ArcGIS API for JavaScript
A Look Under the Hood

Yann Cabon | Yaron Fine

2020 ESRI DEVELOPER SUMMIT | Palm Springs, CA

https://arcg.is/1zDa19

Agenda

- Goals and Challenges
- WebGL basics for GIS
- Data Access
- Off the main thread processing
- GPU based visualization

Goals and Challenges

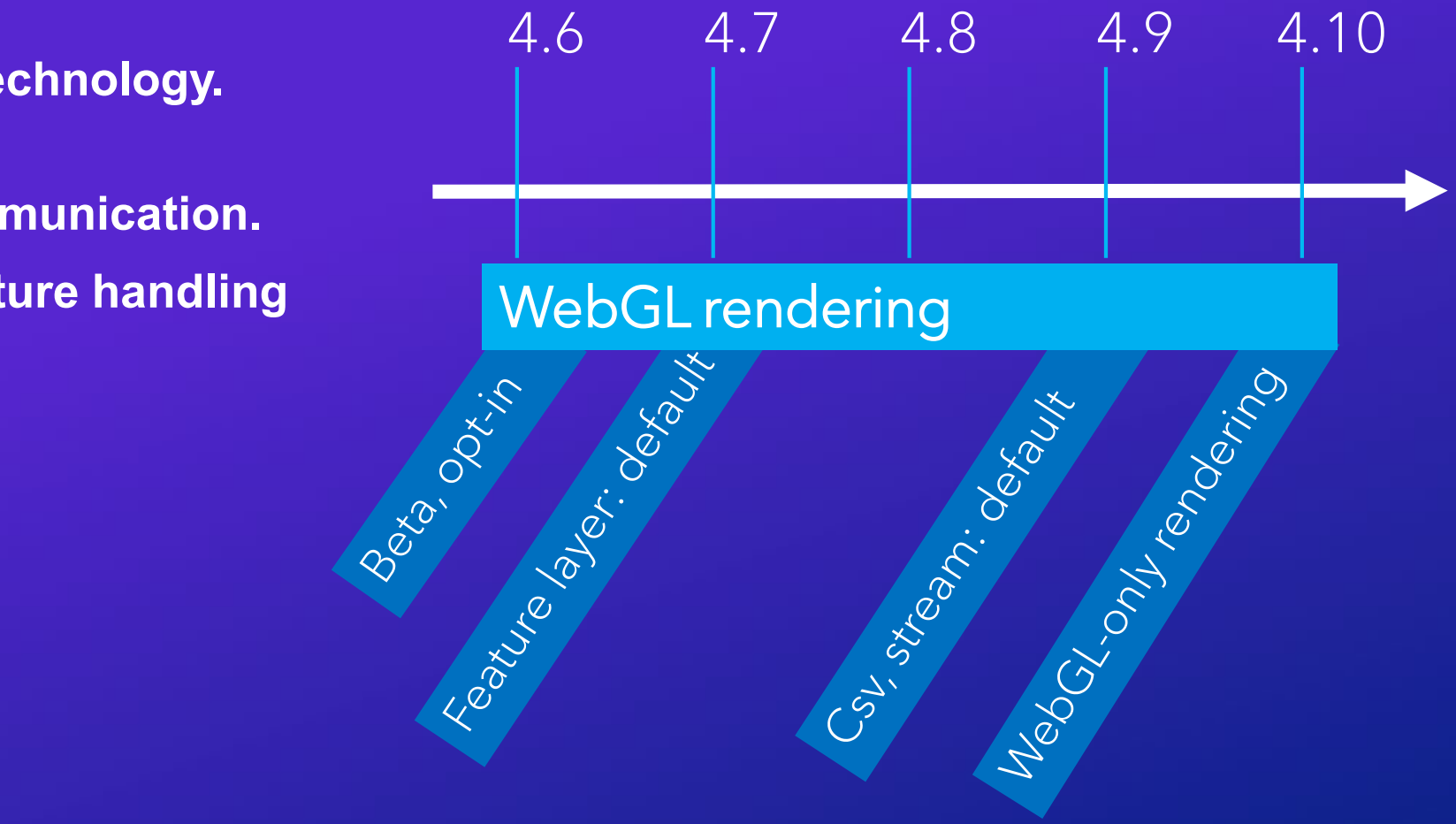
Yann Cabon

Goals and Challenges

- Goals
  - Visualize more data in the web browser.
  - Immediate feedback when changing layer's visualization configuration.
  - Good map navigation experience.
  - Innovate in the visualization space.
  - Do more with the data available in the web browser.
- Challenges
  - No rendering engine available out of the box in web browser.
  - JavaScript is single-threaded. Any CPU processing may freeze the web browser.
  - Datasets may be extremely large in number of features or complexity of geometries.

Goals and Challenges: Technology choices

- **WebGL as the rendering technology. Started with SVG**
- **Efficient client-server communication.**
- **Web Worker to offload feature handling off the main thread.**

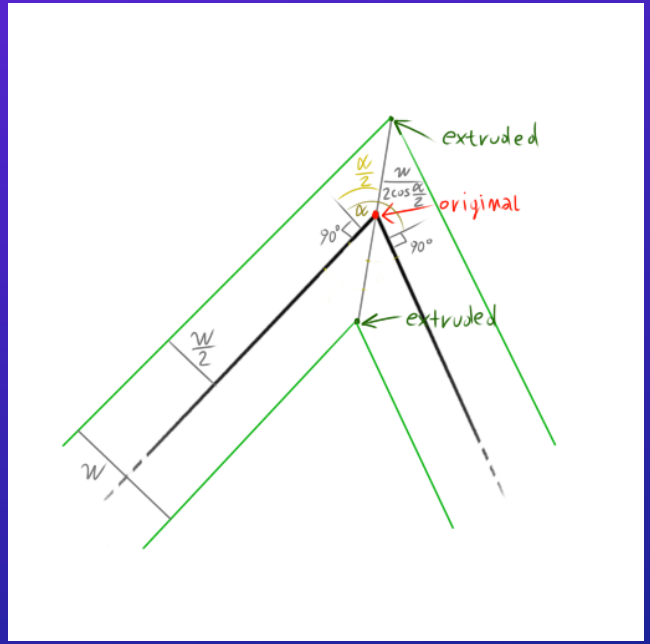


WebGL basics for GIS

Yaron Fine

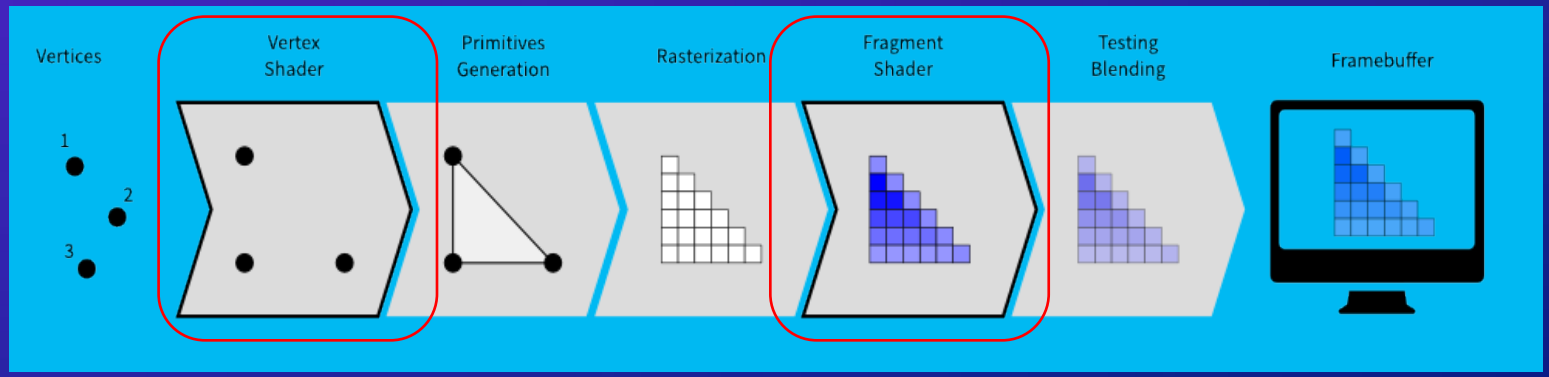
WebGL basics

- WebGL is a low-level graphics API which is very fast at rendering triangles...
- WebGL is using the graphics card of the device when available
- WebGL is a state machine, it does not like state changes...
- GIS geometries needs to be triangulated
  - The generated triangles will reconstruct the geometry at render time
- Drawing is done in a retained mode
  - Once the geometry uploaded to WebGL, it remains on the GPU
  - This is different than how canvas is rendering - *immediate mode*. On each draw-cycle each geometry has to be rendered by itself



