GPU-Centered Font Rendering
Directly from Glyph Outlines

Eric Lengyel, Ph.D.
Terathon Software
About the speaker

- Working in game/graphics dev since 1994
  - Previously at Sierra, Apple, Naughty Dog

- Current projects:
  - Slug Library, C4 Engine, The 31st, FGED
About this talk

- Technical details about the “Slug” font rendering algorithm
- Paper published in JCGT in June 2017
- New developments in past year
Font Rendering Ubiquity

- Text rendered everywhere in 3D applications
  - GUI: Buttons, checkboxes, lists, menus, ...
  - Games: Score, health, ammo, ...
  - In scene: Signs, labels, computer screens, ...
  - Debug info: Console, stats, timings, ...
GPU-Centered Font Rendering Directly from Glyph Outlines

TUNNEL 4-A
Font Rendering Design Goals

- Unified approach
  - Same technique used to render all text in all situations

- Runs in shader on GPU
  - Fully dynamic, but also allows caching
  - Can be combined with other materials
Font Rendering Design Goals

- Total resolution independence
  - No precomputed images or distance fields

- Ability to render with any transform
  - Arbitrary scale, rotation, perspective

- Must look good at large and small font sizes
Font Rendering Design Goals

- Minimal triangulation
  - Don’t want lots of small triangles
  - GPUs perform best with large triangles
  - A fixed per-glyph vertex count is desirable
  - Would like to be able to easily clip text
  - Would like to apply text to curved surfaces
Rendering Algorithm Priorities

1. Works correctly
2. Looks good
3. Runs fast
Glyphs in TrueType

- Glyph defined by one or more closed contours
- Each contour composed of continuous sequence of quadratic Bézier curves
- Each Bézier curve has three control points
GPU-Centered Font Rendering Directly from Glyph Outlines
Glyph Space

- Glyphs are defined on em square
- Coordinates in range [0,1] inside em square
- Curves can extend outside em square
GPU-Centered Font Rendering Directly from Glyph Outlines

Glyph Space

- **em square**
- **bounding box**
- **advance width**
- Point **(0,0)**
- Point **(1,1)**
Winding Number

- To determine whether point inside glyph, calculate \textit{winding number} with respect to each contour and sum

- Point inside glyph outline if sum of winding numbers is nonzero
Winding Number

- The winding number is the count of complete loops a contour makes around a point.

- One direction (arbitrary, either CW or CCW) is considered positive, and opposite direction is then considered negative.
Winding Number

- To calculate winding number, fire a ray from point being rendered to infinity
- Direction doesn’t matter, so use +x direction for convenience
- Look for contour intersections along the ray
Winding Number

- When contour crosses ray from left to right, increment winding number
- When contour crosses ray from right to left, decrement winding number
- Or other way around, as long as consistent
GPU-Centered Font Rendering Directly from Glyph Outlines
Quadratic Bézier Curve

- Three control points \( p_1, p_2, p_3 \)

- Parametric curve with \( 0 \leq t \leq 1 \):

\[
C(t) = (1-t)^2 p_1 + 2t(1-t)p_2 + t^2 p_3
\]
Ray-Curve Intersection

- Translate control points so ray origin is (0,0)
- Assume ray direction is +x axis
- Solve for values of $t$ where $y$ coordinate of Bézier curve is zero
Ray-Curve Intersection

- Let \( p_i = (x_i, y_i) \)

- Ray intersects curve at roots of polynomial

\[
(y_1 - 2y_2 + y_3) t^2 - 2(y_1 - y_2) t + y_1
\]

\[
a = y_1 - 2y_2 + y_3 \quad b = y_1 - y_2 \quad c = y_1
\]
Ray-Curve Intersection

- Roots $t_1$ and $t_2$ given by

$$t_1 = \frac{b - \sqrt{b^2 - ac}}{a} \quad t_2 = \frac{b + \sqrt{b^2 - ac}}{a}$$

