Bellus3D iOS SDK Developers Guide

By Bellus3D, Inc.

Contents

Introduction

The Bellus3D SDK is an iOS framework intended to build realistic 3D model of a person's face. The framework utilizes Apple's TrueDepth camera technology.

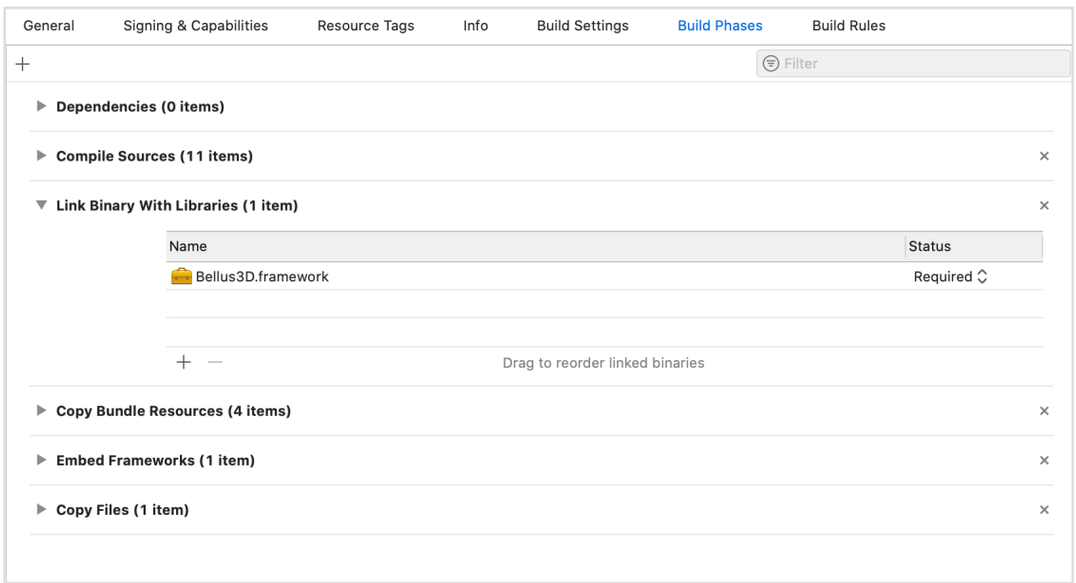
The SDK is designed to work with target iOS version 11.1 or later.

Also, the SDK zip file should include a iOS sample app in the IOSSampleApp folder. That app is written in Swift and will give you some ideas for how to call the Objective-C APIs in Swift. The sample app does scanning, processing and rendering.

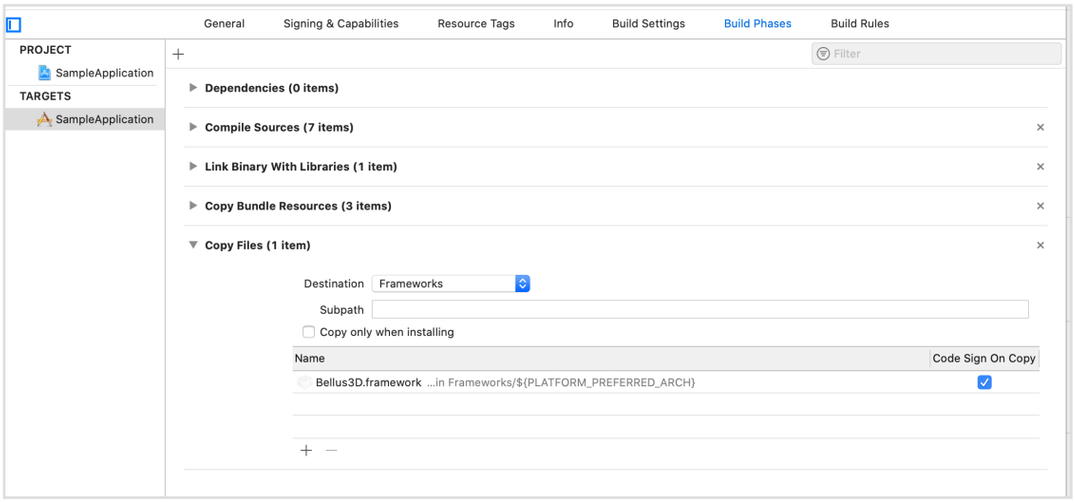
Integrating To Your Project

The Bellus3D SDK is distributed as iOS framework bundle. Client projects using the framework can be written in Swift and/or Objective-C. The framework should be added to the Xcode project as a regular third party framework - no special actions are needed. The following are the steps describing how to integrate Bellus3D framework into your Xcode project. If any of these steps are not clear, please look at how the framework is integrated into the provided Sample application.

