Windows Bridge for iOS

[UI Composition and Layout]

DEV DESIGN specification

# Overview

This document covers the architecture of WinObjC’s UI layout and composition functionality. The design aligns with our [overall UIKit strategy](https://github.com/Microsoft/WinObjC/blob/develop/docs/UIKit/WinObjC.UIKit.docx), and it builds upon the recommendations provided by Jeff Stall on the UI.Composition team (more details can be found in the [Islandwood Xaml and Composition doc](https://microsoft.sharepoint.com/teams/Islandwood/_layouts/15/guestaccess.aspx?guestaccesstoken=BQJXeWeShwOLdSzmOp5fjWnSPO%2bz26ff8MuE1uDCjBc%3d&docid=2_1a51bde0db9d44b0ba1fe29f87c8722d5&rev=1)). The key takeaway is that we should continue to build our [Core Animation](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html) CALayer implementations *on top of Xaml*, so we can support the animations required by Core Animation and UIKit, while also accommodating the specific needs of the various CALayer and UIKit control types and their API usage patterns. and their API usage patterns.

***UIKit and Core Animation are tightly coupled from an app development perspective***, so it’s critical that we provide a feature-rich CALayer and UIView implementation to app developers, regardless of the types and hierarchies of UIViews and CALayers that are being used.

## Design Goals and Requirements

### Support all required UI scenarios in ported WinObjC applications and middleware.

We strive for low-cost ports of apps and middleware - reducing as much friction as possible - particularly in middleware ports where we *ideally* don’t have to make changes to the middleware codebase.

### Build WinObjC’s UIKit and Core Animation implementations on Xaml.

Build full-featured, performant, and deterministic Core Animation and UIKit implementations by *leveraging the UWP platform* *as much as possible*, thus reducing the amount and complexity of WinObjC code.

### Allow for seamless integration with *new* Xaml UI in ported WinObjC applications.

It should be easy for app developers to add new Xaml UI and controls to their apps, use them in their UIViewControllers, lay them out as they do any other UIView, etc.

### Deliver support for a Xib->Xaml conversion/development flow.

This is a *longer-term goal*, but the design laid out in this document must accommodate it.

## Non-Goals

### Provide *default* UIKit controls to non-WinObjC (C#, C++/CX, etc.) UWP applications

Although it may initially seem compelling for us to ship a “UIKit.AutoLayoutPanel” to C# applications, such scenarios are not currently in-scope for this project. ***WinObjC should remain focused on delivering a full-featured iOS application and middleware porting solution, rather than attempting to also fill broad UWP platform gaps.*** The design laid out in this document does not preclude us from adding such support in the future, but it’s *unlikely* that it would be desirable to ship Xaml-backed in-box UIKit controls for use in apps which otherwise would not incur a dependency on the WinObjC runtime (apps which don’t program against UIKit, etc.).

## Synopsis

* iOS app code frequently takes part in ***dynamic and cooperative UI layout***; failure to support this would result in *far more* UI-porting friction than is currently required when porting apps and middleware to WinObjC.
* UIKit views are built on Core Animation layers, and ***most layout and rendering work happens within the Core Animation layer that backs each UIKit view.***
* WinObjC does not *directly* leverage Xaml’s UI layout for the layout of its UIKit views or its Core Animation layers; instead, ***all UI is placed at (0, 0) and is positioned/laid-out via RenderTransforms***.
* Many Core Animation layer (and therefore UIKit view) properties are *optionally* implicitly animated. ***All animations are performed through Xaml StoryBoards***; and the *same* RenderTransforms that are used in explicit layer positioning are used for their respective animations.
* WinObjC’s UIKit controls leverage Xaml *as much as possible*, but ***this is a ‘sliding scale’ which is balanced on a per-control basis***.
* ***The new design*** ***maintains all existing UIKit/Core Animation support***, yet also:
  1. Streamlines the usage of arbitrary Xaml FrameworkElements by [allowing them to be positioned just as any other UIView in the application](#_Arbitrary_Xaml_Element).
  2. [Supports rendering over Xaml-backed UIKit controls; adding adornments, etc.](#_Rendering)
  3. [Supports adding subviews and sublayers to Xaml-backed UIKit controls.](#_SubLayer_Management)
  4. Separates out the complex details of our Core Animation composition layer into [distinct and manageable subcomponents](#_CACompositor) – improving maintainability.
  5. [Defines a concrete relationship between *every* UIView, its root CALayer, and its backing Xaml FrameworkElement.](#_Xaml-Backed_UIViews)
  6. Streamlines the UIElement tree by [removing unnecessary UIElements](#_Layer_Modes)
  7. Simplifies app rendering by removing our two custom Panel implementations; thus eliminating the need to take part in Arrange and Measure passes.
  8. Paves the way for *eventual* Xaml-markup-driven UIElement trees in ported WinObjC apps, because we no longer require re-parenting Xaml FrameworkElements that are positioned by Core Animation.
  9. Moves the rest of the Core Animation composition layer over to C++, to benefit from modern programming constructs such as RAII, external ref-counting (shared\_ptr), etc.

# Technical Prerequisites

Before diving into the specifics of our layout and composition strategy, we must provide background into Core Animation and its close relationship to UIKit on iOS, as well as how they are currently implemented in WinObjC.

## iOS - Core Animation Overview

Core Animation provides the fundamental support for rendering and animating visual content in iOS applications. Core Animation is not a replacement for UIKit and its UIViews; rather, ***UIKit is built directly on top of Core Animation, and applications frequently take advantage of this architecture.***

Basic Core Animation highlights are provided below, but it may also be beneficial to read through the Apple’s [Core Animation Programming Guide](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html) for more details.

### CALayer

[CALayers](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/) are the basis for all rendering, positioning and animation of visual content in iOS apps. CALayers manage information about their positioning, geometry, content, etc.

However, unlike UIViews, CALayers do not *define* their own appearance or content. CALayer content is provided ***either*** by the layer’s [CALayerDelegate](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/) (*usually* a UIView) drawing itself via CoreGraphics, ***or*** by explicitly setting the CALayer’s content to a fixed bitmap image (*also* usually performed by an owning UIView).

### CAAnimation

CALayer animations can be performed either implicitly or explicitly. By default, setting many CALayer properties (bounds, positioning, opacity, etc.) will result in implicit animations being run for those property changes.

Additionally, the app developer can explicitly create one or more animations to run simultaneously on the CALayer via distinct [CAAnimations](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CAAnimation_class/).

### CATransaction

All CALayer property changes, hierarchy changes, animations, etc. are batched up into CATransaction groups. By default, any changes made to Core Animation objects in a given pass of the display update cycle are queued up within a ‘root’ CATransaction, which is committed and processed asynchronously after *that* display update cycle completes.

CATransaction nesting is also supported, wherein the app developer can nest and optionally commit sub-CATransactions as needed.

### UIView Interop

[UIViews](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/) (the basis of all UI controls) on iOS are *always* backed by a single ‘root’ CALayer. Each owning UIView sets itself as its backing CALayer’s [delegate](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/), allowing the UIView to control all rendering and layout for its backing layer.

Custom UIViews can declare the type of their backing CALayer by implementing the static [layerClass](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/) method. By default, most UIViews are backed by the base CALayer class, but specialized CALayer types are provided by Core Animation (CATextLayer, CAShapeLayer, etc.), and app developers can also create custom CALayer implementations for use by their custom UIViews.

It’s standard practice for app developers to build custom/derived UIViews, UIControls, UIButtons, etc. which override and augment their base class’ default layout and rendering behavior.

### SubView/SubLayer Management

CALayers and UIViews are both organized into hierarchies via their respective hierarchy-management APIs ([addSublayer](https://developer.apple.com/library/mac/documentation/GraphicsImaging/Reference/CALayer_class/), [addSubview](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/), etc.). By default, the hierarchy of a set of CALayers that are backing a set of UIViews mirrors the corresponding UIView hierarchy. However, the CALayer hierarchy can be *further* extended beyond the UIView hierarchy.

For example, the first child UIView below is backed by a CALayer which contains *an additional* child CALayer. The additional sub-CALayer may have been added by its owning UIView, or added explicitly by the app developer (by retrieving the UIView’s backing CALayer via its [layer property](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/), and adding a child CALayer directly to it).



### View/Layer Properties

Changes to UIView properties (such as its [frame](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/), [bounds](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/), and [center](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/)) are processed in UIView, but they are ultimately passed down as property changes to the UIView’s backing CALayer to act on those changes (to animate/update visual state, etc.).

It’s common for app developers to *either* work directly on their UIViews, *and/or* to work directly with their UIViews’ backing CALayers (to add sublayers, set properties or animate the layers directly, etc.).

### Animations

By default, many CALayer property changes are implicitly animated. However, when attached to a UIView, those implicit CALayer animations are disabled *unless* they’re modified within a [UIView animation block](https://developer.apple.com/library/ios/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/AnimatingViews/AnimatingViews.html).

More details on animation (including which properties can be animated on CALayers and UIViews) can be found in the [Animating Views](https://developer.apple.com/library/ios/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/AnimatingViews/AnimatingViews.html) documentation.

