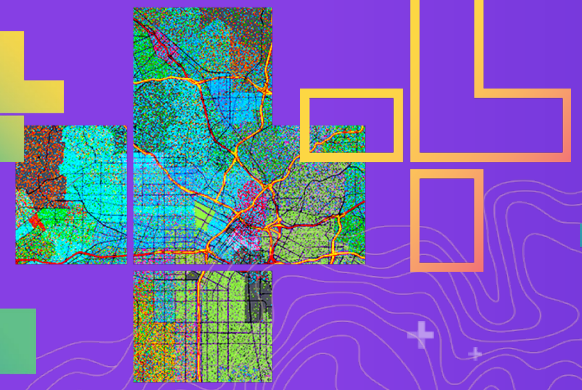
ArcGIS API for JavaScript: Building Custom Visualizations Using WebGL in 2D Map Views

Yaron Fine, Dario D’Amico

2020 ESRI DEVELOPER SUMMIT | Palm Springs, CA



Why custom layers?

- Users have specific visual requirements, give users fine-grain control to implement them
- Integrate with unsupported data formats, services
- Integrate with 3rd party rendering engines
- Performance of specific use cases (update features at a subsecond rate)
- Why as a layer?
- Integrate with the rest of the map
  - The layer paradigm is one of the most familiar concepts in GIS
  - Turn on-off
  - Move layers around, up/down
  - Hittest and popups
  - Whatever you create will behave just like one of the predefined layers
- Apply different effects

Custom layers and layer views

- In the API there is a distinction between layers and views:
  - Layer is the data
    - Save, load, export, exchange, query…
  - Layer view is responsible for visualization
- The developer must implement both a layer and a layer view
- We provide APIs to simplify the process of building custom layers and layer views
  - Including some high level interfacing with third-party libraries
  - We are open to suggestions and requests to integrate with more libraries

Different ways to extend layer views

- We provide two extension points to write layer views
  - Using the Canvas2D API → Extend `BaseLayerView2D`
  - Using WebGL → Extend `BaseLayerViewGL2D`
- Considerations for using Canvas2D vs WebGL
  - Visual requirements
  - Integration with 3D party libraries
  - Performance requirements
  - Developer expertise

Unreal Bloom

Yaron Fine

Demo

Light Sweep

Dario D’Amico

Demo

Implementing a custom layer with WebGL

- Chose whether use tiling or not
- Subclass `Layer`
  - If using tiling, specify a `tileInfo` on the layer
- Subclass `BaseLayerViewGL2D`
  - Implement:
    - `attach()`
    - `detach()`
    - `render()`
    - `hitTest()`
    - If using tiling, handle `tilesChanged()`

Implementing a custom layer with WebGL

```
var CustomLayer = Layer.createSubclass({
  createLayerView: function (view) {
    if (view.type === "2d") {
      return new CustomLayerView2D({ view: view, layer: this });
    }
  }
});
```

Implementing a custom layer with WebGL

If using tiling:

```
1  var CustomTileLayer = Layer.createSubclass({
2    tileInfo: TileInfo.create({ spatialReference: { wkid: 3857 }}),
3    createLayerView(view) {
4      if (view.type === "2d") {
5        return new CustomLayerView2D({
6          view: view,
7          layer: this
8        });
9      }
10   }
11 });
```

- The API will treat the layer as a tile layer and will add and remove tiles when the extent changed
- The implementor can react to changes in tile coverage and handle the rendering accordingly

Implementing a custom layer with WebGL

```
var CustomLayerView2D = BaseLayerViewGL2D.createSubclass({
  attach: function () { … },
  detach: function () { … },
  render: function (renderParameters) { … },
  hitTest: function (x, y) { … }
});

```

The `attach()` method

- Called once after the layer view is added to the map view
  - Typically used for WebGL resource creation
    - `this.texture = this.context.createTexture();`
    - `this.vertexData = this.context.createBuffer();`
    - `this.program = this.context.createProgram();`
  - More resources may need to be created dynamically at a later time
  - Any non-WebGL resource and any other initialization task can happen in the constructor
    - `this.someJson = fetch("./some.json").then(…)`
    - `this.myCustomFlag = false;`
- Access to the WebGL rendering context via `this.context`

The `detach()` method

- Called once after the layer view is removed
  - Typically used to dispose rendering resources
    - `this.context.deleteTexture(this.texture);`
    - `this.context.deleteBuffer(this.vertexData);`
    - `this.context.deleteProgram(this.program);`
- Cancel any loading process/scheduled creation of resources
- Access to the WebGL rendering context via `this.context`

The `render()` method

- The `render()` method receives a `renderParameters` object containing:
  - `context: WebGLRenderingContext` ← **Shared with every other layer!**
  - `stationary: Boolean` ← **Is the user panning, zooming or rotating?**
  - `state: ViewState`
    - `center: Number[]`
    - `extent: Extent`
    - `resolution: Number`
    - `rotation: Number`
    - `scale: Number`
    - `size: Number[]`

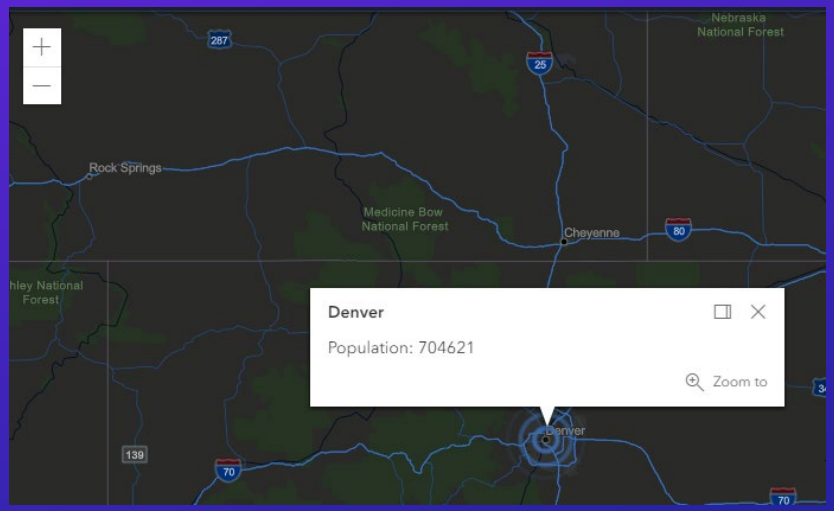
  Completely defines the portion of the map that is displayed
  Also, it has a few useful methods: `toMap()`, `toScreen()`, `toScreenNoRotation()`

- At every frame it must draw a visualization:
  - Of the layer data (e.g. `this.layer.graphics` or `this.layer.myCustomFormat`)
  - Based on the current view state (`renderParameters.state`)
  - By sending WebGL commands to the context (`renderParameters.context`)

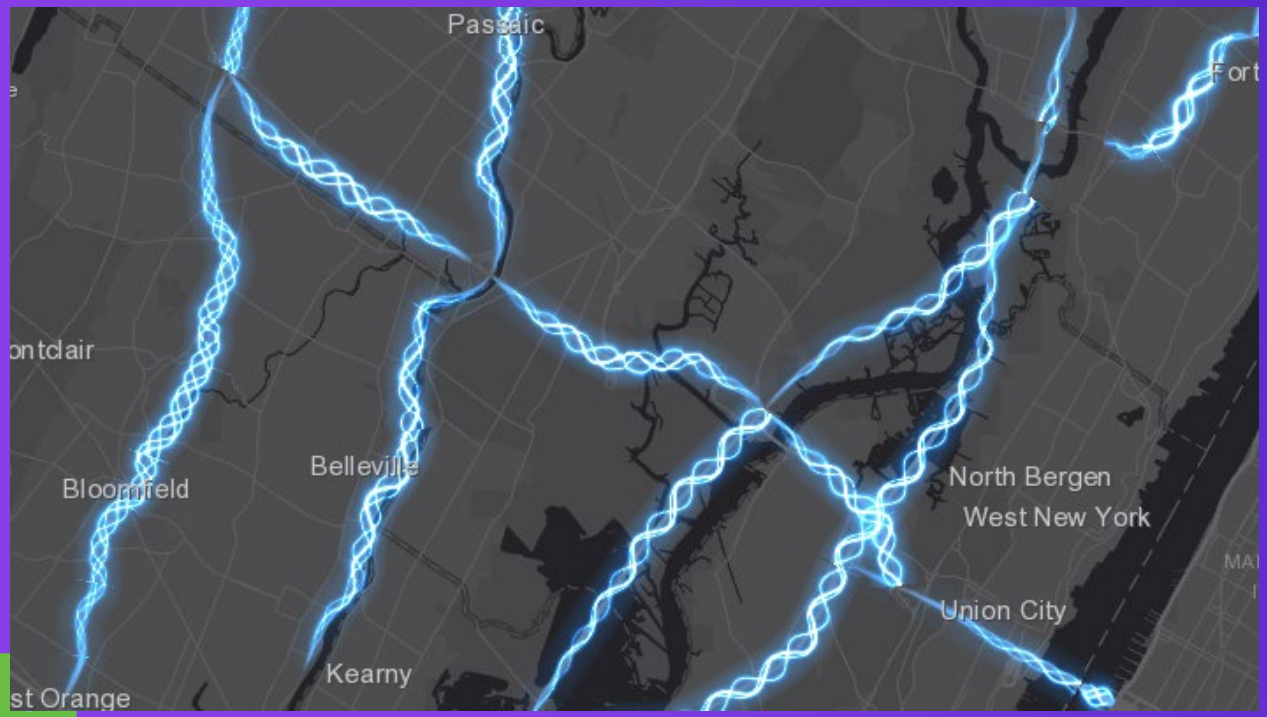
The `hitTest(x, y)` method

- Receives the test coordinates `(x, y)` in screen space
- Returns a `Promise` that resolves to a `Graphic`

```
1 hitTest: function (x, y) {
2   var hit, view = this.view, layer = this.layer;
3   layer.graphics.forEach(function (graphic) {
4     var screenPoint = view.toScreen(graphic.geometry);
5     var dx = x - screenPoint.x, dy = y - screenPoint.y;
6     if (Math.sqrt(dx * dx + dy * dy) < 35) {
7       hit = graphic;
8       hit.sourceLayer = layer;
9     }
10  });
11  return hit;
12 }
13
```