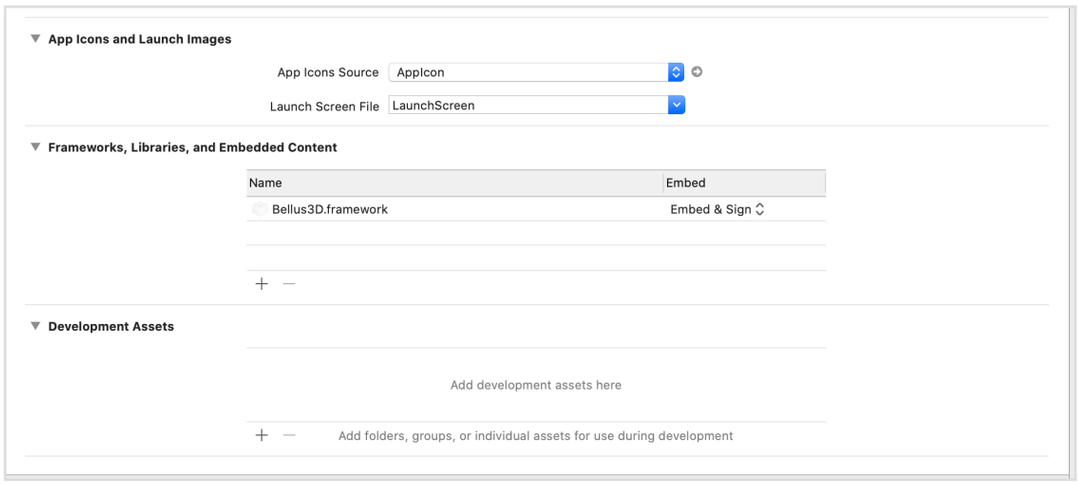
1. Copy the Bellus3D.framework to the project directory and add it to your Xcode project (usually by drag and drop). Ensure the Framework Search Paths of your target contains the path to directory with the framework. Depending on your set up, you may need to disable bitcode in Build Settings.
2. If it is not already there, add the Bellus3D.framework to a Build Phases—>Link Binary With Libraries phase of your project's target(s).



3. Add a build phase for copying the framework into your target's application bundle.

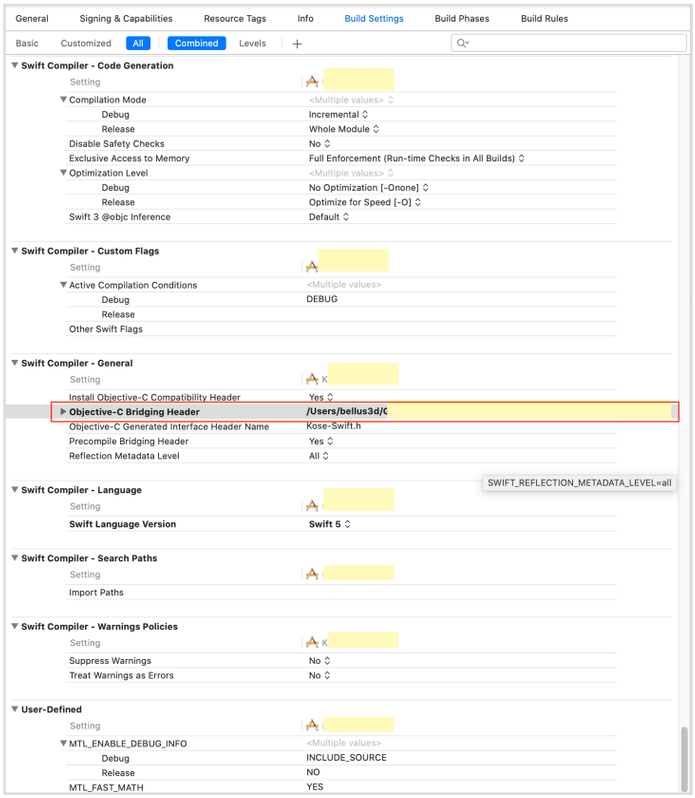


Your General Tab, when selecting the app target, should look like this:



4. If your app is developed using Swift add a bridging header to your project and add this : import <Bellus3D/Bellus3D.h> header line.

