Volume Rendering for Visual Effects

Magnus Wrenninge
Sony Pictures Imageworks
SIGGRAPH course

• For much more details
  • magnuswrenninge.com/productionvolumerendering
What Are Volumetrics?

- Clouds
- Mist
- Dust
- Steam
- Smoke
- Fire
What Are Volumetrics?

- Really tiny bits of matter
- Represented as mass in a volume
- AKA density
- Probabilistic interactions with light
Goals

• Making Volumetric Things
  • Fuzzy
  • Fluffy
  • Sparkly
• Rendering Them Too
  • Art directed
  • Scalable
A Brief History Of Volumetrics

• Cloud Tank Effect
  • Created for “Close Encounters of the Third Kind”
  • Paint sitting on liquid density interface
  • Version 1 cost $20 + aquarium

Cloud Tank Effect In “Independence Day”

© 1996 Twentieth Century Fox and Centropolis Entertainment. All rights reserved.
Particle Systems

- Used in “Star Trek II: The Wrath of Khan” (1982)
- Still used today
- Good for more discrete media
  - Sand
  - Droplets
“BAMF” Effect in “X2”

© 2003 Twentieth Century Fox. All rights reserved.
Advent Of Volumes

- Late ‘90s
- Enough system memory
- Fast enough processing power
Digital Domain’s Storm

• System for modeling and rendering volumes
• Authored by Alan Kapler
• And
  • Lucio Flores
  • Ryo Sakaguchi
  • Josh Krall
  • Nafees Bin Zafar
  • Peter Baldwin
Storm in the “Ford of Bruinen”
Storm in “xXx”

© 2002 Columbia Pictures. All rights reserved.
Storm in “xXx”

- First use of frustum buffers
- Pyroclastic noise
- In-core compression
Storm

- Built in scripting language for modeling
- Fixed shading pipeline
- Custom “deep shadow” format
- Implemented inside Houdini
- Sci-Tech Academy Award 2004
State Of The Production Art

- Mantra
- RenderMan
- MF/D2R
- DNB
- Svea
- GPU Renderers: ILM, Weta, MPC
A simple volumetrics system

- Geometric primitives
- Voxel filling tool
- Voxel buffer
- Camera
- Lights
- Voxel renderer
- Final image
Volume modeling
Volume modeling

- What is volume modeling?
  - Turning “some” data into volumetric data
- Voxel buffers
  - But not only!

- Introduction to some basic volume modeling techniques
Voxel buffers
Voxel buffers

- 2D array of values
  - Pixels → Image
- 3D array of values
  - Voxels → Voxel buffer
Voxel buffers

- Many names
  - Voxel grid
  - Voxel buffer
  - Discrete field
- Scalar field/buffer/grid
- Vector field/buffer/grid

- We will use ‘buffer’
Voxel buffers

- What must a voxel buffer do?
  - Define resolution
  - Define its place in space
    - Transform/mapping
  - Read values
  - Write values
Voxel buffers – Implementing a voxel buffer

- Countless ways to implement a voxel buffer
- Easiest – map 3D coordinates into 1D array
Voxel buffers – Implementing a voxel buffer

```cpp
void setSize(int xSize, int ySize, int zSize)
{
    m_data.resize(xSize * ySize * zSize);
}

float VoxelBuffer::value(int i, int j, int k)
{
    return m_data[i + j * xSize + k * xSize * ySize];
}

More details in the course notes!
```
Voxel buffers – Implementing a voxel buffer

• Don’t want to write one?
  • Field3D
  • Multiple data structures
    • DenseField
    • SparseField
    • MACField
  • http://field3d.googlecode.com/
Voxel buffers – Coordinate spaces

- Voxel space
  - Native coordinate space
- Local space
  - Resolution independent
- World space
  - Global frame of reference

- More about transformations/mappings in a little bit
Reading & writing to voxel buffers
Reading from voxel buffers

• Case A: Reading directly from voxels
  • As previously described
  • Integer-coordinate access

• Case B: Reading values in-between voxels
  • Requires interpolation
Reading from voxel buffers – Interpolation

- Lots of interpolation schemes to choose from
  - Nearest neighbor
  - Linear
  - Cubic
  - etc.
- *See course notes for details & code*
Reading from voxel buffers – Interpolation

- Performance issues
  - Voxels next to each other may be far apart in memory
  - Depends on data structure
  - Linear arrays (dense buffers) give poor cache performance