Enables the default popup behavior



Electric Highways

Code walkthrough

Demo

Electric Highways

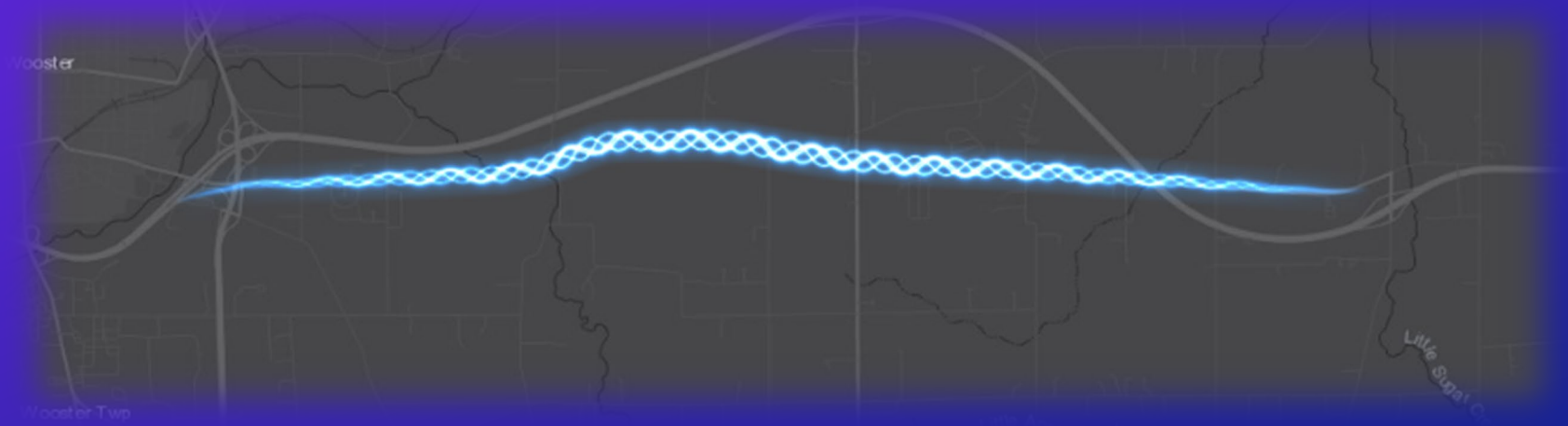
- We subclass `BaseLayerViewGL2D`
- We subclass `GraphicsLayer` to make use of its `graphics` collection

```
var CustomLayerView2D = BaseLayerViewGL2D.createSubclass({
  ...
});

var CustomLayer = GraphicsLayer.createSubclass({
  createLayerView: function(view) {
    if (view.type === "2d") {
      return new CustomLayerView2D({
        view: view,
        layer: this
      });
    }
  }
});
```

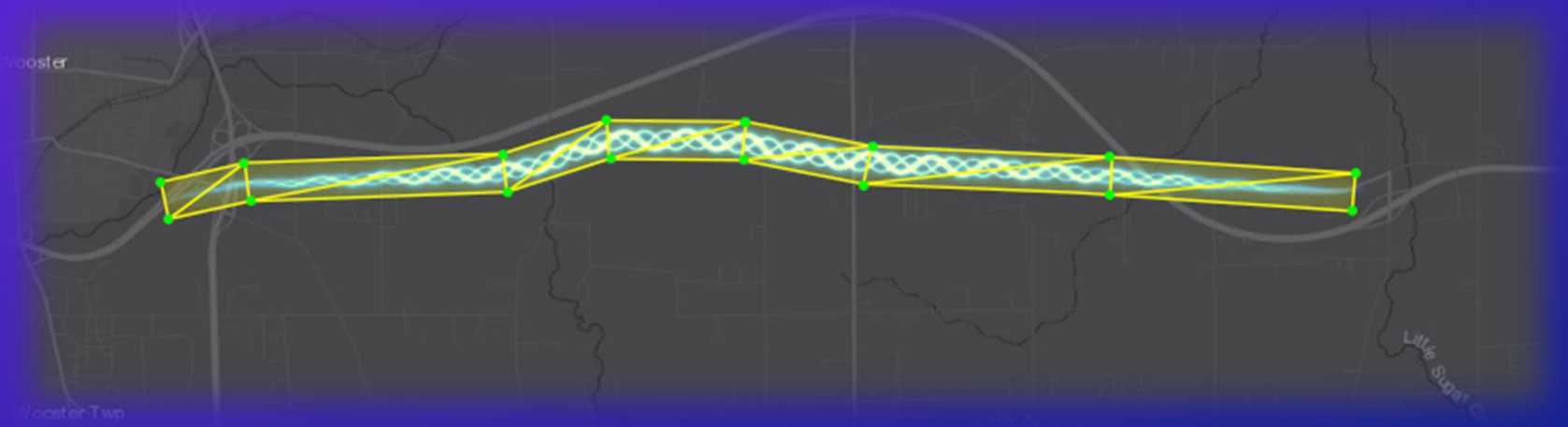
Polyline representation

- These electric lines look rather complex



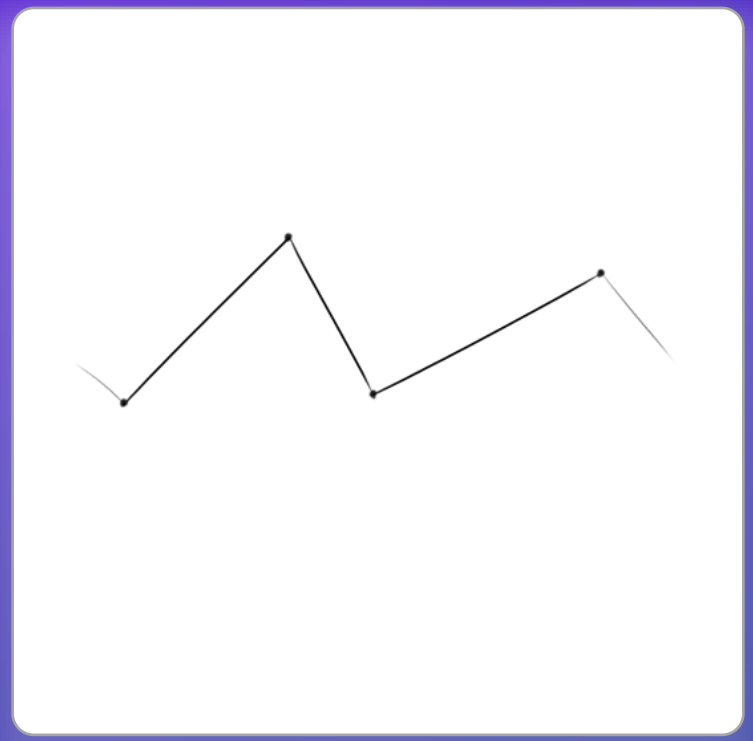
Polyline representation

- These electric lines look rather complex
  - Polylines are represented using triangle meshes
  - The electric effect is just a fancy fragment shader applied on top of a simple geometry



Polyline representation

- Each polyline vertex is represented using two GPU vertices having the same position
  - Positions are expressed in map units
- The two GPU vertices are associated to different offset vectors
  - Offset are expressed in pixels



Extrusion in the vertex shader

- The `u_transform` matrix is applied to vertex position: *Map units → Pixels*
- The `u_rotation` matrix is applied to vertex offset: *Pixels → Pixels*
- The `u_display` matrix is applied to their sum: *Pixels → NDC*

```
1 float getScaleAttenuation() {
2   return min(u_scale < 2000000.0 ? 1.0 : u_scale / 2000000.0, 10.0);
3 }
4
5 void main() {
6   vec3 transformedOffset = u_rotation * vec3(a_offset / getScaleAttenuation(), 0.0);
7   gl_Position.xy = (u_display * (u_transform * vec3(a_position, 1.0) + transformedOffset)).xy;
8   gl_Position.zw = vec2(0.0, 1.0);
9   ...
10 }
```

