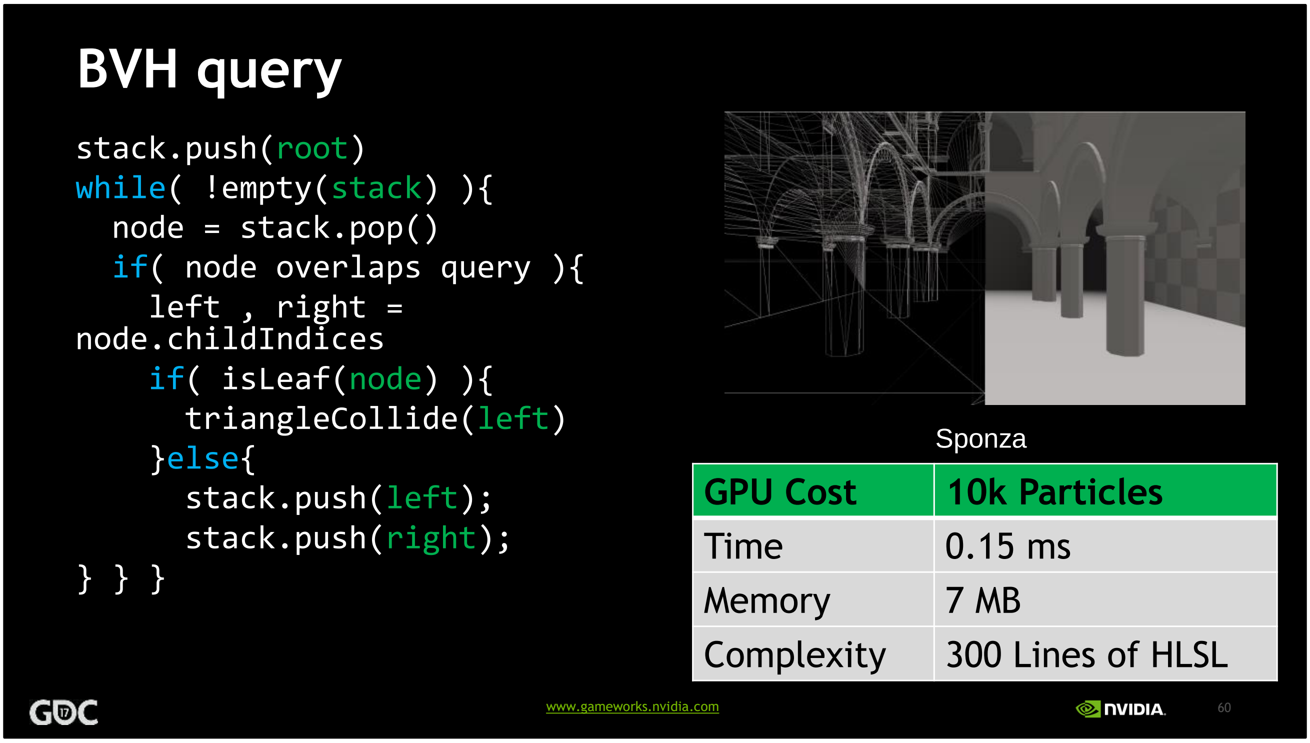
In Flex we traverse the right child first.

In this example, the right child is the one containing the mine shaft, and this isn’t the region that is intersecting the particle.

You traverse the mine shaft branch of the tree until you get to a leaf and realize you’re in the wrong place.

You need a way to backtrack to the left child of the root node, which will lead you to the correct triangle.

To do this backtracking you need to use a stack that, perhaps surprisingly, can be done fast enough in HLSL on a GPU.



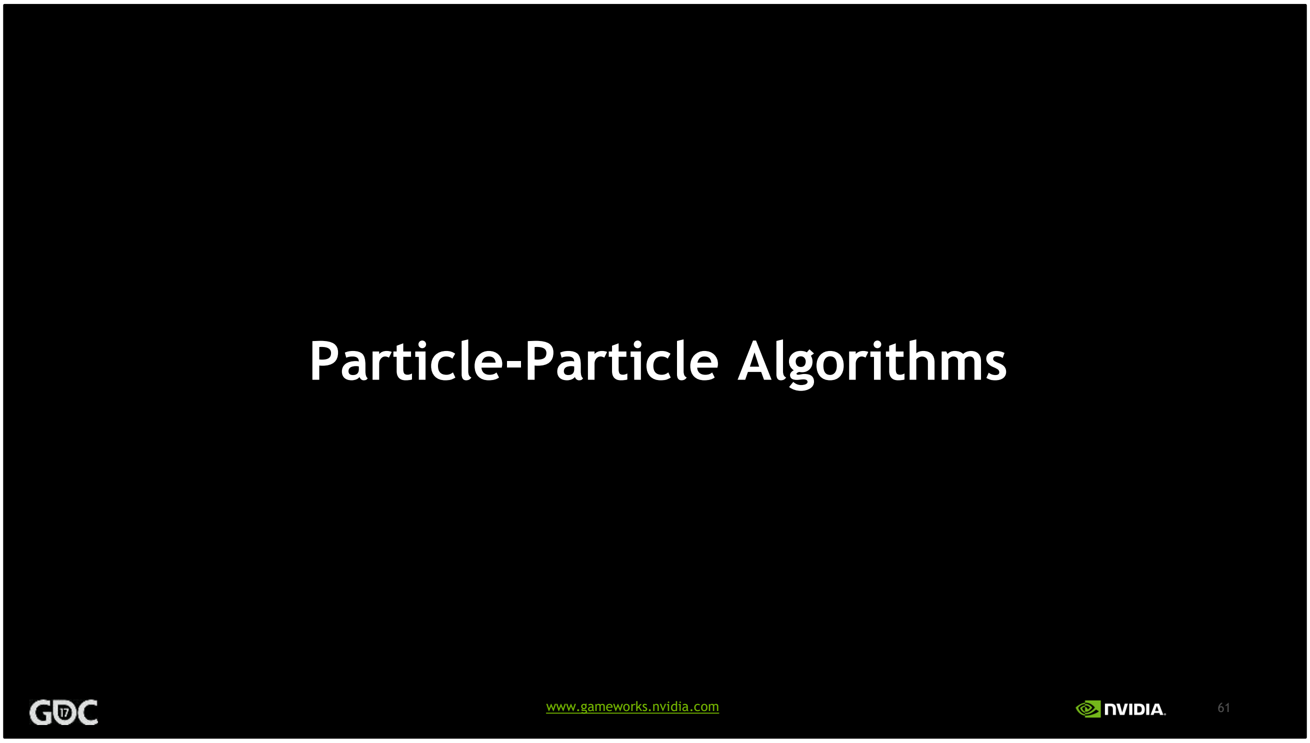
BVH query

```
stack.push(root)
while( !empty(stack) ){
  node = stack.pop()
  if( node overlaps query ){
    left , right =
node.childIndices
    if( isLeaf(node) ){
      triangleCollide(left)
    }else{
      stack.push(left);
      stack.push(right);
} } }
```

Sponza

GPU Cost	10k Particles
Time	0.15 ms
Memory	7 MB
Complexity	300 Lines of HLSL

Here is the BVH traversal algorithm implemented in HLSL in Flex, along with its cost.



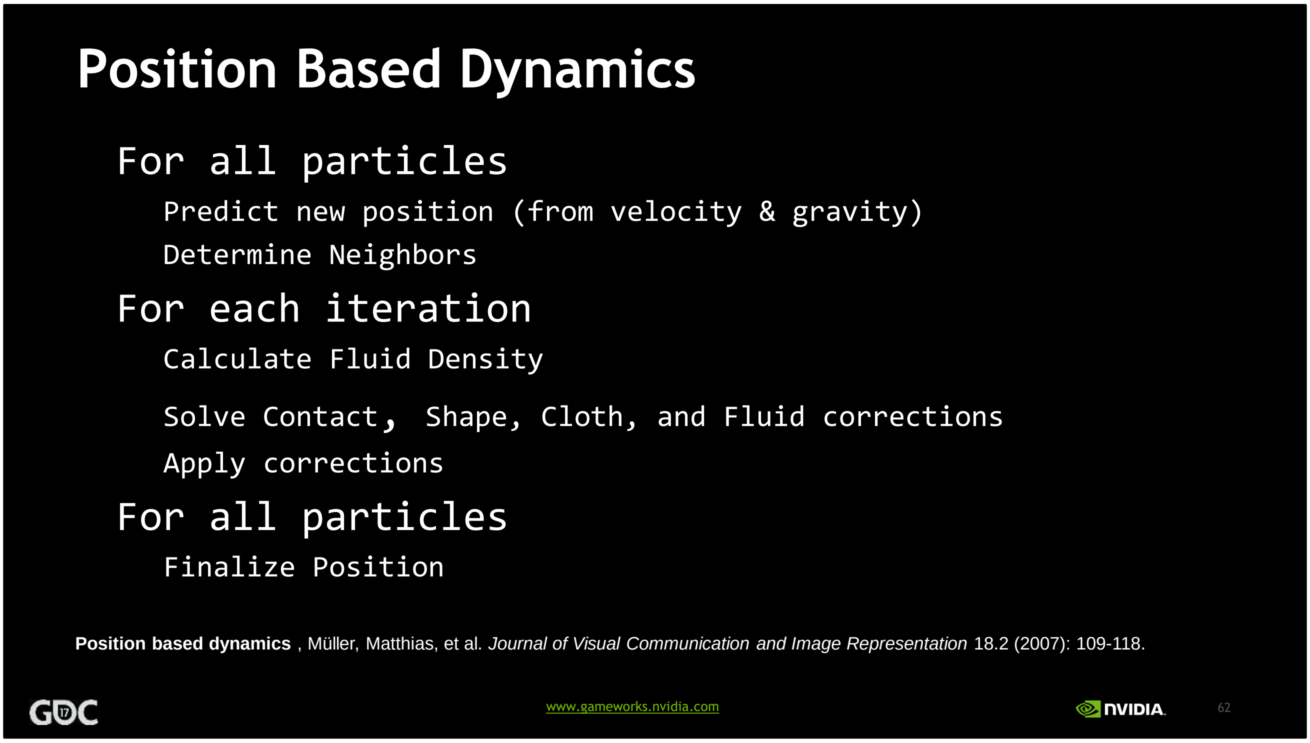
Particle-Particle Algorithms

That was the first step to bridge the gap, higher quality ground collision for not much extra cost using BVH.

