Topic

- ☐ DrawKit Programmer's Reference
  - ☐ Titles, legal mumbo-jumbo, disclaimers, etc.
  - ☐ Contents
  - ☐ Introduction

DrawKit is a software framework that enables the Mac OS X Cocoa developer to rapidly implement vector drawing and illustration features in a custom application. It is comprehensive, powerful and complete, but it is also highly modular so you can make use of only those parts that you need, or go the whole hog and drop it in as a complete vector drawing solution.

Taking its cue from Cocoa's powerful text handling model, DrawKit provides a general purpose and complete drawing model that can be deployed with very little (or in the most general case, no) code at all. The defaults for most classes and created objects have been selected to give a working system out of the box with minimal set up. As you might expect, the more your requirements deviate from the defaults, the more customising you will need to do, but DrawKit has been designed to make this as straightforward as possible without compromising on the graphical power available. Many classes can be operated in a variety of modes for the most obvious of customisations, and can of course be subclassed when necessary to provide more divergent behaviour.

Where possible, familiar Cocoa idioms and conventions are used to ensure that the Cocoa developer will be able to start using DrawKit as if it were a natural extension of the standard Cocoa frameworks (which in a way, of course, it is).

Generally speaking, DrawKit provides the following:

• A general-purpose "drawing" data model consisting of unlimited layers organised hierarchically.
• Separation into model, view and controller classes gives genuine architectural flexibility
• Built-in classes for shapes and path objects, and various derivations of them to cover most typical needs.
• Standard grid and guide layers supporting object snapping and any "real world" measurement system you need.
• Built-in selection of objects and targeting of the selection for commands and user events.
• Separation of an object's geometry from its appearance gives incredible flexibility for creating exciting graphics.
• Attachment of unlimited arbirtrary user data to all objects.
• Style objects can be optionally shared by multiple objects, and contain an entire tree of rasterizers for drawing. This goes way beyond the classic "one stroke and one fill per object" that many drawing applications adopt (though if this is what you want it's easy to implement).
• Built-in gradients, vector pattern fills and hatches.
• Interactively edit any bezier path.
• Image objects support all the formats that Cocoa itself supports.
• Text objects.
• Group objects to any degree of nesting. Groups can be rotated, scaled and moved like any shape.
• Many path operations including boolean (set) operations (requires the inclusion of additional code).
• Tool-based drawing, editing and selection operations.
• Export to PDF or any raster image format, as well as its own keyed archive format.
• Built-in Undo.
** Supports multiple views and multiple view classes.*
• Various caching and quality modulating techniques to improve performance when interacting directly.

DrawKit is a moderately large framework but its architecture is straightforward. While you won't be able to learn it in half an hour, it is designed to be easy to deploy and get working with minimal configuration or fuss.

DrawKit does NOT provide a user interface except that of direct manipulation of objects, which is highly customisable. It is intended to form the core of an application or perhaps find a subsidiary role - it is not in and of itself a drawing application. Some classes are provided to help get started with building a GUI for DrawKit, such as a basic document class and a base class for an inspector type of controller.

As its name suggests, DrawKit is a kit - some assembly is required. However getting a "bare bones" system up and running should be very easy, which is intended to give the programmer confidence in the default operation of the framework, providing an excellent starting point for customising and extending DrawKit to suit your own applications' needs.

The DrawKit Demo application is also an essential download for any prospective user of DrawKit. It provides an interface to most of the kit's features allowing them to be explored in a drawing-type application. It is not a drawing application intended for end users however - it is there as a way for programmers to explore the architecture and capabilities of the framework.

  - ☐ DrawKit and You

Cocoa programmers come in all shapes, sizes and levels of experience. If you're a beginner with Cocoa, using Drawkit effectively is likely to be a challenge; DrawKit is not a beginner's framework. However, DrawKit doesn't require knowledge of Bindings or Core Data, which are considered more advanced topics. DrawKit doesn't use Core Data, and it allows you to use Bindings or not as you wish, since it doesn't concern itself with how you implement your user interface.

