Real-Time Visualization

01: Introduction

Stefan Bruckner

Lectures

1. **Introduction:** Volume rendering basics, GPU architecture, OpenGL, CUDA, case studies, … (03.05.)
2. **Real-Time Volume Graphics 1:** GPU ray-casting, optimizations, memory management, OpenGL vs. CUDA, … (10.05.)
3. **Real-Time Volume Graphics 2:** Advanced illumination, filtering, derivatives, advanced transfer functions, … (17.05.)
4. **Real-Time Computations on GPUs:** Fluid simulation, level-set deformation, … (24.05.)
5. **Volumetric Special Effects:** combining simulation and ray-casting, integration in game engine scenes, … (31.05.)
6. **Final event:** Project presentations (28.06.)

Today

Examples of real-time visualization

- Volume visualization
- Flow visualization
- Volumetric special effects
- Computations: Diffusion filtering, level sets, …

Volume rendering basics

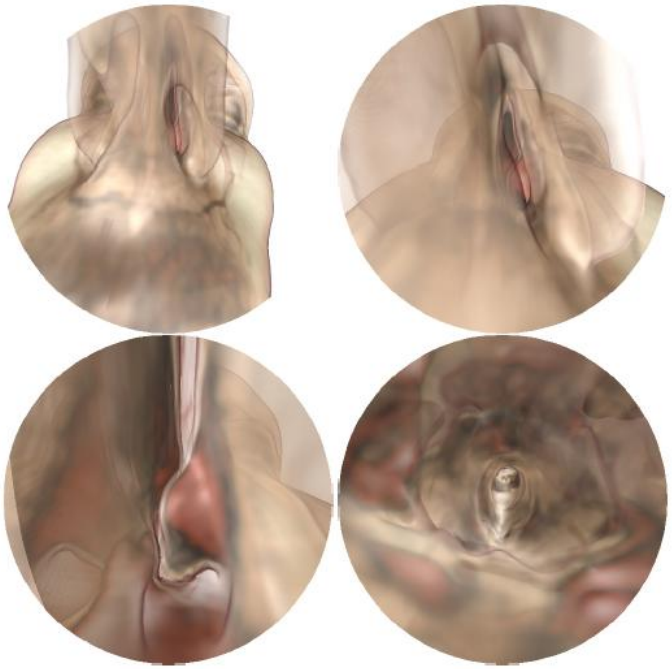
Hardware architectures

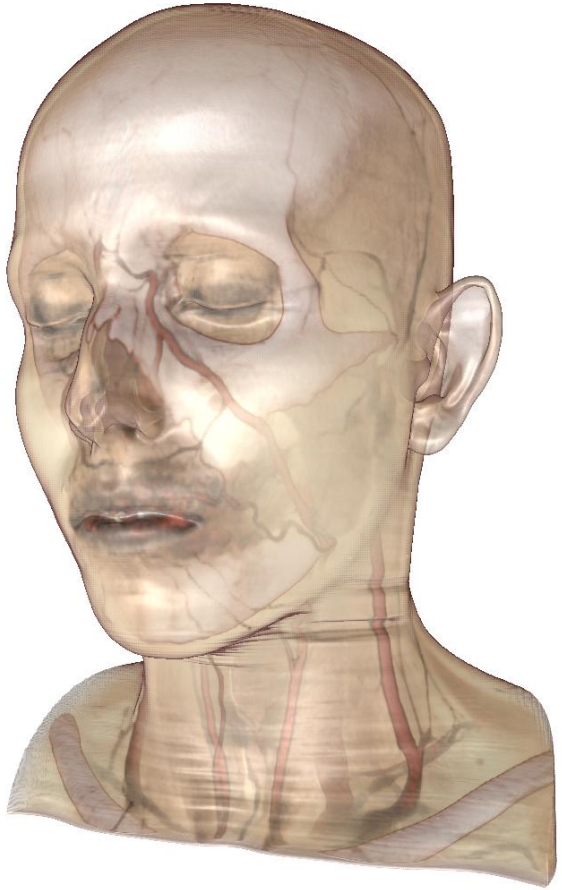
GPGPU & CUDA

Volume Visualization

Real-time ray-casting

- High quality, flexibility, and performance
- Perspective views, advanced lighting, …



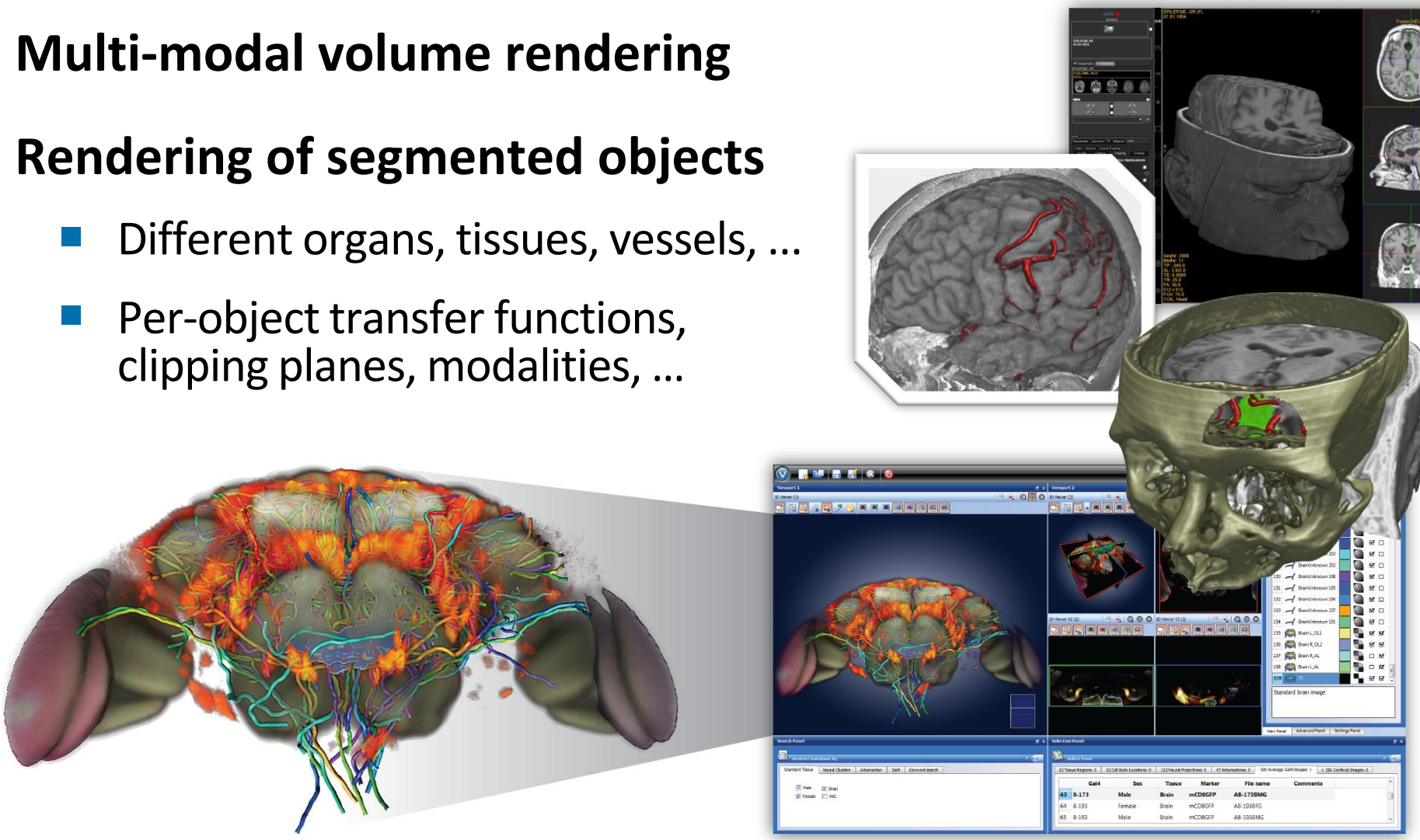


Multi-Volume Rendering

Multi-modal volume rendering

Rendering of segmented objects

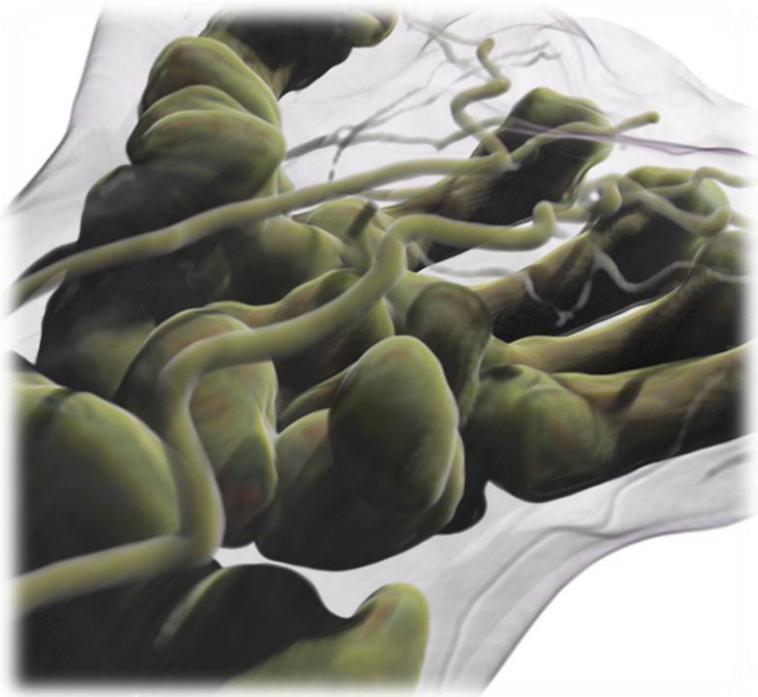
- Different organs, tissues, vessels, …
- Per-object transfer functions, clipping planes, modalities, …

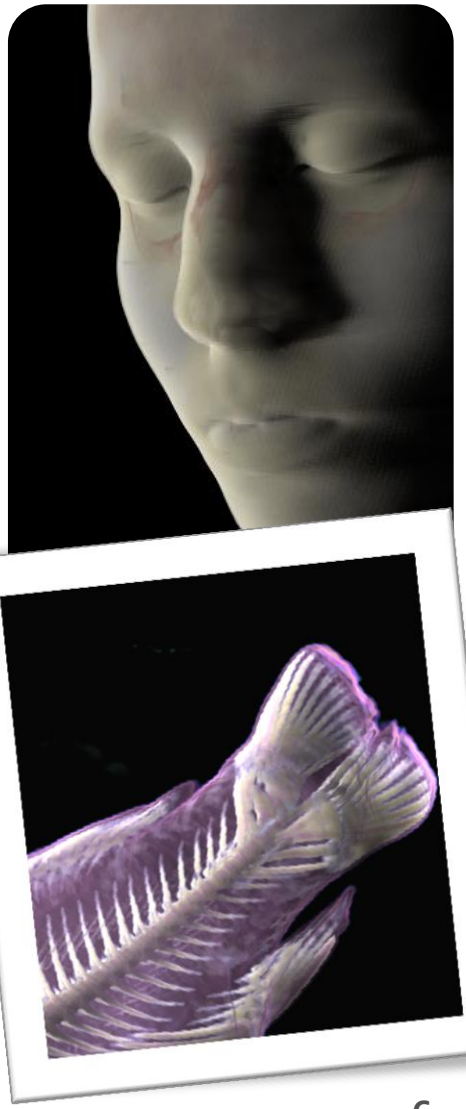


Advanced Lighting (1)

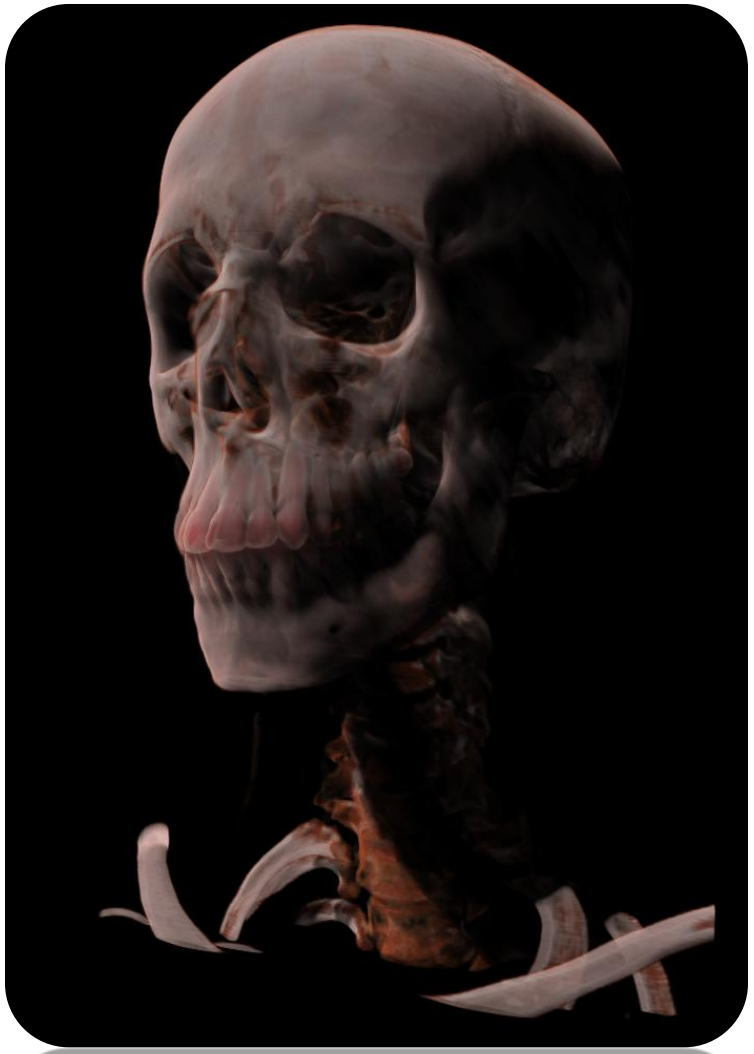
Hard shadows, soft shadows

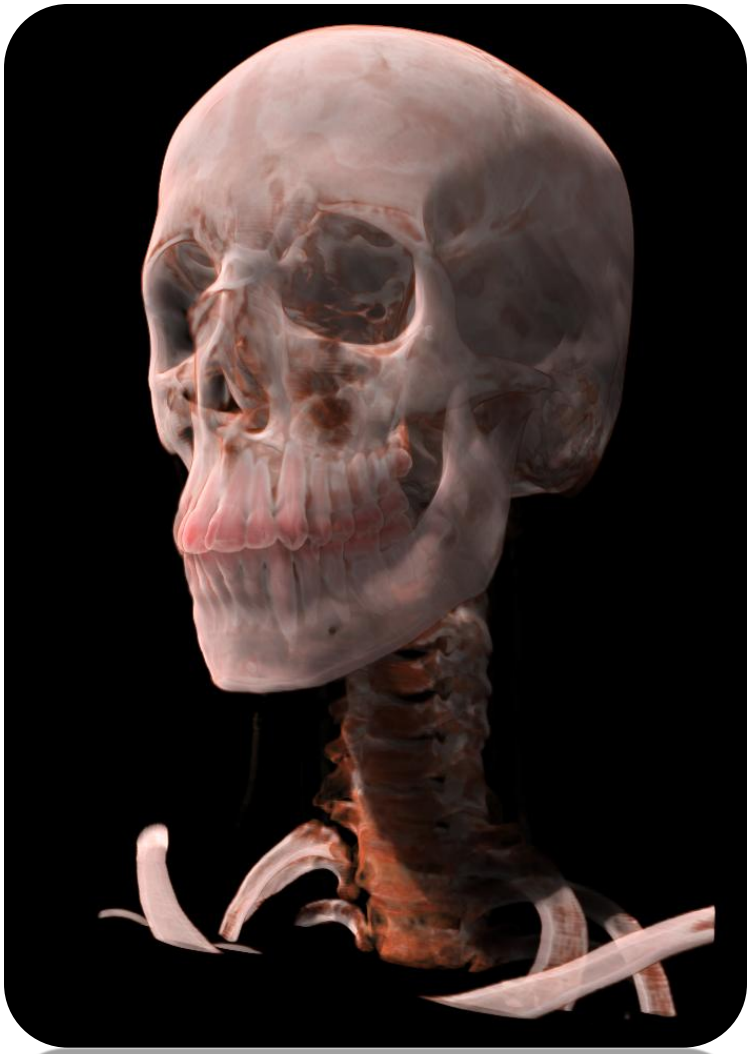
Ambient occlusion, scattering, …





Advanced Lighting (2)



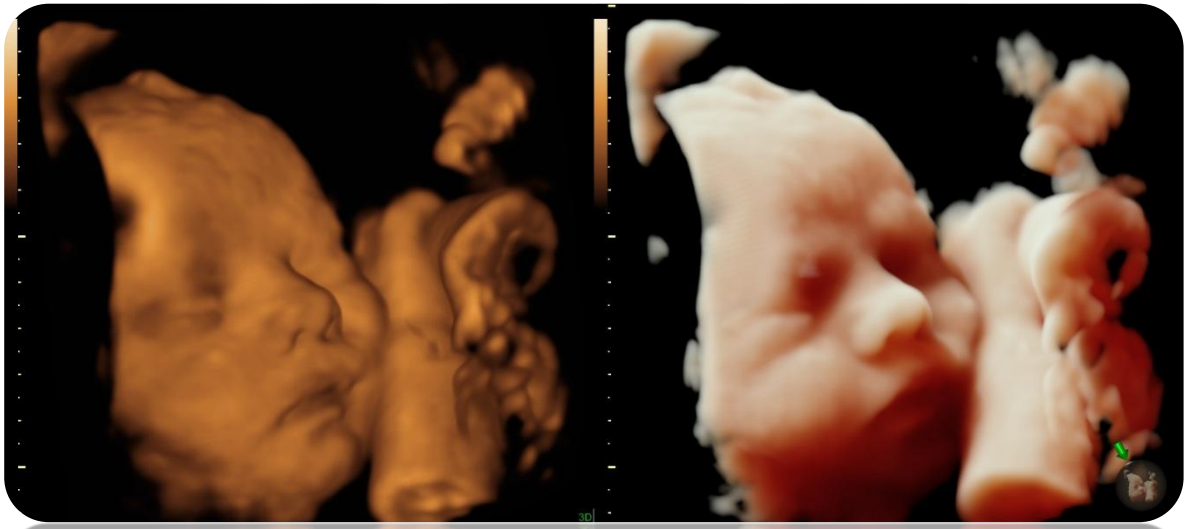


Advanced Lighting (3)

