Volume Rendering For Games

Simon Green
NVIDIA Corporation

Overview

- Why use volume rendering?
- Volume rendering using slices
- Volume rendering using ray marching
- Ray-box intersection
- Procedural volumes
- Conclusion

Why use Volume Rendering?

- **Many phenomena in the real world cannot be easily represented using geometric surfaces**
  - Clouds, smoke, fire, explosions
- **The appearance of these phenomena is caused by the cumulative effect of light emitted, absorbed and scattered by a huge number of tiny particles**

Particle Systems

- Volumetric effects usually approximated in games today using particle systems with point sprites
- Problems with particle systems
  - obvious intersections between sprites and scene geometry
    - can be improved with depth replace
  - accurate lighting, shadowing is difficult
  - texture movies use a lot of memory

Particle System in "Vulcan"

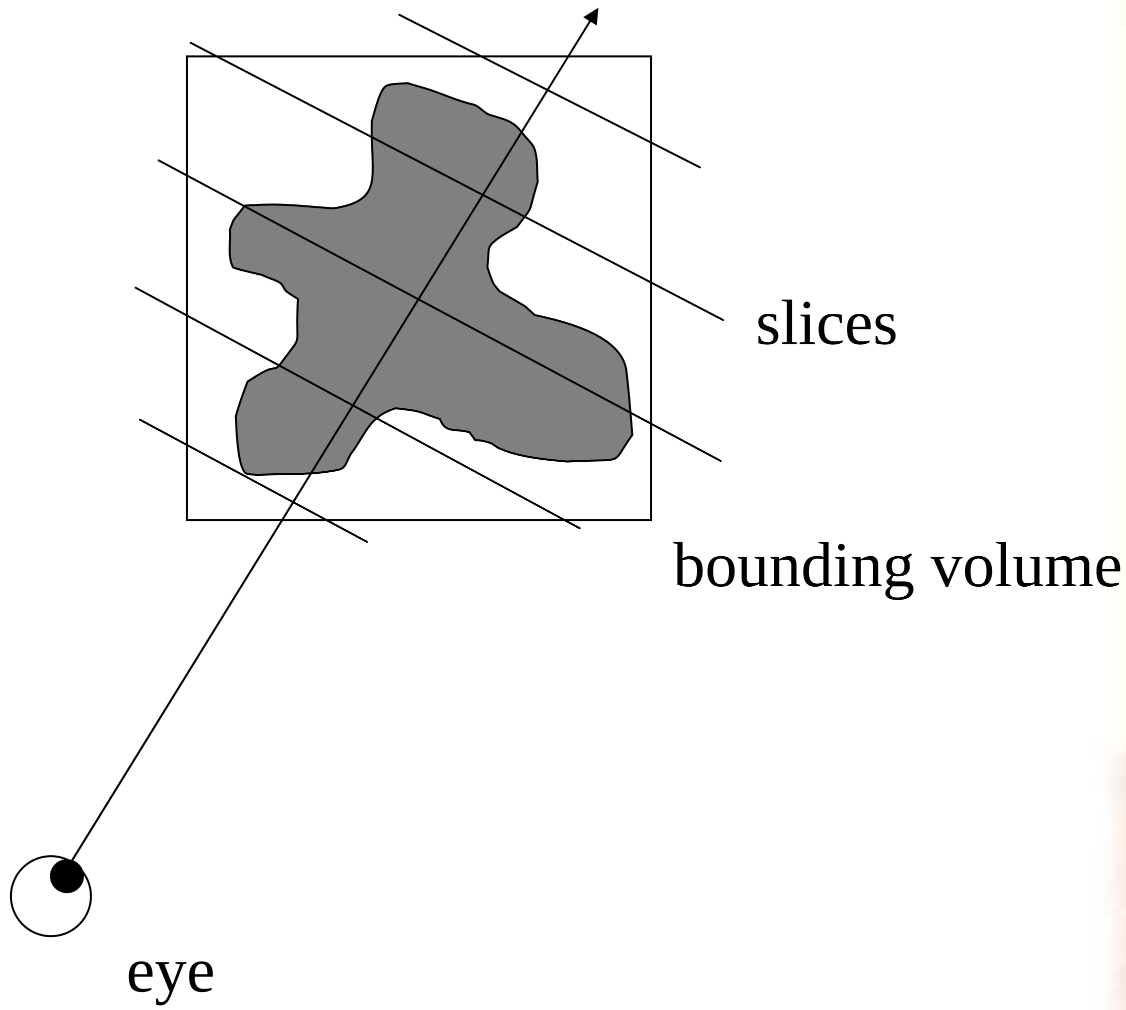
Volume Rendering

- Volume rendering simulates effect of light emitted, absorbed and scattered by large number of tiny particles in volume
- Volume is represented as a uniform 3D array of samples
  - can be pre-computed, or procedural
- Final image is created by sampling the volume along viewing rays and accumulating optical properties