Update matrices

```
mat3.identity(this.transform);
this.screenTranslation[0] = (state.pixelRatio * state.size[0]) / 2;
this.screenTranslation[1] = (state.pixelRatio * state.size[1]) / 2;
mat3.translate(
  this.transform,
  this.transform,
  this.screenTranslation
);
mat3.rotate(
  this.transform,
  this.transform,
  (Math.PI * state.rotation) / 180
);
mat3.scale(this.transform, this.transform, [
  state.pixelRatio / state.resolution,
  -state.pixelRatio / state.resolution
]);
mat3.translate(
  this.transform,
  this.transform,
  this.translationToCenter
);
```

```
mat3.identity(this.rotation);
mat3.rotate(
  this.rotation,
  this.rotation,
  (Math.PI * state.rotation) / 180
);
```

```
mat3.identity(this.display);
mat3.translate(this.display, this.display, [-1, 1]);
mat3.scale(this.display, this.display, [
  2 / (state.pixelRatio * state.size[0]),
  -2 / (state.pixelRatio * state.size[1])
]);
```

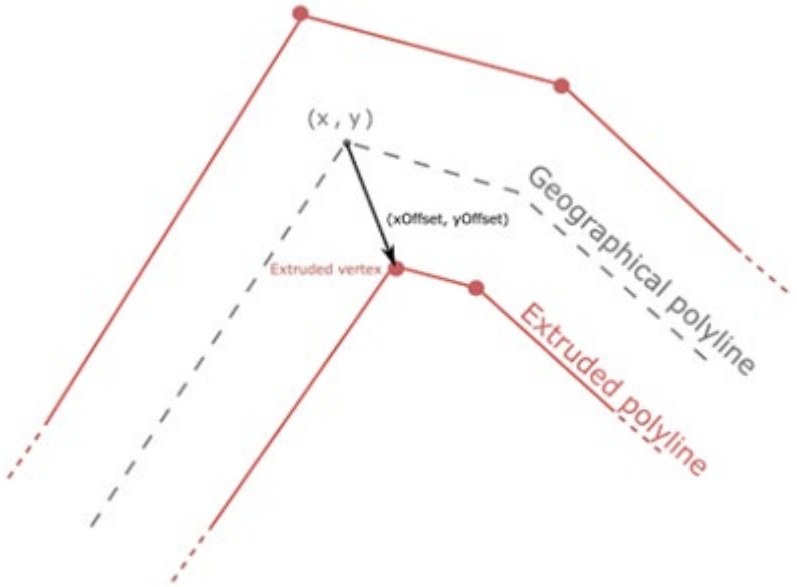
Conversion of graphics into meshes

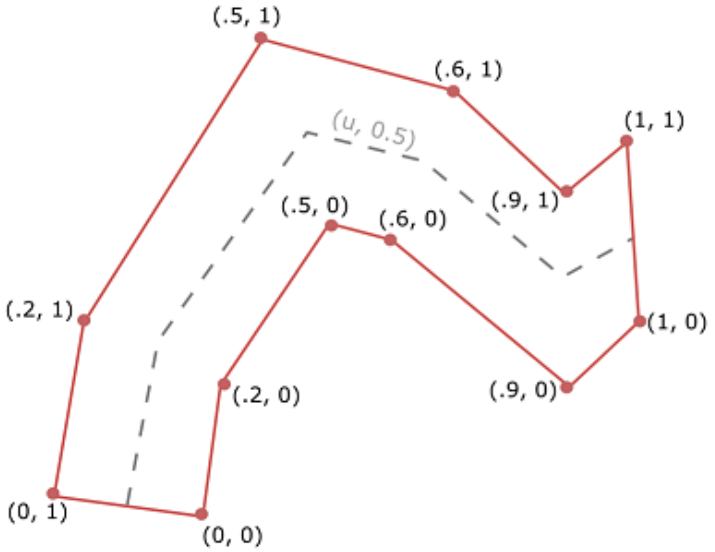
- Use the `BaseLayerViewGL2D.tessellatePolyline` method
  - Takes care of feature normalization and spatial reference reprojection
- The resulting `mesh` object contains a `vertices` and an `indices` array
  - It is up to us to encode the vertices in memory buffers and upload them to the GPU

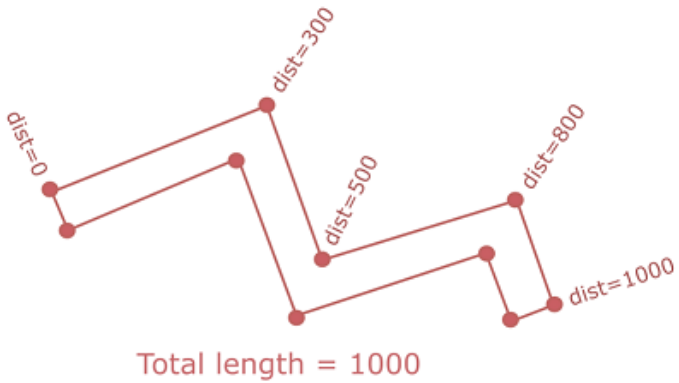
```
processGraphic: function(g) {
  this.promises.push(
    this.tessellatePolyline(g.geometry, 30)
    .then(function(mesh) {
      return {
        mesh: mesh,
        attributes: g.attributes,
        symbol: g.symbol
      };
    })
  );
}
```

Attributes returned by `tessellatePolyline`

- Position `(x, y)`
- Offset vector `(xOffset, yOffset)`
- Distance from the start `distance`
- Texture coordinates `(uTexcoord, vTexcoord)`







Encoding vertex data into buffers

```
1 var seed1 = Math.floor(random( ) * 256);
2 var seed2 = Math.floor(random( ) * 256);
3 var seed3 = Math.floor(random( ) * 256);
4 var seed4 = Math.floor(random( ) * 256);
5
6 var length = mesh.vertices.length && mesh.vertices[mesh.vertices.length - 1].distance;
7
8 for (var i = 0; i < mesh.vertices.length; ++i) {
9   var v = mesh.vertices[i];
10  vertexData[currentVertex * 12 + 0] = v.x;
11  vertexData[currentVertex * 12 + 1] = v.y;
12  vertexData[currentVertex * 12 + 2] = v.xOffset;
13  vertexData[currentVertex * 12 + 3] = v.yOffset;
14  vertexData[currentVertex * 12 + 4] = v.uTexcoord;
15  vertexData[currentVertex * 12 + 5] = v.vTexcoord;
16  vertexData[currentVertex * 12 + 6] = v.distance;
17  vertexData[currentVertex * 12 + 7] = seed1;
18  vertexData[currentVertex * 12 + 8] = seed2;
19  vertexData[currentVertex * 12 + 9] = seed3;
20  vertexData[currentVertex * 12 + 10] = seed4;
21  vertexData[currentVertex * 12 + 11] = length;
22  currentVertex++;
23 }
```

Geographical position

Offset vector

Texture coordinates

Distance from the start

Randomize effect behavior

Total length of polyline

Encoding vertex data into buffers

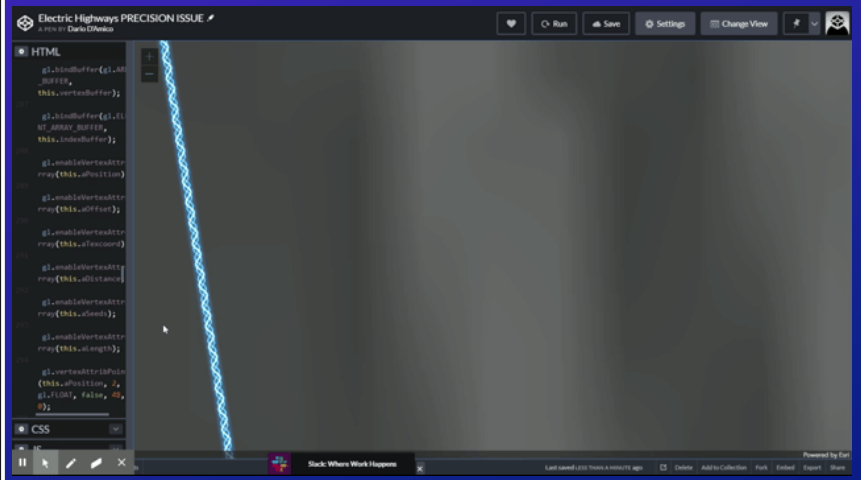
```
var seed1 = Math.floor(random() * 256);
var seed2 = Math.floor(random() * 256);
var seed3 = Math.floor(random() * 256);
var seed4 = Math.floor(random() * 256);

var length = mesh.vertices.length && mesh.vertices[mesh.vertices.length - 1].distance;

for (var i = 0; i < mesh.vertices.length; ++i) {
  var v = mesh.vertices[i];
  vertexData[currentVertex * 12 + 0] = v.x;
  vertexData[currentVertex * 12 + 1] = v.y;
  vertexData[currentVertex * 12 + 2] = v.xOffset;
  vertexData[currentVertex * 12 + 3] = v.yOffset;
  vertexData[currentVertex * 12 + 4] = v.uTexcoord;
  vertexData[currentVertex * 12 + 5] = v.vTexcoord;
  vertexData[currentVertex * 12 + 6] = v.distance;
  vertexData[currentVertex * 12 + 7] = seed1;
  vertexData[currentVertex * 12 + 8] = seed2;
  vertexData[currentVertex * 12 + 9] = seed3;
  vertexData[currentVertex * 12 + 10] = seed4;
  vertexData[currentVertex * 12 + 11] = length;
  currentVertex++;
}
```