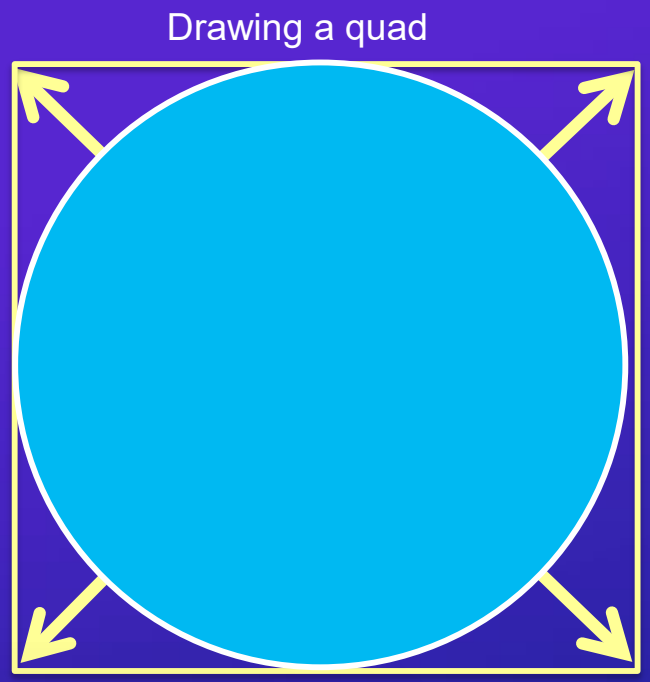
WebGL basics

- The drawing pipeline is built of multiple stages.
  - The result of each stage becomes the input of the next one
  - Two of the stages are programmable (*GLSL*):
    - *Vertex shader*
      - Position geometry on screen
    - *Fragment shader*
      - Assign color to the pixels of the geometry
  - Data is input with several types and traditional usage:
      - Buffers - mesh
      - Textures - material
      - Uniforms - transformations



WebGL drawing marker features

- We render marker as quads (two triangles sharing an edge)
- Coloring is done either by using a texture or shading technique
- We encode much of the information into the geometry
  - Each geometry has enough information such that we can encode different types of features onto the same resource (buffer)

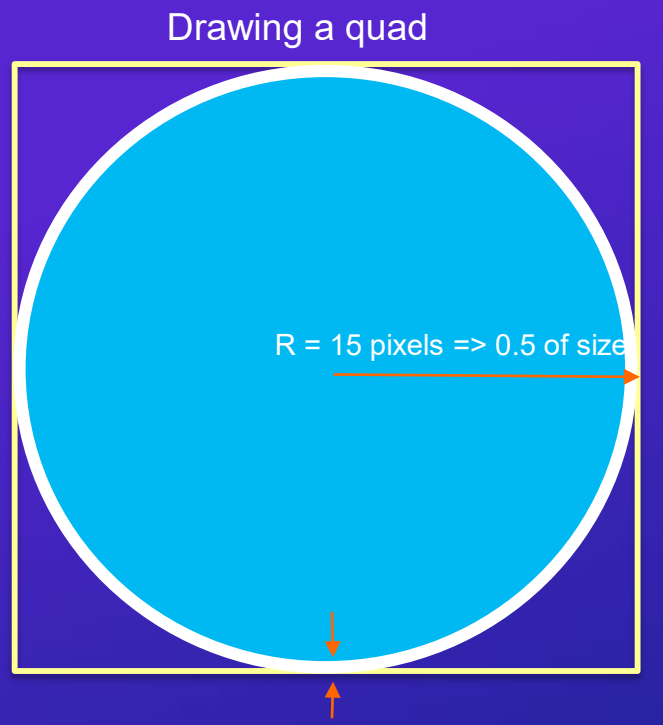


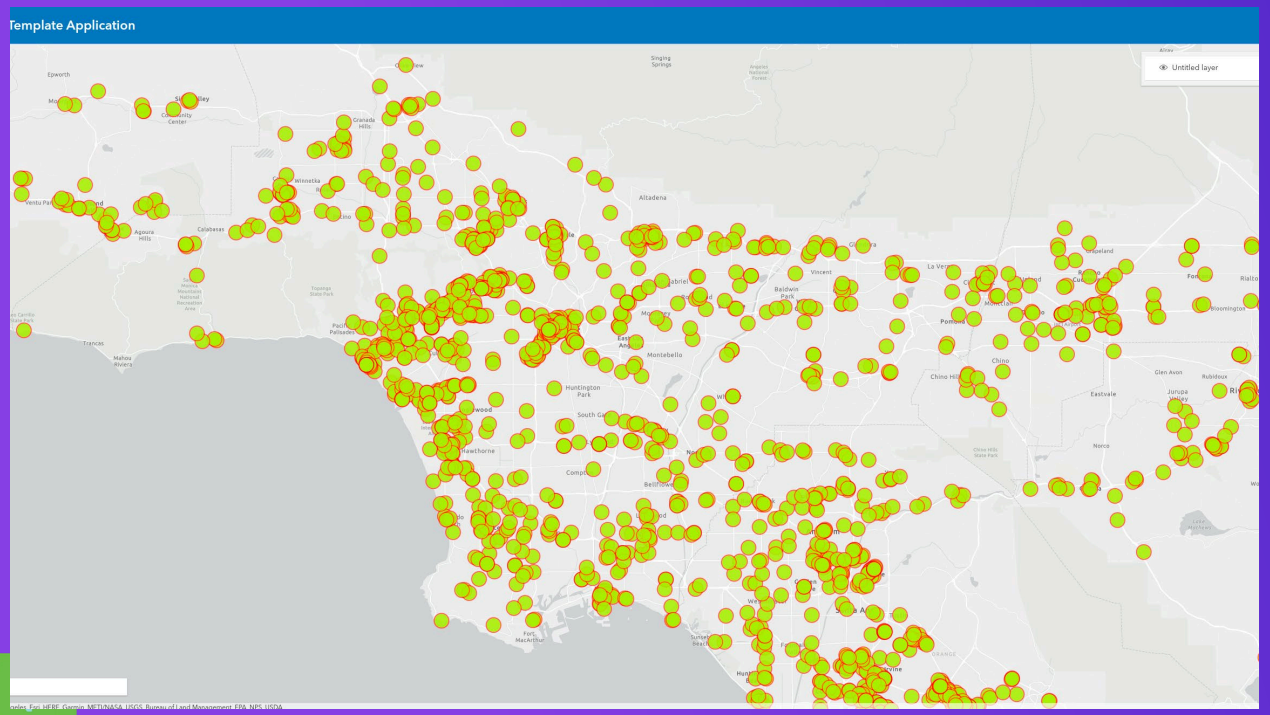

```
vertexBuffer = [x1, y1, offsetx, offsety, x2, y2, offsetx, offsety... x4, y4, offsetx, offsety]
```

WebGL drawing marker features

- Fragment shader color each pixel
  - Use data passed from the vertex shader
  - Values are interpolated per pixel

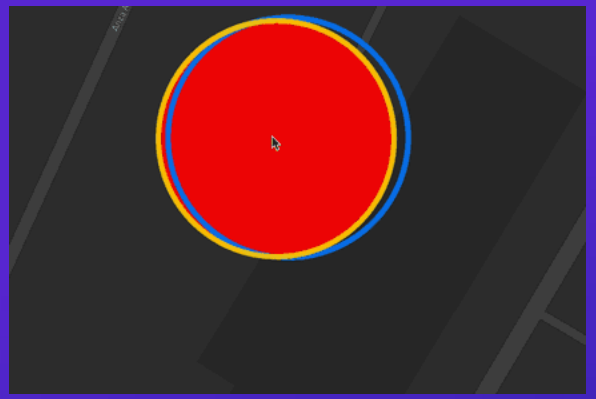
```
precision highp float;

varying vec2 v_offset;

const vec4 fillColor = vec4(0.03, 0.43, 0.90, 0.65);
const vec4 outlineColor = vec4(1.0, 0.0, 0.0, 0.65);
const float outlineSize = 1.0 / 15.0;

void main() {
  mediump float dist = length(v_offset);
  mediump float fillalpha1 = 1.0 - smoothstep(0.45, 0.45 + outlineSize, dist);
  mediump float fillalpha2 = 1.0 - smoothstep(0.40, 0.45, dist);
  vec4 color1 = fillalpha1 * outlineColor;
  vec4 color2 = fillalpha2 * fillColor;
  gl_FragColor = (1.0 - color2.a) * color1 + color2;
}
```