Volume Rendering using Slices

- Volume is sampled using proxy geometry
- Polygons slice through the volume perpendicular to the viewing direction
- Number of slices determines quality of resulting image
- Slices usually drawn back to front
- Compositing performed using alpha blending
  - "over" operator: $C'_i = C_i + (1 - A_i)\, C'_{i+1}$
  - uses a lot of framebuffer bandwidth

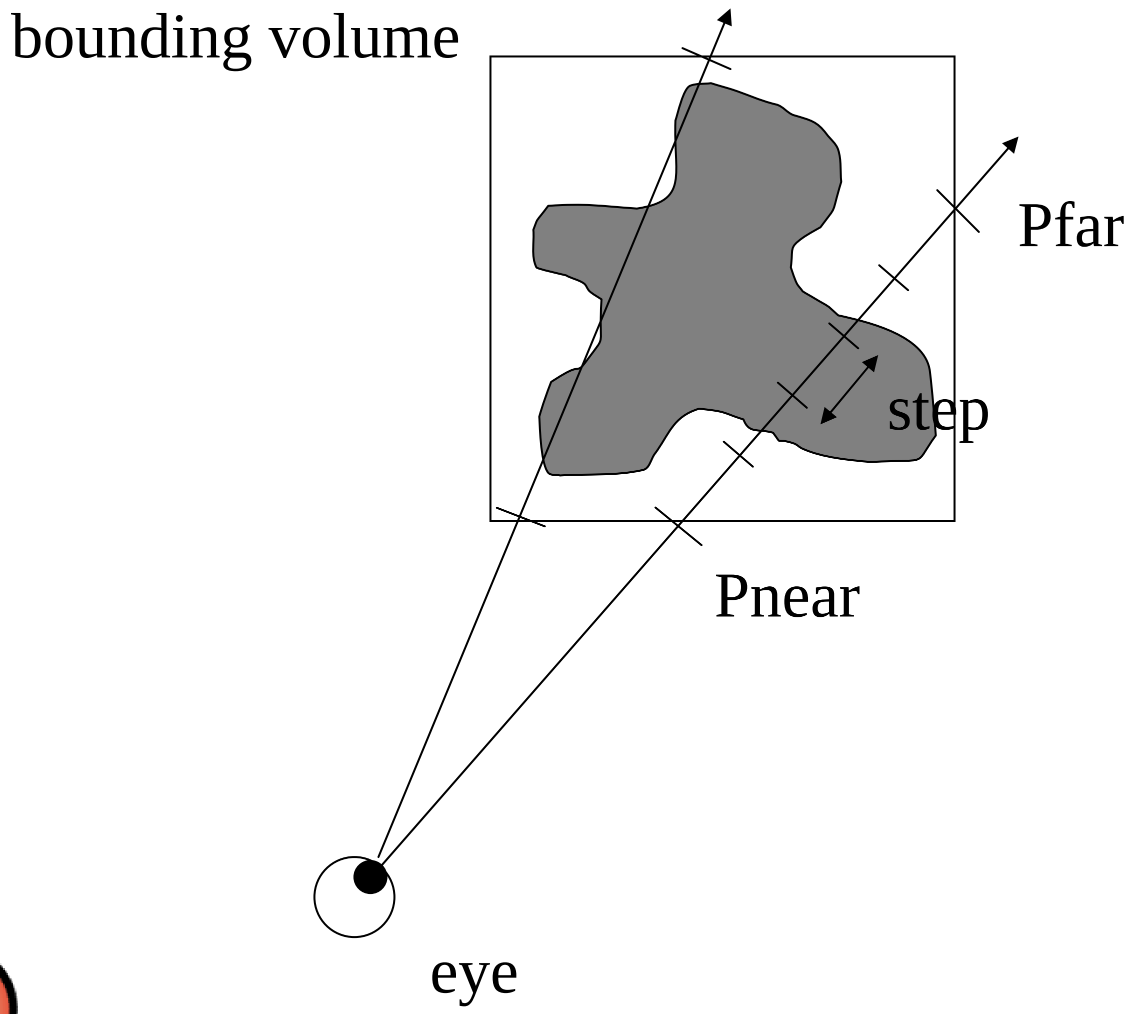
Volume Rendering using Slices



Volume Rendering by Ray Marching

- Calculate intersection between view ray and bounding volume
- March along ray between far and near intersection points, accumulating color and opacity
  - Look up in 3D texture, or evaluate procedural function at each sample
  - Can use uniform of adaptive step sizes