- If $a$ near zero, use root of linear polynomial:

$$t_1 = t_2 = \frac{c}{2b}$$
Ray-Curve Intersection

- **Valid intersection at** $t_i$ **when:**
  - $0 \leq t_i \leq 1$ (between curve endpoints)
  - $C_x(t_i) \geq 0$ (at positive distance along ray)

- $t_i = 1$ specifically disallowed
  - Corresponds to intersection at $t_i = 0$ on next Bézier curve, and don’t want to count twice
Ray-Curve Intersection
Ray-Curve Intersection

- Increment or decrement winding number?
- Look at y values in range $0 \leq t_i \leq 1$
  - Positive before $t_i$ or negative after $t_i$: increment
  - Negative before $t_i$ or positive after $t_i$: decrement
- Can’t rely on derivative
  - Zero if ray tangent to curve
Robustness

- Sound from purely mathematical standpoint
- But plagued by numerical precision errors!
- Floating-point limits cause huge problems for roots near endpoints where $t_i = 0$ or $t_i = 1$
Numerical Precision Errors

- Produce sparkle and streak artifacts
- Hacks like epsilons and coordinate perturbation just shift problem cases around
- Need something that’s 100% robust
Slug Algorithm

- Calculates winding number
  - Input is arbitrary set of closed contours composed of quadratic Bézier curves

- Performs antialiasing
  - Determines fractional coverage at each pixel
Priority #1: Works Correctly

- Robust for all valid inputs
  - Meaning any floating-point coordinates that are not infinity or NaN

- No distortion of glyph outlines

- No sparkle artifacts
Equivalence Class Algorithm

- Infinite problem space reduced to a finite number of equivalence classes
- Same procedure followed for all cases in each equivalence class
- Same abstract idea as Marching Cubes
Bézier Curve Classification

- Look at y coordinates of the three control points
- Each positive, negative, or zero
- 27 classes based on these states
Bézier Curve Classification

- It turns out we can do better than 27 classes
- Classify each control point based on whether $y$ coordinate is nonnegative or negative
- Only 8 equivalence classes
Bézier Curve Classification

- Roots (ray intersections) always occur in same way for all cases in each class
- We care about places where curve transitions between nonnegative and negative
- Only have to decide how to modify winding number for each root
Winding Number Modification

- Consider derivative of $y$ coordinate:

$$y'(t) = 2at - 2b$$

$$a = y_1 - 2y_2 + y_3$$

$$b = y_1 - y_2$$
Winding Number Modification

- An observation about the roots for nonzero discriminant $D$:

\[ t_1 = \frac{b - \sqrt{D}}{a} \quad t_2 = \frac{b + \sqrt{D}}{a} \]

\[ D = b^2 - ac \]

\[ y'(t_1) = -2\sqrt{D} \quad y'(t_2) = +2\sqrt{D} \]
Winding Number Modification

- Root at $t_1$ always crosses ray from left to right
  - Going from nonnegative to negative
  - Always increment winding number

- Root at $t_2$ always crosses ray from right to left
  - Going from negative to nonnegative
  - Always decrement winding number
Winding Number Modification

- We can also incorporate cases where ray intersects an endpoint tangentially.
- Winding number modified only when transition between nonnegative and negative occurs.
- $x$ coordinate at transition must be positive.
GPU-Centered Font Rendering Directly from Glyph Outlines

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Graph A" /></td>
<td><img src="image2" alt="Graph B" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3" alt="Graph C" /></td>
<td><img src="image4" alt="Graph D" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image5" alt="Graph E" /></td>
<td><img src="image6" alt="Graph F" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image7" alt="Graph G" /></td>
<td><img src="image8" alt="Graph H" /></td>
</tr>
</tbody>
</table>
Class A: All Nonnegative

