

CD Quartz Cocoa Lib User Guide.

Overview.	1
Project Repositories.	1
How to Build CD Quartz Cocoa Lib.	1
Quartz Cocoa Lib	2
<i>Description</i>	2
<i>Lifecycle</i>	4
<i>Library Include File.</i>	5
<i>Library Organisation.</i>	5
<i>Context</i>	5
<i>Modifier Queue</i>	6
<i>Context Modifier</i>	7
<i>List of Context Modifiers.</i>	7
<i>A Usage Example.</i>	10

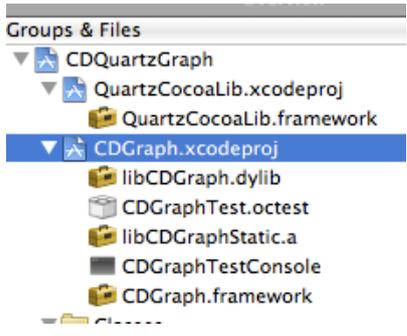
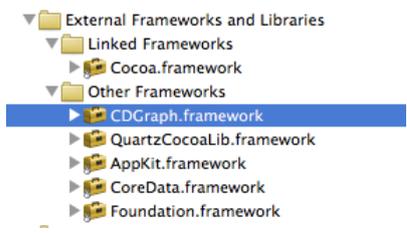
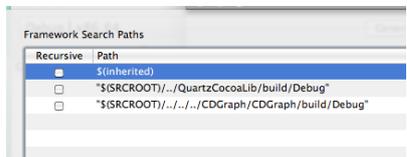
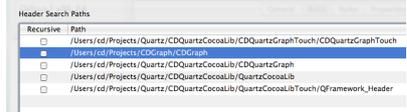
Overview.

This document provides details of how to incorporate this library into your own projects.

Project Repositories.

Project Name	Git Hub URL	Description
CDQuartzCocoa Lib	http://github.com/cxd/CDQuartzCocoaLib	This is the main project.
CDGraph	http://github.com/cxd/CDGraph	This project contains the library CDGraph on which CDQuartzCocoaLib depends. This is located in a separate repository as it is shared by a separate project.

How to Build CD Quartz Cocoa Lib.

1	Checkout a copy of CDGraph to a working directory.	
2	Open the CDGraph project in XCode and build it.	
3	Check out a copy of CDQuartzCocoaLib.	
4	Open the CDQuartzCocoa project in XCode.	
5	Adjust the project dependency for CDGraph.xcodeproj Use Command-I to choose the correct path.	 <p>The screenshot shows the 'Groups & Files' pane in Xcode. The project 'CDQuartzCocoaLib' is expanded, showing sub-projects 'CDGraph.xcodeproj' and 'QuartzCocoaLib.xcodeproj'. Under 'CDGraph.xcodeproj', several files are listed: 'libCDGraph.dylib', 'CDGraphTest.octest', 'libCDGraphStatic.a', 'CDGraphTestConsole', and 'CDGraph.framework'.</p>
6	Adjust the path to the CDGraph.framework in XCode. The framework will be produced by building CDGraph. It will be located in the CDGraph/build/Debug directory.	 <p>The screenshot shows the 'External Frameworks and Libraries' pane. Under 'Other Frameworks', 'CDGraph.framework' is selected and highlighted in blue. Other frameworks like 'Cocoa.framework', 'AppKit.framework', and 'Foundation.framework' are also visible.</p>
7	Adjust the framework search paths. Make sure that the target uses the correct search path for any dependencies.	 <p>The screenshot shows the 'Framework Search Paths' pane. It has a 'Recursive' checkbox checked and a 'Path' column. Three paths are listed: '\$(inherited)', '\$(SRCROOT)/../QuartzCocoaLib/build/Debug', and '\$(SRCROOT)/../../CDGraph/CDGraph/build/Debug'.</p>
8	Check include search paths. Make sure they include the correct paths to CDGraph or CDGraphTouch if compiling a *Touch project.	 <p>The screenshot shows the 'Header Search Paths' pane. It has a 'Recursive' checkbox checked and a 'Path' column. Several paths are listed, including '/Users/ed/Projects/Quartz/CDQuartzCocoaLib/CDQuartzGraphTouch/CDQuartzGraphTouch', '/Users/ed/Projects/CDGraph/CDGraph', and '/Users/ed/Projects/Quartz/CDQuartzCocoaLib/CDQuartzGraph'.</p>
8	Repeat the process for other projects within the CDQuartzCocoaLib project that have a dependency on CDGraph.	<p>CDQuartzGraph CDQuartzGraphTouch</p>

Quartz Cocoa Lib

Description

The Quartz Cocoa Lib is built upon a series of “modifiers” or operations that change a “context”, these modifiers are sequenced in one or more “queues”, a queue can then be executed changing the context by applying the modifiers in sequence.

