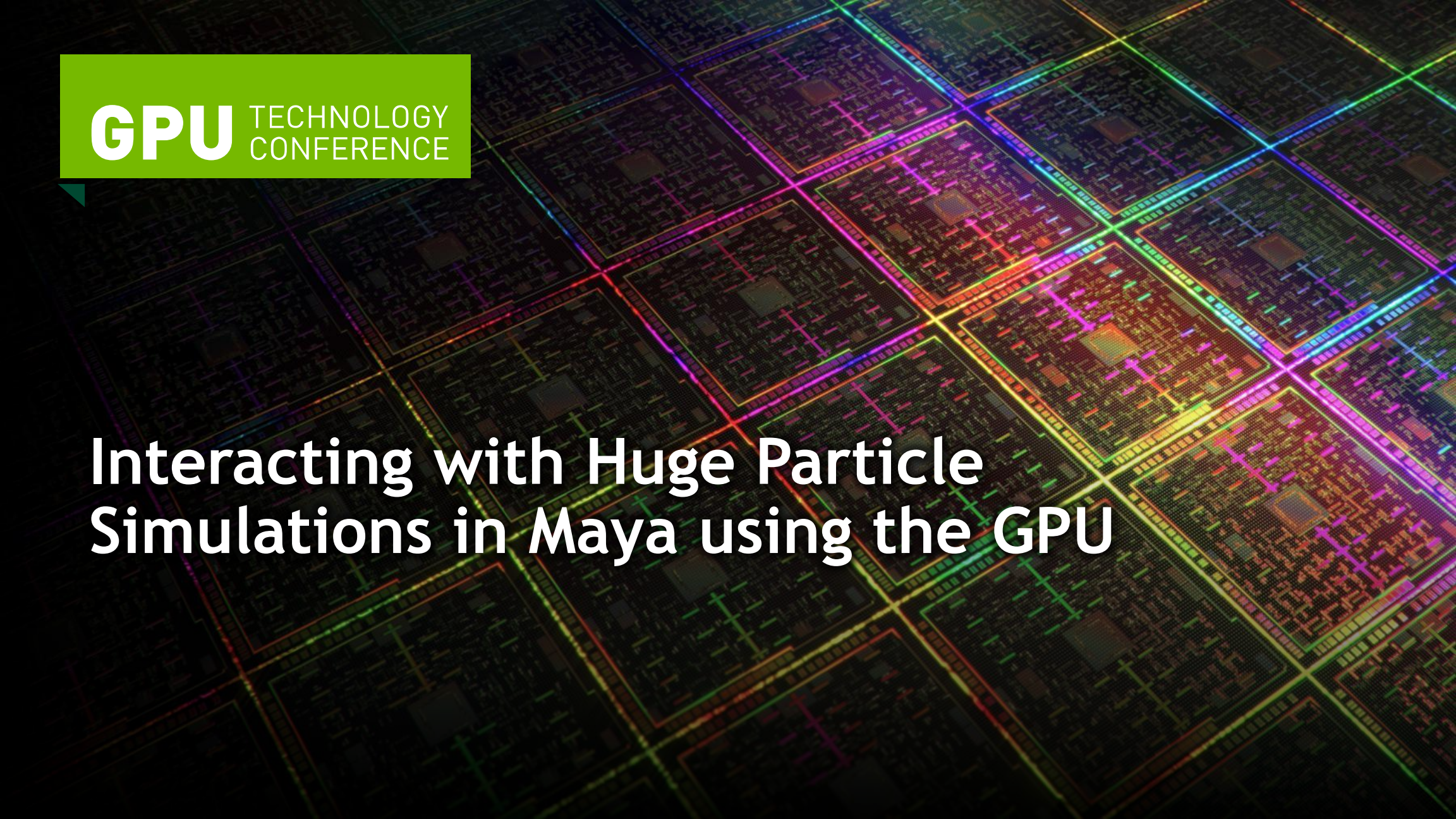
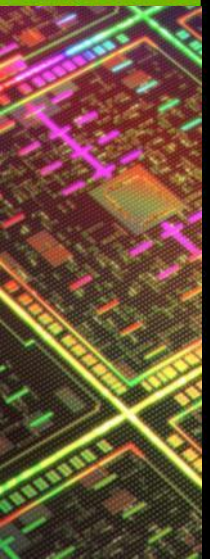


Interacting with Huge Particle Simulations in Maya using the GPU

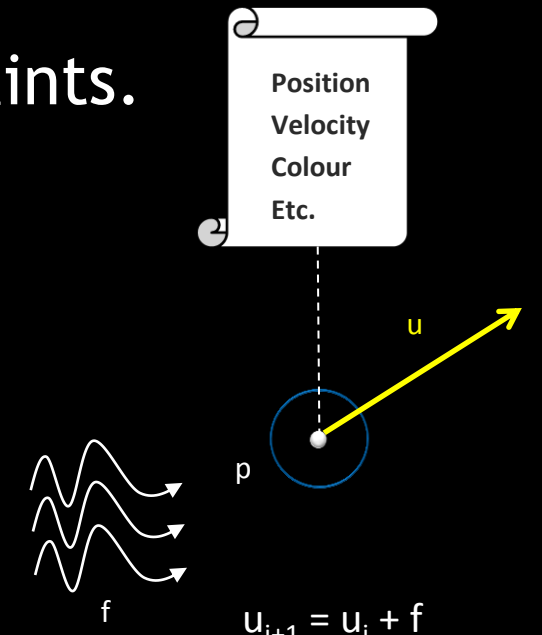


Introduction

- Particles (as used in VFX).
- Demo.
- GPU implementation.
  - Data organization.
  - Emission.
  - Environmental forces & collisions.
- Designing for a professional workflow.
- Fun with fluids.
- Demo & questions.

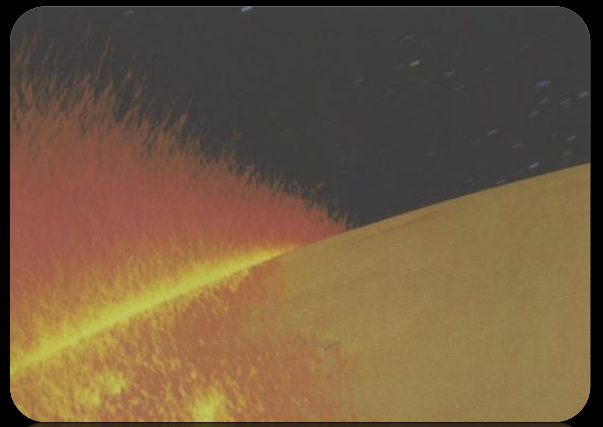
Introduction to Particles

- Discrete points with attributes, in space.
- Movement is defined by forces and constraints.
- They have limited state.
  - e.g. Force modifies velocity, and velocity modifies position.
  - Unconstrained by a grid.
    - Computation and storage are only used where needed.
- Already used extensively in VFX…



Particles in Film VFX

FX



1982 - First CG particle FX in StarTrek 2.

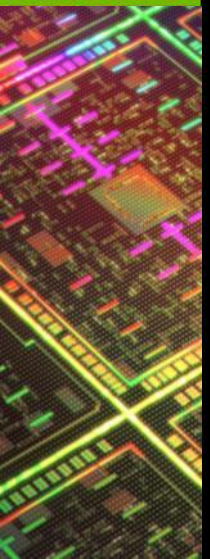
Rendering

2008 - Disney use GPUs to efficiently render particles as sprites.

Physics

2009 - ILM use GPU particles for fast rigid-body dynamics in Transformers 2.

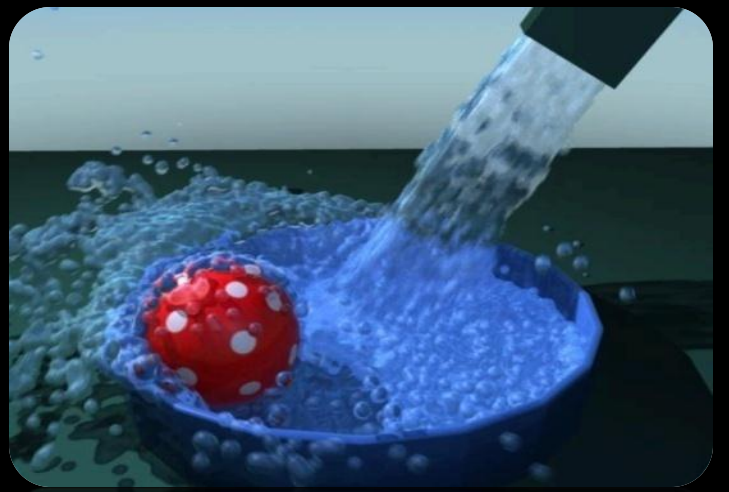
Images courtesy of:
Star Trek 2 – The Wrath of Khan
Bolt, Walt Disney Pictures
Transformers 2, Paramount Pictures



Particle Data Structures

- System-wide attributes
  - Attributes shared by all particles.
  - Typically constant.
  - e.g. gravity, deltaTime, mass, radius, etc.
- Per-particle attributes
  - Specified by the solver-definition / user-defined.
  - Structure of arrays.
  - e.g. position, velocity, color, temperature, etc.

Autodesk Maya - “nParticles”

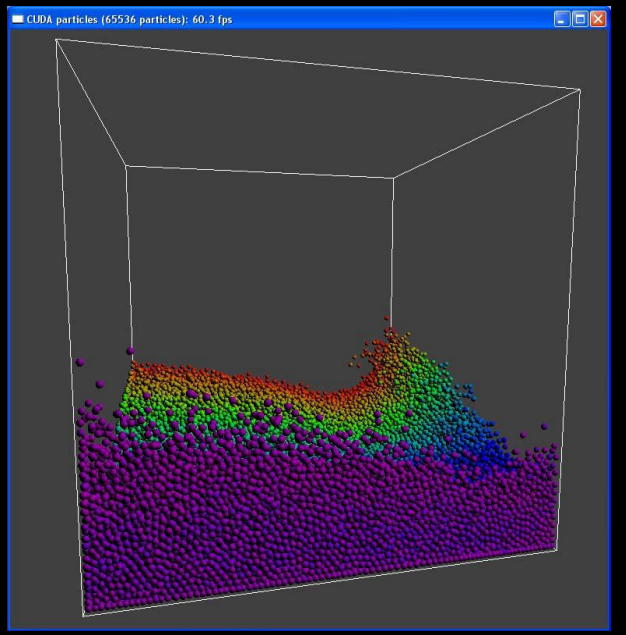


- Emitters.
  - Omni, Volume, Surface, etc.
  - Surfaces can emit particle attributes.
- Force-fields.
  - Uniform, Radial, Vortex, Air, Drag, etc.
- Collision with arbitrary geometry.
- Expression language.
- Allows integration of different kinds of materials (e.g. cloth, fluid) by using a singular physics representation.

