Sample Code

Displaying overlays on a map

Add regions of layered content to a map view.

Download

iOS 16.1+ | iPadOS 16.1+ | Xcode 16.0+

Overview

Overlays offer a way to layer content over regions of a map and have that content scale with the map. This sample code project demonstrates how to draw common shapes, how to perform custom drawing outside of a common shape, and how to use map tiles from additional data sources.

Define overlays with geographic coordinates

Overlays are data objects that represent geographic information. Most overlays use geographic coordinates to create contiguous or noncontiguous sets of lines, rectangles, circles, and other shapes. For example, this sample app defines a rectangular area enclosing San Francisco as an array of `CLLocationCoordinate2D` coordinates.

```
/// A rectangular area containing San Francisco.
static let sanFranciscoRectangle = [
    CLLocationCoordinate2D(latitude: 37.816_41, longitude: -122.522_62),
    CLLocationCoordinate2D(latitude: 37.816_41, longitude: -122.355_54),
    CLLocationCoordinate2D(latitude: 37.702_08, longitude: -122.355_54),
    CLLocationCoordinate2D(latitude: 37.702_08, longitude: -122.522_62)
]
```

The app creates the overlay objects by providing the coordinate data to an object that conforms to the MKOverlay protocol. This data object is responsible for managing the data that defines the overlay. MapKit defines several concrete overlay objects for specifying different types of standard shapes, such as circles and polygons. The app uses the coordinate array above to create one of these provided overlay objects — a polygon.

```
/// Creates a rectangle polygon.
var rectangleOverlay: MKPolygon {
    return MKPolygon(coordinates: LocationData.sanFranciscoRectangle, count: Locatic
}
```

Because MapKit defines overlays using a protocol, any class in an app can be an overlay object by conforming to the `MKOverlay` protocol, or by subclassing MKShape or MKMultiPoint. For example, `PeakGroundAccelerationGrid` in this sample app is a custom data class representing an overlay, so it subclasses `MKShape`.

An overlay contains two key properties, a coordinate that defines the center point of the overlay, and the boundingMapRect that the system expresses as an MKMapRect that completely encompasses the overlay's content. When the app uses system-provided overlay objects, the system automatically computes the values of these properties. When defining a custom overlay object, the class needs to implement these properties and return appropriate values, as the `PeakGroundAccelerationGrid` class in this app demonstrates.

Load overlay data using GeoJSON

GeoJSON is a standards-based data format for representing geographic data, and apps often receive overlay data from a server in GeoJSON format. Rather than connect to a server, this app uses a local GeoJSON file containing `MultiPolygon` features into an `MKMultiPolygon` by using MKGeoJSONDecoder.

```
init() {
    /// In a real app, the event data probably downloads from a server. This sample
    if let jsonUrl = Bundle.main.url(forResource: "event", withExtension: "json") {
        do {
            let eventData = try Data(contentsOf: jsonUrl)

            // Use the `MKGeoJSONDecoder` to convert the JSON data into MapKit objec
            let decoder = MKGeoJSONDecoder()
            let jsonObjects = try decoder.decode(eventData)

            parse(jsonObjects)
```

```swift
        } catch {
            print("Error decoding GeoJSON: \(error).")
        }
    }
}

private func parse(_ jsonObjects: [MKGeoJSONObject]) {
    for object in jsonObjects {

        /**
         In this sample's GeoJSON data, there are only GeoJSON features at the top l
         implementation that parses arbitrary GeoJSON files needs to check for GeoJS
        */
        if let feature = object as? MKGeoJSONFeature {
            for geometry in feature.geometry {

                /**
                 Separate annotation objects from overlay objects because you add th
                 GeoJSON only contains `Point` and `MultiPolygon` geometry. In a gen
                */
                if let multiPolygon = geometry as? MKMultiPolygon {
                    overlays.append(multiPolygon)
                } else if let point = geometry as? MKPointAnnotation {
                    // The name of the annotation passes in the feature properties.
                    // Parse the name and apply it to the annotation.
                    configure(annotation: point, using: feature.properties)
                    annotations.append(point)
                }
            }
        }
    }
```

Display overlays on a map view

The app adds the overlay data objects to the map in a specific order to ensure that certain overlays display on top of others. To specify whether an overlay is above or below content that the map provides, such as roads and labels, the app calls `addOverlay(_:level:)` with the `level` parameter as a value that `MKOverlayLevel` provides.

```swift
mapView.addOverlay(reliefTileOverlay, level: .aboveLabels)
```

