DirectX10 Effects and Performance

Bryan Dudash
Today’s sessions

Now

DX10のエフェクトとパフォーマンスならび使用法

Bryan Dudash NVIDIA

16:50 – 17:00 BREAK

17:00 – 18:30 NVIDIA GPUでの物理演算

Simon Green NVIDIA
Motivation

- Direct3D 10 is Microsoft’s next graphics API
  - Driving the feature set of next generation GPUs
- Many new features
  - New programmability, generality
- New driver model
  - Improved performance
- Cleaned up API
  - Improved state handling. Almost no caps bits!
Agenda

- Short review of DX10 pipeline and features

Effect Case Studies
- Fins for fur – GPU Silhouette detection
- Cloth – GPU simulation in less passes
- Metaballs – GPU isosurface extraction

Conclusions
Direct3D 10 Pipeline

1. Input Assembler
2. Vertex Shader
3. Geometry Shader
4. Rasterizer
5. Pixel Shader
6. Output Merger
7. Video Memory
   (Buffer Texture, Constant Buffer)

Stream Output
Direct3D 10 New Features

- Common shader Instruction Set
  - Integer operations in shaders
  - Everything is indexable
- Geometry shader
- Stream out
- Render Target arrays
- Texture arrays
- 8 MRTs
- Input assembler generated identifiers; InstanceID, VertexID, PrimitiveID
- Alpha to Coverage
Coming up…

- Overview of Geometry Shader
  - Example using GS: fins
    - Some HLSL code showing the use of the GS
- Overview of Stream Out
  - Example using SO: Cloth
    - Some HLSL code showing the use of the SO
- Metaballs: Marching cubes
Geometry Shader

From CPU

Vertex Shader

Geometry Shader

Raster

Pixel Shader

Output Merger

input

Point
Line
Triangle
Line with adjacency
Triangle with adjacency

output

Point list
Line strip
Triangle strip
A quick note regarding GS

- If stream out is enabled
  - GS will output *lists*

- If stream out is disabled
  - GS will output *lists of strips*
Fur; generating fins on the GPU
Shells: problems at silhouettes

8 shells
+

fins
Silhouette detection on the GS

If (dot(eyeVec,N1) * dot(eyeVec,N2) < 0)
// GS shader for the fins
[maxvertexcount(12)]
void GS(triangleadj VS_OUTPUT input[6],
inout TriangleStream<GS_OUTPUT_FINS>TriStream)
{
    // compute the triangle’s normal
    float3 N1 = normalize(cross(input[0].Position - input[2].Position,
    float3 eyeVec = normalize(Eye - input[0].Position);

    // if the central triangle is front facing, check the other triangles
    if(dot(N1, eyeVec) > 0.0f) 
    {
        makeFin(input[2], input[0], input[1], TriStream);
        makeFin(input[4], input[2], input[3], TriStream);
        makeFin(input[0], input[4], input[5], TriStream);
    }
}
void makeFin( VS_OUTPUT v1, VS_OUTPUT v2, VS_OUTPUT vAdj, inout TriangleStream<GS_OUTPUT_FINS> TriStream )
{
    float3 eyeVec = normalize( Eye - v1.Position );

    if( dot(eyeVec,N2) < 0 )
    {
        //this is a silhouette edge, therefore extrude it into 2 triangles
        GS_OUTPUT_FINS Out;

        for(int v=0; v<2; v++)
        {
            Out.Position = mul(v2.Position + v*float4(v2.Normal,0)*length, WorldViewProj );
            Out.Normal = mul( v2.Normal, World );
            Out.TextureMesh = v2.Texture;
            Out.TextureFin = float2(1,1-v);
            Out.Opacity = opacity;
            TriStream.Append(Out);
        }
        TriStream.RestartStrip();
    }
}
Some more Geometry Shader Applications

- Silhouette detection and extrusion for:
  - Shadow volume generation
  - NPR

- Render to cubemap
  - single pass
  - in conjunction with Render Target arrays

- GPGPU
  - variable number of outputs from shader
Stream Out

- Allows storing output from geometry shader or vertex shader to buffer
- Enables multi-pass operations on geometry, e.g.
  - Recursive subdivision
  - Store results of skinning to buffer, reuse for multiple lights
- Can use DrawAuto() function to automatically draw correct no. of primitives
  - No CPU intervention required
Cloth on the GPU
Cloth as a Set of Particles

Each particle is subject to:
- A **force** (gravity, wind, drag, etc.)
- Various **constraints**:
  - To maintain overall shape (springs)
  - To prevent interpenetration with the environment (collision)

The constraints create a system of equations to be solved at each time step.