Ray Marching



Ray Marching Code

```
float4 RayMarchPS(Ray eyeray : TEXCOORD0,
                  uniform int steps) : COLOR
{
    eyeray.d = normalize(eyeray.d);

    // calculate ray intersection with bounding box
    float tnear, tfar;
    bool hit = IntersectBox(eyeray, boxMin, boxMax, tnear, tfar);
    if (!hit) discard;
    if (tnear < 0.0) tnear = 0.0;

    // calculate intersection points
    float3 Pnear = eyeray.o + eyeray.d*tnear;
    float3 Pfar = eyeray.o + eyeray.d*tfar;

    // march along ray, accumulating color
    half4 c = 0;
    half3 step = (Pnear - Pfar) / (steps-1);
    half3 P = Pfar;
    for(int i=0; i<steps; i++) {
            half4 s = VOLUMEFUNC(P);
            c = s.a*s + (1.0-s.a)*c;
            P += step;
    }
    c /= steps;
    return c;
}
```

Ray Marching Advantages

- Loop compiles to REP/ENDREP in PS3.0
  - Allows us to exceed the 512 instruction PS2.0a limit
  - 100 steps is interactive on 6800 Ultra
- All blending is done in floating-point precision in the shader

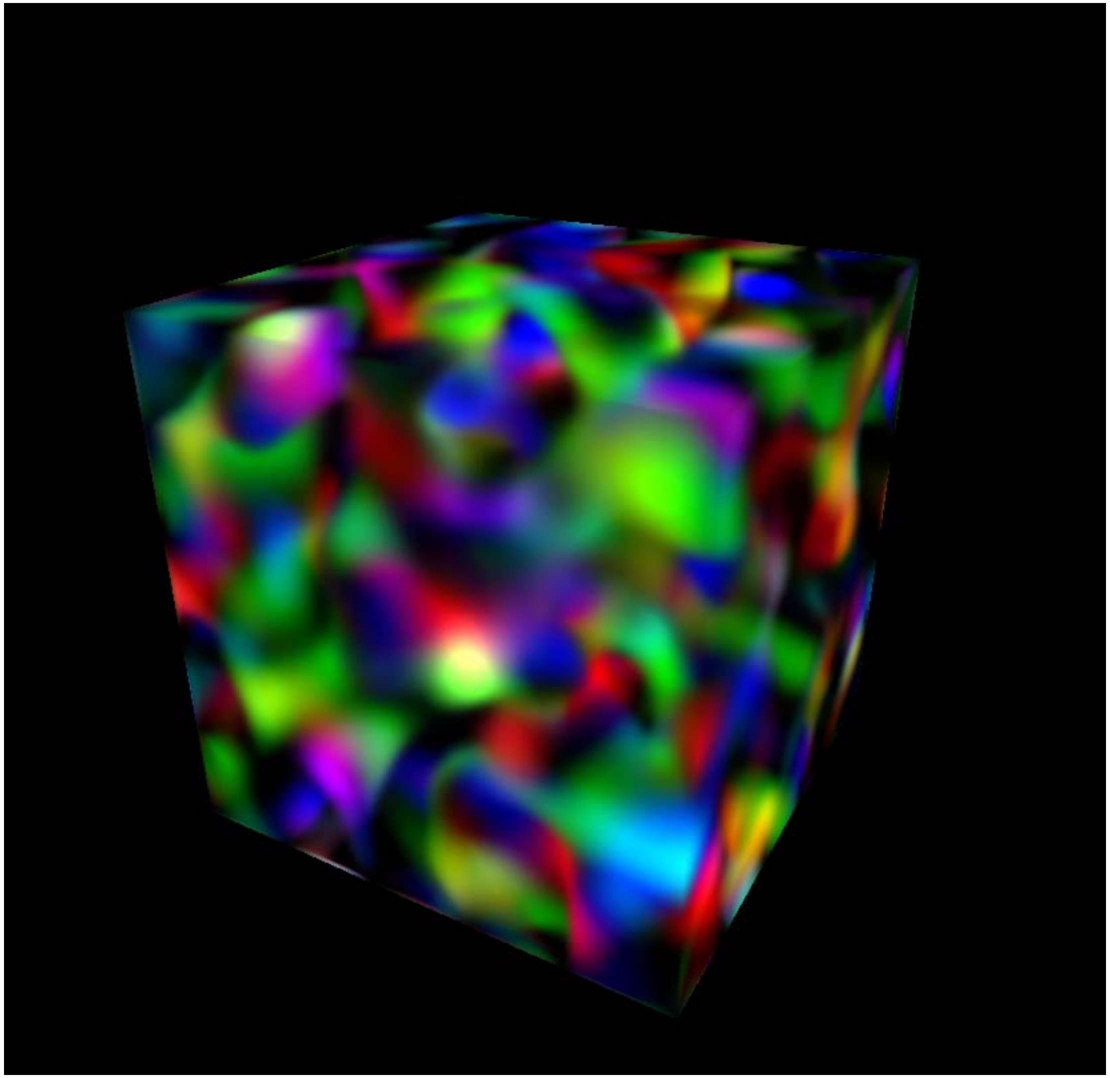
Bounding Volume Intersection

- We use bounding box
- Intersect with ray using Kay & Kajiya “slabs” method
- Axis-aligned box defined by minimum and maximum coordinates
- Slab is the space between two parallel axis-aligned planes
