String™

Augmented Reality SDK for iOS OpenGL

USER GUIDE

Table of Contents

Hello, and thank you for choosing String™

This manual explains how to add String™ to a new or existing project. It requires experience with Cocoa Touch and OpenGL ES. All of the example code can be found in full working form in the tutorial project.

The first thing you do is to add these three files from the Libraries and Headers folders of the String™ SDK to your Xcode project: libStringOGL*.a, StringOGL.h and TrackerOutput.h. You also need to add the frameworks AVFoundation, CoreGraphics, CoreMedia and CoreVideo.

The class in your app which owns the `StringOGL` instance, normally a view controller, needs to include StringOGL.h and implement the `StringOGLDelegate` protocol. This protocol has only one required method, `render`, in which you perform your own rendering.

Please note that the iOS simulator is not supported by String™, as it lacks a camera API. The String library requires iOS 4.0 or newer.

1. Initialization

In your initialization code, please set up your framebuffer, renderbuffers, projection matrix, viewport and other initial states as you normally would. If you're using a third party 3D engine, this is probably taken care of for you.

Then you initialize String™:

```
stringOGL = [[StringOGL alloc] initWithDelegate: self context: myEAGLContext
frameBuffer: myFrameBuffer leftHanded: NO];
```

In your dealloc method:

```
[stringOGL release];
```

Please see the method reference below for a full description of each parameter.

Next, please provide String™ with your geometric rendering parameters:

```
[stringOGL setProjectionMatrix: myProjectionMatrix viewport: myViewport
orientation: [self interfaceOrientation] reorientIPhoneSplash: NO];
```

Please see below for info on how to configure your projection matrix and viewport.

The recommended way to handle orientation is to never rotate your 3D view, as the camera frame and rendered 3D graphics will be orientation-agnostic anyway. If you'd like to have rotating views on top of the 3D view, the best known way at the time of writing is to first add your rotating view controller's view to the window (making it the root view controller), then add the 3D view and call `bringSubviewToFront` for your rotating view controller's view.

The last initialization step is to load image markers:

```
myMarkerID = [stringOGL loadImageMarker: @"MyMarker" ofType: @"png"];
```

This will load MyMarker.png from your main bundle. Please note: this method returns an integer identifier which you can later compare to the imageID of detected markers to match them with their appropriate content. Please see `loadImageMarker:ofType:` method reference for more on image markers.

Important: As of version 1.1.1, `stringOGL`'s default state is *not* paused. As pause requests are counted, this means if you're initializing the `stringOGL` object at startup and are calling `resume` in `applicationDidBecomeActive:`, you'll need to call `pause` after initializing the `stringOGL` object.

Important: After initializing String™, please make sure any views or windows that may obscure the splash screen are hidden. To detect when the splash screen is finished, please implement `StringOGLDelegate`'s `splashScreenFinished` method.

---

2. Rendering and data acquisition

String™ for iOS takes care of clearing, drawing the video frame in the background, and presenting the renderbuffer for you. In other words, all you have to worry about is drawing your content. String™ "owns" your rendering loop for reasons of performance and simplicity. It doesn't mess with your OpenGL states, so there's no need to set all your states each frame. Here's an example `render` method:

```
- (void)render
{
    // Read data for markers that were detected this frame
    const int maxMarkerCount = 10;
    struct MarkerInfoMatrixBased markerInfo[10];

    int markerCount = [stringOGL getMarkerInfoMatrixBased: markerInfo
maxMarkerCount: maxMarkerCount];

    // Iterate through detected markers
    for (int i = 0; i < markerCount; i++)
    {
        // Draw appropriate content for this image marker
    }
}
```

Your `render` method should also handle updating your 3D content's internal state, such as animation, interaction etc.

3. Interpreting marker data

In the structs `MarkerInfoMatrixBased` and `MarkerInfoQuaternionBased`, you'll find all the data you need in order to respond to detected markers:

`transform` or `orientation/position`

`transform` is an OpenGL-style model/view matrix. `orientation` and `position`, the quaternion-based equivalent, represent the same transformation. The transformation's handedness is determined by the handedness you initialized `StringOGL` with. In left-handed coordinate systems, up is positive z. In right-handed systems, such as OpenGL's, up is negative z. In OpenGL 1.x, for example, this would translate into "draw my object so it hovers 0.5 world units above the marker":