- Nothing happens to winding number
Class H: All Negative

- Nothing happens to winding number
Classes B and D: One Transition

- Winding number decremented if \( x(t_2) > 0 \)
Classes E and G: One Transition

- Winding number incremented if $x(t_1) > 0$
Class C: Two Transitions

- Winding number incremented if $x(t_1) > 0$
- Winding number decremented if $x(t_2) > 0$
Class F: Two Transitions

- Winding number incremented if $x(t_1) > 0$
- Winding number decremented if $x(t_2) > 0$
Discriminant Clamping

- In classes C and F, we could have a negative discriminant $D$

- To handle with uniformity, clamp $D$ to zero

- Always two transitions at same $x$ coordinate, so guaranteed to cancel each other out
Root Calculation

```c
float2 SolvePoly(float4 p12, float2 p3) {
    float2 a = p12.xy - p12.zw * 2.0 + p3;    // Calculate coefficients.
    float2 b = p12.xy - p12.zw;
    float ra = 1.0 / a.y;
    float rb = 0.5 / b.y;

    float d = sqrt(max(b.y * b.y - a.y * p12.y, 0.0));    // Clamp discriminant to zero.
    float t1 = (b.y - d) * ra;
    float t2 = (b.y + d) * ra;

    if (abs(a.y) < epsilon) t1 = t2 = p12.y * rb;    // Handle linear case where |a| \neq 0.

    // Return x coordinates at t1 and t2.
    return (float2((a.x * t1 - b.x * 2.0) * t1 + p12.x,
                   (a.x * t2 - b.x * 2.0) * t2 + p12.x));
}
```
Root Eligibility

- We know what to do for each root
- Just need to decide whether to actually do it!
- Use a lookup table for root eligibility
Root Eligibility

- Nonnegative/negative gives us 3-bit state
  - Just use sign bits of $y$ coordinates

- Look up 2-bit root eligibilities for $t_1$ and $t_2$

- Total LUT size is a tiny 16 bits
# Root Eligibility Lookup Table

<table>
<thead>
<tr>
<th>Class</th>
<th>$y_3 &lt; 0$</th>
<th>$y_2 &lt; 0$</th>
<th>$y_1 &lt; 0$</th>
<th>Root 2</th>
<th>Root 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

0x2E 0x74
Calculating Root Codes

```c
uint CalcRootCode(float y1, float y2, float y3)
{
    uint i1 = asuint(y1) >> 31U;
    uint i2 = asuint(y2) >> 30U;
    uint i3 = asuint(y3) >> 29U;

    uint shift = (i2 & 2U) | (i1 & ~2U);
    shift = (i3 & 4U) | (shift & ~4U);

    return ((0x2E74U >> shift) & 0x0101U);
}
```

```c
bool TestCurve(uint code)
{
    return (code != 0U);}
```

```c
bool TestRoot1(uint code)
{
    return ((code & 1U) != 0U);}
```

```c
bool TestRoot2(uint code)
{
    return (code > 1U);}
```
Ternary Logic Instruction

- Recent Nvidia GPUs have a ternary logic instruction (LOP3)
- Maps arbitrary 3-bit input to 1-bit output with 8-bit lookup table encoded in the instruction
- Perfect fit for our algorithm!
Ternary Logic Instruction

- Instruction not directly accessible from pixel shader code

- Compiler can recognize arbitrary sequence of AND, OR, XOR, NOT operations and generate single ternary instruction
Ternary Logic Instruction

• With LOP3, root code calculation requires only 2 instructions where otherwise need at least 7