Really big numbers: ~20,000,000

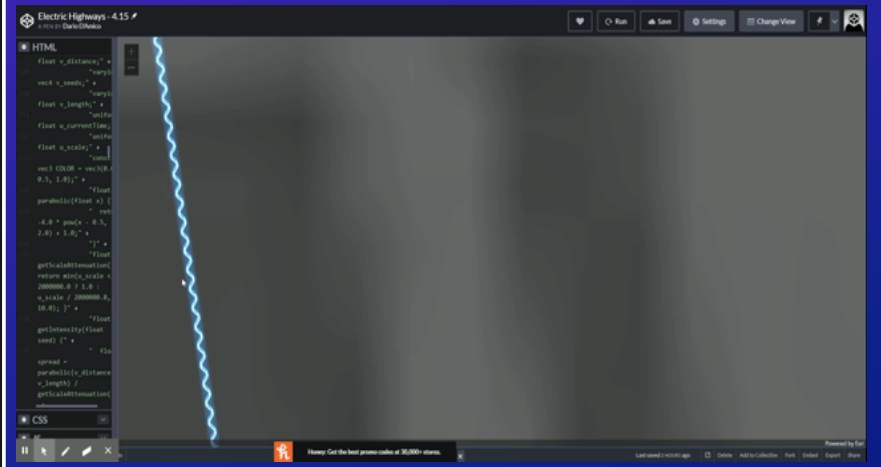
Precision issues with FP32 hardware



Encoding vertex data into buffers

```
1 var seed1 = Math.floor(random() * 256);
2 var seed2 = Math.floor(random() * 256);
3 var seed3 = Math.floor(random() * 256);
4 var seed4 = Math.floor(random() * 256);
5
6 var length = mesh.vertices.length && mesh.vertices[mesh.vertices.length - 1].distance;
7
8 for (var i = 0; i < mesh.vertices.length; ++i) {
9   var v = mesh.vertices[i];
10  vertexData[currentVertex * 12 + 0] = v.x - this.centerAtLastUpdate[0];
11  vertexData[currentVertex * 12 + 1] = v.y - this.centerAtLastUpdate[1];
12  vertexData[currentVertex * 12 + 2] = v.xOffset;
13  vertexData[currentVertex * 12 + 3] = v.yOffset;
14  vertexData[currentVertex * 12 + 4] = v.uTexcoord;
15  vertexData[currentVertex * 12 + 5] = v.vTexcoord;
16  vertexData[currentVertex * 12 + 6] = v.distance;
17  vertexData[currentVertex * 12 + 7] = seed1;
18  vertexData[currentVertex * 12 + 8] = seed2;
19  vertexData[currentVertex * 12 + 9] = seed3;
20  vertexData[currentVertex * 12 + 10] = seed4;
21  vertexData[currentVertex * 12 + 11] = length;
22  currentVertex++;
23 }
```

Solution: adopt a **“local origin”**
Makes numbers smaller
Helps combat FP32 precision issues



Encoding vertex data into buffers

```
var seed1 = Math.floor(random() * 256);
var seed2 = Math.floor(random() * 256);
var seed3 = Math.floor(random() * 256);
var seed4 = Math.floor(random() * 256);

var length = mesh.vertices.length && mesh.vertices[mesh.vertices.length - 1].distance;

for (var i = 0; i < mesh.vertices.length; ++i) {
  var v = mesh.vertices[i];
  vertexData[currentVertex * 12 + 0] = v.x - this.centerAtLastUpdate[0];
  vertexData[currentVertex * 12 + 1] = v.y - this.centerAtLastUpdate[1];
  vertexData[currentVertex * 12 + 2] = v.xOffset;
  v
  v
  v
  v
  v
  vertexData[currentVertex * 12 + 8] = seed2;
  vertexData[currentVertex * 12 + 9] = seed3;
  vertexData[currentVertex * 12 + 10] = seed4;
  vertexData[currentVertex * 12 + 11] = length;
  currentVertex++;
}
```

Makes the numbers smaller

Helps combat FP32 precision issues

More info at: https://developers.arcgis.com/javascript/latest/sample-code/custom-gl-visuals/index.html

Triggering buffer updates

- Every time that `layer.graphics` change we regenerate the vertex buffer
- We wait until the next `render()` to actually do so

```
this.needsUpdate = false;

var requestUpdate = function() {
  this.promises = [];
  this.layer.graphics.forEach(this.processGraphic.bind(this));
  ...
}.bind(this);

this.watcher = watchUtils.on(
  this,
  "layer.graphics",
  "change",
  requestUpdate,
  requestUpdate,
  requestUpdate
);
```

Triggering buffer updates

- We also update the buffers when the view becomes stationary again
- We need to do so in order to move the local origin

```
render: function(renderParameters) {
  ...
  this.updatePositions(renderParameters);
  ...
}

updatePositions: function(renderParameters) {
  ...
  if (!stationary) {
    vec2.sub(this.translationToCenter, this.centerAtLastUpdate, state.center);
    this.requestRender();
    return;
  }
  if (
    !this.needsUpdate &&
    this.translationToCenter[0] === 0 &&
    this.translationToCenter[1] === 0
  ) {
    return;
  }
  ...
}
```

Fragment shading

```
1  float getIntensity(float seed) {
2    ...
3  }
4
5  vec4 shadeIntensity(float intensity) {
6    ...
7  }
8
9  void main() {
10   float diffuse = 0.3 * parabolic(v_position) * parabolic(v_distance / v_length);
11   float beams = getIntensity(v_seeds.x) + getIntensity(v_seeds.y) + getIntensity(v_seeds.z) + getIntensity(v_seeds.w);
12   gl_FragColor = shadeIntensity(diffuse + beams);
13 }
```

Step 1 – Getting the intensity value

Step 2 – Shading the intensity value

Done 4 times with different seeds

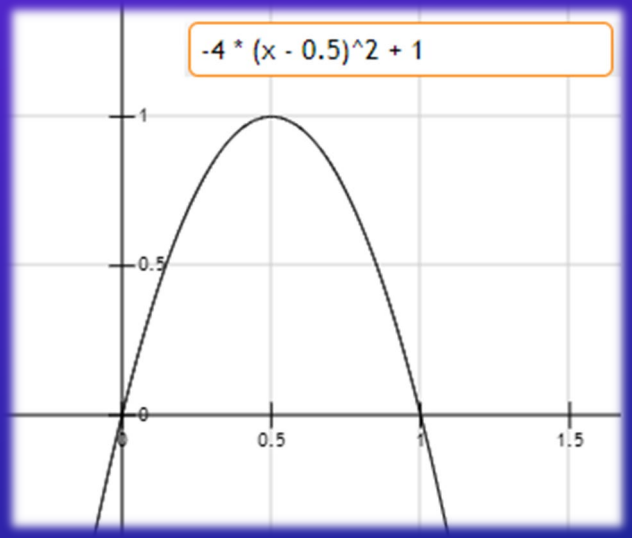
Fragment shading

```
1 float parabolic(float x) {
2   return -4.0 * pow(x - 0.5, 2.0) + 1.0;
3 }
4 void main() {
5   float diffuse = 0.3 * parabolic(v_position) * parabolic(v_distance / v_length);
6   gl_FragColor = vec4(diffuse);
7 }
```

Across

Along

Produces a faint, low-intensity glow

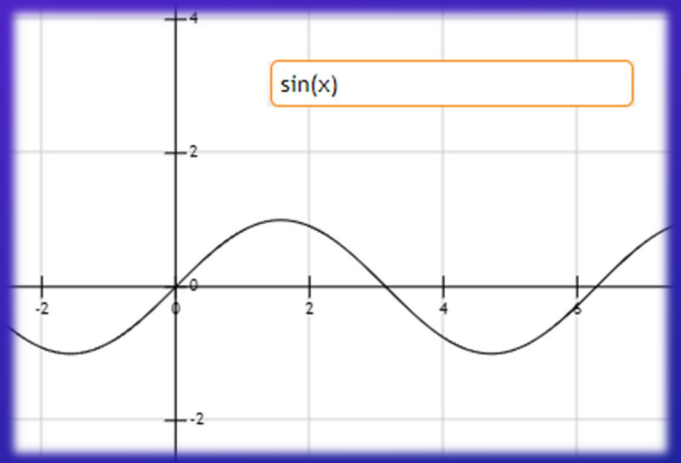


Made with http://fooplot.com/

Fragment shading

```
1 float getIntensity(float seed) {
2   float den = 2000.0 * u_scale / 1200000.0;
3   float particle = 0.5 + 0.2 * sin((v_distance) / den + seed + u_currentTime * seed / 100.0);
4   return 1.5 * exp(-30.0 * length(v_position - particle));
5 }
6 void main() {
7   float diffuse = 0.3 * parabolic(v_position) * parabolic(v_distance / v_length);
8   float beam = getIntensity(v_seeds.x);
9   gl_FragColor = vec4(diffuse + beam);
10 }
```

