GPU TECHNOLOGY CONFERENCE

Tesselating NURBS with CUDA

Brent Oster, NVIDIA Devtech

NURBS

- Non-Uniform Rational B-Splines
- Curved surfaces commonly used in DCC / CAD modeling
- Points on surface defined by basis functions, CVs, weights

$$S(u,v) = \frac{\sum_{i=0}^{p}\sum_{j=0}^{q} B_{i,m}(u).B_{j,n}(v).P_{i,j}.w_{i,j}}{\sum_{i=0}^{p}\sum_{j=0}^{q} B_{i,m}(u).B_{j,n}(v).w_{i,j}}$$

- NURBS Basis $B_{i,k}$ defined recursively in terms of knot vector t_i

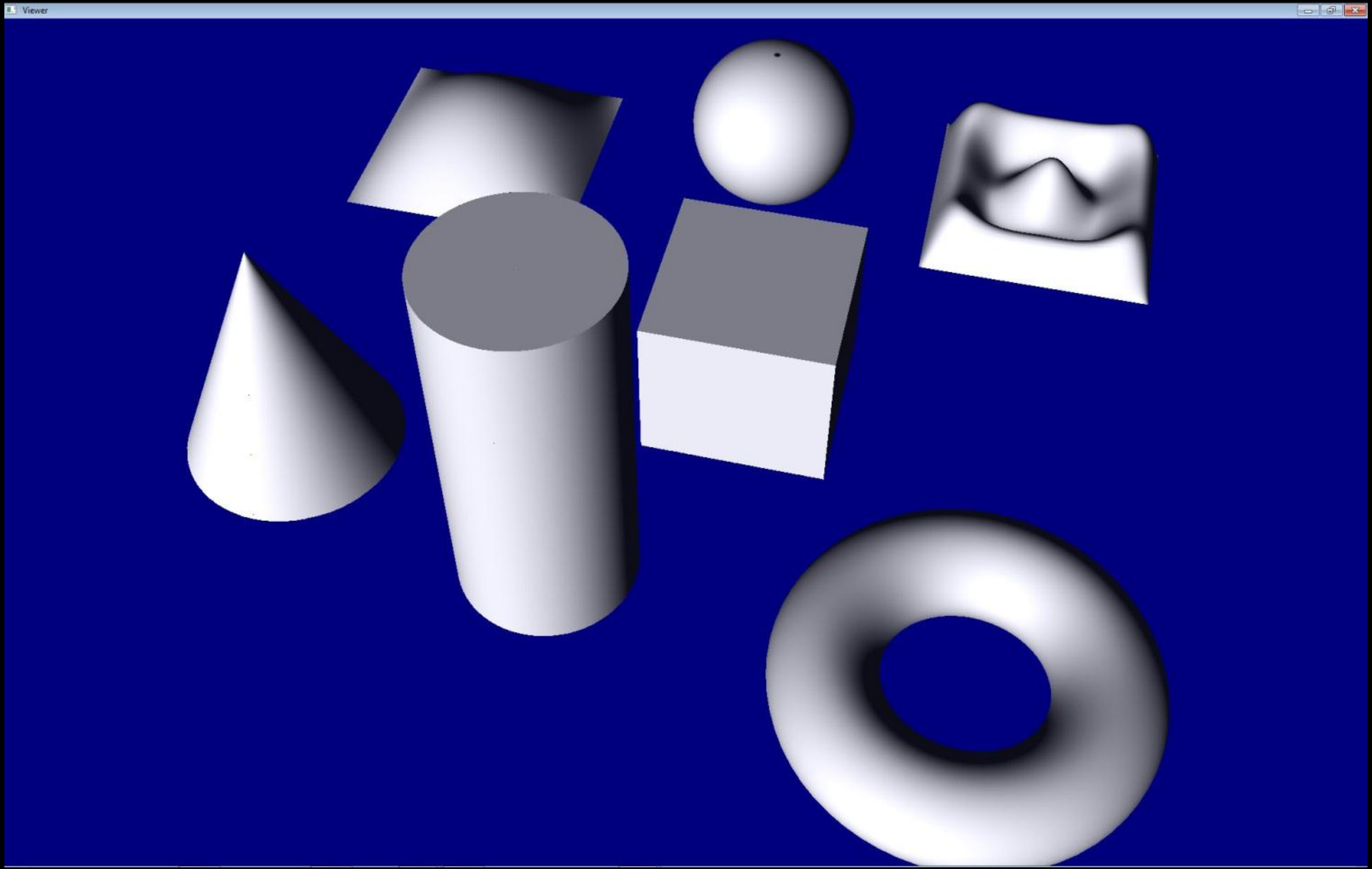
$$B_{i,0}(t) = \begin{cases} 1 \; if \;\; t_i \le t < t_{i+1} \\ 0 \;\; else \end{cases}$$