The next step is enabling each particle to find its neighbors.

Once you can update a particle based on the position of its neighbors, you can use different update rules to simulate rigid bodies, fluids or cloth.

Hammad will describe that next.



Position Based Dynamics

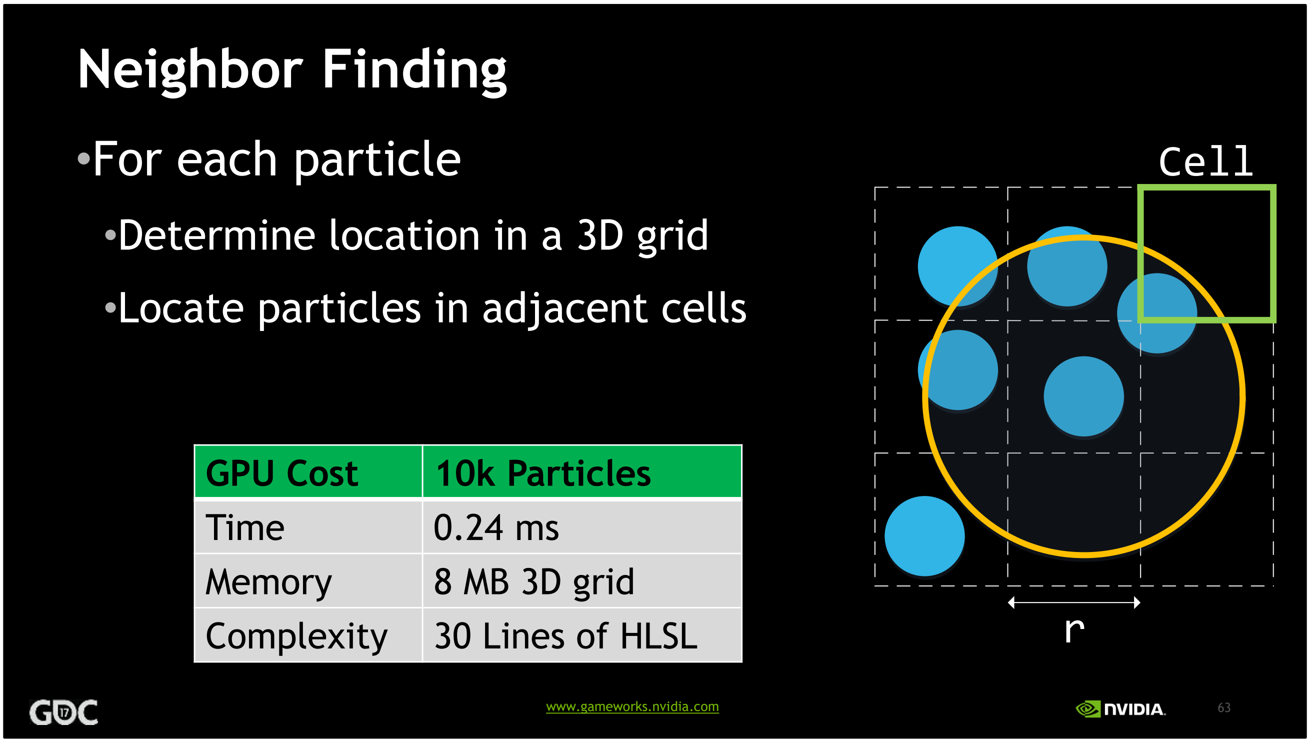
```
For all particles
    Predict new position (from velocity & gravity)
    Determine Neighbors
For each iteration
    Calculate Fluid Density
    Solve Contact, Shape, Cloth, and Fluid corrections
    Apply corrections
For all particles
    Finalize Position
```

Position based dynamics , Müller, Matthias, et al. *Journal of Visual Communication and Image Representation* 18.2 (2007): 109-118.

Position based dynamics has gained popularity in the gaming and visual effects communities because it is fast, robust and position stable.

This means that the positions of each particle is manipulated directly leading to a simulation that is visually stable even at low iterations.

The physics engine has three stages, Determining particle neighbors, iteratively solving the constraints in the system to determine particle corrections and then finally applying the corrections determined by the iterative solve.



Neighbor Finding

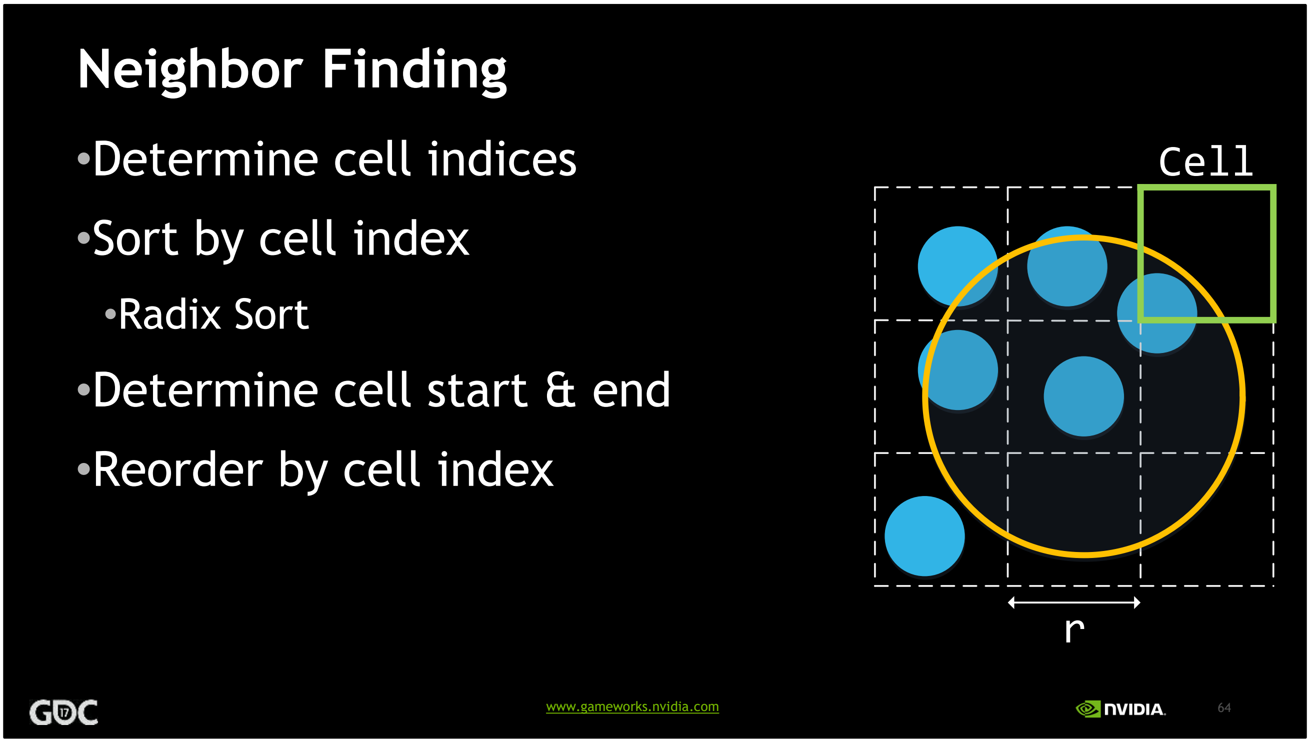
- For each particle
  - Determine location in a 3D grid
  - Locate particles in adjacent cells

GPU Cost	10k Particles
Time	0.24 ms
Memory	8 MB 3D grid
Complexity	30 Lines of HLSL

GDC www.gameworks.nvidia.com NVIDIA 63

Because we handle collisions between particles and fluid like effects we need to know for each particle what the neighboring particles are.

To do this we use a three dimensional grid as our data structure. We map the location of each particle onto a grid cell and by reversing the particle-cell mapping we can locate particles in adjacent cells.



