MOTIVATION

Blueprints / drawings in CAD/graph viewer applications

Documents can contain many LINES and LINE_STRIPS

Various line styles can be used (world-space widths, stippling, joints, caps...)

Potential CPU bottlenecks

- Generating geometry for complex styles
- Collecting and rendering geometry

Model courtesy of PTC
**MOTIVATION**

Not targeting full vector graphics

**NV_path_rendering** covers high fidelity vector graphics rendering

Per-pixel quadratic Bézier evaluation

Stencil & Cover pass to allow sophisticated blending

**Focus of this talk** is rendering lines defined by traditional vertices

Rendering data from OpenGL buffer objects

Single-pass, but does mean not safe for blending (does self-overlap)
DEMO: BASIC DEMONSTRATION
LINE RASTERIZATION

Representation

Standard: skewed rectangle pixel snapped lines

Multisampling: aligned rectangle smooth lines

Both suffer from visible gaps and overlaps on increasing line width
LINE RASTERIZATION

Stippling

Stippling only in screenspace

Patterns must be expressable with 16 bits

LINES re-start pattern every segment

LINE_STRIPs have continuous distance
Create TRIANGLES/QUADS for line segments

Project extruded vertices to keep line width consistent

Clip and color in fragment shader based on UV coordinates and line distance

Appearance on screen

Geometry in world coordinates

Shapes via fragment shader discard
SHADER-DRIVEN LINES

TECH
Create TRIANGLES for line segments, project extrusion to world/screen, discard fragments

FLEXIBILITY
Arbitrary stippling patterns and line widths
Joint- and cap-styles
Different distance metrics
New coloring/animation possibilities via shaders

Thin center line as effect
SHADER-DRIVEN LINES

TECH
Create TRIANGLES for line segments, project extrusion to world/screen, discard fragments

FLEXIBILITY
Arbitrary stippling patterns and line widths
Joint- and cap-styles
Different distance metrics
New coloring/animation possibilities via shaders

CAVEATS
Cannot be as fast as basic line rasterization
Not all data local at rendering time (line strip distances need extra calculation)
Geometry still self-overlaps
C interface library to render different line primitives (LINES, LINE_STRIPS, ARCS) provided as flexible framework rather than black-box

Two different render-modes: render as extruded triangles, or one pixel wide lines

Uses NVIDIA and ARB OpenGL extensions if available
**SHADER-DRIVEN LINES**
Sample implementation/library

Global style and stipple definitions

Stipple from arbitrary bit-pattern, or float values

```c
typedef struct NVLStyleInfo_s {
    NVLSpaceType projectionSpace;
    NVLJoinType join;
    NVLCapsType capsBegin;
    NVLCapsType capsEnd;
    float thickness;
    NVLStippleID stipplePattern;
    float stippleLength;
    float stippleOffsetBegin;
    float stippleOffsetEnd;
    NVLAnchorType stippleAnchor;
    NVLboolean stippleClamp;
} NVLStyleInfo;

typedef enum NVLSpaceType_e {
    NVL_SPACE_SCREEN,
    NVL_SPACE_SCREENDIST3D,
    NVL_SPACE_CUSTOM,
    NVL_SPACE_CUSTOMDIST3D,
    NVL_NUM_SPACES,
} NVLSpaceType;

typedef enum NVLJoinType_e {
    NVL_JOIN_NONE,
    NVL_JOIN_ROUND,
    NVL_JOIN_MITER,
    NVL_NUM_JOINS,
} NVLJoinType;

typedef enum NVLCapsType_e {
    NVL_CAPS_NONE,
    NVL_CAPS_ROUND,
    NVL_CAPS_BOX,
    NVL_NUM_CAPS,
} NVLCapsType;

typedef enum NVLAnchorType_e {
    NVL_ANCHOR_BEGIN,
    NVL_ANCHOR_END,
    NVL_ANCHOR_BOTH,
    NVL_NUM_ANCHORS,
} NVLAnchorType;

typedef enum NVLStippleID_e {
    NVL_STIPPLE_A,
    NVL_STIPPLE_B,
    NVL_NUM_STIPPLES,
} NVLStippleID;
```

Stipple-Definitions

<table>
<thead>
<tr>
<th>Style</th>
<th>Pattern texture A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Pattern texture B</td>
</tr>
<tr>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

Stipple-Patterns
Uses GPU friendly collection mechanism: Record many primitives then render Optionally render sub-sections

Raw Primitives pass vertex data directly

Geometry Primitives reference existing Vertex Buffers

Collections have usage-style flags:
- filled new per-frame
- recorded once, re-used many frames
SHADER-DRIVEN LINES

Quad extrusion

Faster geometry creation by just using Vertex-Shader, avoiding extra Geometry-Shader stage

Render GL_QUADS (4 vertices each segment)

Use gl_VertexID to fetch line points

Use it for the offsets as well

Using custom vertex-fetch generally not recommended, but useful for special situations

---

VertexBuffer

texelFetch(...gl_VertexID/4 + 0 or 1)

gl_VertexID % 4 + 0  gl_VertexID % 4 + 1
SHADER-DRIVEN LINES

Minimize Overdraw

No naive rectangles but adjacency in LINE_STRIP is used to tighten the geometry

Reduces overdraw and minimizes potential artifacts resulting from that
SHADER-DRIVEN LINES

Depth clamping

Joints and caps exceed original line definition

Can cause depth-buffer artifacts

Prevent depth over-shooting by passing closest depth to fragment shader and clamp there

Can use ARB_conservative_depth or just min/max to keep hardware z-cull active

```glsl
#extension GL_ARB_conservative_depth : require
layout (depth_greater) out float gl_FragDepth;

in flat float closestPointDepth;
...

gl_FragDepth = max(gl_FragCoord.z, closestPointDepth);
```
LINE_STRIPS need dedicated calculation phase

Read vertices and calculate distances along the strip

DistanceBuffer D

Distances are fetched at render-time
One `LINE_STRIP` per thread can lead to under utilization and non ideal memory access due to divergence.