Optical Fetascope

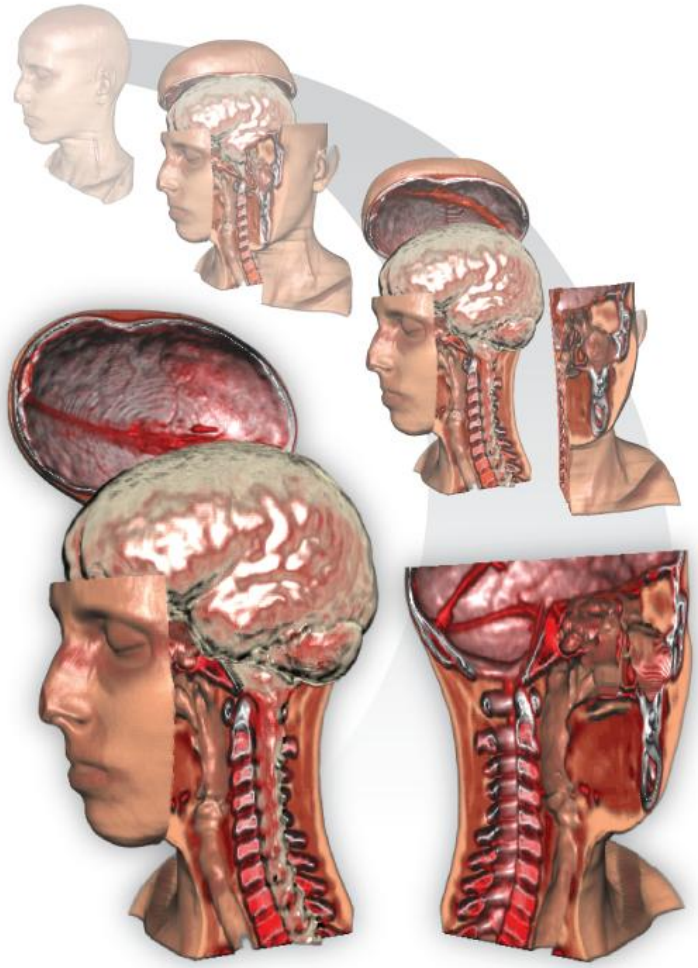
3D Ultrasound

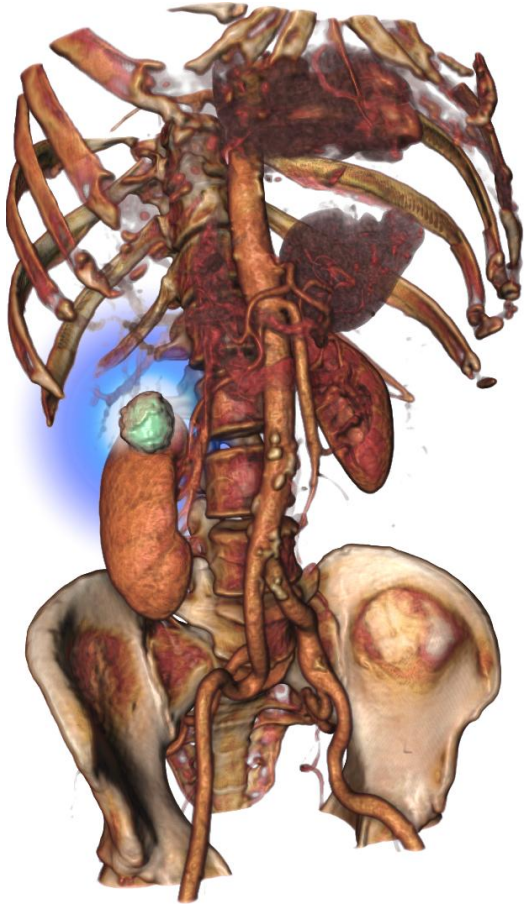
Emission + Absorption + Surface Shading

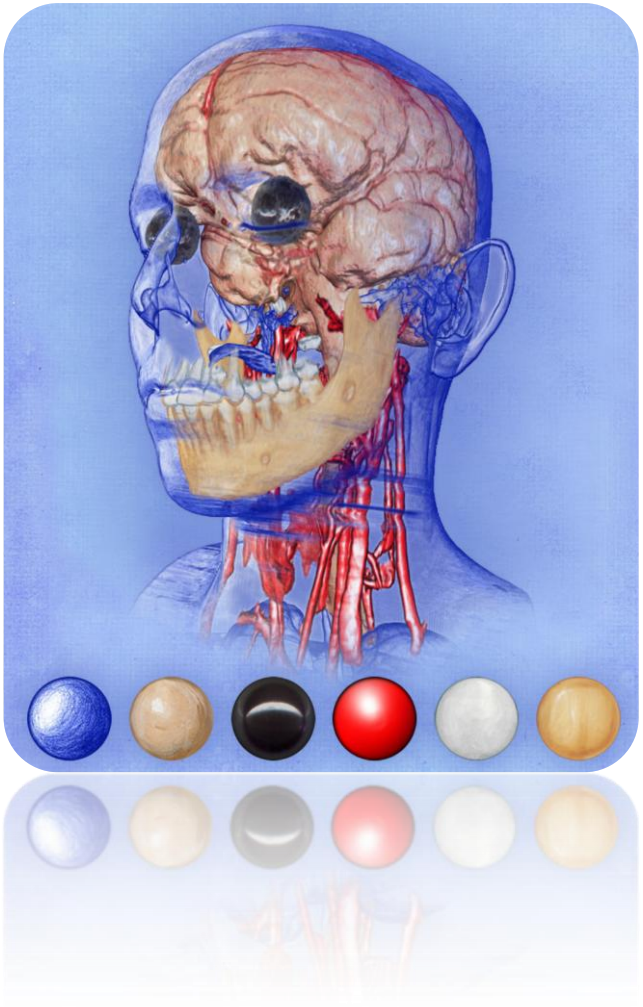


Soft Shadows + Scattering

Direct Volume Illustration



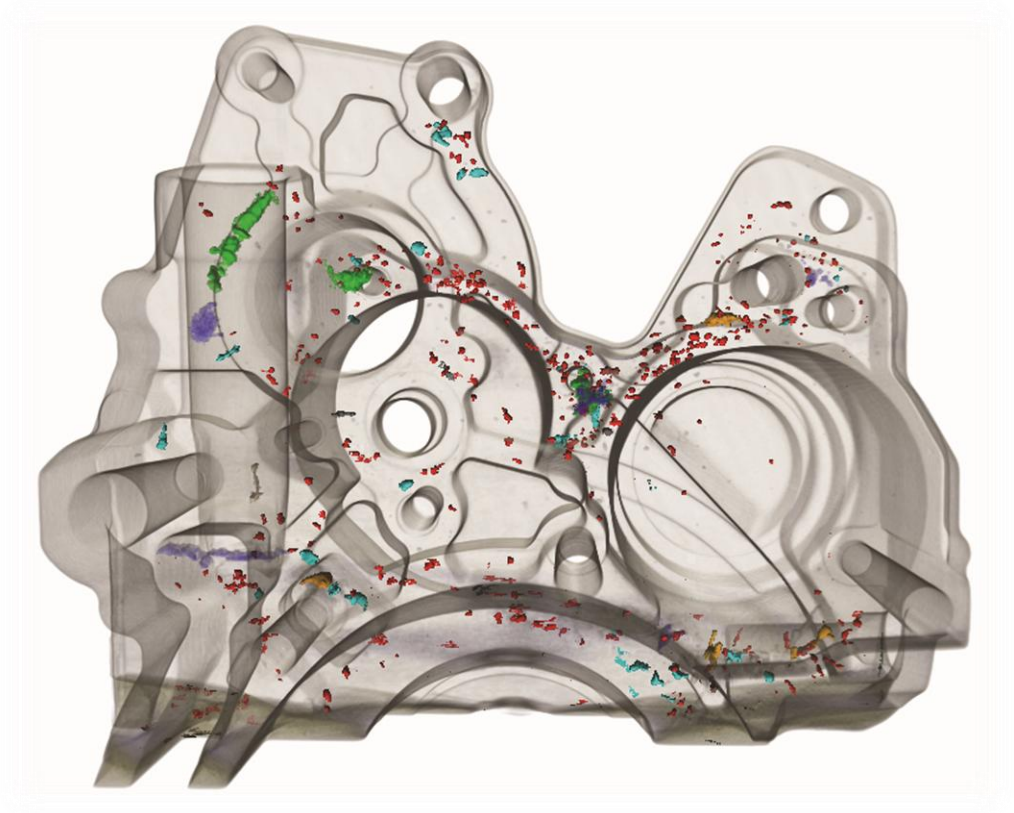


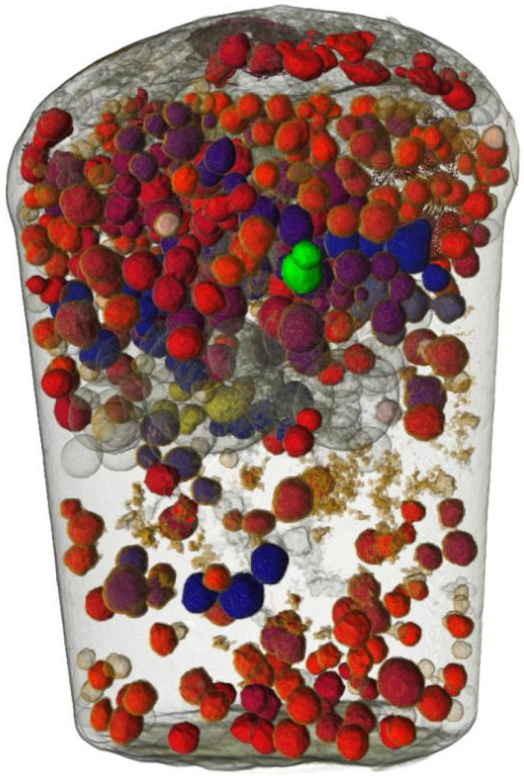


Non-destructive Testing

Visualization of high-resolution CT volumes

- Combination with feature detection & quantification





Fluid Simulation and Rendering

Compute advection of fluid

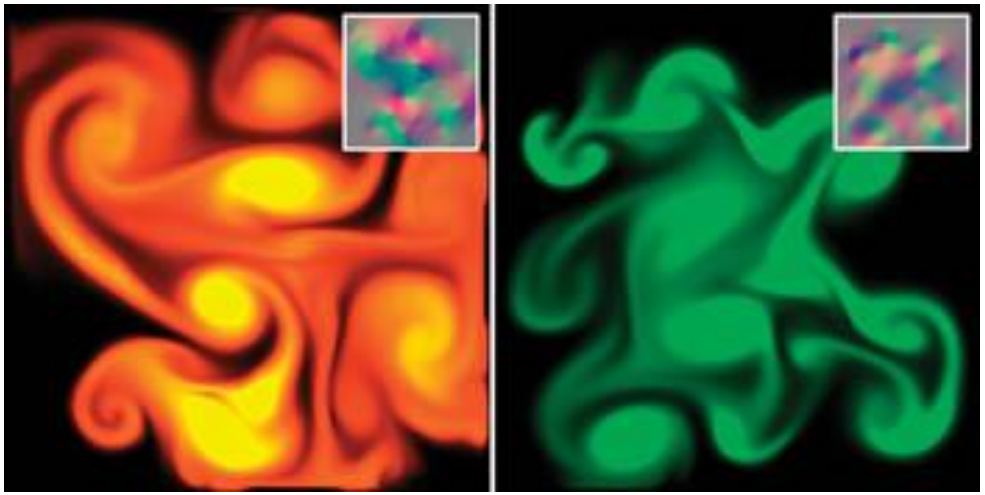
- (Incompressible) Navier-Stokes solvers
- Lattice Boltzmann Method (LBM)

Discretized domain; stored in 2D/3D textures

- Velocity, pressure
- Dye, smoke density, vorticity, …

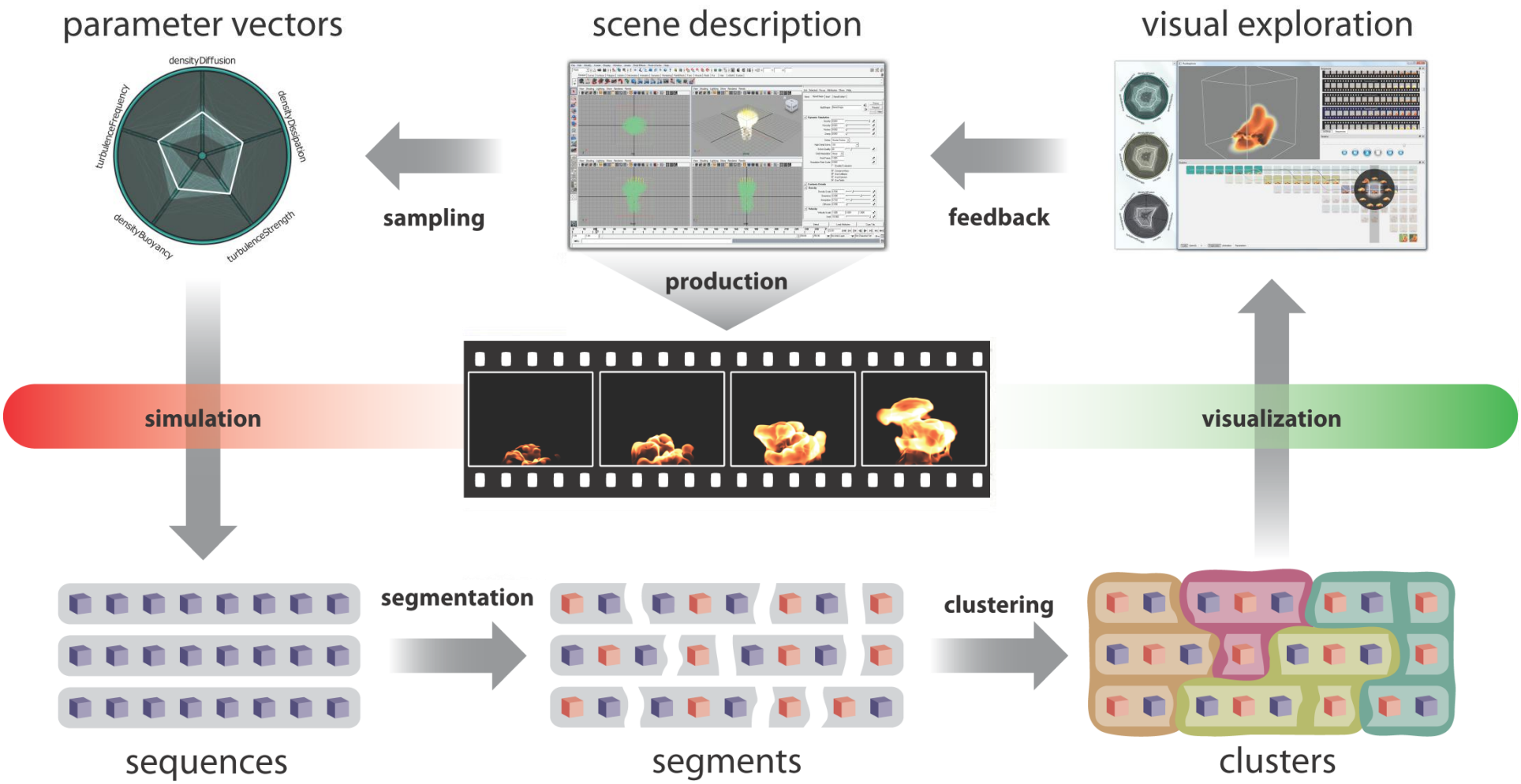
Updates in multi-passes

Render current frame



Courtesy Mark Harris

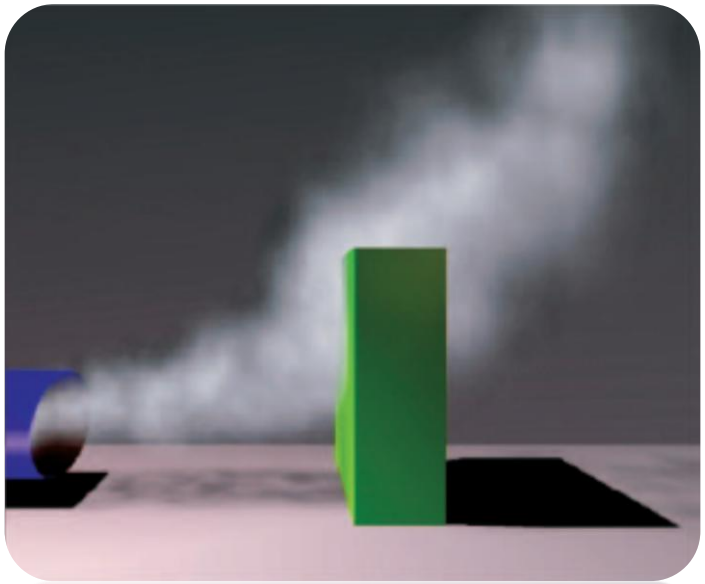
Parameter Space Exploration



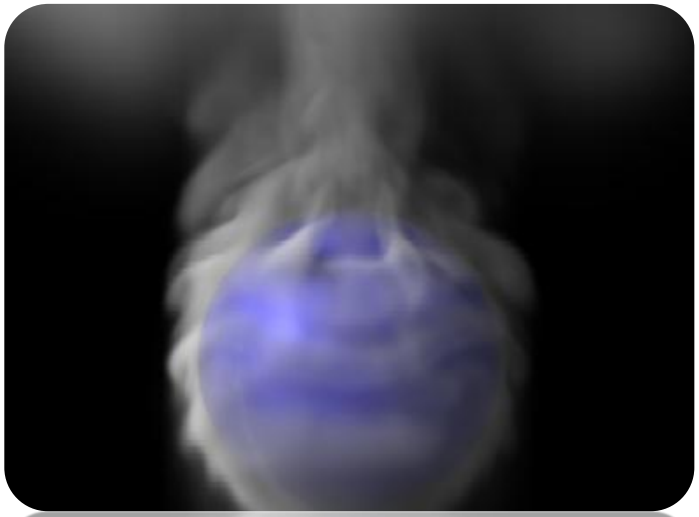
Volumetric Special Effects (1)

Solve on (low-res) 3D grids for volumetric special effects

- Render using ray-casting or splatting



Wei et al.

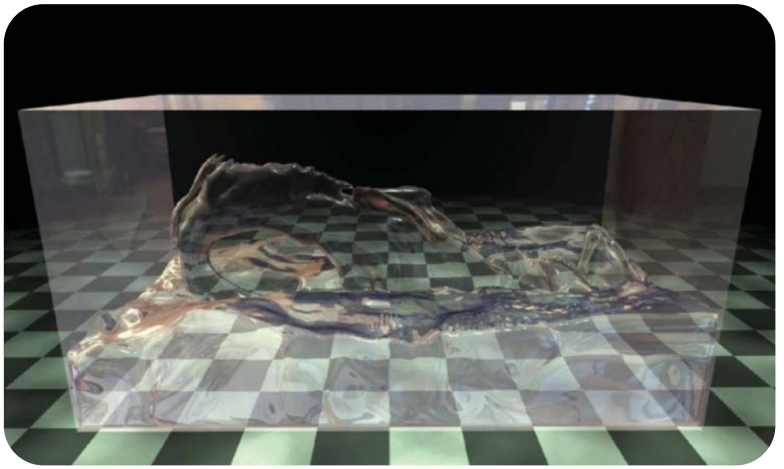


Fedkiw et al.