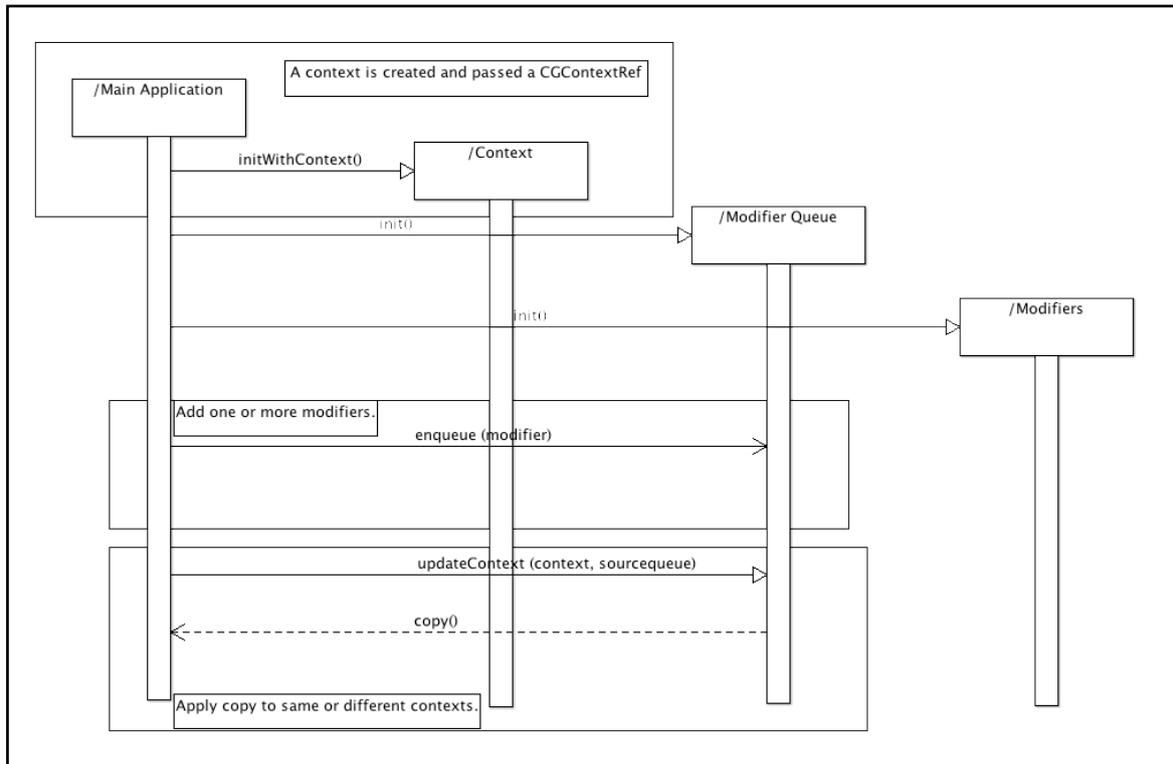
Sequences of modifiers can be constructed and stored for later execution (such as from a serialized archive) or executed on different contexts (such as execution on a graphics context for drawing to the screen or on a graphic context for drawing to a bitmap image).

CD Quartz Cocoa Lib User Guide.

It is possible to add additional “context” types and “modifier” types as needs arises, currently Quartz based contexts and modifiers are supported. However the same concept could be applied to different graphics libraries.

Lifecycle

The following diagram depicts a typical usage lifecycle of the library, although the method calls shown in the diagram are not literal counter parts of the api.



Step	Description	Parameters
Main Application Initialises Context	The “QContext” object is created.	During creation the CGContextRef is retrieved from the environment or created (if writing to an image) and supplied to the QContext object.
Modifier Queue is Created	A “QModifierQueue” is created, this queue will be used to store “modifiers” that will be used to change the context.	
Create and Enqueue Modifiers.	Different “modifiers” are created and stored on the “QModifierQueue” in the order that they should be applied to the context.	Different modifiers may require different initialisation parameters, such as size, colour and the like.

