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

Yaron Fine, Dario D’Amico

2021 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
  - Blend and layer-effects!

Custom layers and layer views

- In the JS 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 for writing custom 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

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

```
1 var CustomLayer = Layer.createSubclass({
2   createLayerView: function (view) {
3     if (view.type === "2d") {
4       return new CustomLayerView2D({ view: view, layer: this });
5     }
6   }
7 });
```

Implementing a custom layer with WebGL

If using tiling:

```
var CustomTileLayer = Layer.createSubclass({
  tileInfo: TileInfo.create({ spatialReference: { wkid: 3857 }}),
  createLayerView(view) {
    if (view.type === "2d") {
      return new CustomLayerView2D({
        view: view,
        layer: this
      });
    }
  }
});
```

- 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 `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()`

Final rendering must target the correct framebuffer!

The `MapView` binds it for you right before calling into `render()`

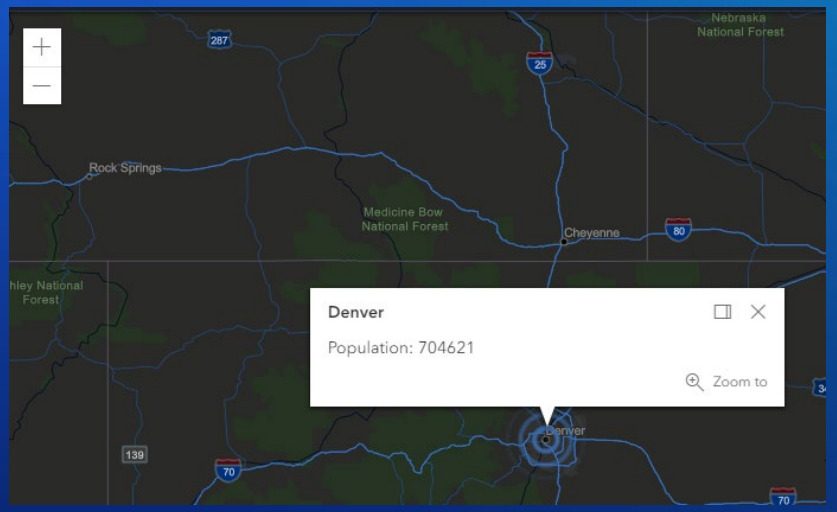
If you don't call `gl.bindFramebuffer` nor `gl.viewport` you don't need to do anything

But if you do, then you need to call `this.bindRenderTarget()` and do a final composite there

The `hitTest(x, y)` method

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

```
hitTest: function (x, y) {
  var hit, view = this.view, layer = this.layer;
  layer.graphics.forEach(function (graphic) {
    var screenPoint = view.toScreen(graphic.geometry);
    var dx = x - screenPoint.x, dy = y - screenPoint.y;
    if (Math.sqrt(dx * dx + dy * dy) < 35) {
      hit = graphic;
      hit.sourceLayer = layer;
    }
  });
  return hit;
}
```

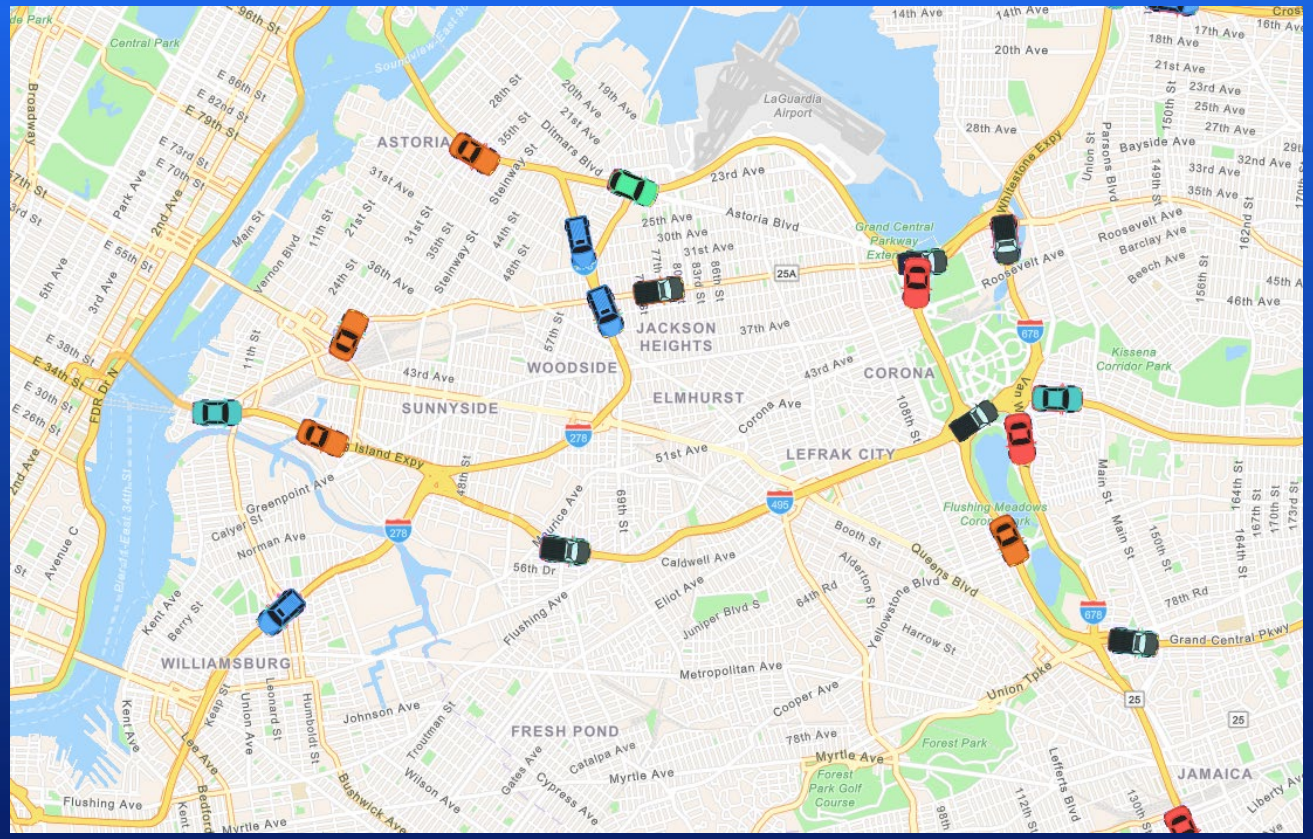


Enables the default popup behavior

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);
```



Tracking cars

Code walkthrough

DEMO

The code for the walkthrough and the stream service can be found here:
stream-service: https://github.com/yaronfine/node-stream-service
demo app: https://github.com/yaronfine/devsummit-2021-demos

Tracking cars

- Objective:

  Consume a dynamic feed of features (cars) through a WebSocket and render the cars on the map

  - Large number of cars (10K or more!)
  - High frequency of updates, each car position is updated every 16MS by the service
  - Must support hittesting
  - Must support highlighting “active” cars

Design considerations:

- Car data in nature is extremely dynamic (sub-second update rate)
- We only need to render markers (points)
- Multiple types of features (car-types), need more than one symbol
- Scale symbols according to the scale/zoom level
- Features must be anchored to their geographic position and can't "wiggle" when zooming or panning.

Tracking cars

Alternatives for implementation:

I. Use tiling architecture

Pros

- Provide multi-level representation for features. Works exceptionally well for lines and fills
- Address WebGL's 32bit precision issue
- Allow GPU representation of feature coordinates using short integers (relying on quantization)
- Very good balance of GPU memory, only the content of the visible tiles is uploaded to the GPU
- Take full advantage of WebGL's retained rendering mode
- Trivial solution for wrap-around

Cons

- Requires complex data structures in order to process and distribute the data to the right tiles at a given time
- Rendering a given scene using tiles requires more draw calls than without tiles
- dealing with Z-order of features can be difficult, particularly when features update over time
- Special logic (and extra churn) is required to avoid features from getting clipped at the edge of tiles
- Updating features is both complicated and CPU/GPU intensive (a lot of housekeeping is required)

Tracking cars

Alternatives for implementation:

II. Single buffer and a local origin

Pros

- Does not require complex data structures in order to manage and render features
- Rendering a given scene does not necessarily require many draw-calls
- Updating is much less complicated
- Can take advantage of WebGL's data streaming to the GPU, allow implementing drawing in `immediate` mode

Cons

- Susceptible to WebGL's 32 bit precision issue if not handled properly (local-origin)
- Since there is no multi-representation at different levels, can lead to artifacts when rendering lines and fills
- Hard to address wrap-around
- Can be wasteful in term of GPU memory. Fit all the data on the GPU even if outside the area of interest
- Encoding position requires 32 bit floating point per ordinate (twice as much compared to shorts when using tiles)

Tracking cars

For this demo, we chose to use a single buffer and a local origin (option II).

Why?

- We are required to render only markers, no lines or fills
- Data is extremely dynamic, features update many times a second, we need to issue frequent updates effectively every draw-cycle
- We don't really care about wrap-around in this case

Tracking cars

How can we mitigate the shortcoming of using a single representation?

- Maintain the data in a simple, naive data structure (Map of *Id* to *feature*), no RTree, QuadTree etc.
- Create the buffer data with a designated stream mode
- Update the entire buffer with each update (once per draw-cycle)
  - Accumulate updates in a queue and issue a full update, per update cycle
- Update the local origin every time that we push the data to the GPU (almost every draw-cycle)
- Clip out features that are outside the visible extent
- Pack all other (except position) per-vertex attributes as tightly as possible, keep GPU memory as low as possible
- Use a texture-atlas in order to minimize the number of state changes and draw-calls

Tracking cars

- We subclass `BaseLayerViewGL2D`
- Implement a CustomLayer hosting a `WebSocketConnection`