$$\forall k > 0, B_{i,k}(t) = \frac{t - t_i}{t_{i+k} - t_i} B_{i,k-1}(t) + \frac{t_{i+k+1} - t}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(t)$$

Tessellating NURBS with CUDA

- Efficient direct tessellation of NURBS surfaces
- Arbitrary order per surface
- Arbitrary knot vectors & number of patches
- Programmable UV tessellation patterns
- Programmable triangulation
- Enable trimmed surfaces (TBD)
- Write Pos, Norm, Indices to OpenGL VBOs
- VBO can then be re-used for multiple purposes

Directly Tessellating NURBS Surfaces



Tessellating NURBS Surfaces

CUDA Tessellation

- Input is an array of NURBS surfaces in device mem
  - Surface: CV's, UV knots, UV degree, boundary cond, …
- One NURBS surface handled per CUDA block
  1) Compute tessellation levels (separate kernel)
  2) Pre-compute polynomial coefficients on knot spans
  3) Compute custom (u,v) coordinates per vertex
  4) Compute vertex position, normal at each (u,v)
  5) Index through all quads, compute triangle indices

1) Compute Tessellation Levels

- Use CUDA Kernel to compute edge tessellation levels
- Simple formula for this demo
  - tessLevel = C*(Σ length(CP[i+1] – CP[i])) / distanceToCamera
  - i = 0 to (#CPs on edge)-1
  - C is user-defined constant
- Generates relatively constant triangle size on screen
- All vertices for all patches tessellated into one large VBO
- Must also compute unique pointers into VBO per patch
  - vertexIndex = atomicAdd(nTotalVertices, nVertsInSurface)
  - patch->vertexVBOptr = VBOStart + vertexIndex*sizeof(float4)

2) Pre-compute basis polynomial coefficients, store in shared memory

- NURBS basis functions expanded in polynomial series

$$B_{i,n}(t) = \sum_{k=0}^{n} C_{i,n,k}(t).t^k$$

- Pre-compute coefficients per knot span (indep of t)