   Also, make sure you have the name of the bridging header added to the Swift Compiler - General section in Build Settings. If you don't then you will get all types of errors as your app won't recognize things like HeadScanner (see Sample App). It should look like this:



For more information about how to use frameworks in your project see Link a target to libraries and frameworks and Framework Programming Guide Apple guides.

Please carefully review the included Sample Application for how to implement some of the APIs. Look at the AppDelegate to see how to set your credentials and how to activate the SDK. You need to activate the SDK every time the app starts, not just once. We do not support offline usage. And be sure to not start tracking for face scanning until the completion handler of the activateSDK method returns.

Also if you are having issues and you need to support please create a support ticket here: https://zfrmz.com/t6ezZ7FJWi4y5l4sh487

And please enable B3DLogger (see example in the Sample Application AppDelegate) and include the created log files in the support ticket.

General Workflow

The general flow is as follows:

1. Use `B3DAuthenticationManager` to authenticate with Bellus3D cloud API using your developer credentials and to activate Bellus3D SDK. Possession of developer account is a requirement and the Bellus3D SDK cannot be used until it will be successfully activated by using `activateSDKWithCompletionHandler:` method. For most of the time Bellus3D SDK can work offline but for activation active internet connection is a must. In is recommended to not activate Bellus3D SDK immediately after application launch but to wait until your application will actually use any of Bellus3D SDK features.
2. Use the tracking mode of `B3DHeadScanner` and its preview layer in your application to properly position the face.
3. Perform a face scan using `B3DHeadScanner` instance.
4. When the scan completes successfully `B3DHeadScanner` object passes a `B3DHeadScanner.SessionData` object to an observer. Use this object to start processing using `B3DHeadProcessor`.
5. When processing is completed `B3DHeadProcessor` passes `B3DHeadMesh` object to an observer. Use this object to render your 3D face model.

1. Scanning

Scanning is performed using `B3DHeadScanner` scanner instance. The following is the work flow for `B3DHeadScanner`:

1. Create a scanner using `-initWithCamera:` initializer. Make sure the camera object is created with the desired orientation.
2. Configure the scanner with the `B3DHeadScannerSettings` instance.
3. To observe the scanning conform to `B3DHeadScannerObserver` protocol and add it as an observer to `B3DHeadScanner` using `-addObserver:` method. Observer methods will be called by the scanner and notified of events as they happen.
4. To ensure the head position is correct for scanning, launch tracking by calling the `-startTracking` method on the scanner object. Tracking will fail if the Bellus3D SDK was not successfully activated. During tracking the Bellus3D framework will provide information about the current head position to observers by calling the `- headScanner:didTrackFacePosition:` method.
5. To see the face during tracking use the `previewLayer` (of type `CALayer`) provided by the `B3DHeadScanner` class. Add this layer to the UI of your application.
6. During tracking, the observer will be notified that the face position is ready to be scanned (by receiving the `B3DHeadScannerFacePositionReadyToScan` value). In this case, the scan can be initiated by calling the `-startScanning` method on the scanner object. Scanning will fail if Bellus3D SDK was not successfully activated.
7. During the scan, the scanner object will provide hints about how to turn the head in order to successfully perform a scan. The scanner object will call the method `-headScanner:didProvideHint:` on its observers.
8. Method `-headScanner:didCompleteScanningSuccessfully:resultingSessionData:` will be called on the observer when the scan is completed.

Scan Modes

There are 3 possible resulting scanned model types: Face Face+Neck, and Full Head. To choose a model type, set the appropriate scan mode of `B3DHeadScannerSettings` instance. See the table below describing the scan modes and how they apply to mode types.