Motivation for Particles on the GPU

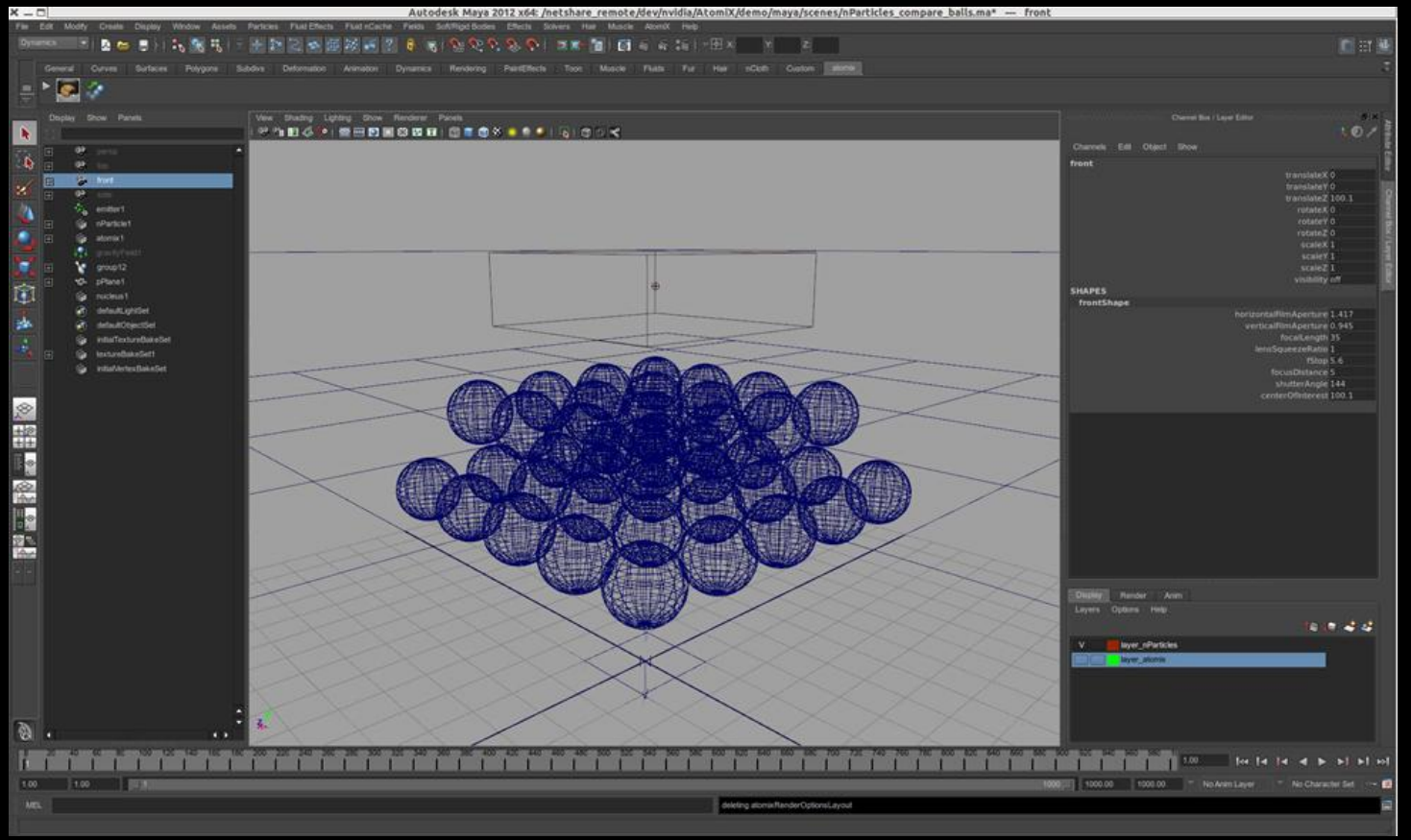
- Particles are simple and elegant.
- Small memory footprint.
- Easy to implement.
- Highly scalable.

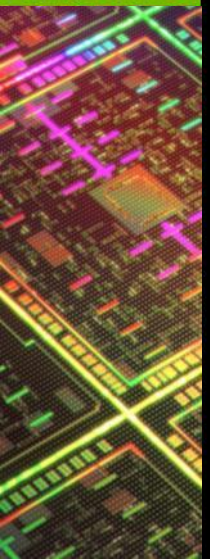
= GPU friendly!



- See S.Green's "particles" example in CUDA SDK.

Demo - nParticles vs. GPU Particles

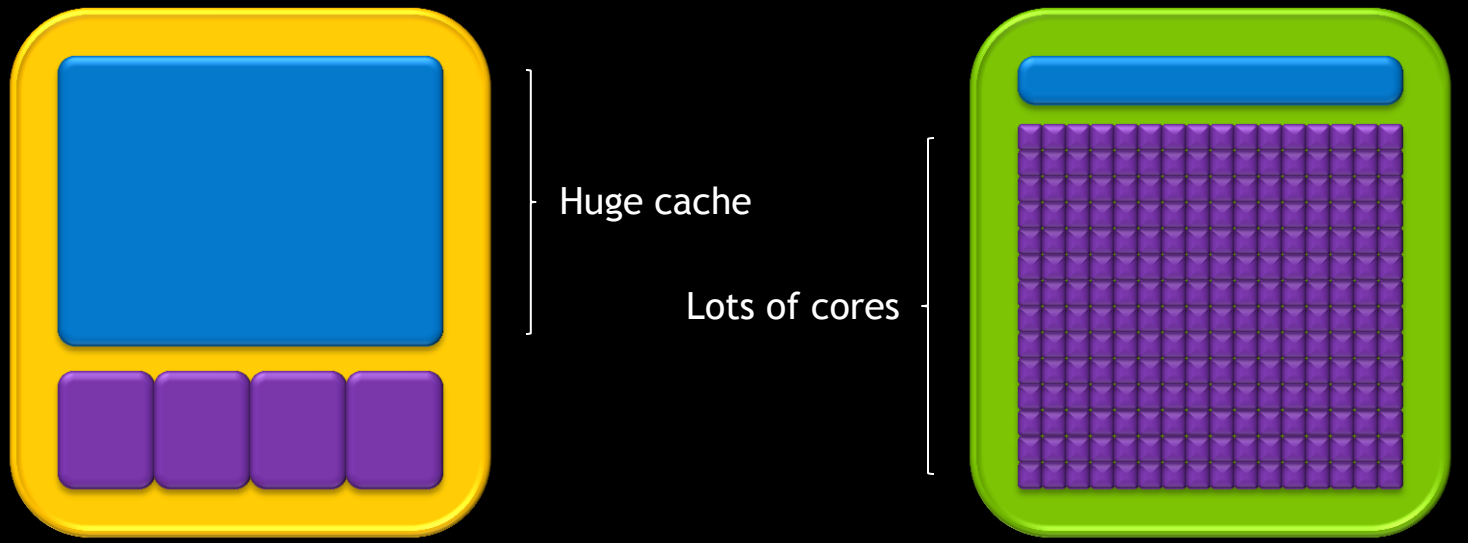




Implementation Goals

- Integrate with existing Maya workflow.
- Be more interactive than current Maya tools.
  - We don’t want to slow down the Maya user-experience.
- Solution for *any* kind of particle problem.
  - rigid-bodies, DEM, fluid, hair, crowds… (smoke, fire)

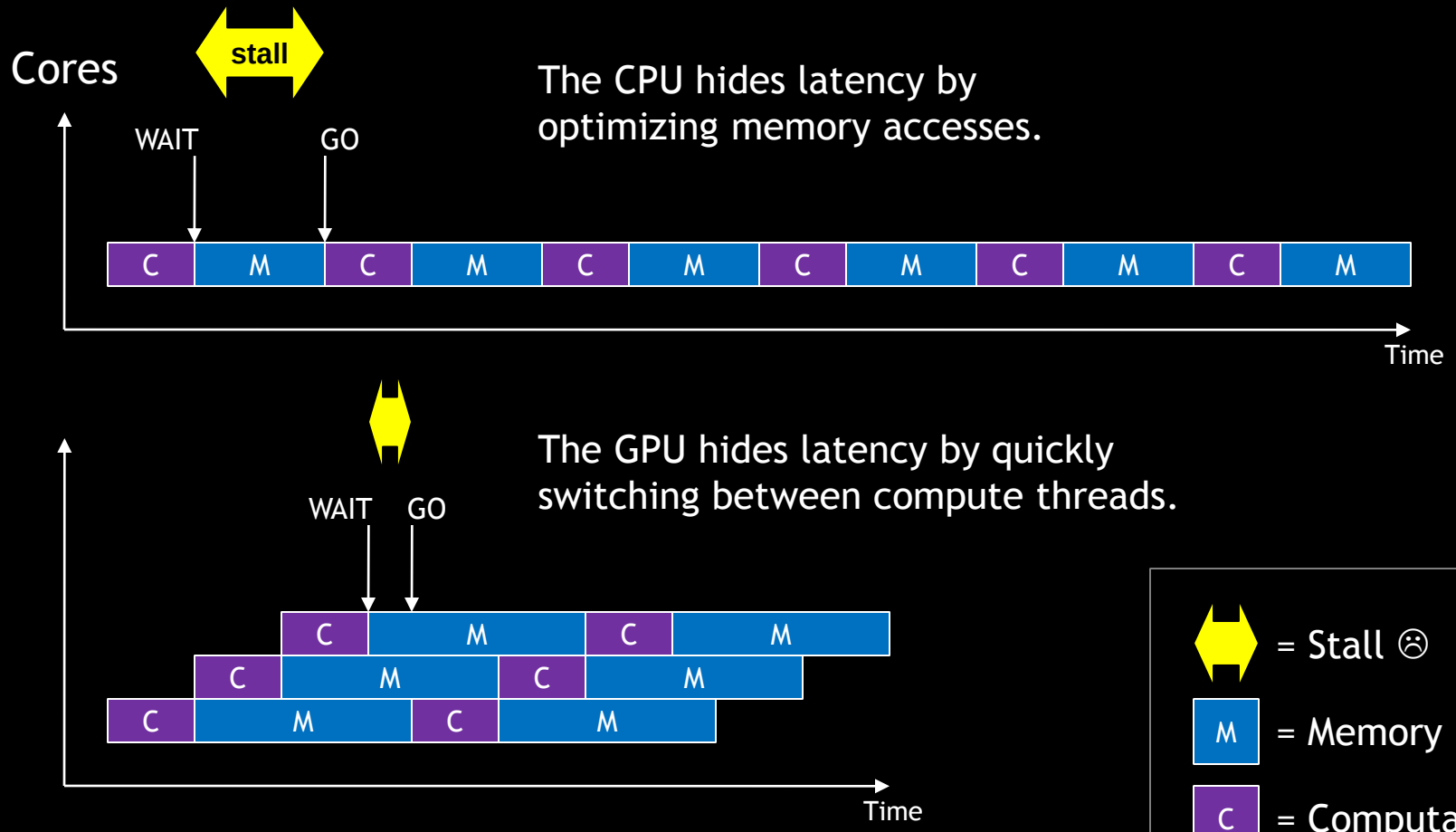
The GPU in 1½ slides



multi-core CPU
(4 cores)