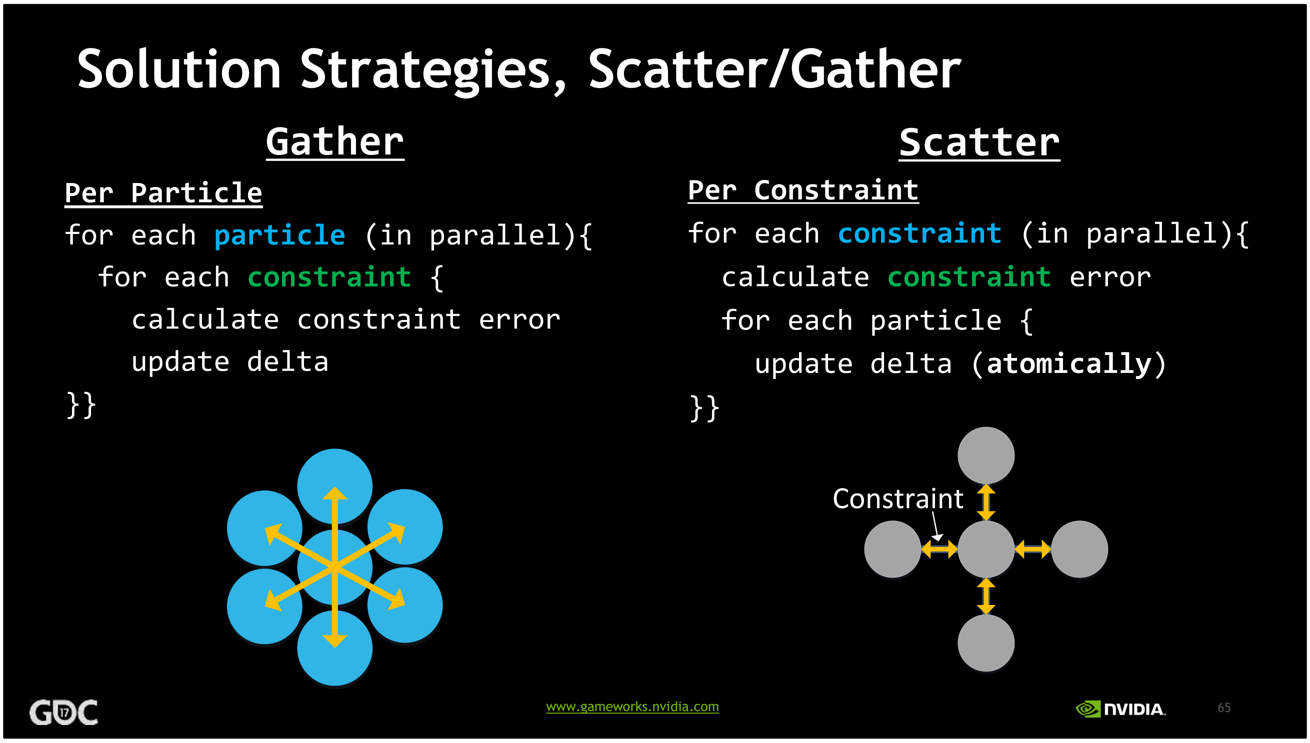
Neighbor Finding

- Determine cell indices
- Sort by cell index
  - Radix Sort
- Determine cell start & end
- Reorder by cell index

For each particle we determine its cell index, then sort this particle-cell mapping by the cell number, count the number of particles per cell and then reorder the particles by the cell index.

This does two things, first particles that are close to each other physically will be relatively close in memory.

Second we can find the neighbors of a particle by determining its cell number and looking up the particles in that cell in the list we generated, we can also look at particles in neighboring cells.



Solution Strategies, Scatter/Gather

Gather

Per Particle

```
for each particle (in parallel){
  for each constraint {
    calculate constraint error
    update delta
}}
```

Scatter

Per Constraint

```
for each constraint (in parallel){
  calculate constraint error
  for each particle {
    update delta (atomically)
}}
```

GDC

www.gameworks.nvidia.com

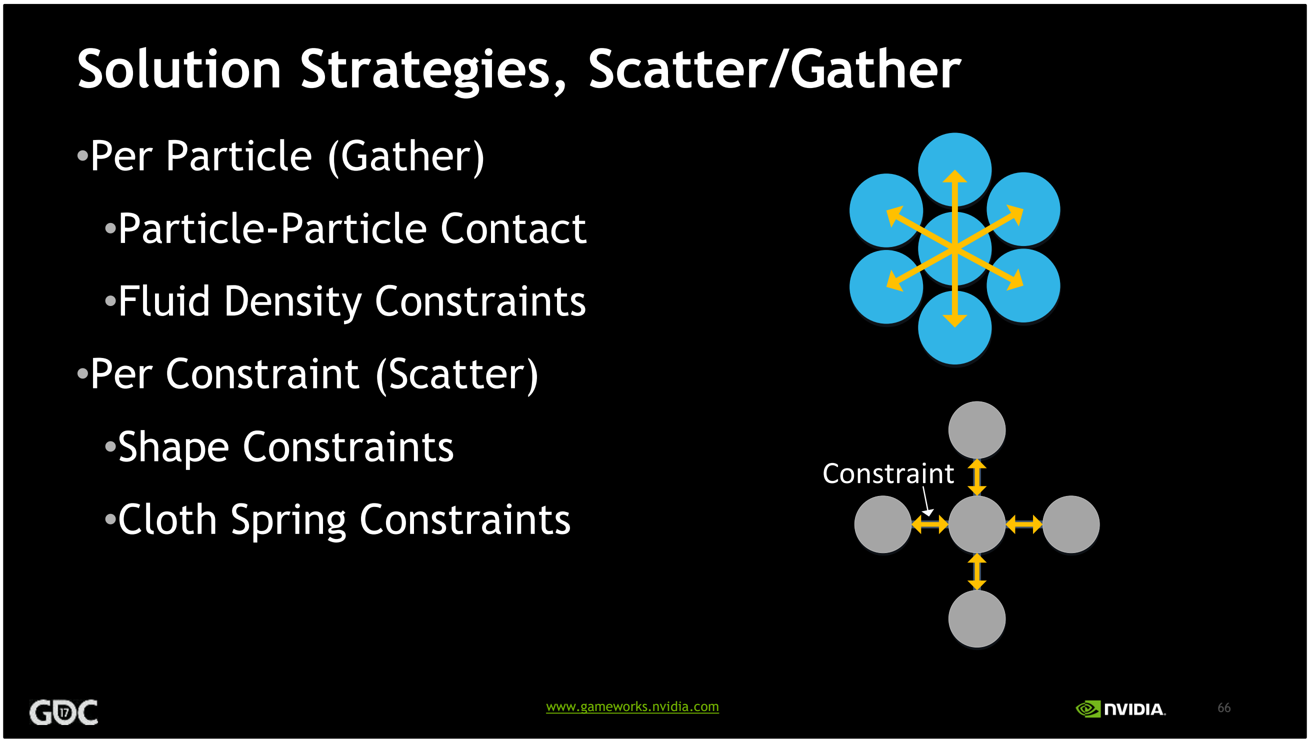
NVIDIA

65

There are two different ways to compute the position update, or delta, for a particle

One is the scatter approach where we loop over each particle and per constraint compute the position update.

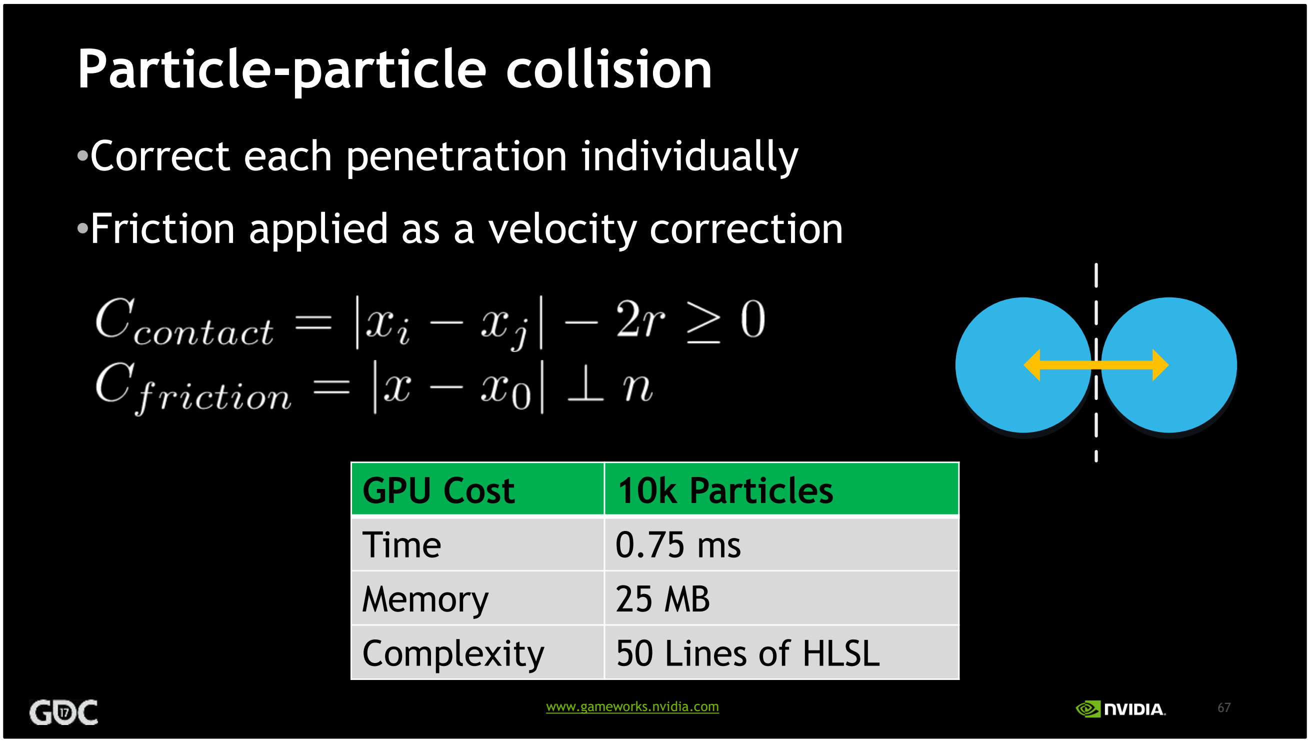
The second is that we loop over each constraint and for each particle involved apply the position update. Because multiple constraints can influence a single particle this needs to be done using atomics.