### Layout

### Queued CALayer layout passes are triggered either explicitly (by calls to [setNeedsLayout](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/)), or implicitly by a CALayer in response to its state (bounds, hierarchy, etc.) changes. Additionally, UIViews often make explicit calls to [setNeedsLayout](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/), in response to UX input, etc. UIView *also* exposes a [setNeedsLayout](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/) method to app developers and derived UIViews, which is a simple pass-through to its backing CALayer.

During each pass of the display update cycle, Core Animation calls the [layoutSublayers](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/) method on any layer that was previously marked as needing to re-layout (via a call to [setNeedsLayout](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/)**)**.

The default implementation of CALayer’s [layoutSublayers](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/) method simply calls out to [layoutSublayersOfLayer](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/index.html) on its CALayerDelegate (which is *usually* its owning UIView).

This is where UIView hooks in and takes part of the layout pass, at which point:

* [viewWillLayoutSubviews](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/) is called on the UIView’s UIViewController.
* Autolayout constraints are recalculated and reapplied.
* [layoutSubviews](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/) is called on the UIView to layout all of its children.
* [viewDidLayoutSubviews](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/) is called on the UIView’s UIViewController.

Additionally, *forced* CALayer/UIView layout passes can be made by calling [layoutIfNeeded](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/index.html) on either CALayer or UIView (the latter simply calls into its backing CALayer’s layoutIfNeeded method).

***The dynamic layout model provided by Core Animation does not align very well with Xaml’s Measure/Arrange pattern;*** Core Animation provides a much more cooperative architecture in which each CALayer/UIView is more deeply integrated into the layout process. Further, iOS app developers frequently take part in this dynamic layout model.

***[TODO: Add clarification, change phrasing based on spec review feedback]***

* *We currently own layout all-up in order to support explicit positioning by CoreAnimation, UIKit, AutoLayout etc.*
  + *Moving to measure/arrange may be possible, but is it desirable? What is the value-add?*
  + *Is it worth further destabilization to switch to measure/arrange?*
* *We’re placing everything at 0,0 and ‘positioning’ via render transforms. Is Xaml’s work lessened or burdened by this approach? We know we need apps and derived controls to take part in dynamic layout; does that help Xaml in any way or is Xaml also doing more work than necessary?*
* *Would we position via Measure(), or via Arrange()?*
* *Don’t UIElements cache state after their Measure calls? If we just call Arrange, will that break things?*
* *Explicit positioning via Measure/Arrange is not very well documented (aside from some UWP documentation on Panels). Is it fully supported?*
* *We’ve seen several bugs our CALayerXaml usage around forgetting to call Measure/Arrange on all children. We’ve also seen a few bugs in Xaml around failing to mark UIElements as ‘dirty’ when within custom panels. Would we be asking for headaches by more formally moving over to measure/arrange?*
* *We need to animate all positioning.*
  + *How would animations work for positioning, resizing, etc. if we were using measure/arrange?*
* *Render transform ordering; how to achieve the same via measure/arrange may be possible, but what is the benefit in doing so?*
* *I think we’d need to keep track of our own position offsets, etc. based on render transforms. Is it worth doing so?*
* *Measure/arrange requires subclassing Xaml elements, so we wouldn’t be able to position arbitrary elements without placing them in an extra UIElement.*

### Rendering

Similar to layout, Core Animation display/render update passes are triggered on CALayers either explicitly (by calls to [setNeedsDisplay](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/index.html)), or implicitly by a CALayer in response to its bounds changes, *unless* [needsDisplayOnBoundsChange](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/index.html) is set to NO (which is the default setting).

CALayer-rendered content is provided by ***one*** of the following methods:

* Directly setting the CALayer’s bitmap image (via its [contents](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/index.html) property).
* Assigning a delegate to the CALayer, thus allowing the delegate to draw or provide the layer’s content (this is the pattern used by UIViews).
* Implementing a custom CALayer subclass, and override one of its drawing methods to explicitly provide the layer contents.

Calling [setNeedsDisplay](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/index.html) on a CALayer or on its owning UIView *usually* results in the CALayer calling *either* the [displayLayer](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/index.html) *or* the [drawLayer:inContext](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/index.html) method of its CALayerDelegate - at which point the existing content within the CALayer is removed, in order to make way for the new incoming content.

#### contents property

If the layer already has explicit image contents set, there’s no additional work needed; that content is just reused rather than asking the CALayer’s delegate to render or provide new content.

#### displayLayer method

[displayLayer](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/index.html) is CALayer’s *first* attempt at obtaining its contents if it doesn’t already have them. If implemented by its CALayerDelegate, [displayLayer](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/index.html) is called early in the render update cycle.

[displayLayer](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/index.html) gives the CALayerDelegate a chance to set the CALayer’s [contents](https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CALayer_class/index.html) property directly; thus avoiding a call to its [drawLayer:inContext](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/index.html) method.

#### drawLayer:inContext method

If [displayLayer](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/index.html) is *not* implemented by the CALayer’s delegate, [drawLayer:inContext](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/index.html) is called on the delegate instead. The CALayerDelegate (*usually* the CALayer’s owning UIView) then renders its contents to the CoreGraphics context that is passed into the [drawLayer:inContext](https://developer.apple.com/library/ios/documentation/QuartzCore/Reference/CALayerDelegate_protocol/index.html) method.

***The dynamic rendering model provided by Core Animation does not align very well with Xaml’s Measure/Arrange pattern;*** Core Animation provides a much more cooperative architecture in which each CALayer/UIView is more deeply integrated into the rendering process.

***[TODO: Add clarification, change phrasing based on spec review feedback]***

* *‘Arrange’ doesn’t sound like ‘Render’; are they analogous?*
* *What’s WinRT’s comparison to a UIView and its subclasses sharing the same CGContext to render into?*
* *I think the best we can do is draw on top of Xaml controls.* 
  + *I also think this is likely ‘good enough’.*

WinObjC - Core Animation Implementation

This section covers the technical details of ***WinObjC’s current implementation***of the Core Animation features that were outlined above. For details on the new architecture; skip to the [design section below](#_Design).

Unsurprisingly, the overall implementation of WinObjC’s UIViewController/UIView/CALayer/CoreGraphics interop closely mimics the reference platform. WinObjC strives for a high level of ported app compatibility at the UI layer, so the team produced a relatively straightforward implementation of the same architecture that’s provided on iOS (albeit with known gaps and stubbed-out features).

***Given the symbiotic relationship between CALayers and UIViews, if we were to deviate very far from the iOS architecture, apps and middleware would not port well; large portions of their UI and ViewControllers would require a ~full rewrite.***

As such, the most relevant portion of WinObjC’s Core Animation design (with respect to this document) is its abstraction layer ***below*** Core Animation (aka the ‘CACompositor’), which is where the actual implementation of WinObjC’s CALayer rendering, positioning, and animations are ***implemented*** ***on top of Xaml***.

### CACompositor

The WinObjC-specific [CACompositor interface](https://github.com/Microsoft/WinObjC/blob/1473d9561ca948d2b845fd588dfdfcb83b4f84a0/Frameworks/include/CACompositor.h) is used across several components in the WinObjC stack for various needs related to UI rendering and layout.

CACompsitor’s responsibilities include:

#### Core Animation Support

Support for CALayer property updates, layer hierarchy changes, animations, etc. This portion of the CACompositor API is used solely by Core Animation (aside from ‘setNodeTopMost’, which is called by UIApplication and UIWindow to move a ‘root’ layer to the top of the stack).

Here’s a *simplified* view of how WinObjC’s Core Animation library leverages CACompositor:



##### DisplayTransaction

DisplayTransaction is the interop layer between all CATransactions and Xaml. Queued DisplayTransactions are executed in the following order during a given UI update cycle:

1. animations
2. sublayer hierarchy management
3. layer property changes

##### DisplayNode

DisplayNode is the interop layer between CALayer and Xaml.

There is a single concrete form of DisplayNode:

* **DisplayNodeXaml** subclass which serves as a proxy between each CALayer and its Xaml representation (aka [CALayerXaml](#_CALayerXaml)).

##### DisplayTexture

DisplayTexture is the interop layer between CALayer and its rendered contents. During a render pass, CALayer obtains a DisplayTexture from the CACompositor, which it assigns to its backing DisplayNode’s content.

DisplayTextures come in three forms;

* **DisplayTexture** base class for rendering CALayer bitmap contents.
* **GenericControlXaml** subclass which ‘renders’ a ‘wrapped’ Xaml FrameworkElement *within* a CALayer as its ‘contents’.
* **DisplayTextureText** subclass which ‘renders’ text via a contained Xaml TextBlock as its ‘contents’.

##### DisplayAnimation

DisplayAnimation is the interop layer between CAAnimations and Xaml.

There are two forms of DisplayAnimations:

* **DisplayAnimationBasic** subclass for animating CALayer property changes such as bounds, position, etc. on the CALayer’s Xaml representation.
* **DisplayAnimationTransition** subclass for animating view navigation ‘swooshes’ and ‘flips’ on the CALayer’s Xaml representation.