The overlay data object doesn’t draw the overlay on the map. A second object, called an *overlay renderer*, handles the drawing responsibilities for displaying the overlay on the map view. After adding an overlay, the map view calls `mapView(_:rendererFor:)` on its delegate to create an appropriate renderer object. Because this app demonstrates many different overlays, its implementation of `mapView(_:rendererFor:)` creates many different types of overlay renderers. Most apps only use a small number of overlay types, so this function only needs to create the small number of corresponding overlay renderer types.

```
func mapView(_ mapView: MKMapView, rendererFor overlay: MKOverlay) -> MKOverlayRende
    switch overlay {
    case let overlay as MKCircle:
        return createCircleRenderer(for: overlay)
    case let overlay as MKGeodesicPolyline:
        return createGeodesicPolylineRenderer(for: overlay)
    case let overlay as MKPolyline where currentExample == .gradientPolyline:
        return createGradientPolylineRenderer(for: overlay)
    case let overlay as MKPolyline:
        return createPolylineRenderer(for: overlay)
    case let overlay as MKPolygon where currentExample == .blendModes:
        return createBlendModesPolygonRenderer(for: overlay)
    case let overlay as MKPolygon:
        return createPolygonRenderer(for: overlay)
    case let overlay as MKMultiPolygon:
        return createMultiPolylineRenderer(for: overlay)
    case let overlay as PeakGroundAccelerationGrid:
        return createCustomRenderer(for: overlay)
    case let overlay as MKTileOverlay:
        return createTileRenderer(for: overlay)
    default:
        return MKOverlayRenderer(overlay: overlay)
    }
}
```

After the app creates the renderer and returns it from `mapView(_:rendererFor:)`, the map view uses the `boundingMapRect` property on the overlay data object to determine when the returned overlay renderer draws the overlay on the map.

The `overlays` property of `MKMapView` stores the registered overlays, but the order of the array doesn’t necessarily reflect their visual order on the map. To understand the rendering order of overlays at a specific level, see `overlays(in:)`.

Use the standard overlay objects for common shapes

The app highlights specific map regions with basic shapes by using the standard overlay classes, including MKCircle, MKPolyline, and MKPolygon. For example, it creates a circle overlay using MKCircle with a center coordinate and a radius specified in meters to highlight San Francisco.

```
/// Create a circle overlay that centers on San Francisco.
let circleOverlay = MKCircle(center: LocationData.sanFranciscoGeographicCenter, radi
mapView.addOverlay(circleOverlay, level: overlayLevel)
```

The standard overlay classes define the basic shape of the overlay, and the app uses them in conjunction with the MKCircleRenderer, MKPolylineRenderer, or MKPolygonRenderer classes to handle the rendering of that shape on the map. The app creates a renderer for the circle described above with the following code:

```
func createCircleRenderer(for circle: MKCircle) -> MKCircleRenderer {
    /**
     Some of the most common customizations for an `MKOverlayRenderer` include custo
     fill color of an enclosed shape, or the stroke color for the edge of the shape.
     */
    let renderer = MKCircleRenderer(circle: circle)
    renderer.lineWidth = 2
    renderer.strokeColor = .systemBlue
    renderer.fillColor = .systemTeal
    renderer.alpha = 0.5

    return renderer
}
```

When the app uses the provided renderer classes for common shapes, MapKit vectorizes overlay shapes so that they always remain sharp while the map scales. The app doesn’t change the default value of the shouldRasterize property, so the standard overlay shapes always remain sharp. Subclassing any of the provided renderer objects and providing a custom implementation of draw(_:zoomScale:in:), like the PeakGroundAccelerationOverlayRenderer class, automatically enables rasterized rendering.

