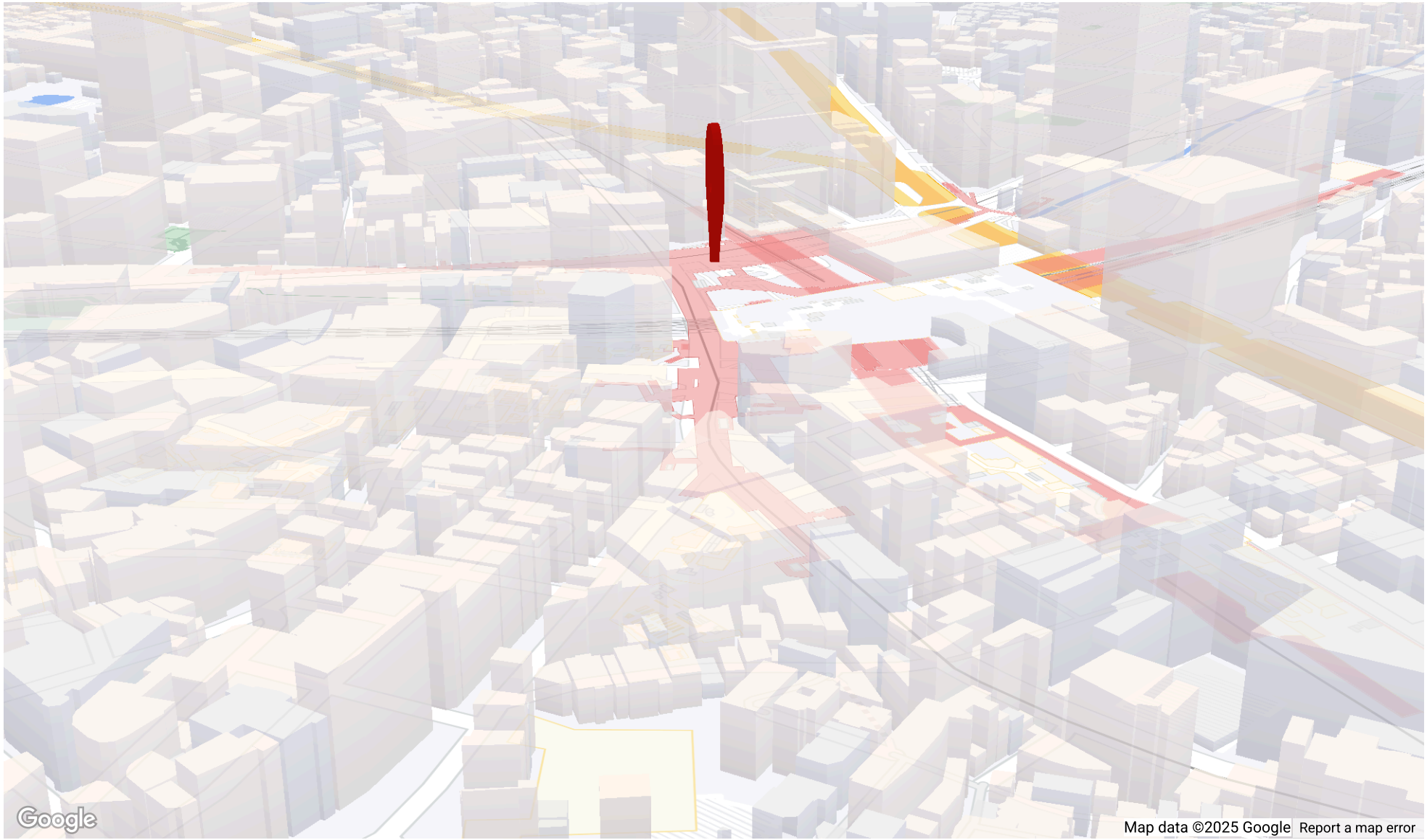
Announcement: New basemap styling is coming soon to Google Maps Platform. This update to map styling includes a new default color palette, modernized pins, and improvements to map experiences and usability. All map styles will be automatically updated in March 2025. For more information on availability and how to opt in earlier, see New map style for Google Maps Platform (/maps/new-map-style-opt-in).