Simple Marker Layer

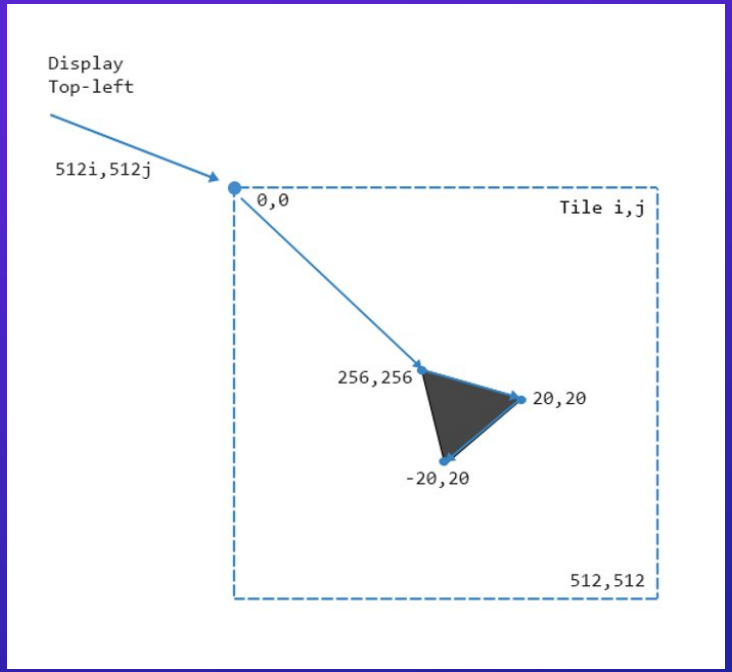
32 bits precision limitations

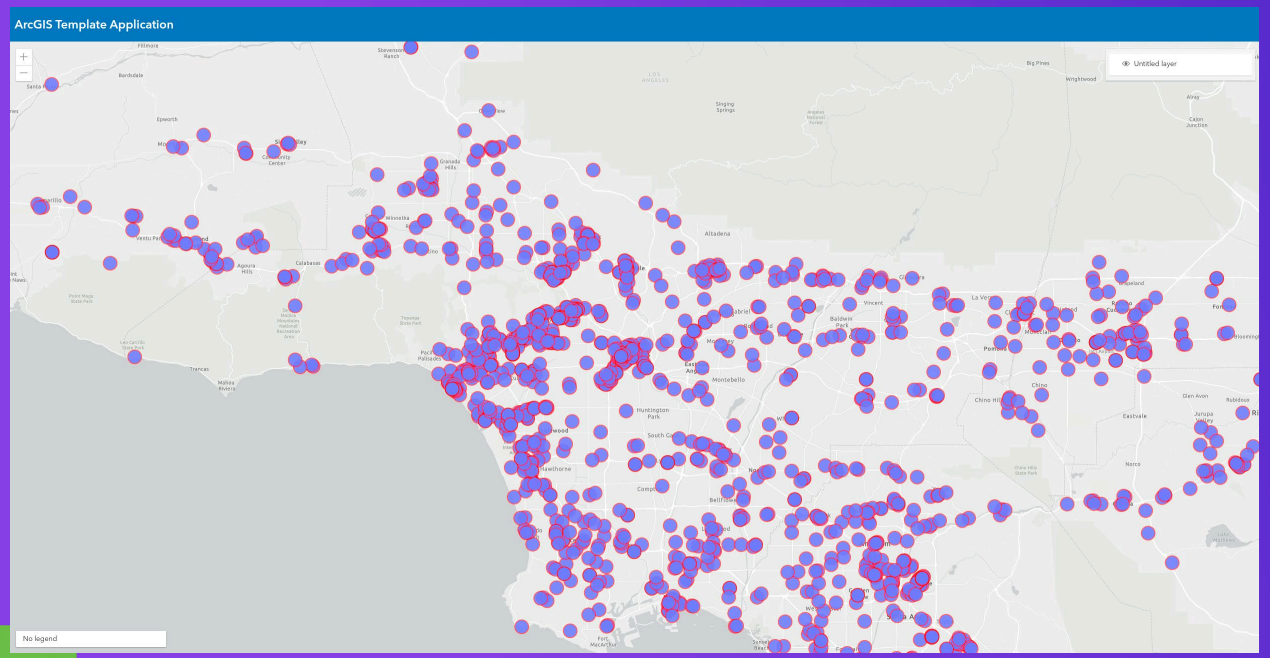


- WebGL can only handle 32-bit floating point
- Not enough precision for geographical applications
- Web Mercator coordinate range spans across ~40,000,000 meters
- To display objects on the screen we apply a translation on the object's coordinates. When user zoom in, the translation gets so big that it exceeds the 32bits precision limit.
- IEEE standard for floating point reserves 23 bits for the number part, meaning that an encoded number can handler about 2^23 ~ 8,400,000
- There are two approaches to address this limitation:
  - Use a local origin (other than the coordinate system's origin)
  - Render the data in tiles

Tiling

- Divide the world into a grid of tiles, with each tile defining a local origin
- Each tile has its own local origin, we can now represent the entire world with single-precision floats
  - Actually, we can fit the positions of the geometry onto the GPU using integers
- We can represent data at different resolution depending on the level of detail of the tiles
- Wraparound becomes trivial to support -> simply repeat tiles...
- By making each tile a power of two, we can cleanly divide each tile into four when zooming in one LOD