Use explicit integration: constraints are resolved by **relaxation**, that is by enforcing them one after the other for a given number of iterations.
Spring Constraints

Particles are linked by springs:

- Structural springs
- Shear springs

A spring is simulated as a **distance constraint** between two particles.
A distance constraint $DC(P, Q)$ between two particles $P$ and $Q$ is enforced by moving them away or towards each other:
Sequential update

Iterate as much as necessary
Parallel update

Batch 1

Diagram of parallel update patterns.
Parallel update

Batch 2
Parallel update

Batch 3
Parallel update

Batch 4
Pseudo-Code: Simulation Loop

Apply **force** to all particles

For as many times as necessary:
- For each batch of independent springs:
  - apply **distance constraints** to all the springs
  - apply **collision constraints**

**Render** mesh
DirectX10 Implementation

- Particles stored in a vertex buffer
  - DirectX9: particles would be stored in a texture

- Computation in **Geometry Shader**

- Synchronization between passes through **Stream Out**
Pseudo-Code: Initialization

- Create **two vertex buffers** to store the particles:
  - Vertex format is current position, old position, normal, etc.
  - One vertex buffer is used as input to the vertex shader
  - One vertex buffer is used as output to the geometry shader (Stream Output)
  - The two buffers are swapped after each rendering pass

- Create **as many index buffers as there are batches of independent springs (4)**
  - Each index buffer feeds the geometry shader with the right 4-tuples of particles

- Create an index buffer for rendering
Pseudo-Code: Simulation Loop

Set a vertex shader that applies **force**
Render to SO as a point list
Swap vertex buffers

For as many times as necessary:
- For each batch of independent springs:
  - Set a geometry shader that applies distance constraints
  - Render to SO as an indexed triangle list with adjacency
  - Swap vertex buffers
  - Set a vertex shader that applies collision constraints
  - Render to SO as a point list
  - Swap vertex buffers

Render to color buffer as indexed triangle list
Pseudo-Code: Simulation Loop

- Set a vertex shader that applies force
- Render to SO as a point list
- Swap vertex buffers
- For as many times as necessary:
  - For each batch of independent springs:
    - Set a geometry shader that applies **distance constraints**
    - Render to SO as an indexed triangle list with adjacency
    - Swap vertex buffers
  - Set a vertex shader that applies collision constraints
  - Render to SO as a point list
  - Swap vertex buffers
- Render to color buffer as indexed triangle list
Pseudo-Code: Simulation Loop

- Set a vertex shader that applies force
- Render to SO as a point list
- Swap vertex buffers
- For as many times as necessary:
  - For each batch of independent springs:
    - Set a geometry shader that applies distance constraints
    - Render to SO as an indexed triangle list with adjacency
    - Swap vertex buffers
  - Set a vertex shader that applies collision constraints
  - Render to SO as a point list
  - Swap vertex buffers
- Render to color buffer as indexed triangle list
Pseudo-Code: Simulation Loop

- Set a vertex shader that applies force
- Render to SO as a point list
- Swap vertex buffers
- For as many times as necessary:
  - For each batch of independent springs:
    - Set a geometry shader that applies distance constraints
    - Render to SO as an indexed triangle list with adjacency
    - Swap vertex buffers
  - Set a vertex shader that applies collision constraints
  - Render to SO as a point list
  - Swap vertex buffers
- Render to color buffer as indexed triangle list
Pseudo-Code: Simulation Loop

- Set a vertex shader that applies force
- Render to SO as a point list
- Swap vertex buffers
- For as many times as necessary:
  - For each batch of independent springs:
    - Set a geometry shader that applies distance constraints
    - Render to SO as an indexed triangle list with adjacency
    - Swap vertex buffers
    - Set a vertex shader that applies collision constraints
    - Render to SO as a point list
    - Swap vertex buffers
- Render to color buffer as indexed triangle list
Apply forces

pass ApplyForces
{
    SetVertexShader(CompileShader(vs_4_0, VS_ApplyForces()));
    SetGeometryShader(ConstructGSWithSO(CompileShader(vs_4_0, VS_ApplyForces()),
                                State.x; Position.xyz));
    SetPixelShader(0);
}

void VS_ApplyForces(inout Particle particle, OldParticle oldParticle)
{
    // Forces
    float3 force = 0;

    // Gravity
    force += float3(0, - GravityStrength, 0);

    // Damping
    float speedCoeff = 1 - TimeStep * 0.3;

    // Ignore position discontinuity (usually due to collision)
    float3 diffPosition = particle.Position - oldParticle.Position;
    if (dot(diffPosition, diffPosition) > 1 * TimeStep * TimeStep)
        speedCoeff = 0;

    // Integration step
    if (IsFree(particle))
        particle.Position += speedCoeff * diffPosition + force * TimeStep * TimeStep;
}
Metaballs on the GPU
What are Isosurfaces?