Step	Description	Parameters
<p>Update the Context.</p>	<p>Once the “QModifierQueue” has been configured with a sequence of modifiers it can then be used to update the context.</p> <p>The call to “updateContext” is used to apply each modifier in the sequence to the graphics context. A copy of the original “QModifierQueue” is returned from this operation and the original queue can be destroyed.</p> <p>In most cases the “UpdateContext” cycle is the repeating action that is applied whenever a context should be changed.</p> <p>Alternately, new “modifiers” may be added to the queue in between repetitions.</p> <p>Additionally multiple queues may be created by different “modifiers” to allow for nested sequences where required.</p>	<p>The “QContext” is supplied as well as the “QModifierQueue” instance that contains the sequence of modifiers to apply to the context.</p>

Library Include File.

To use the library, include the following header.

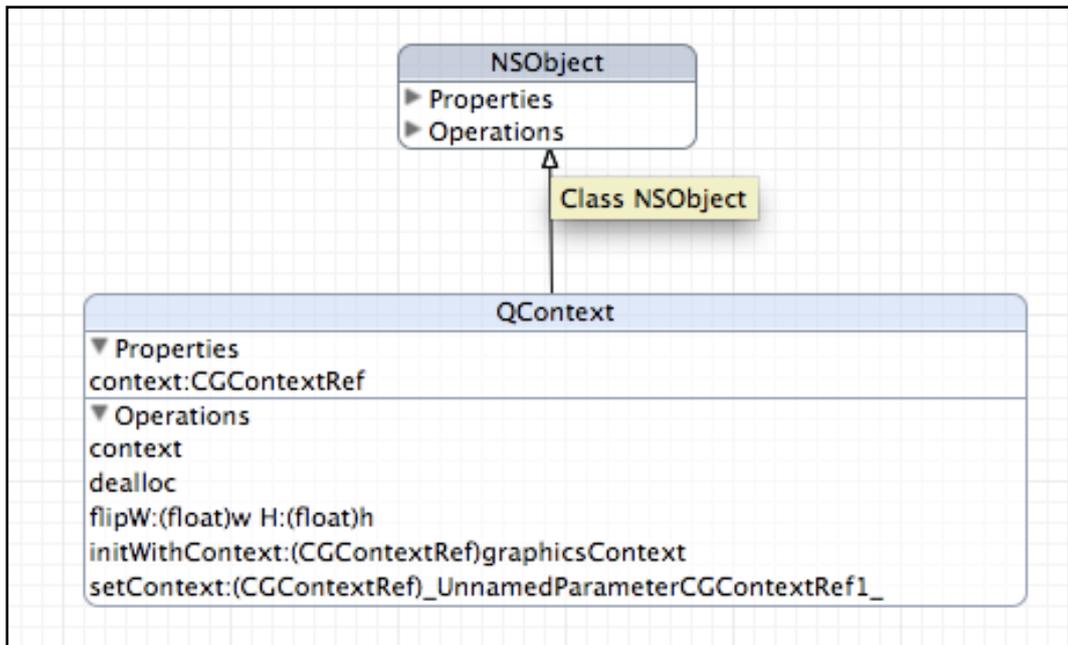
```
#import "QuartzCocoaLib/QuartzCocoaLib.h"
```

Library Organisation.