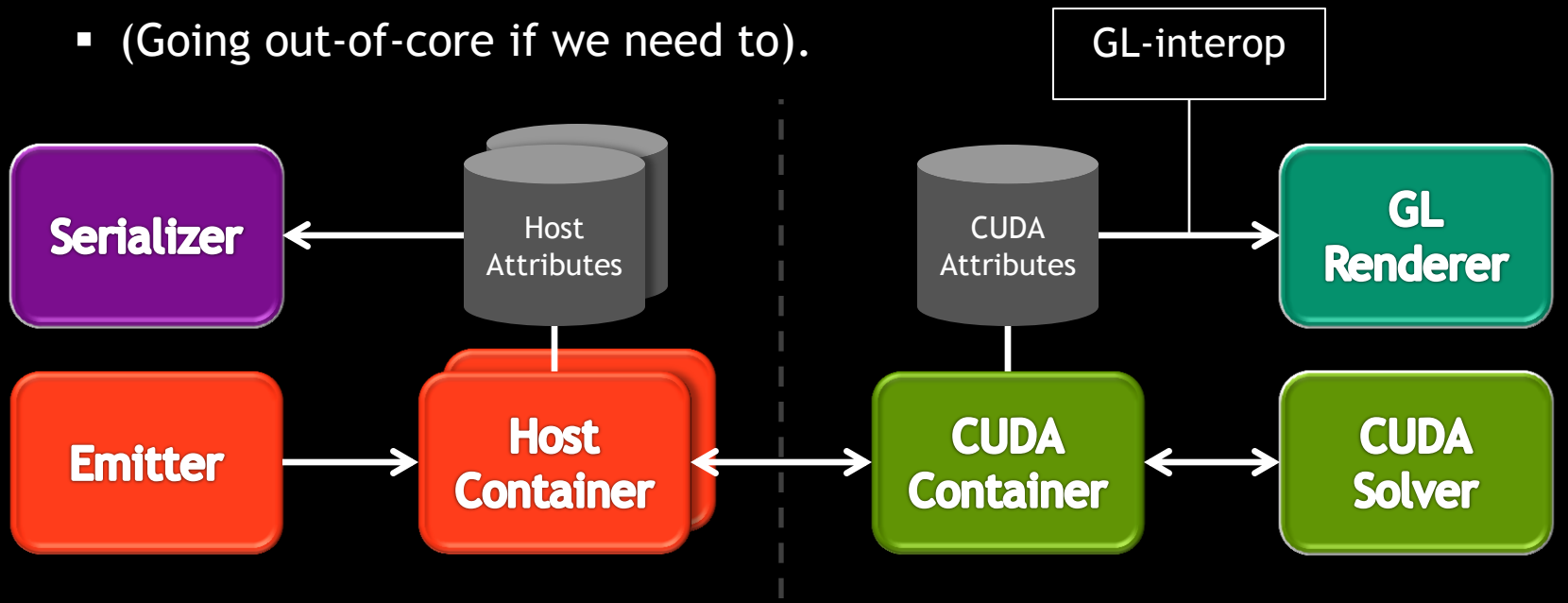
many-core GPU
(448 cores)

Latency-hiding vs. Throughput

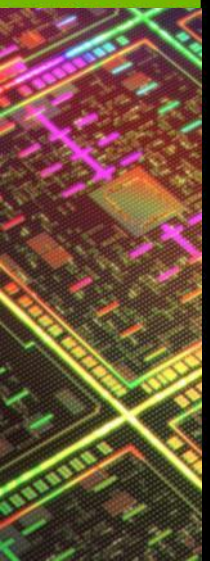


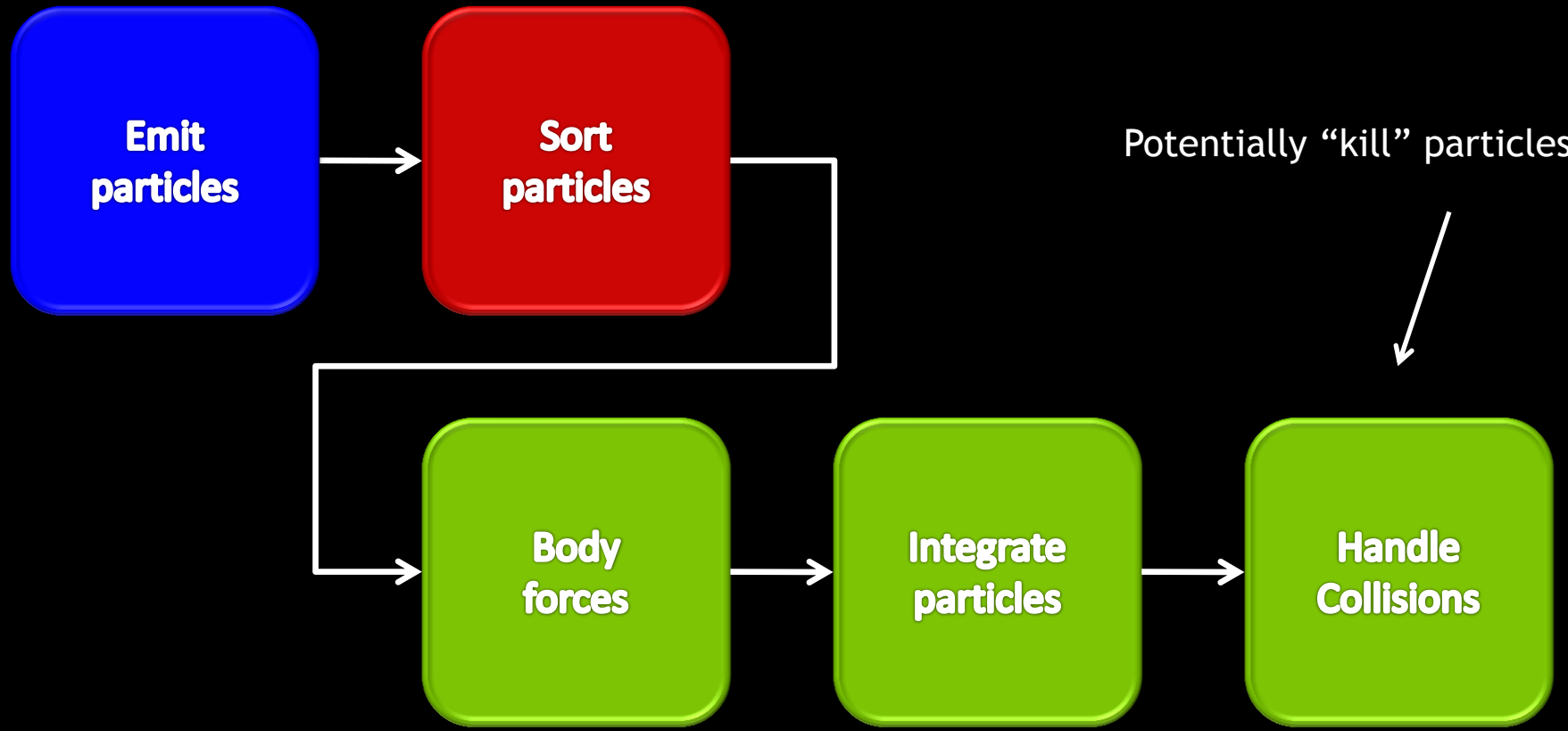
System Architecture

- Solver-definition:
  - Defines attributes & attributeArrays (per-particle attributes)
  - Informs emitters what they *must* provide.
- Containers are mirrored on device
  - (Going out-of-core if we need to).

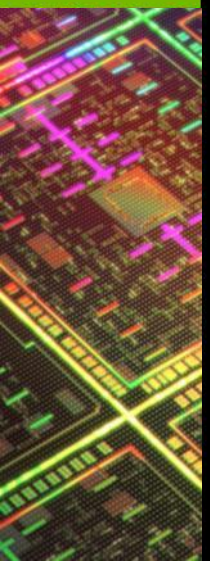


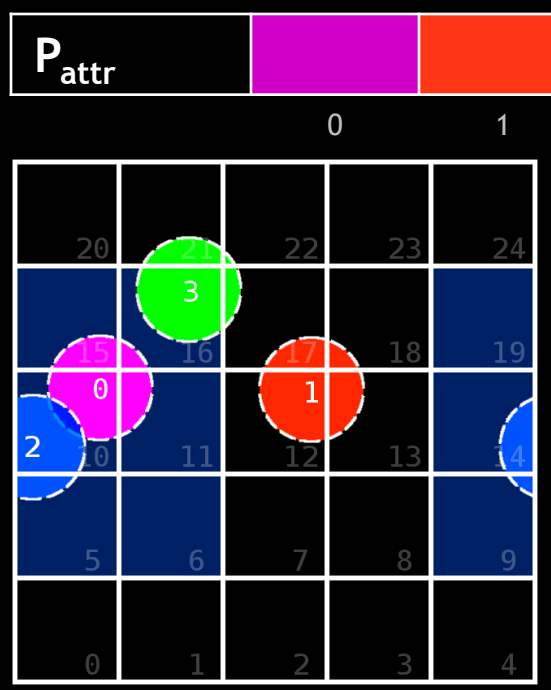
Typical Particle Life-cycle on the Device





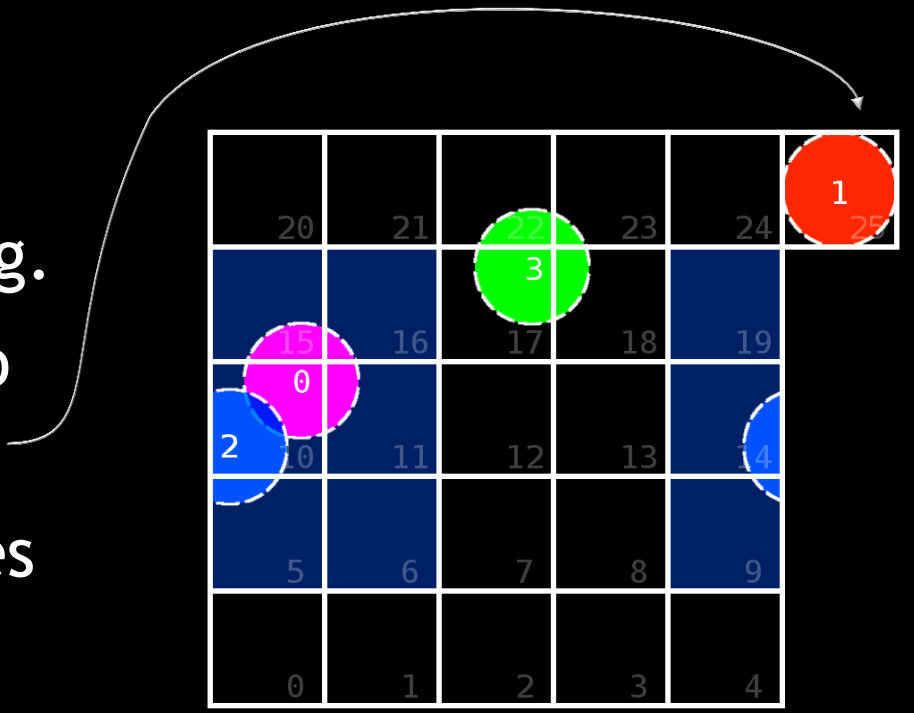
Particle Sort into Uniform Grid

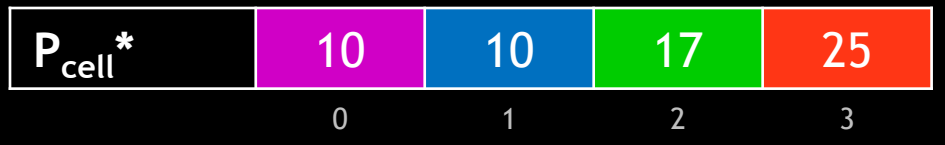




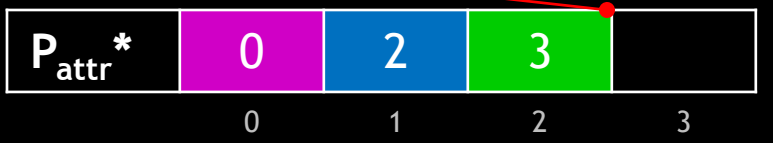
Particle Death

- If particle expires, set death flag.
- Cell assignment then puts it into *deadCell* (numCells+1 = 25)
- Sorting will put all dead particles in a contiguous block.