The sliding sine function traces a high intensity beam



Made with http://fooplot.com/

Fragment shading

```
1 float getIntensity(float seed) {
2   float den = 2000.0 * u_scale / 1200000.0;
3   float p = 0.5 + 0.2 * sin((v_distance) / den + seed + u_currentTime * seed / 100.0);
4   return 1.5 * exp(-(30.0 + 10.0 * sin((u_currentTime + seed) * 10.0)) * length(v_position - p));
5 }
```

Intensity pulses over time

Fragment shading

```
1 float getIntensity(float seed) {
2   float spread = parabolic(v_distance / v_length);
3   float den = 2000.0 * u_scale / 1200000.0;
4   float particle = 0.5 + 0.2 * spread * sin((v_distance) / den + seed + u_currentTime * seed /
  100.0);
5   return 1.5 * exp(-(30.0 + 10.0 * sin((u_currentTime + seed) * 10.0)) * length(v_position -
  particle)) * spread;
6 }
```

Makes the effect less pronounced
At the start and at the end of the line

Fragment shading

```
1 vec4 shadeIntensity(float intensity) {
2   vec3 baseColor = COLOR * clamp(intensity, 0.0, 1.0);
3   return vec4(baseColor, 0.0);
4 }
5 void main() {
6   float diffuse = 0.3 * parabolic(v_position) * parabolic(v_distance / v_length);
7   float beam = getIntensity(v_seeds.x);
8   gl_FragColor = shadeIntensity(diffuse + beam);
9 }
```

Applies an electric blue color
Note how `alpha=0` which signify an emissive source

Fragment shading

```
1 vec4 shadeIntensity(float intensity) {
2   vec3 baseColor = COLOR * clamp(intensity, 0.0, 1.0);
3   float white = max(intensity - 1.0, 0.0);
4   return vec4(vec3(white) + baseColor, 0.0);
5 }
```

The intensity in excess of 1 contributes to the bright white core of the beam

Fragment shading

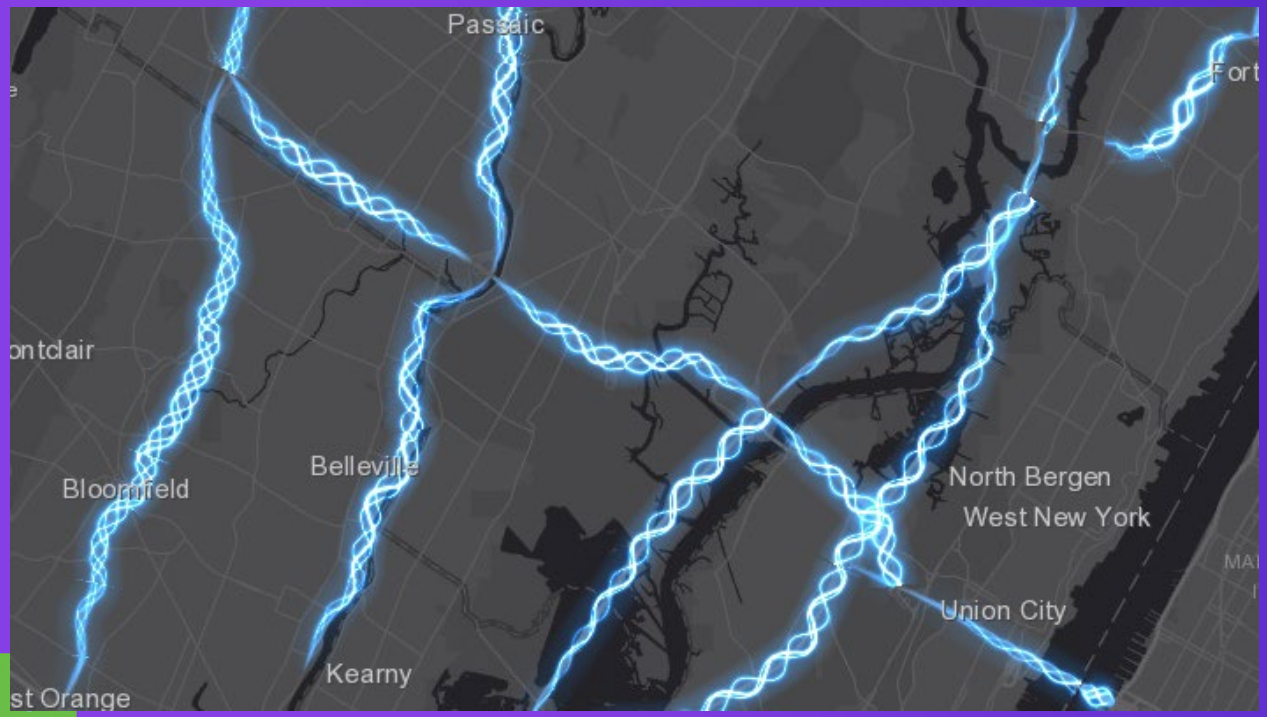
```
1 void main() {
2   float diffuse = 0.3 * parabolic(v_position) * parabolic(v_distance / v_length);
3   float beams = getIntensity(v_seeds.x) + getIntensity(v_seeds.y) + getIntensity(v_seeds.z) +
  getIntensity(v_seeds.w);
4   gl_FragColor = shadeIntensity(diffuse + beams);
5 }
```

Do the same thing for 4 beams, each driven by a different seed value

Rendering

```
1  gl.useProgram(this.program);
2  gl.uniformMatrix3fv(this.uTransform, false, this.transform);
3  gl.uniformMatrix3fv(this.uRotation, false, this.rotation);
4  gl.uniformMatrix3fv(this.uDisplay, false, this.display);
5  gl.uniform1f(this.uCurrentTime, performance.now() / 1000.0);
6  gl.uniform1f(this.uScale, state.scale);
7  gl.bindBuffer(gl.ARRAY_BUFFER, this.vertexBuffer);
8  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.indexBuffer);
9  gl.enableVertexAttribArray(this.aPosition);
10 gl.enableVertexAttribArray(this.aOffset);
11 gl.enableVertexAttribArray(this.aTexcoord);
12 gl.enableVertexAttribArray(this.aDistance);
13 gl.enableVertexAttribArray(this.aSeeds);
14 gl.enableVertexAttribArray(this.aLength);
15 gl.vertexAttribPointer(this.aPosition, 2, gl.FLOAT, false, 48, 0);
16 gl.vertexAttribPointer(this.aOffset, 2, gl.FLOAT, false, 48, 8);
17 gl.vertexAttribPointer(this.aTexcoord, 2, gl.FLOAT, false, 48, 16);
18 gl.vertexAttribPointer(this.aDistance, 1, gl.FLOAT, false, 48, 24);
19 gl.vertexAttribPointer(this.aSeeds, 4, gl.FLOAT, false, 48, 28);
20 gl.vertexAttribPointer(this.aLength, 1, gl.FLOAT, false, 48, 44);
21 gl.enable(gl.BLEND);
22 gl.blendFunc(gl.ONE, gl.ONE_MINUS_SRC_ALPHA);
23 gl.disable(gl.CULL_FACE);
24 gl.drawElements(gl.TRIANGLES, this.indexBufferSize, gl.UNSIGNED_INT, 0);
```

1. Bind program
2. Update uniforms
3. Bind buffers
4. Setup attributes
5. Enable premultiplied alpha and other states
6. Draw