• Shader gets about 4% faster in all cases
Root Codes with Ternary Logic

```c
int2 CalcRootCode(float y1, float y2, float y3) {
    int a = asint(y1);
    int b = asint(y2);
    int c = asint(y3);

    return (int2(~a & (b | c) | (~b & c), a & (~b | ~c) | (b & ~c)));
}

bool TestCurve(int2 code) {
    return ((code.x | code.y) < 0);
}

bool TestRoot1(int2 code) {
    return (code.x < 0);
}

bool TestRoot2(int2 code) {
    return (code.y < 0);
}
```
Total Winding Number

```c
int winding = 0;
for (all Bézier curves)
{
    float4 p12 = first (.xy) and second (.zw) control points
    float2 p3 = third (.xy) control point

    code = CalcRootCode(p12.y, p12.w, p3.y);
    if (TestCurve(code))
    {
        float2 r = SolvePoly(p12, p3);

        if ((TestRoot1(code)) && (r.x > 0.0)) winding += 1;
        if ((TestRoot2(code)) && (r.y > 0.0)) winding -= 1;
    }
}

if (winding != 0) then pixel is inside glyph outline
```
Priority #2: Looks Good

- Performs accurate antialiasing
- Handles arbitrary transforms well
- Handles minification well
Fractional Coverage

- Integer winding number produces simple in/out state for each pixel

- Correct, but has jagged edges everywhere

- We need fractional pixel coverage values
GPU-Centered Font Rendering Directly from Glyph Outlines
Fractional Coverage

- Ray origin is at pixel center

- Previously, we incremented or decremented winding number when \( x(t_i) > 0 \)

- Now, we add or subtract the fractional distance ray makes it through pixel before intersection
Fractional Coverage

- Let $u$ be number of pixels per em
  - Scales coordinates so that width of pixel = 1 unit

- Always change winding number (WN) by

  $\text{saturate} \left( u \cdot x(t_i) + \frac{1}{2} \right)$
Fractional Coverage

- If $u \cdot x(t_i) \leq -0.5$, then no change to WN
- If $u \cdot x(t_i) \geq 0.5$, then WN always changed by 1
- In between, WN changed by fractional value
- Accounts for multiple curves per pixel
GPU-Centered Font Rendering Directly from Glyph Outlines
float coverage = 0.0;
for (all Bézier curves)
{
    float4 p12 = first (.xy) and second (.zw) control points
    float2 p3 = third (.xy) control point

code = CalcRootCode(p12.y, p12.w, p3.y);
if (TestCurve(code))
{
    float2 r = SolvePoly(p12, p3) * pixelsPerEm;

    if (TestRoot1(code)) coverage += saturate(r.x + 0.5);
    if (TestRoot2(code)) coverage -= saturate(r.y + 0.5);
}
}
Antialiasing

- This gives us excellent 1D antialiasing
  - Looks great when curves are mostly vertical

- Doesn’t work well for mostly horizontal curves

- So fire a ray in the y direction, too
  - Looks great when curves are mostly horizontal
Antialiasing

- Calculate coverage for two rays at each pixel
  - One in $x$ direction, and one in $y$ direction
  - Note pixels per em could be different in $x$ and $y$

- Combine two coverage values for good 2D antialiasing
  - Lots of ways to calculate weighted average
Antialiasing

- Output is linear coverage value
- Works best when blended into sRGB framebuffer
Bounding Box Dilation

- Draw one quad per glyph coinciding with glyph’s bounding box
- GPU fills pixels with centers covered by quad
- Could miss pixels on boundary with up to 50% fractional coverage value
GPU-Centered Font Rendering Directly from Glyph Outlines
Bounding Box Dilation

- Must dilate bounding box by half pixel width

- In em space, dilate by $0.5 / \text{font size}$

- If size dynamically change, need to estimate smallest on-screen pixels per em
Minification

- At very small font sizes, lots of detail can occur inside each pixel
- Can’t be captured by single sample position at pixel center
Adaptive Supersampling

- As pixels get larger in em space, increase number of samples
- Use screen space derivatives to dynamically calculate sample counts for horizontal and vertical rays
GPU-Centered Font Rendering Directly from Glyph Outlines
Adaptive Supersampling

- Example sample count calculation
  - 1–4 samples per pixel in each direction

```cpp
float2 emsPerPixel = fwidth(renderCoord);
int2 sampleCount = clamp(int2(emsPerPixel * 32.0 + 1.0), int2(1, 1), int2(4, 4));
```
GPU-Centered Font Rendering Directly from Glyph Outlines

Adaptive Supersampling

Single sample

Supersampling

Your name is still visible, and you're a pilot. You're in the small town of Timber Valley, where you work at the airport. You've been flying for a few years now, and you're getting good at it. You love the excitement of the job, and you always try to make sure that you're doing your best.