#### Screen/Device Settings

Sets/retrieves screen size, device size, screen scale, whether or not it’s in ‘tablet mode’, etc. Not really tied to Core Animation, and should probably be moved out of CACompositor. Callers include; UIApplication, UIWindow, UITouch, CACompositor, UIDevice, and UIGraphicsFunctions (UIKit’s CoreGraphics interop layer).

**Note:** Recommend breaking this component out into a self-contained API.

#### Bitmap Management

Core Animation/CoreGraphics integration point; Core Animation calls this API to create Xaml WriteableBitmaps which are used to set up CoreGraphics contexts to which CALayers and UIViews can render.

Also provides conversion functions between CGImage/Bitmaps.

**Note:** Recommend breaking this component out into a self-contained API.

#### CADisplayLink

Implementation of the CADisplayLink API is spread across the CACompositor code.

**Note:** Recommend breaking this component out into a self-contained API.

### CALayerXaml

CALayerXaml, which ***derives from Windows::UI::Xaml::Controls::Panel***, is WinObjC’s current representation of a CALayer in Xaml. *All UI* in a given WinObjC application is ultimately comprised of CALayerXaml Panels. WinObjC’s CALayers (by way of CACompositor) ultimately control the properties, animations, etc. of their backing CALayerXaml elements.

Below is a ***simplified*** *view* of how WinObjC’s CALayers and CATextLayers are built on top of CALayerXaml objects:



#### CALayerXaml Modes

In order to minimize objects in the Xaml UIElement tree, there are *several* *forms* that CALayerXaml can dynamically morph into as needed:

##### No Contents

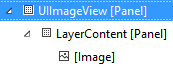
If a CALayer lacks *any* contents, its UIElement tree is as such:



Its rectangle serves as the background of the panel for visual and pointer-input purposes; we need a separate rectangle because the actual CALayerXaml panel is 1x1, and as such, it wouldn’t serve well as the layer’s background or as a surface for receiving pointer input.  **Note:** that the redesign in section 3 below removes the 1x1 CALayerXaml panels (as well as this Rectangle).

##### Rendering Images

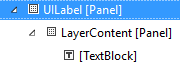
If a CALayer (or its owning UIView) renders to a bitmap via CoreGraphics, the CALayer’s UIElement tree is as such:



Notice that the background rectangle is no longer needed, because the layer contents obscure any background that may have existed.

##### Hosting Xaml FrameworkElements (including CATextLayer/UILabel text rendering)

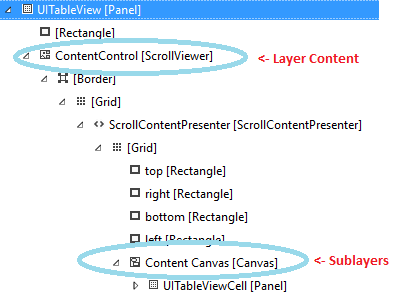
If a CALayer (or its owning UIView) hosts/wraps a Xaml FrameworkElement (TextBlock, TextBox, Button, etc.), the CALayer’s UIElement tree is as such:



Again; notice that the background rectangle is no longer needed because the Xaml FrameworkElement obscures any background that may have existed.

##### Hosting Xaml Content Controls (Xaml ScrollViewer)

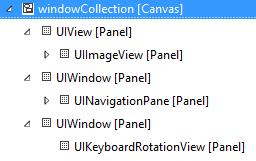
A specialized form of the above Xaml-hosted model, which ‘splits’ the CALayerXaml into *two* distinct parts; 1) the content of the CALayerXaml (its hosted FrameworkElement), and 2) its Canvas for adding the CALayer’s sublayers:



##### ‘Root Node’

There’s a specialized form of top-level-window CALayerXamls (the layers backing UIWindows). These ‘root nodes’ don’t contain a background, and therefore don’t accept pointer input.

These are typically owned by UIWindow, although there are currently ***three*** top-level/root nodes in WinObjC apps today:



**Note:** Recommended modifications to our UIWindow structure are somewhat unrelated to this document, but they’re called out in [a separate section below](#_UIWindow_Notes).

### CALayer/CALayerXaml Properties

The following CALayer properties are *currently* implemented on CALayerXaml, and have sufficed for most ported applications to date (even with the absence of border support):

#### ContentsCenter (Rect)

The rectangle that defines how the layer contents are scaled if the layer’s contents are resized.

* Only applied if the CALayerXaml contains a LayerContent object backed by a Xaml image, and ***is not*** applied to hosted Xaml FrameworkElements.
* Transforms into a 3x3 nine-grid that defines how the **contents** are stretched (if stretching is performed).

#### AnchorPoint (x, y) (0.0 🡪 1.0, 0.0 🡪 1.0) – Optionally Animated

All geometric manipulations to the layer (rotation, positioning, etc.) occur about the specified point within the CALayer; it defaults to (0.5, 0.5) (its center point).

* Applied to the CALayerXaml panel as part of its UIElement.RenderTransform TransformGroup; see [positioning and layout](#_Layout) below for more details.

#### Position (x, y) - Optionally Animated

The layer’s position in its parent layer’s coordinate space.

* Applied to the CALayerXaml panel as part of its UIElement.RenderTransform TransformGroup; see [positioning and layout](#_Layout) below for more details.

#### Bounds.Origin (x, y) - Optionally Animated

Specifies a translation that is applied to the content of the layer and to any of its sublayers. The clipping (’masksToBounds’) rectangle is ***not*** translated by this value. bounds.origin can be thought of as the “scroll offset” of a layer and is useful for pan-able scrolling views.

* Applied to the CALayerXaml panel as part of its UIElement.RenderTransform TransformGroup; see [positioning and layout](#_Layout) below for more details.
* The layer’s clipping behavior is updated as needed, thus ensuring that it’s not affected by this value.

#### Bounds.Size (width, height) - Optionally Animated

Specifies the dimensions of the layer. If the node has clipping enabled, its **contents** (and its **sublayers**) will be clipped to this size.

* Applied to the CALayerXaml panel as part of its UIElement.RenderTransform TransformGroup; see [positioning and layout](#_Layout) below for more details.
* Animated via a dependent (upon the UI thread) animation.
  + UIElement sizes are not *usually* animatable, so this likely leads to degraded perf.

#### Opacity - Optionally Animated

A value specifying the opacity of the contents and its sublayers.

* Set directly on the CALayerXaml.
* Animated via (UIElement.Opacity).

#### Hidden

A Boolean indicating whether the layer is displayed.

#### MaskToBounds

Indicates whether or not sublayers are clipped to the layer’s bounds.

* Re-calculated as the UIElement.Clip rect on the CALayerXaml as its bounds, etc. change.
* Not animated, although it’s apparently animated on the reference platform.

#### Transform (4x4 Transform) - Optionally Animated

A general **4x4** matrix transform that is applied relative to the **anchorPoint** of the layer. Applies both to contents and children.

* Applied to the CALayerXaml panel as part of its UIElement.RenderTransform TransformGroup; see [positioning and layout](#_Layout) below for more details.

#### ContentsOrientation

This is a non-standard property that we expose off of CALayer as though it were part of the public interface. It appears that it’s a helper on another non-standard UIImage API (UIImageSetLayerContents), in order to assign an image to a CALayer while simultaneously applying rotation.

* Orientation is converted into a transform.rotation value that is set on the CALayerXaml.

**Note:** We should determine whether or not we should delete this, or at the very least, make these APIs private.

#### ZIndex

This is a non-standard property used for Z-order positioning of our UIWindows.

**Note:** This should just happen in UIWindow.mm; it doesn’t belong within the CACompositor as it’s not a true CALayer property.

#### ContentsGravity (enum)

Defines how the contents are positioned and sized within the layer if the dimensions of the **contents** and **bounds.size** do not match. E.g. Centered, TopLeft, Left, Fill, FillAspect, etc.

* Applied to the Image within the CALayerXaml’s LayerContent by explicitly setting the width/height and Image’s stretch behavior.
* Not currently applied to hosted Xaml FrameworkElements.
* Not animated, although it’s apparently animated on the reference platform.

#### BackgroundColor

The background color of the CALayer.

* Added as a Rectangle on the CALayerXaml only as needed.
* Not animated, although it’s apparently animated on the reference platform.

### CALayer/CALayerXaml Property Management

The CALayer properties described above are set on, and are retrieved from, DisplayNodeXaml objects (via the DisplayNode interface).

DisplayNodeXaml is responsible for forwarding named property changes to the CALayerXaml instance.

For example;

void DisplayNodeXaml::UpdateProperty(const char\* name, void\* value) {

NSObject\* newValue = (NSObject\*)value;

if (name == NULL)

return;

if ([NSThread currentThread] != [NSThread mainThread]) {

return;

}

if (strcmp(name, "contentsCenter") == 0) {

CGRect value = [(NSValue\*)newValue CGRectValue];

SetContentsCenter(value.origin.x, value.origin.y, value.size.width, value.size.height);

}

else if (strcmp(name, "anchorPoint") == 0) {

CGPoint value = [(NSValue\*)newValue CGPointValue];

SetProperty(L"anchorPoint.x", value.x);

SetProperty(L"anchorPoint.y", value.y);

The above calls transition over to C++/CX (as the DisplayNode/DisplayNodeXaml implementations span *both* languages), and the properties are *then* translated into corresponding changes on the DisplayNodeXaml’s backing CALayerXaml instance.