WebGL Overlay View



View Sample (/maps/documentation/javascript/examples/webgl/webgl-overlay-simple)

With WebGL Overlay View you can add content to your maps using WebGL directly, or popular Graphics libraries like Three.js. WebGL Overlay View provides direct access to the same WebGL rendering context Google Maps Platform uses to render the vector basemap. This use of a shared rendering context provides benefits such as depth occlusion with 3D building geometry, and the ability to sync 2D/3D content with basemap rendering. Objects rendered with the WebGL Overlay View can also be tied to latitude/longitude coordinates, so they move when you drag, zoom, pan, or tilt the map.

Requirements

To use WebGL Overlay View, you must load the map using a map ID with the vector map enabled. We strongly recommend enabling tilt and rotation when you create the map ID, to allow for full 3D camera control. See the overview for details (/maps/documentation/javascript/webgl).

Add WebGL Overlay View

To add the overlay to your map, implement **`google.maps.WebGLOverlayView`**, then pass it your map instance using **`setMap`**:

```
// Create a map instance.
const map = new google.maps.Map(mapDiv, mapOptions);

// Create a WebGL Overlay View instance.
const webglOverlayView = new google.maps.WebGLOverlayView();

// Add the overlay to the map.
webglOverlayView.setMap(map);
```

Lifecycle hooks

WebGL Overlay View provides a set of hooks that are called at various times in the lifecycle of the WebGL rendering context of the vector basemap. These lifecycle hooks are where you setup, draw, and tear down anything you want rendered in the overlay.

- **`onAdd()`** is called when the overlay is created. Use it to fetch or create intermediate data structures before the overlay is drawn that don't require immediate access to the WebGL rendering context.
- **`onContextRestored({gl})`** is called once the rendering context is available. Use it to initialize or bind any WebGL state such as shaders, GL buffer objects, and so on. `onContextRestored()` takes a `WebGLStateOptions` instance, which has a single field:
  - `gl` is a handle to the `WebGLRenderingContext` used by the basemap.
- **`onDraw({gl, transformer})`** renders the scene on the basemap. The parameters for `onDraw()` is a `WebGLDrawOptions` object, which has two fields:
  - `gl` is a handle to the `WebGLRenderingContext` used by the basemap.
  - `transformer` provides helper functions to transform from map coordinates to model-view-projection matrix, which can be used to translate map coordinates to world space, camera space, and screen space.
- **`onContextLost()`** is called when the rendering context is lost for any reason, and is where you should clean up any pre-existing GL state, since it is no longer needed.
- **`onStateUpdate({gl})`** updates the GL state outside of the render loop, and is invoked when `requestStateUpdate` is called. It takes a `WebGLStateOptions` instance, which has a single field:
  - `gl` is a handle to the `WebGLRenderingContext` used by the basemap.
- **`onRemove()`** is called when the overlay is removed from the map with `WebGLOverlayView.setMap(null)`, and is where you should remove all intermediate objects.

For example, the following is a basic implementation of all lifecycle hooks:

```
const webglOverlayView = new google.maps.WebGLOverlayView();

webglOverlayView.onAdd = () => {
  // Do setup that does not require access to rendering context.
}

webglOverlayView.onContextRestored = ({gl}) => {
  // Do setup that requires access to rendering context before onDraw call.
}

webglOverlayView.onStateUpdate = ({gl}) => {
  // Do GL state setup or updates outside of the render loop.
}

webglOverlayView.onDraw = ({gl, transformer}) => {
  // Render objects.
}

webglOverlayView.onContextLost = () => {
  // Clean up pre-existing GL state.
}

webglOverlayView.onRemove = () => {
  // Remove all intermediate objects.
}

webglOverlayView.setMap(map);
```

Resetting GL State

WebGL Overlay View exposes the WebGL rendering context of the basemap. Because of this, it is extremely important that you reset the GL state to its original state when you are done rendering objects. Failure to reset the GL state is likely to result in GL state conflicts, which will cause rendering of both the map and any objects you specify to fail.