Volumetric Special Effects (2)

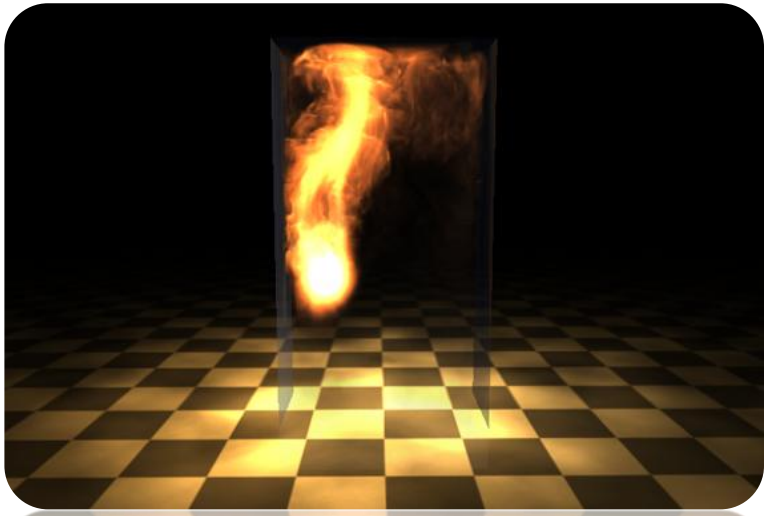
NVIDIA Demos

- Smoke, water
- Collision detection with voxelized solid (Gargoyle)



Ray casting

- Smoke: volume rendering
- Water: level set / isosurface



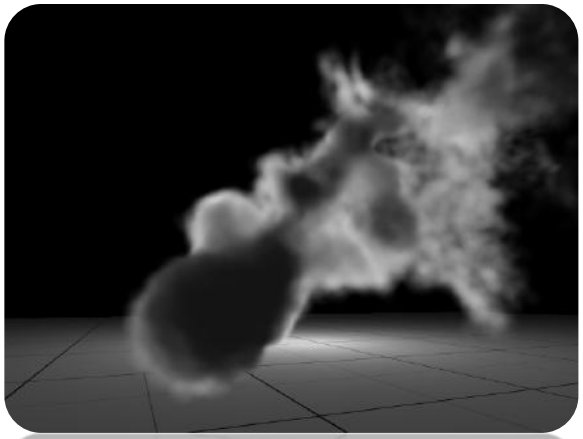
Courtesy Keenan Crane

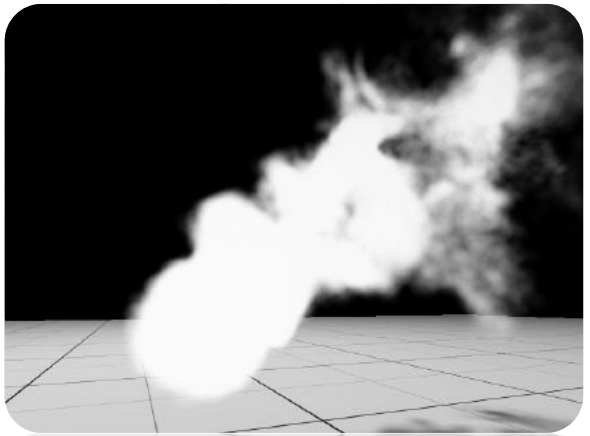
Volumetric Special Effects (3)

Render smoke as particles

Compute shadows using half-angle slicing, similar to use in volume rendering

Simon Green / NVIDIA CUDA SDK





Volumetric Special Effects (4)

Cloud rendering

Soft particles

- MS Flight Simulator
- Crysis

Crysis / Crytek

Artist-oriented approach

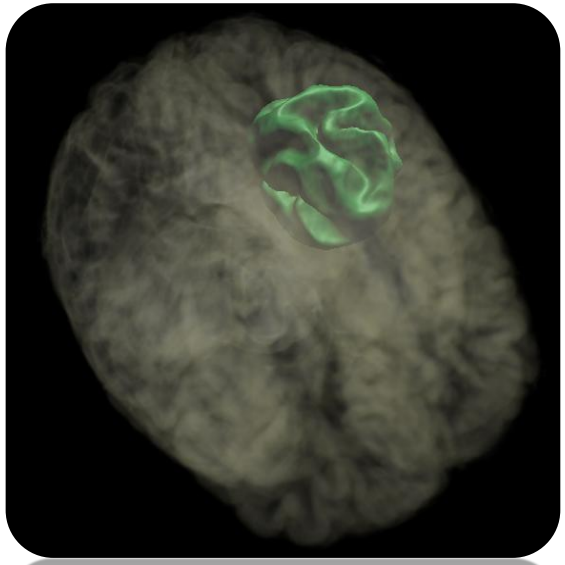
- Niniane Wang, Journal of Graphics Tools article

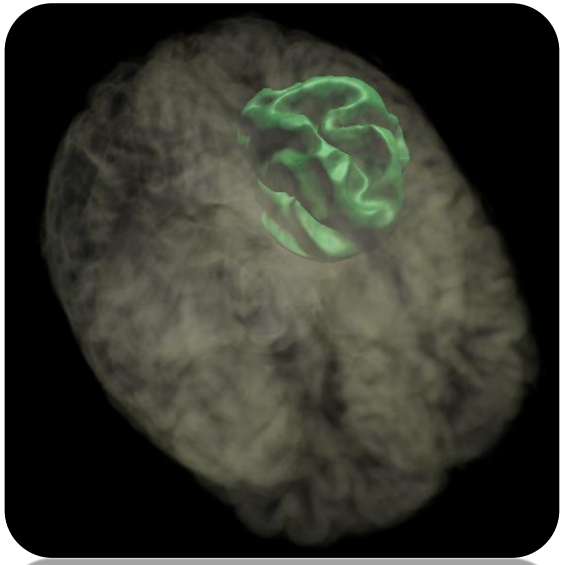
Level-Set Computations

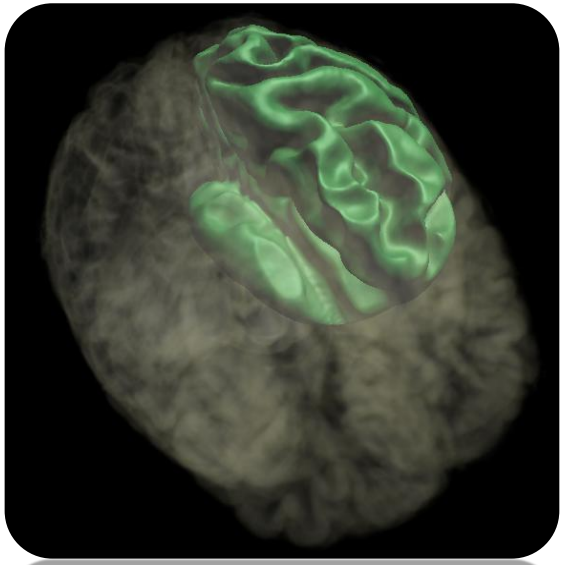
Implicit surface represented by distance field

Level-set PDE is solved to update the distance field

Basic framework with a variety of applications

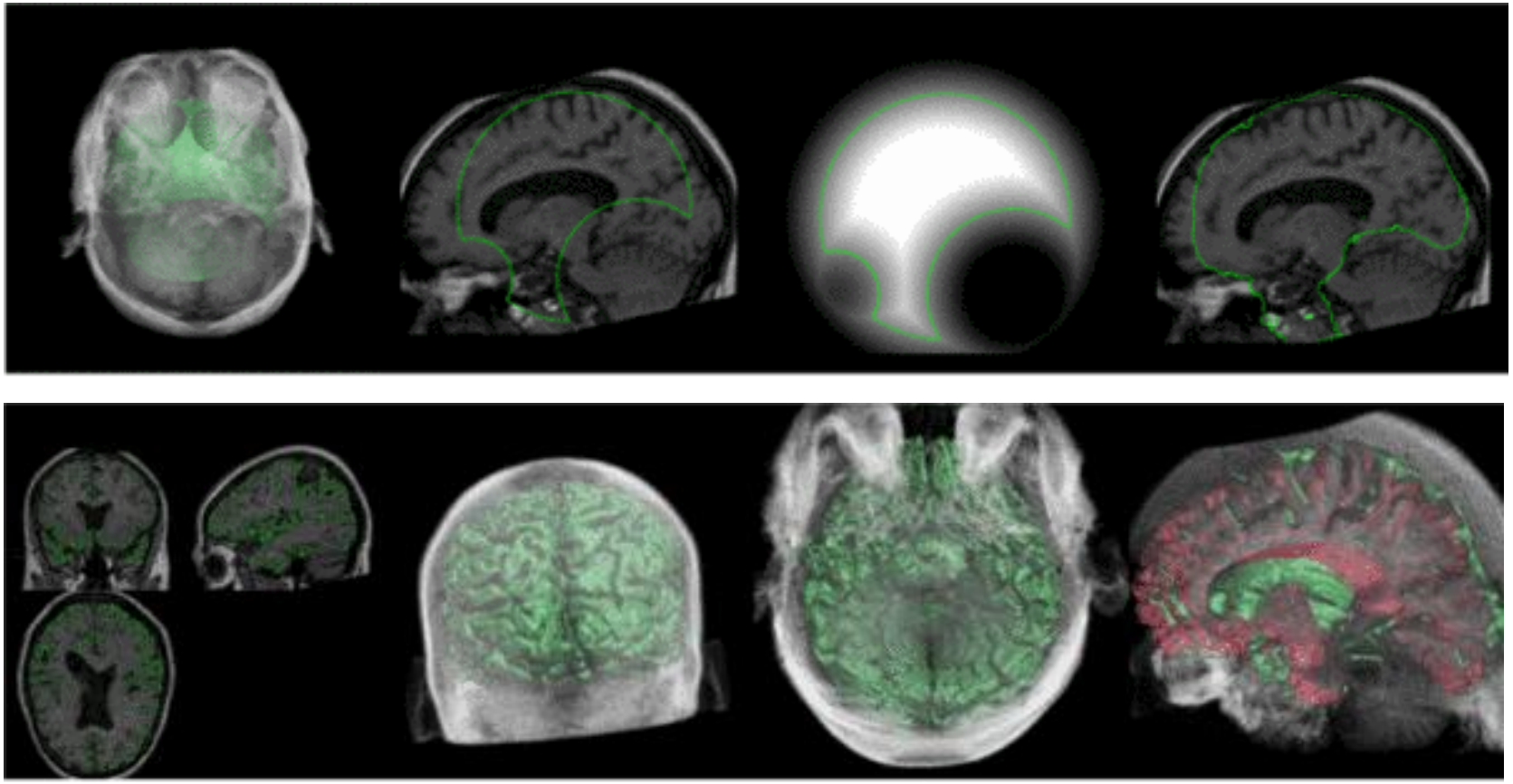






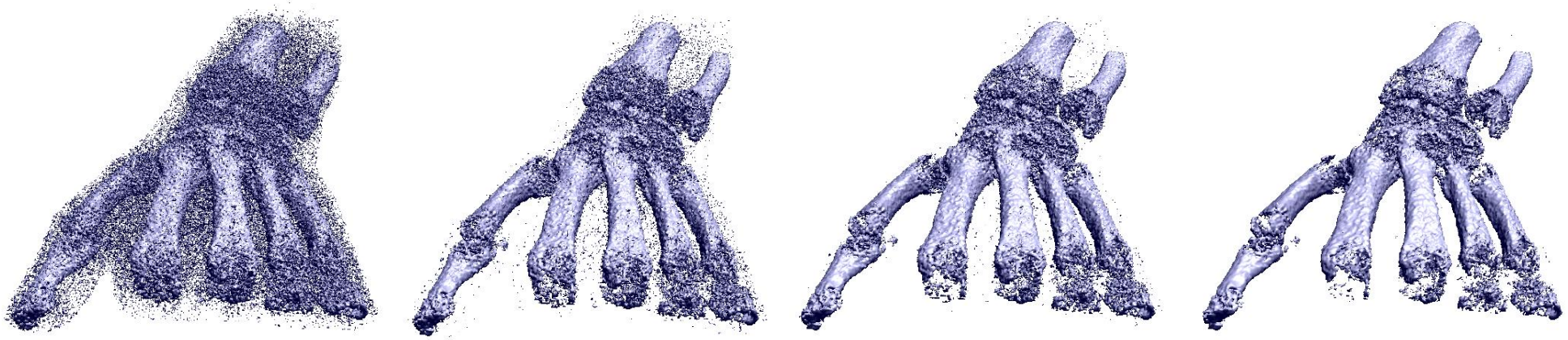
Level-Set Segmentation

Speed function pulls/retracts level set to/from target



Diffusion filtering

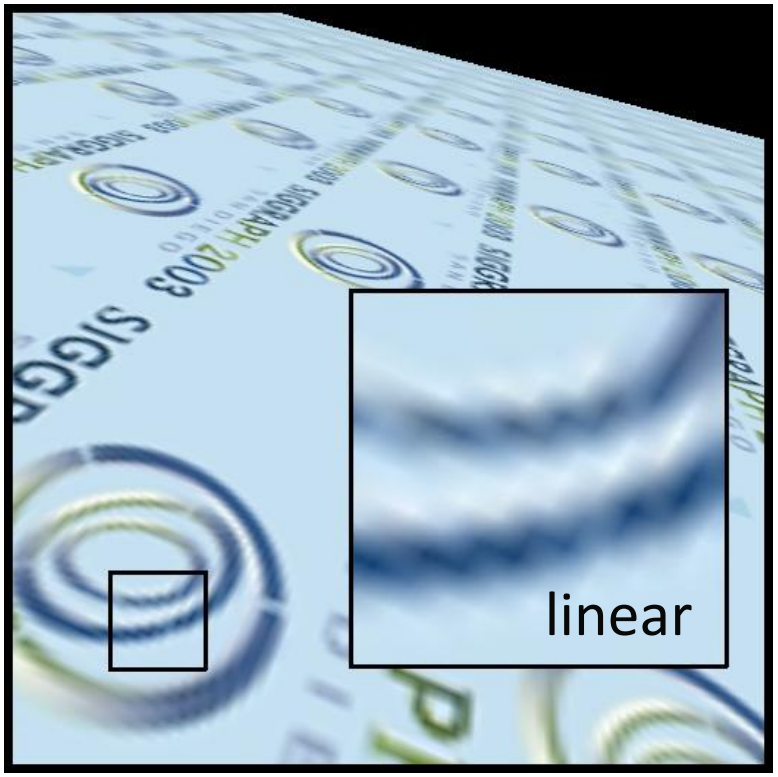
Anisotropic diffusion

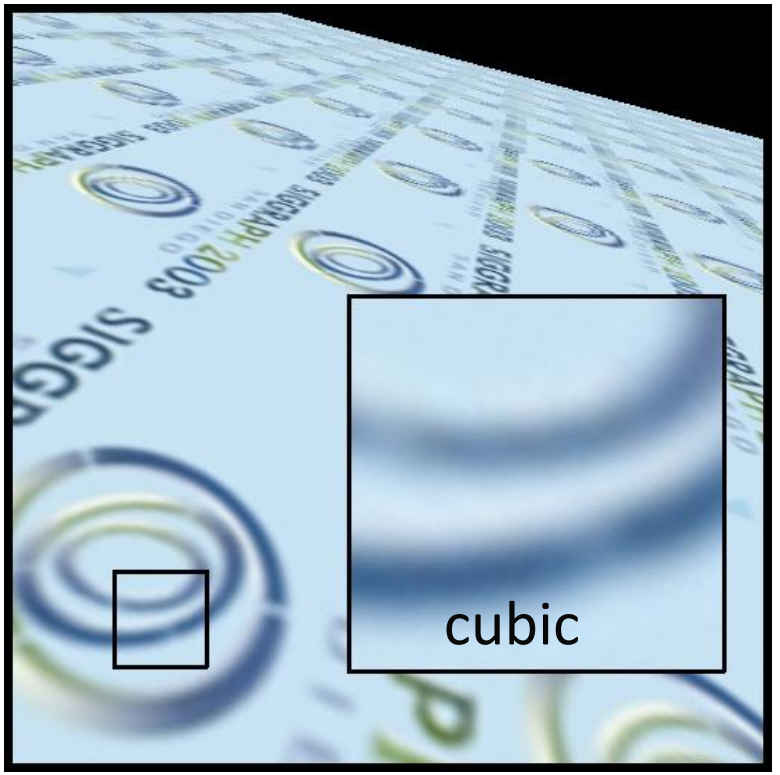


High-Quality Filtering (1)

Use higher-order filter kernels, e.g., cubic B-spline

- Compute weights in shader/kernel, or read from texture

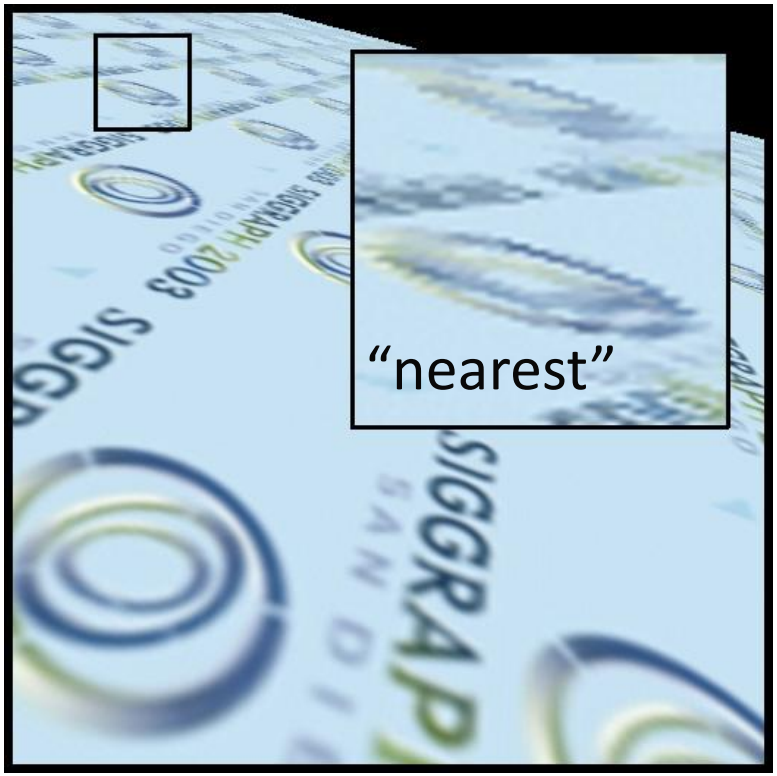


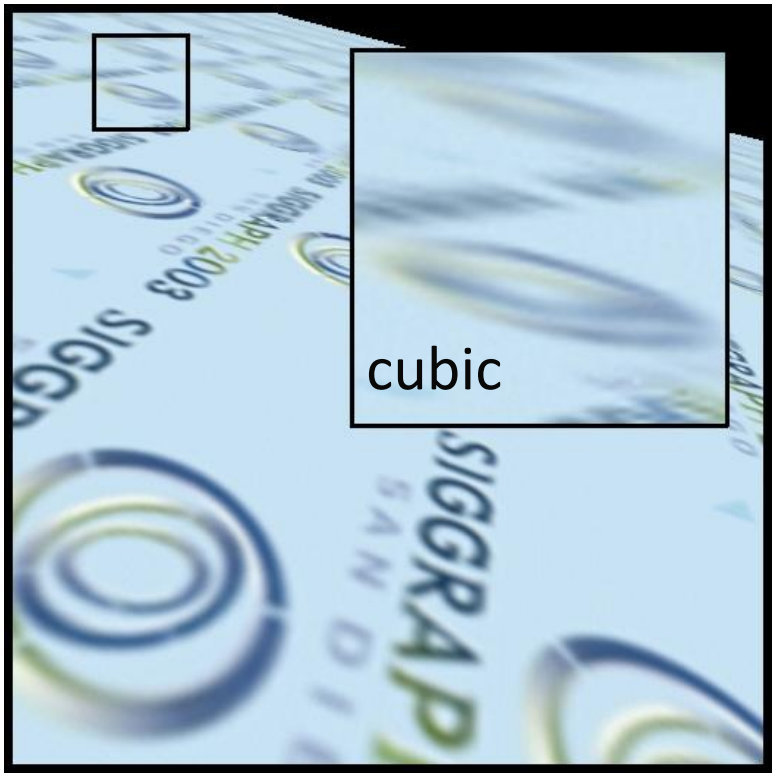


High-Quality Filtering (2)

Works for 1D, 2D, 3D textures

Can be combined with MIP-mapped textures

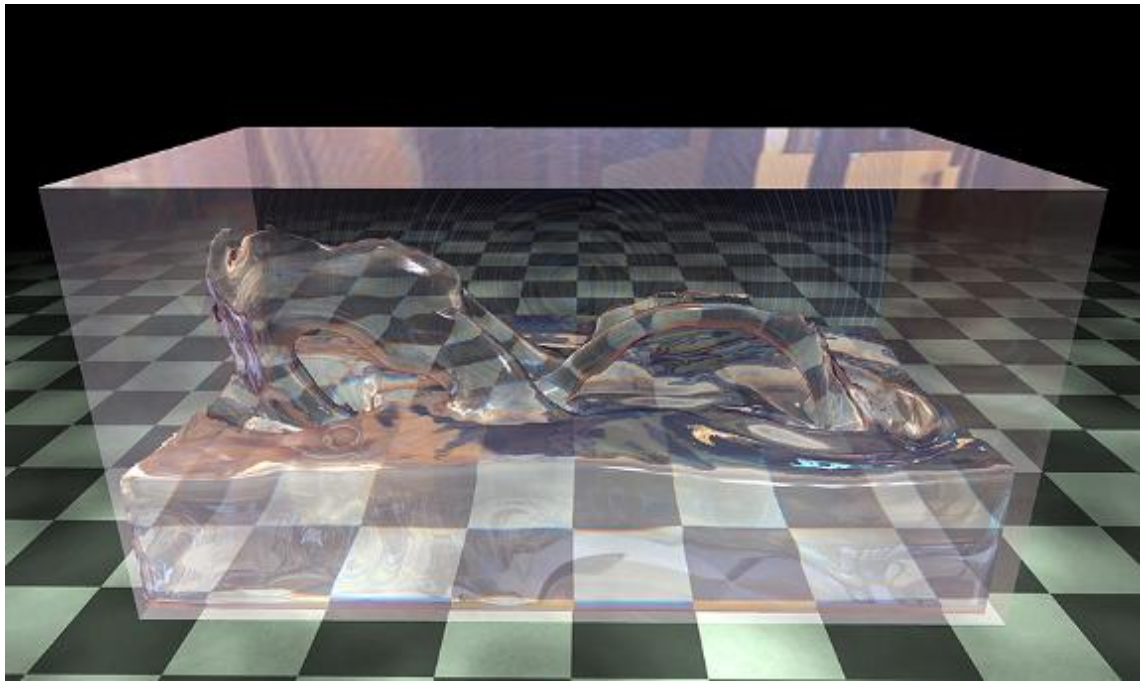




High-Quality Filtering (3)

Fluid surface represented and computed as level set

- Isosurface ray-casting to depict level set
- Tri-cubic filtering for high-quality surface interpolation



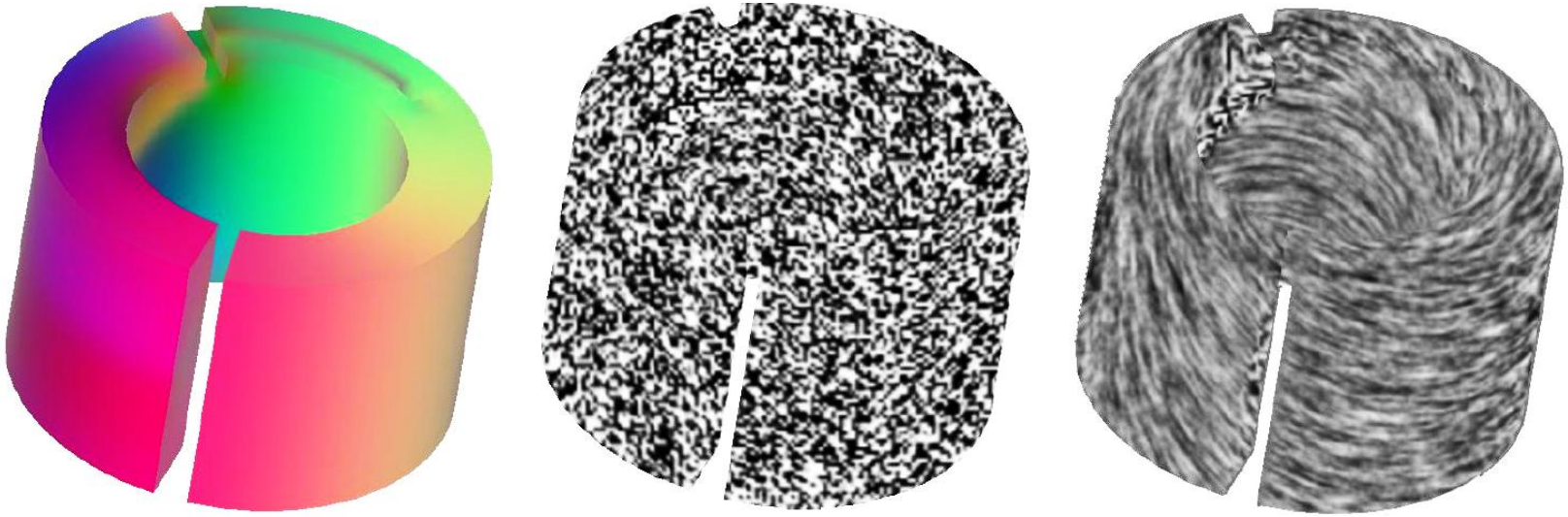
GPU Gems 3,
Keenan Crane et al.