Finally, CALayerXaml performs a second-phase property name/value dispatch, at which point the change is applied to the CALayerXaml instance:

void CALayerXaml::Set(String^ propertyName, Object^ value) {

CALayerXaml::s\_animatableProperties[propertyName]->Set(this, value);

}

...

std::map<String^, CALayerXaml::AnimatableProperty^> CALayerXaml::s\_animatableProperties = {

...

{ "anchorPoint.x",

...

ref new CALayerXaml::ApplyTransformFunction([](CALayerXaml^ target, Object^ toValue) {

double anchorPointX = (double)toValue;

target->m\_anchorPoint.X = (float)anchorPointX;

if (target->m\_createdTransforms) {

double destX = -target->m\_size.Width \* ((double)toValue);

((TranslateTransform^)((TransformGroup^)target->RenderTransform)->Children->GetAt(0))->X = destX;

}

else {

target->\_CalcTransforms();

}

}),

ref new CALayerXaml::GetCurrentValue([](CALayerXaml^ target) -> Object^ { return (double)target->m\_anchorPoint.X; })) },

### CALayer/CALayerXaml Animation Management

#### Property-Change Animations

CALayer property-change animations follow a similar pattern to that of CALayer property changes. However, animations are performed via Xaml Storyboards (rather than by setting properties directly on the CALayerXaml elements), and are therefore kicked off by CACompositor ***before*** the actual property changes are applied to the CALayerXaml elements (in order to avoid ‘glitchy' animations).

From CALayer’s perspective, property changes and their respective animations are *frequently* performed in unison (*1.* kick off animation, then *2.* set the property) as part of the same CATransaction. However, CALayer also frequently sets properties directly (with no animation) if animations are not required (which is common for changes to UIView properties when not performed within an animation block).

In WinObjC’s current implementation, with the exception of opacity and gravity, CALayer property-change animations are applied to the CALayerXaml’s RenderTransforms to affect their origin, anchor, position, etc.

See [positioning and layout](#_Layout) below for more details.

#### Transition Animations

In addition to property-change animations, two special forms of ‘transition animations’ also exists, in order to provide CATransition’s ‘flip’ and ‘left-right’ navigation animations. Such animations are built upon *most* of the same CALayerXaml Storyboard management infrastructure, but they follow a much more complicated sequence of events;

1. Render the given layer to a bitmap.
2. Cover the given layer with its rendered bitmap.
3. Move the given layer off-screen, and replace it with a new layer.
4. Animate the replacement layer back into view, completely obscuring the rendered bitmap ‘layer’.
5. Delete the temporary rendered bitmap ‘layer’.

**Note:** Transition animations have historically been susceptible to glitches as we’ve made tweaks to other layers in the stack (our move off of fibers, for example). Ideally we could remove these in favor of Xaml page navigation or some other form of built-in Xaml animations.

### Positioning and Layout

As described in the [CALayer Layout](#_Layout_1) section above, CALayer/UIView positioning and layout is extremely ***dynamic*** and ***cooperative*** in nature, particularly when compared to Xaml’s highly-declarative model.

***Xaml-markup-based layout does not align well with ported iOS apps***, which often intimately control the positioning and layout of their custom CALayers and UIViews at runtime. Further, although xib files do contain *initial* UIView positions and sizes, those positions often change at runtime based upon app code and/or AutoLayout constraints being honored by the framework.

***Xaml’s dynamic UIElement positioning model is not flexible enough for our needs.*** WinObjC needs to position arbitrary UIElements dynamically at runtime as required by the ported iOS app code. This is difficult functionality to build on top of Xaml’s Measure/Arrange model. For example; with the exception of Xaml’s Canvas, Xaml does not provide a way to say, “put this UIElement at position 10, 10 within its parent”.

Given the above limitations with Xaml’s layout model, as well as the need to maintain CALayer’s explicit transform ordering (more on that below), ***WinObjC’s CACompositor implements all of its CALayer positioning and layout via UIElement RenderTransforms***. All CALayerXaml objects are positioned at (0, 0) within their parent, and are then *transformed* into their visual position, etc. as required by the ported app code.

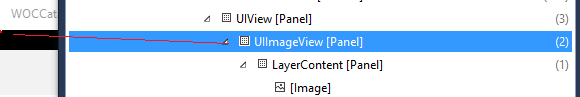
**Note:** Interestingly, a similar approach is used by the Office team in order to take full control of their UIElement layout. Are there any details we can provide to the Xaml team that might make explicit Xaml layout & positioning easier for app developers?

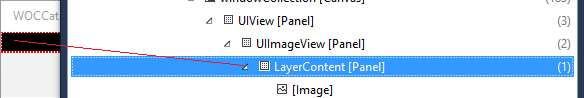
#### Xaml Measure and Arrange Interop

Given that CALayerXaml and its sometimes-contained LayerContent are custom Xaml Panels, they *must* take part in Xaml layout by implementing MeasureOverride and ArrangeOverride, in addition to the explicit RenderTransform positioning and layout described above.

This leads to quirky UIElement tree layout in WinObjC apps; the *actual* LayerContent, background, etc. is sized and positioned correctly, but each owning CALayerXaml instance is an extra 1x1 UIElement. The need for the 1x1 sizing is not currently understood, but if the CALayer’s *actual* size is returned from Measure/Arrange, incorrect layout is produced. The 1x1 panels lead to confusion when debugging an app’s UIElement tree via Visual Studio’s live UIElement tree viewer.

For example; the CALayerXaml below is a 1x1 UIElement, while its *contained* LayerContent is of the correct dimensions:





**Note:** The design outlined below removes the intermediate CALayerXaml panels altogether, so this will no longer be an issue.

#### Layout Updates

WinObjC is ultimately in control of UI layout updates, ***as required for app compatibility***. Custom app code (custom UIViews, CALayers, etc.) ***must*** be able to call setNeedsLayout to trigger a new layout pass, after which their code ***expects*** to be called with the relevant layout callbacks ([viewWillLayoutSubviews](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/), [layoutSubviews](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/), [layoutSublayers](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/), etc.) to their respective classes (UIViewController, UIView, CALayer, etc.)., etc.) to their respective classes (UIViewController, UIView, CALayer, etc.).

This poses a challenge for integration with Xaml UIElements and layout; see the design section regarding layout below for more details.

#### RenderTransform Ordering

CALayerXaml’s RenderTransforms are created in a specific order, to align with the CALayer properties that they are used for.

For example, here’s a concise snippet from the new prototype of the CACompositor refactor outlined in this doc:

// The anchor value modifies how the rest of the transforms are applied to this layer

// Ideally we'd just set UIElement::RenderTransformOrigin, but that doesn't apply to TranslateTransforms

auto sizeAnchorTransform = ref new TranslateTransform(); // anchorpoint

sizeAnchorTransform->X = -width \* GetAnchorPointX(element);

sizeAnchorTransform->Y = -height \* GetAnchorPointY(element);

// Set up the origin transform

auto originTransform = ref new TranslateTransform(); // origin

originTransform->X = -GetOriginX(element);

originTransform->Y = -GetOriginY(element);

// Nested transform group for rotation, scale and translation transforms

auto contentTransform = ref new TransformGroup();

contentTransform->Children->Append(ref new RotateTransform()); // transform.rotation

contentTransform->Children->Append(ref new ScaleTransform()); // transform.scale

contentTransform->Children->Append(ref new TranslateTransform()); // transform.translation

// Positioning transform

auto positionTransform = ref new TranslateTransform(); // position

positionTransform->X = GetPositionX(element);

positionTransform->Y = GetPositionY(element);

auto layerTransforms = ref new TransformGroup();

layerTransforms->Children->Append(sizeAnchorTransform);

layerTransforms->Children->Append(originTransform);

layerTransforms->Children->Append(contentTransform);

layerTransforms->Children->Append(positionTransform);

element->RenderTransform = layerTransforms;

***The ordering above matches the iOS CALayer functionality that we’re emulating.*** With the necessary RenderTransforms applied to the CALayerXaml object in the correct order, property changes and animations can be applied as needed – app code ‘just works’.

#### AutoLayout

UIKit’s [AutoLayout feature](https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/AutolayoutPG/) dynamically calculates the sizes and positions of all UIViews in a scene, according to constraints placed on those UIViews. WinObjC’s AutoLayout implementation leverages the OSS libcassowary constraint-solving engine during the UIView layout pass.

During each UIView layout update cycle, WinObjC’s AutoLayout implementation calculates desired UIView sizes and positions according to the app’s declared AutoLayout constraints. After the desired sizes and positions are determined by AutoLayout, the AutoLayout code simply calls the *same* UIView/CALayer APIs (setFrame, etc.) in order to position and size each UIView as needed.