To properly understand DrawKit, and this documentation, you will need to have some familiarity with Cocoa. Concepts such as memory management, how container classes and strings and many other parts of the Application Kit work is taken for granted, as is the model-view-controller paradigm, KVC, KVO and how views work. DrawKit is heavily based on Cocoa and Quartz graphics, and a good working knowledge of these technologies is highly advantageous. The beginner is directed to any of the many good books and online documentation available.

For the programmer already familiar with Cocoa, DrawKit should present no particular difficulties - it mostly works in the Cocoa domain, rather than Core Graphics or other framework, and employs no unusual techniques or tricky approaches. It makes use of Cocoa's classes for all of its storage, message passing and other needs. In one or two places it does extend Cocoa's classes for improved performance or efficiency (e.g. NSUndoManager) but by and large there should be no surprises. The lower level parts of the code make use of a number of categories on Cocoa classes, such as NSBezierPath and others.

While DrawKit is mostly "back end" or model, some programmers may prefer to approach it as a self-contained custom view, since that is probably how it seems to the user, just as NSTextView is a view (but in fact consists mostly of non-view classes behind the scenes).

  - ☐ DrawKit implementation philosophy

The design of DrawKit is very much driven by the classic 80/20 rule - its defaults and design decisions are intended to cover the 80% of typical uses with minimal set-up or configuration. If you find yourself in the other 20%, you'll need to do a little more work, typically, and the more specialised your needs the more you'll have to do. It's inevitable that DrawKit cannot provide the perfect solution for all possible apps, even within the arena of vector drawing, since every programmer will have different needs.

*One thing that is true of nearly every class, is that they will "just work" with little or no set-up. A high proportion of objects require a straightforward alloc + init to instantiate, with very few requiring complicated designated initializers. Most classes have convenient construction methods to further simplify instantiation for the majority. Of course, these objects may not "just work" how you want them to in this state, but at least they will generally do *something* useful, and any further set up you need to do is really a case of moving them away from their default states. The point of this is to give the user of DrawKit an easier time - instead of wondering if you have marshalled a whole lot of configuration parameters in just the right way to get a working object, you simply start with a working object and gradually adapt it to your needs. It should help give confidence in the framework and be much less error prone. Most development proceeds iteratively, so typically you'll start with a working, but probably non-optimal object, and one step at a time change it to do what you want, testing each time or after each few changes. This way it's much easier to pick up a problem than if you had to start with many tens of settings "up front" . This philosophy reaches its zenith with the complete automatic instantiation of an entire working system by a view if you don't do it for it. It's also typical of much of Cocoa itself.*

While DrawKit has many classes that need to be assembled logically to build a working system, in general there's not usually a strict order for doing things.

Topic

For example, you could add a layer to a drawing, then add objects to the layer - or build a layer complete with objects and add it to the drawing. It's up to you. It's a similar story when styles are assembled from collections of rasterizers - order of assembly isn't too critical.

DrawKit is built for Mac OS 10.4 or later, so does not use Object-C 2.0 language features or any Leopard-only technology.

- ☐ Concepts

  This section fully explores the architecture and design concepts of DrawKit. It provides a comprehensive functional description of how the various classes fit together to provide DrawKit's overall functionality. It does not explore every single class in its entirety - the Class Reference section following this should be referred to for the complete API.

  - ☐ Model, View and Controller

    In common with most object-oriented designs, DrawKit is divided into three major functional areas - the model layer, the controller layer and the view layer. The majority of DrawKit's classes reside in the model layer, with some overlap between this and the controller layer when it comes to handling direct interaction with objects. Distinct "tools" for creating and manipulating objects reside in the controller layer, and the view(s) through which the graphics are displayed are of course part of the view layer.

    The user of a DrawKit application interacts with DrawKit through one or more views. In common with any Cocoa-based application, if the view is key, then keyboard, mouse and menu events are targeted at the view. From there they are processed by the controllers and affect the state of the model. Changes in the model's state, if they affect the visible appearance of an object, are in turn used to mark areas of the view for update.