- Intersection of three slabs defines box
- Easy to vectorize

Ray-Box Intersection Shader Code

```
bool
IntersectBox(Ray r, float3 boxmin, float3 boxmax, out float tnear,
    out float tfar)
{
    // compute intersection of ray with all six bbox planes
    float3 invR = 1.0 / r.d;
    float3 tbot = invR * (boxmin.xyz - r.o);
    float3 ttop = invR * (boxmax.xyz - r.o);

    // re-order intersections to find smallest and largest on each axis
    float3 tmin = min (ttop, tbot);
    float3 tmax = max (ttop, tbot);

    // find the largest tmin and the smallest tmax
    float2 t0 = max (tmin.xx, tmin.yz);
    tnear = max (t0.x, t0.y);
    t0 = min (tmax.xx, tmax.yz);
    tfar = min (t0.x, t0.y);

    // check for hit
    bool hit;
    if ((tnear > tfar))
        hit = false;
    else
        hit = true;
    return hit;
}
```

Ray Marching Static 3D Texture

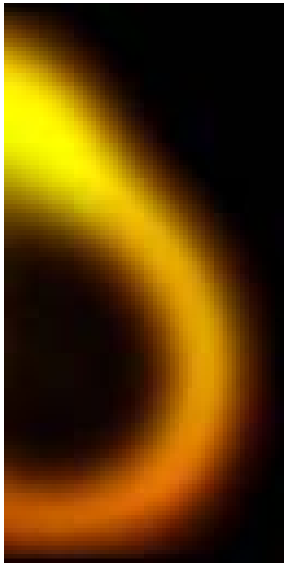


Ray Marching Procedural Volumes

- In addition to ray marching static volumes stored in 3D textures, we can also render procedural volumes defined by functions
- Volume function takes a position in 3D space, returns a color and opacity
- Can produce many different effects by varying parameters and look-up tables
- Procedurals great for generating non-repeating animated effects