Solution Strategies, Scatter/Gather

- Per Particle (Gather)
  - Particle-Particle Contact
  - Fluid Density Constraints
- Per Constraint (Scatter)
  - Shape Constraints
  - Cloth Spring Constraints

Gather is used for per particle contacts and fluid density constraints.
Scatter is used for shape constraints and cloth spring constraints.



Particle-particle collision

- Correct each penetration individually
- Friction applied as a velocity correction

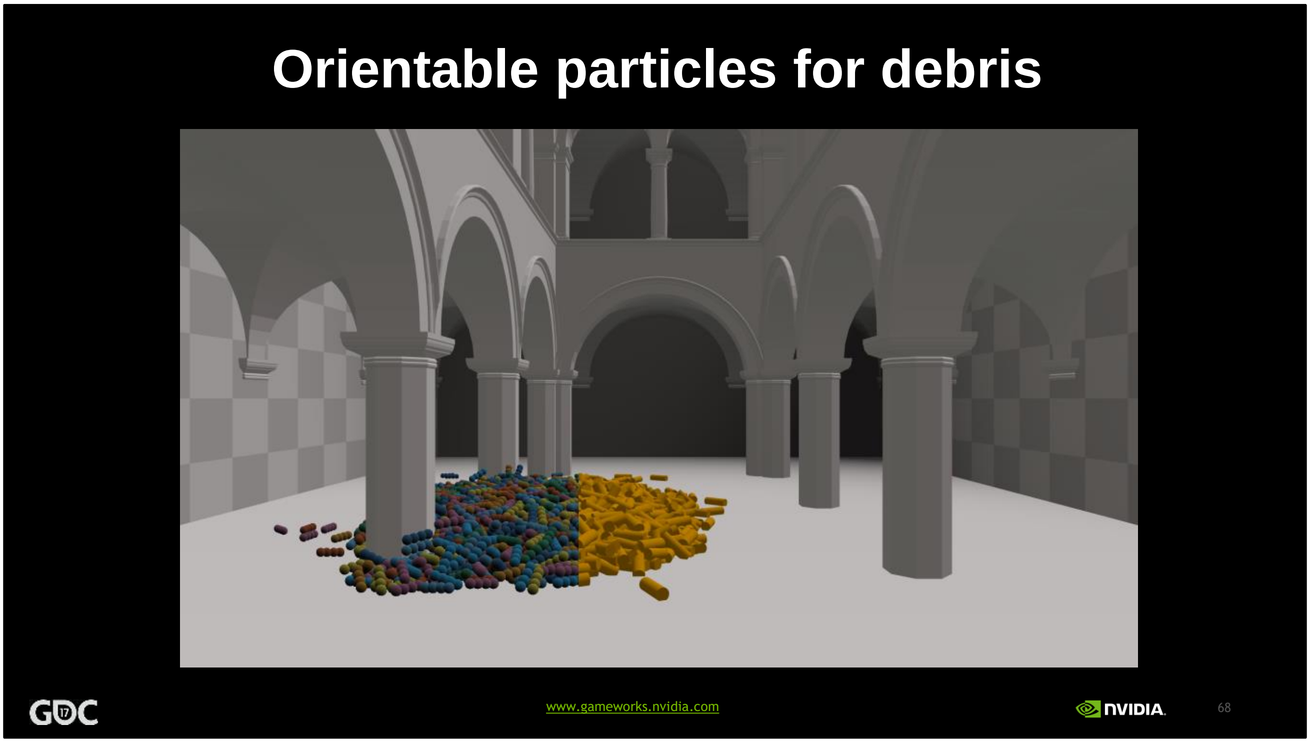
$$C_{contact} = |x_i - x_j| - 2r \geq 0$$
$$C_{friction} = |x - x_0| \perp n$$

GPU Cost	10k Particles
Time	0.75 ms
Memory	25 MB
Complexity	50 Lines of HLSL

www.gameworks.nvidia.com

For each neighbor detected during the neighbor search we determine the penetration depth, if there is a penetration then a correction is applied.

A position level friction constraint is used to approximate the coulomb friction model using the penetration depth to limit the correction.

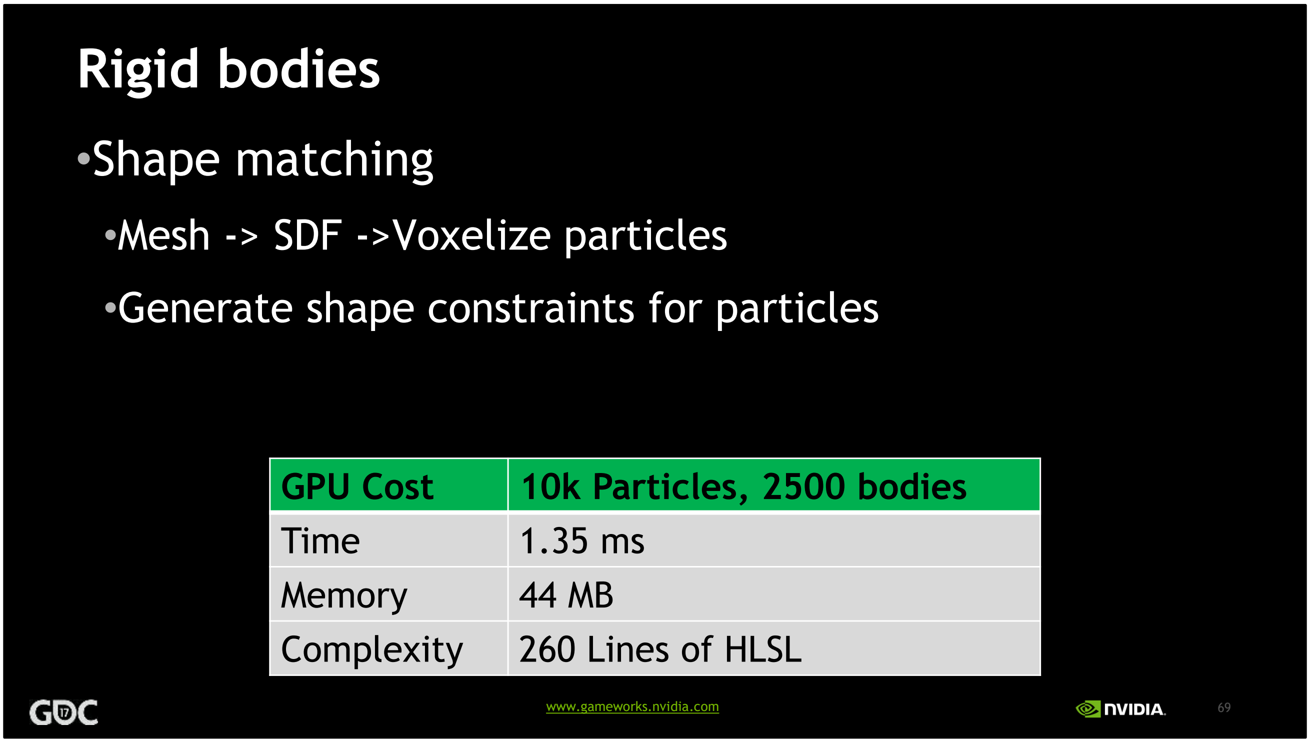


Orientable particles for debris

www.gameworks.nvidia.com

68

Next were going to look at creating solid shapes using a shape matching constraint

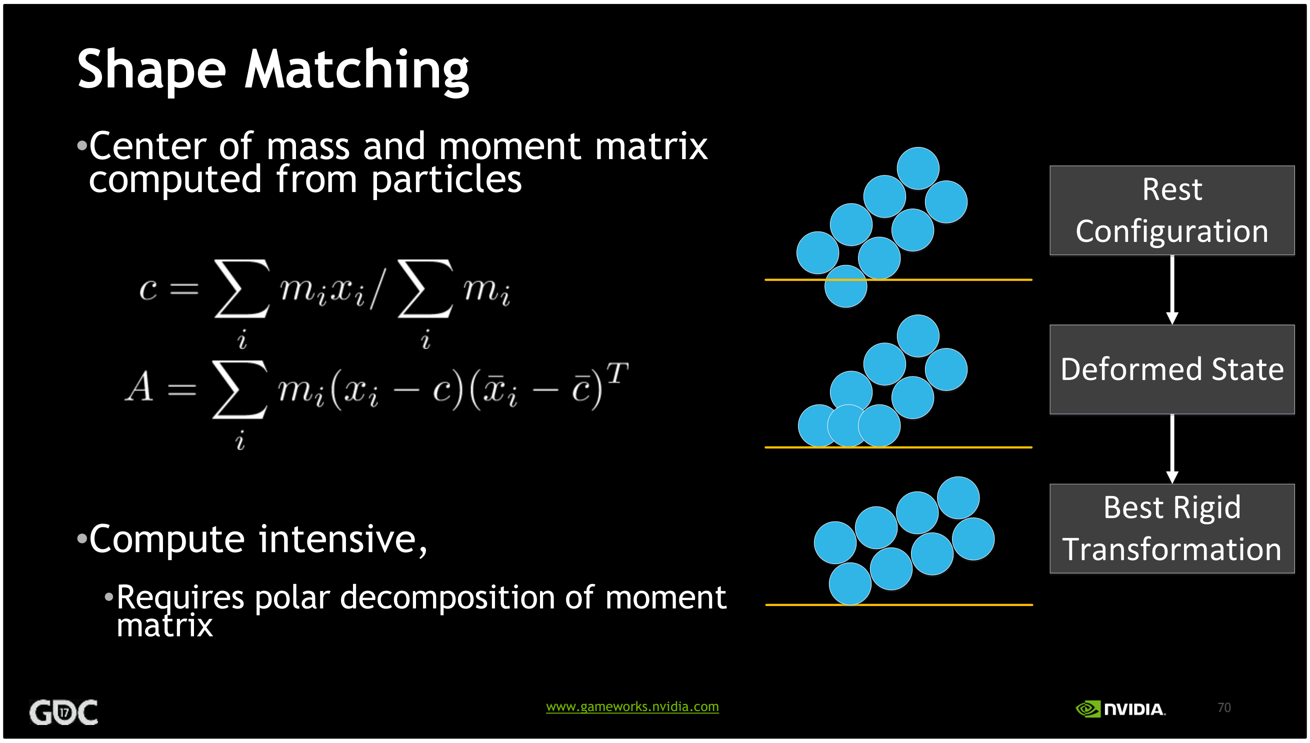


Rigid bodies

- Shape matching
  - Mesh -> SDF ->Voxelize particles
  - Generate shape constraints for particles

GPU Cost	10k Particles, 2500 bodies
Time	1.35 ms
Memory	44 MB
Complexity	260 Lines of HLSL

Starting with the triangle mesh for the shape we generate a signed distance field and voxelize it. At the center of each voxel we place a particle. This corresponds to the rest configuration of the particles in the local reference frame.