See the [design section regarding Autolayout](#_Autolayout) below for more details on how this will continue to work post-refactor.

### Layer Hierarchy Management

Given that all CALayers are backed by CALayerXaml instances (which derive from Xaml Panel), sublayers are *usually* added directly to the CALayerXaml’s Children collection starting at element 0. However, if the parent CALayerXaml instance contains any LayerContent, all of its children are added starting at element 1 (until its LayerContent is removed – at which point all of the children are shifted accordingly). There is an *additional* special-case for UIScrollView’s backing CALayerXaml, because its children are actually added into a separate UIElement.

### Rendering

[TODO: Do we need to go into more details on how content is renderedpre-refactor?]

### CALayer Implementations

[TODO: Do we need a brief overview of which CALayers we have implemented today?]

# Design

## Overview

The current CACompositor architecture meets our needs in many areas, but it poses some technical challenges, particularly around our ability to integrate more closely with Xaml and WinRT. The design outlined in this section *primarily* involves refactoring WinObjC’s existing Core Animation (CACompositor) implementation to maintain existing Core Animation support, while also addressing these issues:

* CACompositor is only able to position CALayerXaml objects. CALayerXaml essentially positions *itself*, and it requires a lot of internal state to do so.
* CALayerXaml requires distinct ownership of each Xaml element, which directly inhibits one of ourprimary goals. Xaml markup should be able to declare/build the UIElement tree for us, but that’s not possible on the current design.
* Since Xaml-backed CALayerXaml instances *contain* UIElements as their ‘layer contents’, we:
  + have many more UIElements in the tree than needed.
  + can’t support custom rendering (drawRect:, etc.) over the Xaml-backed controls.
  + can't support adding subviews/sublayers to Xaml-backed controls.
* The exceedingly dynamic nature of CALayerXaml is a bit *too loose*; more structure would lend well to maintainability and simplification of the codebase.
  + CACompositor in its current form is not very approachable, and has been ~difficult to maintain; we can improve this.
* We must manage *two* Xaml elements within the CALayer and UIView code for Xaml-backed controls; we have the CALayerXaml element in UIView for input purposes, as well as the *actual* Xaml control implementation in the derived UIView class (UISlider’s Xaml Slider for example). ***Ideally we’d only have a single root Xaml element per UIView/CALayer.***
* The 1x1 CALayerXaml Panels scattered throughout the UIElement tree are confusing.
  + Additionally, since the CALayerXaml are 1x1, we must often insert *additional* Rectangles into the UIElement to receive input, even if no background was set on the layer.
* All of the CACompositor code is ***currently contained in only ~3 .cpp/.mm files***; we should break them out into separate files/classes, to more clearly delineate responsibilities.
  + We should also take this opportunity to move to RAII for memory management across CACompositor, rather than requiring Core Animation to call down to CACompositor to free DisplayNodes, DisplayAnimations, etc.
* We see a lot of ‘tearing’ (see below) around some of our UIView-wrapped Xaml elements; the cause is not yet understood, but ***it is fixed with the refactor*** outlined in section 3 below.



In order to address the above issues, we will shift CACompositor/Core Animation/UIKit to an architecture which allows us to ***treat any Xaml FrameworkElement as a CALayer (and therefore as a UIView)***, rather than requiring all CALayers to be backed by the sole CALayerXaml type.

* CACompositor will be refactored to ***support the positioning and animation of arbitrary Xaml FrameworkElements***.
* Xaml FrameworkElements will be able to ***optionally*** ***extend themselves to support the addition of sublayers***. CACompositor will dynamically leverage custom sublayer support *if it exists* on a FrameworkElement; otherwise, CACompositor will gracefully fail the sublayer management operation (and will log telemetry/debug output).
  + For example; our *initial* Xaml UIKit.Button implementation won’t support adding subviews, nor will it allow for custom drawRect calls, but we can easily add those features later if needed.
* Xaml FrameworkElements will be able to ***optionally*** ***extend themselves to support custom rendering*** (drawRect:, etc.). CACompositor will dynamically leverage custom rendering support *if it exists* on a FrameworkElement; otherwise, CACompositor will gracefully fail the render operation (and will log telemetry/debug output).

## CACompositor

The following static structure diagram outlines the primary components of the new CACompositor architecture:

#### CACompositor

The CACompositor interface has been trimmed down; it now serves merely as a class factory for the various CACompositor interfaces.

#### IDisplayTextureManager

A new distinct interface responsible for creating the backing surfaces used by CALayer/UIView CoreGraphics and CoreText rendering.

#### DisplayTexture

We are down to a *single* form of DisplayTexture that is used for rendering CALayer’s content (if it exists). The ‘xaml’ and ‘text’ DisplayTexture types are no longer needed. ***WinObjC’s CoreGraphics and CoreText implementations render to DisplayTextures***.

DisplayTextures are *currently* backed by a Xaml WriteableBitmap, which is rendered within a Xaml Image that is *optionally* exposed off of each UIKit::Private::Core Animation::ILayer.

**Note:** DisplayTexture’s usage of WriteableBitmap and how it is rendered (via a Xaml Image) will likely change after CoreGraphics is moved to D2D. ***Such a shift will only require changes to DisplayTexture and our UIKit::Private::Core Animation::ILayer implementations***.

#### LayerTransactionManager : ILayerTransactionManager

LayerTransactionManager is a class factory for ILayerTransactions (the proxies backing CATransactions). It also manages and processes the ILayerTransaction/CATransaction stack, which is how CALayer properties, animations, etc. are ultimately applied.

#### LayerTransaction : ILayerTransaction

LayerTransactions are the proxies backing CATransactions. They are responsible for translating CALayer property changes, hierarchy changes, and animations into LayerCoordinator calls.

#### ILayerAnimation

ILayerAnimations are the proxies backing CAAnimations. The only changes here are renames of the interface and its concrete classes (DisplayAnimation\* -> LayerAnimation\*) for the sake of code clarity.

We are also switching to shared\_ptr for RAII usage.

#### LayerProxy : ILayerProxy

ILayerProxy allows CALayer, UIView, and app developers to interact *directly* with its backing Xaml element (to subscribe to events, hook pointer input, etc.). ILayerProxy also provides access to the CALayer’s current CACompositor-managed property values (the CALayer’s position, transform values, dimensions, etc.).

Each time a CALayer is created, it creates a LayerProxy instance with which to manage its backing Xaml FrameworkElement. The LayerProxy also initializes itself with the LayerCoordinator, as discussed further in the next section.

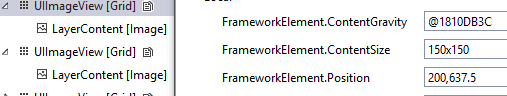
We are also switching to shared\_ptr for RAII usage.

### LayerCoordinator

***LayerCoordinator replaces CALayerXaml*** as our CALayer property-management and layout engine.

CALayerXaml and its associated/internal LayerContent class contained a large amount of internal state that was used for tracking and applying CALayer properties. For example; CALayer’s ‘masks to bounds’ bool was stored as private member in CALayerXaml, and its value was used when CALayerXaml performed any updates to its origin (or any other properties that were affected by the ‘masks to bounds’ setting).

***LayerCoordinator moves to an external property-tracking model for each LayerProxy***, allowing us to treat *any* Xaml FrameworkElement as a CALayer. LayerCoordinator currently leverages AttachedProperties to do so (which is nice when debugging layers in the UIElement tree viewer), but we could easily move these properties into the LayerProxy object if needed (for example, if AttachedProperties become problematic or incur too much of a perf hit). Here’s an example of the previously-CALayerXaml-internal CACompositor state that we’ve moved out to Attached Properties on whatever Xaml FrameworkElement happens to be backing the given LayerProxy:



#### LayerProxy Initialization

When a LayerProxy is constructed, it registers its backing Xaml FrameworkElement with the LayerCoordinator (via a call to LayerCoordinator::InitializeFrameworkElement), in order to configure the FrameworkElement for management by the LayerCoordinator. This initialization step sets up the FrameworkElement’s RenderTransforms, sets its default hit-testability, etc.

#### Layer Modes

Similar to the old CALayerXaml model, LayerCoordinator render LayerProxies in several modes, thus avoiding unnecessary UIElements in the tree.

##### No Contents

If a LayerProxy lacks *any* contents, its UIElement tree is as such:



Notice that there is no longer an extra sub-element until we actually need to render contents or add sublayers.

##### Rendering Images

If a CALayer (or its owning UIView) renders to a bitmap via CoreGraphics, the CALayer’s UIElement tree is as such:



Notice that there is no longer an additional sub-Panel within the UIImageView to host the layer contents.