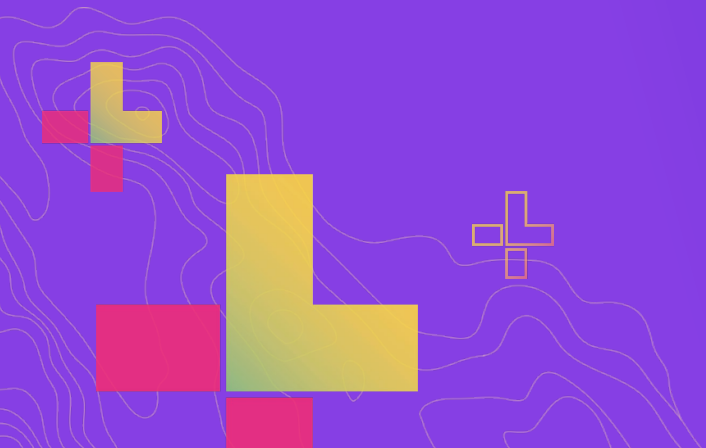




Tiled Marker Layer

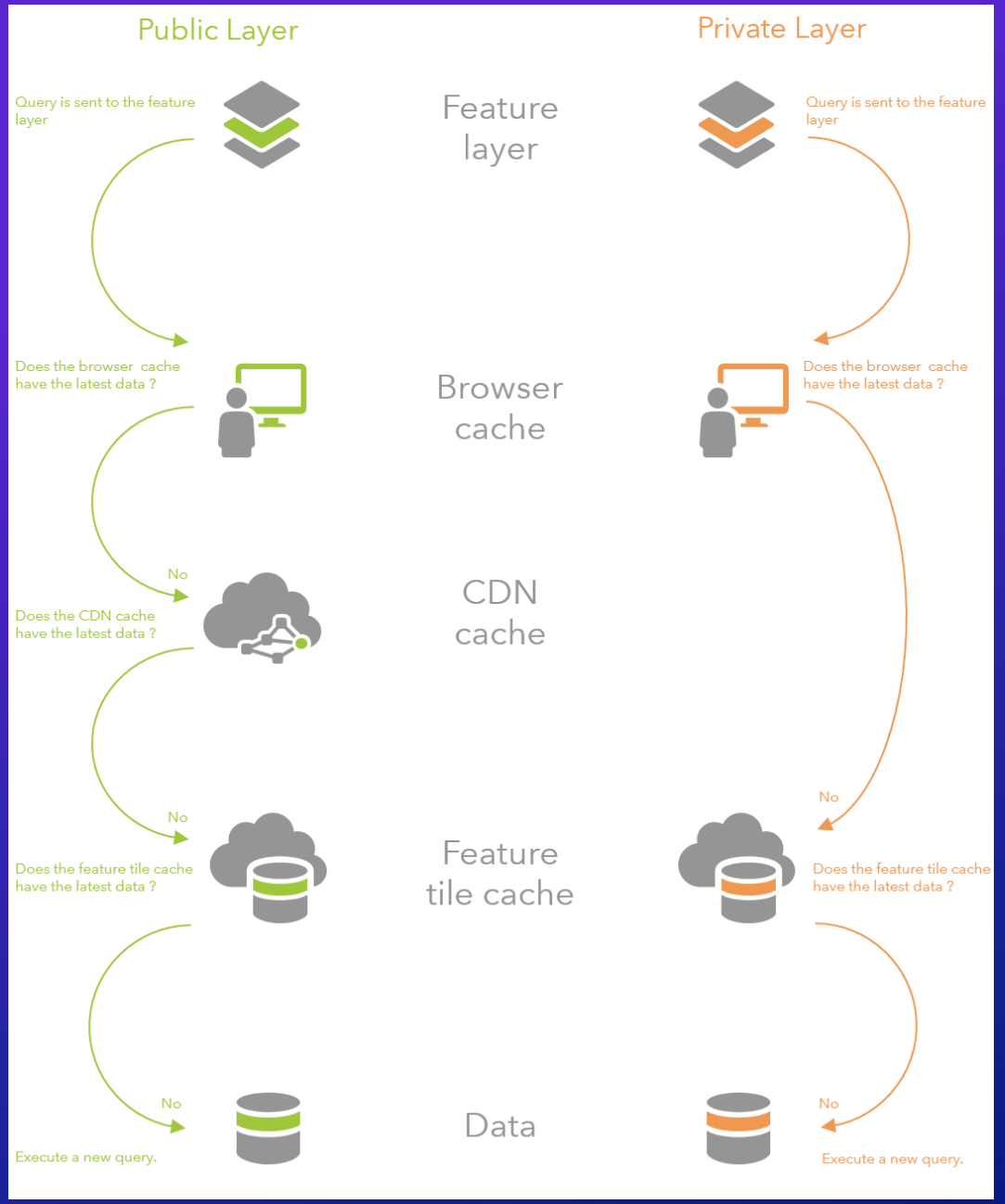
Data Access

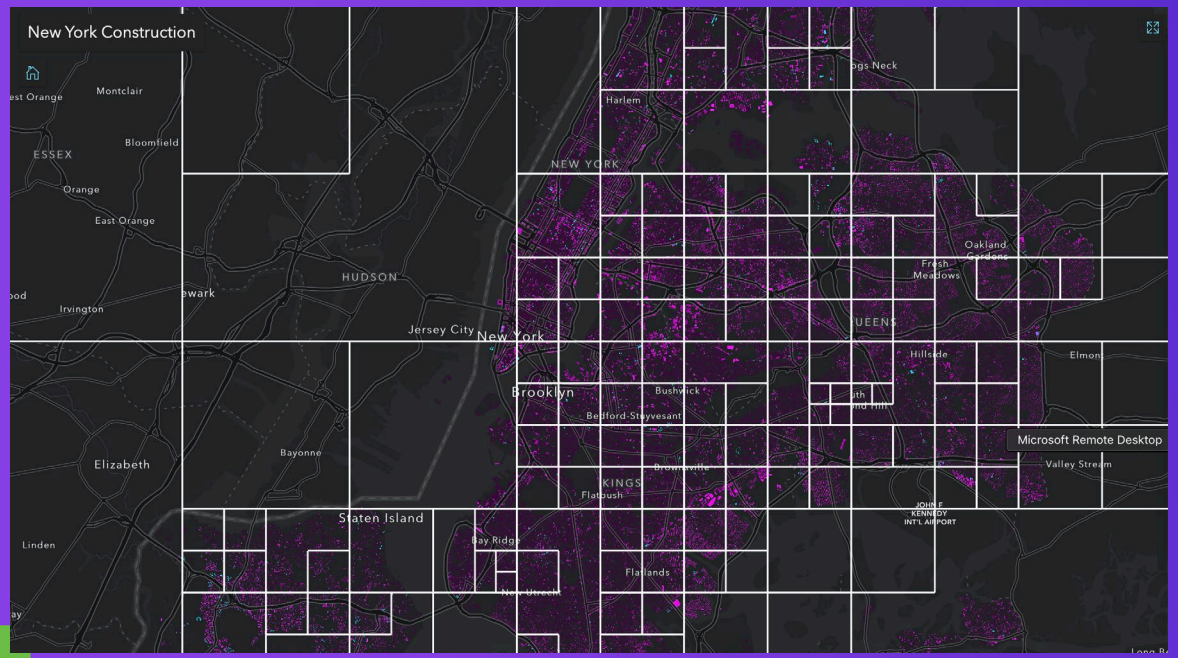
Yann Cabon



Data Access

- Two strategies to efficiently query for features:
  - Feature tiles queries
    - Only fetch data for the visible area.
    - Highly cacheable
    - Small payload – geometries generalized on the fly
  - Snapshot
    - Fetch all the features in one shot.
    - Less queries, higher payload
    - Available in 4.x in a future release.

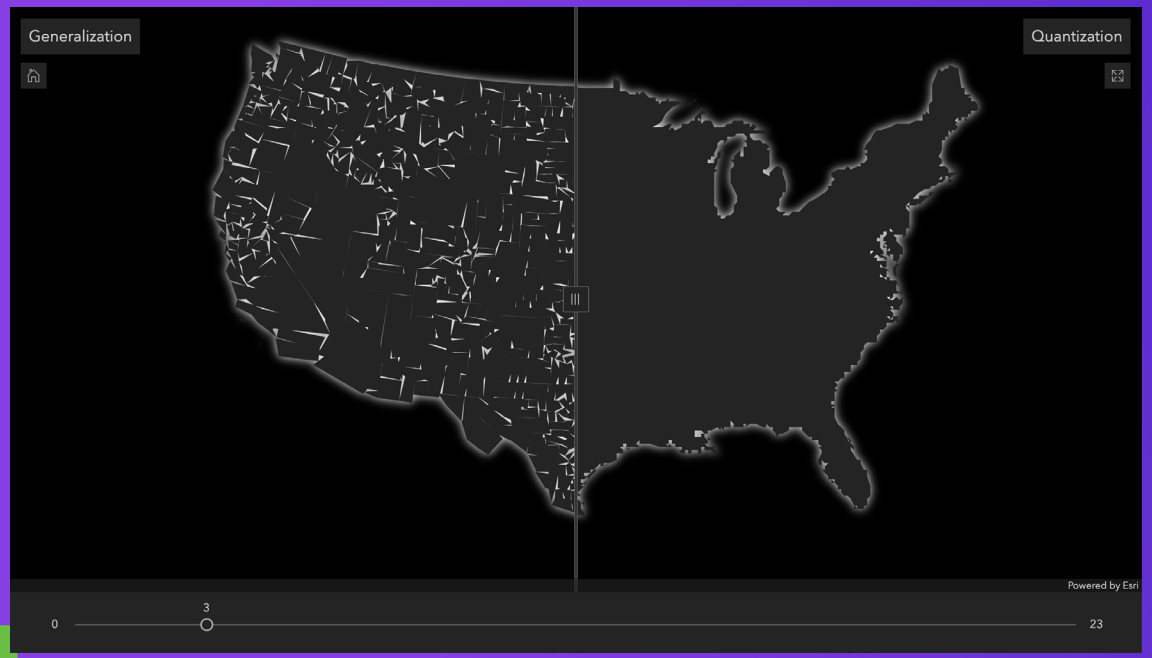




Feature tiles queries

Data Access – Optimizing the payload

- Only fetch the required fields for rendering
  - Renderer, labeling, elevation
  - Popup fields are fetched when user clicks on a feature
  - Can be controlled by `FeatureLayer.outFields`
- Generalize geometries to reduce the size
  - Douglas-Peucker algorithm
    - Very fast and effective at removing vertices
    - Controlled by `Query.maxAllowableOffset`
    - Creates visual artifacts between adjacent polygons
  - Quantization
    - Preserves visual topology
    - High compression rate with GZIP