Shape Matching

- Center of mass and moment matrix computed from particles

$$c = \sum_i m_i x_i / \sum_i m_i$$

$$A = \sum_i m_i (x_i - c)(\bar{x}_i - \bar{c})^T$$

- Compute intensive,
  - Requires polar decomposition of moment matrix

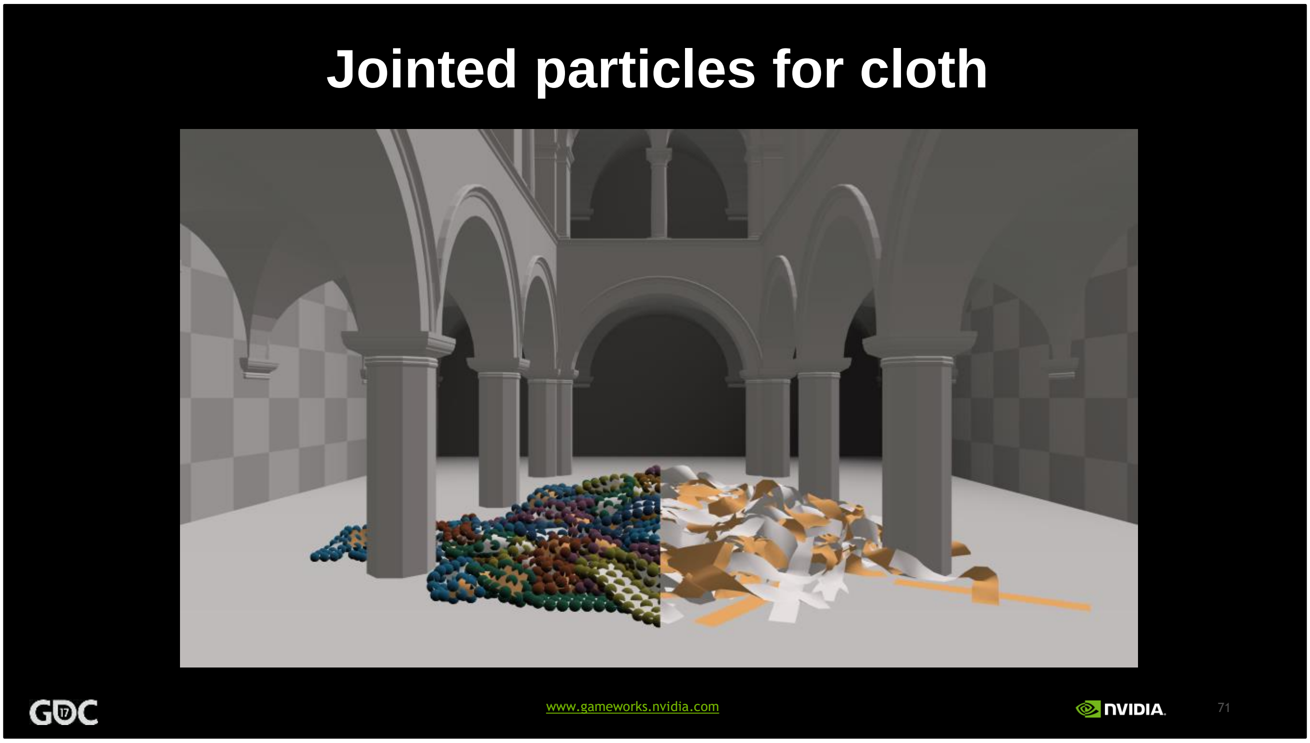
Rest Configuration

Deformed State

Best Rigid Transformation

GDC 17 www.gameworks.nvidia.com NVIDIA. 70

Here is an example of how it works. Starting with the rest configuration we collide all particles with the ground plane and move them such that they are not penetrating the ground. This makes the object lose its rigid shape, as you can see in the middle diagram. Then, using a polar decomposition we work out the best rigid transformation to pull the shape back together. This might make some of the particles penetrate again, but if we iterate we can get a rigid solution that has no penetration.