Electric Highways

Code walkthrough

Demo

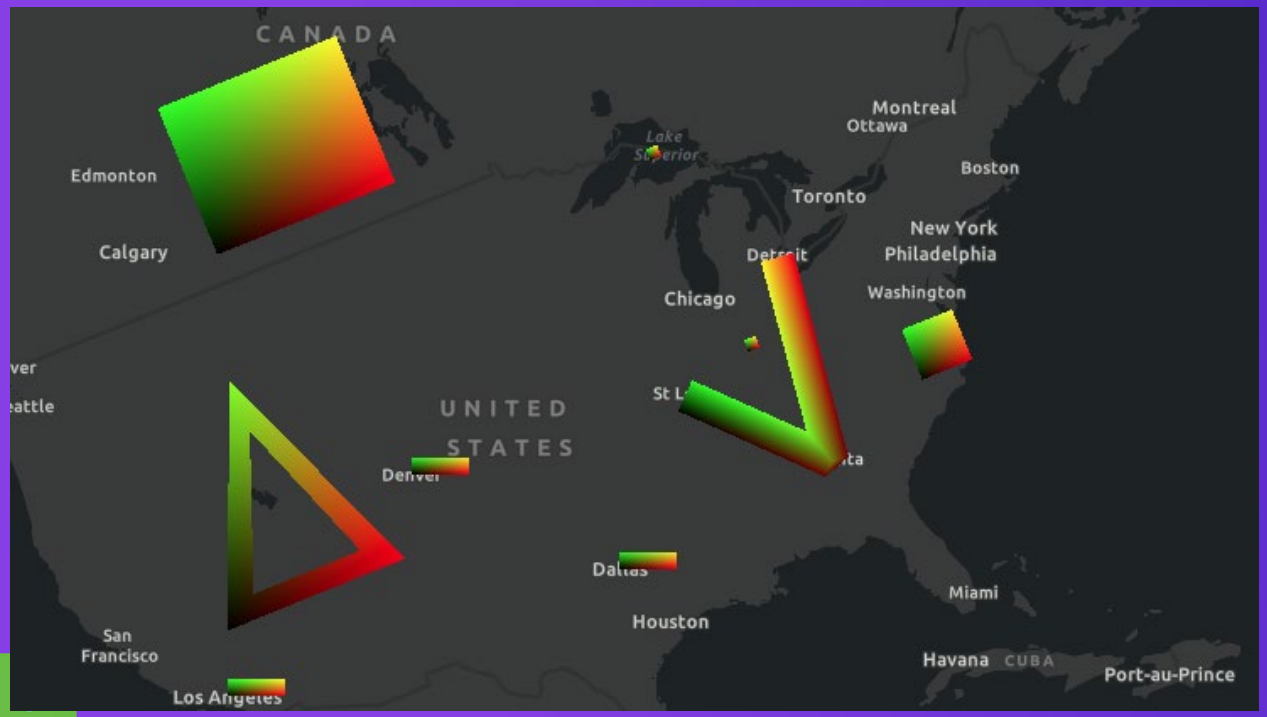
Tessellating lines and polygons

- WebGL, like other graphic APIs can really only handle triangles
  - Lines need to be triangulated
  - Polygons need to be tessellated
- We provide helper API calls to turn GIS geometries into WebGL meshes
  - `BaseLayerViewGL2D.tessellatePolyline`
  - `BaseLayerViewGL2D.tessellatePolygon`
  - `BaseLayerViewGL2D.tessellateExtent`
  - `BaseLayerViewGL2D.tessellatePoint`
  - `BaseLayerViewGL2D.tessellateMultipoint`

```
this.tessellatePolygon(polygon).then(function (mesh) {
  const indexData = new Uint32Array(mesh.indices);
  const vertexData = new Float32Array(2 * mesh.vertices.length);
  for (let i = 0; i < mesh.vertices.length; i++) {
    vertexData[2 * i] = mesh.vertices[i].x;
    vertexData[2 * i + 1] = mesh.vertices[i].y;
  }
});

...

gl.bufferData(gl.ARRAY_BUFFER, vertexData, gl.STATIC_DRAW);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indexData, gl.STATIC_DRAW);
```



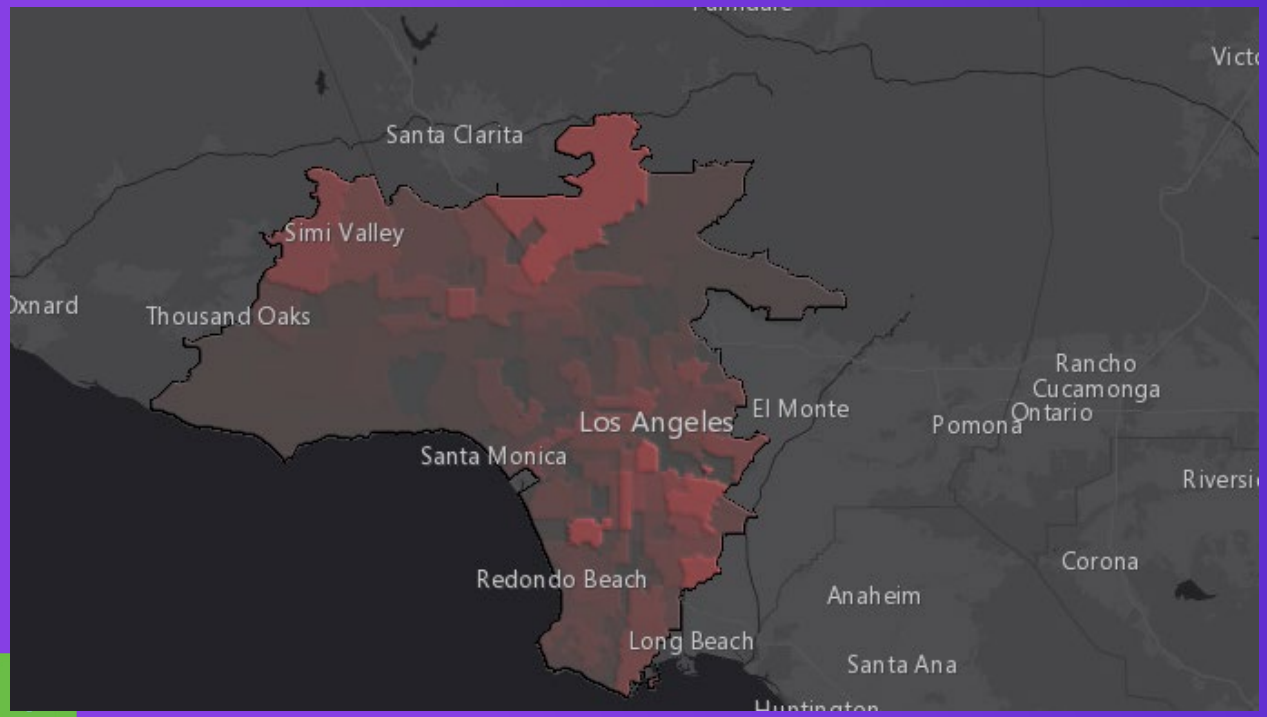
Tessellation Helpers

Creating triangle meshes for Esri geometries

SDK sample

Accessing the backbuffer

- The rendering engine uses framebuffer objects to handle layer composition onto the view
- Sometimes users need to access the backbuffer
  - Reading the content
  - Restoring it after binding a different render target
- We provide APIs to access and restore the framebuffer
  - `BaseLayerViewGL2D.getRenderTarget()`
  - `BaseLayerViewGL2D.bindRenderTarget()`



Beveled Features

Implementing multi-pass rendering with framebuffer objects

Demo

Render data using tiles

- `BaseLayerViewGL2D` provides optional support for tiled rendering
- Opt in by creating a `tileInfo` object on the layer and handling the `tilesChanged` event

```
var CustomLayer = Layer.createSubclass({
  tileInfo: TileInfo.create({ size: 512, spatialReference: { wkid: 3857 } }),

  createLayerView(view) {
    if (view.type === "2d") {
      return new CustomLayerView2D({
        view: view,
        layer: this
      });
    }
  }
});
```

```
const CustomLayerView2D = BaseLayerViewGL2D.createSubclass({
  ...
  render(renderParameters) {
    ...
    for (let i = 0; i < this.tiles.length; i++) {
      // Retrieve the current tile and its associated texture.
      const tile = this.tiles[i];
      ...
      // Draw the tile.
      gl.drawElements(gl.TRIANGLES, 6, gl.UNSIGNED_SHORT, 0);
    }
  },
  ...
  tilesChanged() {
    // Respond to a change in the tile screen coverage, for
    // instance by fetching the required data from a server.
  }
});
```

Tiling Support

Render data by tiles and solve FP32 precision issues

SDK sample

Integration with 3rd party libraries

- We try to make it easy to integrate 3rd party rendering engines and GIS visualization libraries into the JavaScript API
- We worked on integrating deck.gl, a geospatial visualization toolkit developed by Uber
- We are open to requests for integration with more libraries

Using `@deck.gl/arcgis`

Integration with 3rd party libraries

- We try to make it easy to integrate 3rd party rendering engines and GIS visualization libraries into the JavaScript API
- We worked on integrating deck.gl, a geospatial visualization toolkit developed by Uber
- We are open to requests for integration with more libraries

Using `@deck.gl/arcgis`

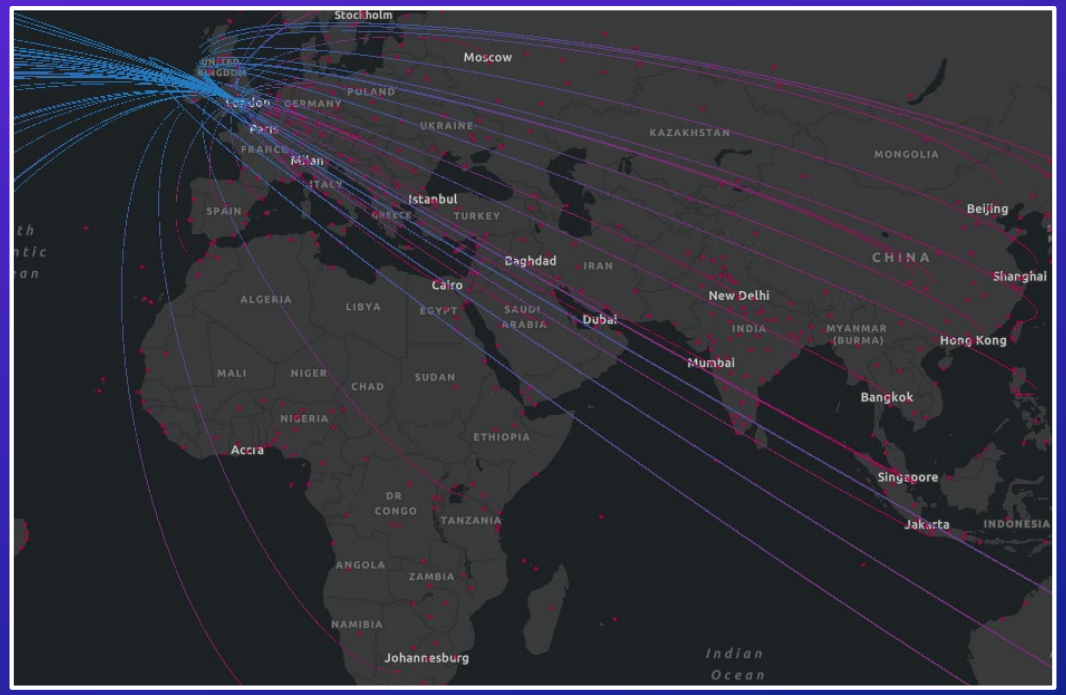
Under the hood we create a custom layer view which provides the rendering context and the transformations to deck.gl