You use the weather conditions and the time of day to plan your flights. You always try to take advantage of the wind, and you never fly against it. You're good at navigating, and you always try to stay safe.

The town is small, but it's close to many other places. You often fly to other towns and cities, and you always try to make sure that you're on time.

Your name is still visible, and you're a pilot. You're in the small town of Timber Valley, where you work at the airport. You've been flying for a few years now, and you're getting good at it. You love the excitement of the job, and you always try to make sure that you're doing your best.

You use the weather conditions and the time of day to plan your flights. You always try to take advantage of the wind, and you never fly against it. You're good at navigating, and you always try to stay safe.

The town is small, but it's close to many other places. You often fly to other towns and cities, and you always try to make sure that you're on time.

Your name is still visible, and you're a pilot. You're in the small town of Timber Valley, where you work at the airport. You've been flying for a few years now, and you're getting good at it. You love the excitement of the job, and you always try to make sure that you're doing your best.

You use the weather conditions and the time of day to plan your flights. You always try to take advantage of the wind, and you never fly against it. You're good at navigating, and you always try to stay safe.

The town is small, but it's close to many other places. You often fly to other towns and cities, and you always try to make sure that you're on time.
Priority #3: Runs Fast

- Minimize raw computation
  - We want to examine as few Bézier curves as possible in the pixel shader

- Promote high GPU resource utilization
  - We want low thread divergence in the pixel shader
Computation

- Looking for ray intersections with all Bézier curves would be very slow

- Many curves far away from ray and never contribute to coverage (classes A and H)

- Need to reduce active set of curves
Banding

- Divide glyph’s bounding box into many horizontal and vertical bands
Banding

- Bézier curves are sorted into the bands
  - A curve can belong to multiple bands
  - When rendering, band selected based on ray origin

- Doesn’t matter how large pixel footprint gets
  - Pixel size only matters in ray direction
  - Band parallel to ray extends forever
Banding

- Perfectly horizontal lines are never added to horizontal bands

- Perfectly vertical lines are never added to vertical bands

- Ray intersections with these can’t happen
Banding

- Further dividing into cells causes problems
  - Pixel could cover multiple cells along ray direction
  - Those cells often won’t have disjoint curve sets
  - Can’t calculate final winding number without additional per-cell fix-ups that aren’t robust

- Bands are a much cleaner and faster solution
Banding

- Using large numbers of bands is faster
  - Allows fewer curves per band

- Minimize number of curves in worst band
  - GPU thread coherence makes shader wait for highest number of loop iterations in a group of pixels (32 or 64, hardware dependent)
Banding

- Can merge data for bands containing identical sets of Bézier curves
Curve Sorting

- Curves in each horizontal band are sorted in descending order by the maximum $x$ coordinate of the three control points.

- This is an early-exit optimization that makes the shader about twice as fast compared to not sorting curves at all.
Curve Sorting

- Translate control points so that pixel center is at (0,0), and perform test:

  \[
  \text{if } (\max(\max(p12.x, p12.z), p3.x) \times \text{pixelsPerEm}.x < -0.5) \text{ break;}
  \]

- If true, then it’s not possible to hit this curve or any that follow
  - Sorted in descending order, so must also be true for all later curves in the band
Symmetric Band Optimization

- Can do even better at large font sizes
- Also sort in ascending order by minimum $x$ coordinate of the three control points
- Use a left-pointing ray when pixel $x$ coordinate is less than a band split value
Symmetric Band Optimization
Symmetric Band Optimization

