Game Developers Conference

MARCH 20-24
SAN JOSE, CALIFORNIA

WHAT'S NEXT
GDC:06

www.gdconf.com

GAME DEVELOPERS CHOICE AWARDS
INDEPENDENT GAMES FESTIVAL

GDC MOBILE
SERIOUS GAMES SUMMIT
GAME CONNECTION
Practical Metaballs and Implicit Surfaces

Yury Uralsky
NVIDIA Developer Technology
Agenda

- The idea and motivation
- Implementation details
- Caveats & optimizations
- Where to go from here
- Conclusion
What are isosurfaces?

- Consider a function \( f(x, y, z) \)
  - Defines a scalar field in 3D-space
- Isosurface \( S \) is a set of points for which
  \[ f(x, y, z) = \text{const} \]
- \( f(x, y, z) = \text{const} \) can be thought of as an implicit function relating \( x, y \) and \( z \)
  - Sometimes called implicit surfaces
What are isosurfaces?

- $f(x, y, z)$ can come from
  - Scattered data array
  - Mathematical formula

- Isosurfaces are important data visualization tool
  - Medical imaging
  - Science visualization
  - Hydrodynamics
  - Cool effects for games!
Metaballs

- A particularly interesting case
- Use implicit equation of the form
  \[ \sum_{i=1}^{N} \frac{r_i^2}{\|x - p_i\|^2} = 1 \]
- Gradient can be computed directly
  \[ \text{grad}(f) = -\sum_{i=1}^{N} \frac{2 \cdot r_i^2}{\|x - p_i\|^4} \cdot (x - p_i) \]
- Soft/blobby objects that blend into each other
  Perfect for modelling fluids
  T1000-like effects
Metaballs are cool!
The marching cubes algorithm

- A well-known method for scalar field polygonization
- Sample $f(x, y, z)$ on a cubic lattice
- For each cubic cell…
  - Estimate where isosurface intersects cell edges by linear interpolation
  - Tessellate depending on values of $f()$ at cell vertices
The marching cubes algorithm

- Each vertex can be either “inside” or “outside”.
- For each cube cell there are 256 ways for isosurface to intersect it.
  Can be simplified down to 15 unique cases.
Geometry shaders in DX10

From CPU

- Vertex Shader
- Geometry Shader
- Raster
- Stream Out
- Pixel Shader
- To Framebuffer
Implementation - basic idea

- App feeds a GPU with a grid of vertices
- VS transforms grid vertices and computes \( f(x, y, z) \), feeds to GS
- GS processes each cell in turn and emits triangles
A problem...

- Topology of GS input is restricted
  - Points
  - Lines
  - Triangles
  - with optional adjacency info

- Our “primitive” is a cubic cell
  Need to input 8 vertices to a GS
  A maximum we can input is 6 (with triangleadj)
Solution

- First, note that actual input topology is irrelevant for GS
  E.g. lineadj can be treated as quad input

- Work at tetrahedra level
  Tetrahedron is 4 vertices - perfect fit for lineadj!

- We’ll subdivide each cell into tetrahedra
Marching Tetrahedra (MT)

- Tetrahedra are easier to handle in GS
  - No ambiguities in isosurface reconstruction
  - Always output either 1 or 2 triangles
Generating a sampling grid

- There’s a variety of ways to subdivide
  - Along main diagonal into 6 tetrahedra – MT6
  - Tessellate into 5 tetrahedra – MT5
  - Body-centered tessellation – CCL

- Can also generate tetrahedral grid directly
  - AKA simplex grid
    - Doesn’t fit well within rectilinear volume
Sampling grids
# Sampling grids comparison

<table>
<thead>
<tr>
<th></th>
<th>Generation Complexity</th>
<th>Sampling Effectiveness</th>
<th>Regularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT5</td>
<td>Med</td>
<td>Med</td>
<td>Low</td>
</tr>
<tr>
<td>MT6</td>
<td>Low</td>
<td>Med</td>
<td>Low</td>
</tr>
<tr>
<td>CCL</td>
<td>High</td>
<td>High</td>
<td>Med</td>
</tr>
<tr>
<td>Simplex</td>
<td>Low</td>
<td>Med</td>
<td>High</td>
</tr>
</tbody>
</table>
VS/GS Input/output

// Grid vertex
struct SampleData
{
    float4 Pos : SV_POSITION;       // Sample position
    float3 N : NORMAL;              // Scalar field gradient
    float Field : TEXCOORD0;        // Scalar field value
    uint IsInside : TEXCOORD1;      // “Inside” flag
};

// Surface vertex
struct SurfaceVertex
{
    float4 Pos : SV_POSITION;       // Surface vertex position
    float3 N : NORMAL;              // Surface normal
};
// Metaball function
// Returns metaball function value in .w
// and its gradient in .xyz

float4 Metaball(float3 Pos, float3 Center, float RadiusSq)
{
    float4 o;

    float3 Dist = Pos - Center;
    float InvDistSq = 1 / dot(Dist, Dist);

    o.xyz = -2 * RadiusSq * InvDistSq * InvDistSq * Dist;
    o.w = RadiusSq * InvDistSq;

    return o;
}

Vertex Shader
Vertex Shader

#define MAX_METABALLS 32

SampleData VS_SampleField(float3 Pos : POSITION,
uniform float4x4 WorldViewProj,
uniform float3x3 WorldViewProjIT,
uniform uint NumMetaballs, uniform float4 Metaballs[MAX_METABALLS])
{
    SampleData o;
    float4 Field = 0;

    for (uint i = 0; i<NumMetaballs; i++)
        Field += Metaball(Pos, Metaballs[i].xyz, Metaballs[i].w);

    o.Pos = mul(float4(Pos, 1), WorldViewProj);
    o.N = mul(Field.xyz, WorldViewProjIT);
    o.Field = Field.w;
    o.IsInside = Field.w > 1 ? 1 : 0;

    return o;
}
Geometry Shader

// Estimate where isosurface intersects grid edge
SurfaceVertex CalcIntersection(SampleData v0, SampleData v1) {
    SurfaceVertex o;
    
    float t = (1.0 - v0.Field) / (v1.Field - v0.Field);
    o.Pos = lerp(v0.Pos, v1.Pos, t);
    o.N = lerp(v0.N, v1.N, t);
    return o;
}
Geometry Shader