Example: Volumetric Flame

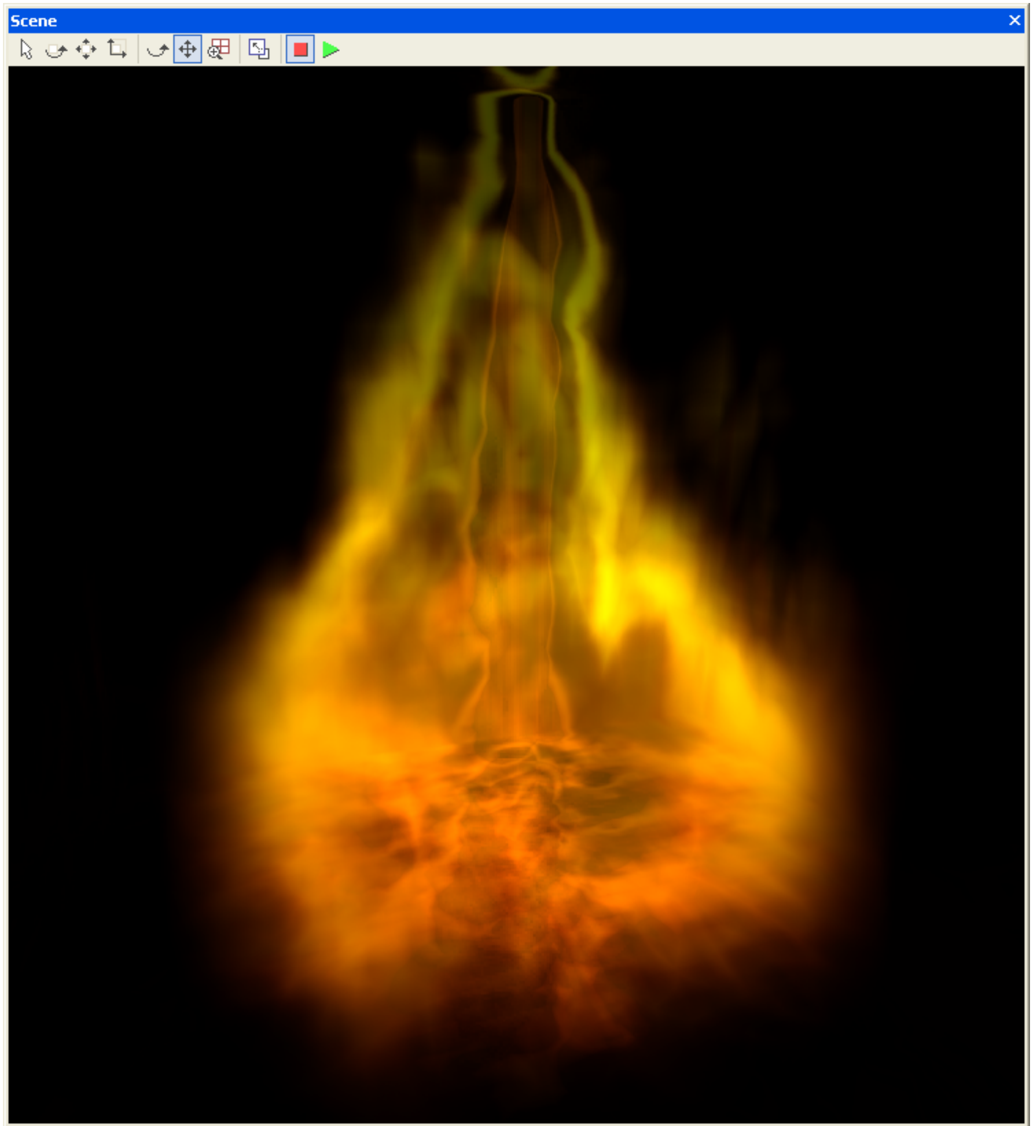
- Revolves image of cross-section of flame around Y axis to produce volume
- Perturbs texture coordinates using 4 octaves of noise (stored in 3D texture)
- Based on a shader by Yury Uralsky



Procedural Flame Code

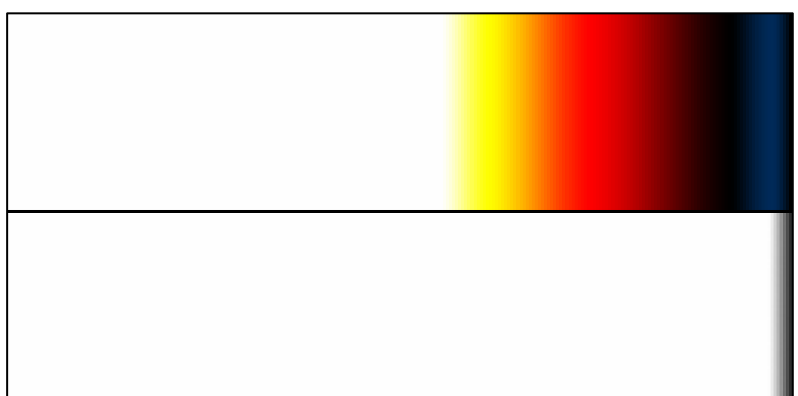
```
float4 flame(float3 P)
{
    P = P*flameScale + flameTrans;

    // calculate radial distance in XZ plane
    float2 uv;
    uv.x = length(P.xz);
    uv.y = P.y + turbulence4(noiseSampler, noisePos) * noiseStrength;

    return tex2D(flameSampler, P.xy);
}
```