![Diagram showing sample location with indices i,j,k, i-1,j,k, i-1,j+1,k, i,j+1,k, i,j,k, i,j+1,k, i-1,j+1,k, i-1,j,k, i,j,k.](image)
Writing to voxel buffers

• Case A: Writing a value directly to a voxel
  • As previously described
  • Integer-coordinate access

• Case B: Writing a sample in-between voxels
  • Samples rarely line up with voxels
  • Splatting/stamping/baking
  • Many solutions available
Writing to voxel buffers – Nearest neighbor

- Simplest solution
- Just round to nearest integer
- No anti-aliasing
  - Still useful, especially when using lots of points
Writing to voxel buffers – Trilinear splat

- Use triangle filter
- Radius 1.0 ⇒ only touches 8 voxels
- *Details and code in course notes*
Noise & noise spaces
Noise coordinate systems

- Noise functions used almost everywhere in volume rendering
- Geometric primitives need simple parameterization to be used as noise space
  - Along surface
  - And also away from surface
- Transforms need to evaluate quickly
  - Difficult for primitives without closed-form solutions
- Transforming out of a primitive's space is sometimes cheaper than into it
Noise coordinate systems – Spheres

\((e_1, e_2, e_3) = (x, y, z)\)
Noise coordinate systems – Curves

\[(e_1, e_2, e_3) = (N \times T, N, T)\]
Noise coordinate systems – Surfaces

\[ (e_1, e_2, e_3) = (dP/du, dP/dv, N) \]
Volume modeling using geometry
Geometry-based volume modeling

• Four conceptual layers
  1. Integer-coordinate access
  2. Splatting
  3. Rasterization primitives
  4. Instantiation primitives
Rasterization-based primitives
Rasterization primitives
Rasterization – Algorithm

- Get primitive’s attributes
  - Position, size, noise settings, etc.
- Compute bounds in voxel space
- For each voxel in bounds
  - Calculate density

- Details in course notes
Rasterization – Solid noise primitive

- A very simple point-based primitive
- Needs two components
  - Noise function
  - A way to fade noise off at edges
- Pick your favorite noise function
  - fbm fractal used here
- How do we fade edges and apply noise?
  - $1 - |x| + \text{fbm}(x)$
Rasterization – Pyroclastic noise primitive

- Previous example produces density by modulating by noise
- Let’s consider distance functions
  - Simple example – sphere
  - Measure distance from center
  - Threshold to create a sharp interface
Rasterization – Pyroclastic noise primitive

- Adding noise to distance function produces pyroclastic noise
- Basic function
  - distance(P) + fbm(P);
- Absolute noise looks better
  - distance(P) + abs(fbm(P));
- Vary amplitude
  - distance(P) + A * abs(fbm(P));
Comparing displacements

2D displacement

3D displacement
Animating noise amplitude

2D displacement

3D displacement
Instantiation-based primitives
Instantiation-based primitives

• Builds primitives out of lower-level primitives
• Point instantiation most common
• Different names at different facilities
  • Wisps
  • Generators
Instantiation-based primitives

- Rasterization primitives great for continuous primitives
  - Clouds, fog, etc.
- Negative space can be a performance issue
  - All voxels need to be calculated
- Rasterization **pulls** density into voxels
  - Need to calculate all voxels

- Instantiation-primitives are different
  - **Pushes** values into voxels
  - Calculate density first, then figure out where it winds up
  - Can handle primitives that self-intersect
  - Can handle cases where nearest-point query is ambiguous
Instantiation-based primitives

- Consider raytracing vs. micropolygon rendering
  - Raytracing treats **pixels first**, figures out which primitive(s) is visible to a pixel
  - REYES computes **geometry first**, later which pixels it goes into

- Rasterization primitives are like raytracing
  - **For each voxel**, see how each primitive contributes
- Instantiation primitives are like micropolygon rendering
  - For each piece of geometry, **generate data first**, then rasterize into voxels
Instantiation-based primitives

- Transforming *out of* coordinate spaces often more efficient than *into*
  - Sum of bases vs. search
- Rasterization has to transform *into* the primitive’s space
- Instantiation transforms *out of* the primitive’s space
The simplest point instantiation primitive

- Pick a random locations on surface of sphere
- Rasterize points into voxel buffer
Slightly more interesting...