```
@subclass("CustomFeatureLayer")
export default class CustomFeatureLayer extends declared(Layer) {
  ...
  // ------------------------------------------------------------------------
  //
  //  Public methods
  //
  // ------------------------------------------------------------------------
  connect(config: IWebSocketConnectionConfig, onConnectionStatusChange: (status: string) => void) {
    if (this.connected) {
      this._connection.destroy();
    }

    this._connection = new WebSocketConnection(config, (feature: Feature) => this.emit("onFeature", feature),
onConnectionStatusChange);
  }

  disconnect(): void {
    if (this.connected) {
      this._connection.destroy();
    }
  }

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

Tracking cars - implementing LayerView

An *update queue* and an *update()* call

- messages arrive from the layer at a very high speed. A queue regulates the number of updates and avoid overwhelming the system
- Provide a separation between accessing system memory and WebGL memory (update vs draw)
- During a call to *update()*, process all the memory needed to upload to the GPU. The actual upload will happen on the next draw

```
// start the update cycle
this._updateTimer = window.setInterval(() => this._doUpdate(), 16);
...
// on each feature update
private _onFeature(feature: Feature): void {
  if (!feature) {
    return;
  }

  const { attributes } = feature;
  const trackId = attributes["TRACKID"];


  this._featuresToUpdate.set(trackId, feature);
  this._updateRequested = true;
}
// The update method, called every 16ms
private _doUpdate(): void {
 if (!this._updateRequested) {
   return;
 }

 const featuresToUpdateMap = this._featuresToUpdate;
 const features = this._features;

 const updateBuffer = featuresToUpdateMap.size > 0;
 if (!updateBuffer) {
   return;
 }

 const { extent, viewpoint } = this.view;
 const featuresToUpdate = featuresToUpdateMap.entries();
 let updateResult = featuresToUpdate.next();
 while (!updateResult.done) {
   const [trackId, feature] = updateResult.value;
   features.set(trackId, feature);
   updateResult = featuresToUpdate.next();
 }

 featuresToUpdateMap.clear();
 ...
```

Tracking cars - implementing LayerView

Implementing `attach()` method

- The attach method provide the ability to do a one off initialization of WebGL resources. this includes (but not restricted to):
  - Create shader programs
  - Set event handlers (`onFeature()`, `doUpdate()` etc.)
  - Initialize WebGL extensions
  - Initialize WebGL buffers and vertex array objects
  - load images and textures
  - Initialize framebuffers and other WebGL resources needed to render the layer's data

Tracking cars - implementing LayerView

Implementing *detach()* method

- As the name implies, the detach method is the opposite of the attach method. It provides means to safely destruct of WebGL resources created during the calls to *attach()* and *render()*.

```
detach(): void {
  if (this._updateTimer !== 0) {
    clearInterval(this._updateTimer);
  }

  const gl = this.context;

  if (this._vao) {
    this._vaoExt.deleteVertexArray(this._vao);
    this._vao = null;
    this._vaoExt = null;

    gl.deleteBuffer(this._vertexBuffer);
    this._vertexBuffer = null
  }

  if (this._wglProgram) {
    gl.deleteProgram(this._wglProgram);
    this._wglProgram = null;
  }

  if (this._carTexture) {
    gl.deleteTexture(this._carTexture);
    this._carTexture = null;
  }
}
```

Tracking cars - implementing LayerView

Local origin

- Use the *view.viewpoint* as the origin
- Encoded positions, use the vector between the position and the viewpoint, should have enough precision

Clip features which are outside the visible extent

```
const { x, y } = viewpoint.targetGeometry as Point;
// update the local origin
const localX = this._localOrigin.x = x;
const localY = this._localOrigin.y = y;

const clipExtent = extent.clone();
clipExtent.expand(1.15); // expand by 15 percent

const allFeatures = features.values();
let result = allFeatures.next();
while (!result.done) {
  feature = result.value;
  result = allFeatures.next();
  const { x, y } = feature.geometry;
  // clip features which fall outside the clip extent
  if (x < clipExtent.xmin || x > clipExtent.xmax || y < clipExtent.ymin || y > clipExtent.ymax) {
    continue;
  }
...
```

Tracking cars - implementing LayerView

Texture atlas with all car symbols, together with the metrics

```
{
  car1: {
    xmin: 23,
    ymin: 14,
    xmax: 79,
    ymax: 107
  },
  car2: {
    xmin: 85,
    ymin: 14,
    xmax: 145,
    ymax: 107
  }...
```

```
const carsInfo = [];
    const carsJson = JSON.parse(carsJSON)
    for (const type of Object.keys(carsJson)) {
      const carInfo = carsJson[type];
      const info = i8888to32(carInfo.xmin, carInfo.ymin, carInfo.xmax - carInfo.xmin, carInfo.ymax - carInfo.ymin);
      carsInfo.push(info)
    }
```

- Single texture (less state changes)
- Use metrics to designate the relevant texture coordinate for each feature
- pre-pack the metrics into a single 32bit integer

Tracking cars - implementing LayerView

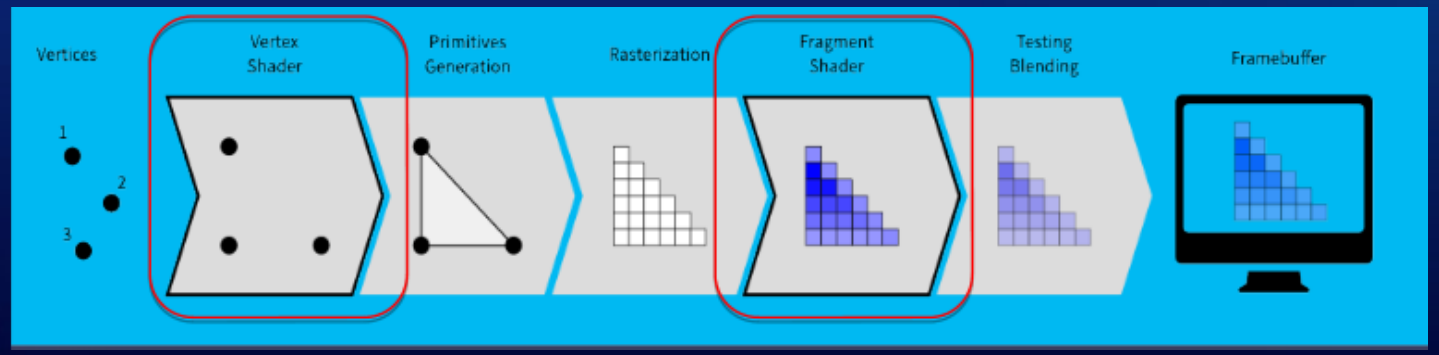
Pack per-feature attributes into each vertex (we use 6 vertices per marker)

```
// packed metrics per car type
const carsInfo = this._carsInfo;
// allocate the memory to encode the cars vertex buffer
const vertexBufferLength = ATTRS_PER_VERTEX * VERTS_PER_MARKER * features.size;
const vertexData = new Float32Array(vertexBufferLength);
const vertexDataU32 = new Uint32Array(vertexData.buffer);

...
const attributes = feature.attributes;
// position is the vector between the feature position and the view's viewpoint
dx = x - localX;
dy = y - localY;
// we normalize the heading into an angle between 0 and 255 dgrees (full circle)
heading = C_RAD_TO_256 * attributes["HEADING"];
// texture coordinates are pre-packes
texInfo = carsInfo[attributes["TYPE"]];

// encode a single vertex
vertexData[i * ATTRS_PER_VERTEX * VERTS_PER_MARKER + 0]    = dx;
vertexData[i * ATTRS_PER_VERTEX * VERTS_PER_MARKER + 1]    = dy;
vertexDataU32[i * ATTRS_PER_VERTEX * VERTS_PER_MARKER + 2] = texInfo;
// the first two components are the extrude offset
vertexDataU32[i * ATTRS_PER_VERTEX * VERTS_PER_MARKER + 3] = i8888to32(0, 0, heading, 0);
```

Tracking cars - implementing shaders

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, convert positions to normalized device coordinates (NDC)
  - ***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