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(markerInfo[i].transform);
glTranslatef(0, 0, -0.5f);
```

String™ assumes all markers have a diagonal length of 1. If you want your marker to have a different world size, just scale the tracked position. So if you'd like your marker's diagonal to have a world length of 10, you multiply the tracked position by 10.

`color`

`color` represents the average color of the detected marker in the frame, relative to the loaded marker image. In other words, if your camera captured a frame in which colors corresponded exactly to the original marker image, you'd get `{1, 1, 1}`. If the ambient light is warm, you might get something like `{1.2, 1, 0.9}` for example. How you use this is up to you. For some types of content, it can be used to imitate real-word ambient lighting conditions to great effect.

`imageID`

`imageID` is the marker image index of the detected marker. To use this to distinguish between different markers, compare it against the return values from `loadImageMarker:ofType:`.

`uniqueInstanceID`

`uniqueInstanceID` is a unique number identifying each instance of a marker. For instance, if you point the camera at two identical markers, they'll have the same imageID, but different `uniqueInstanceID`. If the tracker loses sight of a marker and rediscovers it later, it'll have a different `uniqueInstanceID`. This can be used, for instance, to distinguish between several concurrent occurrences of the same marker.

4. Configuring your projection matrix and viewport

String™ captures video in 4:3, while the iPhone and iPod Touch has a screen aspect ratio of 3:2. Here are two ways to display the video on those devices: shrink-to-fit or full screen. A more general solution for this will be added to the documentation soon.

4.1 Shrink-to-fit

In this mode you scale the video to fit in its entirety within the screen bounds. This will yield some left-over space on the screen, that can used for a toolbar for instance. This is how iOS's camera app works.

In landscape mode, set the projection matrix's vertical FOV to 36.3°, the approximate vertical FOV of the lens. Set the viewport height to your framebuffer height, and the viewport width to framebuffer height * 4 / 3.

In portrait mode, set the projection matrix's vertical FOV to 47.22° (47.22° ≈ atan(tan(36.3° / 2) * 4 / 3) * 2). Set the viewport width to your framebuffer width, and the viewport height to framebuffer width * 4 / 3.

You may position your viewport according to where you'd like to place your toolbar, or in the middle of the screen if you don't have one.

4.2 Full Screen

In this mode, the video frame is scaled to fill the entire screen, with the downside being that some video pixels are clipped and lost upon display. They are still tracked, however.

In landscape mode, set the projection matrix's vertical FOV to 32.5°. Set your viewport size to match your framebuffer size.

In portrait mode, set the projection matrix's vertical FOV to 47.22°. Set your viewport size to match your framebuffer size.

Keep in mind that these are just suggestions. You may of course set up your view geometry any way you like. See the tutorial project for an example of how to set this up.

5. Method Reference

initWithDelegate:context:frameBuffer:leftHanded:

```
- (id)initWithDelegate: (NSObject<StringOGLDelegate> *)delegate
    context: (EAGLContext *)context
    frameBuffer: (GLuint)frameBuffer
    leftHanded: (BOOL)leftHanded;
```

This method initializes String™. Should typically be called once in your app's lifetime.

`delegate`: An object that implements `StringOGLDelegate`, typically the owner of the `StringOGL` instance.

`context`: The `EAGLContext` for String™ to operate on.

`frameBuffer`: The framebuffer to which String™ renders, and whose color attachment is presented each frame.

`leftHanded`: If `NO`, String™ assumes a right-handed coordinate system, which is the norm for most OpenGL-based apps.

setProjectionMatrix:viewport:orientation:

```
- (void)setProjectionMatrix: (const float *)projectionMatrix
    viewport: (const int *)viewport
    orientation: (UIInterfaceOrientation)orientation
    reorientIPhoneSplash: (BOOL)reorientIPhoneSplash;
```

Sets the view geometry for your app.

`projectionMatrix`: Your 4×4 projection matrix; the same one you'd pass to glLoadMatrixf() in OpenGL ES 1.x

`viewport`: Your OpenGL viewport.

`orientation`: The interface orientation of your String™ view. Please see notes on orientation in the "Initialization" section.

`reorientIPhoneSplash`: Only applies to iPhone and iPod devices. Pass NO to have the String splash screen always be displayed in portrait mode on those devices, or YES to have it respect the current orientation.

setProjectionMatrix:viewport:displayOrientation:outputOrientation:

```
- (void)setProjectionMatrix: (const float *)projectionMatrix
    viewport: (const int *)viewport
    displayOrientation: (unsigned)displayOrientation
    outputOrientation: (unsigned)outputOrientation
    reorientIPhoneSplash: (BOOL)reorientIPhoneSplash;
```

This is the same as the above `setProjectionMatrix:viewport:orientation:`, except it allows you to set the display and output orientations individually. This is normally not necessary.

`projectionMatrix`, `viewport` and `reorientBranding` are the same as above.

`displayOrientation` and `outputOrientation`: These values are String's internal notion of screen orientation. Each is an integer value from 0 up to and including 3, representing a 0°, 90°, 180° or 270° rotation, respectively, relative to the internal camera output's actual orientation (`UIInterfaceOrientationLandscapeRight`).

So calling the above `setProjectionMatrix:viewport:orientation:` with an orientation of `UIInterfaceOrientationPortrait` would correspond to calling `setProjectionMatrix:viewport:displayOrientation:outputOrientation:` with `displayOrientation` and `outputOrientation` both 3.

loadImageMarker:ofType:

```
- (int)loadImageMarker:(NSString *)filename ofType: (NSString *)ext;
```

Loads an image marker from an image file. The larger the image, the more time it takes to load and decompress; 200×150 pixels, for example, is more than enough. The dimensions of the image do affect load time, but not the memory footprint once the image is loaded. When you scale down your marker image from the print-sized version, make sure you use a high quality downscaler such as Photoshop. The aspect ratio of the image is used when matching it to markers in the frame; make sure you don't change the aspect ratio when scaling. Non-square image markers are actually slightly more performance-efficient, as they make it easier for the tracker to understand the marker's orientation in the video frame.

`filename`: The name of an image file in your main bundle.

`ext`: Its extension.

Please look at Marker 1-3.png in the tutorial project for examples of image markers. The images should include the black border, but not the white around it. The black border should be reasonably thick, as this allows for better tracking in motion. The border thickness should be about 1/10th of the longest side of the image.

getMarkerInfoMatrixBased:maxMarkerCount:

```
- (unsigned)getMarkerInfoMatrixBased: (struct MarkerInfoMatrixBased
*)markerInfo maxMarkerCount: (unsigned)maxMarkerCount;
```

Called from within `render`. This retrieves info on the markers detected in a given frame. See the "Interpreting marker data" section for detailed descriptions of the marker info structs.

`markerInfo`: This is a pointer to a C-style array of at least `maxMarkerCount` marker info structs. String™ does not take ownership of the array, nor care how it's allocated.

`maxMarkerCount`: The maximum number of markers you wish to load info for.

getMarkerInfoQuaternionBased:maxMarkerCount:

```
- (unsigned)getMarkerInfoQuaternionBased: (struct MarkerInfoQuaternionBased
*)markerInfo maxMarkerCount: (unsigned)maxMarkerCount;
```

Same as `getMarkerInfoMatrixBased:maxMarkerCount:`, only retrieving quaternion-based transforms instead of matrices. See the "Interpreting marker data" section for detailed descriptions of the marker info structs.

pause and **resume**

```
- (unsigned)pause;

- (unsigned)resume;
```

Pauses and resumes String's run loop. It's important to call these when you open/close a full-screen GUI ontop of the 3D view. Also, you need to call `pause`, `resume` and `pause`, respectively, from `applicationWillResignActive:`, `applicationDidBecomeActive:` and `applicationWillTerminate:`. This is because iOS doesn't allow OpenGL calls unless the app is active.

`pause` and `resume` are counted. They both return the current "pause count" after the call. What this means is if you call `pause` twice, you have to call `resume` twice to start capturing/rendering again. This is a convenience feature.

Please make sure to call `resume` exactly once for each `pause` call.

Important: As of version 1.1.1, `stringOGL`'s default state is *not* paused. As pause requests are counted, this means if you're initializing the `stringOGL` object at startup and are calling `resume` in `applicationDidBecomeActive:`, you'll need to call `pause` after initializing the `stringOGL` object.

pauseCapture and **resumeCapture**

```
- (unsigned)pauseCapture;

- (unsigned)resumeCapture;
```

These work just like `pause` and `resume`, except they also pause video capture. The advantage to this is that you eliminate all String™-related CPU overhead while paused. The drawback is that it takes a short while for video capture to resume. With `pause` and `resume`, resumption is instant.

Please make sure to call `resumeCapture` exactly once for each `pauseCapture` call.

takeSnapshotAndPause

```
- (void)takeSnapshotAndPause;
```

Takes a snapshot of the current framebuffer, with both the video frame and your 3D content, and increases the pause count by 1. If you intend to call this, please make sure you implement the delegate method `handleSnapshot:` as defined in section “Delegate method reference”. Do not call `resume` immediately after calling `takeSnapshotAndPause`. Instead, call it in or after `handleSnapshot:`.

getCurrentVideoBuffer:viewToVideoTextureTransform:

```
- (void)getCurrentVideoBuffer: (unsigned *)buffer viewToVideoTextureTransform:
(float *)viewToVideoTextureTransform;
```

Retrieves the current video texture and a matrix to transform coordinates from view space to texture space. Should be called each frame if used, as the video stream is double buffered.

`buffer`: A pointer to an unsigned integer in which the OpenGL ES name of the current video texture will be stored.

`viewToVideoTextureTransform`: A matrix you can use to transform coordinates from view space to texture space. It should be treated as a projection matrix in that you need to divide the resulting coordinates by their w components.

6. Delegate method reference

render

```
- (void)render;
```

This is where `StringOGL`'s delegate, i.e. your app, performs its own rendering. Explained in detail in the "Rendering and data acquisition" section.

handleSnapshot:

```
- (void)handleSnapshot: (UIImage *)snapshot;
```

This optional method is called each time a snapshot (see `takeSnapshotAndPause`) has been acquired. Remember to call `resume` exactly once for each `handleSnapshot:`, though not necessarily from `handleSnapshot:` itself.

`snapshot`: An autoreleased `UIImage` containing a snapshot of the current framebuffer.

splashScreenFinished

```
- (void)splashScreenFinished;
```

This optional method is called when String's splash screen is finished displaying. You can perform OpenGL state changes here, but not rendering, as the method is called before the render loop starts. This is a good place to unhide any views that were hidden during the splash screen.

For consistency, it's also called (when present) in String™ builds that don't have a splash screen.