- Sorting and split values also apply to vertical bands

- Splits values introduce divergence in shader
  - Faster for large font sizes where lots of pixels will choose same execution path
  - Slower for small font sizes due to decoherence
Bounding Polygons

- Glyphs tend to have empty space near the corners of their bounding boxes
- Clip these corners off to reduce pixels filled
- Adds more triangles, but roughly 10% faster with larger font sizes
Bounding Polygons
Bounding Polygons

- As with symmetric band splits, bounding polygons increase performance for large font sizes, but can hurt at small font sizes.

- Greater number of triangles increase number of pixels double-shaded in 2x2 quads along triangle edges.
Rectangle Primitives

- Even when rendering quads, double-shading along the interior edge can be a significant expense at small font sizes
- Expense can be eliminated for text that’s aligned to screen axes
Rectangle Primitives

- Most GPUs support rectangle primitives
  - Exposed through GL_NV_fill_rectangle extension

- Specify three vertices, and screen-aligned enclosing rect is drawn without internal edge

- Up to 15% faster for typical font sizes
Rectangle Primitives

- Can be combined with conservative rasterization to handle glyph dilation
- Automatically shades pixels with any amount of coverage
Multicolor Glyphs

- Microsoft emoji font uses vector artwork
  - Based on same TrueType quadratic Bézier curves
- Multiple layers composited back to front
  - Adds an outer loop to the pixel shader
Data Stored in Two Texture Maps

- **Curve texture**
  - 4-channel 16-bit floating-point
  - Stores all Bézier curve control points

- **Band texture**
  - 4-channel 16-bit integer
  - Stores lists of curves for all bands
Curve Texture

- Third control point of one curve is always the same as the first control point of the next curve in the contour.

<table>
<thead>
<tr>
<th>Red (16 bits)</th>
<th>Green (16 bits)</th>
<th>Blue (16 bits)</th>
<th>Alpha (16 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1.x (curve 1)</td>
<td>p1.y (curve 1)</td>
<td>p2.x (curve 1)</td>
<td>p2.y (curve 1)</td>
</tr>
<tr>
<td>p3.x (curve 1)</td>
<td>p3.y (curve 1)</td>
<td>p2.x (curve 2)</td>
<td>p2.y (curve 2)</td>
</tr>
<tr>
<td>p1.x (curve 2)</td>
<td>p1.y (curve 2)</td>
<td>p2.x (curve 2)</td>
<td>p2.y (curve 2)</td>
</tr>
<tr>
<td>p3.x (curve 2)</td>
<td>p3.y (curve 2)</td>
<td>p2.x (curve 3)</td>
<td>p2.y (curve 3)</td>
</tr>
<tr>
<td>p1.x (curve 3)</td>
<td>p1.y (curve 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Band Texture

- Each glyph has a list of H bands and V bands.
- Each band contains a list of curves, sorted in both directions.
## Results

- 4K display filled with text, timed on NV GeForce 1060

<table>
<thead>
<tr>
<th>Font</th>
<th>Sample</th>
<th>Complexity</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arial</td>
<td>ABCDEFG</td>
<td>28</td>
<td>0.70</td>
</tr>
<tr>
<td>Minion</td>
<td>ABCDEFG</td>
<td>35</td>
<td>0.71</td>
</tr>
<tr>
<td>Times</td>
<td>ABCDEFG</td>
<td>35</td>
<td>0.73</td>
</tr>
<tr>
<td>Jokerman</td>
<td>ABCDEFG</td>
<td>60</td>
<td>1.1</td>
</tr>
<tr>
<td>Spider</td>
<td>!ABCDEFG</td>
<td>500</td>
<td>2.8</td>
</tr>
</tbody>
</table>
Results

ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Results
Results
Questions?

- lengyel@terathon.com
- Twitter: @EricLengyel