##### Hosting Xaml FrameworkElements (including UILabel text rendering)

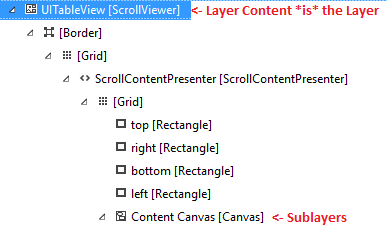
If a CALayer (or its owning UIView) hosts/wraps a Xaml FrameworkElement (TextBlock, TextBox, Button, etc.), the CALayer’s UIElement tree is as such:



In the above example, our UILabel is backed by a UIKit.Label, which consists of a TextBlock *within* a Grid (to achieve vertically-centered text). Note that even though we unfortunately need two UIElements for our Xaml-backed UILabel, it’s still one UIElement less than the CALayerXaml model.

##### Hosting Xaml Content Controls (Xaml ScrollViewer)

A specialized form of the above Xaml-hosted model, which ‘splits’ the LayerProxy into *two* distinct parts; *1)* the content of the LayerProxy (its hosted FrameworkElement), and *2)* its Canvas for adding the CALayer’s sublayers:



Note that this version hosts two fewer UIElements than the CALayerXaml model.

#### Layout

LayerCoordinator’s CALayer layout functionality is *similar* to that of CALayerXaml (described above); it still uses RenderTransforms for layout and positioning, but we are now able to ***manage the properties of arbitrary Xaml FrameworkElements***, rather than only working with CALayerXaml types.

Jump to the below section on UIKit layout for more details on how this affects our wrapped Xaml controls.



#### Animations

Animations closely follow the refactor we’ve done for layout and property changes (see Layout above). We can now animate property changes on *any* Xaml FrameworkElement, rather than just on CALayerXamls. We’ve also pulled the storyboard management out into a distinct StoryboardManager API for code clarity and maintainability.

#### SubLayer Management

LayerCoordinator/LayerProxy are able to manage the sublayers of ‘enlightened’ Xaml FrameworkElements. This ‘enlightenment’ exists in two forms:

##### ILayer::SublayerCanvas (interface property)

If the Xaml FrameworkElement backing the LayerProxy implements [UIKitPrivate::Core Animation::ILayer, we retrieve the value of its SublayerCanvas property. If non-null, we leverage i](#_UIKit::Private::Core_Animation::ILa)t for sublayer management; otherwise, we fail gracefully and trace that the call isn’t supported by the given Xaml FrameworkElement. the given Xaml FrameworkElement.

##### Layer::SublayerCanvasProperty (attached property)

A fallback mechanism for the above; primarily here for back-compat until we can get all of our controls ported to Xaml. Currently only used by UIScrollView, to specify where we should add its sublayers. UIScrollView sets this attached property on its Xaml ScrollViewer, and then calls down to Core Animation to create its backing CALayer (and therefore its backing LayerProxy).



#### Rendering

LayerCoordinator/LayerProxy are able to manage custom rendering for any ‘enlightened’ Xaml FrameworkElements.

When any custom rendering (via drawRect:, setting a CALayer’s contents, etc.) is performed on a CALayer, LayerCoordinator can do so on ***any*** Xaml FrameworkElement which both *a)* implements UIKit::Private::Core Animation::ILayer, and *b)* provides a Xaml Image with which to render the CALayer’s bitmap contents.

If the Xaml FrameworkElement doesn’t implement ILayer, or if it returns nullptr from its LayerContent property, LayerCoordinator fails gracefully and traces that the call isn’t supported by that FrameworkElement.



### UIKit::Private::CoreAnimation::ILayer

ILayer is an interface which allows LayerCoordinator to adapt its functionality based upon the type of Xaml FrameworkElement that’s backing a given LayerProxy instance.

Our default Xaml Layer implementation will implement this interface, as will any of our Xaml-backed UIKit control implementations that need to support custom rendering and/or subviews.

[Windows::Foundation::Metadata::WebHostHidden]

public interface class ILayer {

public:

// Accessor for our Layer content; we create one on demand

property Windows::UI::Xaml::Controls::Image^ LayerContent {

Windows::UI::Xaml::Controls::Image^ get();

}

// Accessor to check for existing Layer content

property bool HasLayerContent {

bool get();

}

// Accessor for our SublayerCanvas; we create one on demand

property Windows::UI::Xaml::Controls::Canvas^ SublayerCanvas {

Windows::UI::Xaml::Controls::Canvas^ get();

}

};

### UIKit::Private::CoreAnimation::Layer

UIKit::Private::Core Animation::Layer is our default Xaml implementation of the ILayer interface, and it backs all of our default CALayer objects and UIViews *that don’t provide their own backing Xaml FrameworkElement*.

Layer creates its *LayerContent* and its *SublayerCanvas* only when requested by the LayerCoordinator, so ***we don’t wastefully create unnecessary UIElements***.

Upon initial construction, ***Layer consists solely of a Xaml Canvas***. We chose a Canvas (rather than a Grid) because it doesn’t clip its children, which is required for CALayer contents that exceed the bounds of the CALayer. Unfortunately, this means that we can’t leverage Grid’s intrinsic/low-cost border. Perhaps should request that the Xaml team adds implicit border support to the base Panel class, or at least to Canvas.

From there, Layer lights up more functionality when requested by LayerCoordinator:

// Accessor for our Layer content

Image^ Layer::LayerContent::get() {

if (!\_content) {

\_content = ref new Image();

\_content->Name = "LayerContent";

Children->InsertAt(0, \_content);

}

return \_content;

}

// Accessor for our Layer content

bool Layer::HasLayerContent::get() {

return \_content != nullptr;

}

// Accessor for our SublayerCanvas

Canvas^ Layer::SublayerCanvas::get() {

if (!\_sublayerCanvas) {

\_sublayerCanvas = ref new Canvas();

\_sublayerCanvas->Name = "Sublayers";

Children->Append(\_sublayerCanvas);

}

return \_sublayerCanvas;

}

#### Layer - No Content

The default mode of our Layer is a simple Xaml Grid; in this mode, we:

* Feed input events through UIKit.
* Leverage the backing Grid’s intrinsic background as needed (*if* the background is set on the CALayer).
* Leverage the Grid’s intrinsic border as needed (*if* the border is set on the CALayer).
  + **Note:** Border support is not yet implemented.



#### Layer – With Content

If the CALayer renders or sets static bitmap contents, LayerCoordinator accesses the relevant ILayer properties to create a Xaml Image on demand. This Xaml Image now renders the CALayer’s contents:



#### Layer – With Sublayer(s)

If the CALayer contains one or more sublayers, LayerCoordinator accesses the relevant ILayer property to create a Sublayer canvas on demand. This Xaml Canvas now contains the CALayer’s sublayers:



#### Layer – With Content and Sublayer(s)

A Layer that contains contents and sublayers. Notice that the sublayers are rendered *in front* of this layer’s content image.



## UIKit

This section provides details and guidance on how WinObjC’s UIKit controls and views are *simultaneously* implemented on Xaml and WinObjCs UIKit/Core Animation implementation.

The design details aim to address (at least at a high level) several of the ‘big rock’ questions such as:

1. [What do we gain by building Xaml-backed controls?](#_Leveraging_Xaml)
2. [Who *ultimately* controls UI layout (UIKit vs. Xaml)?](#_Layout/Autolayout)
3. How are Xaml-backed controls composed?
4. [How will custom rendering (drawRect, etc.) work with Xaml-backed controls?](#_Custom_Rendering)
5. [How can I add subviews to Xaml-backed controls?](#_Subview/Sublayer_Management)
6. [How do UIElement commands (button click, etc.) feed into UIKit?](#_Control_Events)

### Leveraging Xaml

It’s important to reiterate what our goals (and non-goals) are for leveraging Xaml in our UIKit control implementations:

#### Goals

1. Reuse existing Xaml control implementations where possible, thus reducing the amount of code owned and maintained by our team.
2. Keyboard focus and navigation support.
3. Touch/pointer input and hit-testing integration.
4. Accessibility support.

#### Non-Goals

1. Move more of our UIKit code into C++/CX.
2. Provide fancy controls (collectionview, etc.) to non-WinObjC applications.

### UIView

Just as on iOS, all UIViews on our platform are backed by a root CALayer. However, as demonstrated in the Core Animation section above, WinObjC’s CALayers are simply proxies to various Xaml FrameworkElements. This design allows us to easily integrate arbitrary Xaml UI into ported iOS applications by wrapping them within UIView (and therefore CALayer) instances.

#### Xaml-Backed UIViews

WinObjC adds rich support for Xaml integration within apps, by allowing app developers (including ourselves) to wrap arbitrary Xaml UI within UIView instances.  ***Xaml-backed UIViews are a tremendous value-add for the WinObjC platform.*** App developers can instantiate arbitrary Xaml elements (via WinRT projections into ObjectiveC), which they can position within their UIView hierarchy *just as they do with the rest of the UI in their application*.

There two ways to wrap a Xaml FrameworkElement within a UIView; both are exposed off of the *new* UIView (UIKitXamlExtensions) class category:

##### - (instancetype)initWithFrame:(CGRect)frame xamlElement:(WXFrameworkElement\*)xamlElement

*Any* Xaml FrameworkElement can be passed into our custom UIView init function, and that FrameworkElement is the *same object* that is passed into the UIView’s backing CALayer (and its backing LayerProxy) for layout and rendering purposes.