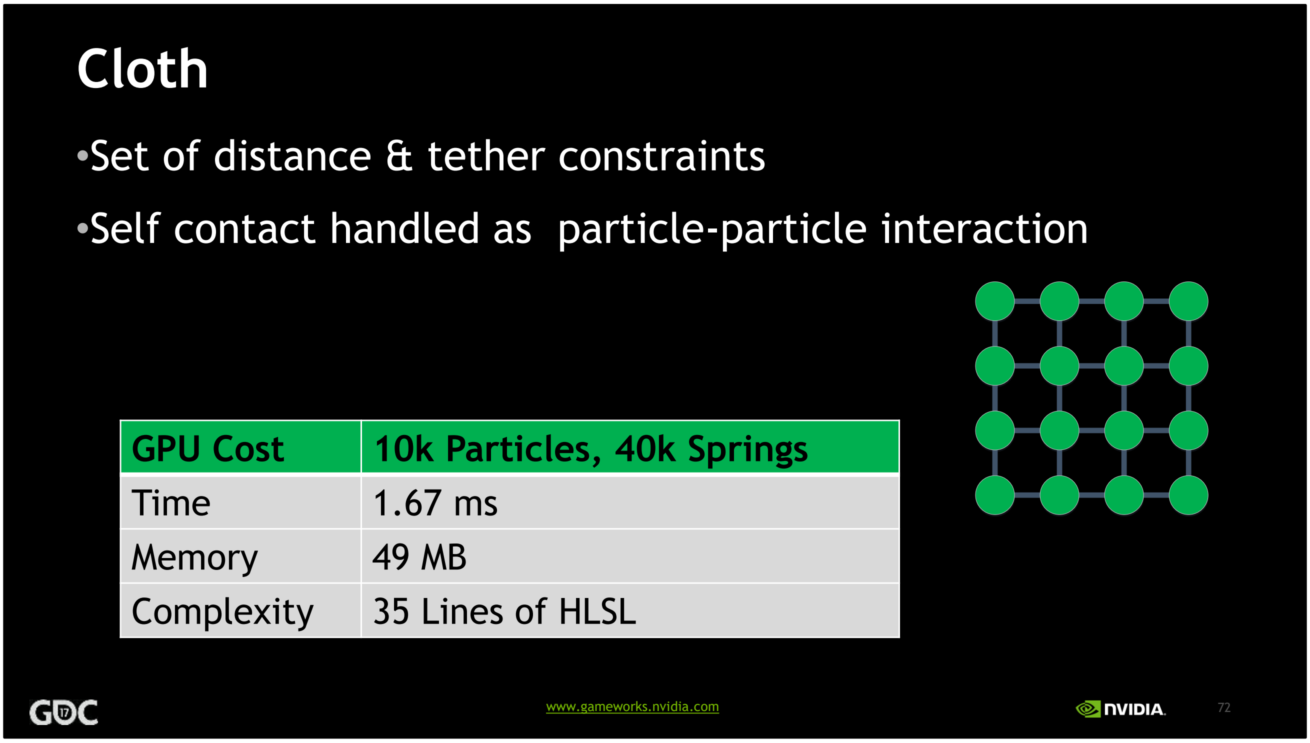


Jointed particles for cloth

www.gameworks.nvidia.com

71

Next we are going to look at cloth simulated using distance constraints



Cloth

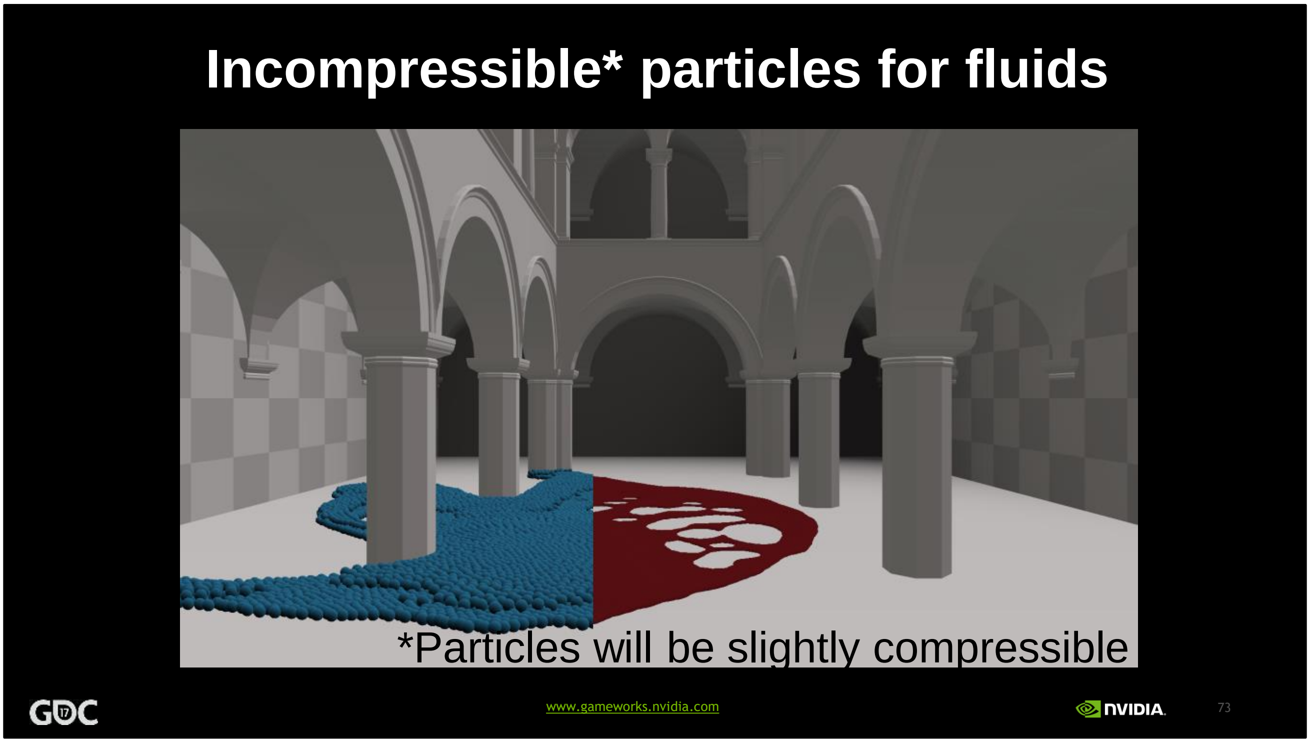
- Set of distance & tether constraints
- Self contact handled as particle-particle interaction

GPU Cost	10k Particles, 40k Springs
Time	1.67 ms
Memory	49 MB
Complexity	35 Lines of HLSL

www.gameworks.nvidia.com

Cloth is modeled by connecting particles to each other through distance constraints. Stretching can also be modeled using a stiffness parameter for each constraint.

Tether constraints are used as long range attachments to improve stability.



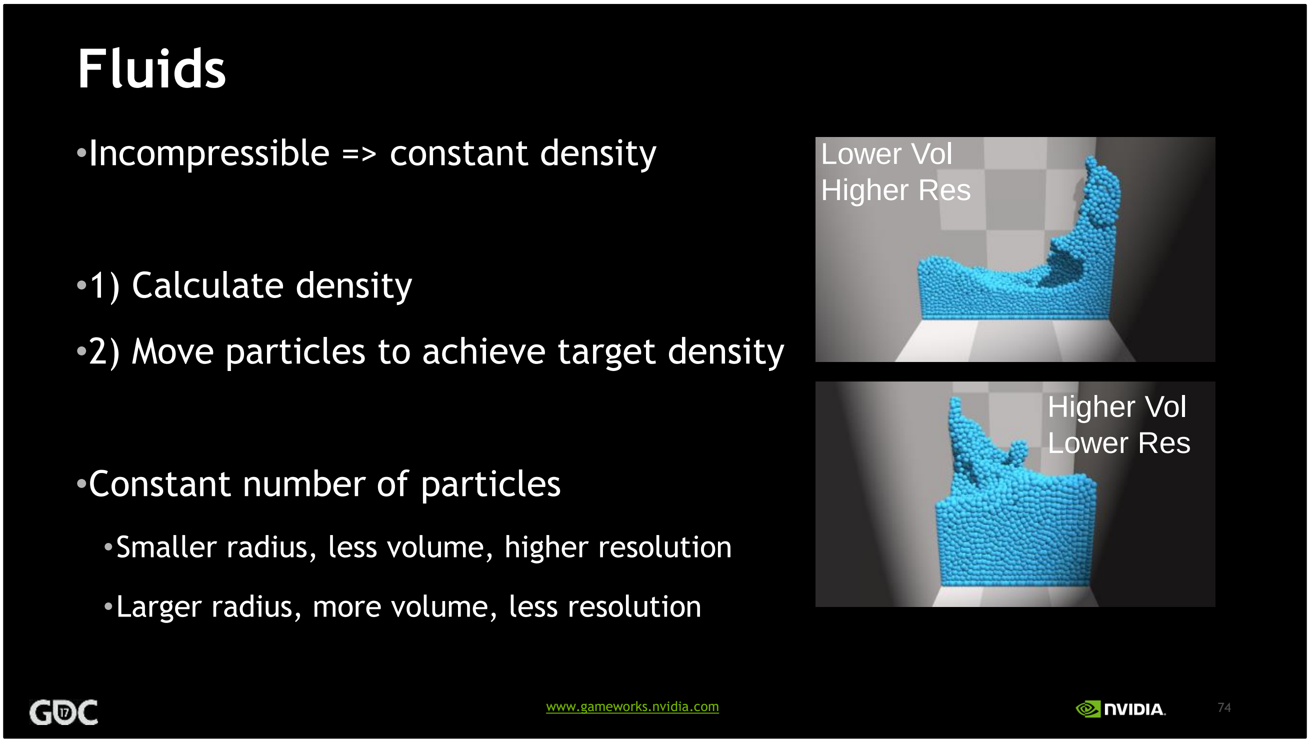
Incompressible* particles for fluids

*Particles will be slightly compressible

www.gameworks.nvidia.com

73

And finally we look at simulating and incompressible fluid using particles, the fluid will be approximately incompressible because we need to run in real time and have a fixed number of iterations.



Fluids

- Incompressible => constant density
- 1) Calculate density
- 2) Move particles to achieve target density
- Constant number of particles
  - Smaller radius, less volume, higher resolution
  - Larger radius, more volume, less resolution

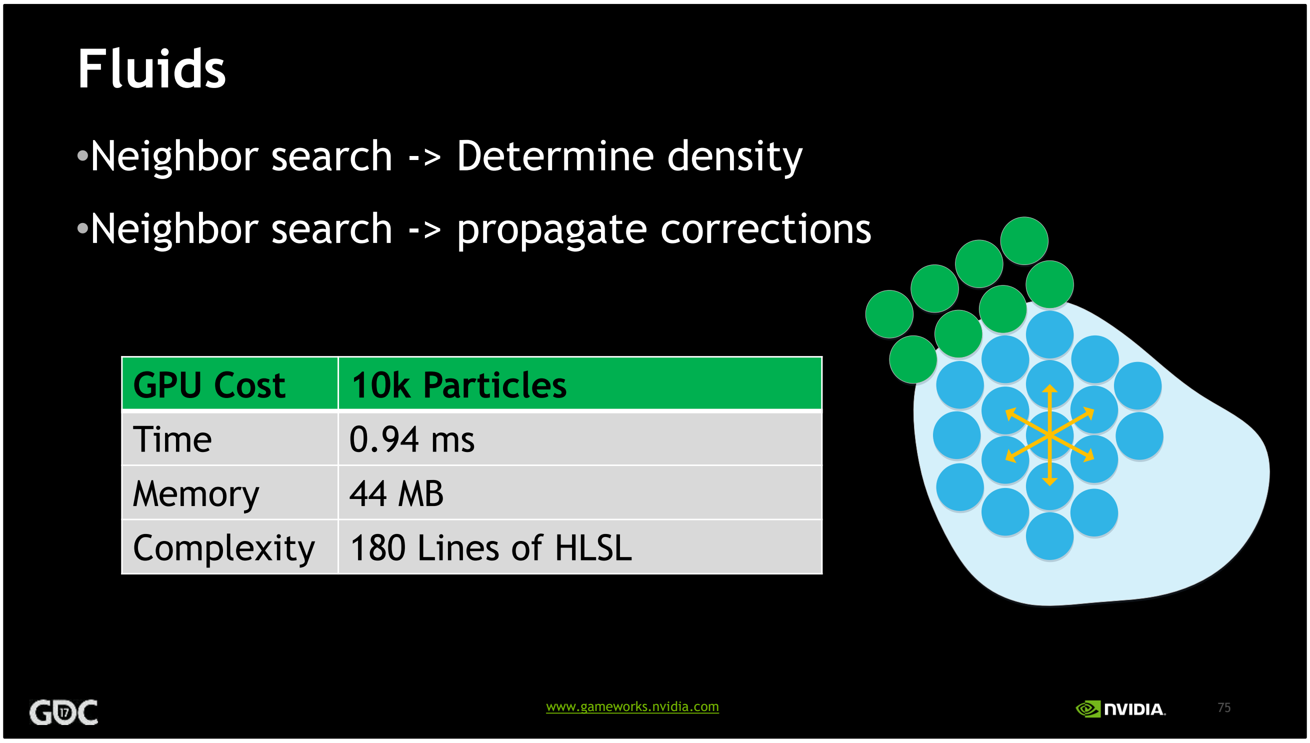
Lower Vol
Higher Res

Higher Vol
Lower Res

www.gameworks.nvidia.com

The constant density constraint that is enforced to get an incompressible fluid is solved in two steps. First calculating the particle density and then solving for the positon correction.

For a constant number of particles we can change the particle radius to get different effects. For a smaller particles size we have a smaller fluid volume but finer features are resolved better. For a larger fluid volume we can increase the particle radius but finer features cannot be resolved as nicely.