    In DrawKit, objects that are part of the model are responsible for both drawing themselves and to a large extent, handling their own editing via the mouse. The controller layers coordinate both of these processes. In turn the appearance of an object (how it is actually painted) is handled by its style, and this is separate from the geometry of the object (its path or shape). Each functional area is handled by a distinct class of object. This makes DrawKit both powerful and flexible as well as easier to manage from a maintenance or customisation standpoint.

    At the centre of the DrawKit architecture stands DKDrawing, the "drawing" object. Understanding DrawKit begins here.

  - ☐ The Drawing and Layers

    DKDrawing is the class at the heart of DrawKit, and is at the root of the model. The concept of DrawKit is based on that of the "drawing" as a draughtsman might think of it - a single large "sheet of paper" on which drawing can be done. The principal property of a drawing is its size - how big it is in its horizontal and vertical dimensions.

    DrawKit internally uses Quartz coordinates throughout, and there are approximately 72 Quartz points per inch of drawing space, or 28.35 points per centimetre. Of course a drawing may be viewed on the screen at any scale so this does not necessarily mean one inch on the screen. However it should equal one inch in any printed output. Externally, DrawKit translates any coordinate values it displays to the current grid, which is set up entirely by you the programmer, who in turn may let the user set this. Thus from a user interface perspective, DrawKit works in "real world" values such as millimetres, inches, kilometres or what have you.

    The drawing consists of any number of layers which are overlaid on top of each other. Each layer is transparent by default and exactly covers the full drawing area. Layers are where actual graphical content is generated - the drawing itself is merely a container for its layers. All drawings must have at least one layer to have any content (at least this is true as long as we are not discussing subclassing DKDrawing).

    The drawing object represents a single drawing with all of its content. Typically an application might associate each document with a single drawing, though it is not required to do so - because DKDrawing is a separate object you could have more than one per document, or share a drawing among several though this would be unusual. A drawing can be viewed through as many views as you wish. A view can display the drawing content, or it might simply present a user interface to some aspect of the drawing - a list of its layers, for example. In any case, each view will be associated with a suitable controller, and the drawing maintains a list of all of the controllers that work with that particular drawing. The drawing owns the controllers while they exist, and each controller is associated with exactly one view.

    - ☐ The layer containment hierarchy

      Layers, based on the DKLayer semi-abstract base class, are organised into hierarchical groups. DKLayer is an object that provides support for drawing content on demand, and possibly providing some basic mouse event handling. In DrawKit, everything you see in a drawing is drawn in a layer, including any grid or guides. Layers have a front-to-back (Z) ordering that you can change at will. This means that whether a grid is placed in front of or behind other content is your simple choice - just put the layers in the order you require.

      Layers are hierarchical. They can be, but are not required to be, organised into related groups. DKLayerGroup is a DKLayer subclass that can contain any number of other layers. Layers that are grouped together can be hidden or shown as a group for example. However, even grouped layers share the root drawing's overall size. DKLayerGroup provides methods for (undoably) changing the order of the layers that it immediately contains.

      DKDrawing is itself a subclass of DKLayerGroup, because it is at the root of the layer containment hierarchy.

      Principal properties of DKLayer include whether it is visible or not, locked (locked layers cannot have their content edited), whether the layer is included in printed output, its name and some properties that affect the appearance of selected objects in that layer.

      In some respects layers can be though of a little like views - they draw stuff when requested and mouse events are directed to them.