Context

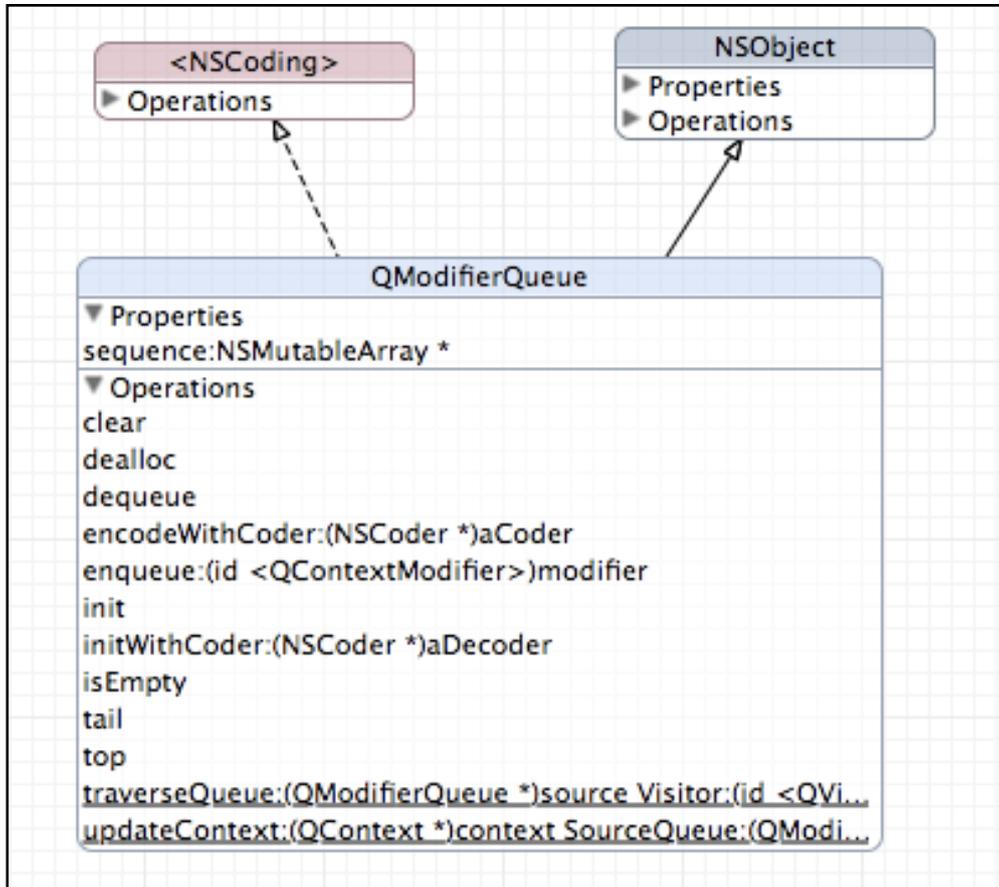
The context is a container object for the graphics context that will be acted upon by modifiers.

This context is represented by the “**QContext**” object which is used to store a reference to the “CGContextRef”. The “CGContextRef” allows different types of graphics contexts to be stored.



Modifier Queue

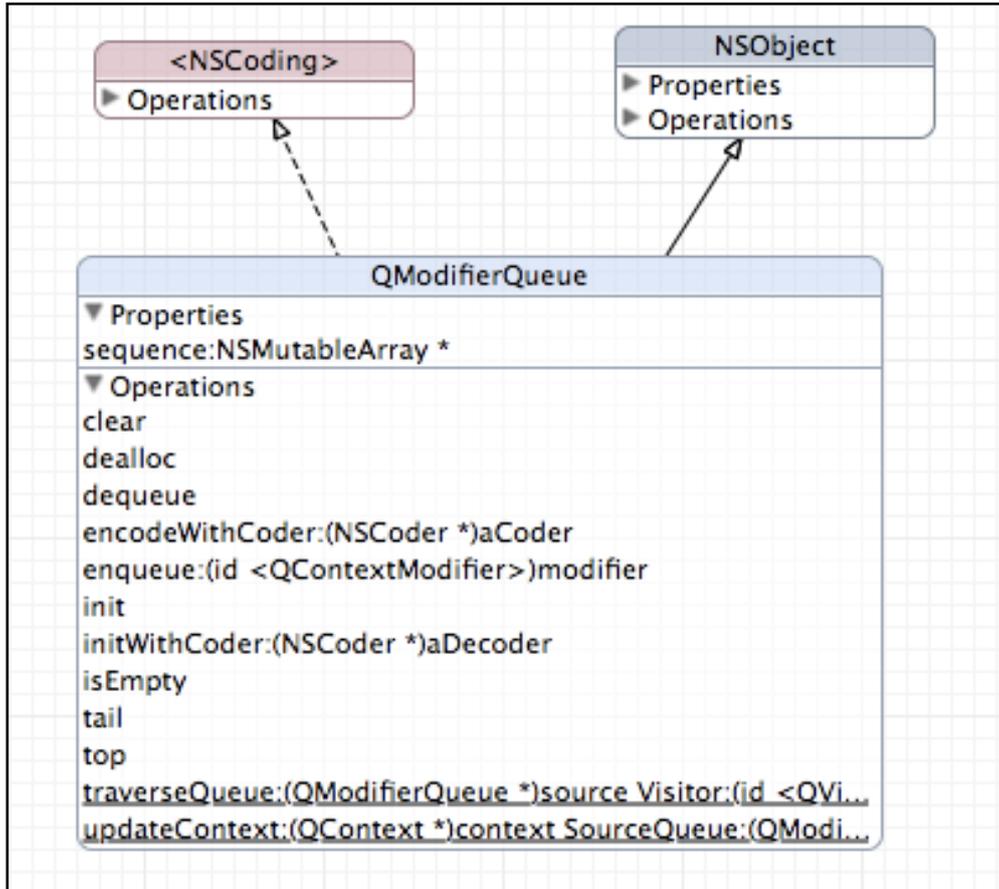
The modifier queue “**QModifierQueue**” is used to store sequences of context modifiers which can be applied to one or more “Contexts”. The modifier queue is a short lived object, when it is applied to the context a copy of the queue is returned and the original object can be destroyed.



