DirectX10 Effects

Sarah Tariq
Motivation

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

- Short review of DX10 pipeline and features
- Effect Case Studies
  - Fins for fur
  - Cloth
  - Metaballs
- Conclusions
Direct3D 10 New Features

- Geometry shader
- Stream out
- SM 4.0
  - Common shader Instruction Set
  - Integer operations in shaders
  - Load from VB/IB
  - Unlimited instruction count
- Resource Views
  - read depth/stencil buffer
  - Render to vertex buffer
- 8 MRTs
- Render Target arrays
- Texture arrays
- Input assembler generated identifiers; InstanceID, VertexID, PrimitiveID
- Alpha to Coverage
Coming up…

- **Overview of Geometry Shader**
  - Effect: fur
    - Some HLSL code showing the use of the GS
- **Overview of Stream Out**
  - Effect: Cloth
    - Some HLSL code showing the use of the SO
- **Effect: Rendering IsoSurfaces**
Geometry Shader

From CPU

Vertex Shader

Geometry Shader

Raster

Pixel Shader

Output Merger

input

Point
Line
Triangle
Line with adjacency

output

Triangle with adjacency

Point list
Line strip
Triangle strip
Fur; generating fins on the GPU
Shells and Fins Overview
Shells and Fins Overview
Shells and Fins Overview
Shells: problems at silhouettes

8 shells

→

fins
fins
Silhouette detection on the GS

\[
\text{if} \left( \text{dot}(\text{eyeVec}, \text{N1}) > 0 \ \&\& \ \text{dot}(\text{eyeVec}, \text{N2}) < 0 \right)
\]
// 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 this is a silhouette edge, extrude it into 2 triangles
    if( dot(eyeVec,N2) < 0 )
    {
        GS_OUTPUT_FINS Out;
        for(int v=0; v<2; v++)
        {
            Out.Position = mul(v1.Position + v*float4(v1.Normal,0)*length, WorldViewProj );
            Out.Normal = mul( v1.Normal, World );
            Out.TextureMesh = v1.Texture;
            Out.TextureFin = float2(0,1-v);
            Out.Opacity = opacity;
            //append new vertices to the stream to create the fin
            TriStream.Append(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
- Access to topology for calculating things like curvature
- Render to cube map in single pass
  - In conjunction with Render Target arrays
- GPGPU
  - Enables 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
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)
Cloth Simulation

- Apply **force** to all particles
- For as many times as necessary:
  - Apply **spring constraints**
  - Apply **collision constraints**
- **Render** mesh
Constraints

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

How do we update the springs? We could update them one at a time ...
Sequential update

But that’s too slow.
On the GPU we want to do as many updates in parallel as possible

However, we cannot update all springs in parallel; for example, the green spring cannot be updated at the same time as the red springs, since they affect the same particles.
Parallel update

The solution is to partition the springs into batches of independent sets, such that any given particle is only affected by one set in a batch.
Parallel update

Batch 1

Set 1
Set 2
Set 3
Set 4
Set 5
Set 6
Parallel update

Batch 2
Parallel update

Batch 3
Parallel update

Batch 4
Cloth Simulation

- Apply **force** to all particles
- Synchronize
- For as many times as necessary:
  - For all 4 batches
    - Apply **spring constraints**
    - Synchronize
  - Apply **collision constraints**
    - Synchronize
- Render mesh
DirectX10 Implementation

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

- Computation in **Geometry Shader** and **Vertex Shader**
  - DirectX9: computation in pixel shader

- Synchronization (between passes) through **Stream Out**
  - DirectX9: synchronization with writes to frame buffer and read from texture

- Cloth cutting by adding triangles
  - DirectX9: removing triangles

- Fewer passes in DirectX10: 4 vs. 8 in DirectX9
Apply Force: Vertex Shader

Each vertex gets a force applied to it
Satisfy distance constraints: Geometry Shader

The GS processes a set of vertices at a time.
For each set we satisfy each of the spring constraints.
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
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)
{
    // Apply Forces
    ...
    // Integrate
    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

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

\[ f(x, y, z) = \text{const} \]
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.
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” the isosurface.

$f(x, y, z) < 0.2$  $f(x, y, z) > 0.2$
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” the isosurface.
- Approximate the surface at each cube cell by a set of polygons.

The surface crosses these edges since they span vertices that are on different sides of the surface.
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” the isosurface.

Approximate the surface at each cube cell by a set of polygons.
The Marching Cubes Algorithm

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.
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
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!
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
  \[ Field = f(v) > 1 \Rightarrow 1 : 0 \]
- The normal of the scalar field
- The projected position of the vertex
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

Vertex Shader

float3: Position
inputVertices

GPU

TransformedVertices

Stream Out

Pass 2

CPU

Vertex Shader

uint VertexIndex[8]
cubeIndices

GPU

TransformedVertices

Geometry Shader

Load()
Geometry Shader

```cpp
[MaxVertexCount(16)]
void GS_TessellateCube(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, 0, 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”
// 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])
    ));
}
}
The Geometry Shader can be efficiently used for isosurface extraction

Allows for class of totally new cool effects
- Animating organic forms
- Modeling fluid like behavior in games (particle systems which model fluids)
- Add noise to create turbulent fields

Marching cubes can also be used for visualization of medical data
Conclusions

DirectX10 offers new functionality that enables the GPU to run algorithms that used to only run on the CPU

- Marching Cubes

Increased flexibility allows for easier and more efficient implementation for other applications like GPGPU

- Cloth
- Fins generation
Acknowledgments

NVIDIA Developer Technology Team
Yury Uralsky
Cyril Zeller
Cem Cebenoyan
Randy Fernando
Hubert Nguyen
Questions?

stariq@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