      - ☐ DKLayer

        DKLayer is the semi-abstract base class for all layers. It provides some basic state variables common to all layers, and numerous useful methods for updating part of themselves (via the drawing, its controllers and views), handling basic mouse events, and other utility methods. The properties common to all layers are:

        • its name (visible in a UI, perhaps, otherwise not used in DrawKit)
        • whether it is visible or not
        • whether it is locked or not
        • what group it belongs to
        • the "selection" colour value
        • the DKKnob helper class that draws the selection handles for objects
        • arbitrary metadata attached by the user

        Useful utility methods that can be used by all layers include:

        • getting informed when the layer is made active or inactive
        • displaying a small information window near the mouse point with some value of your choice
        • updating itself or any part of itself
        • supporting a general drag/drop functionality
        • supporting contextual menus

        One problem facing designers of applications that have multiple layers is making sure the user is clear about which layer is active. One way to help distinguish layers is by the use of colour, so the "selection" colour is provided by DKLayer to assist. Primarily intended for showing selection highlights in object layers, it is available to all layers. Classes such as DKGuideLayer use this as a default colour for its guides, for example. A simple mechanism is used to initially assign a different colour to each layer as it is initialized, but of course you can set it to whatever you like. The same colour is used as a background to the information window, again reinforcing which layer the information originates from. The DK demo application has a user interface for setting this colour directly, in its layers palette.

        The information window is a handy feature that can be used to help supply direct feedback for some kinds of operations. For example when an object is resized the info window is used to display its current width and height. In general this tooltip-like window should be used for numeric information that takes up one, or at most two lines. As it is displayed in front of everything, it must not be so large as to obscure the content. Using the info window is easy - simply supply it with a string and a position in local coordinates, and DKLayer will do the rest. When you are finished with it, ask the layer to

Topic

hide it.

DKLayer is able to respond to mouse events originating in a view and passed to it when it is the active layer. This passing on of events is performed by DKViewController. In general, layers should be designed to respond to their own mouse events only if their needs are simple and easily handled in a self-contained manner. So DKGuideLayer implements these methods for dragging guides, but the much more complex requirements of selecting and manipulating objects in a DKObjectDrawingLayer is handled by a variety of different tool objects instead.

Layers can be locked, which prevents their content being changed. Subclasses of DKLayer are responsible for checking and honouring this state to ensure that locking is consistent. Likewise, hidden layers should not be edited either, as the results cannot be seen and so the user should be gently prevented from giving themselves a nasty surprise. Hidden layers are automatically not drawn, but subclasses of DKLayer need to check for this state to prevent editing. The method -isLockedOrHidden usefully covers both states that should disallow editing.

Layers may or may not be required to appear in printed output. "Structural" layers such as DKGuideLayer probably shouldn't be, whereas of course layers with actual content should be. This can be easily set using the -setShouldDrawToPrinter: method. Subclasses usually set this to some appropriate default themselves.

A layer's name can be a useful way in a user interface to tell layers apart. DKLayer retains a name, and all layer subclasses set this to some useful default, but DrawKit itself does not use or interpret the name - it is entirely for the benefit of your user interface.

- ☐ DKLayerGroup

  DKLayerGroup is a layer subclass that can contain other layers, including other groups. It has no function other than as a container for layers, but is responsible for handling the drawing of the layers it contains in the correct order, and for implementing the Z-order altering commands and methods. DKDrawing inherits from this class, being the root of the layer tree.

  Layers are stored in an ordered array such that index #0 is the top layer, and index #(count -1) is the bottom layer. While accessing layers by index is occasionally necessary, your application should avoid depending on layer stacking order where possible. To help in this respect, there are convenient methods such as -topToBottomEnumerator which will return an enumerator for iterating over the layers in the specified order.

  One reason that layers are stored in the order stated above (and in this respect they happen to differ from the order that drawable objects are stacked within a layer), is to permit a user-interface such as one based on NSTableView to display layers naturally, that is, with the topmost layer at the top of the list, with no special effort. Earlier versions of DrawKit (prior to beta3) would cause such tables to appear upside-down unless they were coded to compensate. DKLayerGroup is able to detect and automatically reverse layer stacks in archives saved prior to this change.

  Layer groups, being layers, are able to be locked and hidden, and when locked, changing the Z-order of layers, or adding and deleting layers is disabled. If a drawing as a whole is locked, the active layer can't be changed either.