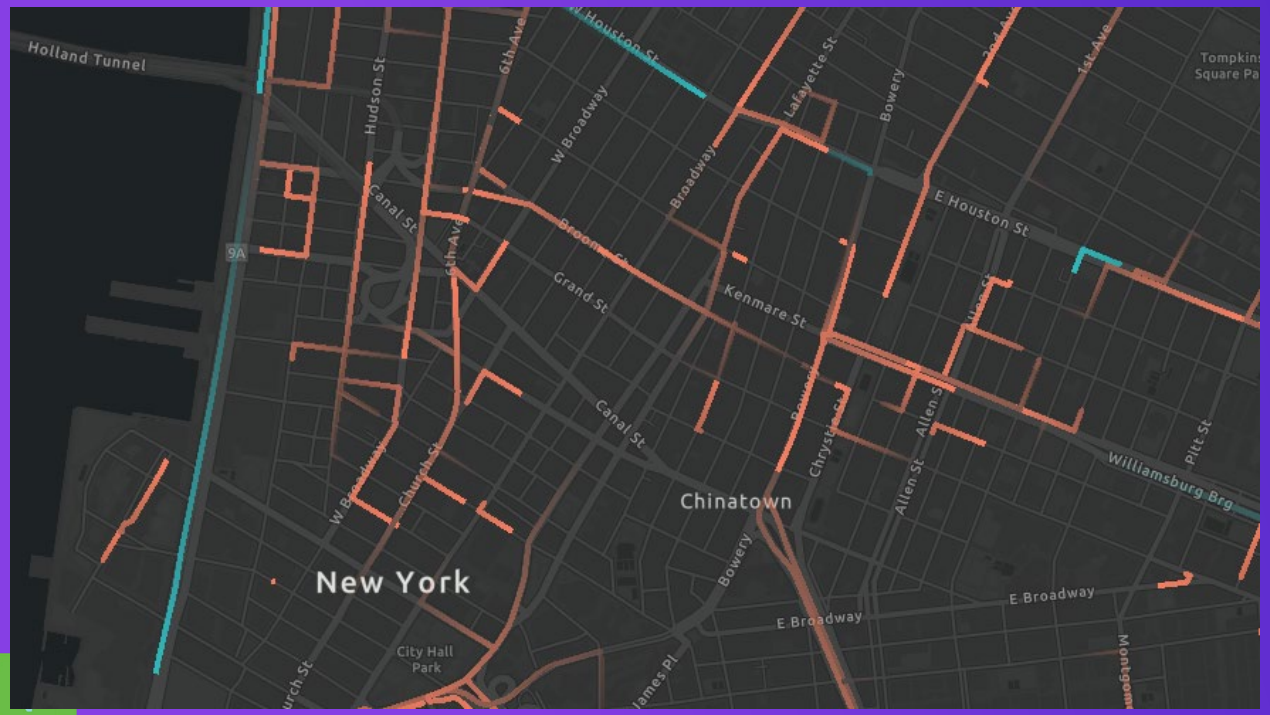
```
1    <script src="https://unpkg.com/deck.gl@8.1.0-beta.3/dist.min.js"></script>
2    <script src="https://unpkg.com/@deck.gl/arcgis@8.1.0-beta.3/dist.min.js"></script>
3    <link rel="stylesheet" href="https://js.arcgis.com/4.15/esri/themes/light/main.css"/>
4    <script src="https://js.arcgis.com/4.15/"></script>
```

Using `@deck.gl/arcgis`

```
require(["esri/Map", "esri/views/MapView"], function(Map, MapView) {
  deck.loadArcGISModules().then(function (arcGIS) {
    const layer = new arcGIS.DeckLayer();
    layer.deck.layers = [
      new deck.GeoJsonLayer({
        ...
      }),
      new deck.ArcLayer({
        ...
      })
    ];

    const mapView = new MapView({
      container: "viewDiv",
      map: new Map({
        basemap: "dark-gray-vector",
        layers: [layer]
      })
    });
  });
});
```





deck.gl in `MapView`

SDK sample

Integration with luma.gl

- Low-level WebGL toolkit
- Increased productivity, but still flexible enough to do anything
- Provides full access to WebGL2
  - Fallbacks to WebGL 1 when possible
- Part of the deck.gl distribution

Using `luma.gl`

- Import the main deck.gl module
- luma.gl will be available on a global `luma` object

```
1 <script src="https://unpkg.com/deck.gl@8.1.0-beta.3/dist.min.js"></script>
2 <link rel="stylesheet" href="https://js.arcgis.com/4.15/esri/themes/dark-green/main.css"/>
3 <script src="https://js.arcgis.com/4.15/"></script>
```

Instrumenting the WebGL context

```
1 attach: function() {
2   var gl = this.context;
3
4   ...
5   luma.instrumentGLContext(gl);
6   ...
7 }
```

The MapView WebGL context must be augmented in order to be usable in conjunction with luma.gl

Geometry and models in luma.gl

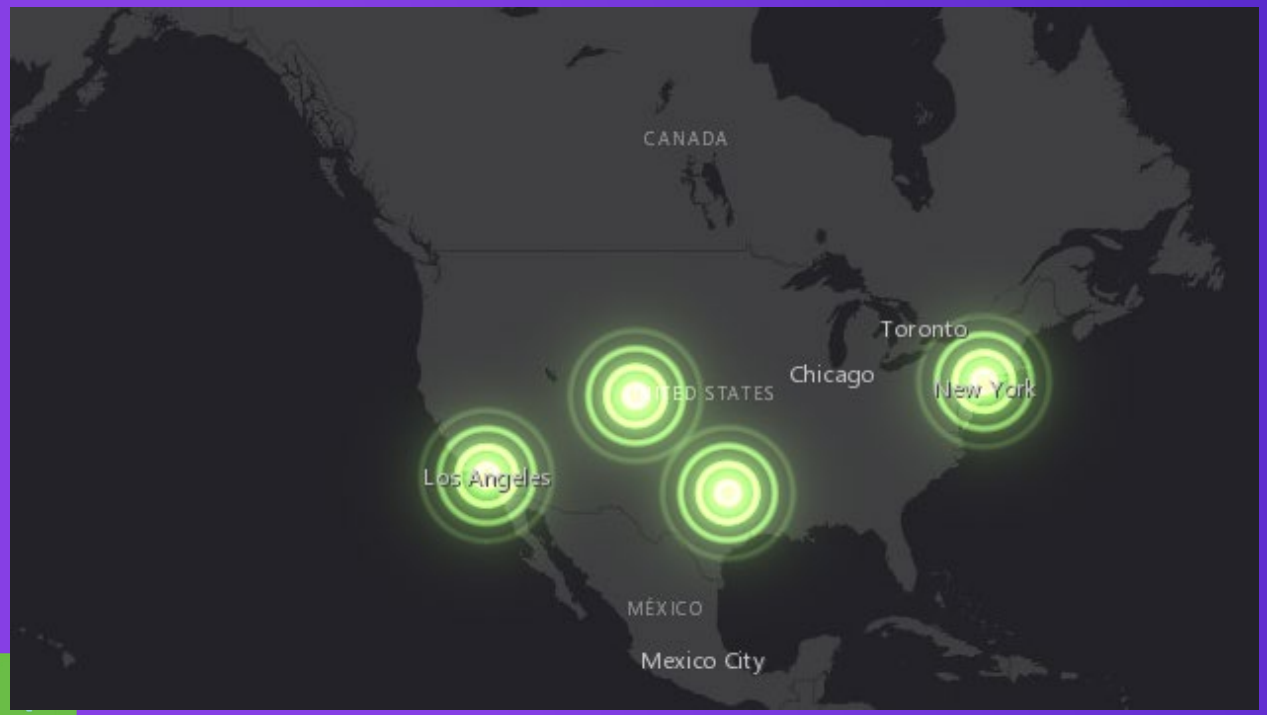
```
var geometry = new luma.Geometry({
  drawMode: gl.TRIANGLES,
  attributes: {
    a_position: {
      size: 2,
      value: positionData
    },
    a_offset: {
      size: 2,
      value: offsetData
    }
  },
  indices: indexData
});
```

```
var model = new luma.Model(gl, {
  vs: this.vertexSource,
  fs: this.fragmentSource,
  geometry: geometry,
  uniforms: {
    u_transform: this.transform,
    u_display: this.model,
    u_current_time: performance.now() / 1000.0
  }
});
```