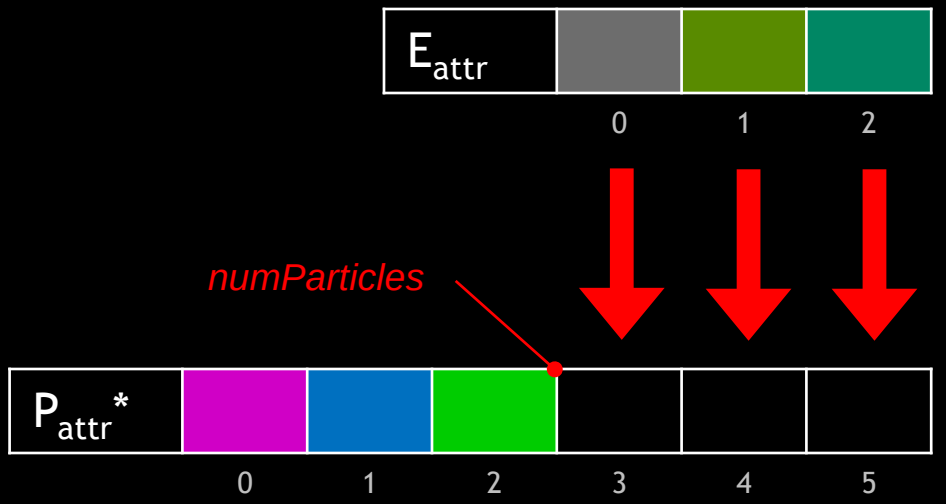


numParticles



Particle Emission

- Host Container may have “new” particles.
- Upload emitted data using numParticles as offset.
- numParticles += numEmitted



- We use fast host-to-device transfer for emission data.
  - i.e. page-locked & write-combined host memory.

Maya's Body Forces

- Typically a function of position.

$$\mathbf{F}_{uniform} = \alpha k \mathbf{d}$$

$$\mathbf{F}_{radial} = \alpha k \frac{\mathbf{r}_i}{|\mathbf{r}_i|}$$

$$\mathbf{F}_{vortex} = \alpha k \left(\frac{\mathbf{r}_i}{|\mathbf{r}_i|} \times \mathbf{d} \right)$$

$$\mathbf{d} = \text{axis}$$

$$k = \text{magnitude}$$

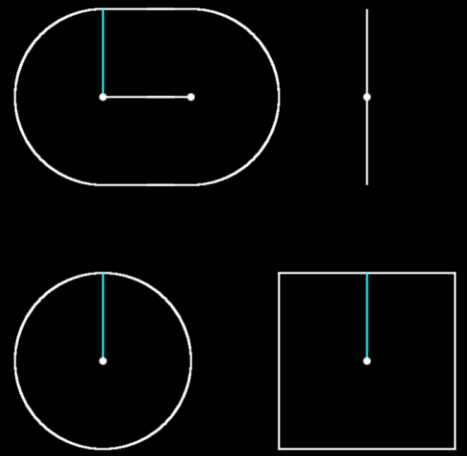
$$\mathbf{r}_i = \mathbf{x}_i - \mathbf{x}_{origin}$$

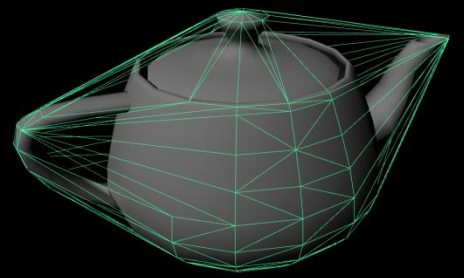
$$\alpha = \frac{1}{\exp\left(\beta |\mathbf{r}_i|\right)}$$

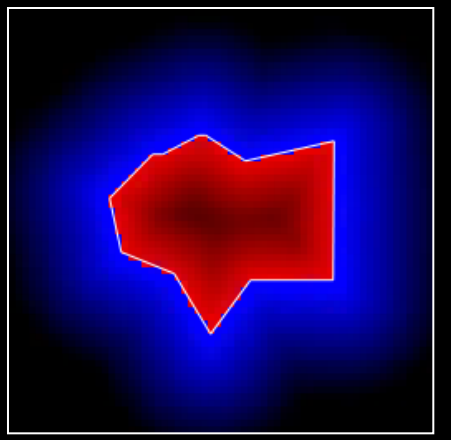
where Beta is the attenuation factor

Primitives

- Multi-purpose geometric primitives.
- Support affine transforms.
- 3 types:
  - Implicit
    - Simple for fast math.
  - Convex-hull
    - Only convex objects.
    - HACD algorithm (K. Mamou et al.)
  - Signed distance fields
    - Handles any topology but with memory cost.

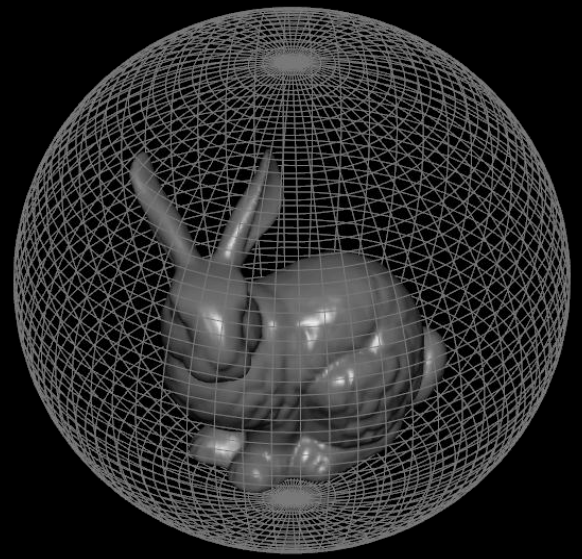


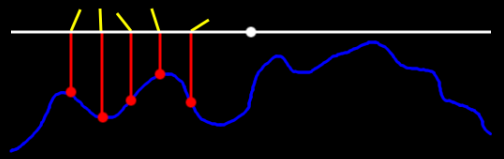


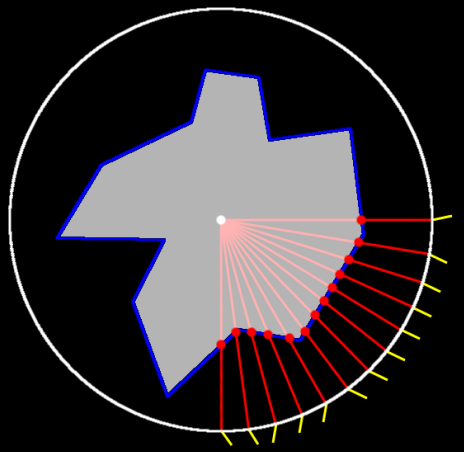


Displacement maps

- Allows more complex objects.
- Effectively a 2-level BVH.
- xNormal for map creation.