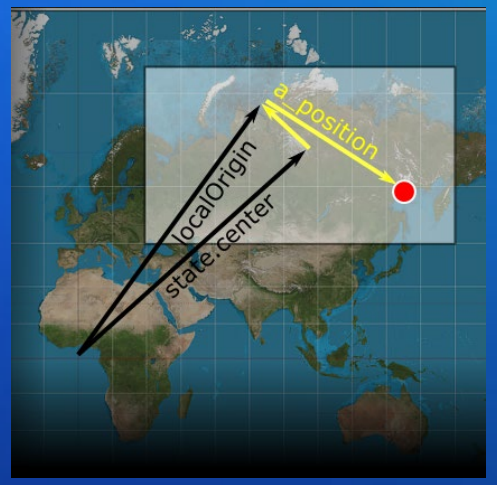
Tracking cars - implementing shaders

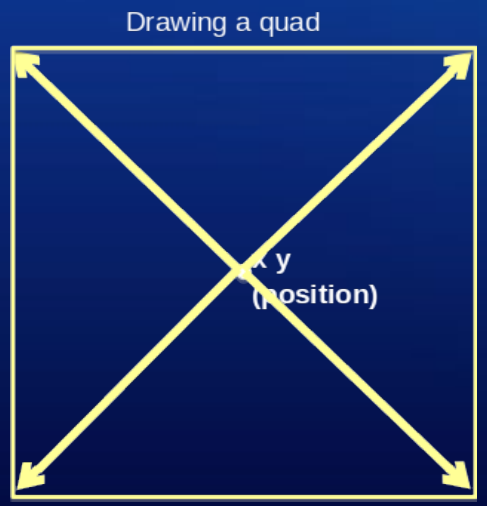
vertex shader

- position (given in pseudo map coordinates)
  - vertex position

    Use display-model-view matrix to transform from relative map coordinates to NDC

  - extrude
    - Move each vertex from the position of the feature to the right corner, counting for the size of the car image in pixels

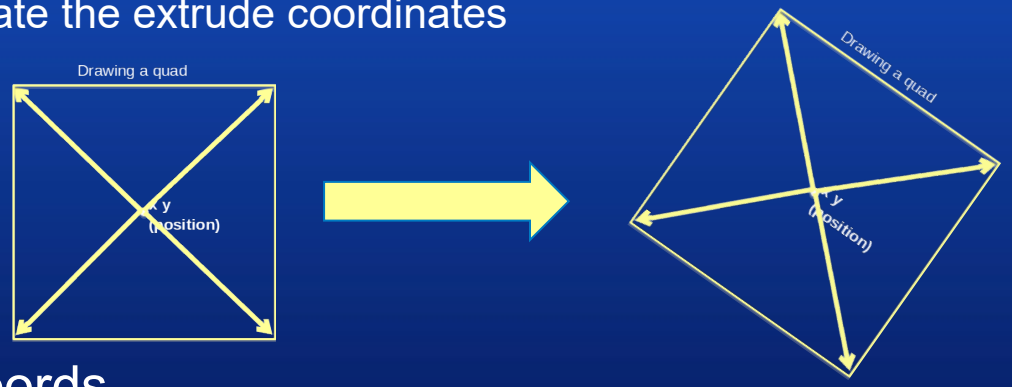




Tracking cars - implementing shaders

vertex shader

- Car heading angle
  - convert from 256 (since we wanted to fit in into a single byte as an attribute) rotation angle to rads
  - compute rotation matrix
  - rotate the extrude coordinates




```
mat3 getRotationMat(float rotationValue) {
   float angle = rotationValue;
   float sinA = sin(angle);
   float cosA = cos(angle);

   return mat3( cosA, -sinA, 0.0,
                sinA,  cosA, 0.0,
                 0.0,   0.0, 1.0);
    }
}
```

- Tex coords
  - we encoded absolute image coordinates (fit in into a single byte as an attribute)
  - Divide by the image size, given as a uniform in order to get normalized tex coords in the range of [0..1]
  - The texture coords will get interpolated at the fragment shader given as a varying

```
v_texCoord = (a_texInfo.xy + a_offestHeading.xy * imageSize) / u_texSize;
```

Tracking cars - implementing shaders

vertex shader

Putting it all together

```
// we encode the width and height of each car image into the texture info
mediump vec2 imageSize = a_texInfo.zw;
// convert angle to radians
mediump float heading = C_256_TO_RAD * a_offestHeading.z;
// get the offset and convertit to the range of [-0.5..0.5]
mediump vec2 offset = a_offestHeading.xy - vec2(0.5);

// compute the vertex position, NDC then extrude
mediump vec3 pos = u_dvsMat3 * vec3(a_position, 1.0) +
  u_displayViewMat3 * getRotationMat(heading) * vec3(u_iconRatio * imageSize * offset, 0.0);
// update the built-in variable gl_Posision
gl_Position = vec4(pos.xy, 0.0, 1.0);
// setthe texture coordinate varying
v_texCoord = (a_texInfo.xy + a_offestHeading.xy * imageSize) / u_texSize;
```

Tracking cars - implementing shaders

fragment shader

This is the easy code :-), all that the fragment shader code has to do is sample the texture at the given texture coordinates

- The code relies on the textCoord varying, which gets interpolated at the time it gets to the fragment shader stage. It samples the texture-atlas at the interpolated coord and sets the fragment color

```
uniform sampler2D u_texture;

varying vec2 v_texCoord;

void main() {
  gl_FragColor = texture2D(u_texture, v_texCoord);
}`;
```

Tracking cars - implementing LayerView

Implementing `render()` method

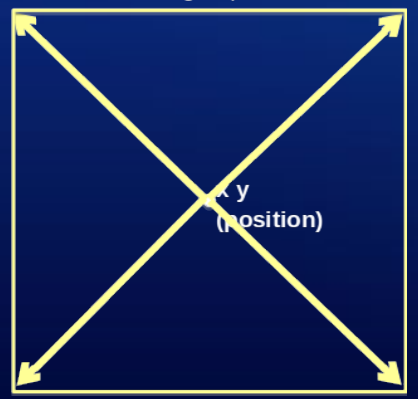
This is where the actual drawing with WebGL happens

```
gl.bindBuffer(gl.ARRAY_BUFFER, this._vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, this._bufferData, gl.STREAM_DRAW);
gl.bindBuffer(gl.ARRAY_BUFFER, null);
```

- Start with updating the cars buffer
- Update the transformation matrices
  1. display-view-model transformation. Transform from world to NDC
     - We consider the offset from the designated local origin
  2. Extrude matrix, move each vertex relative to the car's position
     - We need to consider the map's rotation in order to keep the right heading

```
const centerX = viewPointGeometry.x - this._localOrigin.x;
const centerY = viewPointGeometry.y - this._localOrigin.y;
const widthInMapUnits = resolution * size[0];
const heightInMapUnits = resolution * size[1];

const viewMat3 = mat3.identity(this._dvsMat3);
// compute the DMV transformation using the local origin
mat3.multiply(viewMat3, displayViewMat3, viewMat3);
mat3.translate(viewMat3, viewMat3, vec2.fromValues(size[0] / 2, size[1] / 2));
mat3.scale(viewMat3, viewMat3, vec2.fromValues(size[0] / widthInMapUnits, -size[1] / heightInMapUnits));
mat3.rotate(viewMat3, viewMat3, -rads);
mat3.translate(viewMat3, viewMat3, vec2.fromValues(-centerX, -centerY));

// we want the cars to be map aligned, so we need to deal with the map's rotation
mat3.translate(displayViewMat3, displayViewMat3, vec2.fromValues(size[0] / 2, size[1] / 2));
mat3.rotate(displayViewMat3, displayViewMat3, rads);
mat3.translate(displayViewMat3, displayViewMat3, vec2.fromValues(-size[0] / 2, -size[1] / 2));
```

extrude vertices



Tracking cars - implementing LayerView

Implementing `render()` method (cont.)

- Compute the scale of the markers

  We want the markers to be a bit smaller when we zoom out, and bigger when we zoom in

  - Compute a ratio given the current scale. We chose a logarithmic scale, in order to make it subtle
  - Pass the ration to the shader in order to scale the size of the markers

```
const scale = state.scale;
const iconRatio = 10.0 / Math.log2(scale);
...

gl.uniform1f(this._iconRatioLocation, iconRatio);
```

Tracking cars - implementing LayerView

Implementing `render()` method (cont.)

All that is left to do is bind the car's texture, buffer and shader. Set uniforms and draw!

```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, this._carTexture);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

// bind the shader
gl.useProgram(this._wglProgram);

// bind the cars vao
this._vaoExt.bindVertexArray(this._vao);

// set the uniforms
gl.uniformMatrix3fv(this._dvsMatrixLocation, false, this._dvsMat3);
gl.uniformMatrix3fv(this._displayViewMatrixLocation, false, this._displayViewMat3);
gl.uniform1i(this._carTextureLocation, 0);
gl.uniform2fv(this._carTexSizeLocation, this._carTexSize);
gl.uniform1f(this._iconRatioLocation, iconRatio);

// draw the cars
gl.drawArrays(
  gl.TRIANGLES,
  0,
  this._vertexBufferLength / ATTRS_PER_VERTEX
);
```

Tracking cars - implementing hittesting

Hittesting is the process of identifying the features which intersect with a given screen point, usually the result of a user-interaction (mouse-click, mouseover etc.)