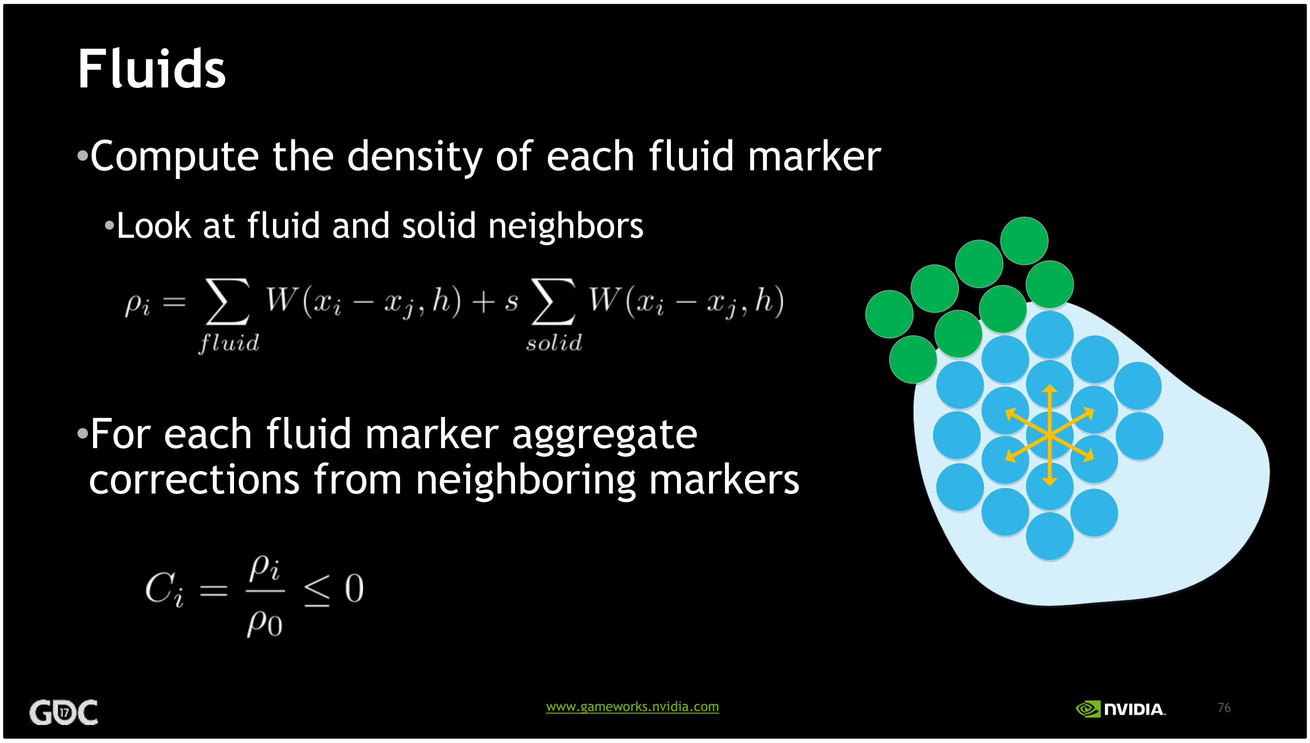


Fluids

- Neighbor search -> Determine density
- Neighbor search -> propagate corrections

GPU Cost	10k Particles
Time	0.94 ms
Memory	44 MB
Complexity	180 Lines of HLSL

The neighbor search is used twice, first to determine the density of each fluid particle and second to apply position corrections to achieve the correct density at the end of the step.



Fluids

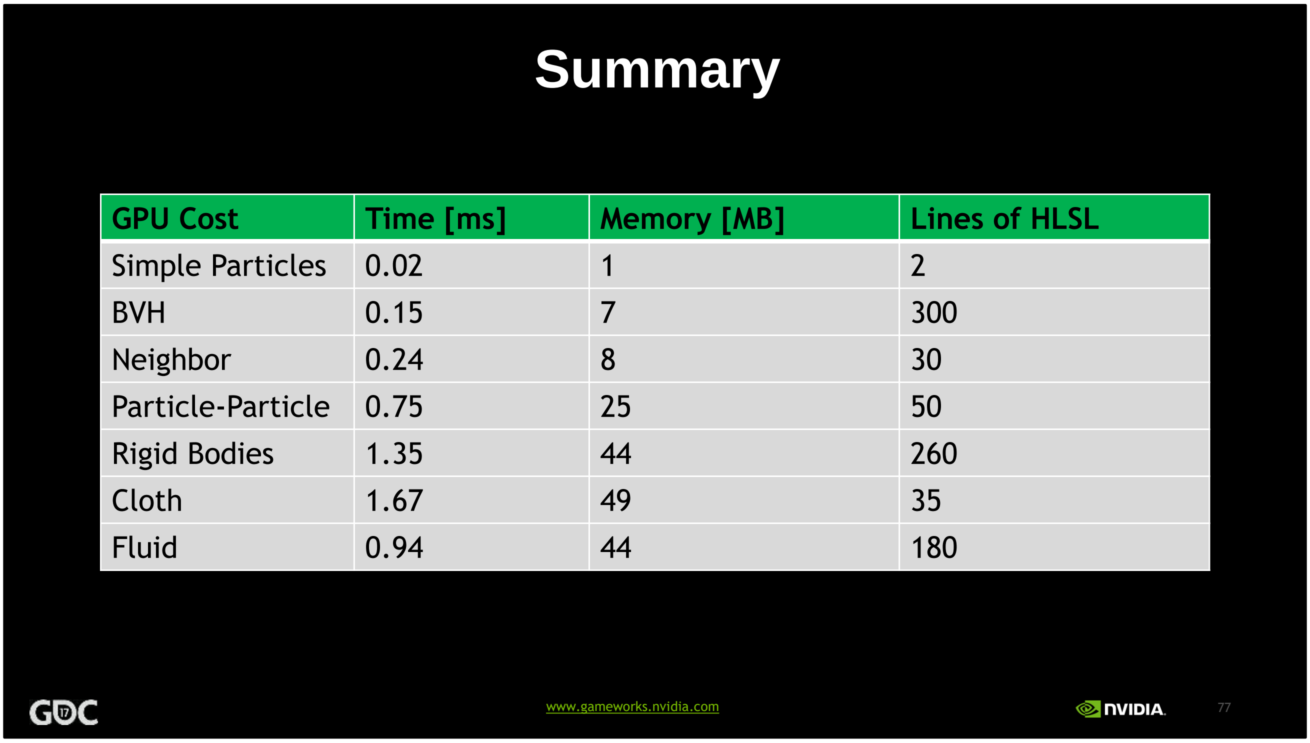
- Compute the density of each fluid marker
  - Look at fluid and solid neighbors

$$\rho_i = \sum_{fluid} W(x_i - x_j, h) + s \sum_{solid} W(x_i - x_j, h)$$

- For each fluid marker aggregate corrections from neighboring markers

$$C_i = \frac{\rho_i}{\rho_0} \leq 0$$

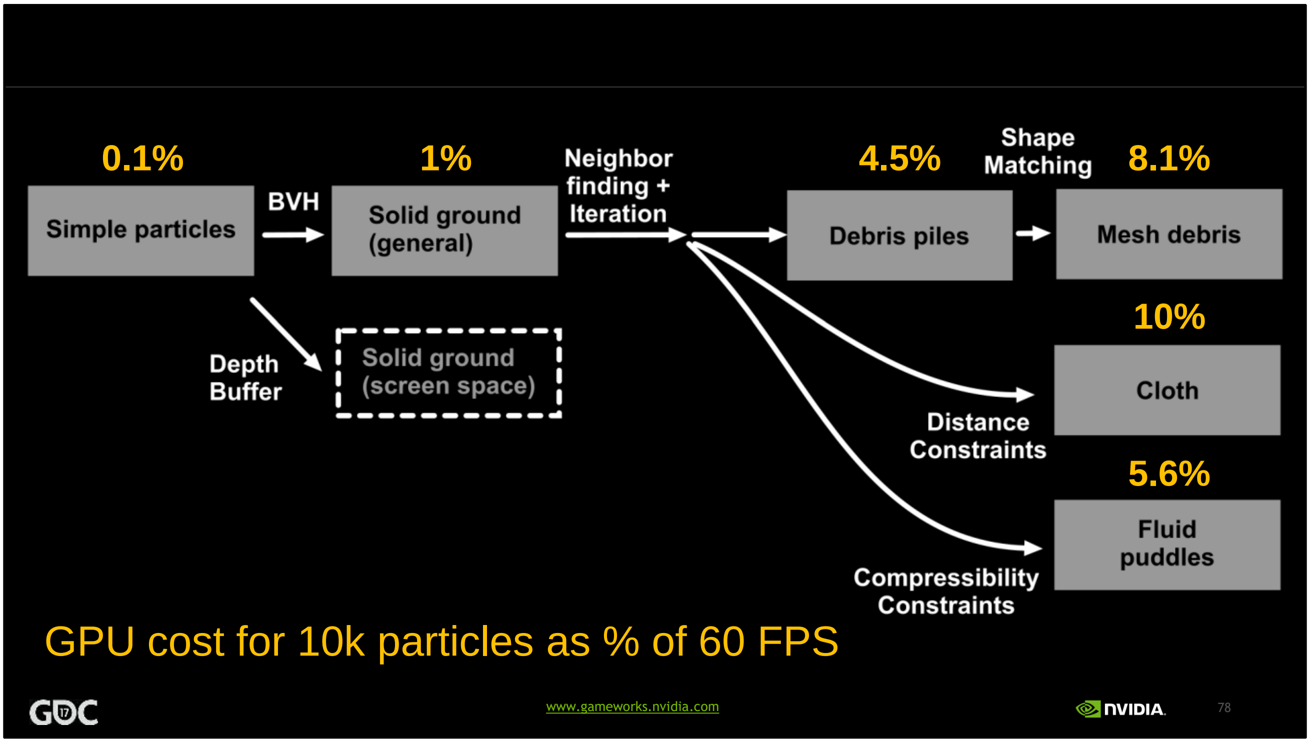
Because we can have density deficiencies at boundaries we take into account boundary particles in the particle density value. This also results in a more visually stable simulation at boundaries.