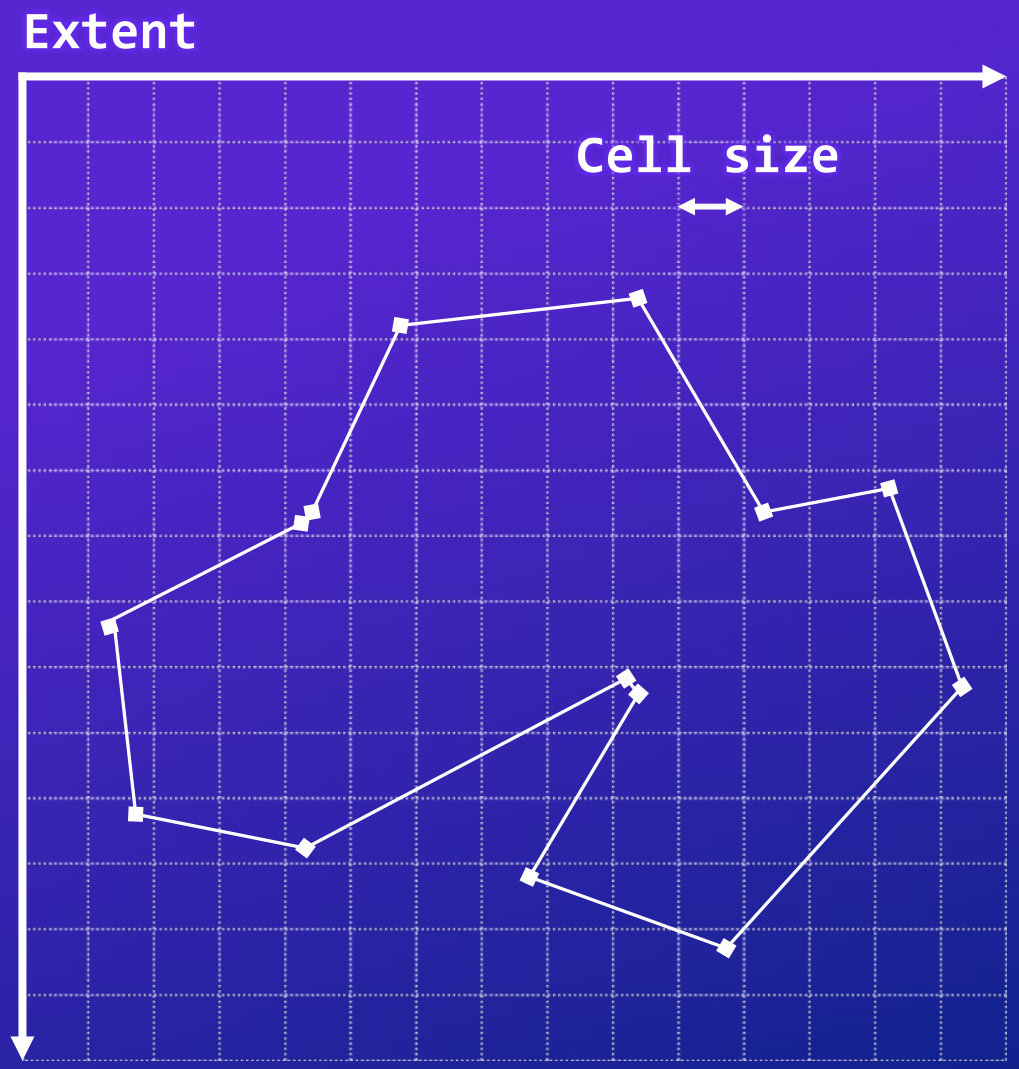


Quantization compression

Data Access - Quantization

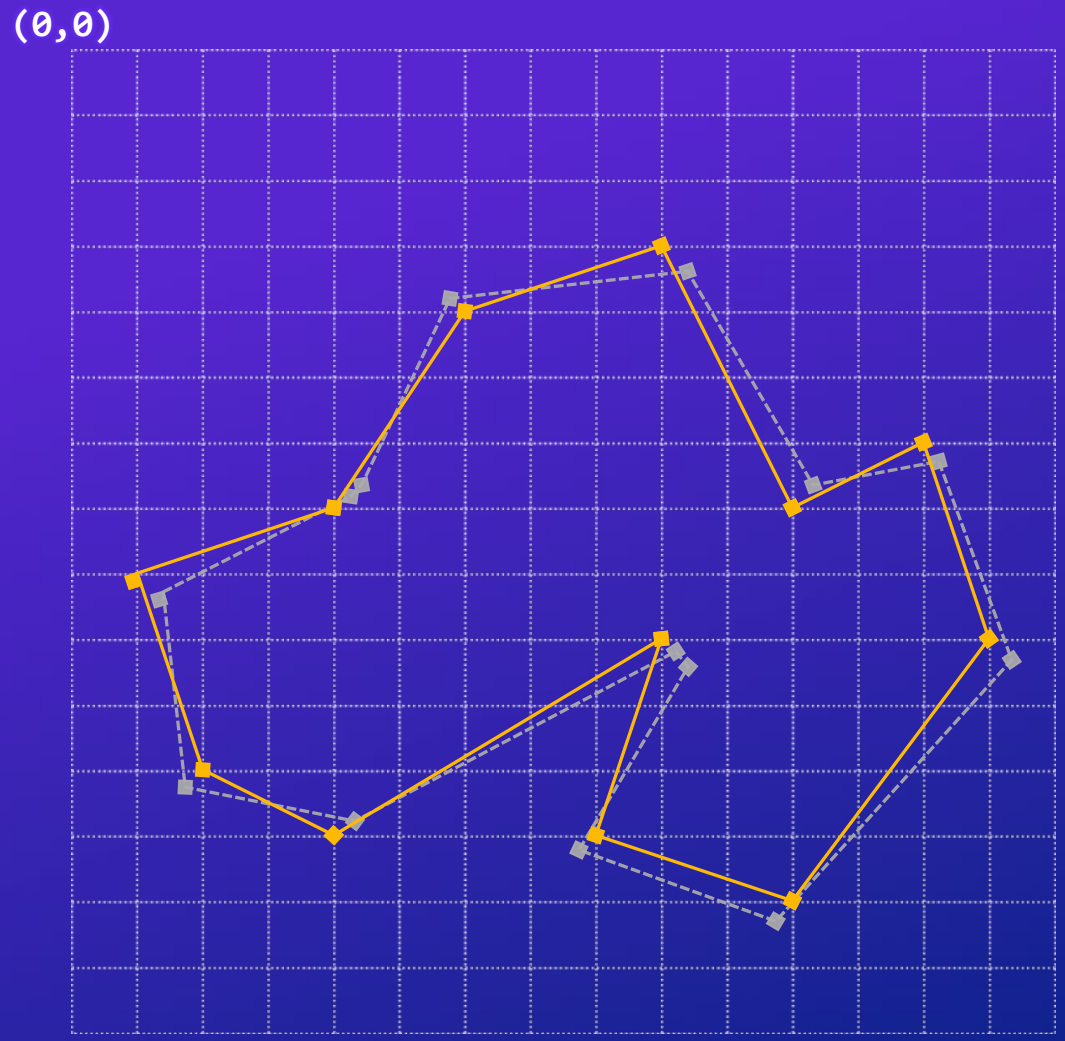
Data Access - Quantization

- Determine the grid



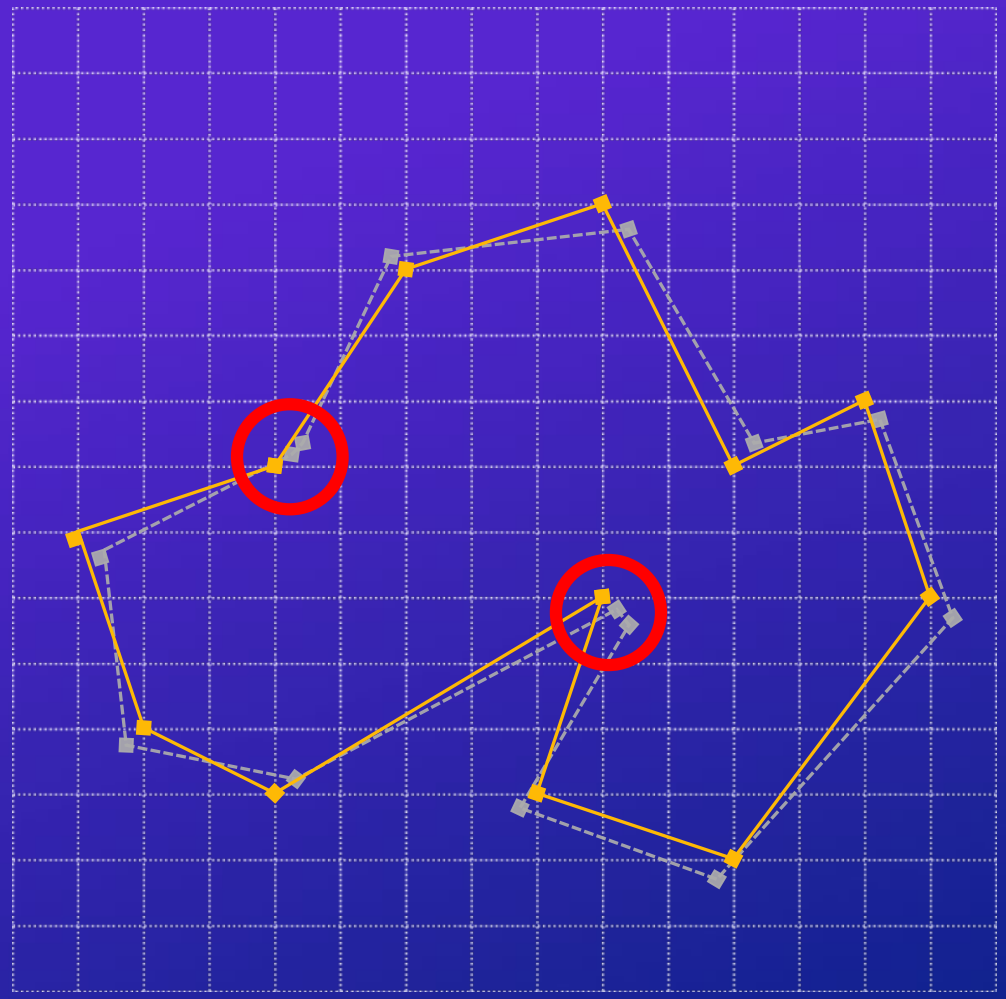
Data Access - Quantization

- Determine the grid
- Snap points on the grid.
  - Each coordinate is rounded to the nearest grid point
  - [x,y] is a grid position - integers