Consider a function \( f(x, y, z) \)

- Defines a *scalar field* in 3D-space
- Can come from procedural function, or 3D simulation

**Isosurface** \( S \) is a set of points which satisfy the implicit equation

\[
f(x, y, z) = \text{const}
\]

Also called *implicit* surfaces
Metaballs

- A simple and interesting case
- Soft/blobby objects that blend into each other
  - Perfect for modeling fluids, explosions in games

- Use implicit equation of the form
  \[
  \sum_{i=1}^{N} \frac{r_i^2}{\|x - p_i\|^2} = 1
  \]

- Gradient can be computed directly
  \[
  \text{grad}(f) = -\sum_{i=1}^{N} \frac{2 \cdot r_i^2}{\|x - p_i\|^4} \cdot (x - p_i)
  \]
The Marching Cubes Algorithm

- To render an isosurface we can either ray trace it or polygonalize it.
- Marching cubes: well-known method for polygonization of an isosurface.
- Sample $f(x, y, z)$ on a cubic lattice.
- Each vertex can be either “inside” or “outside”.
- Approximate the surface at each cube cell by a set of polygons.
For each cubic cell:
- If any edge connects a vertex that is in and one that is out, then the isosurface intersects that edge
- Estimate where isosurface intersects edge by linear interpolation
- Emit variable number of triangles depending on how many edges the surface intersects
Implementation - Pseudo-Code

App feeds a GPU with a grid of vertices

VS transforms grid vertices and computes $f(x, y, z)$, feeds cubes to GS

GS processes each cube in turn and emits triangles
Implementation - Pseudo-Code

App feeds a GPU with a grid of vertices

VS transforms grid vertices and computes $f(x, y, z)$, feeds cubes to GS

GS processes each cube in turn and emits triangles
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!
Implementation - Pseudo-Code

App feeds a GPU with a grid of vertices

VS transforms grid vertices and computes \( f(x, y, z) \), feeds cubes to GS

GS processes each cube in turn and emits triangles
Vertex shader

Calculate the following values for each vertex $v$:

- The Scalar field value
  \[ f(v) = \sum_{i=1}^{N} \frac{r_i^2}{\|v - p_i\|^2} \]

- A flag specifying whether the vertex is inside the field
  \[ Field = f(v) > 1 ? 1 : 0 \]

- The normal of the scalar field
- The projected position of the vertex
Implementation - Pseudo-Code

App feeds a GPU with a grid of vertices

VS transforms grid vertices and computes $f(x, y, z)$, feeds cubes to GS

GS processes each cube in turn and emits triangles
How do we get 8 vertices in the GS

We can read the value at a given index inside a vertex buffer directly from the Geometry Shader:

vertexValue = VertexBuffer.Load(index);

Can issue 8 such statements to fetch all vertices for a given cube
How do we get 8 vertices in the GS?

Pass 1

CPU
inputVertices

Vertex Shader

Float3: Position

TransformedVertices

GPU
Float4: Position
Float3: Normal
Float: Field

Stream Out

Pass 2

CPU

CubeIndices

Vertex Shader

Float3: Position

TransformedVertices

GPU
Float4: Position
Float3: Normal
Float: Field

Geometry Shader

Load()
Geometry Shader

[MaxVertexCount(16)]
void GS_TesselateCube(point CubePrimitive In[1], inout TriangleStream<SurfaceVertex> Stream)
{
    // 1. Construct index and load field data into temporaries
    uint index = 0;

    for (uint i = 0; i<8; i++)
    {
        // Construct bit field with a bit set for every vertex inside surface
        index |= SampleDataBuffer.Load( In[0].VertexIndex[i] ).Field > 1 ? 1 : 0;
        index <<= 1;
    }
}
Edge table construction

// StripCount contains number of triangle strips to generate for particular index value
const uint2 StripCount[256] = {
{ 0, //index = 0
  1, //index = 1
  // ...
};

// EdgeTable stores precomputed vertex indices for each cube edge which needs to be interpolated
const uint2 EdgeTable[256][16] = {
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 }, //index = 0
{ 7, 3, 7, 5, 7, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }, //index = 1
// ...
};

Index = 00000001,
i.e. vertex 7 is “inside”
Geometry Shader

// 2. Generate triangle strips according to "index" value

// Get number of triangle strips for this index
uint NumStrips = StripsCount[index];

// Emit that many triangle strips...
uint j = 0;
for (uint i = 0; i < NumStrips; i++)
{
    while (1)
    {
        uint2 edge = EdgeTable[index][j++];
        if (edge.x == edge.y) {    // edge.x == edge.y indicates a restart
            Stream.RestartStrip();
            break;
        }
        Stream.Append( CalcIntersection(
            SampleDataBuffer.Load( In[0].VertexIndex[edge.x] ),
            SampleDataBuffer.Load( In[0].VertexIndex[edge.y] )
        ) );
    }
}
Metaballs

The Geometry Shader can be efficiently used for isosurface extraction

- Marching cubes can also be used for visualization of medical data
- Allows for class of totally new cool effects
  - Organic forms with moving bulges
  - Modeling fluid like behavior in games (particle systems which model fluids)
  - GPGPU to animate metaballs
  - Add noise to create turbulent fields
Conclusions

- Offers new functionality that ... algorithms that used to only run on the cpu to run on the gpu:
  - metaballs,
  - Fins

- Increased flexibility ... Allows for easier+ more efficient implementation for other applications like gpgpu
  - Cloth
Bryan Dudash
bdudash@nvidia.com
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.

developer.nvidia.com