Context Modifier

The context modifier is an object that is capable of changing the graphics context, this may be a shape or some sort of mode for the graphics context. Lines, Circles, Rectangles, Shadows and Colours are represented as Context Modifiers.

A context modifier will implement the protocol “**QContextModifier**” and for convenience may extend the base class “**QAbstractContextModifier**”.



The role of the modifier is to “update” a context that is supplied to it. A modifier may be composed of nested sequences of “modifiers” that can be used to construct complex operations, and generally uses an internal “**QModifierQueue**” for this purpose.

A “**QContextModifier**” should also implement the “**NSCoding**” protocol in order to support serialization so that sequences of changes can be serialized and deserialized and potentially applied across different processes or different times.

The process of using the library generally consists of using existing “**QContextModifiers**” and potentially building more complex “**QContextModifiers**” in order to affect further changes in the graphics context.

List of Context Modifiers.

The following table lists the available context modifiers. It is recommended to review the source code for details on how to initialise them.

Generally, the process of using a modifier only requires that they be initialised with their various initialisation parameters and then added to a modifier queue (via the “enqueue”

call) the modifiers will then apply their changes in the order that they have been added to the queue (in this way the process of building up a sequence of modifiers is somewhat declarative or similar to “configuring” a queue of operations).

For details of the initialisation method calls, review your copy of the source code.

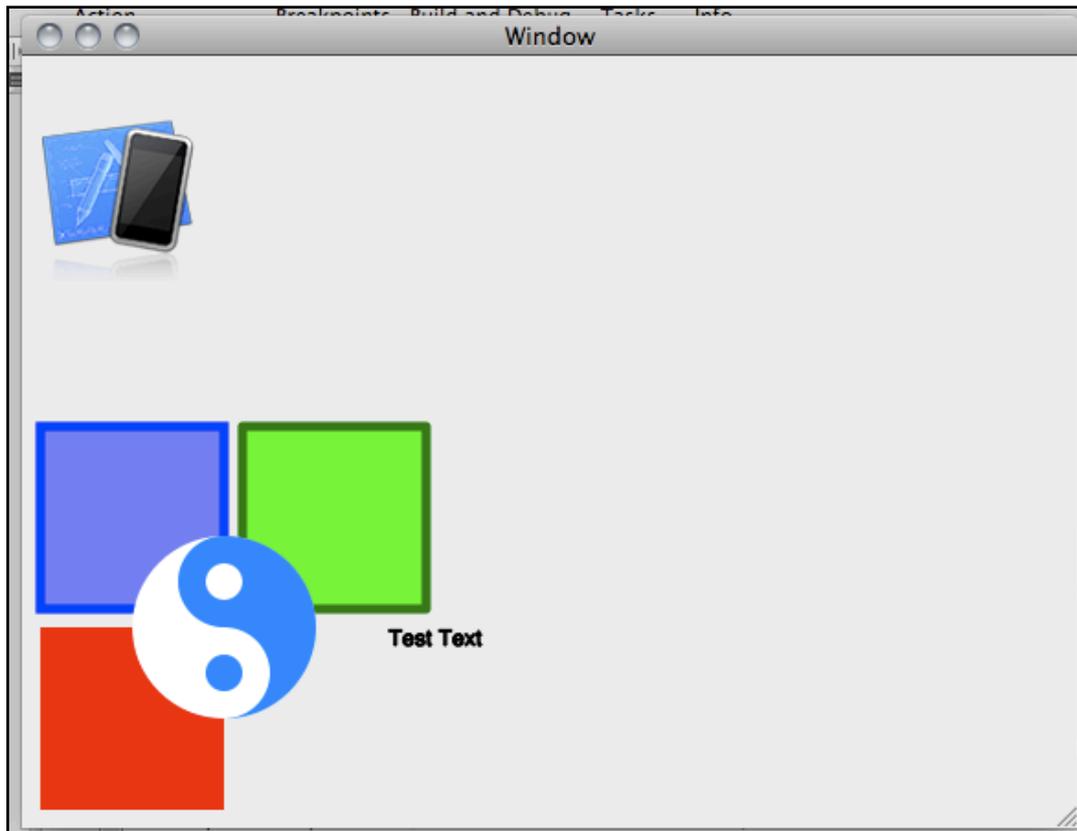
Type Name	Description	Type
QArc	An arc can be used to draw curves and uses a centre point, a radius and start and end angles to define its extent. Multiple arcs can be concatenated together.	Concrete
QBezierCurve	A bezier curve is a line that is drawn using two control points to determine magnitude of the curve at the start and end.	Concrete
QBlendMode	A blend mode will set the graphics context to use a specific type of blending when drawing.	Concrete
QBoundedObject	The bounded object is a protocol declaration for any shape that may require the bounds to be computed.	Protocol
QColor	A color is used to describe RGBA components for drawing and is the base type for color modifiers.	Abstract Base
QFillColor	A fill color describes the color with which shapes should be filled within the current context.	Concrete
QFilledRectangle	A filled rectangle describes a rectangle that will be filled with a color.	Concrete
QFillPath	A fill path describes a series of lines, either “QArc”, “QBezierCurve” or “QLine” that will be filled with a “QFillColor”.	Concrete