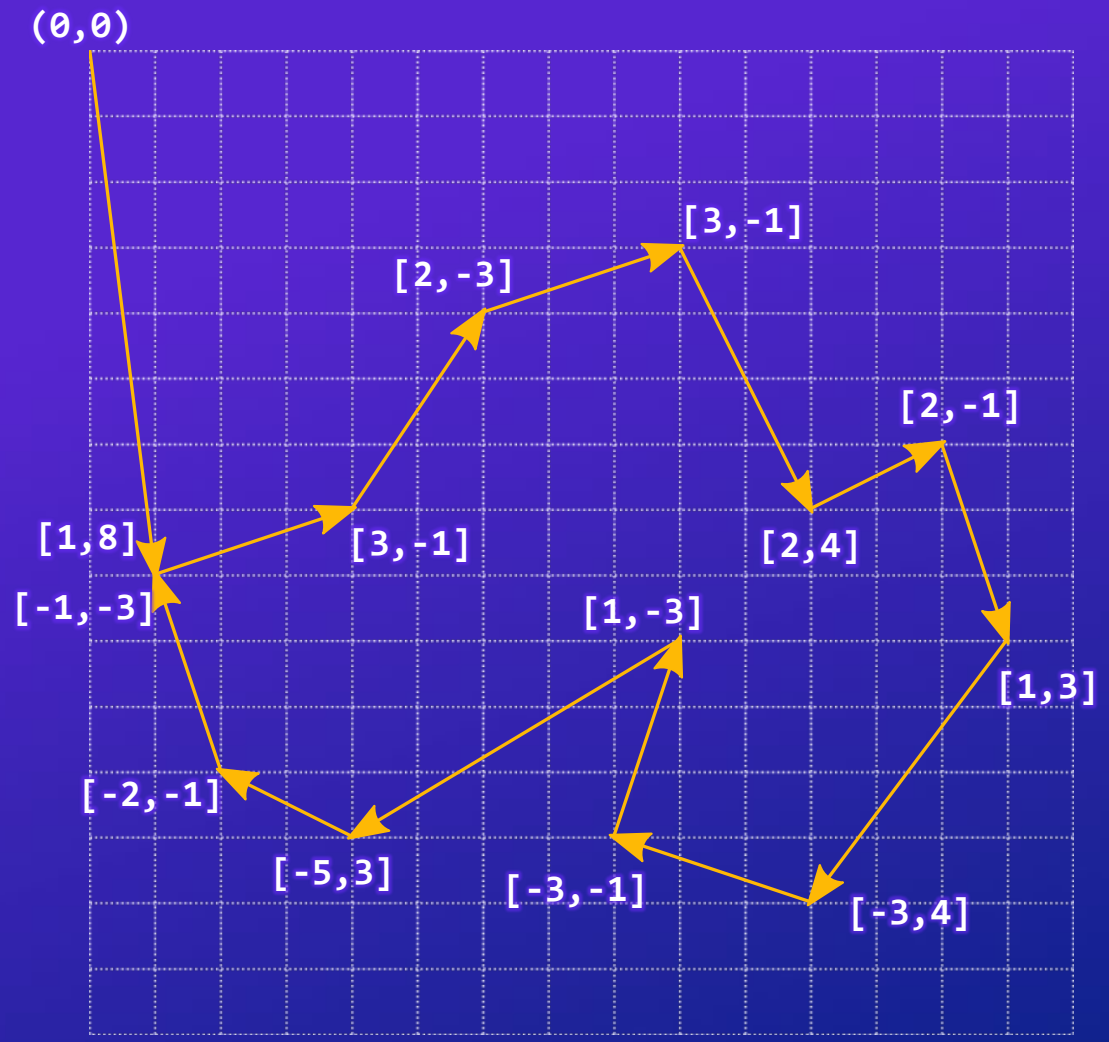
Data Access - Quantization

- Determine the grid
- Snap points on the grid.
  - Each coordinate is rounded to the nearest grid point
  - [x,y] is a grid position - integers
- Remove points that collapsed to the same grid coordinates



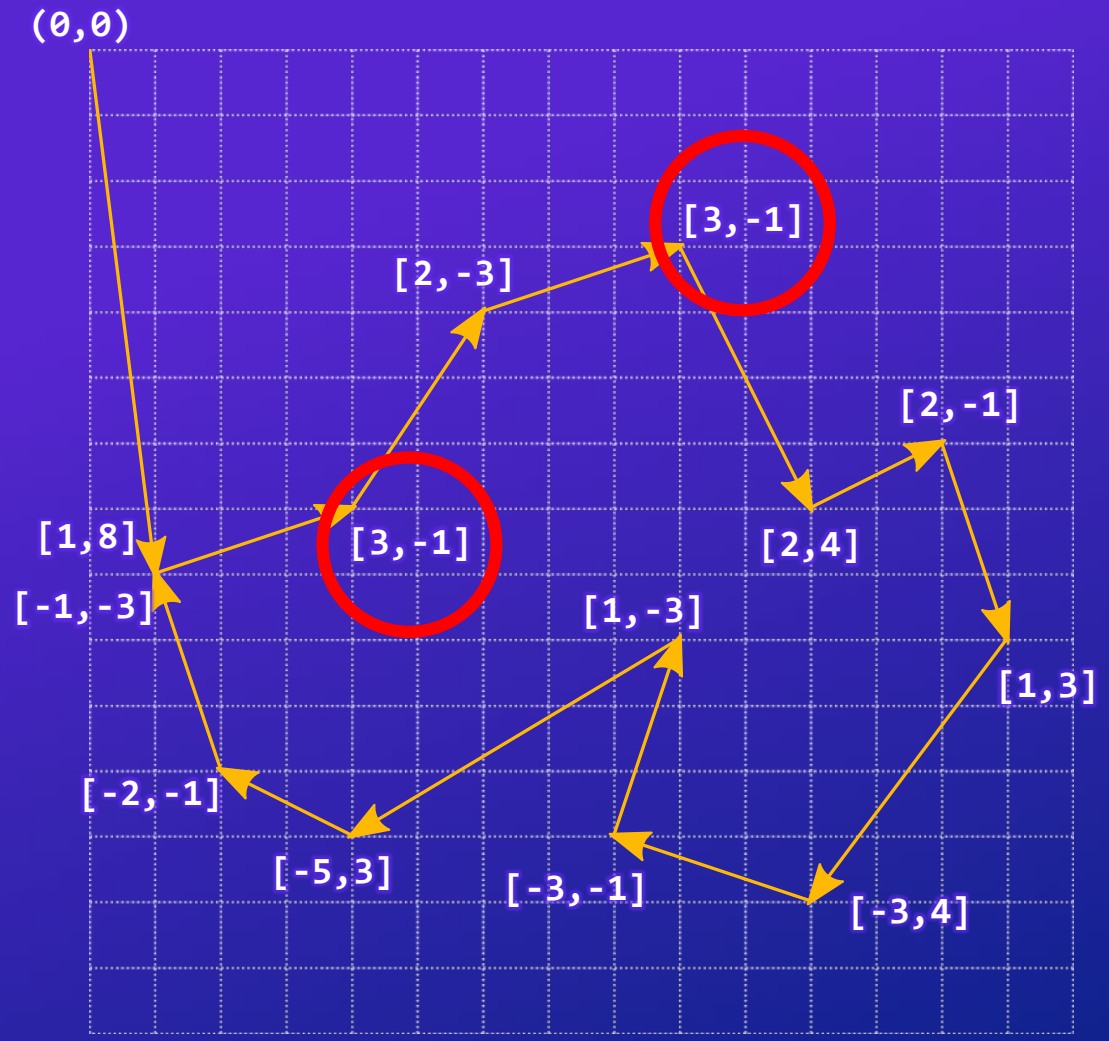
Data Access - Quantization

- Determine the grid
- Snap points on the grid.
  - Each coordinate is rounded to the nearest grid point
  - [x,y] is a grid position - integers
- Remove points that collapsed to the same grid coordinates
- Encode deltas



Data Access - Quantization

- Determine the grid
- Snap points on the grid.
  - Each coordinate is rounded to the nearest grid point
  - [x,y] is a grid position - integers
- Remove points that collapsed to the same grid coordinates
- Encode deltas
- GZIP compresses really well repeating patterns