Flow on Surfaces

Advect noise patterns in direction of vector field

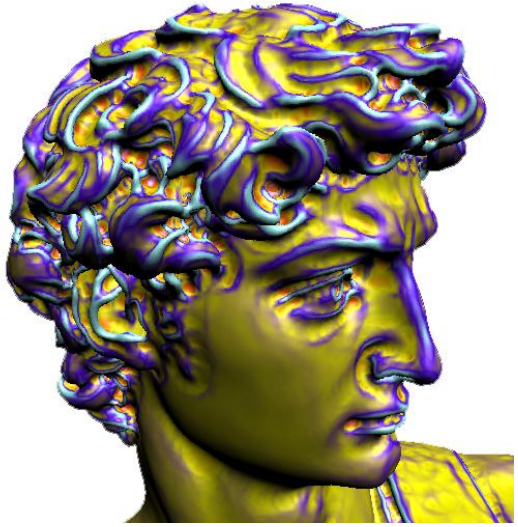
Done using 3D vector field projected to 2D image space

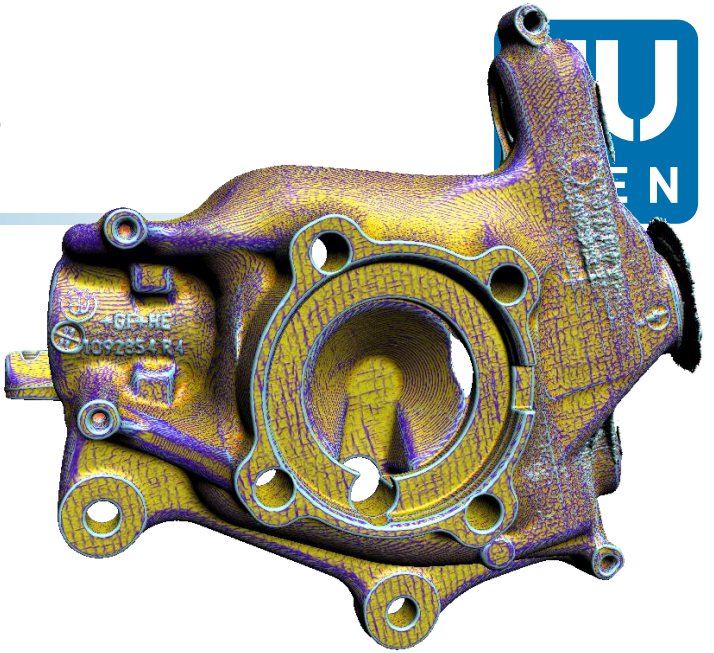
Flow visualization on curved surfaces



Courtesy Bob Laramee

Curvature of Implicit Surfaces





Volume Rendering

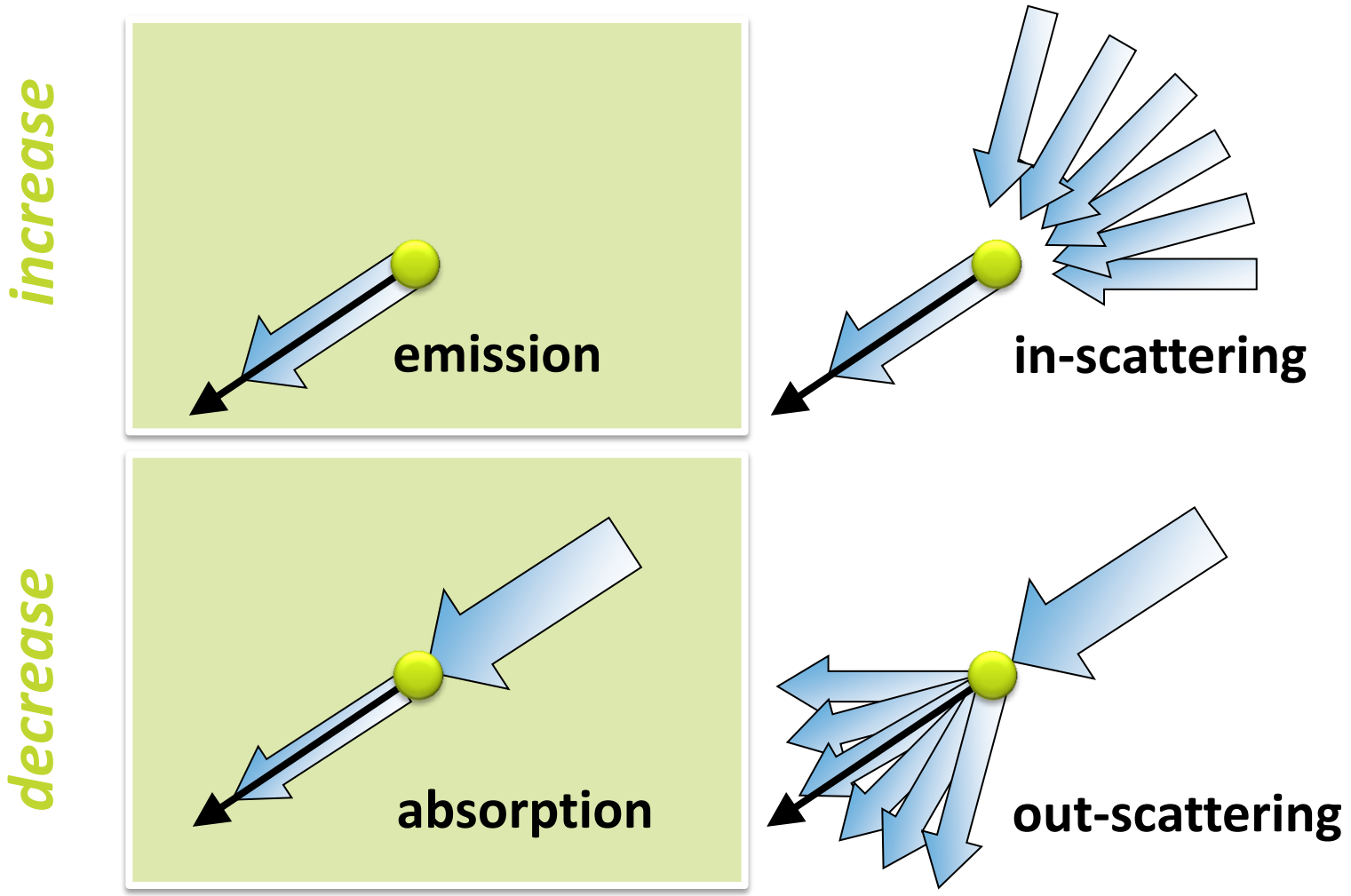
Visualization of a 3D field

- No explicitly defined surfaces
- Each point in space can emit and absorb energy

Most commonly based on sampled representation

- Regular grids
- Curvilinear grids
- Point-based representations

Physical Model



Physical Model

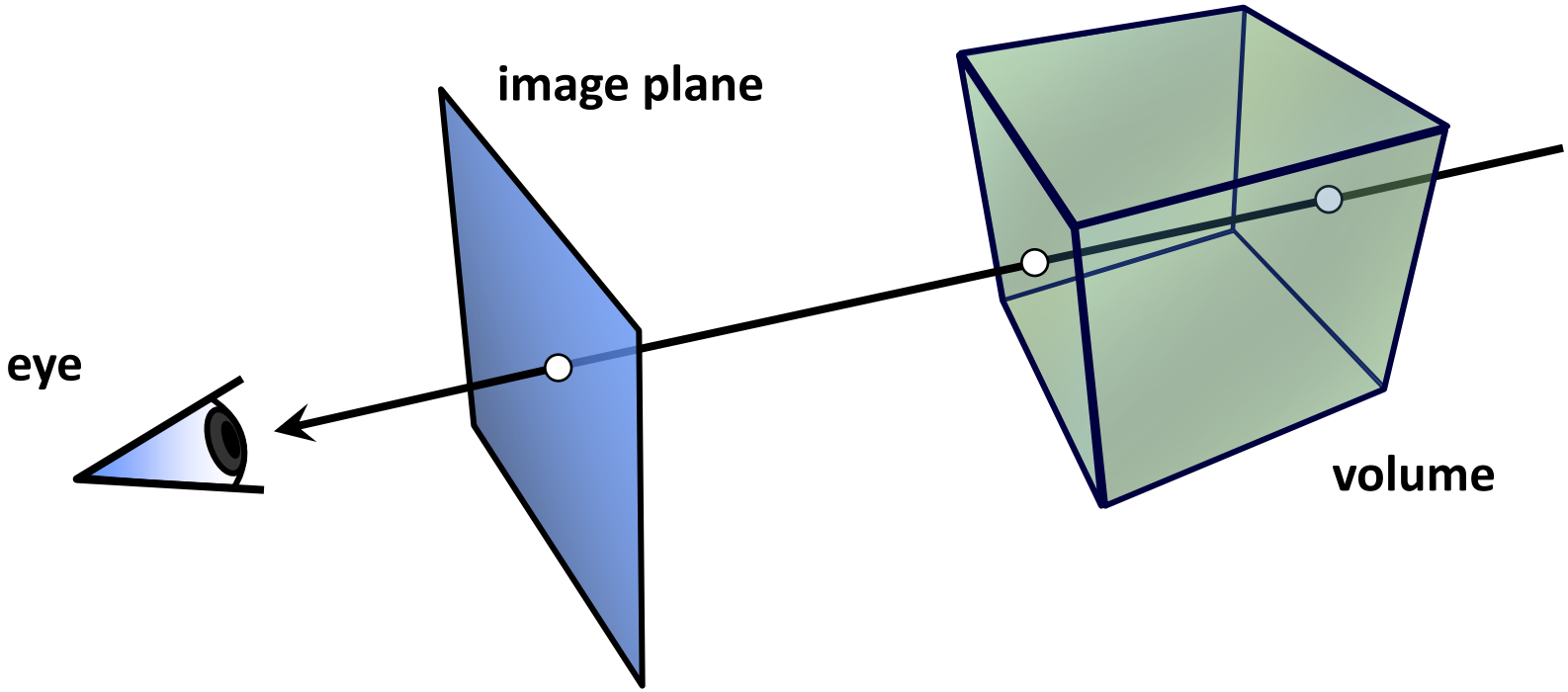
Conventional volume rendering uses an emission-absorption model

Scattering effects are usually ignored due to high computational complexity

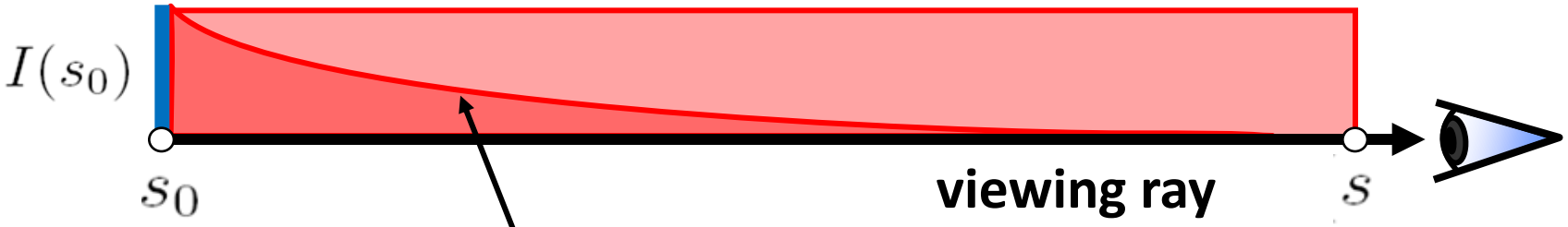
For each pixel on the image plane, a the ray integral has to be solved

Image-space approach for solving the ray integral: volume ray casting

Ray Integration (1)



Ray Integration (2)



Absorption along the ray segment $s_0 - s$

Initial intensity at s_0

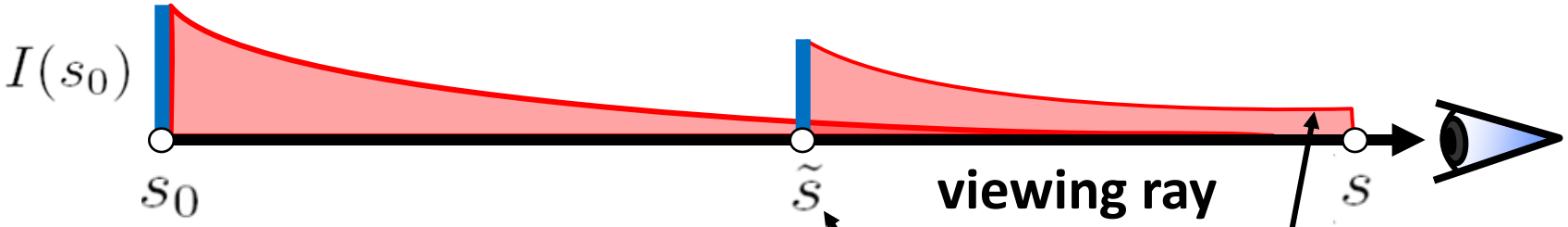
$$I(s) = I(s_0)\, e^{-\tau(s_0,s)}$$

Extinction τ
Absorption κ

Without absorption all the initial radiant energy would reach the point s.

$$\tau(s_1, s_2) = \int_{s_1}^{s_2} \kappa(s)\, ds.$$

Ray Integration (3)



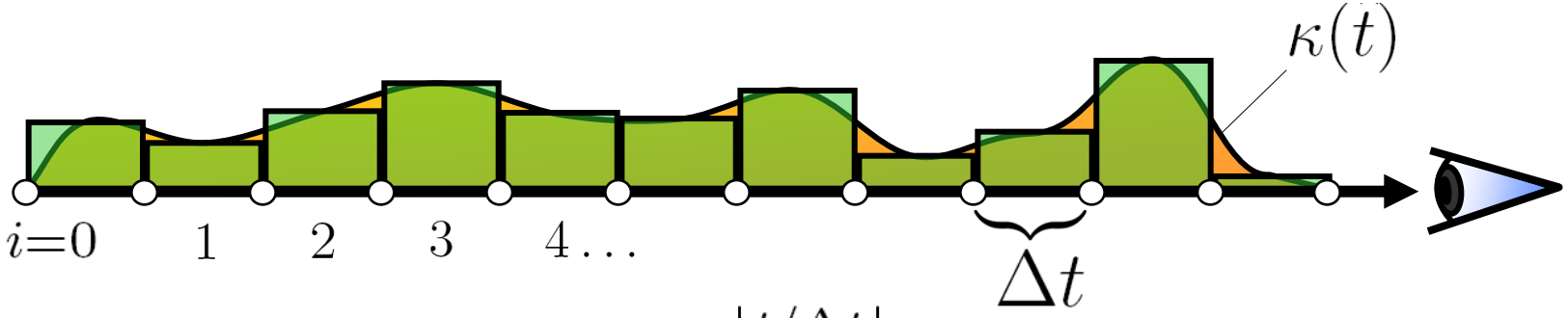
Every point $\tilde{s}$ along the viewing ray emits additional radiant energy

Active emission at point $\tilde{s}$

Absorption along the distance $s - \tilde{s}$

$$I(s) = I(s_0)\, e^{-\tau(s_0,s)} + \int_{s_0}^{s} q(\tilde{s})\, e^{-\tau(\tilde{s},s)}\, d\tilde{s}$$

Numerical Solution (1)



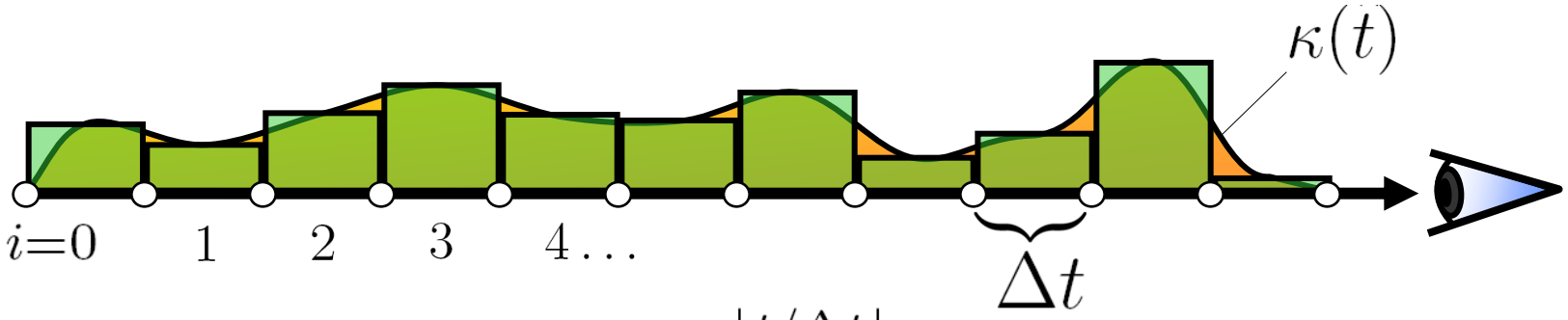
Extinction $\approx \quad \tilde{\tau}(0,t) \approx \sum_{i=0}^{\lfloor t/\Delta t \rfloor} \kappa(i \cdot \Delta t)\, \Delta t$

Approximate integral by Riemann sum

$$e^{-\tilde{\tau}(0,t)} = e^{-\sum_{i=0}^{\lfloor t/\Delta t \rfloor} \kappa(i \cdot \Delta t)\, \Delta t}$$

$$\tau(0,t) \approx \sum_{i=0}^{\lfloor t/\Delta t \rfloor} \kappa(i \cdot \Delta t)\, \Delta t$$

Numerical Solution (2)



