Visualization

“the process of converting data into visual representations”

Today

Visualization pipeline revisited
Examples of real-time 3D visualization
GPGPU for visualization
GPU volume visualization basics

Classical Visualization Pipeline (1)

Aquire Data
- Projected densities from CT
- Images of particles in a fluid
- Concentrations of contrast agent in blood flow
- Weather radar data
- Sonar data

Classical Visualization Pipeline (2)

Data Acquisition
- Sampling to grid
- Registration
- Missing values (interpolation)
- Reduction (remove non-relevant data)
- Smoothing
- Characteristic properties (gradient, extrema, segmentation)

Classical Visualization Pipeline (3)

Filering & Mapping
- Select and Filter
- Apply transfer function (to get from density values to color and opacity)
- Compute presentable graphical primitives (triangles, glyphs)
- Consider effectiveness

Decide
- Which parts of the data are shown
- How to represent them
Classical Visualization Pipeline (4)

- Rendering
  - Generate 2D image
  - Reconstruction: compute data value at sampling point
  - Classification: assign color and opacity
  - Shading: evaluate the illumination model
  - Compositing: determine contribution to final image

- Decide
  - Visibility of data representatives
  - 3D display of representatives

Background

Visualization research has moved from
- How can we generate a visualization at all? to
- How can we create effective visualizations?

New
- Types of data
- Styles and approaches
- Requirements and benchmarks

“Data” – Characteristics

Source data can have the different properties
  - Multiple (independent) dimensions (i.e. space + time)
  - Multiple (possibly correlated) variables (i.e. temperature, velocity, density)
  - Multiple (differently generated) fields (i.e. from different modalities)

Different fields have different properties regarding scaling, sampling, measurement errors,…

“Style” – Requirements

Efficient vs. effective visualization
  - Efficient – the most efficient algorithm is the one that produces the result the fastest
  - Effective – the most effective visualization technique is the one that allows the user to get to his results quickest

Selective: Does the visualization focus on relevant information?

Purpose-driven: Does the technique apply to some real-world task?

Feature-based: Does the visualization algorithm regard relevant features (i.e. tumors, vortices)?

Volume Visualization

Real-time ray-casting
  - High quality, flexibility, and performance
  - Perspective views, advanced lighting,…

Advanced Lighting (1)

Hard shadows, soft shadows

Ambient occlusion, scattering, …
Advanced Lighting (2)

Ultrasound Data

Optical Fetoscope

3D Ultrasound

Emission + Absorption + Surface Shading

Soft Shadows + Scattering

Seismic Data

Illustrative Visualization

Focus+Context Visualization

Multivariate Data

Illustration of full data set + „ghost“ of object in focus

Local detail=global overview

Combining point sprites and triangle strips

Combination improves quality of lines

Multi-modal volume rendering

Rendering of segmented objects

Different organs, tissues, vessels, ...

Per-object transfer functions, clipping planes, modalities, ...
Multiple Views

Multiple coordinated views allow to combine the advantages of „infovis“ and „scivis“ techniques.
Alternate representations of the data to show interrelations between variables.
Linking between views for high-level interaction.

Non-destructive Testing

Visualization of high-resolution CT volumes
- Combination with feature detection & quantification.

Integrated Views

Combination of different ways to depict data in a single view.
Integrated visualization of heart model and artery tree with patient data from MRI.
Diagnosis of coronary artery disease.

Fluid Simulation (1)

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.

Fluid Simulation (2)

Can be solved on (low-res) 3D grids in real-time
- Render using ray casting or splatting.