Volumetric Flame Image



Procedural Fireball

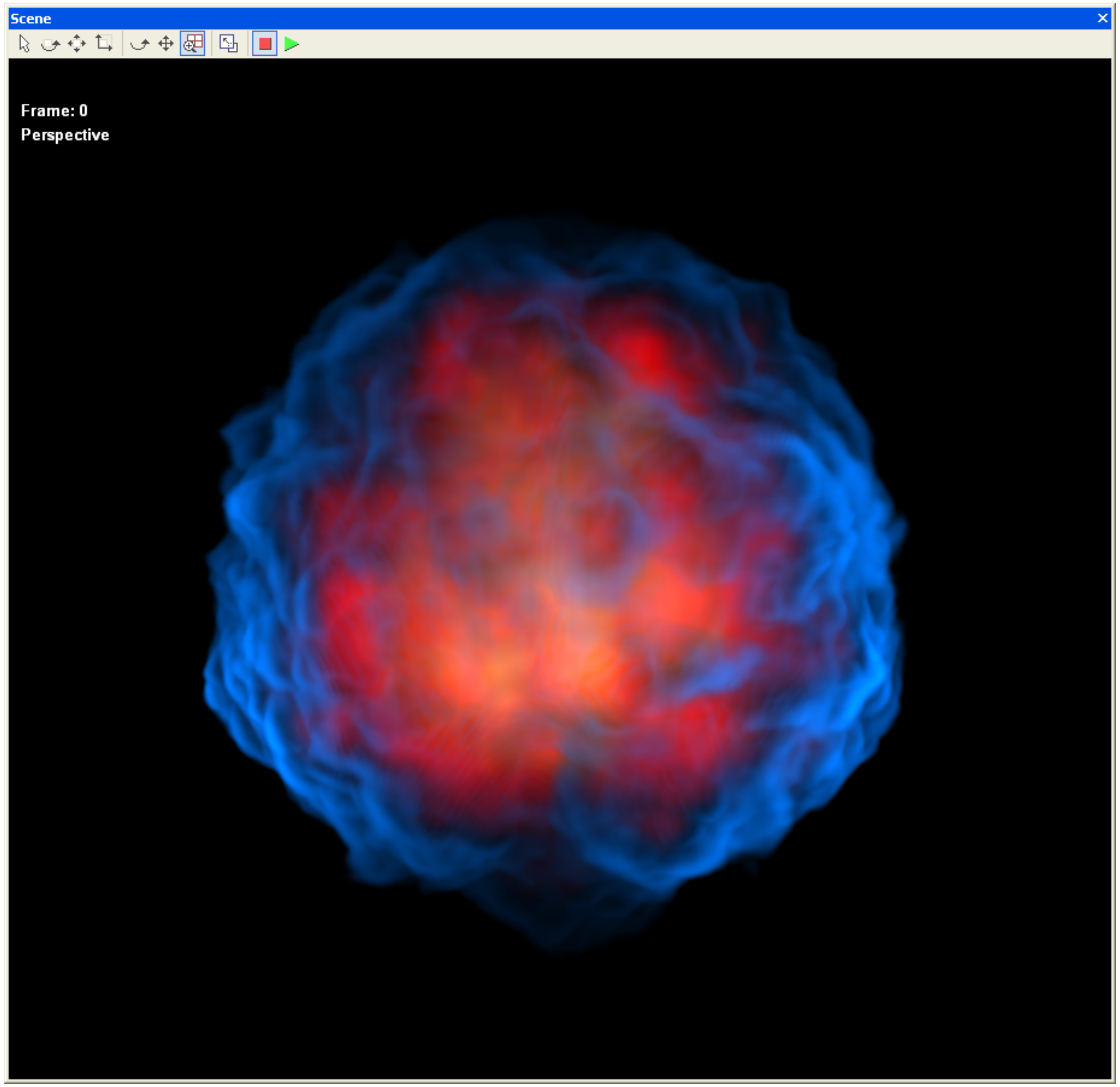
- Similar to flame, but spherical
- Calculates distance from center of volume
- Perturbs distance using noise
- Maps distance to color and opacity using 1D texture