***This is the recommended approach for developers to easily integrate new Xaml UI into their ported applications.***

[TODO: Sequence diagram and ultimate diagram showing the view, layer, and its xaml element?]

##### + (WXFrameworkElement\*)createXamlElement

Custom UIViews can ‘declare’ the type of their Xaml FrameworkElement by implementing the static *createXamlElement* method that we’ve added to a new UIView (UIKitXamlExtensions) class category.

WinObjC’s default UIView init path calls this method (if it exists) on custom UIView types, similar to the pattern that UIView uses when creating its backing [layerClass](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIView_Class/).

***This is the recommended approach for custom UIViews that are built upon a specific Xaml FrameworkElement type***, and it’s the approach we use for all of our Xaml-backed UIKit controls (UIButton, etc.).

[TODO: Sequence diagram and ultimate diagram showing the view, layer, and its xaml element?]

### UIKit Control Types

UIKit controls come in *essentially* three forms:

#### Opaque Controls

Controls such as UILabel, UIImageView, UIActionSheet, UIDatePicker, and UIProgressRing all fall into this category.

Such controls are relatively easy to port to Xaml, because their internal pre-canned subviews (if any) are not exposed to subclasses for dynamic layout purposes.

The opaque nature of these controls allows us to build their *entire control* implementation in Xaml (by subclassing a Xaml control, providing a custom control template, etc.), and we can perform straightforward mappings of the UIKit control’s customization properties (fonts, colors, insets, images, text, etc.) across to our Xaml implementation as needed.

***Opaque controls are the easiest for us to build on Xaml***, and fortunately, they make up *the* *majority* of the in-box UIKit controls.

#### Composited Controls

Controls such as UIButton fall into this category. UIButton contains two pre-canned subviews; titleLabel (a UILabel), and imageView (a UIImageView). UIButton subclasses *frequently* override certain UIButton methods ([contentRectForBounds](https://developer.apple.com/reference/uikit/uibutton/1624027-contentrectforbounds?language=objc), etc.), in order to explicitly position the pre-canned subviews at runtime.

Other composited controls (such as UISlider and UITextField) have similar dynamically-override-able layout behavior for their pre-canned subviews, but we haven’t (*yet*) seen that functionality used in ported apps or middleware.

Opaque control implementations have sufficed for UISlider and UITextField so far, but we had to keep dynamic layout support intact for our UIButton implementation for compatibility with our target middleware.

Composited controls ***can in many cases be implemented primarily in Xaml*** (as we’ve done for UITextField and UISlider), yet ***others such as UIButton require more work*** (particularly around layout) on the UIKit side, in order to enable their P0/P1 scenarios. It’s extremely common for UIButton subclasses to explicitly lay out their pre-canned subviews at runtime, so we must support it.

#### Collection Controls

Controls such as UICollectionView and UITableView fall into this category; they provide *complex* layout and positioning support for collections of UIViews. Given the dynamic nature of layout on iOS, these are *extremely difficult* for us to map to anything useful on the Xaml side.

For example, Xaml’s GridView (or similar) *may* work for simple UITableView scenarios, but it will quickly fall over for many seemingly-basic scenarios that are commonly used in ported apps. Attempting to port such controls to Xaml may not be practical for this project, as it will require us to write the implementations ourselves, in Xaml-derived C++/CX controls, while *also* being tied very closely to the UIKit side in order to integrate with UIKit, call into derived classes, and to fire registered delegate callbacks/events. ***Such an exercise likely becomes a question of where we write our custom control code (essentially a C++/CX vs ObjC discussion).***

***Collection controls*** are difficult (*potentially impossible*) to implement sensibly in Xaml, although we plan to add *some amount* of support for them in Xaml markup. At the *very least*, we’ll start by building objects that we can express in markup and use to bootstrap the UIKit collection controls. Keep in mind that, as stated at the beginning of this document, WinObjC is not currently tasked with building ‘fancy’ custom controls that are consumed by arbitrary UWPs

The following chart lists all in-box UIKit controls and their respective ‘control types’ as it stands today. Keep in mind that this is a current snapshot; just because something isn’t yet Xaml-Backed doesn’t mean that we won’t move it to Xaml in the future.

|  |  |  |  |
| --- | --- | --- | --- |
| **UIView** | **Type** | **Notes** | **Xaml-Backed** |
| MKMapView | Opaque | Not implemented | N/A |
| UIActionSheet | Opaque |  | Y |
| UIActivityIndicatorView | Opaque |  | Y |
| UIAlertView | Opaque |  | N |
| UIButton | Composited |  | Y (1609) |
| UICollectionView | Collection |  | N |
| UIDatePicker | Opaque |  | N |
| UIImageView | Opaque |  | N |
| UILabel | Opaque |  | Y (with composition refactor) |
| UINavigationBar | Opaque |  | N |
| UIPageControl | Opaque | Not implemented | N/A |
| UIPickerView | Opaque |  | N |
| UIPopoverBackgroundView | Opaque | Not implemented | N/A |
| UIProgressView | Opaque | Maps to Xaml ProgressBar | N |
| UIRefreshControl | Opaque | Not implemented | N/A |
| UIScrollView | Unique | ~In a category of its own; ported to Xaml in hopes of better scrolling and gesture integration. | Y |
| UISearchBar | Opaque |  | N |
| UISegmentedControl | Opaque | Relatively simple UIKit implementation. | N |
| UISlider | Composited | Built entirely on Xaml; failure to expose positioning of subviews is non-blocking so far. | Y (via an opaque Xaml implementation even though it’s a composited control) |
| UIStackView | Collection | Not implemented | N/A |
| UIStepper | Opaque | Not implemented | N/A |
| UISwitch | Opaque | Easy map to Xaml ToggleSwitch? | N |
| UITabBar | Opaque | Port in progress. | Port in progress. |
| UITableView | Collection |  | N |
| UITextField | Composited | Built entirely on Xaml; failure to expose positioning of subviews is non-blocking so far. | Y (via an opaque Xaml implementation even though it’s a composited control) |
| ;UITextView | Composited |  | N |
| UIToolbar | Opaque |  | N |
| UIWebView | Opaque | UIWebView exposes its scrollView, but we can't expose off of Xaml Web control. Non-blocker. | Y |

### UIControls

Several UIKit controls derive from [UIControl](https://developer.apple.com/reference/uikit/uicontrol), and therefore provide *additional* control-specific functionality to app code and to UIControl-derived classes.

#### UIControl States

UIControls transition through various [UIControlStates](https://developer.apple.com/reference/uikit/uicontrolstate?language=objc): ‘normal’, ‘highlighted’ (aka pressed), ‘disabled’, ‘selected’ (a state that is toggled by app code), as well as ‘application’ (app-defined states that are defined and toggled by app code).

At first glance, it would appear that UIControlStates map well to Xaml’s VisualStateManager VisualStateGroups, but we’ve hit some *significant* gaps when attempting to build our UIControlStates on VSM:

1. ***UIControlStates are a bit mask***; a button can be ‘pressed’ *and* ‘selected’, whereas VSM only allows a control to be in a *single* *state at a time* for a given property.
2. It’s difficult to track and query the current state in VSM, yet iOS app code often queries or dynamically responds to UIControl state changes.
3. VSM doesn’t support arbitrary new application states (although admittedly low-pri at this point and not yet implemented).

These are *significant* limitations with VSM that we can’t easily work around, so we must continue to track our UIControlStates within the UIControl (and its derived implementations in our codebase and ported app code), and we must update their UI as needed from within the derived UIControls (just as the UIControl-derived classes expect).

For our control implementations, the actual UI state updates will ***simply toggle properties on their backing Xaml implementations*** (change their background color, label text, etc.), rather than doing any form of custom rendering for the state transition. The key point is that ***we won’t be using VSM for UIControl state transitions***.

**Note:** This means that we’ll ***also*** have to manage/track states that are toggled via keyboard input. For example; if we *only* key off of pointer events for ‘pressing’ a button, the button won’t appear pressed when using the Enter key or the GamepadA button on an Xbox controller. ***We’ll need to schedule time to design a reusable solution for this in the UIControl base class.***

#### UIControl Events

UIControls follow the [target-action pattern](https://developer.apple.com/library/content/documentation/General/Conceptual/Devpedia-CocoaApp/TargetAction.html), and similar to touch input events, UIControl actions are fired up the UIResponder chain until they’re handled by a UIResponder. Each UIControl fires specific events for that control type. For example, UIButton fires UIControlEventTouchUpInside for a ‘button click’, and UITextField fires UIControlEventEditingChanged as the user modifies text within the text field.

***Each of our UIControl implementations must ensure that their respective UIControl events are fired at the correct time for their scenarios.***

**Note:** Furthermore, our current UIControl implementation is incomplete. We don’t handle drags outside of a control very well, and there are many events/state transitions that we don’t have hooked up at all. ***We’ll need to schedule time to build out the remaining functionality needed in our UIControl base class.***

#### UIControl Keyboard/Commanding Events

UIControl ‘touch’ events are already fired correctly by our UIControl base class, but those events *are not* fired when the Xaml object is interacted with via the keyboard or through accessibility tools. Therefore, we must subscribe to our backing Xaml controls’ events, and fire them through UIControl as needed, while ***ensuring that we never ‘double-fire’ events***.

