Bellus3D SDK Guide for iOS iPhone X

Contents

---

1. Introduction

Bellus3D SDK is an iOS framework intended to build realistic 3D models of user's face. To build the model the framework utilizes Apple TrueDepth camera technology.

2. Integrating the SDK to your project

Bellus3D SDK is distributed as iOS framework bundle. The client project that will use the framework may be written in Swift and/or Objective-C. Framework should be added to Xcode projects as a regular third-party framework - no special additional actions needed. The following are the basic steps describing how to integrate the Bellus3D framework into Xcode project.

1. Copy Bellus3D.framework to the project directory and add it to Xcode project (usually by drag and drop it). Ensure the Framework Search Paths of your target contain paths to the directory with the framework.
2. Add framework to a link build phase of your project's target(s).
3. Add build phase for copying framework into your target's application bundle.
4. In case the code framework will be used from Swift add the bridging header to your project and import the header. For more information about how to use frameworks in your project see the Apple Developer guides for linking a target to libraries and frameworks and the Apple Developer Framework Programming Guide.

3. Basic workflow

The general flow is as follows.

1. **Scanning:** Perform face scan using `B3DHeadScanner` instance
2. **Building 3D Model:** If the scan is completed successfully B3DHeadScanner object passes `B3DHeadScanner.SessionData` object to observer. Use this object to initiate scan processing using `B3DHeadProcessor.`
3. **Rendering 3D Model:** If the processing is completed successfully `B3DHeadProcessor` passes `B3DHeadMesh` object to observer. Use this object to render 3D model.

4. Scanning

Scan is performed using the `B3DHeadScanner` scanner instance. The following is the basic work flow for `B3DHeadScanner`

1. **Create scanner using** `-initWithCamera`: initializer. In turn camera object should be created passing `ARSession` object. Session object is usually taken from `ARSCNView`.
2. **Configure scanner** with `B3DHeadScannerSettings` instance.
3. **To observe scan process**, conform to `B3DHeadScannerObserver` protocol and add it as an observer to `B3DHeadScanner` using `-addObserver:` method. Observer methods will be called by scanner to notify about scan events.
4. **To ensure head position is correct** and ready for scan, launch tracking by calling - `startTracking` method on scanner object. During tracking Bellus3D framework will provide info about current head position to observers by calling - `headScanner:didTrackFacePosition`: method.
5. During tracking when observer will be notified that face position is ready to scan (receiving `B3DHeadScannerFacePositionReadyToScan` value). In this case scan can be initialed by calling `-startScanning` method on scanner object.
6. During scan scanner object will provide hints about how to turn head in order to successfully perform scan. For this purpose scanner object will call method - `headScanner:didProvideHint:` on its observers.
7. Method `-headScanner:didCompleteScanningSuccessfully:` resultingSessionData: will be called on observer when scan is completed.

5. Building 3D model

1. **Create `B3DHeadProcessor`** instance using default initializer.
2. **Configure processor** with `B3DHeadProcessorSettings` instance.
3. **To observe processing** conform to `B3DHeadProcessorObserver` protocol and add it as an observer to B3DHeadScanner using `-addObserver:` method. Observer methods will be called by processor object to notify about processing events.

4. **Initiate model processing** calling `-startProcessingWithSessionData:` method on processor object. Pass `sessionData` retrieved from scan completion method as a parameter.
5. **Method** `-headProcessor:didCompleteProcessingWithMesh:` will be called when processing is completed.

6. Rendering 3D model

After processing the successful B3DHeadMesh object is returned in completion method to observers. The object should be used to get data needed for model rendering and to generate model files if needed. Head mesh provides the model data as a separate texture image and a number of vertices forming polygons mesh.

1. **To get texture** use `-texture` method returning `CGImageRef` instance.
2. **To get mesh data** use `-polygonListReturningError:` method. It returns `B3DPolygonList` object that provides the ability to obtain polygons data in order to build mesh.

a) Polygon mesh low-level data

Since the client application may use an arbitrary renderer (Open GL, Metal or some high-level API) `B3DPolygonList` provides generic format that can be converted to a specific one accepted by a particular renderer. That is, polygon list represents a sequence of triangles that form the mesh. Each triangle is represented by 3 `B3DVertex` objects. To iterate all the triangles the client code should call `-enumerateTrianglesUsingBlock:` method of `B3DPolygonList`. A closure passed as a parameter will be called for each triangle receiving 3 vertices as its arguments.

There is also an option to get raw vertex data. For this purpose `B3DPolygonList` methods: `-getVertices:bufferElementCount:` and `-getTriangleFormingVertexPositions:bufferElementCount:` should be used.

Method `-getVertices:count:` fills the passed buffer with vertices. Each 8 elements of the buffer represent a single vertex in the following format:
`(x, y, z, nx, ny, nz, u, v)`, where:

- `x, y, z` - vertex coordinate
- `nx, ny, nz` - normal vector for the vertex
- `u, v` - texture coordinates.

Method `-getTriangleFormingVertexNumbers:count:` fills passed buffer with ordinal numbers of vertices where each 3 numbers correspond to the vertices forming triangle.

Consider the following example. Let's assume we have 4 vertices forming 2 triangles: `vertex0, vertex1, vertex2` and `vertex1, vertex2, vertex3` are forming first and second triangle correspondingly. In this case `-getVertices:count:` will provide the following vertices array of float values:

- `x0, y0, z0, nx0, ny0, nz0, u0, v0 |`
- `x1, y1, z1, nx1, ny1, nz1, u1, v1 |`
- `x2, y2, z2, nx2, ny2, nz2, u2, v2 |`
- `x3, y3, z3, nx3, ny3, nz3, u3, v3`

where:

- `x0, y0, z0, nx0, ny0, nz0, u0, v0 are forming vertex0,`
- `x1, y1, z1, nx1, ny1, nz1, u1, v1 are forming vertex1 etc.`

Array referencing triangles provided by `-getTriangleFormingVertexNumbers:count:` will have the following values: 0, 1, 2, 1, 2, 3. In this array 0, 1, 2 correspond to vertex0, vertex1, vertex2, and 1, 2, 3 are correspond to vertex1, vertex2, vertex3.

For convenience B3DPolygonList class also provides methods -vertexData and -triangleFormingVertexPositionData returning the same data as previously described -getVertices:bufferElementCount: and -getTriangleFormingVertexPositions:bufferElementCount: but wrapped in Data object.

b) Polygon mesh high-level data

On iOS the SceneKit framework is widely used for rendering. For this reason, B3DPolygonList provides method -modelMesh returning MDLMesh object representing the mesh. The object may be used for rendering by SceneKit.

c) Material settings

Class B3DHeadMesh provides information about geometry of the mesh and a separate texture that should be applied for rendering realistic model. It's a responsibility of a particular client to choose rendering options such as lightning, material settings etc. to achieve the desired result of rendering. However, for optimal rendering quality B3DHeadMesh also provide preferred material settings that may be used during rendering. They are represented by B3DMaterialSettings object that is returned from +preferredMaterialSettings method of B3DHeadMesh.