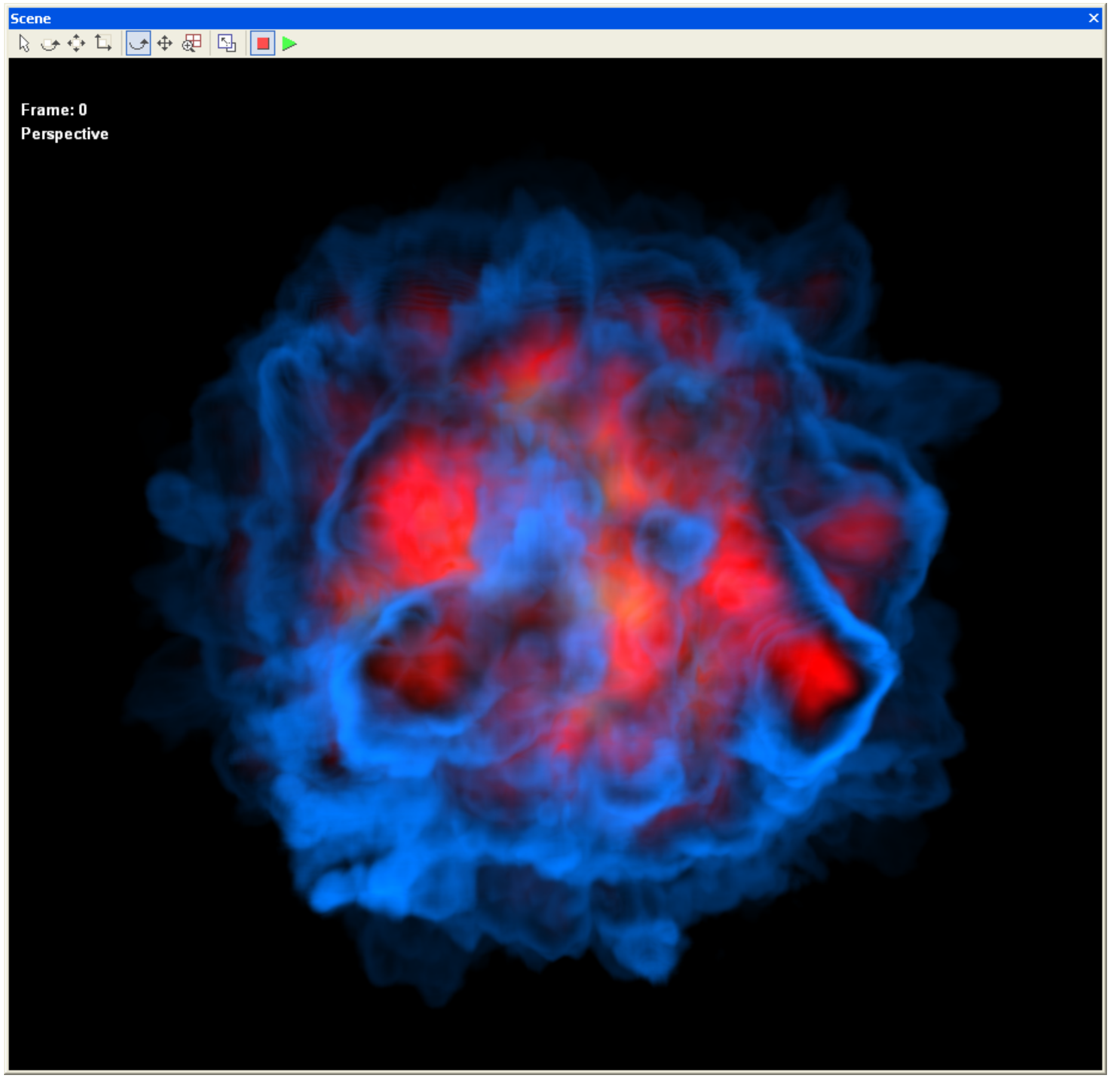
Fireball Code

```
float4 fireball(float3 p, float time)
{
    float d = length(p);
    d += turbulence(p*noiseFreq + time*timeScale).x * noiseAmp;
    float4 c = tex1D(gradientSampler, d*distanceScale);
    return c;
}
```