- Pick a random locations on surface of sphere
- Displace points by perlin noise
  - \( P += \text{fbm}(P) \);
- Rasterize points into voxel buffer
Adding another noise

- Pick a random locations on surface of sphere
- Displace points by perlin noise
  - \( P += \text{fbm}(P); \)
- Modulate density by perlin noise
  - \( \text{density} *= \text{fbm}(P); \)
- Rasterize points into voxel buffer
Extending concept to other primitives
Instantiating along a curve

pick segment
pick random position in segment
find $P(u)$
pick random position on disk
find $T(u)$
find $N(u)$
compute $N \times T$
P $\leftarrow s \times N \times T \times radius$
P $\leftarrow t \times N \times radius$
Example using a few more points

- 10,000 × 6,000 image
- 10,000 × 6,000 × 400 resolution voxel buffer
  - Sparse, frustum shaped
- 15 billion instanced points
- Single curve primitive
Point-based instantiation – Curves

- Smooth primitives possible
  - Just need a lot of points
- Constant-radius curve
- 40 million points @ 1024x1024 pixels
Applying noise to instantiation primitives

• Let’s look at some ways to introduce variation procedurally
• Noise/fractals can be added in many different forms
Point-based instantiation – Curves – Noises

- Modulate density by perlin noise

```
density *= fbm(P);
```
Point-based instantiation – Curves – Noises

- Modulate density by absolute perlin noise

```
density *= fbm_abs(P);
```
Point-based instantiation – Curves – Noises

- Displace along normal by absolute perlin noise
  - Same principle as pyroclastic noise points

\[ P += N * \text{fbm}_\text{abs}(P); \]
Point-based instantiation –
Curves –
Noises

• Displace along normal by absolute perlin noise
• Same principle as pyroclastic noise points
• $P += N \times \text{std::abs(fbm}(P)$
Point-based instantiation –
Curves –
Noises

• Displace along normal by absolute perlin noise
• Same principle as pyroclastic noise points

\[ P += N \times \text{std::abs(fbm}(P)) \]
Point-based instantiation – Curves – Noises

• Displace along normal by absolute perlin noise
• Same principle as pyroclastic noise points

\[ P += N \times \text{std::abs}(\text{fbm}(P)) \]
Point-based instantiation – Curves – Noises

- Displace by perlin noise
  - Distorts shape in all directions

\[ P \leftarrow \text{vec}_f \text{bm}(P); \]
Point-based instantiation

- Curves
- Noises

- Displace by perlin noise
- Distorts shape in all directions

\[ P += \text{fbm}(P) \]
Point-based instantiation –

- Curves
- Noises

- Displace by perlin noise
- Distorts shape in all directions

\[ P += f_{bmf}(P) \]
Point-based instantiation – Curves – Noises

- Combinations
  - Modulate density by perlin noise
  - Displace by perlin noise

```plaintext
density *= fbm(P);
P += vec_fbm(P);
```
Surface-based instantiation primitives

- Similar to the curve primitives, but uses a surface as control primitive
- Can be poly mesh or parametric surface
- *Code in course notes!*
Motion blur in volume modeling
Motion blur in volume modeling

- Motion blur is required for any sort of production rendering
- Raytracers and micropoly renderers (can) both handle motion blur well
- “Correct” motion blur almost impossible to achieve
  - So we cheat!

- Question: How to create reasonably correct motion blur?
Motion blur in volume modeling

• One common solution
• **Smear the samples**
  • As each voxel is computed, distribute value across multiple voxels
Motion blur using smearing

- Two approaches

- Line drawing
  - Standard DDA/Bresenham

- Multiple splats
  - Easier to implement, reuse code from voxel splatting
  - *Code example in course notes*
Problems with smearing

- Technically incorrect
  - Shading is applied after motion blur
  - Results in overly soft lighting
  - We are folding time into the spatial domain

- However!
  - Correct blur is hard to accomplish
No motion blur

Smeared motion blur

Time-averaged, “correct”
High resolution voxel buffers
High resolution voxel buffers

- Densely allocated buffers scale to around $1000^3$
- Film frame resolution commonly worked on at 2K
  - $2048 \times 1556$
- How do we make a buffer with enough detail?
- And what if we zoom into that buffer?