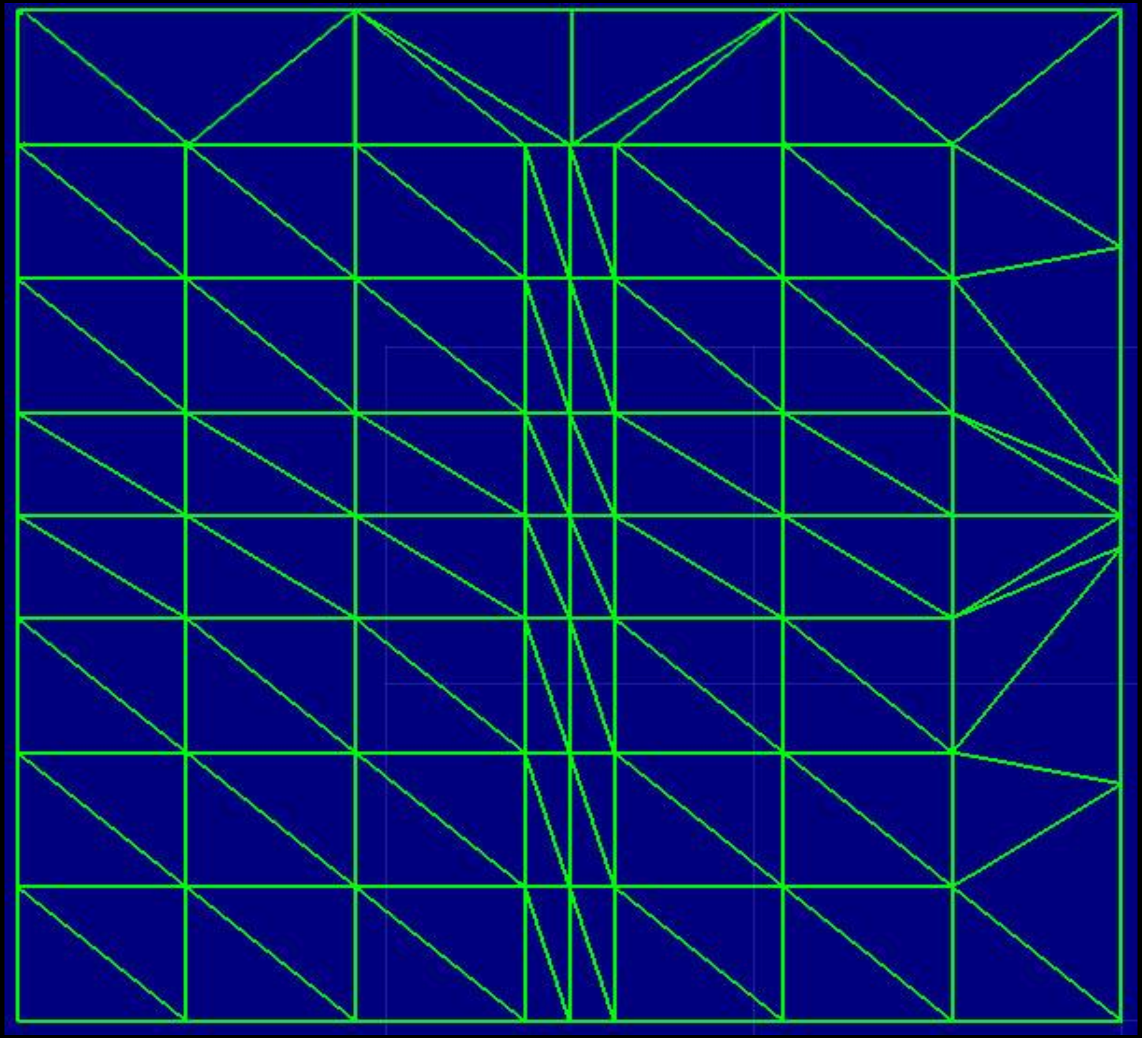
$$C_{i,0,0}(t) = B_{i,0}(t)$$

$$C_{i,n,0}(t) = \frac{t_{i+n+1}}{t_{i+n+1} - t_{i+1}} C_{i+1,n-1,0}(t) - \frac{t_i}{t_{i+n} - t_i} C_{i,n-1,0}(t)$$

$$C_{i,n,n}(t) = \frac{1}{t_{i+n} - t_i} C_{i,n-1,n-1}(t) - \frac{1}{t_{i+n+1} - t_{i+1}} C_{i+1,n-1,n-1}(t)$$

$$\forall k \in \{1..n-1\}, C_{i,n,k}(t) = \frac{C_{i,n-1,k-1}(t) - t_i.C_{i,n-1,k}(t)}{t_{i+n} - t_i} - \frac{C_{i+1,n-1,k-1}(t) - t_{i+n+1}.C_{i+1,n-1,k}(t)}{t_{i+n+1} - t_{i+1}}$$

3) Fractional, Symmetric U,V Tessellation



3) Compute U,V Tessellation Pattern

- Loop over all verts, increment i by blockDim.x

```
idx = i + threadIdx.x;
idxU = idx % nVertsU;
idxV = idx / nVertsU;
```

- Compute symmetric, fractional UV tessellation

```
u = u0 + (float)idxU*w;                            idxU < ceilTessU*0.5f - EPSILON
u = un - EPSILON - (ceilTessU - (float)idxU)*w;    idxU > ceilTessU*0.5f + EPSILON
u = u = u0 + 0.5f*(un - u0);                       otherwise
```

- For differing edge tess, make some vertices redundant:

```
idxU0 = (int)((ceil(edgeTess[0])/ceil(tessFactorU))*(float)idxU);
//compute u with idxU0 as above
```