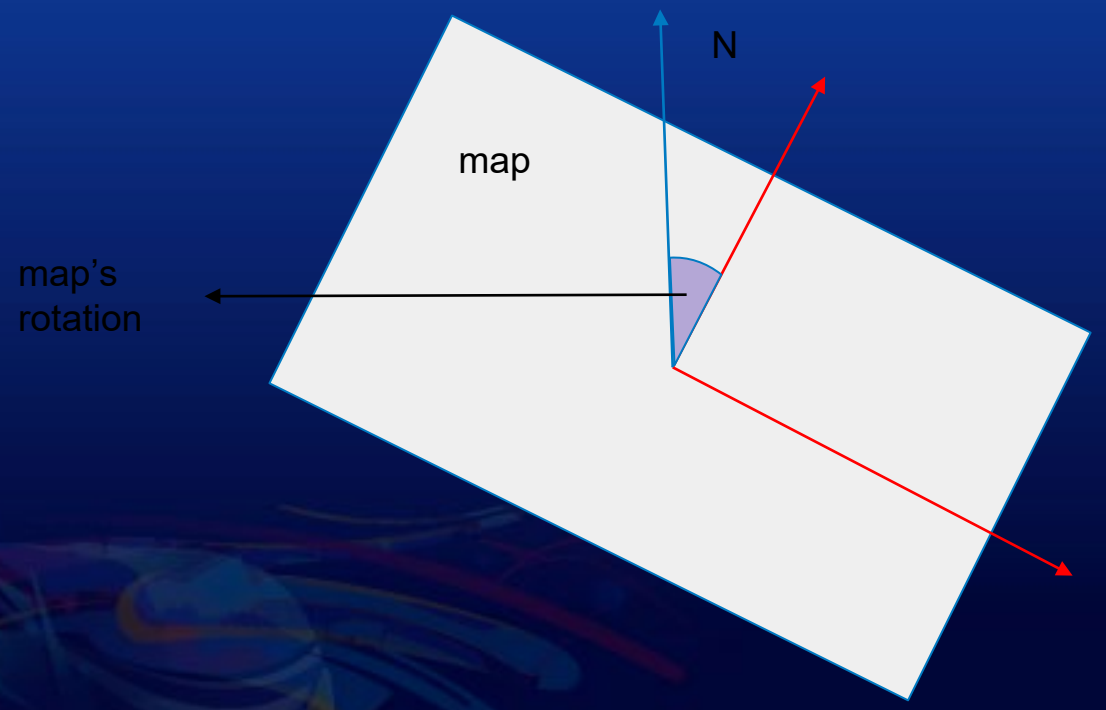
- There are several ways to implement hittest. Two examples are:
  - As a draw operation using the feature Ids as color, then a readback
    - very accurate!
    - Requires a dedicated shader variation
    - Add a dedicated draw-cycle
    - Readback operation (reading data from WebGL to system memory) is slow
  - Iterate over the features and geometrically test for the features which intersect with the hittest point
    - A rather fast operation
    - Need to consider the symbol of the feature, offsets, angle, map rotation etc.
    - Depending on the internal data-structures used to manage the features

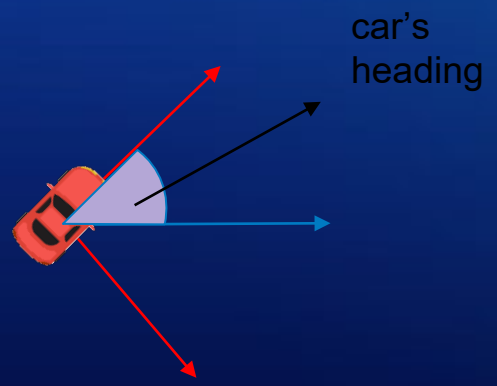
In this demo we have chosen to implement a geometrical computation over a draw-cycle based hittesing

Tracking cars - implementing hittesting

In the current walkthrough we have chosen to compute the hittest in screen coordinates.

- Car symbol metrics are given in pixels
- hittest point is in screen coordinates
- In theory, we can return the first car which contains the hittest point, but we look for the car which is closest to the point
- We need to compensate for the map's rotation and the car's heading





Tracking cars - implementing hittesting

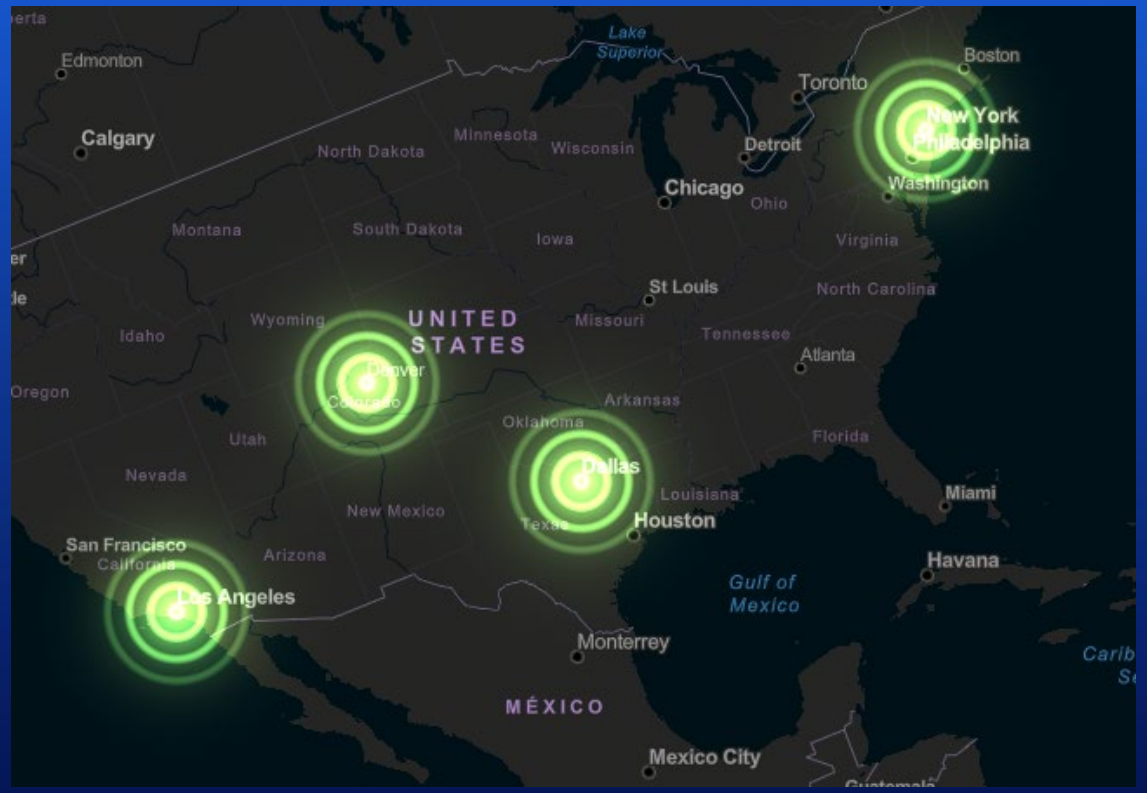
Putting it all together:

```
function testVectorInsideCar(W: number, H: number, rotation: number, dx: number, dy: number): boolean {
  const c = Math.cos(rotation);
  const s = Math.sin(rotation);

  const alpha = 2 * (c * dx + dy * s) / W;
  const beta = 2 * (-s * dx + dy * c) / H;

  return alpha >= -1 && alpha <= +1 && beta >= -1 && beta <= +1;
}
```

```
hitTest(x: number, y: number): Promise<Graphic> {
    let minDistance = Infinity;
    let foundProperties: any;
    const scale = this.view.scale;
    const iconRatio = 10.0 / Math.log2(scale);
    const spatialReference = this.view.spatialReference;

    for (const feature of Array.from(this._features.values())) {
      const { x: xMap, y: yMap } = feature.geometry;
      const { x: xScreen, y: yScreen } = this.view.toScreen(new Point({ x: xMap, y: yMap, spatialReference }));

      const metrics = this._carsMetrics[feature.attributes["TYPE"]];
      const W = metrics.width * iconRatio;
      const H = metrics.height * iconRatio;

      // Early rejection by checking the distance
      // against a circumscribed circle that encloses
      // the car.
      const R = Math.max(W, H);
      const dx = x - xScreen;
      const dy = y - yScreen;
      if (dx * dx + dy * dy > R * R) {
        continue;
      }

      const isInside = testVectorInsideCar(
        W, H,
        Math.PI * this.view.rotation / 180 - feature.attributes["HEADING"],
        dx, dy
      );
      const distanceFromCenter = Math.sqrt(dx * dx + dy * dy);
      if (isInside && distanceFromCenter < minDistance) {
        minDistance = distanceFromCenter;
        foundProperties = {
          attributes: feature.attributes
        };
      }
    }

    if (foundProperties) {
      const g = new Graphic(foundProperties);
      (g as any).sourceLayer = this.layer;
      return promiseUtils.resolve(g);
    }

    return promiseUtils.resolve();
  }
}
```

Using 3rd party WebGL libraries

- Writing a WebGL renderer from scratch is a lot of work
  - WebGL is verbose
  - Error-prone (shared global state!)
  - Debug tools are still somewhat flaky
- Using an existing WebGL helper library or engine can increase productivity
- Helper libraries are relatively thin abstractions on top of WebGL
  - luma.gl
  - regl
  - TWGL
- Engine are full-fledged solutions which introduce higher level concepts
  - PixiJS
  - deck.gl
- The choice of a library depends on
  - Functional requirements of the project
  - Non functional requirements
  - Team expertise
  - Licensing