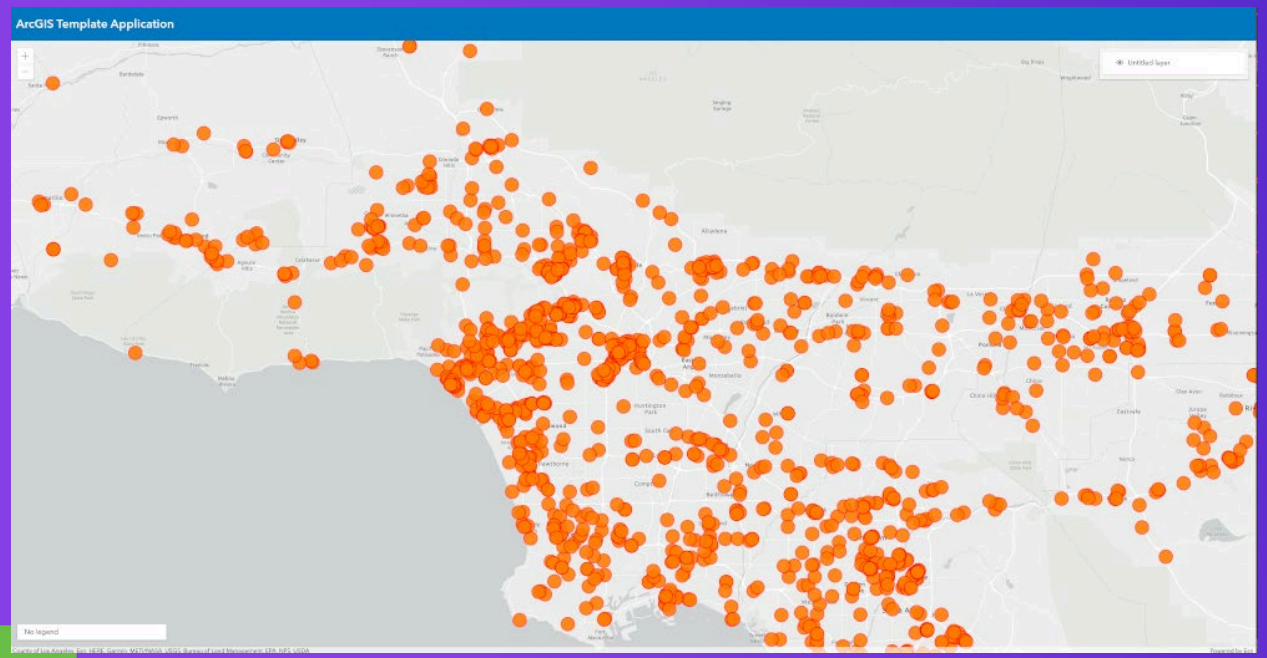


Data Access - Quantization

```
const tileExtent = new Extent({
  xmin:-8238077.160463991,
  ymin: 4975133.297027238,
  xmax:-8233185.190653741,
  ymax: 4980025.266837489,
  spatialReference: { wkid: 102100 }
});

const tolerance = tileExtent.width / 512;
// const tolerance = view.resolution;

const featureSet = await layer.queryFeatures({
  geometry: tileExtent,
  quantizationParameters: {
    originPosition: "upper-left",
    extent: tileExtent,
    tolerance
  }
});

featureSet.unquantize();
```

Data Access

- Quantization is available:
  - in hosted feature services in arcgis.com since 2014
  - In ArcGIS Enterprise since 10.6.1
- Generalization Limitations
- Protocol buffers
  - PBF is a mechanism to serialize data in binary format https://developers.google.com/protocol-buffers
  - Extra layer of compression, around 20%
  - Available in ArcGIS Enterprise 10.8 Hosted feature services



Tiled Marker Layer with server quantization

Off the main thread processing

Yann Cabon

Off the main thread processing

- **Web Workers**
  - Lightweight threads in the web browser.
  - Run heavy tasks without impacting the performance of the main thread.
  - No access to memory from the main thread.

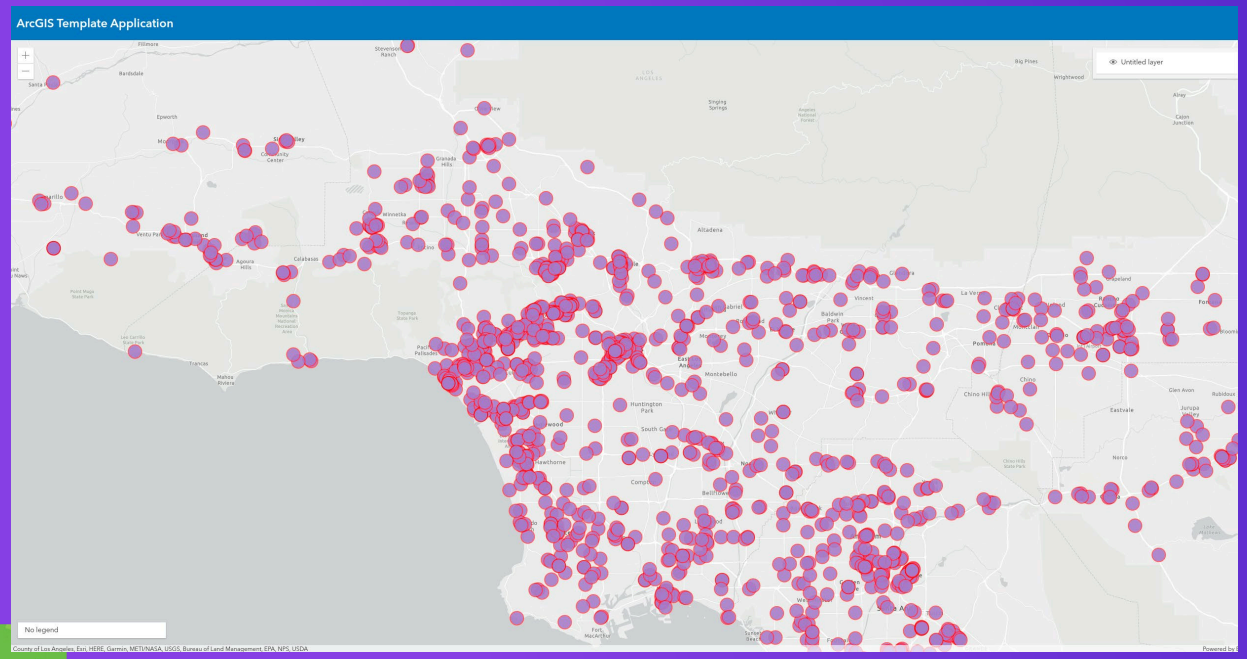
- **Demos:**
  - Create a simple worker
  - Making the API nicer using Promises
  - Loading modules in a Web Worker

Off the main thread processing

- API worker framework
  - Built to share a fix number of Workers for all the layers.
    - The work is distributed across all workers for vector tiles.
    - Or a worker is assigned to a layer.
  - Can load AMD modules and then work with them
  - Requests are executed in the main thread to handle security
  - In 2D the framework is used to run:
    - Feature Fetching
    - Storage
    - Client-side queries
    - WebGL pre-processing.
- Demo

Off the main thread processing

- Other options
  - Comlink
  - Greenlet
- Webpack plugin
  - Webpack Worker Plugin



Tiled Marker Layer with server quantization and WebWorkers

GPU based visualization

Yaron Fine

GPU based visualization

- **WebGL can access data through buffers, texture and uniforms**
  - **Buffers and Texture**
    - **Can fit a lot of data onto the GPU**
    - **Relatively slow to update**
    - **Usually contain positions and attribute values***
  - **Uniforms**
    - **Small amount of data compared to texture and buffers**
    - **Very fast to update (even between draw-calls)**
      - **Allow us to make changes immediately on the fly**
      - **Evaluate properties per feature**