4) Compute Vertex Positions and Normals

- Use pre-computed polynomial coefficients
- Compute vertex positions

$$S(u,v) = \frac{\sum_{i=0}^{p}\sum_{j=0}^{q} B_{i,m}(u).B_{j,n}(v).P_{i,j}.w_{i,j}}{\sum_{i=0}^{p}\sum_{j=0}^{q} B_{i,m}(u).B_{j,n}(v).w_{i,j}}$$

$$B_{i,n}(t) = \sum_{k=0}^{n} C_{i,n,k}(t).t^{k}$$

$$\frac{dB_{i,n}(t)}{dt} = \sum_{k=1}^{n} k.C_{i,n,k}(t).t^{k-1}$$

- Use Horner's rule to efficiently evaluate polynomials
- Compute U,V tangent vectors -> vertex normals
  - Use polynomial coefficients to compute derivatives

5) Compute Triangulation Indices

- Compute # of quads in surface
- Loop over all quads, increment i by blockIdx.x
- idx = i + threadIdx.x
- Compute **idxU** = idx % nQuadsU, **idxV** = idx / nQuadsU
- Compute indices for triangles

index[idx*6] = offset + **idxU** + **idxV***nVerticesU;

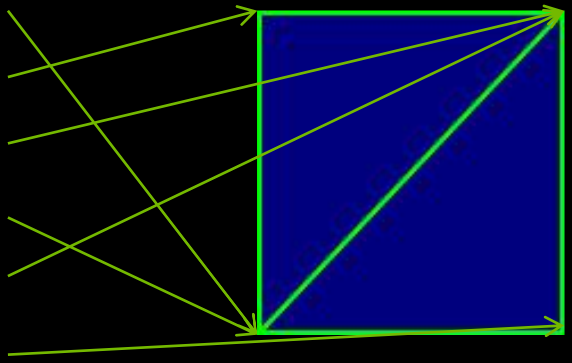
index[idx*6+1] = offset + **idxU** + (**idxV+1**)*nVerticesU

index[idx*6+2] = offset + **idxU+1** + (**idxV+1**)*nVerticesU;