Type Name	Description	Type
QImage	A QImage provides a convenient way to draw images in the current context.	Concrete
QJoinCap	A QJoinCap describes the style with which joints in curves should be drawn.	Concrete
QLabel	A QLabel provides a convenient way to draw a single line of text.	Concrete
QLine	A QLine is stroked path with a start and end point.	Concrete
QLineCap	A QLineCap specifies the style with which terminated lines will be drawn.	Concrete
QMove	A move is akin to lifting a pen from the paper and moving it to another point on the paper.	Concrete
QPoint	A point is a 2 dimensional unit of measure. The QPoint object also has a number of geometry helper methods.	Unit of Measure
QRectangle	A rectangle is the base class for shapes that are drawn in the graphics context. It can also be used as a unit of measure and provides a number of geometry methods.	Abstract Base

Type Name	Description	Type
QRestoreContext	<p>In quartz it is possible to save and restore the graphics context state. When a state is saved new graphics operations can be applied without direct impact on the previous graphics state. When restored the previous graphics state can be modified directly. Using save and restore state allows for nested modifications to the graphics context.</p>	Concrete
QSaveContext	<p>Save context allows the current graphics state to be saved prior to any further changes. Save should always be followed by a “restore” at some later point.</p>	Concrete
QShadow	<p>A shadow will be applied to graphics shapes within the current context.</p>	Concrete
QStrokeColor	<p>The specified stroke color to be used for outlined shapes.</p>	Concrete
QStrokedRectangle	<p>A rectangle that also has an outline.</p>	Concrete
QStrokeWidth	<p>The width to use for shape outlines.</p>	Concrete
QTestYinYang	<p>A test shape that draws a yin yang symbol in a nested modifier queue.</p>	Concrete

An Example.

The following example is an excerpt from the test project “**TestGraphicsContext**”, when run the application will draw a number of shapes into a view resulting in the following image.



The following listing illustrates the configuration of the “QModifierQueue” and the sequence of “modifiers” that were created to generate the test program.

```

- (void)drawRect:(NSRect)rect {
    // a context reference.
    // contexts can reference any type of graphics context.
    if (context == nil)
    {
        context = [[QContext alloc] initWithContext:[[NSGraphicsContext
currentContext] graphicsPort]];
        [context retain];
        queue = [[QModifierQueue alloc] init];
        [queue retain];

        // the set of instructions to store within a queue.
        [queue enqueue:[[QFillColor alloc] initWithRGBA:1 G:0 B:0 A: 1]];
        [queue enqueue:[[QFilledRectangle alloc] initWithX:10 Y:10 WIDTH:100 HEIGHT:
100]];
        [queue enqueue:[[QFillColor alloc] initWithRGBA:0 G:0 B:1 A: 0.5]];
        [queue enqueue:[[QFilledRectangle alloc] initWithX:10 Y:120 WIDTH:100 HEIGHT:
100]];

        // test stroke rectangle.
        [queue enqueue:[[QSaveContext alloc] init]];
        [queue enqueue:[[QStrokeWidth alloc] initWithWidth:5.0]];
        [queue enqueue:[[QStrokeColor alloc] initWithRGBA:0 G:0 B:1 A: 1]];
        [queue enqueue:[[QStrokedRectangle alloc] initWithX:10 Y:120 WIDTH:100 HEIGHT:
100]];
        [queue enqueue:[[QRestoreContext alloc] init]];

        // test shadow and outline
        [queue enqueue:[[QSaveContext alloc] init]];
        [queue enqueue:[[QShadow alloc] initWithBlur:2.0 0:5.0 Y0:-5.0]];
    }
}