- **Idea 1**
  - Only allocate voxels where you need to store data
- **Idea 2**
  - If the camera can’t see it, don’t worry about it
Idea 1 – Sparse buffers

- Only allocate storage for voxels where needed
  - From programming we know about copy-on-write
  - For voxel buffers: allocate-on-write
- Pretend that the voxels are there until you actually need them
Idea 1 – Sparse buffers

- Block sparse buffer
  - Start with $512^3$ buffer
  - Divide into $8^3$ coarse blocks
  - Each block stores just one value, until we write to a voxel
- After writing to voxels
  - Allocation of $2 \times 64^3$ voxels

- Most buffers in volume rendering contain unused space
Idea 1 – Sparse buffers

- $O(k)$ access time
- Higher $k$ than densely allocated buffers, due to overhead
Idea 2 – Frustum shaped buffers

- Second idea – only place voxels where the camera can see them
- Voxels farther away from camera are larger
- Seen from camera, looks like uniform sampling
Idea 2 – Frustum shaped buffers

- X/Y resolution usually on the same order as the final image (~2K)
- Z resolution much lower
  - Usually a few hundred ‘slices’
- Simple transform
  - Use camera’s `worldToNDC()`
High resolution voxel buffers

- Idea 2.5
- Combine sparse buffer and frustum buffer
- One is a data structure, the other is a mapping
- Orthogonal concepts, can work in conjunction
Problems with frustum buffers
Problems with frustum buffers
Volume rendering
But How Do We **DO** It?

- Ray marching
Some Definitions

- Transmittance: Fraction of light that passes through ($T$)
- Opacity: Fraction of light that is absorbed ($1-T$)
- Beer’s Law
  - Relates absorption capacity to $T$
  - $T = e^{-\sigma \rho \Delta x}$
  - Uses material properties
Very Simple Lighting

Repeat

- Sample density \( \rho \)
- Calculate local transmission \( T_i = e^{-\sigma \rho \Delta x} \)
- Lookup lighting \( L \)
- Lookup material color \( c \)
  - Calculate color: \( C_i = T_{\text{new}} L c \rho \Delta x \)
  - \( C_{\text{total}} = C_{\text{old}} + C_i \)
  - \( \alpha = \alpha_{\text{old}} + (1 - T_i) (1 - \alpha_{\text{old}}) \)
  - Repeat
Don’t Take My Word For It
Same Process Makes Dirt
And Water
Hints On Software Design

- Raymarcher
  - Generates rays
  - Advances them
- Occlusion module
  - Beer’s law or something else
- Shader
  - Computes sample color
- Integrator
  - Accumulates transmittance, color, alpha
Notes on Artistic Controls

• There are no laws
  • Don’t have to use Beer’s law
  • Even accumulation does not have to mean straight addition
• Separate extinction coefficient for lighting and rendering
• Vary parameters per color

• Step length, $\Delta x$, trades quality for speed
Changing Rendering Extinction Coefficient
Changing Lighting Extinction Coefficient
Pre-computed lighting
Pre-computed lighting

• One of the most expensive steps in raymarching is calculating lighting
• Each step along primary ray requires another raymarch toward each
  each
• Exponential explosion

• How can we avoid this?
  • Pre-compute information
Pre-computed lighting – Voxelized lighting

• Idea 1: Compute incoming light on a voxel grid

• Benefits:
  • Decouple sampling of incoming light from sampling of density

• Drawback:
  • Still need to perform full raymarch per voxel
Pre-computed lighting – Shadow maps

- Idea 2: Consider path of light
  - Light travels linearly from each light source
  - Transmittance at $V_2$ depends on $V_1$, $V_3$ depends on $V_2$, etc.
- Shadow maps work well for surface rendering
- Calculate transmittance function of light source
Pre-computed lighting – Shadow maps

- Raymarching primary rays calculates transmittance
- Re-use code
- Raymarch from light source
Deep shadows

- Once constructed, store shadow map with a transmittance function per pixel.
- When rendering, perform lighting calculation:
  - \( L_{in} = L_{light} \times T_{light} \)

Putting it all together
Voxel buffer

Voxel buffer types
- Dense buffer
- Sparse buffer
- Other types

&
- Frustum shaped
- Uniform/orthogonal

Voxel-based modeling
- Splatting
- Rasterization primitives
- Instantiation-based primitives

Procedural noise

Motion blur

Ray distribution
- Volumes
- Camera
- Lights

Scene description
- Camera rays
- Improving sampling

Raymarcher
- Empty space optimization

Scattering model
- Lighting calculation

Holdouts
- Holdout maps
- Blocker maps
Questions?
Next: Production Work