- ☐ DKDrawing - the root of the layer tree

  DKDrawing is a subclass of DKLayerGroup, because it contains layers. The only drawing done by DKDrawing that does not originate in one of its contained layers is the erasure of the background or "paper" colour when drawing to the screen, which defaults to white.

  DKDrawing sits at the root of the entire drawing hierarchy, and importantly provides ownership of any number of controller objects (DKViewController or subclass) which forge the link between the drawing and its views. It also owns its layers and layer groups.

  Because DKDrawing owns the controllers, it provides a number of 'view-like' methods that are used to mark various areas of the drawing for update. In turn these methods pass that update request back via the controllers to the views that display the drawing. Since every layer maintains a back-reference to the drawing it is easy for any object to mark its own area as needing update whenever needed. While DKDrawing supports a general -setNeedsDisplay: method (like NSView), for performance reasons it is important to only invalidate the smallest possible area that you can get away with. For individual DKDrawableObject instances, this is usually accomplished using its -notifyVisualChange method.

  A key property of DKDrawing is the "active" layer - this is a layer nominated as the target for the user interface and user input events. The active layer can be any layer that does not refuse the active layer status.

  DKDrawing also provides a central place for handling snapping to grid and so forth. Because the grid is itself implemented as a layer, most objects will find it much more convenient to use their back-references to the drawing to handle grid snapping, rather than having to locate the grid themselves and work with it directly. This indirection also allows the grid behaviour to be substantially altered if required without breaking the protocols of existing objects.

  Finally, DKDrawing provides a convenient place to move drawing data into and out of the model, in a variety of forms. These data handling methods are the basis for saving drawings to disk, etc.

  DKDrawing is the object that provides a connection between the DrawKit system and the undo manager in use. At some point you need to tell DKDrawing which undo manager it should direct all of its undo tasks to - this could come from a document (typically) or perhaps a view. Without an undo manager, nothing in DrawKit itself can be undoable. Your application is required to supply DKDrawing with its undo manager - though this is done for you by DKDrawingDocument and when a view automatically creates a drawing.

- ☐ The active layer

  In order to make sense of a layered drawing and to provide a context for user input, DrawKit supports one active layer per drawing at a time. DKDrawing is the object that maintains the active layer reference - other objects such as DKViewController also provide convenient access to this value.

  The active layer is one nominated layer to which all input is directed. The input may come from menu commands, the keyboard or the mouse being used in a view, or items dragged into a view. For many simple kinds of layers, a layer is able to handle this input directly without the intervention of a controller - in fact by default DKViewController merely forwards all such events to the current active layer without interpretation. The active layer is analogous within the DrawKit system to the active window within an application (or entire computer desktop) as a whole - it provides a context for the focus of a user's current work.

  Input events come from the key view to the active layer through the view's controller by way of invocation forwarding (except mouse events, which are forwarded directly). This allows a layer to simply implement whatever commands it wants to respond to directly as if it were itself a subclass of NSResponder and part of the responder chain. Effectively the active layer is automatically a delegate of the key view. Object layers extend this forwarding to the selection and even individual target objects. This general approach means that DrawKit is able to offer many commands and features at a variety of different levels that an application can take advantage of (or choose not to, simply by ignoring their presence). For example if your application wishes to implement commands such as Move Layer Forward or Move Layer Backward, it can simply make the appropriate methods available to 1st Responder in Interface Builder and hook up menu items to them - the forwarding mechanism originating in DKDrawingView will see to it that they "just work" as expected.

  The current active layer needs to be set appropriately. If your application has many potential active layers, it will need to arrange a user interface to select which one is active. Since the current active layer establishes so much of the basic context of DrawKit, it is important that a UI makes the active layer reasonably obvious to the user. Alternatively a simple application might only have one layer, or not allow it to be changed at all. The active layer can also be selected automatically by a hit being detected in the layer. DKViewController implements this automatic switching, and also allows it to be disabled. The layer itself is required to return whether it was hit or not - typically for object layers this means whether any contained object was hit.