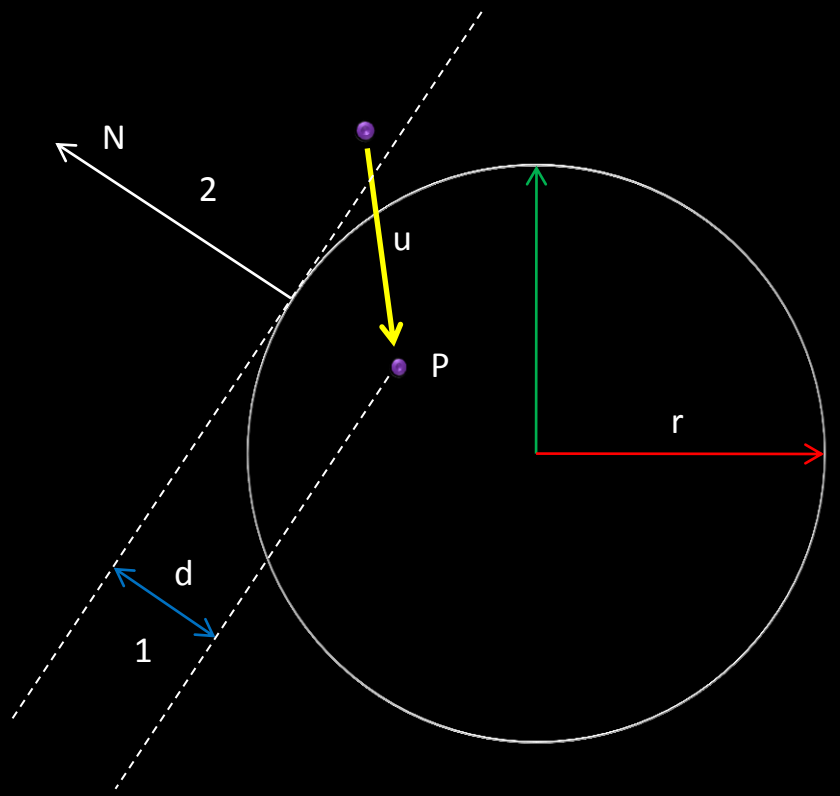


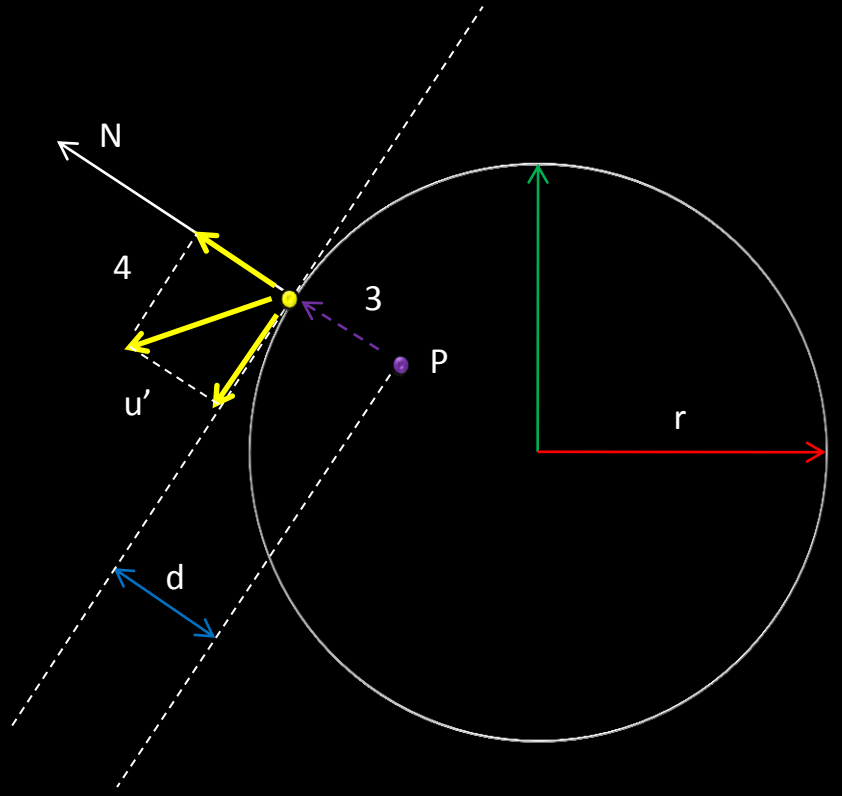


Normals

Displacement

Collision response

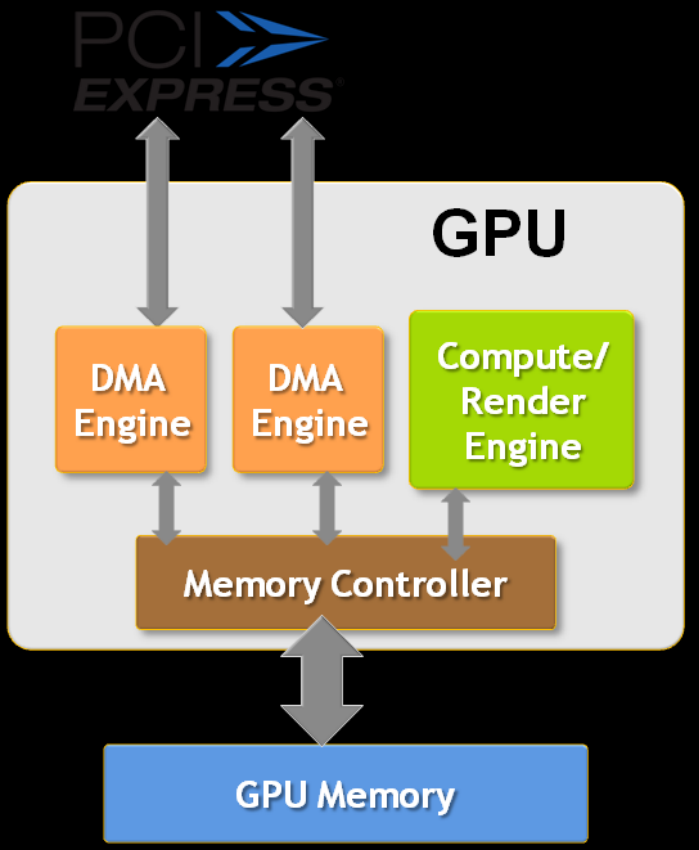




1. Transform particle (P) into unit-space and calculate penetration (d)
2. Determine normal (N)
3. Move particle out of collider (along normal)
4. Update velocity based on collider's friction and restitution coefficients

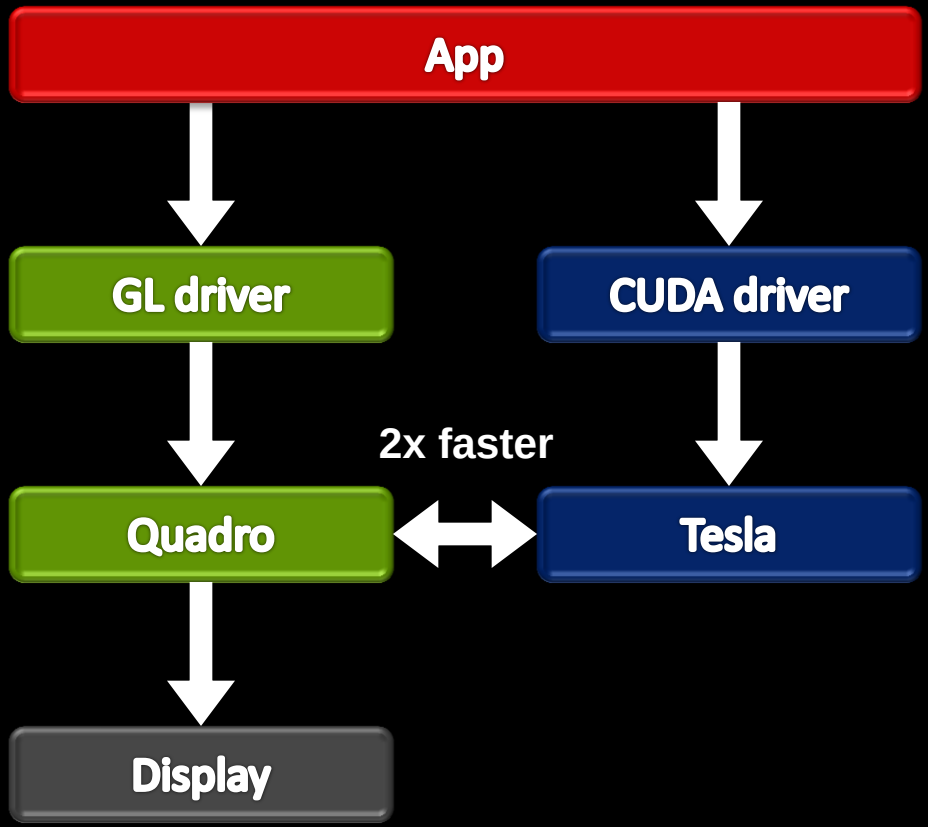
Getting data back to Maya

- We want to use the particles' data inside Maya.
- Modern NVIDIA GPUs can overlap compute with copy.
  - Quadro/Tesla have 2 copy engines.
- We can remove the cost of the data transfer if we allow some latency…



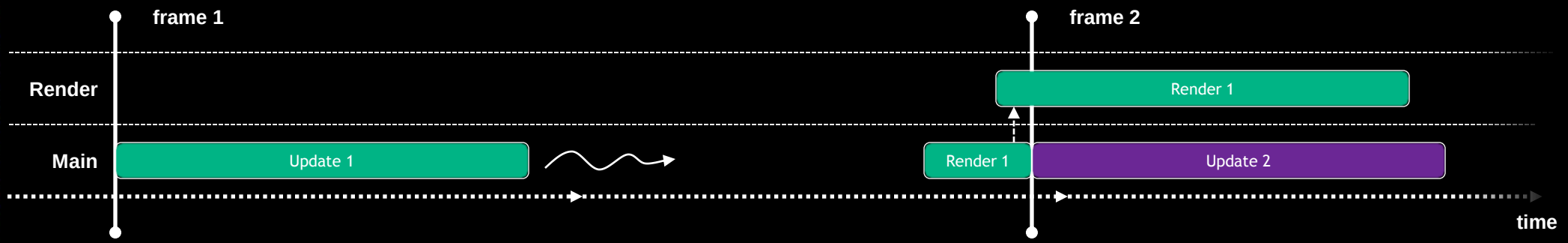
NVIDIA Maximus (Quadro & Tesla)

- 6GB & 448 cores each.
- DCC apps like Maya already use many GPU resources.
- Option of multi-GPU scaling.
- GL-interop
  - No CPU involved.
  - Tesla-CUDA to Quadro-GL.
  - Driver always looks to optimize.



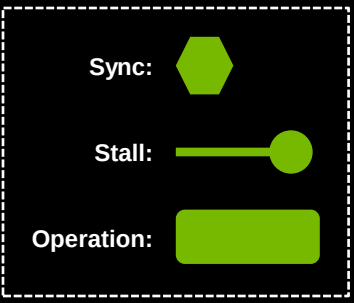
The Host Application - (Autodesk Maya)

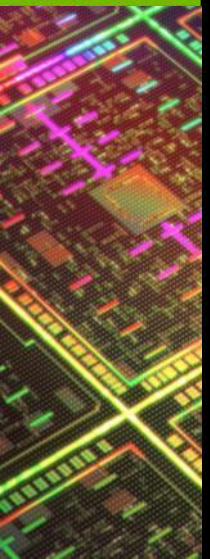
- Maya plug-in callbacks:
  - Update (called during data computation)
    - Upload & Simulate the particles.
    - Download the particles for export.
  - Render (called during Maya's scene-graph traversal)
    - Render particles.

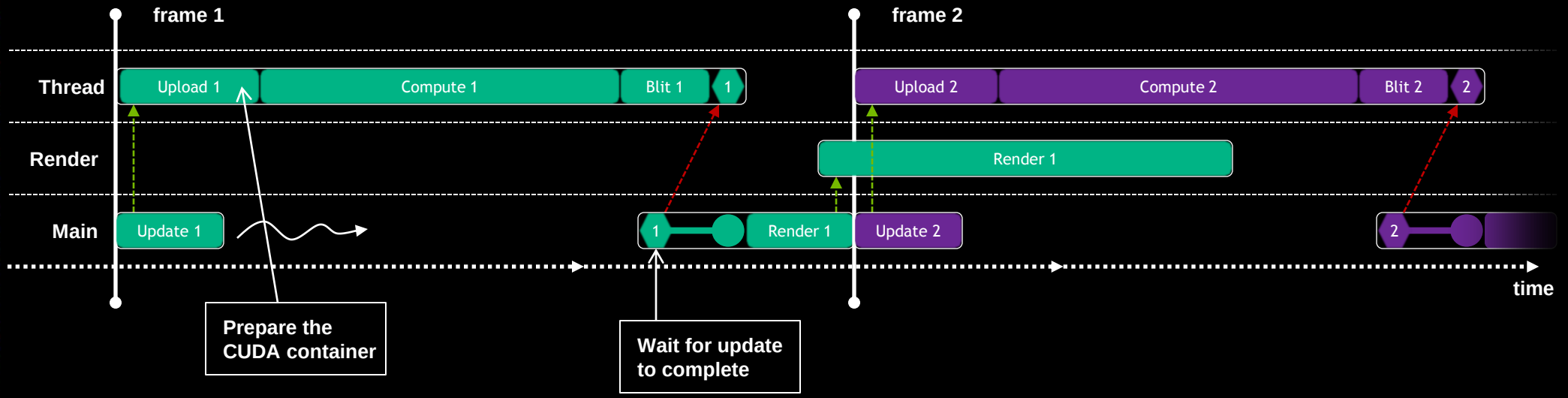


Example 1: Synchronous CUDA thread

- Create “GPU worker” thread for CUDA context.
  - Protection for both app & plug-in.
- Blit computed buffers to app’s OpenGL context
  - Use the CUDA GL-interop