Set drawing properties to customize the overlay rendering

The standard overlay renderers allow customization of common drawing properties for the fill and edges. For example, the app displays an MKPolyline overlay using dashes instead of a solid line, and sets a customized dash pattern using the `lineDashPattern` property of an `MKPolylineRenderer`.

```
/**
 Apply a custom pattern to the line, alternating dash length with space length in di
 The pattern repeats for the length of the polyline.
 */
renderer.lineDashPattern = [20 as NSNumber,   // Long dash
                            10 as NSNumber,   // Space
                             5 as NSNumber,   // Shorter dash
                            10 as NSNumber,   // Space
                             1 as NSNumber,   // Dot
                            10 as NSNumber]   // Space
```

MapKit also provides `MKGradientPolylineRenderer` to draw a polyline with a color gradient. The app configures a gradient renderer in the following way:

```
func createGradientPolylineRenderer(for line: MKPolyline) -> MKGradientPolylineRende
    let renderer = MKGradientPolylineRenderer(polyline: line)

    let colorPalette: [UIColor] = [.systemPurple, .systemMint, .systemOrange, .syste

    /**
     Gradient polylines take an array of colors and an array of locations to place e
     The system describes the location values as a fractional distance along the pol
     1.0 (representing the last point).

     For apps that add a color to the gradient per point in the polyline, `MKPolylir
     compute the location value for use with the gradient polyline.
     */
    var unitDistances = [CGFloat]()
    var colors = [UIColor]()
    var index = 0
    while index < line.pointCount {
        // Figure out the location of a point in the polyline as a fraction of unit
        unitDistances.append(line.location(atPointIndex: index))

        // Pick a color to add to the gradient.
        colors.append(colorPalette[index % colorPalette.count])

        index += 1
    }
    renderer.setColors(colors, locations: unitDistances)
    renderer.lineWidth = 2

    return renderer
}
```

Render multiple overlays with the same style efficiently

It's common to have multiple related overlays appear on the map with an identical visual style. For example, the app displays a map of an outdoor event that uses multiple overlays to show where the stage is located in relation to different event booths. Because the app shows each of these overlays using the same color scheme, it groups the individual overlay objects together using an `MKMultiPolygon` object.

The app then adds the grouped overlay to the map view, rather than adding the individual overlays, to avoid requesting a separate renderer for each overlay from its delegate. Instead, the app returns an `MKMultiPolygonRenderer` from the delegate. This returned renderer applies the same drawing properties to all overlays within the `MKMultiPolygon`. This is more efficient than creating a renderer for each overlay.

```
func createMultiPolylineRenderer(for multiPolygon: MKMultiPolygon) -> MKMultiPolygo
    let renderer = MKMultiPolygonRenderer(multiPolygon: multiPolygon)
    renderer.fillColor = UIColor(named: "MultiPolygonOverlayFill")
    renderer.strokeColor = UIColor(named: "MultiPolygonOverlayStroke")
    renderer.lineWidth = 2.0

    return renderer
}
```

> Note
>
> To maximize rendering efficiency, developers need to group overlays using `MKMultiPolyline` or `MKMultiPolygon` whenever the visual style is the same. This is especially important when an app displays significant numbers of visually identical overlays.

Create visual effects using blend modes

Blend modes relate the content that draws in an overlay to the content that draws behind the overlay. This enables creating visual effects on the map by adding overlays with a specific Z-order and applying a blend mode on the different overlays. For example, the app highlights a park hosting an outdoor event by using blend modes to lighten the map areas outside the park and to amplify the colors within the park.

To create such an effect, the app uses two overlays. The first overlay covers the entire map except for an inner polygon for the park, and the second overlay is a polygon outlining only the park.

```
/// Turn an array of points into a polygon. You can also load the polygon from a Geo
let parkPolygon = MKPolygon(coordinates: LocationData.plazaDeCesarChavezParkOutline,
                                  count: LocationData.plazaDeCesarChavezParkOutline.c

/// Create an overlay polygon that covers the entire world, except for a cutout of t
let worldPoints = [MKMapRect.world.origin,
                   MKMapPoint(x: MKMapRect.world.origin.x, y: MKMapRect.world.origin
                   MKMapPoint(x: MKMapRect.world.origin.x + MKMapRect.world.size.wid
                   MKMapPoint(x: MKMapRect.world.origin.x + MKMapRect.world.size.wid
                              y: MKMapRect.world.origin.y + MKMapRect.world.size.hei
let desaturatedBase = MKPolygon(points: worldPoints, count: worldPoints.count, inter
```

When the map view requests renderer objects from the map delegate for these overlay objects, the app configures the blendMode property with the screen blend mode to lighten the map area outside the park, and the colorBurn blend mode to darken the colors within the park.

```
func createBlendModesPolygonRenderer(for overlay: MKPolygon) -> MKPolygonRenderer {
    let renderer = MKPolygonRenderer(polygon: overlay)

    if overlay.interiorPolygons == nil {
        /// An overlay without `interiorPolygons` is the overlay highlighting the pa
        renderer.fillColor = traitCollection.userInterfaceStyle == .light ? .darkGra
        renderer.blendMode = .colorBurn
    } else {
        /// An overlay with `interiorPolygons` is the background overlay to desatura
        renderer.fillColor = .gray
        renderer.blendMode = .screen
    }
    return renderer
}
```

For further information on blend modes, see “Setting Blend Modes” and “Using Blend Modes with Images” in Quartz 2D Programming Guide.