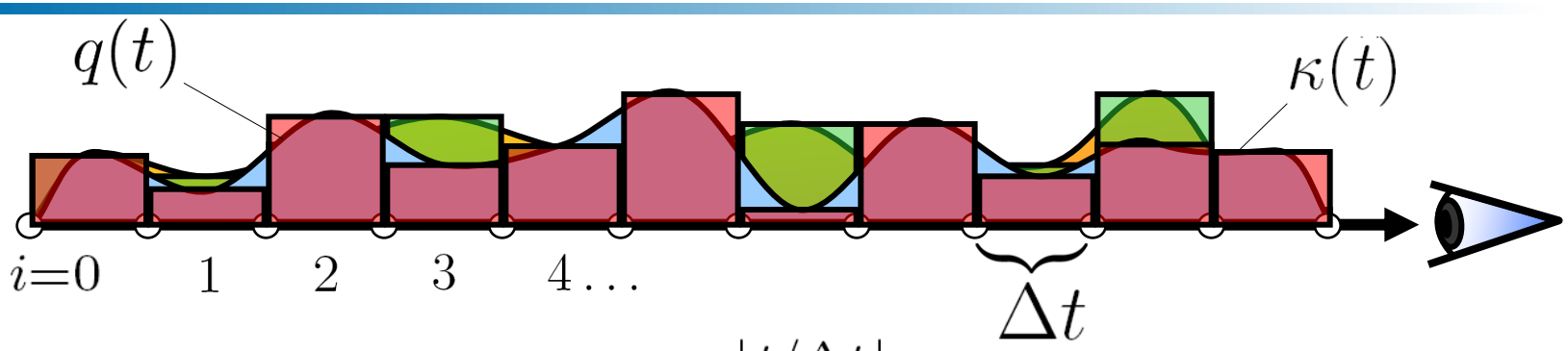
$$\tau(0,t) \approx \tilde{\tau}(0,t) = \sum_{i=0}^{\lfloor t/\Delta t \rfloor} \kappa(i \cdot \Delta t)\, \Delta t$$

$$e^{-\tilde{\tau}(0,t)} = \prod_{i=0}^{\lfloor t/\Delta t \rfloor} e^{-\kappa(i \cdot \Delta t)\, \Delta t}$$

Now we introduce opacity

$$(1 - A_i) = 1 - e^{-\kappa(i \cdot \Delta t)\, \Delta t}$$

Numerical Solution (3)



$$\tau(0,t) \approx \tilde{\tau}(0,t) = \sum_{i=0}^{\lfloor t/\Delta t \rfloor} \kappa(i \cdot \Delta t)\, \Delta t$$

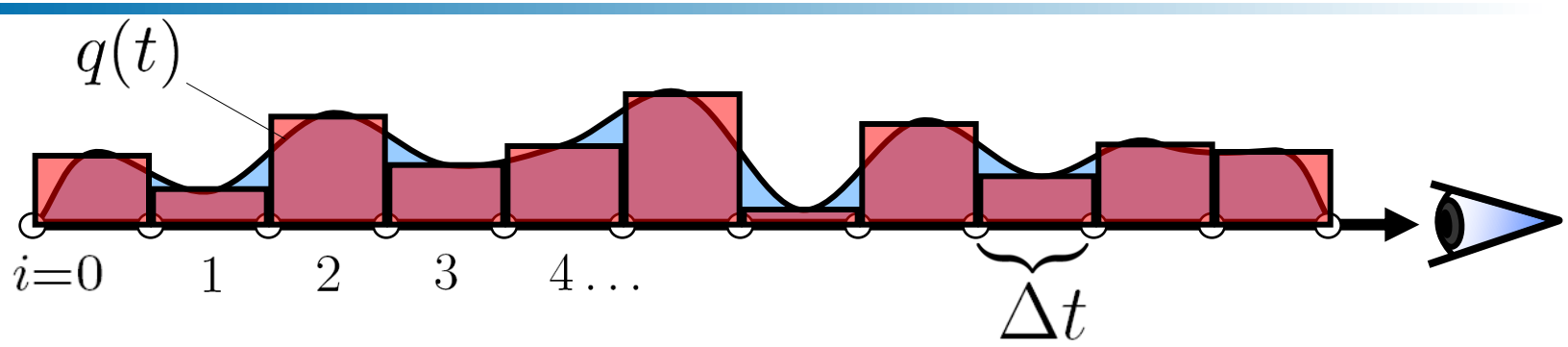
$$q(t) \approx e^{-\tilde{\tau}(0,t)} = \prod_{i=0}^{\lfloor t/\Delta t \rfloor} (1 - A_i)$$

Now we introduce opacity

$$C = \sum_{i=0}^{\lfloor T/\Delta t \rfloor} C_i \, e^{-\tilde{\tau}(0,t)}$$

$$1 - A_i = e^{-\kappa(i \cdot \Delta t)\, \Delta t}$$

Numerical Solution (4)



$$e^{-\tilde{\tau}(0,t)} = \prod_{i=0}^{\lfloor t/\Delta t \rfloor} (1 - A_i)$$

Can be computed recursively

$$q(t) \approx C_i = c(i \cdot \Delta t)\,\Delta t$$

$$C'_i = (1 - A_{i-1})C'_{i-1}$$

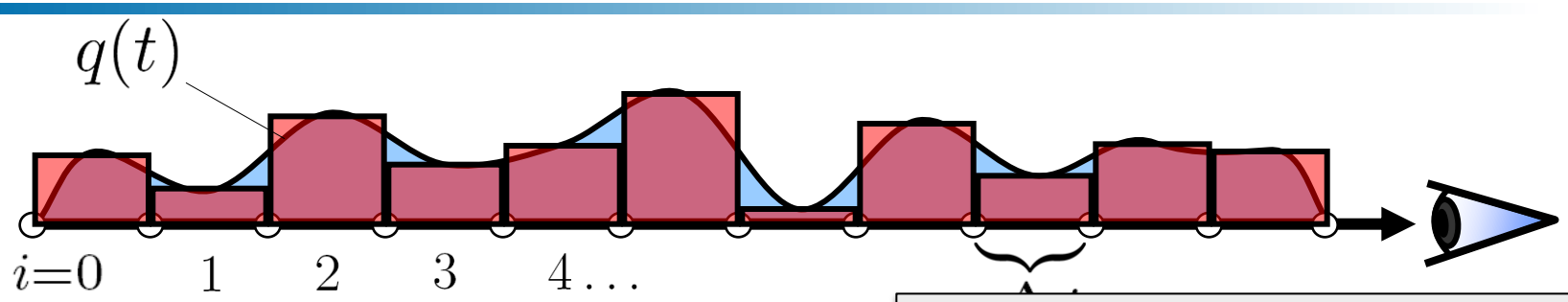
radiant energy observed at position i

radiant energy emitted at position i

absorption at position i

radiant energy observed at position $i-1$

Numerical Solution (5)



Back-to-front compositing

$$\tilde{C} = \sum_{i=0}^{\lfloor T/\Delta t \rfloor} C_i$$

Can be computed recursively

Front-to-back compositing

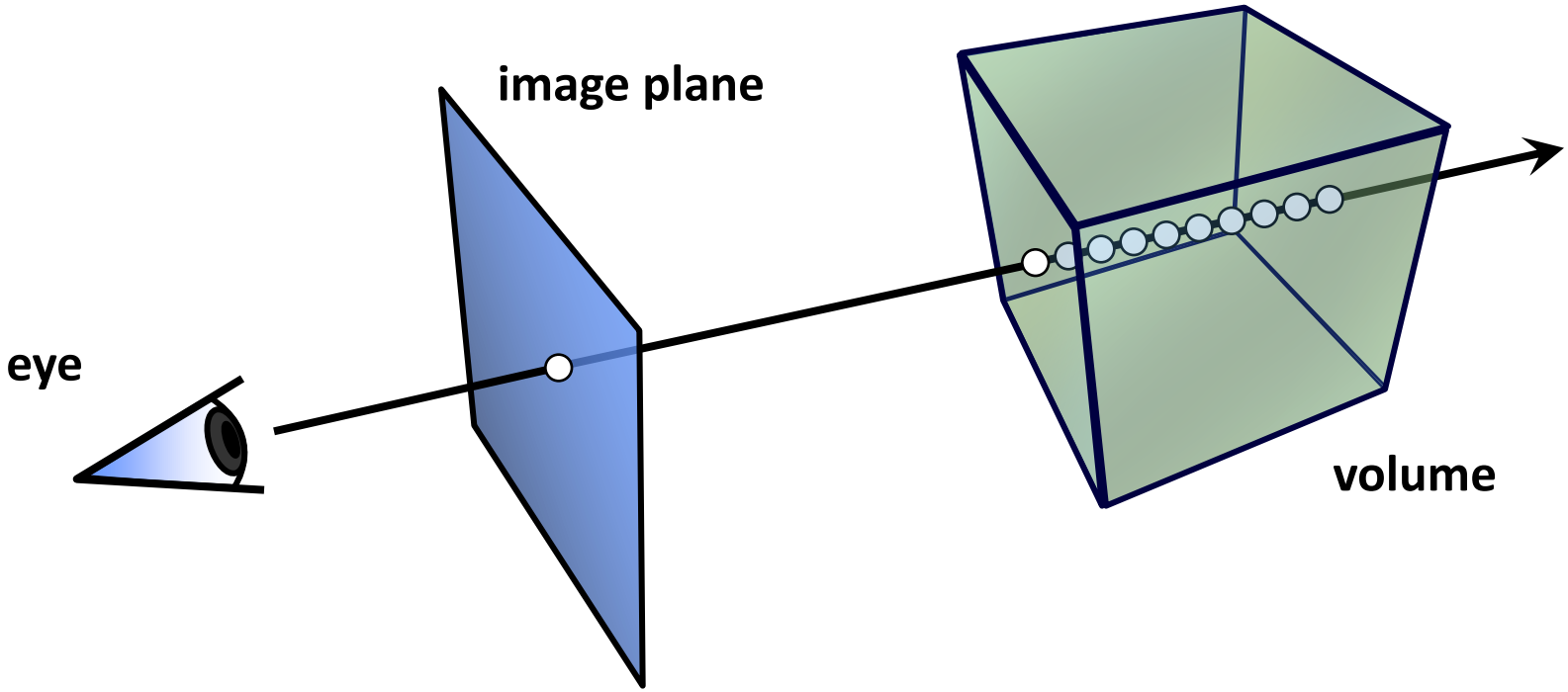
$$C'_i = C'_{i+1} + (1 - A'_{i+1}) C_i$$
$$A'_i = A'_{i+1} + (1 - A'_{i+1}) A_i$$

Early Ray Termination

Stop the calculation when

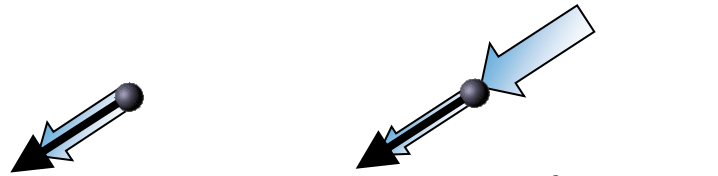
$$A'_i \approx 1$$

Numerical Solution (6)



Summary

Emission Absorption Model



$$I(s) = I(s_0)\, e^{-\tau(s_0,s)} + \int_{s_0}^{s} q(\tilde{s})\, e^{-\tau(\tilde{s},s)}\, \mathrm{d}\tilde{s}$$

Back-to-front iteration

$$C'_i = C_i + (1 - A_i)C'_{i-1}$$

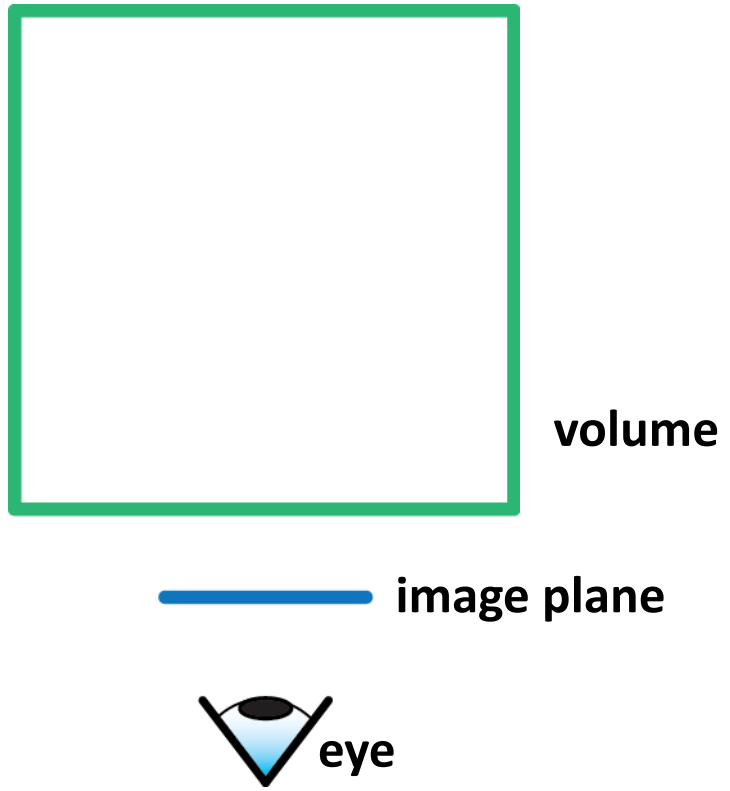
Front-to-back iteration

$$C'_i = C'_{i+1} + (1 - A'_{i+1})C_i$$
$$A'_i = A'_{i+1} + (1 - A'_{i+1})A_i$$

GPU Volume Rendering

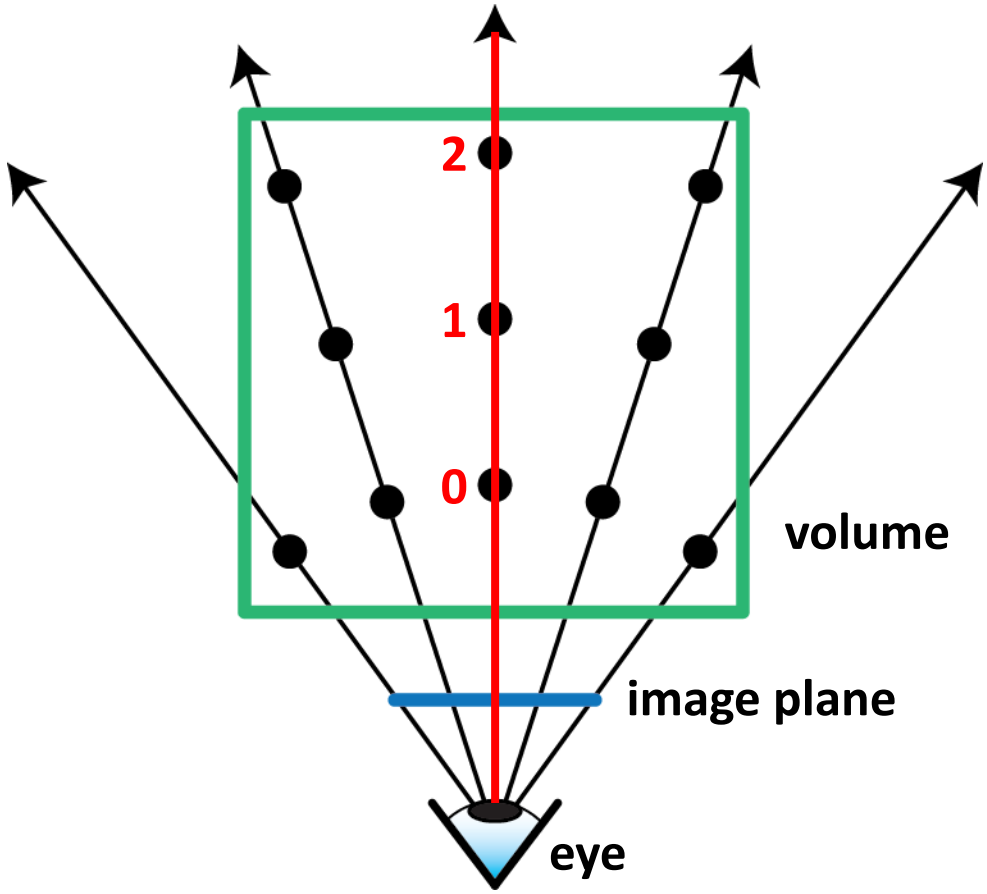
Today, standard ray casting can be implemented on the GPU

On previous hardware generations, only slicing was possible



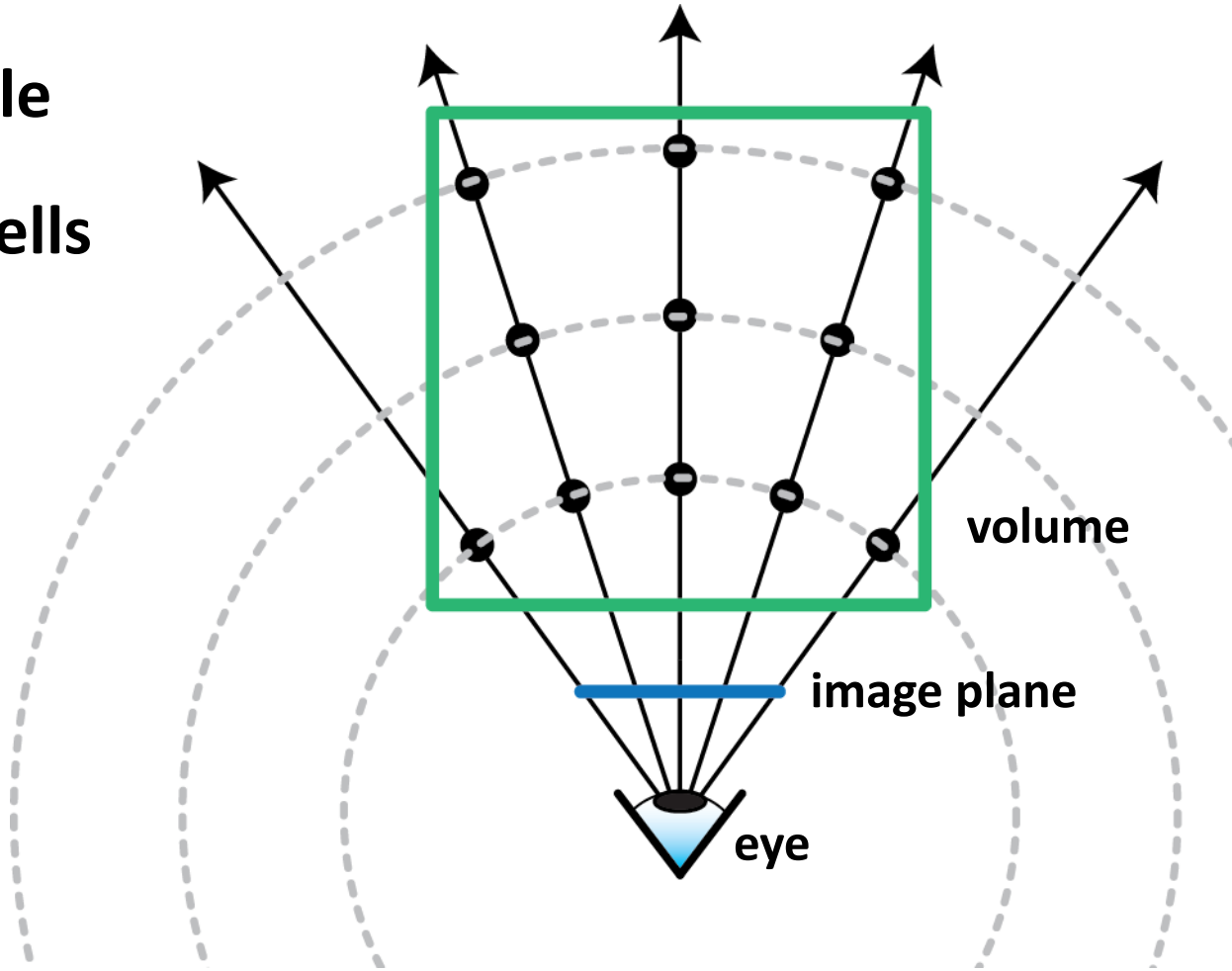
GPU Volume Rendering

In raycasting, sampling is typically performed at equidistant points along each ray