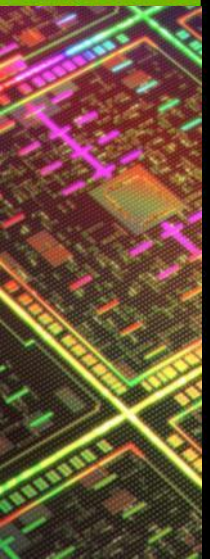






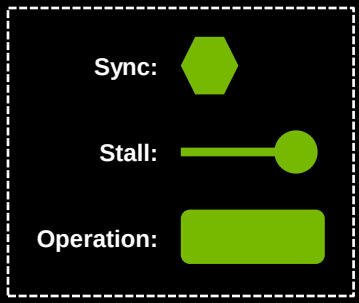
Example 1: Synchronous CUDA thread

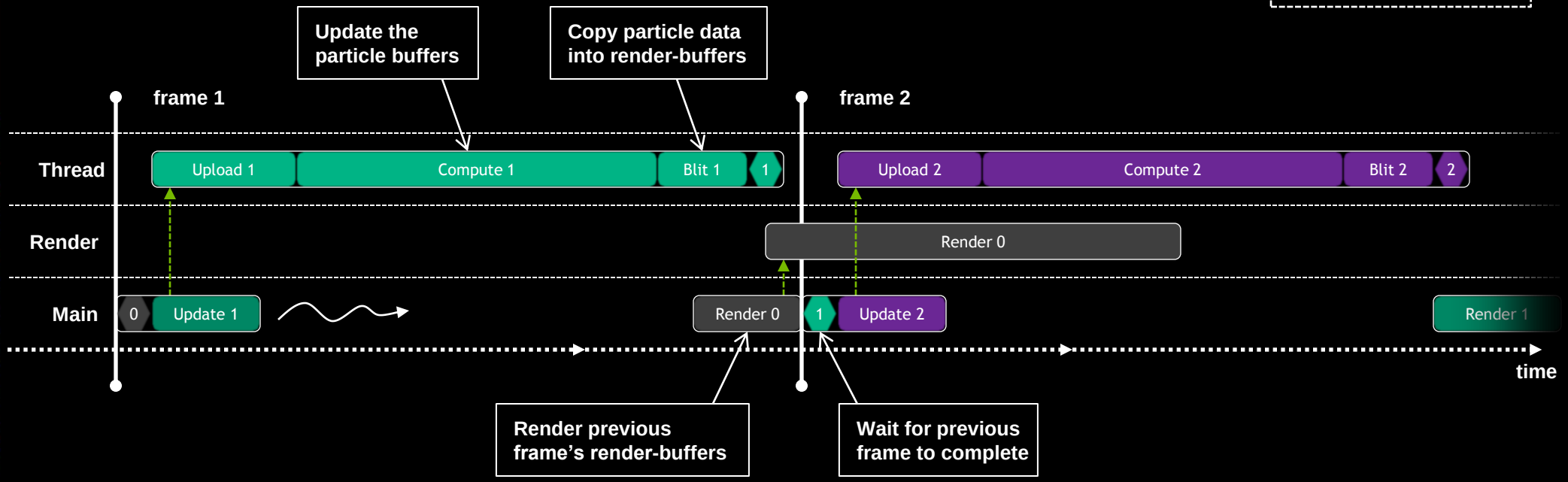
- Advantages:
  - Simple to implement.
  - Multi-threading protects our plug-in / host app.
  - Uses GL-interop for efficient data movement to the Quadro GPU
- Disadvantages:
  - We are relying on Maya calling our update at the start, and our render at the end!

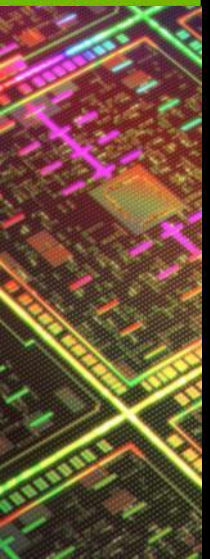


Example 2: Double-buffered CUDA thread

- Use latency to overlap update with host app.





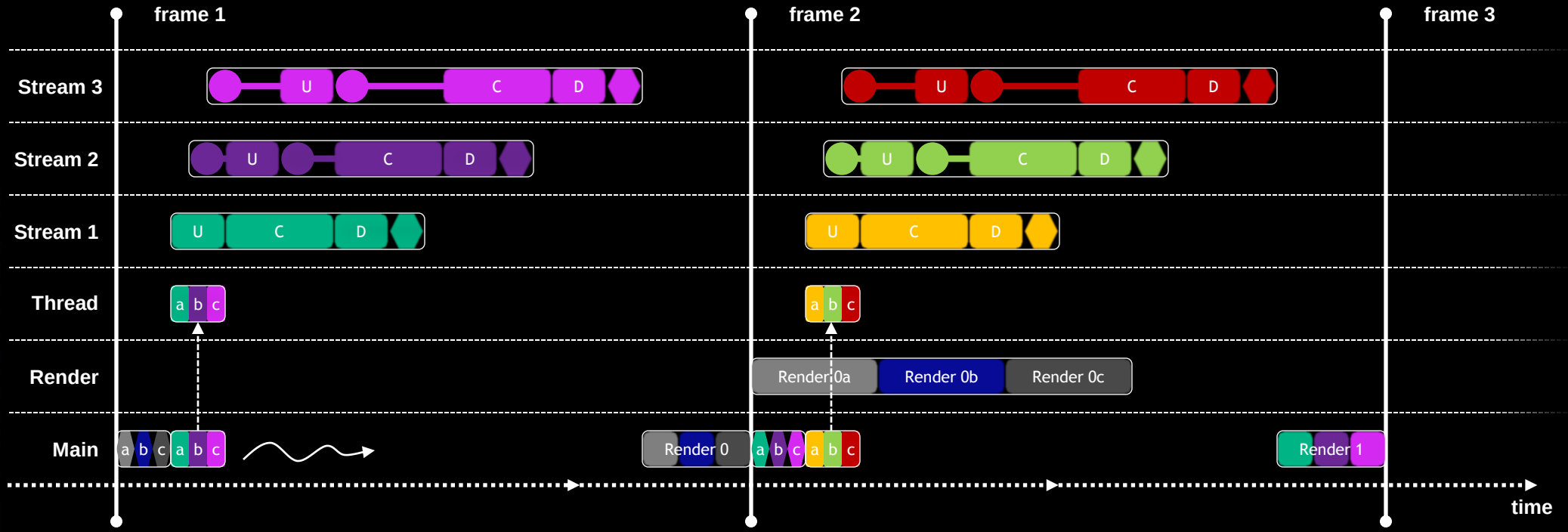


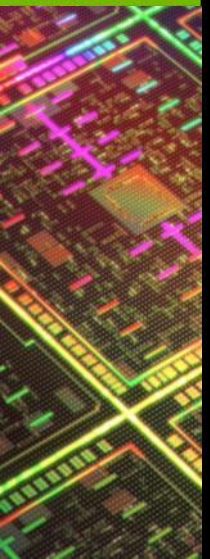
Double-buffered CUDA thread: Conclusions

- Advantages
  - Simple to implement.
  - Multi-threaded.
    - protects our plug-in / app.
    - Allows overlap of our computation and app.
- Disadvantages
  - One frame of latency.
  - No overlapping of compute with upload/download.
  - No frame-accurate access to particle data.

Example 3: Asynchronous CUDA streams

- One CUDA stream per batch



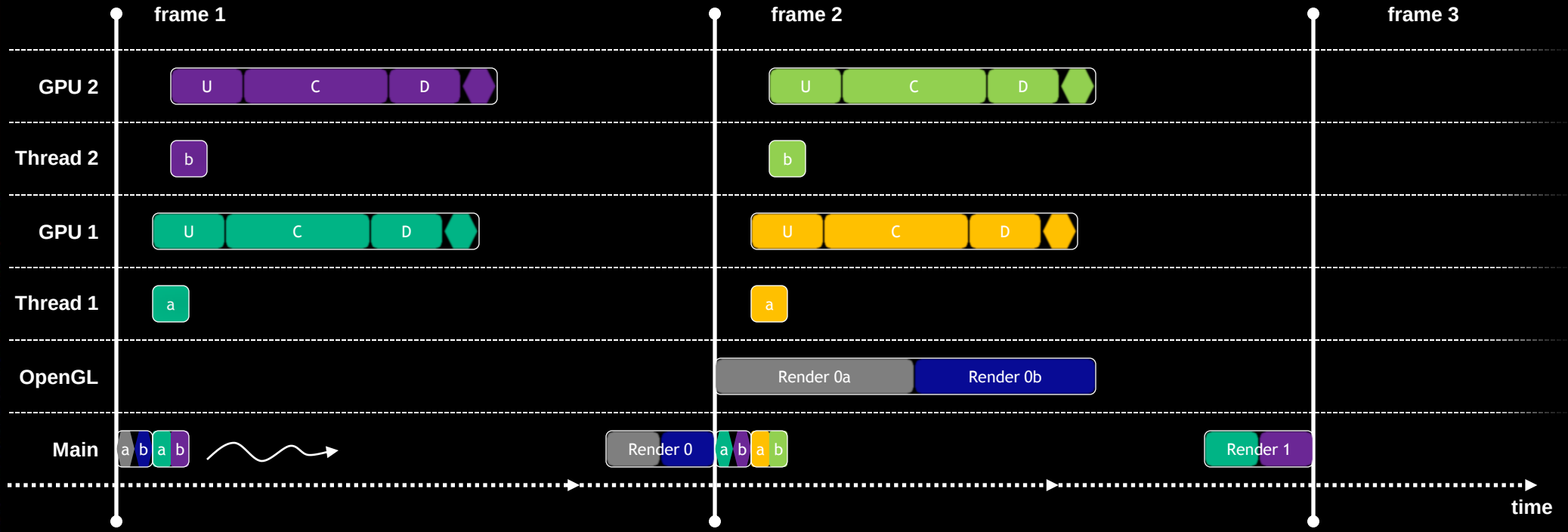


Asynchronous CUDA streams: Conclusions

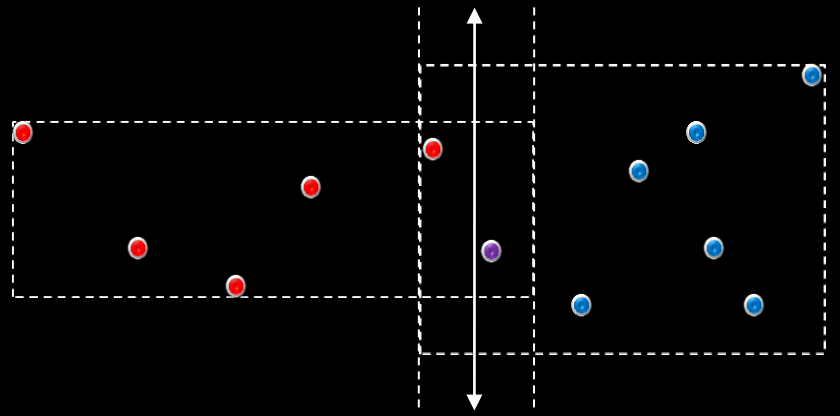
- Batching the data means…
  - We hide the cost of data transfer between device and host.
  - Extension to Multi-GPU is now trivial.
- NB. Not all algorithms can batch data.
- Each batch’s stream requires resource allocation.
  - Be sure to do this up-front, before OpenGL gets it all!
  - Number of batches can be chosen based on available resources.