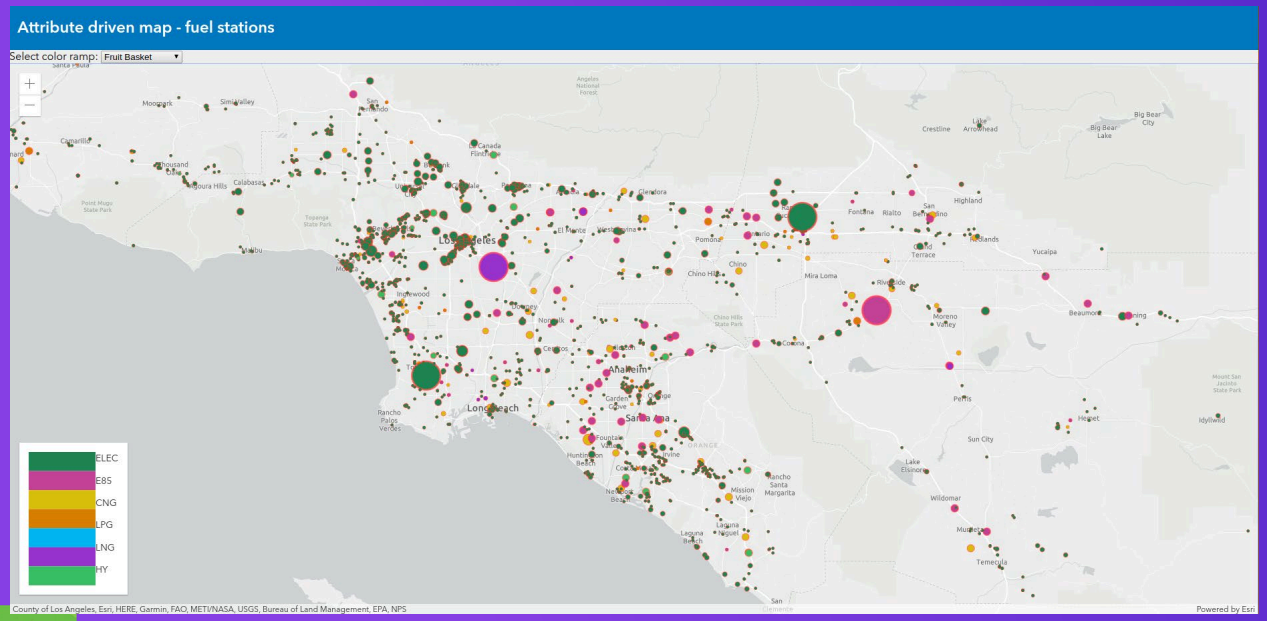
Feature 1 Feature 2 Feature n

```
const vertexData = [x1, y1, vv1, x2, y2, vv2,..., xn, yn, vvn];
cont vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertexData, gl.STATIC_DRAW);
```

```
private _widthInfo: [number, number, number, number];

setWidthValues(minMaxValues: [number, number], minMaxWidth: [number, number]): void {
  this._widthInfo[0] = minMaxValues[0];
  this._widthInfo[1] = minMaxValues[1];
  this._widthInfo[2] = pt2px(minMaxWidth[0]);
  this._widthInfo[3] = pt2px(minMaxWidth[1]);

  this.requestRender();
}
...
gl.uniform4fv(this._widthInfoLocation, this._widthInfo);
```

We can render all features in just a few draw calls!

*attribute data is normalized by the rendering engine



All the above + data driven rendering

GPU based visualization - VV evaluation GPU side

- For each geometry we store more than just a position.
  - We encode the feature attributes (field) that are visualized
  - For size, color, opacity, rotation
- The shader interpolates the value per feature on the fly. FAST!
  - This is how each feature may have different color, size, opacity etc.

```
float getWidth(float widthAttribute, float defaultWidth) {
  if (attributeValue == NaN) {
    return defaultWidth;
  }


  float interpolationRatio = (widthAttribute  - u_widthInfo.x) / (u_widthInfo.y - u_widthInfo.x);
  interpolationRatio = clamp(interpolationRatio, 0.0, 1.0);

  return u_widthInfo.z + interpolationRatio * (u_widthInfo.w - u_widthInfo.z);
}
...

float width = getWidth(widthAttribute, DEFAULT_WIDTH);
vec3 pos = u_dvsMat3 * vec3(a_position, 1.0) + u_displayViewMat3 * vec3(width * a_offset, 0.0);
gl_Position = vec4(pos.xy, z, 1.0);
```

GPU based visualization - VV evaluation CPU side

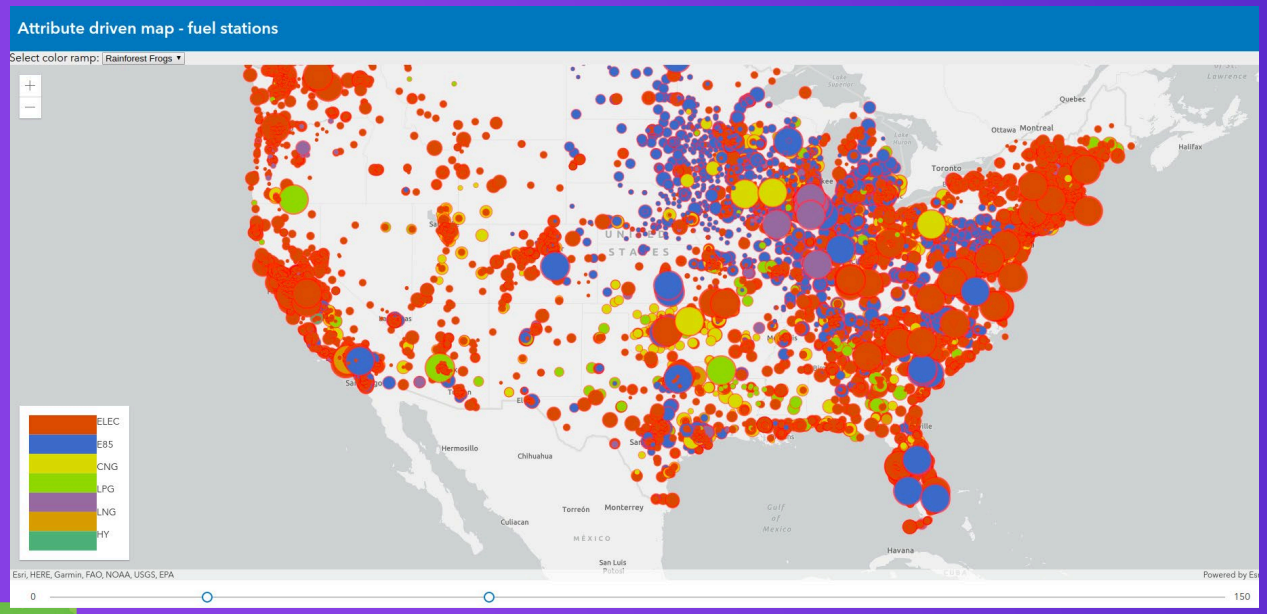
- We define uniforms to map a field value on a range of size/color/opacity/rotation values.

- Changing the range values in the uniform value will result in immediate change to the drawing features
  - Color-ramps
  - Sizes
  - opacities

```
render(renderParameters: any): void {
  const gl = renderParameters.context;
  ...

   // set uniforms
  gl.uniformMatrix3fv(this._displayViewMatrixLocation, false, this._dvm3);
  gl.uniform1f(this._pixelRatioLocation, window.devicePixelRatio);
  gl.uniform4fv(this._vvSizeMinMaxValue, this._minMinMaxVV);
  gl.uniform4fv(this._uniqueValueColors, this._colors);

  tilesToRender.forEach((tile) => {
    this._renderTile(gl, tile);
  });
}
```

GPU based visualization - Batching

- WebGL does not like state changes (changing shaders, textures, buffers etc.)
- Dispatching JavaScript commands is slow
- In order to mitigate this and optimize drawing we encode as many geometries as possible into a single buffer (this is where we store geometries), thus merging geometries.
  - This technique reduces the number of draw calls.
  - **Batched** objects must have similar geometries, the same shader, same texture and same WebGL state

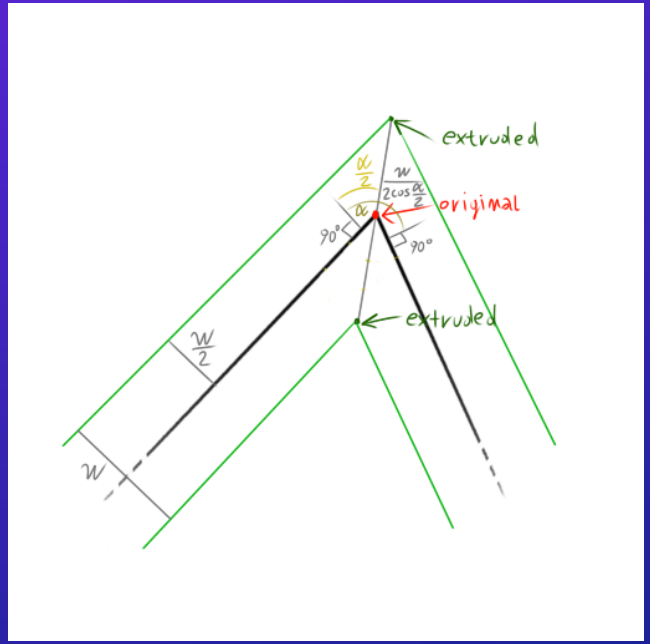


Fast updates

What next?

- What about other geometry types?
  - Drawing Polygons and Polylines are more complicated.
  - BaseLayerView2DGL has utilities to help the triangulation.
- API reference and samples about custom layerviews in 2D

- To learn more about custom visualization, checkout the presentation:
  ArcGIS API for JavaScript: Building Custom Visualizations Using WebGL in 2D Map Views



ArcGIS API for JavaScript
A Look Under the Hood

Yann Cabon | Yaron Fine

2020 ESRI DEVELOPER SUMMIT | Palm Springs, CA

https://arcg.is/1zDa19