Scan Mode	Description	Resulting Model Type
`B3DHeadScannerSettingsScanMode180Degrees`	Simple frontal scan of 180 degrees angle of view of the head	**Face**
`B3DHeadScannerSettingsScanMode270Degrees`	Frontal scan of 270 degrees angle of view of the head	
`B3DHeadScannerSettingsScanMode270DegreesLongScan`	Scan of 270 degrees angle of view of the head including neck	**Face + Neck**
`B3DHeadScannerSettingsScanMode360`	360 model	**Full Head**

Texture Resolution

The resulting texture quality depends on the camera capabilities. Our SDK automatically chooses the maximum video stream resolution available for the scan.

2. Processing

1. Create a `B3DHeadProcessor` instance using the default initializer.
2. Configure the processor with the `B3DHeadProcessorSettings` instance.
3. To observe processing, conform to the `B3DHeadProcessorObserver` protocol and add it as an observer to `B3DHeadScanner` using the `-addObserver:` method. Observer methods will be called by the processor object to notify events as they happen.
4. Initiate model processing by calling the `-startProcessingWithSessionData:` method on processor object. Pass the `sessionData` retrieved from the scan complete method as a parameter. Processing will fail if Bellus3D SDK was not successfully activated.
5. The method `-headProcessor:didCompleteProcessingWithMesh:` will be called when processing is completed. If Bellus3D SDK was activated using a free trial developer account then the generated texture will have embedded Bellus3D watermark. To get rid of watermark your account needs to be upgraded to the paid account.

3. Rendering 3D Model

After processing succeeds, a `B3DHeadMesh` object is returned in the completion method to the observers. The object should be used to get the data needed for model rendering. The head mesh provides the model data as separate texture images and a number of vertices to form a polygon mesh.

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

Polygon Mesh Low-Level Data

Since client applications may use an arbitrary renderer (Open GL, Metal or some high-level API) `B3DPolygonList` provides a generic format that can be converted to a specific one accepted by a particular renderer. The polygon list represents a sequence of triangles that form the mesh. Each triangle is represented by 3 `B3DVertex` objects. To iterate over all the triangles the client code should call the `-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 the raw vertex data. For this purpose, use the `B3DPolygonList` methods:

```
-getVertices:bufferElementCount:
-getVertexCoordinatePositions:bufferElementCount:
-getVertexNormalPositions:bufferElementCount:
-getVertexTextureCoordinatePositions:bufferElementCount:
```

The method `-getVertices:bufferElementCount:` fills the passed buffer with vertices. Each of the 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` - are vertex XYZ coordinates
- `nx, ny, nz` - are normal vectors for the vertex
- `u, v` - are texture coordinates.

The three remaining methods are used to fill passed buffers with indices of XYZ coordinates, normal vectors and texture coordinates which should be used to find data in vertices buffer.

To export to the file format:	use the file extension:
OBJ	`obj`
PLY	`ply`
Zipped OBJ	`zip`
Zipped PLY	`ply.zip`
GLB	`glb`
STL	`stl`

Assuming that:

Method `-getVertices:bufferElementCount:` will return:

```
[ 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  ]
```

Method `-getVertexCoordinatePositions:bufferElementCount:` will return:

```
[ 0, 1, 2 ]
```

Method `-getVertexNormalPositions:bufferElementCount:` will return:

```
[ 0, 2, 1 ]
```

Method `-getVertexTextureCoordinatePositions:bufferElementCount:` will return:

```
[ 0, 1, 3 ]
```

Then single continuous array of vertices used by most renderers will look like this:

```
[ x0, y0, z0, nx0, ny0, nz0, u0, v0,
  x1, y1, z1, nx2, ny2, nz1, u1, v1,
  x2, y2, z2, nx1, ny1, nz3, u3, v3, ]
```

Polygon Mesh High-Level Data

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

Material Settings

The class `B3DHeadMesh` provides information about the geometry of the mesh. A separate texture should be applied for rendering realistic models. It is the responsibility of the developer to choose rendering options such as lightning, material settings, etc to achieve the desired rendering result.