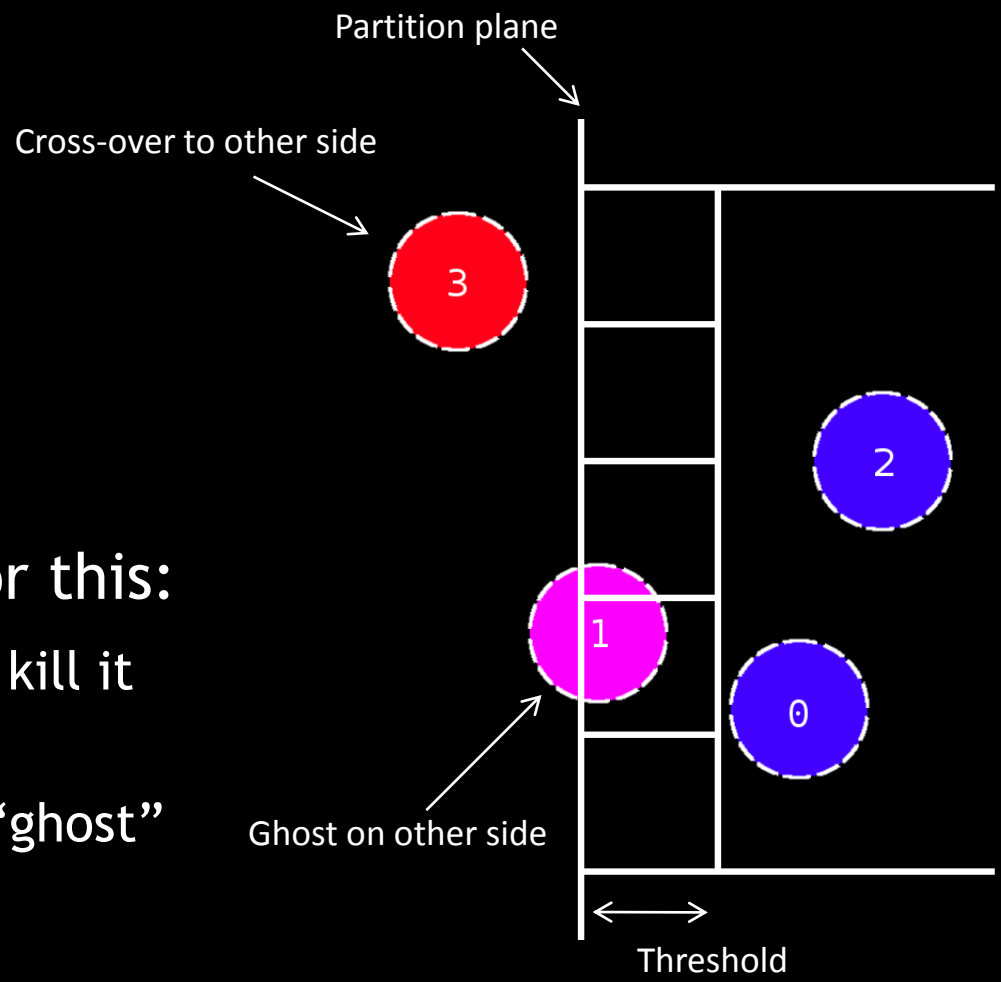
Example 4: Multi-GPU CUDA

- One GPU per batch



Multi-GPU – sharing the compute



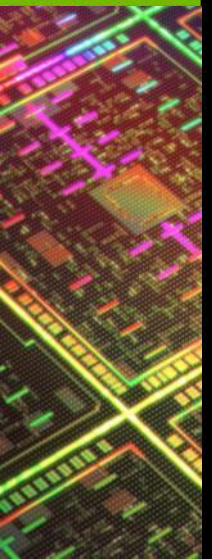


We already have the framework for this:

- If particle is beyond partition then kill it and emit on other device.
- If within threshold then emit as a “ghost” (i.e. it lives for only one update).

Fluid dynamics (in 1 slide)

- More interesting than boring particles.
- Works well on GPUs; it requires many data elements.
- Existing techniques:
  - Eulerian (grid-based)
  - Heightfield (2D / 2.5D grid)
  - Lagrangian (particle-based)



$$\rho\frac{d\mathbf{v}}{dt} = -\nabla P + \nabla.\mathbf{S} + \mathbf{F}$$

Momentum equation

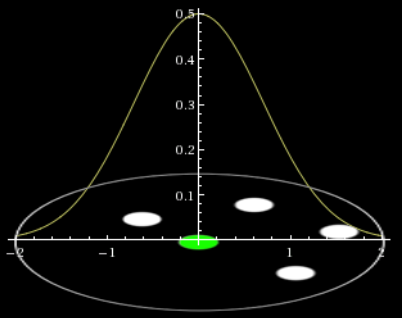
$$\frac{d\rho}{dt} + \rho(\nabla.\mathbf{v}) = 0$$

Continuity equation

SPH (in 1 slide)

$$A(\vec{x}) = \int A_b W(|\vec{x} - \vec{x_b}|, h) dV$$

$$A(\vec{x}) \approx \sum_b A_b \frac{m_b}{\rho_b} W(|\vec{x} - \vec{x_b}|, h)$$



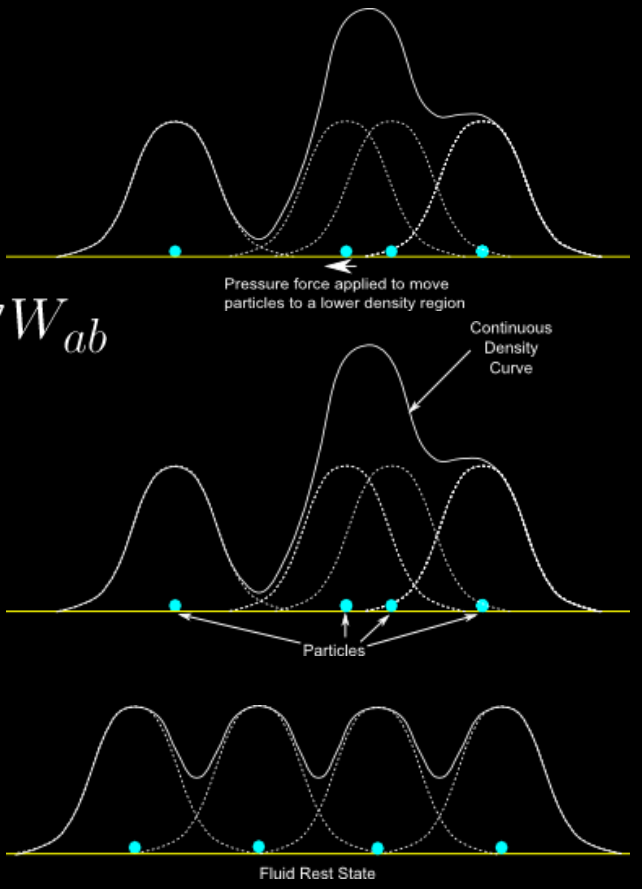
- Continuity equation:

$$\rho_a = \sum_b m_b W_{ab}$$

- Momentum equation:

$$\frac{d\vec{v}_a}{dt} = -\sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} \right) \nabla W_{ab}$$

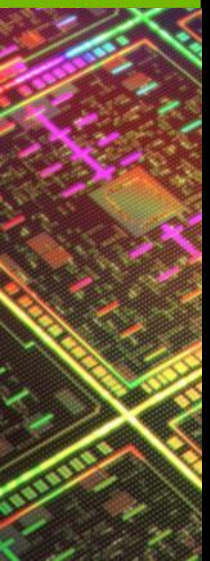
$$P = c^2(\rho - \rho_0)$$



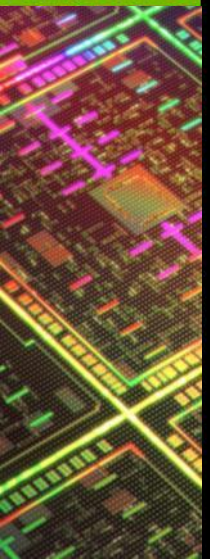
- Motion equation:

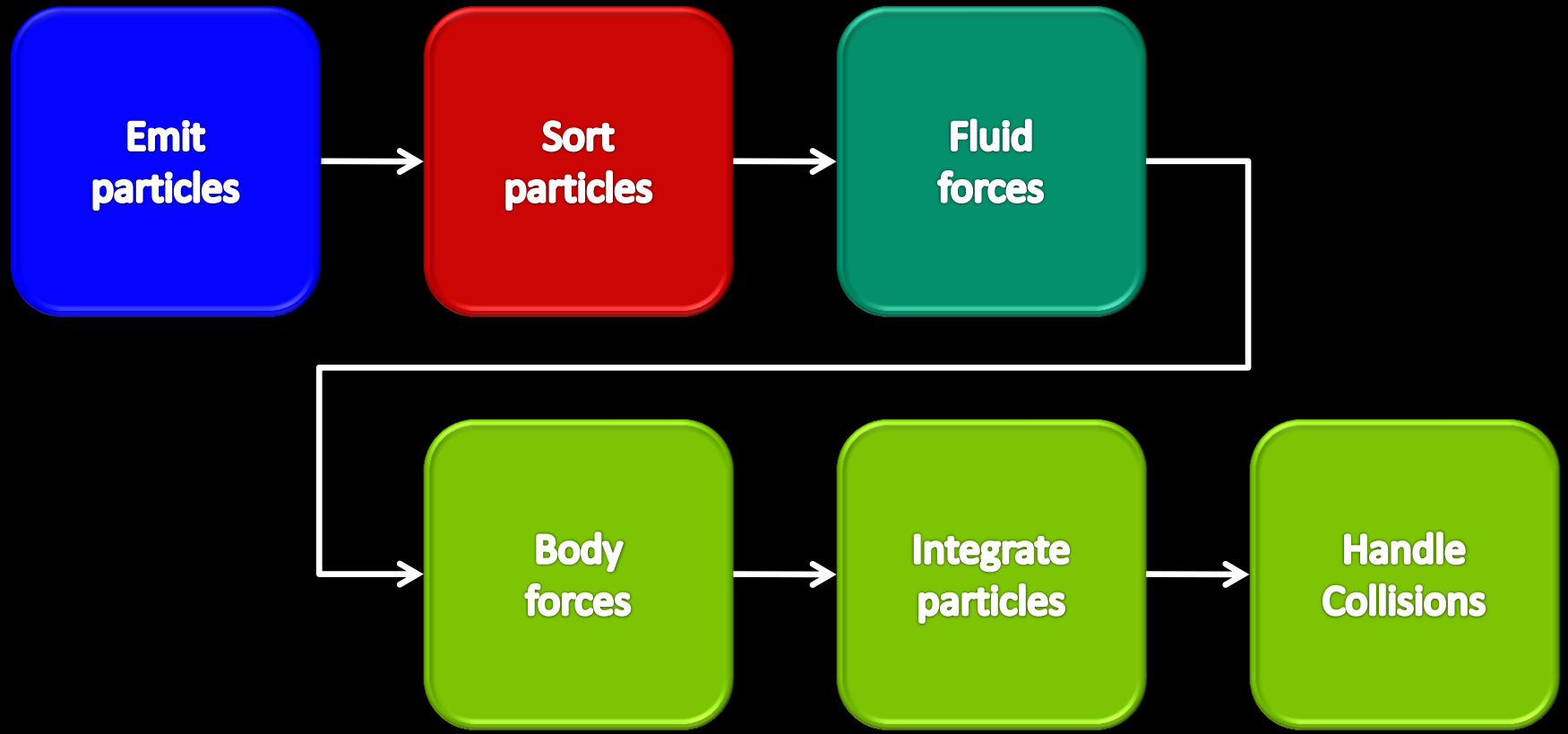
$$\vec{x}_i = \vec{x}_{i-1} + \vec{v}_{i-1/2} \Delta t$$