Summary

GPU Cost	Time [ms]	Memory [MB]	Lines of HLSL
Simple Particles	0.02	1	2
BVH	0.15	7	300
Neighbor	0.24	8	30
Particle-Particle	0.75	25	50
Rigid Bodies	1.35	44	260
Cloth	1.67	49	35
Fluid	0.94	44	180

This table summarizes the different algorithms, the associated computational cost, memory use and lines of HLSL.



0.1%

Simple particles

BVH

1%

Solid ground (general)

Depth Buffer

Solid ground (screen space)

Neighbor finding + Iteration

4.5%

Debris piles

Shape Matching

8.1%

Mesh debris

Distance Constraints

10%

Cloth

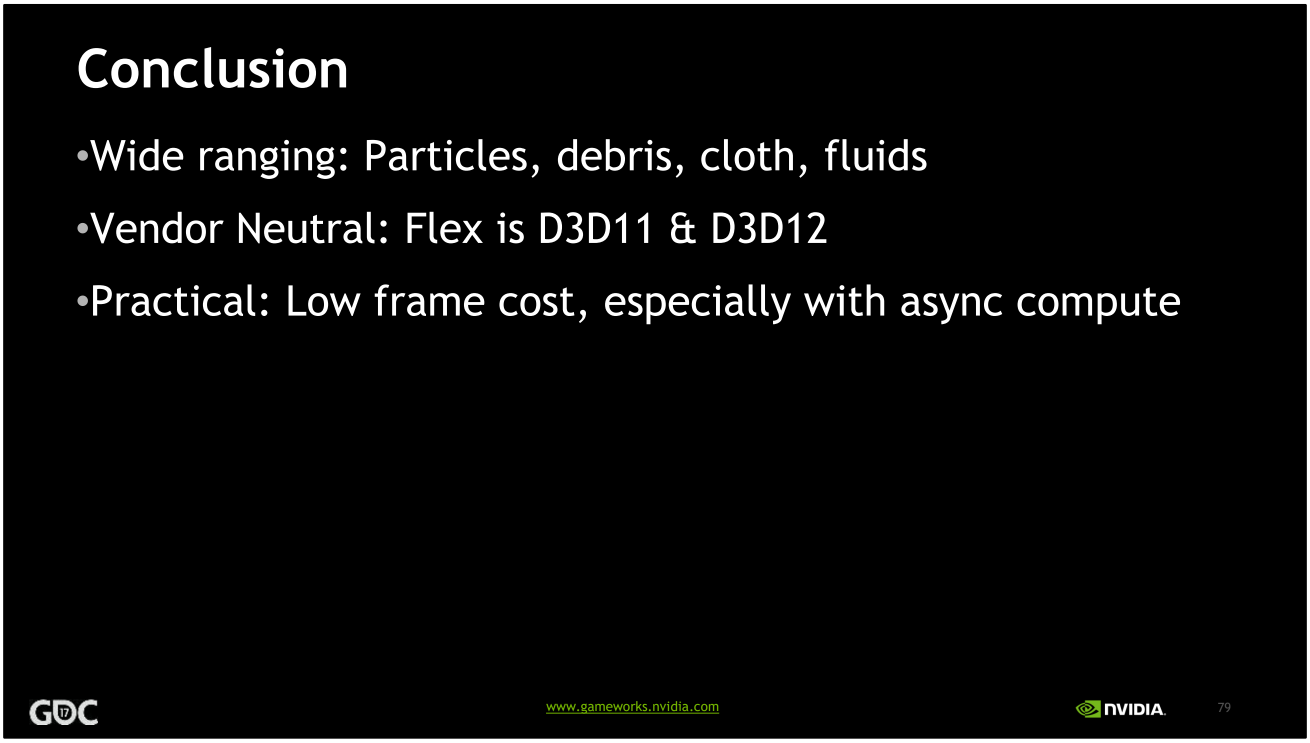
Compressibility Constraints

5.6%

Fluid puddles

GPU cost for 10k particles as % of 60 FPS

Here’s a recap of how you can go from a simple GPU particle system to rigid bodies, fluid and cloth.



Conclusion

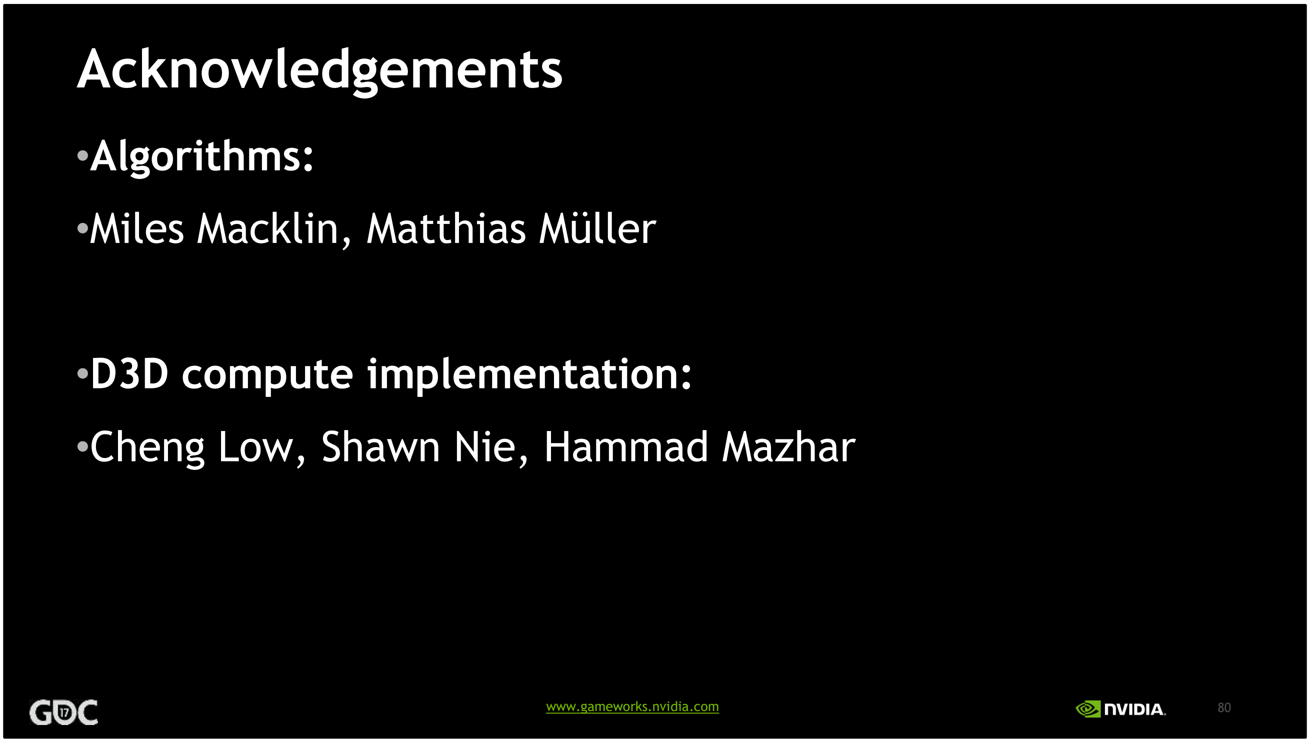
- Wide ranging: Particles, debris, cloth, fluids
- Vendor Neutral: Flex is D3D11 & D3D12
- Practical: Low frame cost, especially with async compute

What did you just see?

We released Flex on D3D11 and D3D12, so now all your users can enjoy it.

You saw how to fit particles into any GPU budget using async compute.

You saw how Flex is a natural extension of simple particles and the depth-buffer particle collision method you might already be using on the GPU.



Acknowledgements

- **Algorithms:**
- Miles Macklin, Matthias Müller

- **D3D compute implementation:**
- Cheng Low, Shawn Nie, Hammad Mazhar

Hammad and I didn’t invent Flex, we’d like to acknowledge those who did.

Questions?

Download SDK and Demos:
https://developer.nvidia.com/flex

rtonge@nvidia.com

hmazhar@nvidia.com

I can now take your questions.

Thankyou very much.

Position based dynamics , Müller, Matthias, et al. *Journal of Visual Communication and Image Representation* 18.2 (2007): 109-118.

Position-Based Simulation Methods in Computer Graphics, Jan. Bender, Matthias. Müller, Miles. Macklin, EUROGRAPHICS Tutorial Notes, 2015, Zürich, May 4-8

Unified Particle Physics for Real-Time Applications, Miles Macklin, Matthias Müller, Nuttapong Chentanez, Tae-Yong Kim, ACM Transactions on Graphics (SIGGRAPH 2014), 33(4)

A Survey on Position-Based Simulation Methods in Computer Graphics, Jan Bender, Matthias Müller, Miguel A. Otaduy, Matthias Teschner, Miles Macklin, in Computer Graphics Forum, 2014, pp. 1--25.

Position Based Fluids, Miles Macklin, Matthias Müller, ACM Transactions on Graphics (SIGGRAPH 2013), 32(4)

Thinking Parallel Part II: Tree Traversal on the GPU, Tero Karras, https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-ii-tree-traversal-gpu/