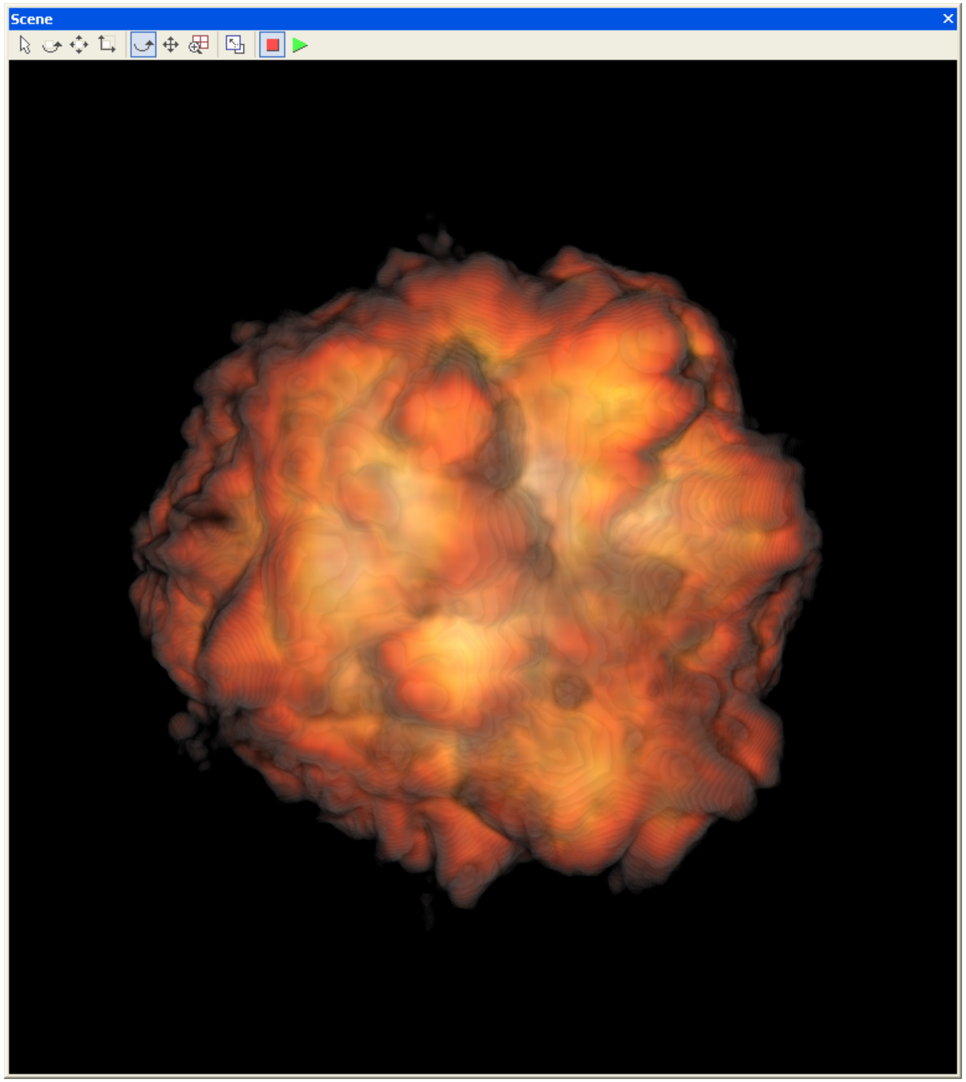
Procedural Fireball



Procedural Fireball



Procedural Explosion



Future Work

- Lighting
  - For static volumes, can pre-calculate gradients and store in RGB
  - Volumetric shadowing
    - At each sample, can march along another ray toward light
    - Very expensive
- Use early exit from loop once opacity reaches threshold
- Integrating volume effects with scene geometry

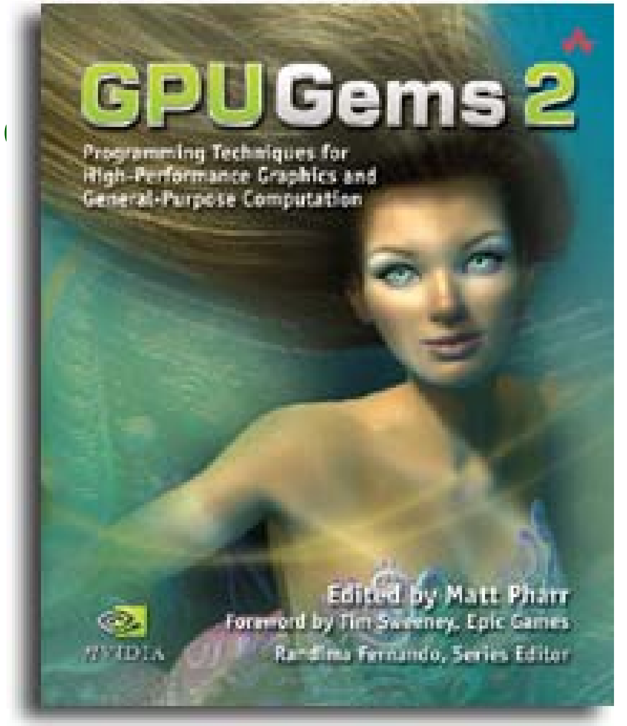
Conclusion

- Volume rendering is fun
  - Becoming practical for use in games
- Shader model 3.0 looping makes new effects possible


The Source for
GPU Programming
developer.nvidia.com
• Latest News
• Developer Events Calendar
• Technical Documentation
• Conference Presentations
• GPU Programming Guide
• Powerful Tools, SDKs and more ...
Join our FREE registered developer program for early access to NVIDIA drivers, cutting edge tools, online support forums, and more.
NVIDIA
developer.nvidia.com
©2004 NVIDIA Corporation. NVIDIA, and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation. Nalu is ©2004 NVIDIA Corporation. All rights reserved.

GPU Gems 2

Programming Techniques for High-Performance Graphi
and General-Purpose Computation



- **880 full-color pages, 330 figures, hard cover**
- **$59.99**
- **Experts from universities and industry**

"The topics covered in *GPU Gems 2* are critical to the next generation of game engines."

— Gary McTaggart, Software Engineer at Valve, Creators of Half-Life and Counter-Strike

"*GPU Gems 2* isn't meant to simply adorn your bookshelf—it's required reading for anyone trying to keep pace with the rapid evolution of programmable graphics. If you're serious about graphics, this book will take you to the edge of what the GPU can do."

—Rémi Arnaud, Graphics Architect at Sony Computer Entertainment