Using a WebGL helper library

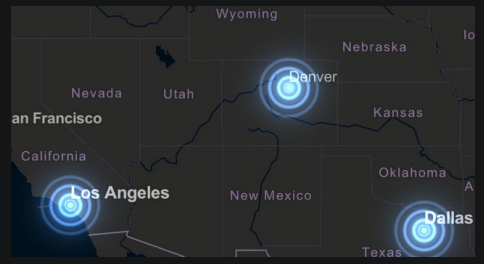
SDK sample

Using luma.gl
Using regl
Using twgl

When to use a WebGL helper library

- To use a WebGL helper library the developer still needs some understanding of WebGL
  - Lower-level than engines → easier to integrate in existing projects
- Some of what they offer include:
  - Simplified resource creation
  - Simplified/safer WebGL state management
  - Automatic opt-in into and fall-back from advanced WebGL functionality and extension
    - WebGL 2
    - Uniform buffers
    - Vertex Array Objects
- You still need to write quite a lot of code
  - Most notably, you usually have to supply your own shaders

A look at three different, 2D-focused, general purpose, WebGL toolkits

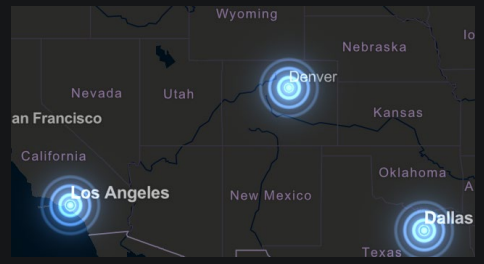


Raw WebGL (original SDK sample)

https://developers.arcgis.com/javascript/latest/sample-code/custom-gl-visuals/

We rewrote this sample using three different WebGL helper libraries/toolkits

A look at three different, 2D-focused, general purpose, WebGL toolkits

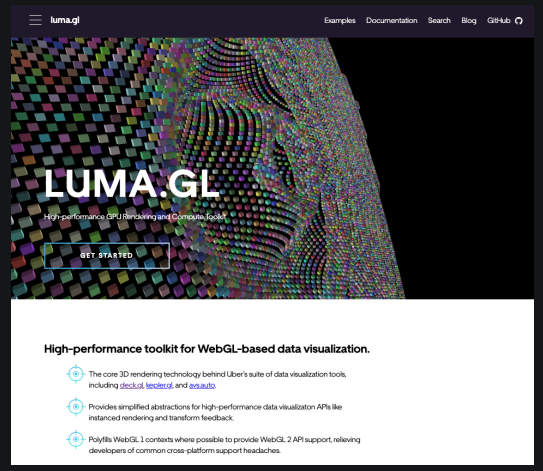


Raw WebGL (original SDK sample)

https://developers.arcgis.com/javascript/latest/sample-code/custom-gl-visuals/

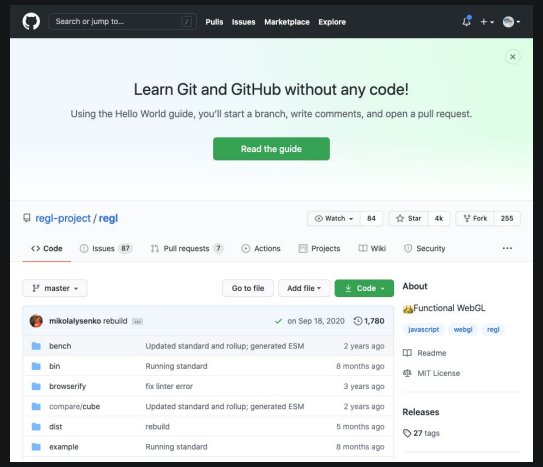
luma.gl

https://luma.gl/



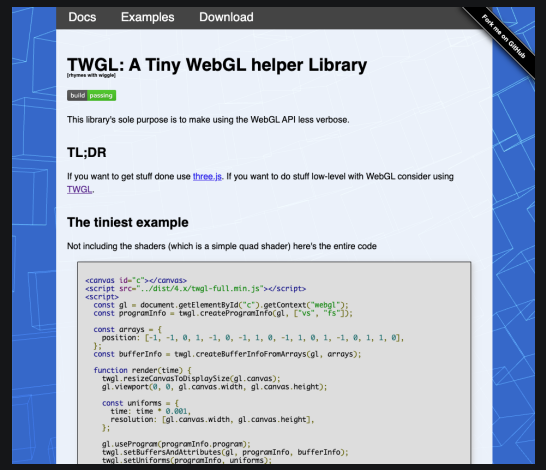
regl

https://github.com/regl-project/regl

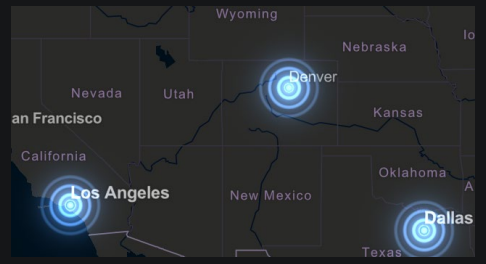


TWGL

https://twgljs.org/

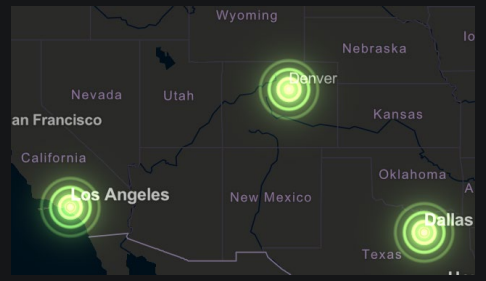


A look at three different, 2D-focused, general purpose, WebGL toolkits



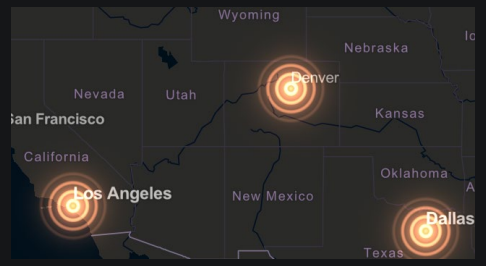
Raw WebGL (original SDK sample)

https://developers.arcgis.com/javascript/latest/sample-code/custom-gl-visuals/



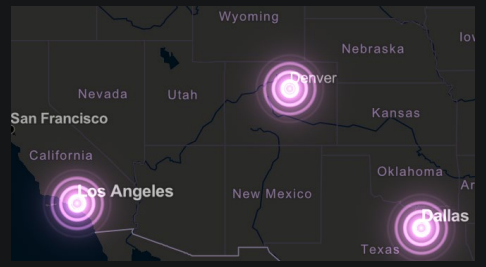
Done with luma.gl

https://codepen.io/dawken/pen/RwGOwpz



Done with regl

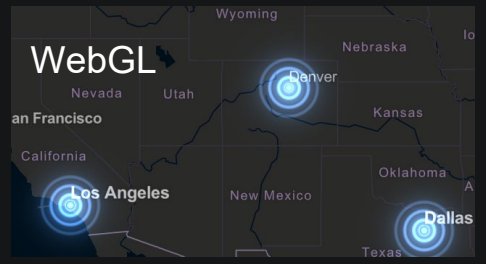
https://codepen.io/dawken/pen/jOMRPdP

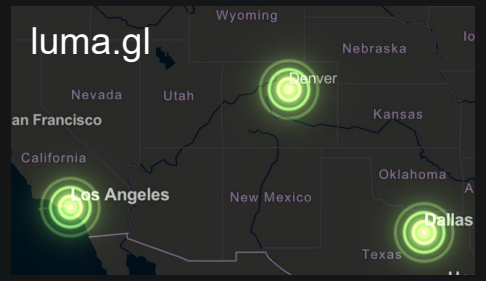


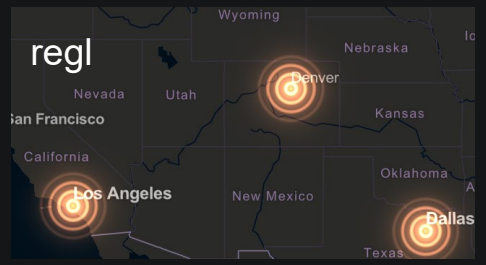
Done with TWGL

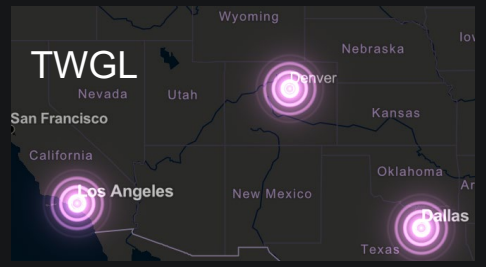
https://codepen.io/dawken/pen/poEBPdo

Importing the libraries









😎

```
<script src="https://cdn.jsdelivr.net/npm/@luma.gl/core@8.3.3/dist/dist.js"></script>
```