For example, UIButton should fire UIControlEventTouchUpInside if its backing Xaml Button fires its OnClick event due to keyboard input, but *not* if it’s clicked through mouse/touch input (as the latter is already handled by UIControl’s touch input processing). ***Each UIControl implementation must take this into account.***

**Note:** Although this is somewhat control-specific, we should make sure each of our UIControl implementations hooks up to accessibility/keyboard/commanding properly.

### Layout

UIView layout is built directly upon the Core Animation layout functionality discussed above. Custom UIViews, and/or app/middlware code running on WinObjC set the frame, bounds, etc. of a UIView *just as they do on iOS*.

These property-changes are passed into the UIView’s backing CALayer, which in turn passes them into our CACompositor to apply the values on the UIView’s backing Xaml FrameworkElement. The Core Animation layout section above contains a detailed diagram.

#### Arbitrary Xaml Element

[TODO: Add diagram showing where layout is managed by us, and where it’s managed by Xaml?]

#### Opaque Controls

For opaque controls, we are able to leverage Xaml for all layout *within* the Xaml control implementation, but we must continue to leverage Core Animation for layout of the UIView (and therefore its CALayer and backing Xaml FrameworkElement) with the UIView tree, in order to achieve a high level of ported UI reuse.

[TODO: Add diagram showing where layout is managed by us, and where it’s managed by Xaml?]

#### Composited Controls

For composited controls such as UIButton, ported apps expect the ability to dynamically lay out the control’s pre-canned subviews, so we must provide that same functionality in WinObjC, in order to achieve a high level of ported UI reuse.

We do so by adding the pre-canned Xaml FrameworkElements to the Xaml control implementation’s ControlTemplate, but we wrap them their respective UIViews on the UIKit side, so app/middleware code can *continue* to explicitly lay them out (by setting the subview frame/bounds directly, and/or by taking part in derived callbacks such as [contentRectForBounds](https://developer.apple.com/reference/uikit/uibutton/1624027-contentrectforbounds?language=objc), which provide ‘hints’ as to where the subview should be placed).

#### Collection Controls

Collection controls are *essentially* a more complicated version of composited controls, but there will likely be fewer markup-backed UIElements within the Xaml side of collection control implementations, because we only place the control’s ‘pre-canned’ subviews into their Xaml markup (UIButton’s titleLabel and imageView for example).

#### Autolayout

Autolayout is built upon the exiting UIView/CALayer layout engine, so ***there are no immediate changes needed to our Autolayout implementation***.

However, Xaml-triggered resizing may occur in some cases. For example, app code may change the text on a UILabel’s TextBox. In such cases, we can trigger a re-layout pass on the UIKit side, so Autolayout is able to recalculate and reapply all relevant layout constraints. We can do so by implementing ArrangeOverride and calling back out to the UIView to kick off a layout update pass.

// This method is called multiple times by XAML, and we call back to UIButton to layout the views.

Windows::Foundation::Size Button::ArrangeOverride(Windows::Foundation::Size finalSize) {

\_\_super::ArrangeOverride(finalSize);

\_layoutHook->Invoke(); // Calls setNeedsLayout on the owning UIButton

return finalSize;

}

[TODO: Add diagram showing how a UILabel must call back out to trigger AutoLayout recalculations.]

### Rendering

Leveraging Xaml for our control implementations doesn’t affect the *default* UIView/CALayer rendering path, which will continue to function as-is, although it will leverage the new/refactored CACompositor architecture.

#### Custom Rendering

Default UIViews will be backed by default CALayers, which are in-turn backed by our default Xaml representation of a Layer (UIKit::Private::CoreAnimation::Layer). CACompositor will grab the Layer’s Image for rendering purposes as needed by the application/middleware. ***No changes are required to CALayer or UIView to ensure this behavior, as the correct logic is already present.***



#### Xaml-Rendered

Our UIKit controls that are Xaml-backed won’t implement a drawRect: method, and therefore won’t be doing any custom rendering via CoreGraphics or CoreText. ***No changes are required to CALayer or UIView to ensure this behavior, as the correct logic is already present.***



#### Xaml-Augmented

If apps or middleware derive from our Xaml-backed controls, they can (and often will) provide a drawRect: method. In such cases, ***we’ll allow them to augment the Xaml-Rendered UI with their own embellishments***, as long as the Xaml FrameworkElement backing the UIView both *a)*implements UIKit::Private::CoreAnimation::ILayer, and*b)*provides a non-null Image from its LayerContent property). ***No changes are required to CALayer or UIView to ensure this behavior, as the correct logic is already present.***



#### Custom-Rendered (Xaml-Backed)

A fourth rendering option is *likely possible*. We can potentially allow app developers to ‘disable’ our default Xaml-Rendered content, and instead perform their own custom rendering for all of the Xaml-backed control’s visual contents.

**Note:** This feature would require more thought and design, so it’s not currently implemented and won’t roll out with the initial CACompositor refactor. More customer input would be required to build something useful.

### Subview Management

Subviews (and therefore well as Sublayers) are supported by the same LayerCoordinator sublayer functionality documented above. We will ‘enlighten’ our existing Xaml-backed UIKit control implementations as needed by customers, and all new Xaml-backed control implementations should default to supporting Subviews. The end goal is to support Subviews on all of our UIKit control implementations.

### Windows-Specific Control States

[TODO: Cover how we support Windows-specific visual control states such has hover and keyboard focus.]

#### Hover (mouse over)

#### Focus

### Keyboard Navigation and Input

[TODO: Cover how we’ll handle button clicks, etc. via keyboard input, including how we’ll transition between UIControl states]

## UIKit.Xaml

UIKit.Xaml.dll contains the Xaml implementations of our UIKit controls. We currently instantiate and interact with them between UIKit.dll<->UIKit.Xaml.dll via a flat hand-rolled C API. However, we plan to project UIKit.Xaml’s .winmd into ObjectiveC, so UIKit can make calls directly into the projected controls (thus removing the need for the hand-rolled C API).

Projecting the controls to UIKit will also help ensure that our Xaml controls present sensible APIs.

**Note:** We should add this work to our backlog and work with the tools team as needed.

## C# Middleware

[TODO: Talk about the challenges here, and the solutions.

Talk about how we had to move/isolate the dolayout stuff, but have a section on it above and just point to that??

How to ‘present’ ViewControllers – probably in a ContentDialog

We’ll need some type of proxy for null window/rootviewcontroller scenarios

How to bootstrap from Designer]

## Xaml Markup

[TODO: How will our UIElements be bootstrapped ALL FROM MARKUP? How will we work in the designer?

UIKit.UIView, how that is basically just another Private::Core Animation::Layer that also knows how to interpret AutoLayout attached properties.

Data binding to ViewControllers

Hooking up our controls, as well as arbitrary Xaml elements (wrapped in UIViews)

How that bootstrapping will work.]

## Testing

We’ll have distinct sets of tests for each stack of our UI framework;

### Core Animation

1. Unit/functional tests.
2. Core Animation Test app.
3. Regression testing in existing test, sample, and production apps and middleware.

### UIKit

1. Unit/functional tests.
2. UIKit Test app.
3. Regression testing in existing test, sample, and production apps and middleware.

## Appendix / Scratch Space

### Open Questions

* CALayer maxes rendering at 2048x2048; any reason for this? We should probably just remove this hard-coded limitation.

### UIWindow Notes

* setNodeTopmost on UIWindow calls all the way down into CACompositor; we should just set Z-index on the backing xamlElement!
* Get rid of the keyboard one, how to remove the popover one. On iOS can you just create new windows willy-nilly? How will we handle multi-window scenarios? How will we support middleware when we don’t have any window or superview?
* Window resize -> gridsizechagned->resetwindow size->setneedslayout->etc.
* Window Resize/Rotation – or in UIApplication below?

### UIViewController Notes

* Dead code around rotation
* What else – navigation concerns? Interop with layout?

### UIApplication Notes

* Tons of dead/legacy code can/should be removed.

### Screen/Device Settings Notes

* Need to document WocDisplayMode, how it works, etc.
* Need to document and streamline our display orientation and rotation strategy.

### Keyboard Focus and Navigation

* Deserves separate dev doc.
* Need to figure out how to make any UIView focusable.
* How to hook into UIKit API surface, or maybe just document a pattern of APIs to call on the Xaml element for controlling focus?

### Accessibility

* Deserves separate dev doc.
* Clicks to touchesBegan, touchUpInside, etc…?

### CALayer Types

This section covers the various Core Animation types, and their respective properties.

### CALayer

Still need to add ‘border’ support; what else?

### CATextLayer

Used to exist, but stubbed-out post-refactor since it was only used by UILabel (which now has a concrete UIKit.Label type).

### CAEAGLLayer

### CAShapeLayer

### CAEmitterLayer

### CAAnimation

### CAAnimationGroup

### CATransactionManager