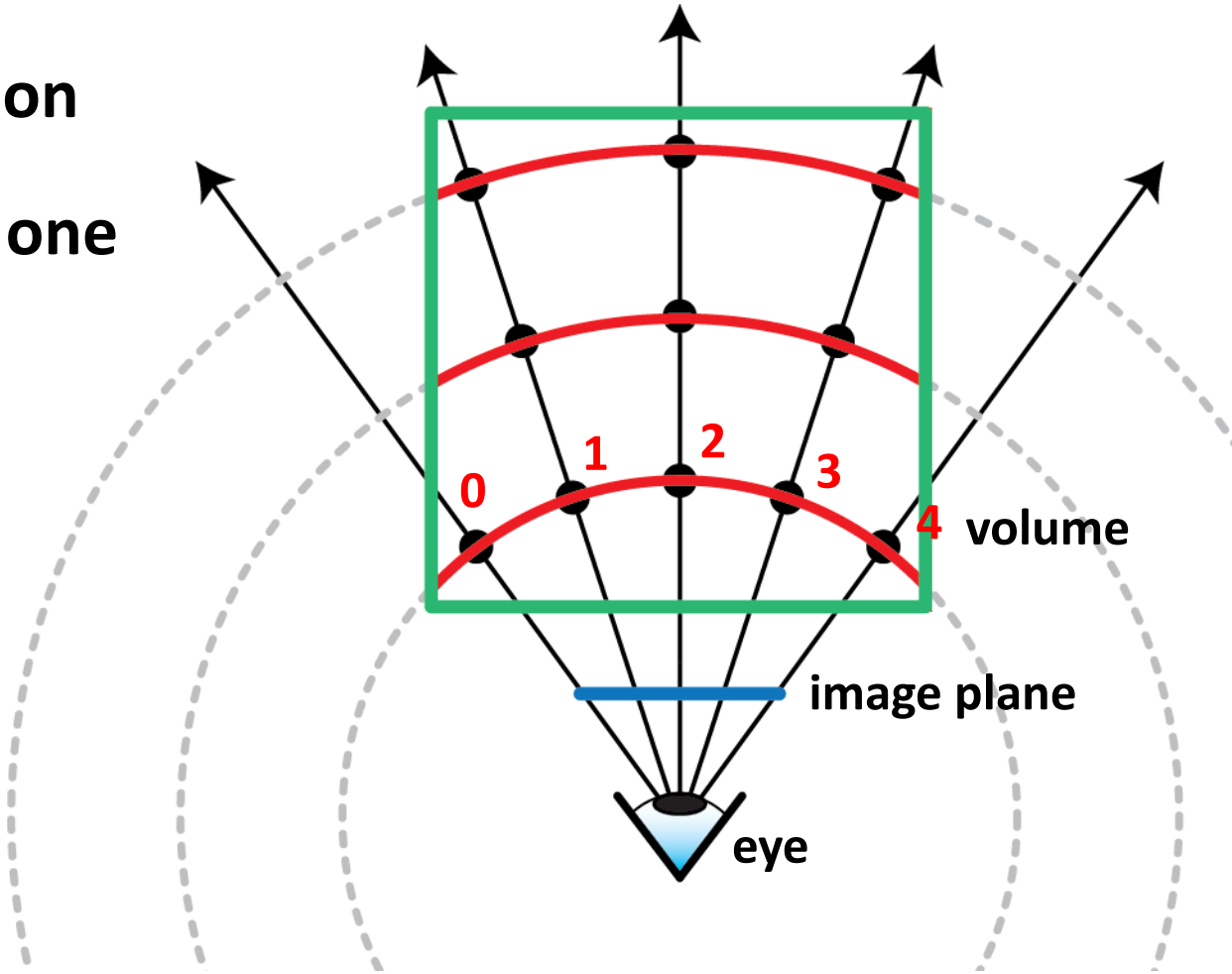
GPU Volume Rendering

Under perspective projection, these sample points are located on concentric spherical shells



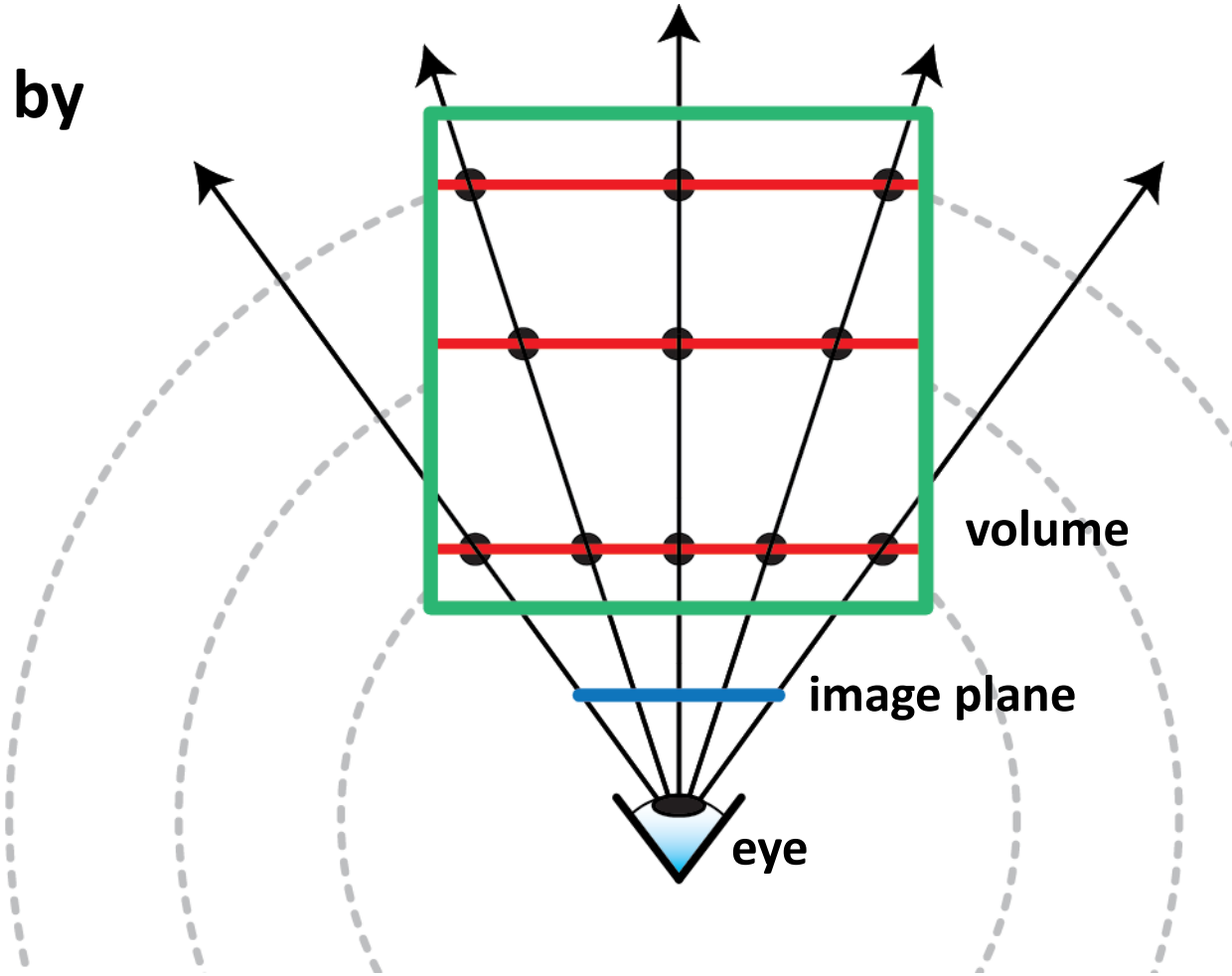
GPU Volume Rendering

This is equivalent to sampling all the points on one shell before proceeding to the next one



GPU Volume Rendering

In slicing, the shells are typically approximated by view-aligned planes



Raycasting vs. Slicing

Slicing

- May have advantages in terms of memory access pattern
- Sampling pattern can cause artifacts

Raycasting

- Easier to implement on current hardware
- Early ray termination is trivial

GPGPU

General Purpose Computations on GPUs

- Focus on data-parallel algorithms

Legacy GPGPU uses graphics APIs

- OpenGL, Direct3D

http://www.gpgpu.org

Current GPGPU

- CUDA, OpenCL, AMD Stream

Large number of publications

Lots of examples on NVIDIA webpage

Shading Languages

Old assembly language (OpenGL, Direct3D)

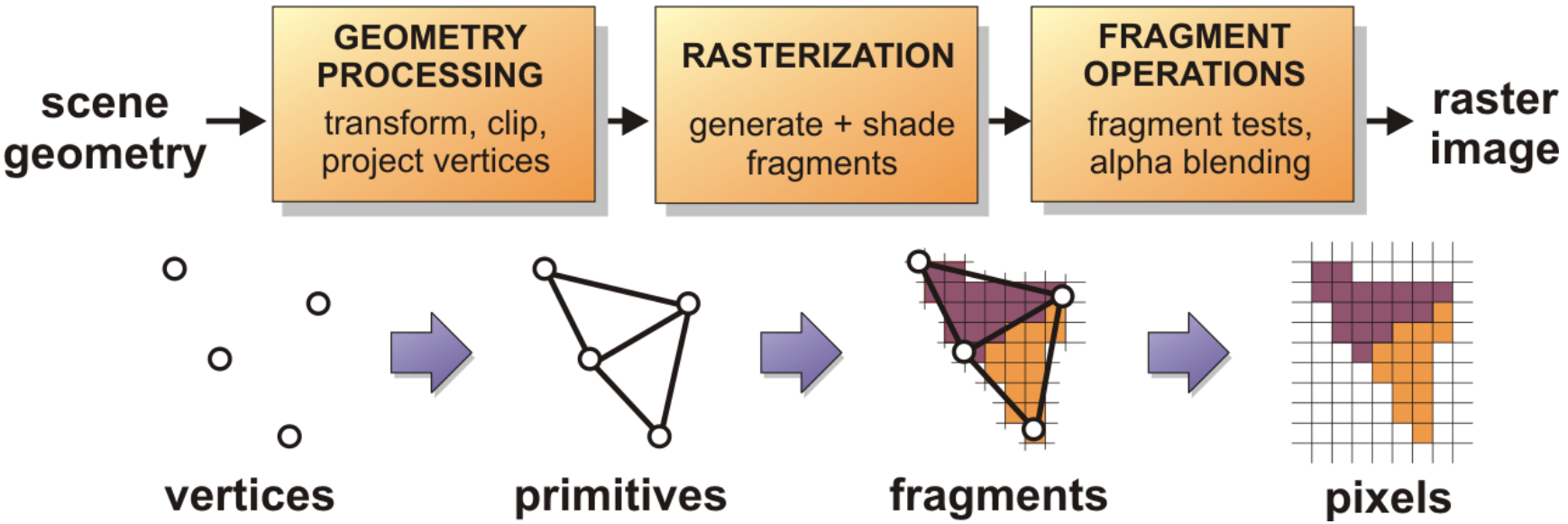
High-level C-like shading languages

- NVIDIA Cg
- DirectX HLSL
- OpenGL shading language (GLSL)

Combination of

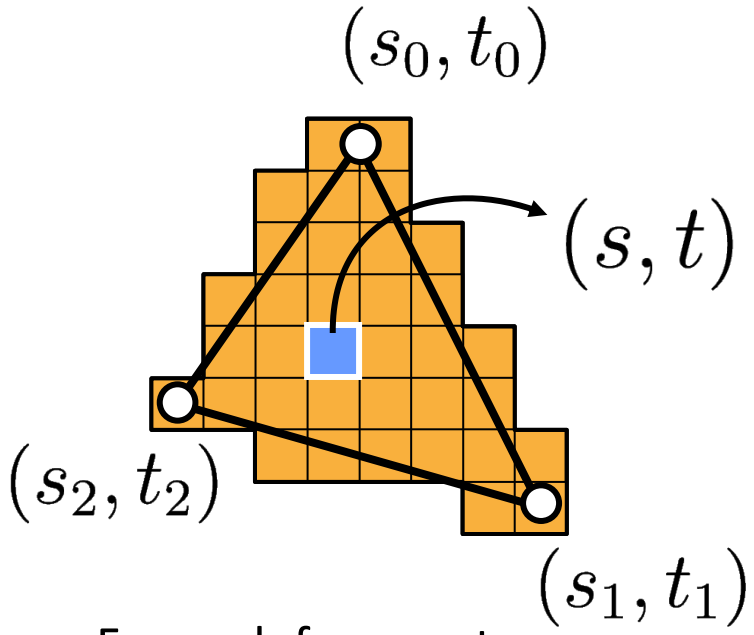
- Fragment shaders
- Vertex shaders
- Geometry shaders

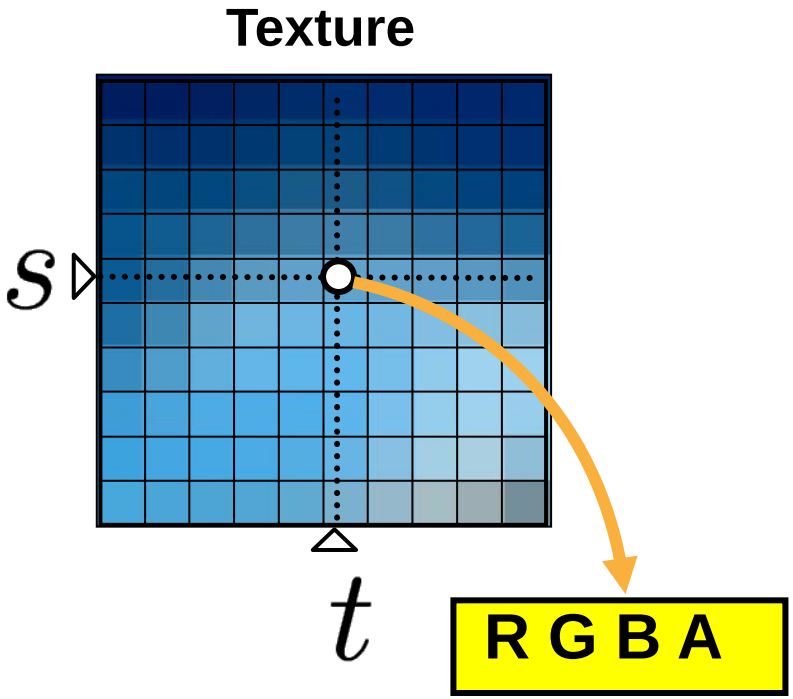
Legacy GPU Pipeline



- Current hardware: high-level perspective still similar, but now very programmable (not fully)
- Still some fixed-function elements (projection, blending, depth test, ...)

2D Texture Mapping



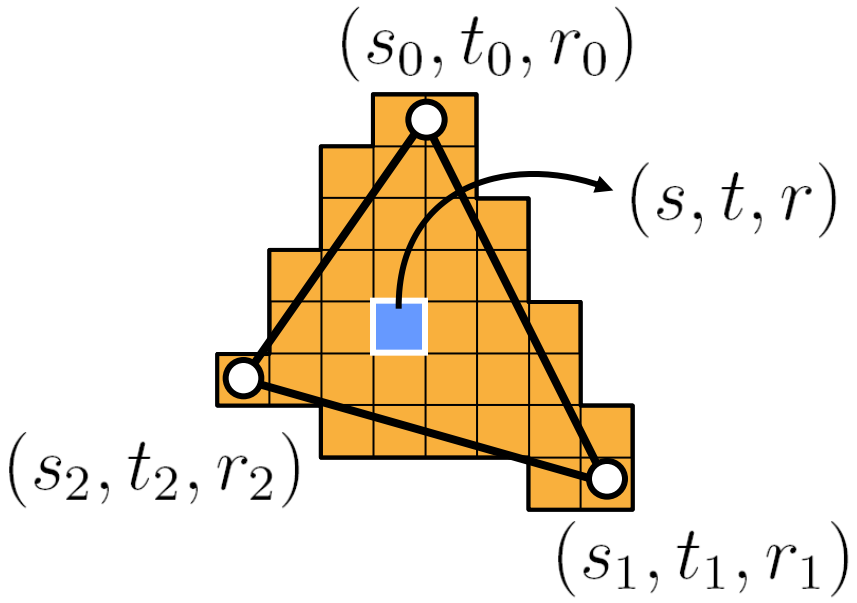


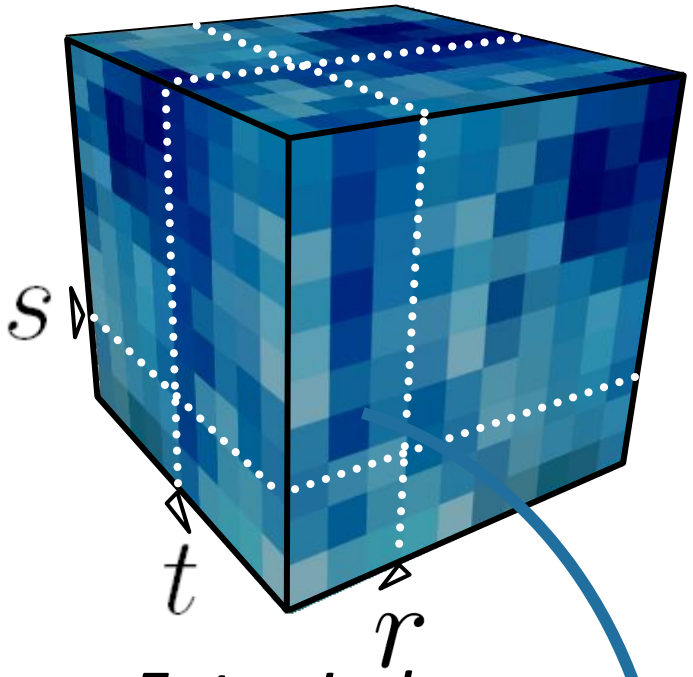
For each fragment:
interpolate the
texture coordinates
(barycentric)
Or:
Use arbitrary, computed coordinates

Texture-Lookup:
interpolate the
texture data
(bi-linear)
Or:
Nearest-neighbor for "array lookup"

Courtesy Christof Rezk-Salama

3D Texture Mapping



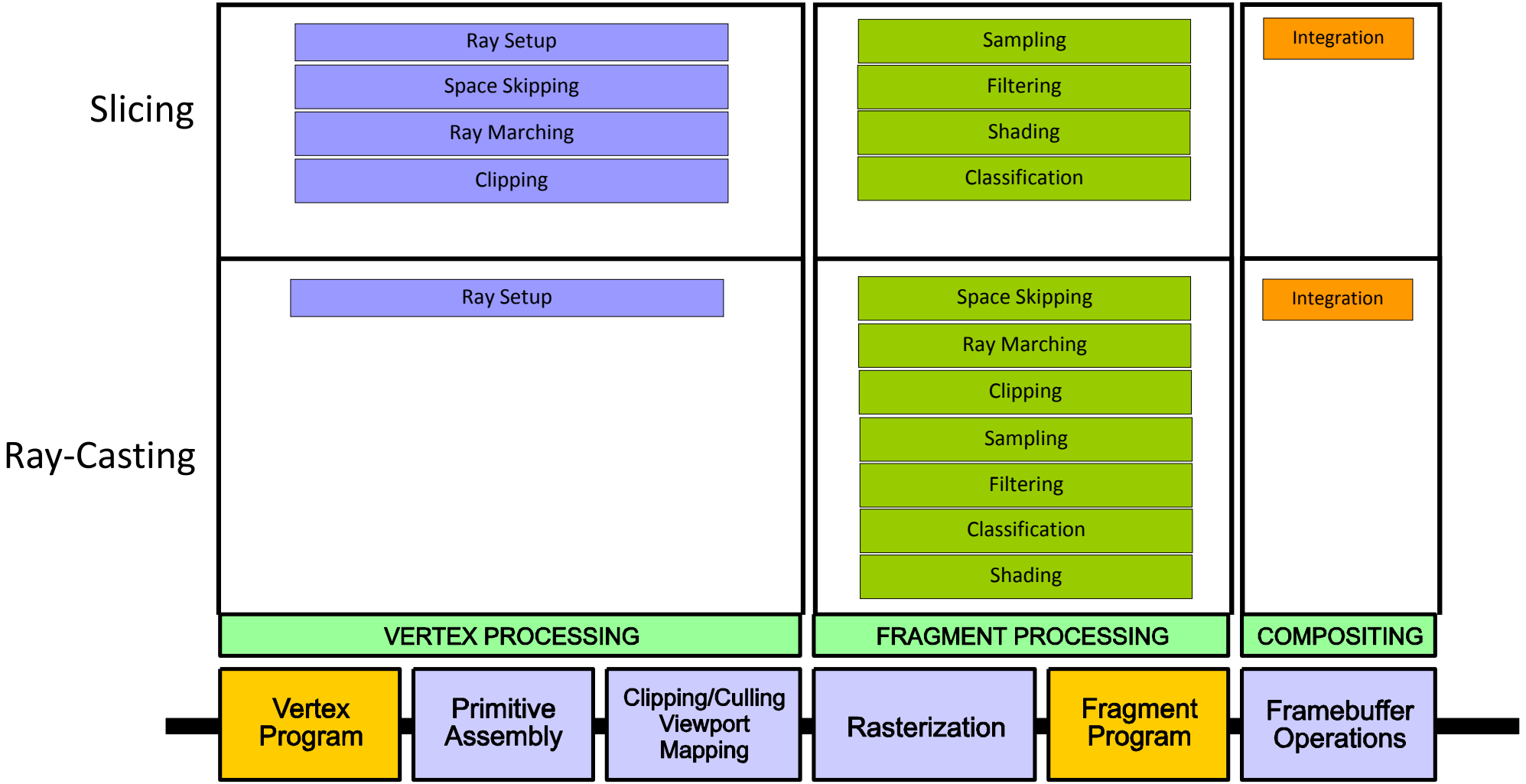


For each fragment:
interpolate the
texture coordinates
(barycentric)
Or:
Use arbitrary, computed coordinates