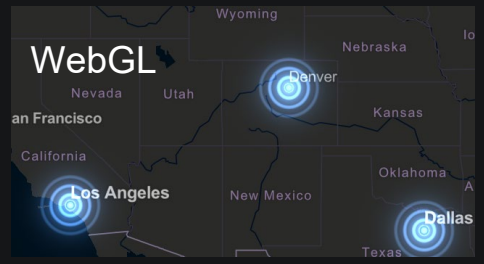
+

```
luma.instrumentGLContext(gl);
```

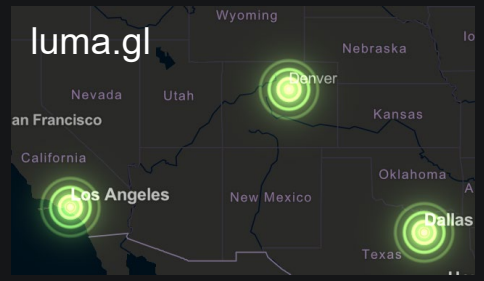
```
<script src="https://npmcdn.com/regl@2.0.1/dist/regl.js"></script>
```

```
<script src="https://cdn.jsdelivr.net/npm/twgl.js@4.18.0/dist/4.x/twgl-full.min.js"></script>
```

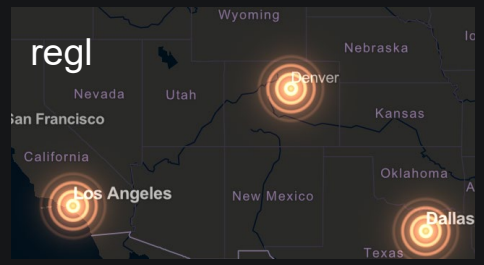
Resource creation - Shaders



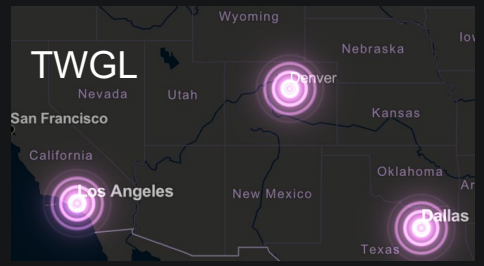

```
const vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexSource);
gl.compileShader(vertexShader);
const fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentSource);
gl.compileShader(fragmentShader);
...
```




```
const model = new luma.Model(gl, {
  vs: "attribute vec2 a_position...",
  fs: "precision mediump float...",
  ...
});
```

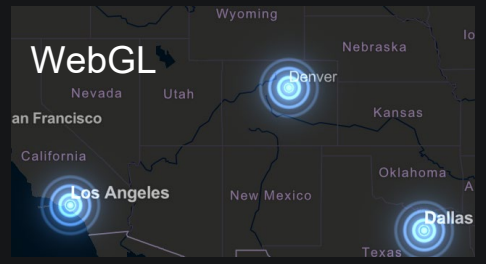



```
const preparedCommand = regl({
  ...
  vert: "attribute vec2 a_position...",
  frag: "precision mediump float..."
  ...
});
```

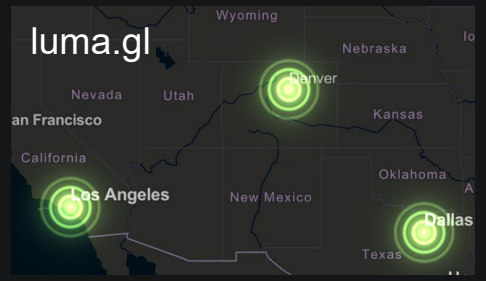



```
<script id="vs" type="x-shader/x-vertex">
  attribute vec2 a_position;
  ...
</script>
<script id="fs" type="x-shader/x-fragment">
  precision mediump float;
  ...
</script>
...
programInfo = twgl.createProgramInfo(gl, ["vs", "fs"]);
```

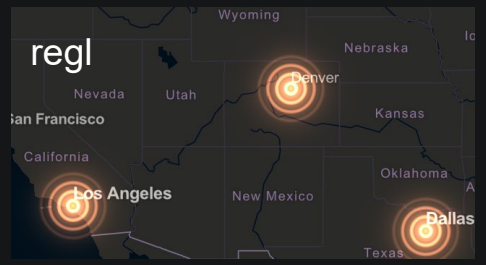
Resource creation - Buffers



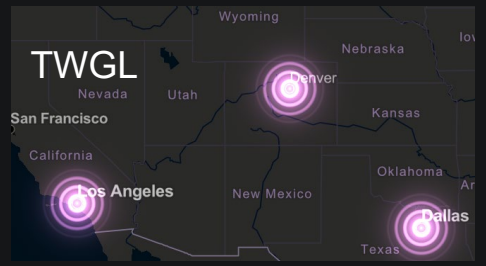

```webgl
const vertexBuffer = gl.createBuffer();
const indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertexData, gl.STATIC_DRAW);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indexData, gl.STATIC_DRAW);
```




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

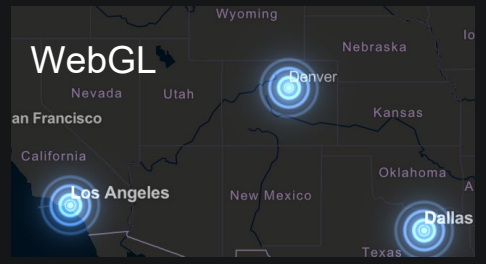



```regl
const preparedCommand = regl({
  ...
  attributes: {
    a_position: positionData,
    a_offset: offsetData
  },
  elements: this.regl.elements(indexData),
  count: 6 * graphics.length
});
```

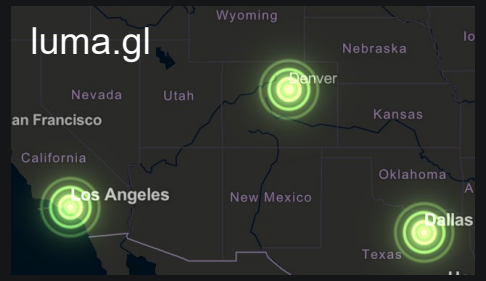



```twgl
const bufferInfo = twgl.createBufferInfoFromArrays(gl, {
  a_position: { numComponents: 2, data: positionData },
  a_offset: { numComponents: 2, data: offsetData },
  indices
});
```

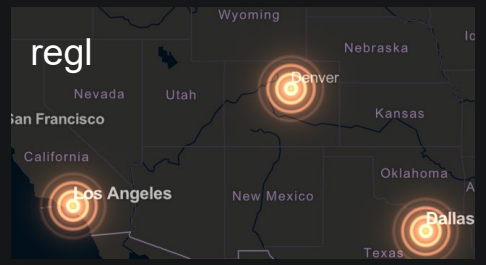
Setting uniforms



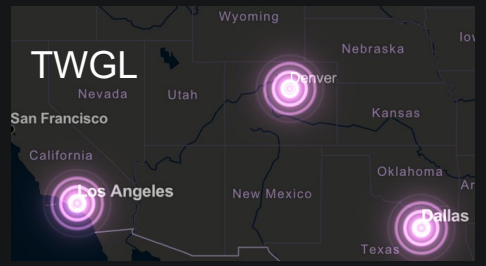

```
gl.useProgram(program);
gl.uniformMatrix3fv(uTransform, false, transform);
gl.uniformMatrix3fv(uDisplay, false, display);
gl.uniform1f(uCurrentTime, performance.now() / 1000.0);
```




```
luma.withParameters(gl, {
  blend: true, blendFunc: [gl.ONE, gl.ONE]
}, function () {
  model.setUniforms({
    u_transform: transform,
    u_display: display,
    u_current_time: performance.now() / 1000.0
  });
  model.draw();
});
```

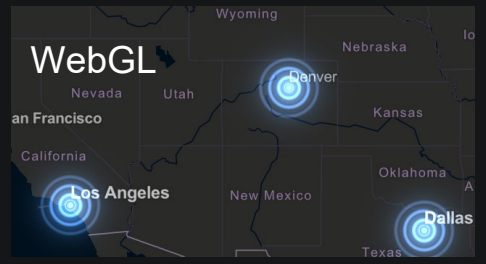



```
const preparedCommand = regl({
  ...
  uniforms: {
    u_this_is_a_static_property: 42,
    u_this_is_a_dynamic_property: regl.prop("currentTime")
    ...
  },
  ...
});
```

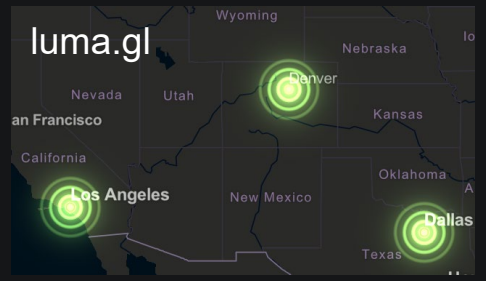



```
twgl.setUniforms(this.programInfo, {
  u_transform: this.transform,
  u_display: this.display,
  u_current_time: performance.now() / 1000.0
});
```