  A layer may refuse to become the active layer by returning NO to -layerMayBecomeActive. This response is always honoured, whether switching the active layer through DKDrawing's -setActiveLayer: method, or automatically when the mouse hits the layer. In addition a layer can get first shot at the mouseDown that might cause automatic layer activation by overriding -shouldAutoActivateWithEvent: and returning YES or NO. This is addition to the hit test itself, which will have already been performed and found to have hit something.

Topic

Note that the active layer is expected to be an individual layer, in general. It is theoretically possible to make a layer group the active layer, but unless that group is a special subclass that implements some particular behaviour, it is not going to do anything useful. Your application's UI should generally prevent the user from making a layer group active unless it has a special reason to allow it. By default, DKLayerGroup always refuses the active status.

- ☐ Object Layers

  Object layers are layers that contain graphical objects that can be operated on individually, as distinct from layers that present some fixed type of content such as the grid. Most of the "interesting" content (user created) in a DrawKit application is likely to live in an object layer.

  - ☐ Object Owning Layers and Drawables

    Object layers are functionally split into two classes - DKObjectOwnerLayer and DKObjectDrawingLayer, which subclasses it.

    The names may be slightly misleading, as a DKObjectOwnerLayer is perfectly able to draw the objects it owns. The name is meant to reflect the main functional purpose of the class - that is, to own objects. In DrawKit, the term 'object' is rather ambiguous and open to misinterpretation, so a graphical object that has its own distinct identity is called a "drawable". Every single shape, path or other distinct selectable item in DrawKit is a subclass of the semi-abstract drawable base class, DKDrawableObject.

    DKObjectOwnerLayer provides the basic ownership of drawables, and is responsible for maintaining them as a related set. It also deals with the front-to-back (Z) ordering of the drawables it owns, and provides methods for changing these. Like DKLayerGroup, it provides user-action methods that can be simply hooked to menus to provide commands such as Move To Front, Move Backwards, etc.

    DKObjectDrawingLayer is a subclass of DKObjectOwnerLayer that brings the concept of a selection into the picture. The reason for the functional split is twofold: first, one of convenience in that even with the split, both are quite large classes in terms of number of methods, so this keeps it manageable. Second, it permits a DrawKit developer to subclass at either level if they wish, giving a more fine-grained opportunity to do this.

    DKObjectOwnerLayer undoably supports the adding and removing of objects to the layer, changing their Z-order and other basic operations that do not require the concept of a "selection". It also handles the essential requirements of handling drags of external data into the layer.

    - ☐ Object Storage

      DKObjectOwnerLayer abstracts the actual storage of the drawables it owns into a further class, based on the DKObjectStorage formal protocol. Concrete objects that implement this protocol currently are DKLinearObjectStorage and DKBSPObjectStorage. The storage abstraction is entirely transparent to applications, but the ability to use different types of storage can have useful advantages for the programmer.

      The storage class for new instances of DKObjectOwnerLayer can be set using +setStorageClass: The default is DKLInearObjectStorage.

      The layer will return, on demand, a list of objects to be drawn based on a rect or a view needing update. This list is built by the storage using whatever internal algorithm it implements. The result is always the same - a list of objects to be drawn, in bottom-to-top order.

      - ☐ DKLinearObjectStorage

        Linear storage is very simple and reliable - all objects are simply kept in a single list (array). Such an array strictly defines the back-to-front ordering of objects (Z-ordering). For many uses, linear storage is entirely adequate, providing good performance up to a few hundred objects per layer. Conceptually, the indexed nature of linear storage is what all storage "looks like" to client code (such as DKObjectOwnerLayer).

      - ☐ DKBSPObjectStorage

        The drawback of linear storage starts to become noticeable when the data sets become much larger than a few hundred objects. While DrawKit always avoids drawing objects that it doesn't need to draw by carefully using NSView's update rects mechanism, when data sets are large there is still the issue of having to iterate over the entire objects list to determine which objects should be drawn.