```

CD Quartz Cocoa Lib User Guide.

```
[queue enqueue:[[QFillColor alloc] initWithRGBA:0 G:1 B:0 A:1]];
[queue enqueue:[[QFilledRectangle alloc] initWithX:120 Y:120 WIDTH:100 HEIGHT:
100]];
[queue enqueue:[[QRestoreContext alloc] init]];
[queue enqueue:[[QSaveContext alloc] init]];
[queue enqueue:[[QLineCap alloc] initWithStyle:QLineCapRounded]];
[queue enqueue:[[QJoinCap alloc] initWithStyle:QJoinCapRounded]];
[queue enqueue:[[QStrokeWidth alloc] initWithWidth:5.0]];
[queue enqueue:[[QStrokeColor alloc] initWithRGBA:0.0 G:0.5 B:0.0 A: 1]];
[queue enqueue:[[QStrokedRectangle alloc] initWithX:120 Y:120 WIDTH:100 HEIGHT:
100]];
[queue enqueue:[[QRestoreContext alloc] init]];
// create test shape yin/yan circle.
[queue enqueue:[[QTestYinYang alloc] initWithCentre:[[QPoint alloc] initWithX:
110 Y:110]];
// test loading an image from a url.
[queue enqueue:[[QImage alloc] initWithUrl:@"http://devimages.apple.com/
home/images/iphonedevcenter.png" X:10 Y:300]];
[queue enqueue:[[QSaveContext alloc] init]];
[queue enqueue:[[QStrokeColor alloc] initWithRGBA:0 G:0 B:0 A: 1]];
[queue enqueue:[[QLabel alloc] initWithText:@"Test Text" X: 100 Y: 50
WIDTH:250 HEIGHT:100]];
[queue enqueue:[[QRestoreContext alloc] init]];
}
QModifierQueue *copy = [QModifierQueue updateContext:context SourceQueue:queue];
[queue autorelease];
queue = copy;
[copy retain];
}
```

The body of code above can be separated into three sections (as shown in the section on the “lifecycle”).

The first section creates a reference to the context and the queue.

```
context = [[QContext alloc] initWithContext:[[NSGraphicsContext
currentContext] graphicsPort]];
[context retain];
queue = [[QModifierQueue alloc] init];
[queue retain];
```

The second section is concerned with creating different types of modifiers and adding them in sequence to the modifier queue. For example:

```
[queue enqueue:[[QFillColor alloc] initWithRGBA:1 G:0 B:0 A: 1]];
```

The final section executes the changes on the current context, destroys the previous instance of the queue and stores the new copy for use. The last section is executed every time “drawRect” is called (whenever the view is redrawn), the current copy of the queue always contains the sequence of instructions that are used to draw into the context.

```
QModifierQueue *copy = [QModifierQueue updateContext:context
SourceQueue:queue];
[queue autorelease];
queue = copy;
```

```
[copy retain];
```

How to Create a Context Modifier

A context modifier for convenience should derive from the base class “**AbstractContextModifier**”.

When defining a new context modifier there are two general approaches:

- Compose the new modifier out of smaller existing modifiers.
- Perform all context changes within the modifier itself.

In both cases some initialisation parameters may be required (this is up to the type of work the modifier will perform).

The modifier will apply changes to the supplied context in the “update” method implementation. The method signature is:

```
/**  
 * Change the supplied context.  
 **/  
-(void)update:(QContext *)context;
```

The supplied context will have a reference to the CGContextRef where changes can be applied using the quartz api.

Additionally the “modifier” should implement the “**NSCoding**” protocol in order to permit serialization, as it is convenient be able to read and write modifier queues to file.