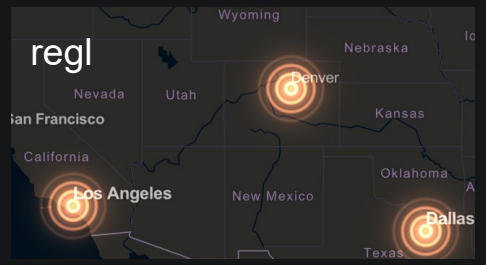
Setting fixed-function state



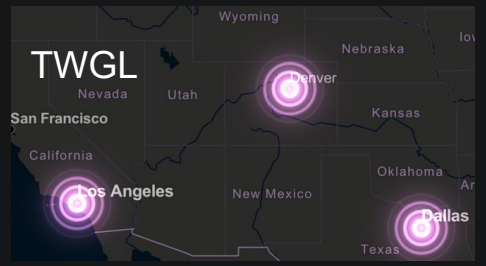

```webgl
gl.enable(gl.BLEND);
gl.blendFunc(gl.ONE, gl.ONE);
```




```luma.gl
luma.withParameters(gl, {
  blend: true, blendFunc: [gl.ONE, gl.ONE]
}, function () {
  model.setUniforms({
    u_transform: transform,
    u_display: display,
    u_current_time: performance.now() / 1000.0
  });
  model.draw();
});
```

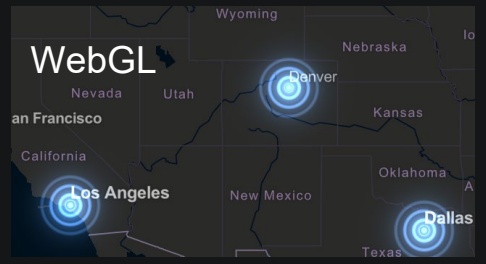



```regl
const preparedCommand = regl({
  ...
  depth: { enable: false },
  blend: { enable: true, func: { src: "one", dst: "one" } },
  ...
});
```

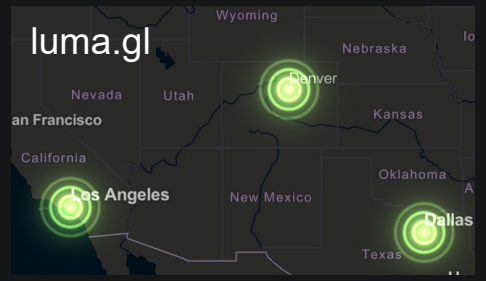



```twgl
// You use plain WebGL calls
gl.enable(gl.BLEND);
gl.blendFunc(gl.ONE, gl.ONE);
```

Draw

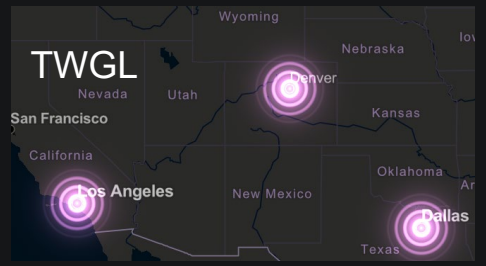



```
gl.drawElements(gl.TRIANGLES, this.indexBufferSize, gl.UNSIGNED_SHORT, 0);
```




```
luma.withParameters(gl, {
  blend: true, blendFunc: [gl.ONE, gl.ONE]
}, function () {
  model.setUniforms({
    u_transform: transform,
    u_display: display,
    u_current_time: performance.now() / 1000.0
  });
  model.draw();
});
```

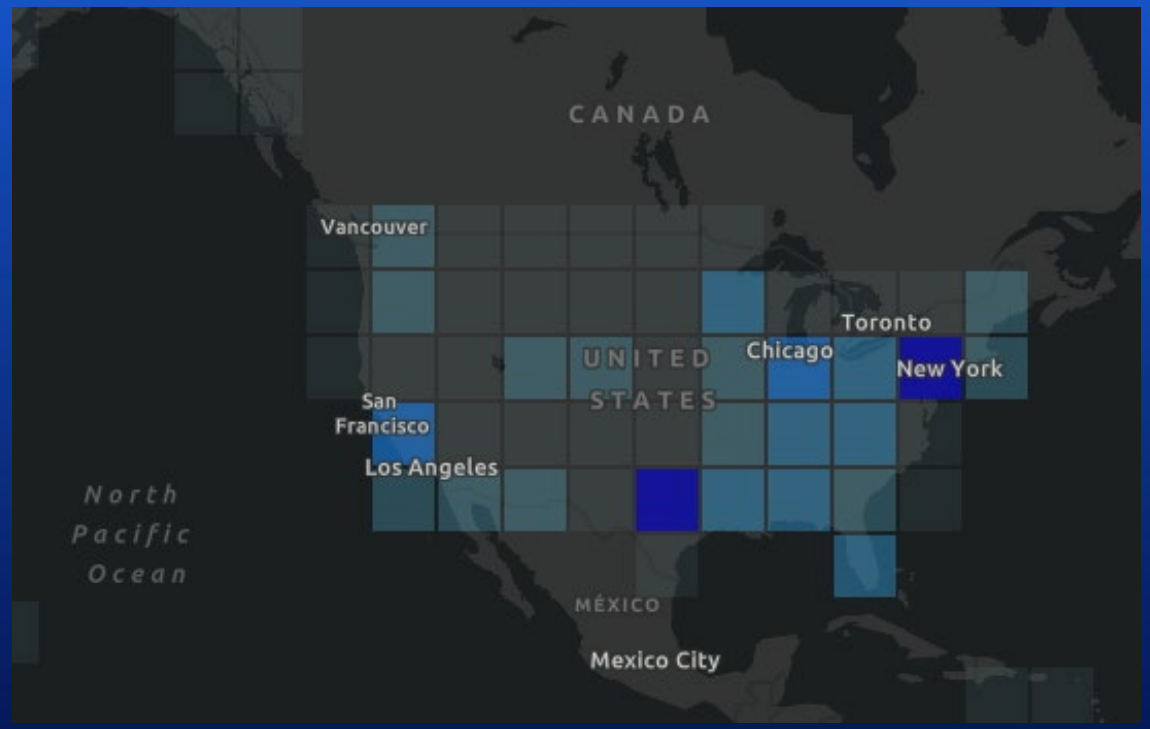

```
preparedCommand({
  transform,
  display,
  currentTime
});
```




```
twgl.drawBufferInfo(gl, this.bufferInfo);
```

When to use a WebGL engine

- WebGL engines such as PixiJS or deck.gl completely abstract away the WebGL API
  - **More “opinionated” than helpers; sometimes they don’t play nice with existing code**
- Some of what they offer include:
  - A concept of object and hierarchical grouping of objects
  - Some sort of camera model
  - Simplified resource loading and some form of garbage collection
  - Render optimizers and batchers to save draw calls
  - An automated render loop
- You can get the same result with much less code than using WebGL or a helper library
  - When used stand-alone, you can often forget that WebGL even exists
  - ***When integrating with other engines and existing code, careful planning is needed***



Using deck.gl and ArcGIS together

CodePen

Loading @deck.gl/arcgis
Loading the projection engine
Querying an ArcGIS feature layer
Adapt the returned feature set to deck.gl
Add a deck.gl layer on top of an ArcGIS basemap

Importing deck.gl and the ArcGIS plugin

- deck.gl is a *“WebGL powered visualization framework for large-scale datasets”*
- Esri has worked with the vis.gl foundation to publish the `@deck.gl/arcgis` NPM package
  - It enables easy integration of deck.gl with the ArcGIS API for JavaScript
  - Internally it uses a custom layer view

```
<script src="https://unpkg.com/deck.gl@8.3.0/dist.min.js"></script>
<script src="https://unpkg.com/@deck.gl/arcgis@8.3.0/dist.min.js"></script>
<link rel="stylesheet" href="https://js.arcgis.com/4.19/esri/themes/dark-blue/main.css">
<script src="https://js.arcgis.com/4.19/"></script>
```

Require the ArcGIS modules

- We are going to use:
  - `FeatureLayer` to query the data
  - The projection engine to adapt the data to deck.gl

```
require([
  "esri/Map",
  "esri/views/MapView",
  "esri/layers/FeatureLayer",
  "esri/geometry/projection"
], function (
  Map,
  MapView,
  FeatureLayer,
  projection
) {
  ...
```

Query the features using `FeatureLayer`

- We will not be adding the layer to the map; we only use it to query the service
  - Querying returns a promise

```
const featureLayer = new FeatureLayer({ url });
const query = featureLayer.createQuery();
query.where = "1=1";
const queryPromise = featureLayer.queryFeatures(query);
```

Load the `@deck.gl/arcgis` modules

- Call the `loadArcGISModules()` method on the global `deck` object
  - The call returns a promise

```
const deckglArcGISPromise = deck.loadArcGISModules();
```

Load the projection engine

- The projection engine exposes a `load()` method that must be called before its use
  - The call returns a promise

```
const projectionEnginePromise = projection.load();
```

Putting everything together

- We wait on the three promises; when they resolve, the callback will receive:
  - The queried features in the `result` parameter
  - The `@deck.gl/arcgis` classes in the `arcGIS` object

```
Promise.all([queryPromise, deckglArcGISPromise, projectionEnginePromise]).then(([result, arcGIS]) => {
  ...
});
```

Reprojecting the features

- The position of the features must be expressed in `[longitude, latitude]`
- We use the projection engine with WGS-84 as the target SR (wkid 4326)

```
Promise.all([queryPromise, deckglArcGISPromise, projectionEnginePromise]).then(([result, arcGIS]) => {
  const data = result.features
    .map((feature) => projection.project(feature.geometry, { wkid: 4326 }));

  ...
});
```