Define a custom overlay renderer

To draw complex overlays that go beyond drawing boundaries and filling standard overlay shapes, this sample code project creates a custom overlay renderer. The app contains data related to earthquake hazards, and defines a custom `MKOverlay` to represent that hazard data. It also defines a custom overlay renderer to draw a color-coded, shaded map of hazards based on the data.

To create a custom overlay renderer, the app subclasses `MKOverlayRenderer` and implements `draw(_:zoomScale:in:)` to draw the earthquake data into the provided `CGContext`. MapKit calls this method concurrently on multiple background queues for the app to draw the overlay, with each call rendering a specific section of the overlay within the bounds of the `mapRect` parameter.

```
override func draw(_ mapRect: MKMapRect, zoomScale: MKZoomScale, in context: CGConte
    // Don't draw anything that doesn't intersect the data set.
    guard mapRect.intersects(data.boundingMapRect) else { return }

    /**
     Determine the section of the overlay to render. MapKit breaks overlays into mul
     Each call to `draw(_:zoomScale:in:)` should only render within bounds of the pr
     If your drawing implementation needs to draw content outside of the provided `m
     rectangle by calling `clip(to:)` on the `CGContext`.
     */
    let intersection = mapRect.intersection(data.boundingMapRect)
```

When the app draws the custom overlay, it uses `MKMapPoint` data associated with the overlay to define shapes. When it needs to convert data between MapKit geometry and Core Graphics geometry, it uses `point(for:)`.

```
let point1Conversion = point(for: coord1.mapPoint)
```

MapKit also provides `mapRect(for:)` for converting rectangles between MapKit geometry and Core Graphics geometry. When implementing a custom renderer, the app doesn't use the `bounds` or `frame` of the `MKMapView` as reference points during drawing.

Load custom map tiles

MapKit supports using custom bitmap map tiles to provide an underlying map that's customizable. For example, this app displays map tiles that emphasize rivers and mountains.

To use a custom bitmap map tile overlay, the app uses `MKTileOverlay` to manage loading the tile data and `MKTileOverlayRenderer` to render the map tiles. When creating the tile overlay, the app provides a URL template with placeholder values for the tile position, zoom level, and scale factor to the `MKTileOverlay`. When the tile overlay loads the data, MapKit replaces the placeholder values with the required values to load tiles for a specific map region according to the EPSG:3857 spherical Mercator projection coordinate system.

The URL template can be either an HTTP URL or a file URL, and this app uses both. For example, it loads some map tiles bundled with the app and specifies a file URL template that locates the map tiles within the app's bundle.

```
let tileDirectoryName = "tileData"
guard let resourcePath = Bundle.main.resourcePath else { return }
let localPath = "file://\(resourcePath)/\(tileDirectoryName)/{z}/{x}/{y}.jpg"
let tileOverlay = MKTileOverlay(urlTemplate: localPath)
```

When the app loads tiles from a server, it also does so with a URL template, replacing the file URL with an HTTP URL.

The `CustomLoadingTileOverlay` class in this sample code project implements `loadTile(at:result:)` to show how to customize tile-loading behavior for specialized loading needs.

Use an overlay as an annotation

The `MKOverlay` protocol conforms to the `MKAnnotation` protocol. As a result, all overlay objects are also annotation objects. When adding an overlay as an annotation to the map, `MKMapView` displays it at the overlay's `coordinate` property. For example, the app uses a polygon outlining the park for an outdoor concert as an annotation to label the concert location.

```
/**
 Types that derive from `MKOverlay`, such as `MKPolygon`, also conform to `MKAnnotat
 as well as place an annotation on the overlay to label it.
 */
parkPolygon.title = "Concert Location"
mapView.addAnnotation(parkPolygon)
```

Because the app treats the overlay object as both an overlay and an annotation, it's responsible for adding and removing the object from the map view as both an annotation and as an overlay.

See Also

Samples

{} Displaying an updating path of a user’s location history

Continually update a MapKit overlay displaying the path a user travels.