Texture-Lookup:
interpolate the
texture data
(tri-linear)
Or:
Nearest-neighbor for "array lookup"

Courtesy Christof Rezk-Salama

Volume Rendering GPU Pipeline Load



Courtesy Klaus Engel

GPGPU Hardware

NVIDIA Tesla

Fermi architecture

448 CUDA Cores

Over 1 Teraflop / device

Up to 6GB RAM / device

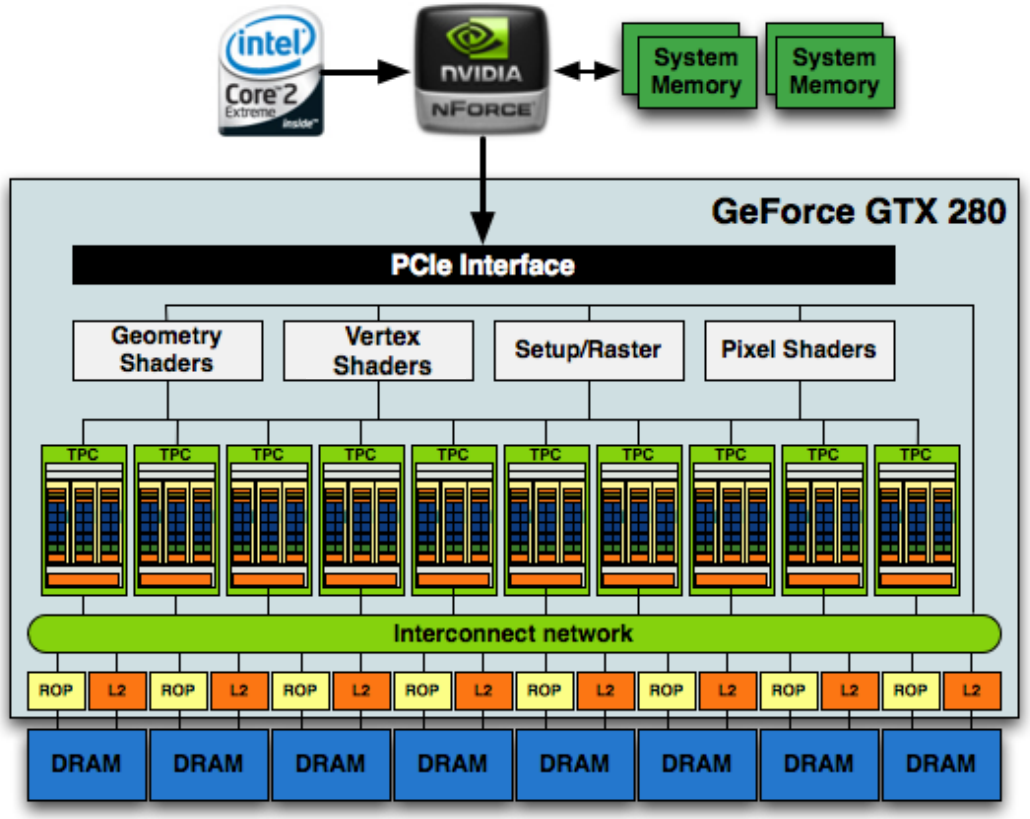
Multiple devices per node / machine

Tesla C2070

G80/GT200 Architecture

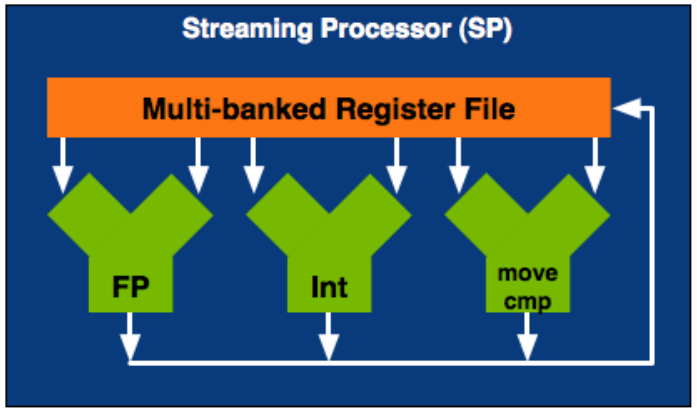
Streaming processors can execute

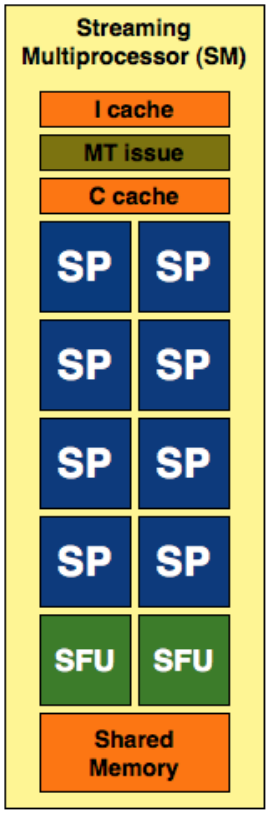
- Geometry shaders
- Vertex shaders
- Fragment shaders
- CUDA kernels

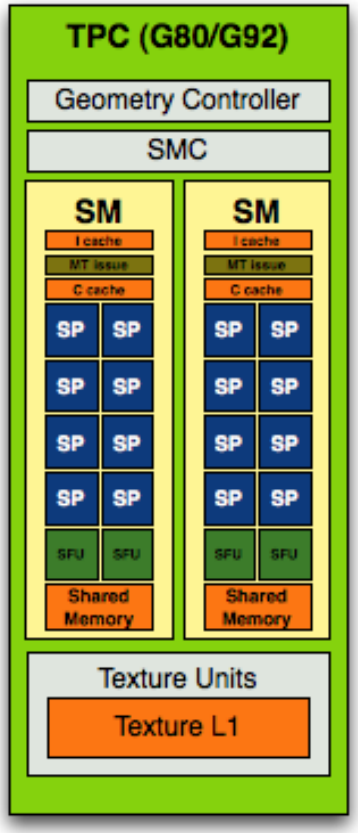


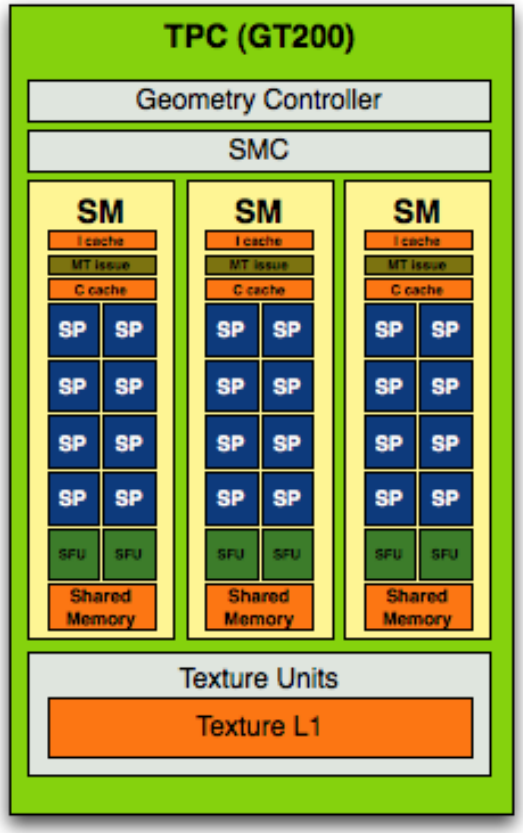
Courtesy AnandTech

G80/GT200 Architecture









Courtesy AnandTech

Streaming Processor (SP)

Streaming Multiprocessor (SM)

Texture/Processor Cluster (TPC)

Fermi Architecture

Third generation streaming multiprocessor (SM)

- 32 CUDA cores per SM, 4x over GT200
- 8x the peak double precision floating point performance over GT200
- Dual Warp Scheduler simultaneously schedules and dispatches instructions from two independent warps

Faster context switching, concurrent kernel execution, out of order thread block execution, …

NVIDIA CUDA (1)

Data-parallel programming interface

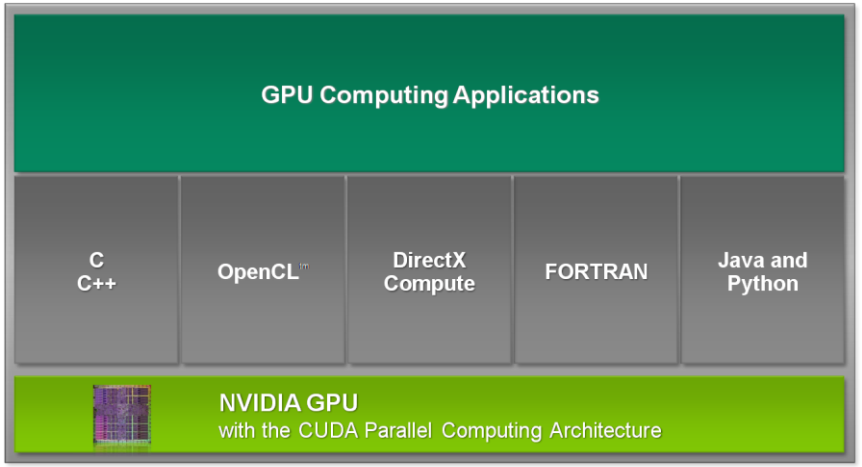
- Data to be operated on is discretized into independent partition of memory
- Each thread performs roughly same computation to different partition of data
- When appropriate, easy to express and very efficient parallelization

Programmer expresses

- Functions to be launched on GPU, and how to launch
- Data organization and movement between host and GPU
- Synchronization, memory management, testing, …

NVIDIA CUDA (2)

“Compute Unified Device Architecture”



CUDA C/C++

- Compile `.cu` files with NVCC
- Uses general C compiler (Visual C, gcc, ...)
- Link with CUDA run-time (`cudart.lib`) and cuda core (`cuda.lib`)

CUDA Advantages

Random access byte-addressable memory

- Any memory location can be accessed

Unlimited access to memory

- As many read/write accesses as needed possible

Shared memory and synchronization

- Cooperation between threads

Low learning curve

- Just a few C extensions, no graphics knowledge required

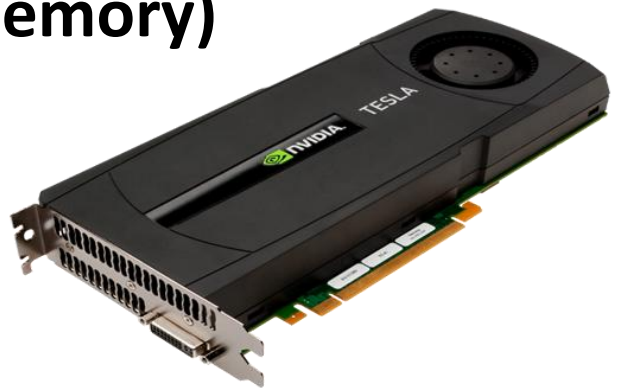
No graphics API overhead

Terminology

Host – the CPU and its memory (host memory)

- Host pointers point to CPU memory
  - May be passed to and from device code
  - May not be dereferenced from device code

Device – the GPU and its memory (device memory)

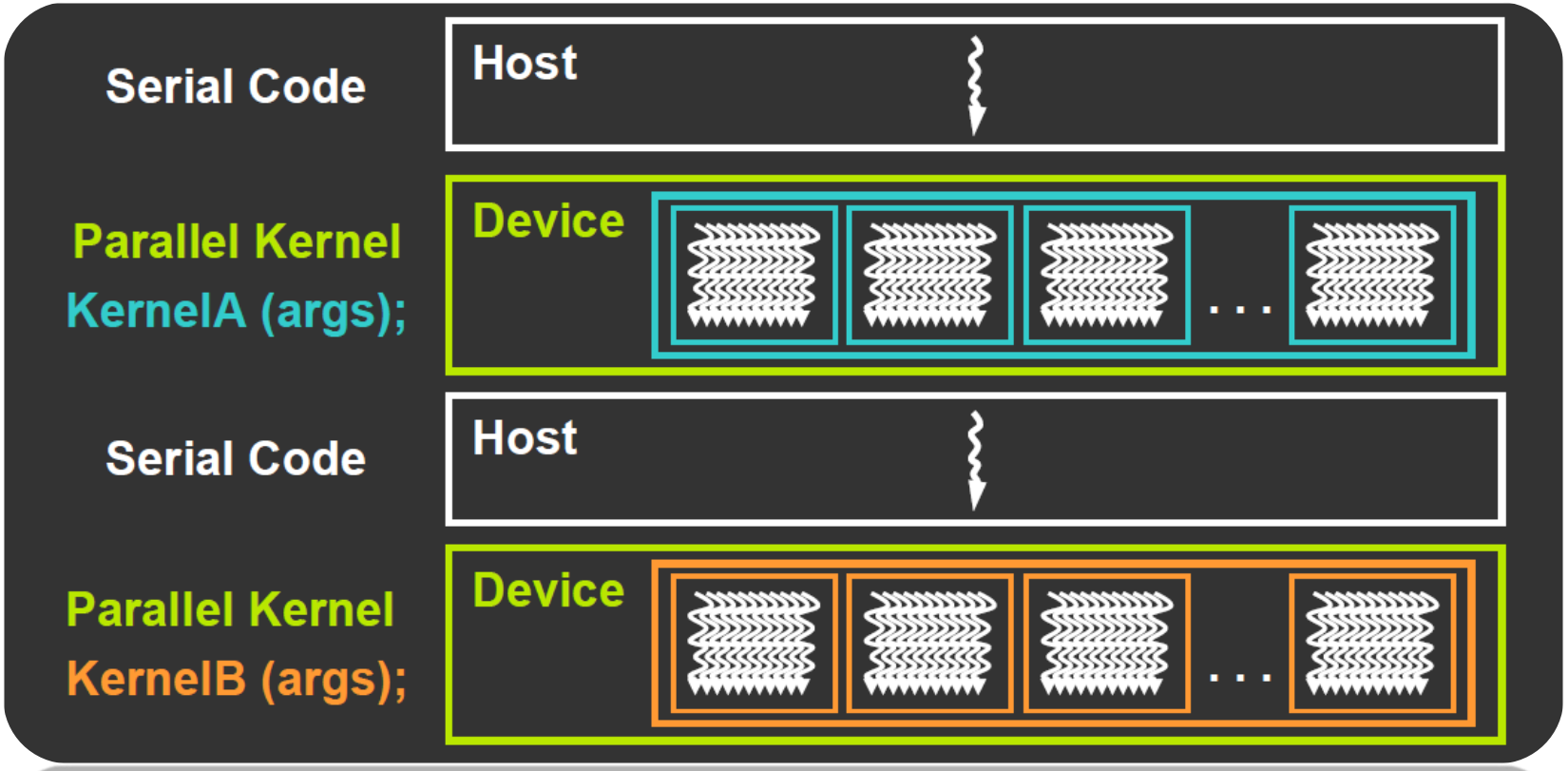


- Device pointers point to GPU memory
  - May be passed to and from host code
  - May not be dereferenced from host code

Kernels

A kernel is a function executed on the device in parallel

- Serial code executes on the host, parallel code runs on the device

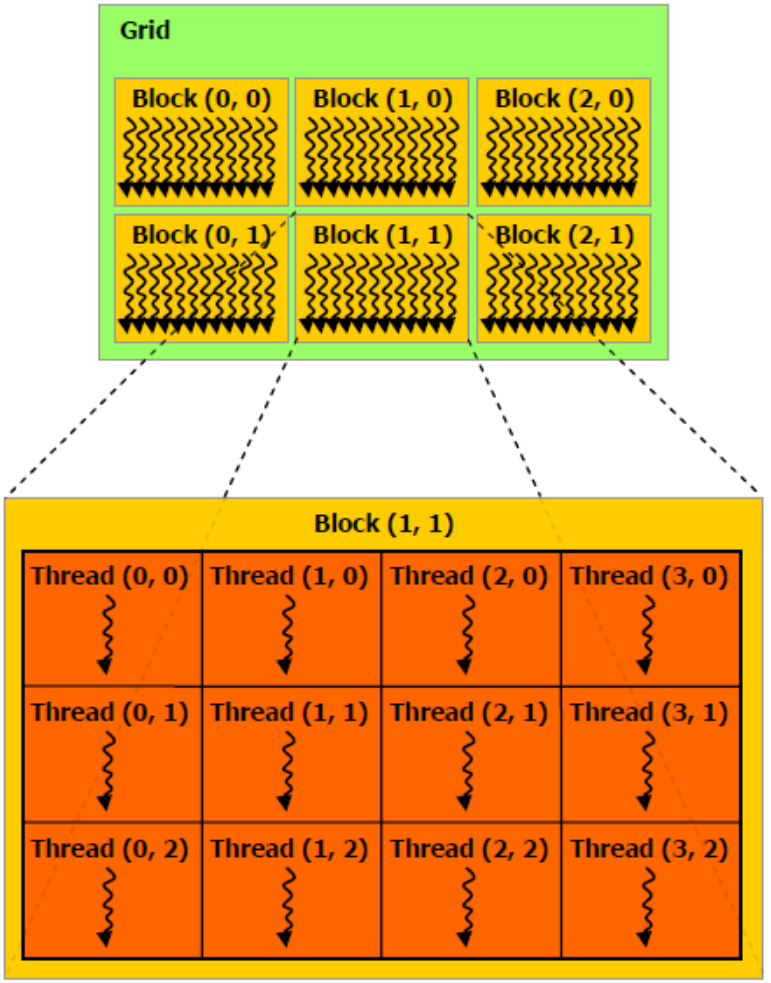


Blocks

Kernels are executed by blocks of threads arranged in a grid of blocks

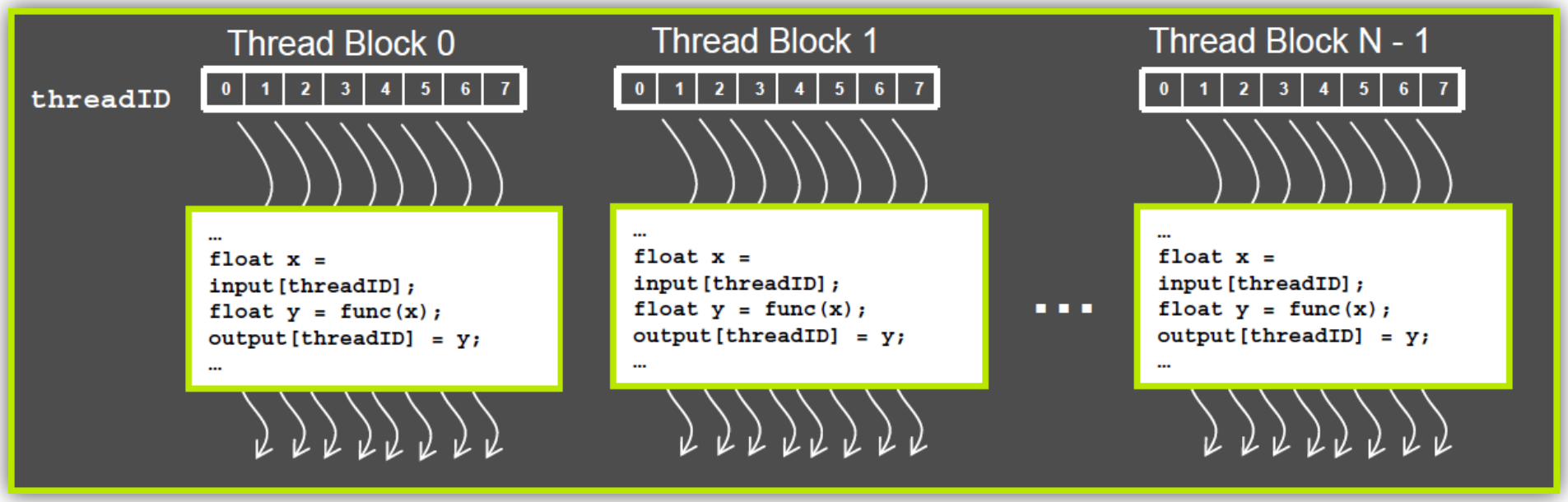
- Used to partition data into smaller units by defining a grid
- Kernel refers to its block index to determine its position in the grid