Importing deck.gl and the ArcGIS plugin

- The class `arcGIS.DeckLayer` is an ArcGIS layer that can host deck.gl layers
  - It acts as a pseudo-group layer for deck.gl layers
  - In this sample, we only add a single deck.gl layer called `deck.ScreenGridLayer`

```
Promise.all([queryPromise, deckglArcGISPromise, projectionEnginePromise]).then(([result, arcGIS]) => {
  ...

  const layer = new arcGIS.DeckLayer({
    "deck.layers": [
      new deck.ScreenGridLayer({
        id: "grid",
        data,
        opacity: 0.8,
        getPosition: d => [d.x, d.y],
        cellSizePixels: 32,
        colorRange: [
          [180, 255, 255, 20],
          [120, 220, 250, 80],
          [80, 180, 250, 120],
          [60, 140, 250, 170],
          [30, 60, 240, 210],
          [40, 0, 180, 255]
        ],
        aggregation: "SUM",
        gpuAggregation: false
      })
    ]
  });

  ...
});
```

Create the map and mapview as usual

- The created `arcGIS.DeckLayer` instance can be finally added to a map
- Check out the sample at https://codepen.io/dawken/pen/rNMEzOp for more details!

```
Promise.all([queryPromise, deckglArcGISPromise, projectionEnginePromise]).then(([result, arcGIS]) => {
  ...

  const mapView = new MapView({
    container: "viewDiv",
    map: new Map({
      basemap: "dark-gray-vector",
      layers: [layer]
    }),
    center: [-98, 39],
    zoom: 2
  });
});
```

Custom sprites and particle systems with PixiJS

CodePen

Working with PixiJS
Working with pixi-particles, a particle engine for PixiJS

Importing PixiJS and pixi-particles

- PixiJS is a high performance WebGL 2D engine
  - It is more high-level than luma.gl, regl and TWGL, and is optimized for 2D use cases
- pixi-particles is a plugin for PixiJS that adds support for particle systems
  - Check out the interactive particle editor:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/pixi.js/5.3.7/pixi.js"></script>
<script src="https://cdn.jsdelivr.net/npm/pixi-particles-latest@3.2.0/dist/pixi-particles.min.js"></script>
<link rel="stylesheet"href="https://js.arcgis.com/4.19/esri/themes/light/main.css">
<script src="https://js.arcgis.com/4.19/"></script>
```

- Remember what we said about engines being sometimes difficult to integrate?
- Integration with PixiJS is still experimental
  - This sample is very much a proof-of-concept
  - Exercise caution if you decide to use this technique in production
  - Check the ArcGIS SDK! Stay tuned for updates!

Create the needed PixiJS objects

- We create everything that we need for rendering with PIXIJS in the `attach()` method
- In the real world, the layer view needs to consult the layer to know what to render
  - *e.g.* by accessing `this.layer.graphics` or `this.layer.url`
    - And possibly listening for changes
  - This is just a super-simplified app that focuses on integrating with PixiJS

```
const CustomLayerView2D = BaseLayerViewGL2D.createSubclass({
  ...
  attach: function () {
    const gl = this.context;
    ...
    this.renderer = new PIXI.Renderer({ context: gl, view: gl.canvas });
    this.stage = new PIXI.Container();
    this.particles = new PIXI.particles.ParticleContainer();
    this.sprite = PIXI.Sprite.from(someImageURL);
    this.sprite.anchor.set(0.5);
    this.stage.addChild(this.sprite);
    this.stage.addChild(this.particles);
    this.emitter = new PIXI.particles.Emitter(
      this.particles,
      [anotherImageURL],
      {
        // See https://github.com/pixijs/pixi-particles#sample-usage for a
        // sample definition of a particle system or check out the interactive
        // editor at https://pixijs.io/pixi-particles-editor/#pixieDust
      }
    });
    this.emitter.emit = true;
  }
  ...
});
```

Set up an offscreen framebuffer and a way to composite on top of the map

- PixiJS does not support rendering to an external framebuffer
- We will render to a `PIXI.RenderTexture` and then overlay to screen using raw WebGL

```
this.renderTexture = PIXI.RenderTexture.create(64, 64);
```

PixiJS will render here

```
const vs = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vs, `attribute vec2 a_Position;
  varying vec2 v_TexCoord;
  void main(void) {
    gl_Position = vec4(a_Position, 0.0, 1.0);
    v_TexCoord = (a_Position + 1.0) / 2.0;
    v_TexCoord.y = 1.0 - v_TexCoord.y;
  }
`);
gl.compileShader(vs);
```

```
const fs = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fs, `precision mediump float;
varying vec2 v_TexCoord;
uniform sampler2D u_Texture;
void main(void) {
  gl_FragColor = texture2D(u_Texture, v_TexCoord);
}
`);
gl.compileShader(fs);
```

```
const quadProgram = gl.createProgram();
gl.attachShader(quadProgram, vs);
gl.attachShader(quadProgram, fs);
gl.bindAttribLocation(quadProgram, 0, "a_Position");
gl.linkProgram(quadProgram);
```

```
const quadBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, quadBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Int8Array([-1, -1, 1, -1, -1, 1, 1, 1]), gl.STATIC_DRAW);
gl.bindBuffer(gl.ARRAY_BUFFER, null);
this.quadBuffer = quadBuffer;
```

Then we will composite with the existing content, using a fullscreen quad and a simple texture mapping program

The `render()` method

- Having to render to an offscreen surface is common when integrating two engines
  - We also do it to integrate with deck.gl, but in that case we do it for you; more on this later on
  - For PixiJS, we need a little hack in order to feed the rendered image back to WebGL

```
const CustomLayerView2D = BaseLayerViewGL2D.createSubclass({
  ...

  render: function (renderParameters) {
    // 1. Render with PixiJS to the render texture
    // 2. Recover the WebGL texture from the PIXI render texture
    // 3. Using WebGL, map the texture to a full-screen quad
  },

  ...
});
```

1. Render with PixiJS

- Both the MapView and PixiJS modify the global WebGL state
- We wrap the PixiJS code in calls to `PIXI.Renderer.reset()`
  - This shields both the `MapView` and PixiJS from working with an unknown state

```
const CustomLayerView2D = BaseLayerViewGL2D.createSubclass({
  ...

  render: function (renderParameters) {
    this.renderer.reset();
    const w = renderParameters.state.size[0] * renderParameters.state.pixelRatio;
    const h = renderParameters.state.size[1] * renderParameters.state.pixelRatio;
    this.emitter.update(0.01667);
    const spritePosition = [0, 0];
    renderParameters.state.toScreen(spritePosition, -13429829, 4183452);
    this.sprite.position.set(spritePosition[0], spritePosition[1]);
    this.sprite.scale.set(1 + 0.3 * Math.cos(performance.now() / 100));
    const particlesPosition = [0, 0];
    renderParameters.state.toScreen(particlesPosition, -13430024, 4182709);
    this.particles.position.set(particlesPosition[0], particlesPosition[1]);
    this.renderer.resize(w, h);
    this.renderTexture.resize(w, h);
    this.renderer.render(this.stage, this.renderTexture);
    this.renderer.reset();

    ...
  },

  ...
});
```

Reset

Positioning and animating

Match view size

Render!

Reset

Offscreen surface

2. and 3. Recover the WebGL texture and map it to the screen

- Mapping a texture to the screen is quite straightforward
- Check out the sample at https://codepen.io/dawken/pen/bGBGKvg for more details!

Recover the texture

Bind the framebuffer

Full-screen rendering

Re-render because we are animating

```
const CustomLayerView2D = BaseLayerViewGL2D.createSubclass({
  ...

  render: function (renderParameters) {
    ...

    const texture = this.renderTexture.baseTexture._glTextures[this.renderer.texture.CONTEXT_UID].texture;
    this.bindRenderTarget();
    gl.useProgram(this.programs.quad.program);
    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.uniform1i(this.programs.quad.uniformLocations.u_Texture, 0);
    gl.bindBuffer(gl.ARRAY_BUFFER, this.quadBuffer);
    gl.vertexAttribPointer(0, 2, gl.BYTE, false, 2, 0);
    gl.enableVertexAttribArray(0);
    gl.bindBuffer(gl.ARRAY_BUFFER, null);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
    gl.useProgram(null);
    this.requestRender();
  },

  ...
});
```

This is where PixiJS should have rendered things in the first place

Thank you!

- **Tracking cars sample (TypeScript)**
  - Tracking cars app
    - Repo: https://github.com/yaronfine/devsummit-2021-demos
    - Demo: https://damix911-tracking-cars.s3-us-west-1.amazonaws.com/index.html
  - Node stream service (thanks to Matthew George for the original server!)
    - Repo: https://github.com/yaronfine/node-stream-service
    - Test instance: `wss://damix911-node-stream-service.herokuapp.com/`

- **Integration with 3rd party libraries (JavaScript)**
  - Collection on CodePen: https://codepen.io/collection/DKLkmp

esri | THE SCIENCE OF WHERE®

Please provide your feedback for this session by clicking on the session survey link directly below the video.