Rendering in luma.gl

```
1 luma.withParameters(gl, {
2   blend: true,
3   blendFunc: [gl.ONE, gl.ONE]
4 }, function () {
5   model.setUniforms({
6     u_transform: transform,
7     u_display: display,
8     u_current_time: performance.now() / 1000.0
9   });
10
11  model.draw();
12 });
```

luma.gl provides ways to manage the WebGL state, abstracting away its global nature and making coding less error-prone



Using luma.gl

Demo

Integration with Three.js

- High-level 3D WebGL rendering engine
- Dramatically reduces the amount of LOC needed

Using `Three.js`

- Import the main module and any needed additional module
- Core classes and additional classes will be both available on the global `THREE` object

```
<script src="https://threejs.org/build/three.js"></script>
<script src="https://threejs.org/examples/js/loaders/GLTFLoader.js"></script>
<link rel="stylesheet" href="https://js.arcgis.com/4.15/esri/themes/dark-green/main.css"/>
<script src="https://js.arcgis.com/4.15/"></script>
```

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000);
...
var loader = new THREE.GLTFLoader();
```

Making MapView and Three.js play nice

```
// In attach()
var renderer = new THREE.WebGLRenderer({
  context: this.context,
  alpha: true,
  transparent: true,
  premultipliedAlpha: true
});
renderer.autoClearDepth = true;
renderer.autoClearStencil = true;
renderer.autoClearColor = false;
renderer.setClearColor(0x000000, 0);
```

Prevents Three.js from erasing the map

```
// In render()
renderer.state.reset();
renderer.render(scene, camera);
```

Informs Three.js that the MapView is resetting the state every frame

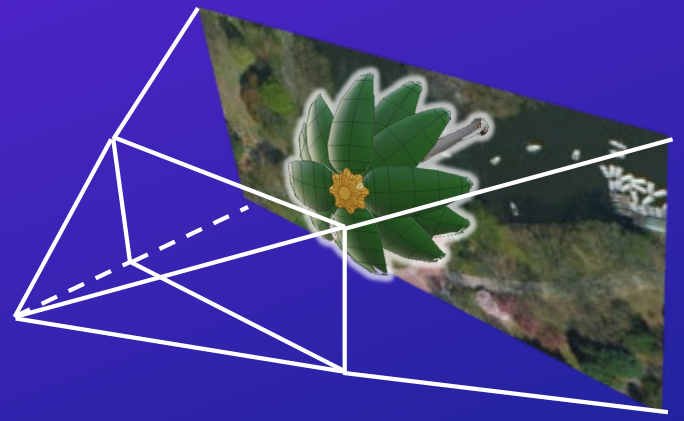
Matching the 3D camera with the 2D viewpoint

```
camera.aspect = width / height;
var fovyRadians = Math.PI * camera.fov / 180;
camera.far = (height * state.resolution) / (2 * Math.sin(fovyRadians / 2));
camera.updateProjectionMatrix();

this.graphicToObject.forEach(function (o, g) {
  var screenPoint = view.toScreen({
    x: g.geometry.longitude,
    y: g.geometry.latitude,
    spatialReference: {
      wkid: 4326
    }
  });

  var vector = new THREE.Vector3(
    2 * screenPoint.x / width - 1,
    1 - 2 * screenPoint.y / height,
    1 // z === 1 means that this is a point on the far plane
  ).unproject(camera);
  o.position.x = vector.x;
  o.position.y = vector.y;
  o.position.z = vector.z;
  o.rotation.z = -Math.PI * state.rotation / 180;
});
```

Set the distance of the far plane so that the back of the viewing frustum is a rectangle that matches the extent



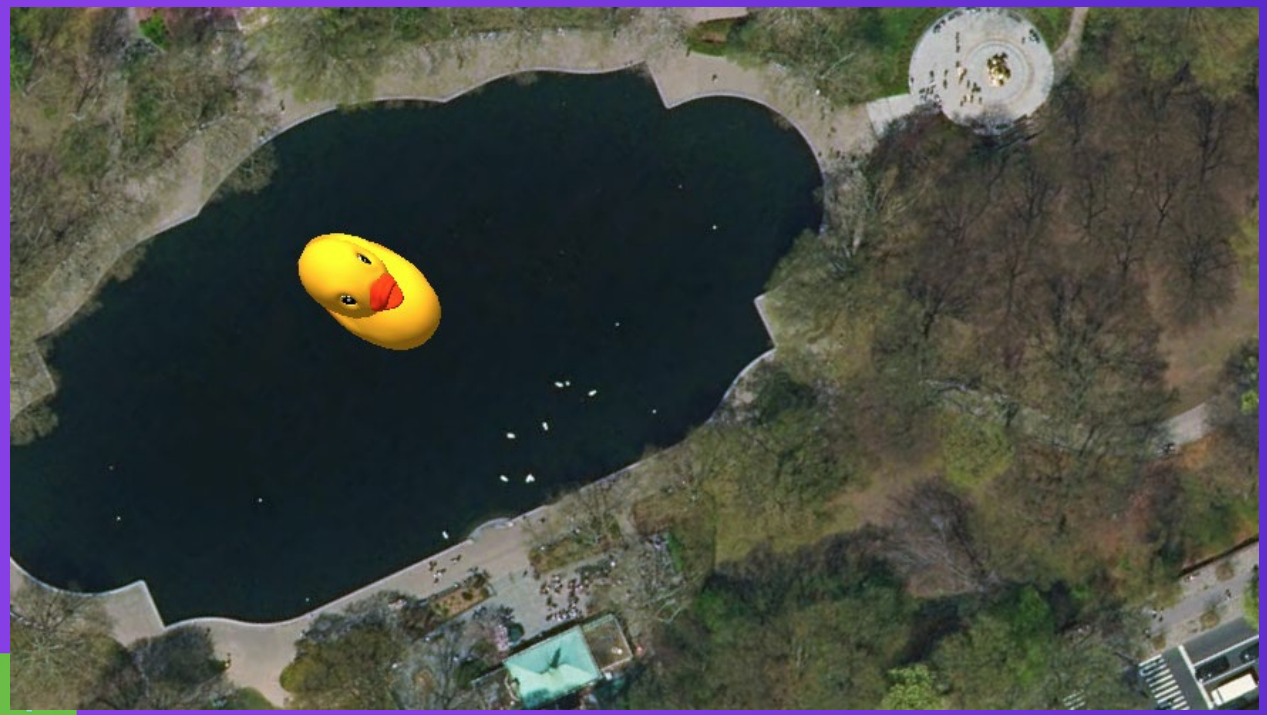
Place your scene objects on the far plane

Hit testing Three.js scenes

- You can implement `BaseLayerViewGL2D.hitTest` using `THREE.Raycaster`
- Be careful that 3D objects are often hierarchical
  - The raycast operation may return a child of the object that you would like to be returned
- Then create/lookup and return a `esri/Graphic` instance that correspond to that object

```
1 var raycaster = new THREE.Raycaster();
2 var width = view.state.pixelRatio * view.state.size[0];
3 var height = view.state.pixelRatio * view.state.size[1];
4 raycaster.setFromCamera(new THREE.Vector3((x / width) * 2 - 1, -(y / height) * 2 + 1, -1), camera);
5 var hit = raycaster.intersectObjects(scene.children, true)[0].object;
```

Recursive



Using Three.js

Demo

WebGL gotchas

- Numeric precision
  - WebGL can only handle 32-bit floating point
  - Not enough precision for geographical applications
  - WebMercator coordinate range spans ~40’000’000
  - IEEE standard for floating point reserves 23 bits for the number part, meaning that 32-bit floating point numbers cannot maintain the precision required to store the coordinates.
  - Solution: use local origin or resort to using tiling
- State-machine nature of WebGL
  - Custom layer views share the same rendering context with the rest of the `MapView`
  - The state is reset before handing control over to `render()`
  - Implementors cannot assume that the WebGL state is persisted across `render()` invocations
- Performance suffers when making WebGL calls outside of `requestAnimationFrame`
  - Beware of asynchronous JavaScript calls
  - Only dispatch WebGL commands from within `attach()`, `detach()` and `render()`

Thank you!

- **SDK samples**
  - **Tessellation helpers:** https://developers.arcgis.com/javascript/latest/sample-code/custom-gl-tessellation-helpers/index.html
  - **Tiling support:** https://developers.arcgis.com/javascript/latest/sample-code/custom-gl-tiles/index.html
  - **Using deck.gl:** http://developers.arcgis.com/javascript/latest/sample-code/custom-lv-deckgl/index.html

- **CodePen samples**
  - https://codepen.io/collection/XRjrYm