Blocks allow to partition a problem into multiple sub-problems to be solved in parallel



Threads (1)

A block is split into parallel threads, each thread executes the same code



Threads (2)

Thread synchronization (within the same block) via barriers

- Threads wait at the barrier until all threads in the same block reach it

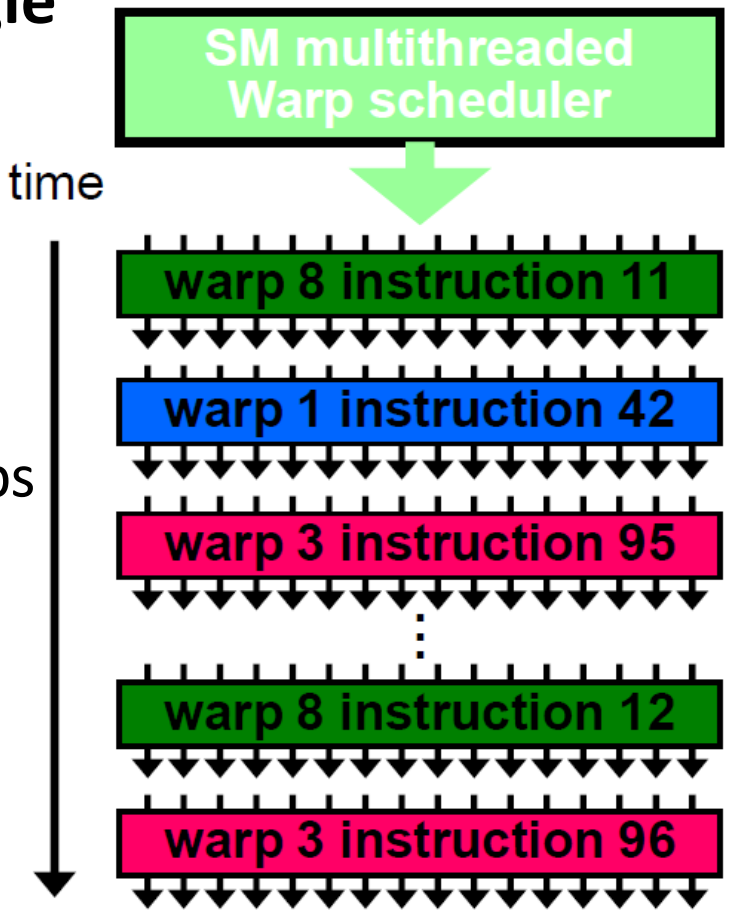
Thread communication via atomic operations

- Shared memory
- Global memory

Warps

Execution based on SIMT model (single instruction, multiple thread); akin to SIMD (single instruction, multiple data)

- Thread-level parallelism for maximum hardware utilization (latency hiding)
- Thread blocks are partitioned into warps
- Warps are the basic scheduling units
- Warps always perform the same instruction
- Warp size is 32 threads on current hardware



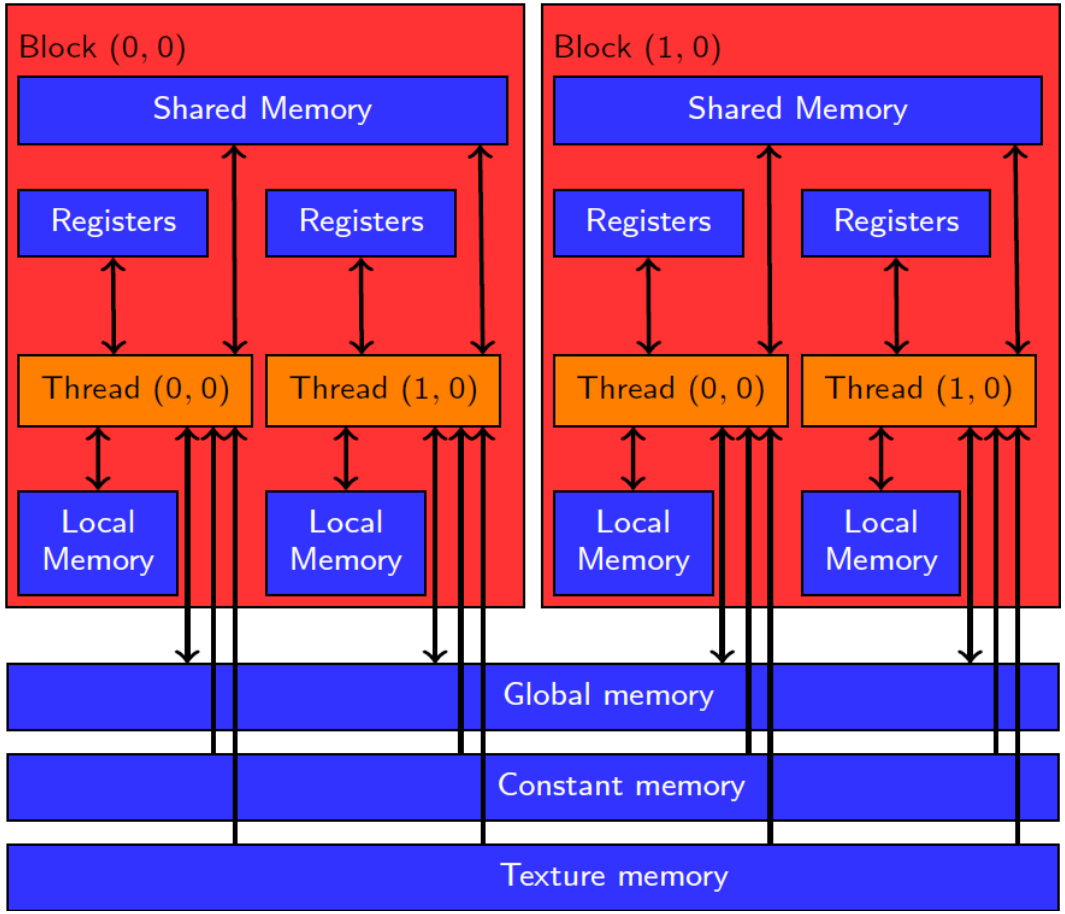
Memory (1)

Host memory

- Non-page locked memory
- Page-locked memory

Device memory

- Local memory
- Shared memory
- Global memory
- Constant memory
- Texture memory



Memory (2)

Registers	Per thread	Read-Write	
Local memory	Per thread	Read-Write	
Shared memory	Per block	Read-Write	For sharing data within a block
Global memory	Per grid	Read-Write	Not cached
Constant memory	Per grid	Read-only	Cached
Texture memory	Per grid	Read-only	Spatially cached

Texture memory can be helpful for moving leagcy GPGPU algorithms to CUDA

Combination of global & shared memory allows fine-grained manual cache control

Memory (3)

Shared memory: memory shared between a block of threads

- Very fast on-chip memory
- Basically a user-managed cache
- Declared with the __shared__ keyword
- Not visible to threads in other blocks

Efficient use of shared memory can be crucial for performance

Streams

Asynchronous API for kernel invocation and memory transfers

- A stream is a sequence of operations that execute in order
- Multiple streams can execute in parallel
- Synchronization via stream queries and events

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, stream1);
kernel<<<grid, block, 0, stream2>>>(…);
cudaStreamQuery(stream2);
```

overlapped

Courtesy Simon Green

Kernel Invocation

Modified C function call syntax

- kernel<<<dim3 grid, dim3 block, int smem, int stream>>>(…)

Execution Configuration (“<<< >>>”)

- Grid dimensions
- Block dimensions
- Size of shared memory
- Stream ID

Built-In Device Variables

dim3 gridDim;

- Dimensions of the grid in blocks

dim3 blockDim;

- Dimensions of the block in threads

dim3 blockIdx;

- Block index within the grid

dim3 threadIdx;

- Thread index within the block

Example: Matrix Addition

Courtesy Simon Green

CPU C program

```
void addMatrixC(float *a, float *b,
                float *c, int N)
{
  int i, j, index;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      index = i + j * N;
      c[index]=a[index] + b[index];
    }
  }
}

void main()
{
  .....
  addMatrixC(a,b,c,N);
}
```

CUDA C program

```
__global__
void addMatrixG(float *a, float *b,
                float *c, int N)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  int j=blockIdx.y*blockDim.y+threadIdx.y;
  int index = i + j * N;
  if ( i < N && j < N)
    c[index]= a[index] + b[index];
}


void main()
{
  .....
  dim3 dimBlk (16,16);
  dim3 dimGrd (N/dimBlk.x,N/dimBlk.y);
  addMatrixG<<<dimGrd,dimBlk>>>(a,b,c,N);
}
```

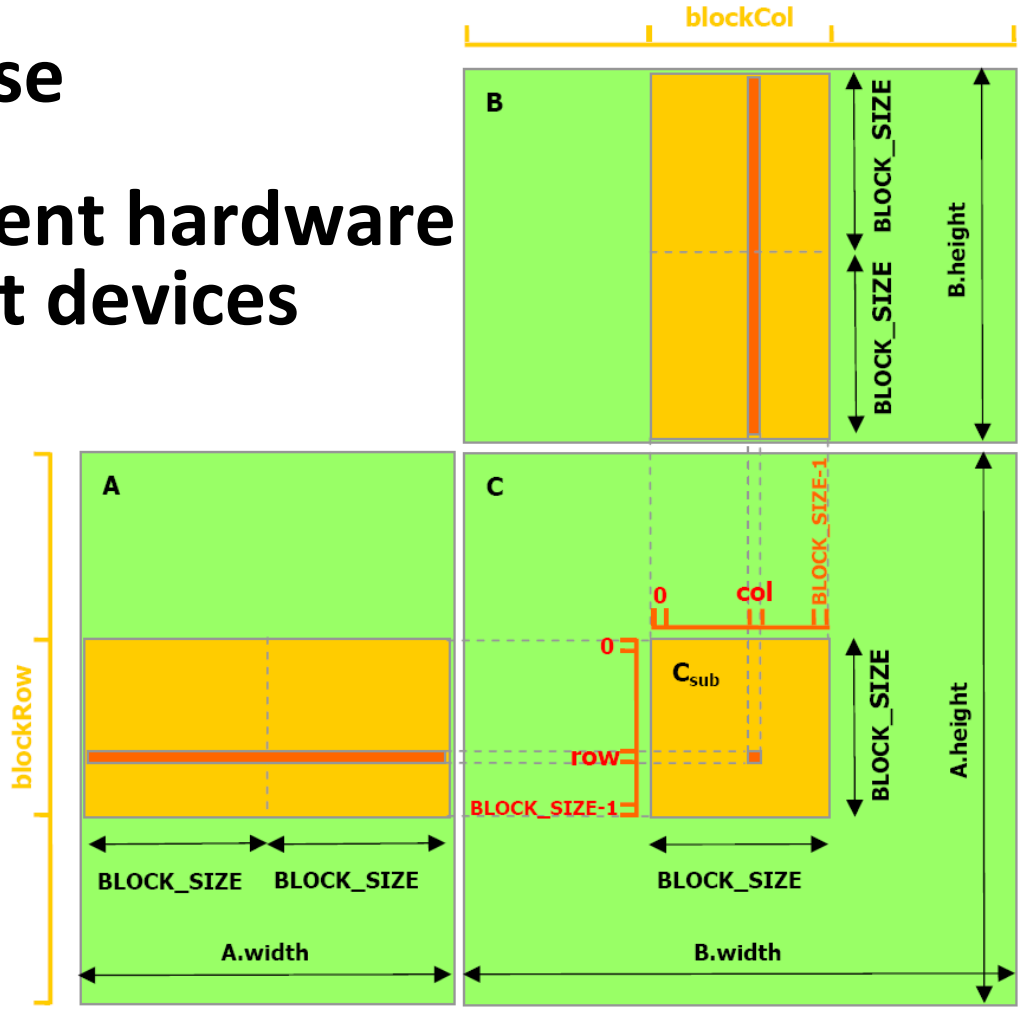
Example: Matrix Multiplication (1)

Multiply matrix block-wise

Set BLOCK_SIZE for efficient hardware use, e.g., to 16 on current devices

Maximize parallelism

- Launch as many threads per block as block elements
- Each thread fetches one element of block
- Perform row * column dot products in parallel



Example: Matrix Multiplication (2)

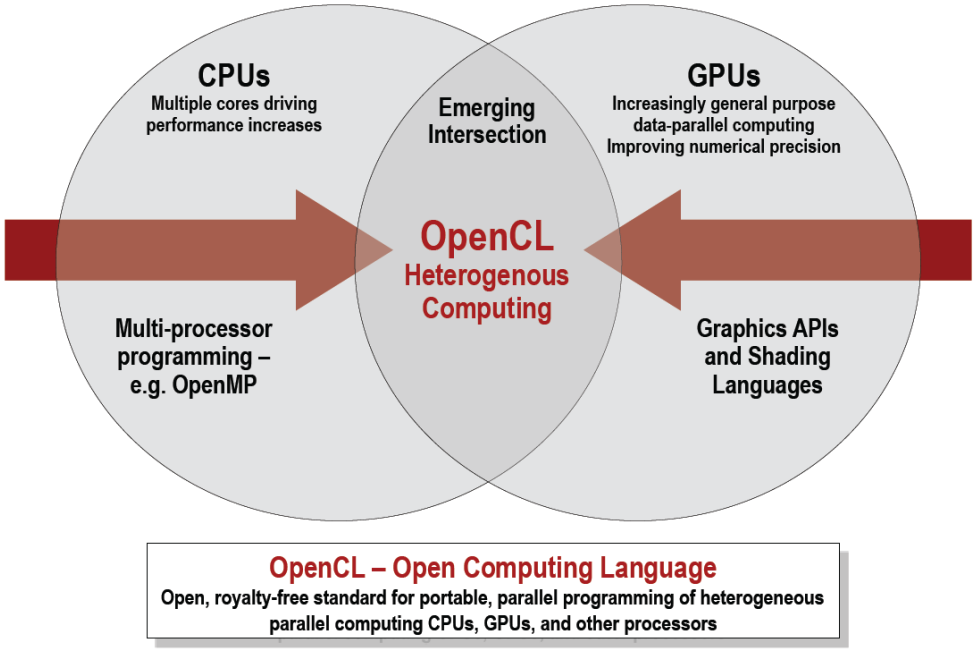
```
__global__ void MatrixMul( float *matA, float *matB, float *matC, int w )

{
    __shared__ float blockA[ BLOCK_SIZE ][ BLOCK_SIZE ];
    __shared__ float blockB[ BLOCK_SIZE ][ BLOCK_SIZE ];

    int bx = blockIdx.x; int tx = threadIdx.x;
    int by = blockIdx.y; int ty = threadIdx.y;

    int col = bx * BLOCK_SIZE + tx;
    int row = by * BLOCK_SIZE + ty;

    float out = 0.0f;
    for ( int m = 0; m < w / BLOCK_SIZE; m++ ) {

        blockA[ ty ][ tx ] = matA[ row * w +   m * BLOCK_SIZE + tx       ];
        blockB[ ty ][ tx ] = matB[ col     + ( m * BLOCK_SIZE + ty ) * w ];
        __syncthreads();

        for ( int k = 0; k < BLOCK_SIZE; k++ ) {
            out += blockA[ ty ][ k ] * blockB[ k ][ tx ];
        }
        __syncthreads();
    }

    matC[ row * w + col ] = out;

}
```

OpenCL (1)

Current Specification: OpenCL 1.1 (June 2010)

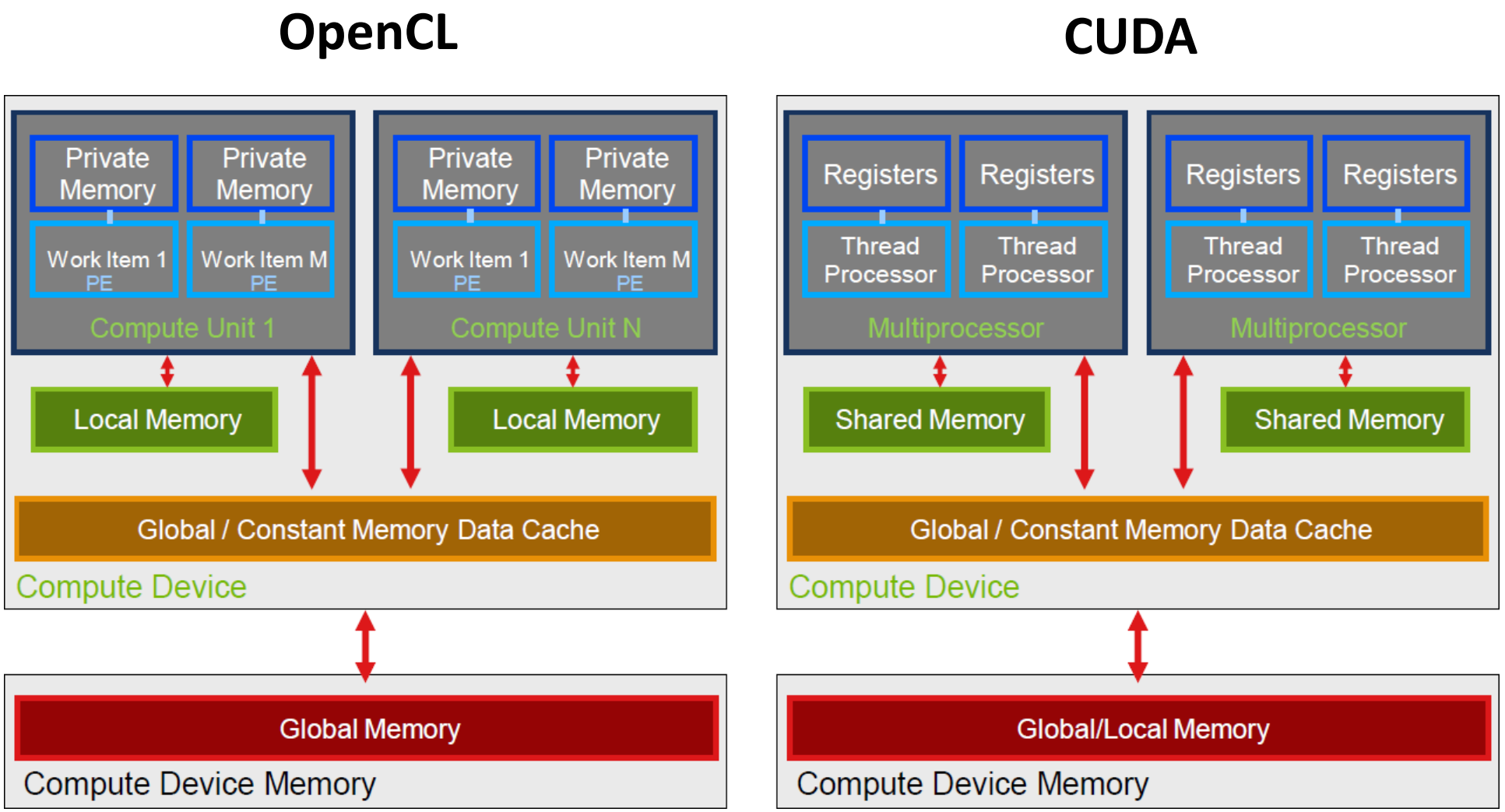
Based on the same concepts as CUDA

Portable, maps to CUDA on NVIDIA hardware



Courtesy Khronos Group

OpenCL (2)


OpenCL
Private Memory
Private Memory
Work Item 1
PE
Work Item M
PE
Compute Unit 1
Private Memory
Private Memory
Work Item 1
PE
Work Item M
PE
Compute Unit N
Local Memory
Local Memory
Global / Constant Memory Data Cache
Compute Device
Global Memory
Compute Device Memory
CUDA
Registers
Registers
Thread Processor
Thread Processor
Multiprocessor
Registers
Registers
Thread Processor
Thread Processor
Multiprocessor
Shared Memory
Shared Memory
Global / Constant Memory Data Cache
Compute Device
Global/Local Memory
Compute Device Memory

Thanks!

- Markus Hadwiger
- Christof Rezk-Salama, Klaus Engel, Bob Laramee
- David Kirk and Wen-mei W. Hwu
- NVIDIA, Keenan Crane
- AnandTech
- Khronos Group