SIMT hardware processes threads together in lock-step, common instruction pointer (masks out inactive threads). NVIDIA: 1 warp = 32 threads.
DISTANCE COMPUTATION

Shader Tips

**Compute** one LINE_STRIP at a time **across** warp, gives nice memory fetch

**NV_shader_thread_shuffle** to access neighbors and do prefix-sum calculation

```cpp
vec3 posA = getPosition(gl_ThreadInWarpNV + ...)
vec3 posB = shuffleUpNV(posA, 1, gl_WarpSizeNV);
... Handle first thread point differently
float dist = distance(posA, posB);
```

Short strips may still under-utilize warp, but are taking only one iteration

---

<table>
<thead>
<tr>
<th>Thread: 0 ... 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strip Length</td>
</tr>
<tr>
<td>9 9 9 9</td>
</tr>
</tbody>
</table>

**VertexBuffer**

**Distance Accumulation Loop**

Access neighbor point via `shuffleUpNV` and compute distance

... Prefix-sum over distances ...

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
Memory intensive operations prefer **many threads to hide latency** of fetch.

Would not „compute“ distance for a single strip, but need many strips to work on.

**Use one warp per strip** if total amount of threads is low.

**Hardware switches activity between entire warps**
DISTANCE COMPUTATION
Batching & Latency hiding

Launch overhead of compute dispatch not negligible for < 10 000 threads

Use glEnable(GL_RASTERIZER_DISCARD); and Vertex-Shader to do compute work

No shared memory but warp data sharing as seen before (ARB_shader_ballot or NV_shader_thread_shuffle)

... “Compute” alternative for few threads
if (numThreads < FEW_THREADS){
  glUseProgram( vs );
  glEnable ( GL_RASTERIZER_DISCARD );
  glDrawArrays( GL_POINTS, 0, numThreads );
  glDisable ( GL_RASTERIZER_DISCARD );
} else {
  glUseProgram( cs );
  numGroups = (numThreads+GroupSize-1)/GroupSize;
  glUniformi1(0, numThreads);
  glDispatchCompute ( numGroups, 1, 1 );
}

... Shader
#if USE_COMPUTE
    layout (local_size_x=GROUP_SIZE) in;
    layout (location=0) uniform int numThreads;
    int threadID = int( gl_GlobalInvocationID.x );
#else
    int threadID = int( gl_VertexID );
#endif
SMOOTH TRANSITIONS
Anti-aliasing edges within shader

Fragment shader effects cause outlines of visible shapes to be within geometry

MSAA will not add quality „within triangle“

Need to compute coverage accurately (sample-shading) or approximate

Use of gl_SampleID (e.g. with interpolateAtSample) automatically makes shader run per-sample, „discard“ will affect coverage mask properly

Cheaper: GL_SAMPLE_ALPHA_TO_COVERAGE or clear bits in gl_SampleMask

No geometric edges → No MSAA benefit

in float stippleCoord;
...
sc = interpolateAtSample (stippleCoord, gl_SampleID);
stippleResult = computeStippling( sc );
if (stippleResult < 0) discard;
SMOOTH TRANSITIONS
Using Pixel Derivatives

Simple trick to get smooth transitions, also works well on surface contour lines

Use a signed distance field, instead of step function

Find if sample is close to transition (zero crossing) via fwidth

Compute smooth weight if required

```c
float weight = signal < 0 ? -1 : 1;
float zone = fwidth(signal) * 0.5;
if (abs(signal) < zone){
    weight = signal / zone;
}
```
RECORDING RAW DATA
Using persistent mapped buffers

When primitives & vertices are not re-used, but regenerated by CPU, we want a fast way to get them to GPU

Use `ARB_buffer_storage/OpenGL 4.3` to have buffers in CPU memory for fast copying

Need fences to avoid overwriting data still used by GPU, 3 frames typically enough to avoid synchronization

CPU memory access „okayish“ if data only read rarely (once for stipple-compute, once for render)
RENDEING ARCS

Not trivial to compute distance along an arbitrary projected arc/circle

Approximate circle as line strip

Allocate maximum subdivision

Compute adaptively based on screen-space size (or frustum cull)

Rendering only needs to fetch distance values, can still compute position on the fly
Preserving all primitive order not optimal for performance, ideally application can operate in layers.

Code your own special primitives for annotations (arrows...)

Use of shaders can increase visual quality beyond „fancy surface shading“

Do not need actual geometry for everything (distance fields are great)

GPU programmable enough to move more effects from CPU to GPU
THANK YOU

JOIN THE CONVERSATION
#GTC16  

JOIN THE NVIDIA DEVELOPER PROGRAM AT developer.nvidia.com/join

ckubisch@nvidia.com @pixeljetstream