$$\vec{v}_{i+1/2} = \vec{v}_{i-1/2} + \frac{d\vec{v}}{dt} \Delta t$$



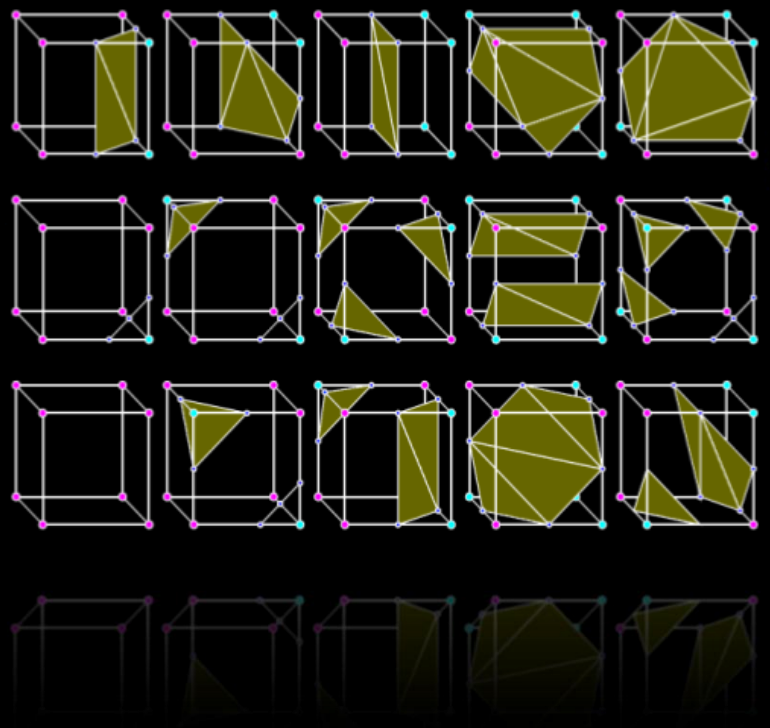
The life-cycle of particles (with SPH)



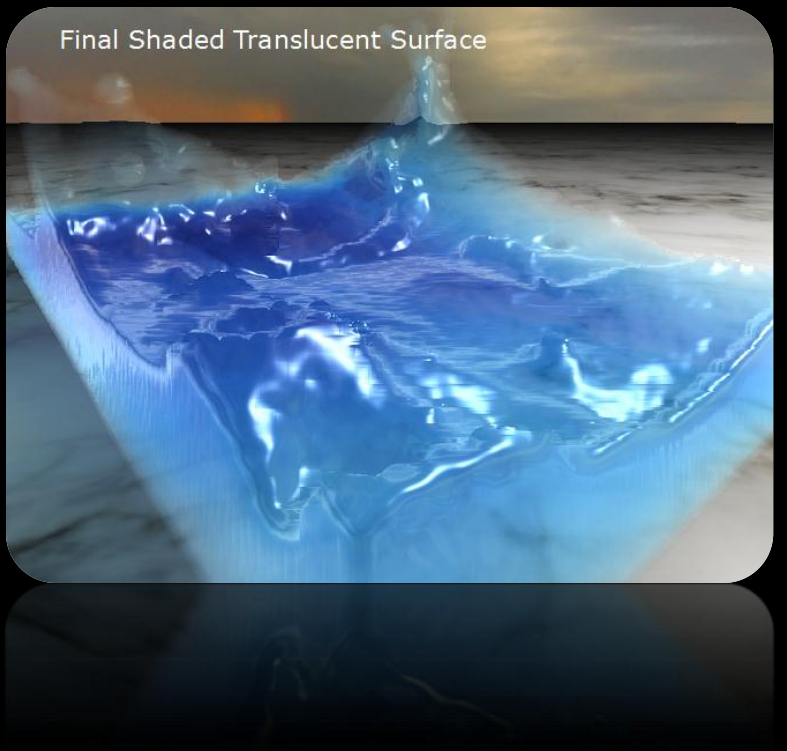


Rendering Particles as a fluid

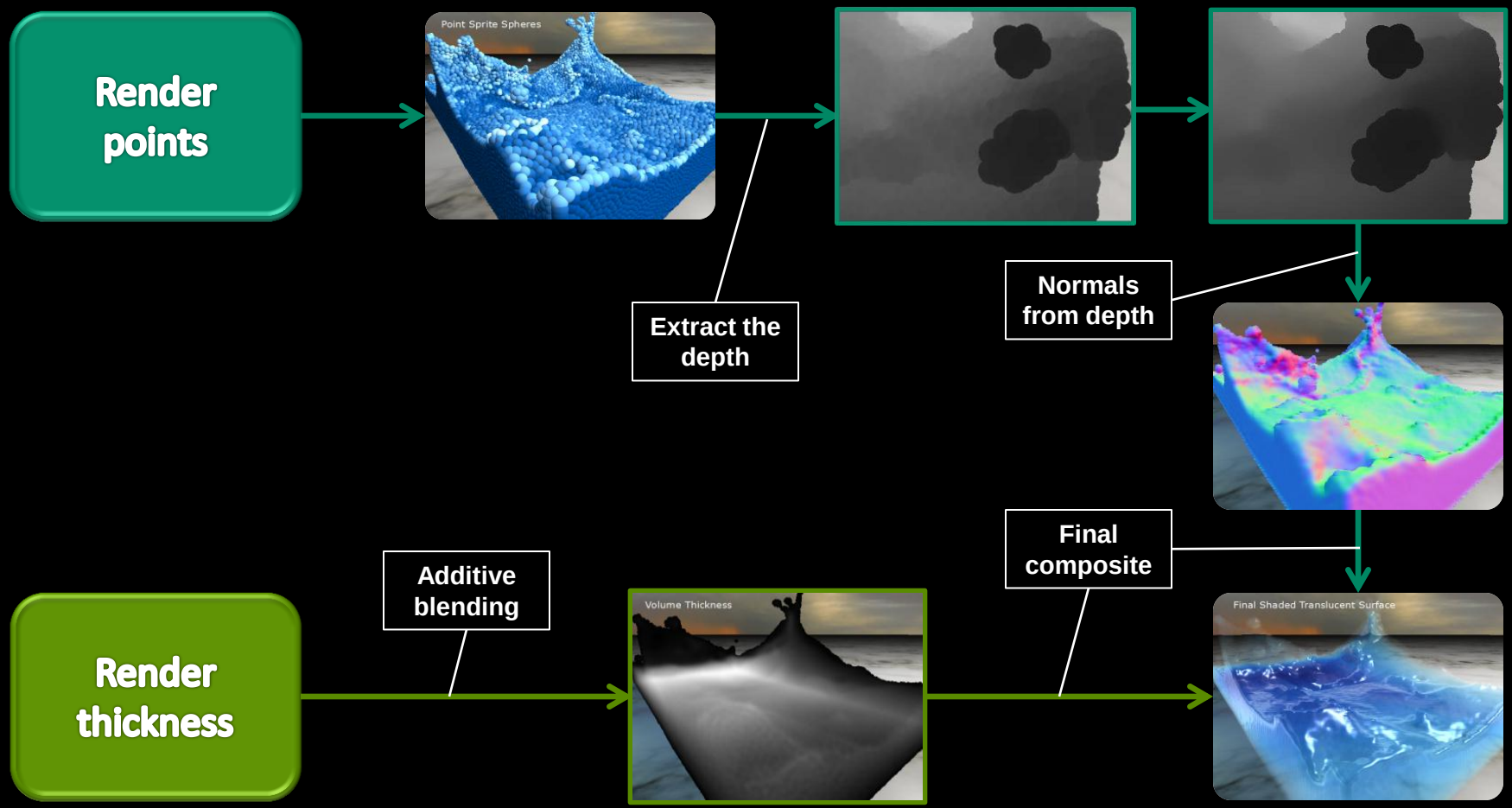
- Polygonize density field
  - e.g. Marching cubes, etc.



- Screen-space rendering



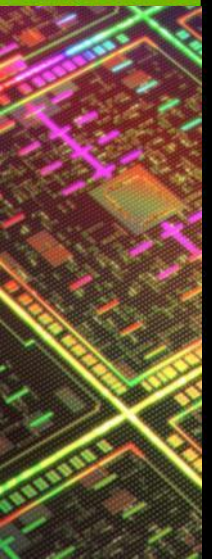
Rendering Particles in Screen-space

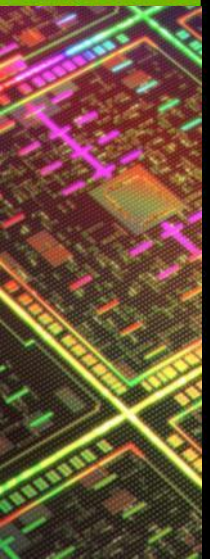


Future enhancements

- More solvers:
  - Crowds, rigid-bodies, non-newtonian materials, etc.
- More rendering styles:
  - OptiX, Volumes, Sprites, Instanced geometry, etc.

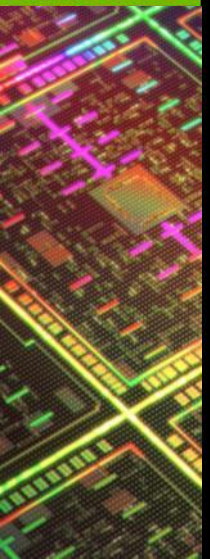
- Better fluid meshing using anisotropic kernels.
- Scriptable particle expressions using a virtual-machine.
- Maya 2013 & Viewport 2.0 integration.





Conclusions

- Building CUDA-accelerated particles in Maya is easy.
  - Leverages some of Maya's existing framework.
- Using techniques like overlapping and multi-GPU can provide even more performance.
- By using Maximus and simulating on the Tesla card, Maya remains interactive.



Acknowledgments

- R. Bridson: “Fluid Simulation for Computer Graphics”, A K Peters, 2008
- W. J Van Der Laan, S. Green, Miguel Sainz:
  “Screen space fluid rendering with curvature flow” - Proceedings of the 2009 symposium on Interactive 3D graphics and games
- M. Becker, M. Teschner: “Weakly compressible SPH for free surface flows” – Eurographics / ACM SIGGRAPH Symposium on Computer Animation, 2007
- M. Ihmsen, N. Akinci, M. Gissler, M.Teschner: “Boundary handling and adaptive time-stepping for PCISPH” – VRIPHYS, 2010
- S. Green, “Particles” demo, NVIDIA CUDA SDK.