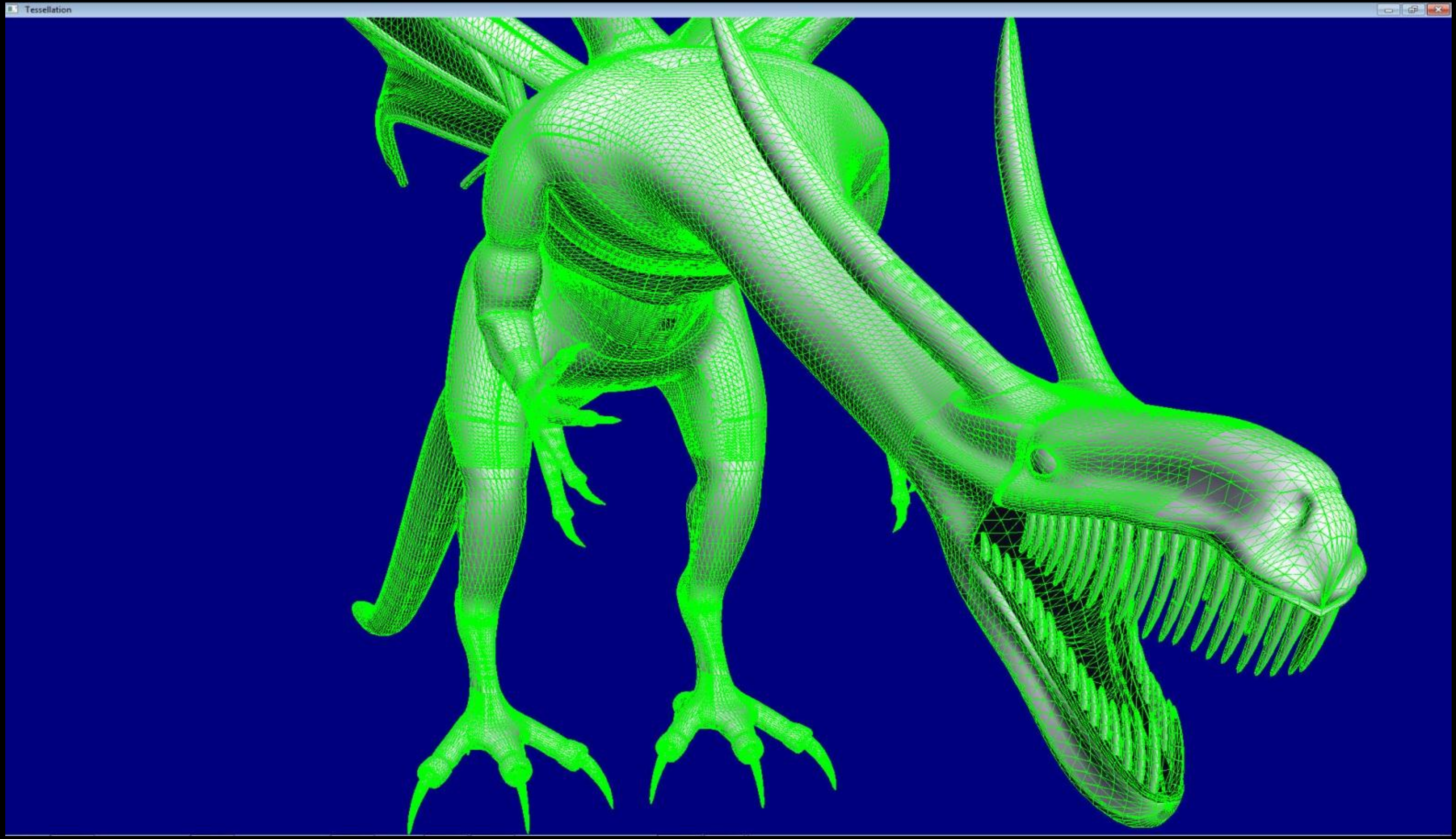
index[idx*6+3] = offset + **idxU** + **idxV***nVerticesU;

index[idx*6+4] = offset + **idxU+1** + (**idxV+1**)*nVerticesU;

index[idx*6+5] = offset + **idxU+1** + **idxV***nVerticesU;



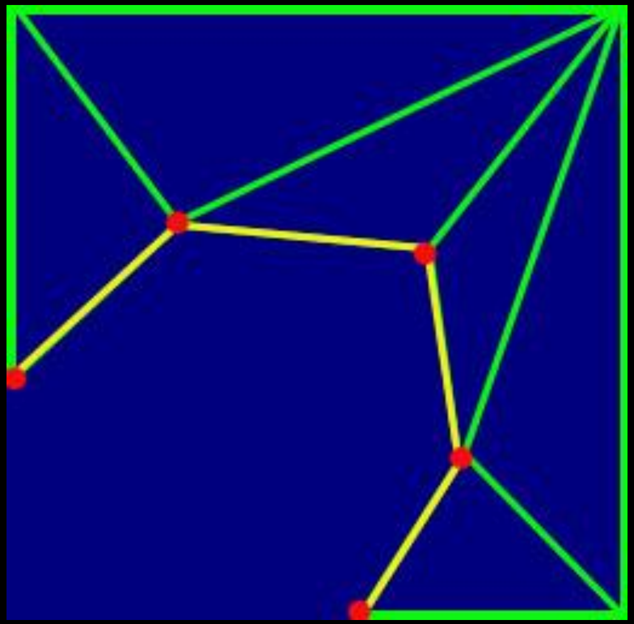
Results (438 surf, 3.5M tri in 19.2 ms)



Performance Optimizations

- Set **cudaFuncSetCacheConfig(**cudaFuncCachePreferShared)
  - Increases the shared memory available to 48 KB
  - Increases occupancy by allowing more blocks to run simultaneously
- Launch with blockDim (128,1,1)
  - Empirically tested to be the most efficient configuration
- Set max regcount to 32
  - Spills more registers to L1
  - Increases occupancy

Trimmed Surfaces



Questions?

- Feel free to contact me:
  boster@nvidia.com

Samples will be posted after GTC