`[MaxVertexCount(4)]
void GS_TesselateTetrahedra(lineadj SampleData In[4],
inout TriangleStream<SurfaceVertex> Stream) {
    // construct index for this tetrahedron
    uint index =
        (In[0].IsInside << 3) | (In[1].IsInside << 2) |
        (In[2].IsInside << 1) | In[3].IsInside;

    const struct { uint4 e0; uint4 e1; } EdgeTable[] = {
        { 0, 0, 0, 0, 0, 0, 0, 1 }, // all vertices out
        { 3, 0, 3, 1, 3, 2, 0, 0 }, // 0001
        { 2, 1, 2, 0, 2, 3, 0, 0 }, // 0010
        { 2, 0, 3, 0, 2, 1, 3, 1 }, // 0011 - 2 triangles
        { 1, 2, 1, 3, 1, 0, 0, 0 }, // 0100
        { 1, 0, 1, 2, 3, 0, 3, 2 }, // 0101 - 2 triangles
        { 1, 0, 2, 0, 1, 3, 2, 3 }, // 0110 - 2 triangles
        { 3, 0, 1, 0, 2, 0, 0, 0 }, // 0111
        { 0, 2, 0, 1, 0, 3, 0, 0 }, // 1000
        { 0, 1, 3, 1, 0, 2, 3, 2 }, // 1001 - 2 triangles
        { 0, 1, 0, 3, 2, 1, 2, 3 }, // 1010 - 2 triangles
        { 3, 1, 2, 1, 0, 1, 0, 0 }, // 1011
        { 0, 2, 1, 2, 0, 3, 1, 3 }, // 1100 - 2 triangles
        { 1, 2, 3, 2, 0, 2, 0, 0 }, // 1101
        { 0, 3, 2, 3, 1, 3, 0, 0 }  // 1110
    };
};
const struct { uint4 e0; uint4 e1; } EdgeTable[] = {
    // ...
    { 3, 0, 3, 1, 3, 2, 0, 0 }, // index = 1
    // ...
};

Index = 0001,
i.e. vertex 3 is “inside”
Geometry Shader

// ... continued
// don't bother if all vertices out or all vertices in
if (index > 0 && index < 15)
{
    uint4 e0 = EdgeTable[index].e0;
    uint4 e1 = EdgeTable[index].e1;

    // Emit a triangle
    Stream.Append(CalcIntersection(In[e0.x], In[e0.y]));
    Stream.Append(CalcIntersection(In[e0.z], In[e0.w]));
    Stream.Append(CalcIntersection(In[e1.x], In[e1.y]));

    // Emit additional triangle, if necessary
    if (e1.z != 0)
        Stream.Append(CalcIntersection(In[e1.z], In[e1.w]));
}
}
Respect your vertex cache!

- $f(x, y, z)$ can be arbitrary complex
  - E.g., many metaballs influencing a vertex
- Need to be careful about walk order
  - Worst case is 4x more work than necessary!
  - Straightforward linear work is not particularly cache friendly either
- Alternatively, can pre-transform with StreamOut
Respect your vertex cache!

- Can use space-filling fractal curves
  - Hilbert curve
  - Swizzled walk
- We’ll use swizzled walk
- To compute swizzled offset, just interleave x, y and z bits

\[
\begin{align*}
x &= x_1x_0 \\
y &= y_3y_2y_1y_0 \\
z &= z_2z_1z_0 \\
\text{swizzle}(x, y, z) &= y_3z_2y_2z_1y_1x_1z_0y_0x_0
\end{align*}
\]
Linear walk vs swizzled walk

Linear walk

Swizzled walk
Tessellation space

- **Object space**
  Works if you can calculate BB around your metaballs

- **View space**
  Better, but sampling rate is distributed inadequately
Tessellation in post-projection space

Post-projective space
Probably the best option
We also get LOD for free!
Problems with current approach

- Generated mesh is over-tessellated
  - General problem with MT algorithms
- Many triangles end up irregular and skinny
  - Good sampling grid helps a bit

(a) MT, smooth  (b) MT, triangles
Possible enhancements

- Regularized Marching Tetrahedra (RMT)
  Vertex clustering prior to polygonization
  Generated triangles are more regular
  For details refer to [2]

- Need to run a pre-pass at vertex level, looking at immediate neighbors
  For CCL, each vertex has 14 neighbors
  GS input is too limited for this 😞
More speed optimizations

- Can cull metaballs based on ROI
  Only 3 or 4 need to be computed per-vertex
- Can use bounding sphere tree to cull
  Re-compute it dynamically on a GPU as metaballs move
- Cool effect idea – particle system metaballs
  Mass-spring can also be interesting
Conclusion

- DX10 Geometry Shader can be efficiently used for isosurface extraction
- Allows for class of totally new cool effects
  - Organic forms with moving bulges
  - GPGPU to animate metaballs
  - Add noise to create turbulent fields
  - Terminator2 anyone?
References


Questions?

✉️ yuralsky@nvidia.com