However, for optimal rendering quality, `B3DHeadMesh` also provides a preferred Material Settings that is recommended for rendering. The Settings are represented by the B3DMaterialSettings object that is returned from `+preferredMaterialSettings` method of `B3DHeadMesh`. Material Settings contain information about various material rendering options. Additionally, if SceneKit is used for rendering, `B3DMaterialSettings` contains a transform that should be applied to the texture for correct rendering.

Exporting model data

Export Feature Activation

The class `B3DHeadMesh` provides the ability to export model data. To enable exporting, the feature should be activated. It can be done using one of the following activation methods:

- By sharing the model with Bellus3D (uploading model data).
- By retrieving a token from Bellus3D (there are a limited number of tokens per developer).

To activate export, a method:

```
-requestActivationOfPremiumFeaturesWithModelInfo:activationMethod:completionHandler:
```

of `B3DHeadMesh` class should be used. To use this method:

- Pass the desired Premium Feature activation method as an activationMethod parameter.
- In case the activation method is "by sharing," you may also pass optional modelInfo describing the user ID and model name.

This method may connect to the Bellus3D server in order to activate exporting using the chosen method. If activation by token was chosen and sufficient number of tokens is stored locally on the device that activation can be done offline. In case activation succeeds, a completion block is called with the corresponding instance of `B3DHeadMeshExporter` class and nil error. If activation fails, a completion is called with the appropriate error and a nil `B3DHeadMeshExporter` object.

If you are using the SDK on an iPad Pro for scanning, you have to use the token activation method.

Export Usage

The class `B3DHeadMeshExporter` provides the following export functionality:

1. **Change model resolution:**
   Number of triangles and texture resolution can be changed by modifying `meshResolution` property of `B3DHeadMeshExporter` object. Three resolutions are available:

```
/// 64K triangles & 4096x4096 texture size.
B3DHeadProcessorSettingsMeshResolutionStandard,

/// 250K triangles & 4096x4096 texture size.
B3DHeadProcessorSettingsMeshResolutionHigh,

/// 8K triangles & 1024x1024 texture size.
B3DHeadProcessorSettingsMeshResolutionLow
```

2. **Export the model data. Export to the following formats is supported:**
   - OBJ (.obj file with texture and supporting files)
   - PLY (.ply file with the texture)
   - Zipped OBJ (.obj file with texture and supporting files archived in zip format)
   - Zipped PLY (.ply file with the texture file archived in zip format)
   - GLB
   - STL

For exporting, the following method of `B3DHeadMeshExporter` should be used:

```
-(BOOL)exportMeshFilesToDirectoryAtURL:(NSURL *)directoryURL
                              filename:(NSString *)filename
                                 error:(NSError ** _Nullable)error
```

The method chooses the format of the resulting file basing on the file extension as described below.

2. Export of the landmark data:

For this purpose the following methods of `B3DHeadMeshExporter` should be used. Each method has a detailed description in the `B3DHeadMeshExporter.h` file.

1) For exporting landmark data only:

```
- (BOOL)writeLandmarksToFileAtURL:(NSURL *)fileURL
      inModelFileCoordinateSpace:(BOOL)inModelFileCoordinateSpace
                           error:(NSError ** _Nullable)error
```

2) For exporting both landmark data file and model in .OBJ format:

```
- (BOOL)writeLandmarksToDirectoryAtURL:(NSURL *)directoryURL
            landmarksDataFileName:(NSString *)landmarksDataFileName
                    modelFileName:(NSString *)modelFileName
                           error:(NSError ** _Nullable)error
```

3) For getting landmark data in memory

```
- (NSDictionary<NSString *, NSArray *> *)landmarkData
```

3.Convert 2D landmarks to 3D landmarks.

Class `B3DHeadMeshExporter` allows conversion of 2D landmarks detected on model texture by using a custom face detector to 3D landmarks. Use the following method to do conversion:

```
- (nullable NSArray *)landmarks3DFrom2D:(NSArray *)landmarks2d
                            textureSize:(CGSize)textureSize
                                  error:(NSError ** _Nullable)error;
```

4. Export the mesh data in the internal file format.

Class `B3DHeadMeshExporter` allows to write internal mesh data to file. The data represents head mesh in Bellus3D specific internal format. Use the following method to write the data:

```
- (BOOL)writeMeshDataToFileAtURL:(NSURL *)fileURL
                           error:(NSError ** _Nullable)error
```