Resetting the GL state is normally handled in the `onDraw()` hook. For example, Three.js provides a helper function that clears any changes to the GL state:

```
webglOverlayView.onDraw = ({gl, transformer}) => {
  // Specify an object to render.
  renderer.render(scene, camera);
  renderer.resetState();
}
```

If the map or your objects fail to render, it is very likely that the GL state has not been reset.

Coordinate Transformations

The position of an object on the vector map is specified by providing a combination of latitude and longitude coordinates, as well as altitude. 3D graphics, however, are specified in world space, camera space, or screen space. To make it easier to transform map coordinates to these more commonly used spaces, WebGL Overlay View provides the `coordinateTransformer.fromLatLngAltitude(latLngAltitude, rotationArr, scalarArr)` helper function in the `onDraw()` hook that takes the following and returns a `Float64Array`:

- `latLngAltitude`: Latitude/longitude/altitude coordinates either as a `LatLngAltitude` or `LatLngAltitudeLiteral`.
- `rotationArr`: `Float32Array` of euler rotation angles specified in degrees.
- `scalarArr`: `Float32Array` of scalars to apply to the cardinal axis.

For example, the following uses `fromLatLngAltitude()` to create a camera projection matrix in Three.js:

```
const camera = new THREE.PerspectiveCamera();
const matrix = coordinateTransformer.fromLatLngAltitude({
    lat: mapOptions.center.lat,
    lng: mapOptions.center.lng,
    altitude: 120,
});
camera.projectionMatrix = new THREE.Matrix4().fromArray(matrix);
```

Example

The following is a simple example of using Three.js (https://threejs.org), a popular, open source WebGL library, to place a 3D object on the map. For a complete walkthrough of using WebGL Overlay View to build the example you see running at the top of this page, try the Building WebGL-accelerated Map Experiences codelab (http://goo.gle/maps-platform-webgl-codelab).

```
const webglOverlayView = new google.maps.WebGLOverlayView();
let scene, renderer, camera, loader;

webglOverlayView.onAdd = () => {
  // Set up the Three.js scene.
  scene = new THREE.Scene();
  camera = new THREE.PerspectiveCamera();
  const ambientLight = new THREE.AmbientLight( 0xffffff, 0.75 ); // Soft white light.
  scene.add(ambientLight);
```

```
  // Load the 3D model with GLTF Loader from Three.js.
  loader = new GLTFLoader();
  loader.load("pin.gltf");
}

webglOverlayView.onContextRestored = ({gl}) => {
  // Create the Three.js renderer, using the
  // maps's WebGL rendering context.
  renderer = new THREE.WebGLRenderer({
    canvas: gl.canvas,
    context: gl,
    ...gl.getContextAttributes(),
  });
  renderer.autoClear = false;
}

webglOverlayView.onDraw = ({gl, transformer}) => {
  // Update camera matrix to ensure the model is georeferenced correctly on the map.
  const matrix = transformer.fromLatLngAltitude({
      lat: mapOptions.center.lat,
      lng: mapOptions.center.lng,
      altitude: 120,
  });
camera.projectionMatrix = new THREE.Matrix4().fromArray(matrix);

  // Request a redraw and render the scene.
  webglOverlayView.requestRedraw();
  renderer.render(scene, camera);

  // Always reset the GL state.
  renderer.resetState();
}
```

```
// Add the overlay to the map.
webglOverlayView.setMap(map);
```


Last updated 2025-01-29 UTC.