Fluid Simulation (3)

NVIDIA Demos
- Smoke, water
- Collision detection with voxelized solid (Gargoyle)
Ray casting
- Smoke: volume rendering
- Water: level set / isosurface

Courtey Markus Harris

Parameter Space Exploration

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

Curvature of Implicit Surfaces

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

GPGPU

General Purpose Computations on GPUs
- Focus on data-parallel algorithms
- Legacy GPGPU uses graphics APIs
  - OpenGL, Direct3D
- 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:
- Texture coordinates (barycentric)
- Texture-lookup: interpolate the texture data (bi-linear)
- Nearest-neighbor for "array lookup"
- Use arbitrary, computed coordinates

3D Texture Mapping

For each fragment:
- Texture coordinates (barycentric)
- Texture-lookup: interpolate the texture data (tri-linear)
- Nearest-neighbor for "array lookup"
- Use arbitrary, computed coordinates

GPGPU Hardware

NVIDIA Tesla
- Fermi architecture
- 448 CUDA Cores
- Over 1 Teraflop / device
- Up to 6GB RAM / device
- Multiple devices per node / machine

G80/GT200 Architecture

Streaming processors can execute
- Geometry shaders
- Vertex shaders
- Fragment shaders
- CUDA kernels
**G80/GT200 Architecture**

- 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, ...

**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

**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)

**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

**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.

- Serial Code
  - Host
- Parallel Kernel
  - Device
- Serial Code
  - Host
- Parallel Kernel (args):
  - 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.

- Thread Block 0
  - Shared Memory
  - Global Memory
  - Constant Memory
  - Texture Memory

- Thread Block 1
  - Shared Memory
  - Global Memory
  - Constant Memory
  - Texture Memory

- Thread Block 2
  - Shared Memory
  - Global Memory
  - Constant Memory
  - Texture Memory

- Thread Block 3
  - Shared Memory
  - Global Memory
  - Constant Memory
  - Texture Memory

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

Warp Synchronization

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)

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Per thread</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Local memory</td>
<td>Per thread</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Per block</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Global memory</td>
<td>Per grid</td>
<td>Read-Write</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Per grid</td>
<td>Read-only</td>
</tr>
<tr>
<td>Texture memory</td>
<td>Per grid</td>
<td>Read-only, Spatially cached</td>
</tr>
</tbody>
</table>

Texture memory can be helpful for moving legacy 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

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

```cpp
// CUDA program

void addMatrix(float A, float B, float C[], int M, int N) {
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
}
```
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

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

Increase

- emission
- in-scattering

Decrease

- absorption
- out-scattering
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)

Ray Integration (3)

Numerical Solution (1)

Numerical Solution (2)
Now we introduce opacity.

\[
\tau(0,t) \approx \tau(0,0) = 1 - \sum_{i=0}^{\lfloor i/\Delta t \rfloor} K(\Delta t) \Delta t
\]

\[
q(t) \approx Ec(t) + e^{-\tau(t)} = e^{-\tau(t)} \left[ 1 - \sum_{i=0}^{\lfloor i/\Delta t \rfloor} K(\Delta t) \Delta t \right]
\]

Now we introduce opacity.

\[
C_i = \frac{1}{1 - A_i} \sum_{j=0}^{\lfloor j/\Delta t \rfloor} C_j e^{-\tau(j,0)}
\]

Can be computed recursively.

\[
q(t) = \prod_{i=0}^{\lfloor i/\Delta t \rfloor} (1 - A_i)
\]

Early Ray Termination

Stop the calculation when

\[
A_i' \approx 1
\]

Back-to-front iteration

\[
C_i' = C_i + (1 - A_i)C_i'
\]

\[
A_i' = A_i' + (1 - A_i + A_i')A_i
\]

Summary

Emission Absorption Model

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

Back-to-front iteration

Front-to-back iteration

\[
C_i' = C_i + (1 - A_i)C_i'
\]

\[
A_i' = A_i' + (1 - A_i + A_i')A_i
\]

GPU Volume Rendering

Today, standard ray casting can be implemented on the GPU

On previous hardware generations, only slicing was possible
In raycasting, sampling is typically performed at equidistant points along each ray.

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

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

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
Thanks!

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