        BSP (Binary Search Partition) Storage improves performance on larger data sets by avoiding the need to iterate all objects to find those that need to be drawn. BSP storage uses a tree structure which conceptually repeatedly subdivides the overall drawing space into smaller and smaller binary partitions. When an area needs updating, the tree is used to rapidly determine which partitions and hence which objects are affected. Thus while the returned list of objects to be drawn is the same as for the linear storage, the time to build that list can be much less. In theory data sets having hundreds of thousands of objects should be possible, and performance will only be dictated by the objects that are currently visible.

        DKBSPObjectStorage subclasses DKLinearObjectStorage so objects are still stored in a single array, but the BSP tree is used to efficiently index it on a spatial basis. Z-ordering is thus strictly defined as for the linear case. DK's implementation dynamically sizes the BSP Tree as needed to maintain efficiency as objects are added and removed, though the programmer has the option to set a fixed-size tree if they wish (Note - a fixed-size tree can still store any number of objects, but efficiency may decline if the number of objects greatly exceeds the optimal tree depth).

      - ☐ DKRStarTreeObjectStorage

        R-Trees are another way to spatially index objects to improve efficiency when dealing with very large data sets. R*-Trees are able to efficiently store millions of objects. This class is currently under development.*

    - ☐ The Selection

      DKObjectDrawingLayer is a subclass of DKObjectOwnerLayer that supports a selection. The selection is simply a set (literally an NSSet) of objects that are "selected", that is they are a member of the set. When drawables are drawn, their membership of this set is passed along as a boolean parameter so that the object is able to draw itself in a way that gives the user appropriate feedback that the object is indeed selected. For example shapes have "handles" (also known as "knobs") arranged around their bounding rectangles, and paths have draggable control points.

      The selection permits objects to be more finely targeted for input. Some commands for example are able to operate on all of the objects in the selection. Some commands need to target one single unambiguous object. DKObjectDrawingLayer provides support for doing both of these things - in the first case by implementing numerous "selection targeted" commands of its own, and in the second case by automatically forwarding messages it cannot handle itself to a single selected object which is able to respond. Once again this permits individual drawable objects to implement actions and methods as if it were an NSResponder in the responder chain.*

      **Note: some specialised commands may operate on specific numbers of objects, typically two - for example joining paths.*

      Because DKObjectDrawingLayer is able to provide so many "selection targeted" actions, for ease of maintenance and understanding they have been split across several categories:

      DKObjectDrawingLayer+Alignment - handles a host of alignment actions
      DKObjectDrawingLayer+BooleanOps - handles high-level boolean (set) operations between multiple objects (using the optional third-party GPC library)
      DKObjectDrawingLayer+Duplication - handles a number of duplication actions

      The main class implements high-level actions for cut, copy, paste, delete, move (by keyboard), Z-ordering, grouping and ungrouping, show, hide lock and unlock among others. Your app is free to ignore them if it doesn't need them or use them to get many useful features for next to no effort. All actions are undoable.

      To support drag and drop, DKObjectDrawingLayer allows the object under the mouse location to be selected dynamically during a drag, if the object is able to receive whatever it is that is being dragged. The coordination of this is fairly complex but the upshot is that objects can be dragged into the layer, or in many cases into a drawable object within the layer. A drawable that receives the drag is itself responsible for accepting and ultimately handling the data that is dropped.

Topic

DKObjectDrawingLayer can be set either to treat selection changes as undoable actions in their own right, or not (and so selection changes are only undone as part of some other operation). Different applications will take different views on this.

- ☐ DKKnob

  DKKnob is a small helper class that is responsible for drawing the knobs or "handles" shown on selected paths and shapes. This helper object can be owned by a layer, or (by default, and because it subclasses DKLayer) the drawing. Showing the selected state of an object involves several steps - first, the object is added to the selection set in DKObjectDrawingLayer. Note that being "selected" means membership of this set, and nothing else - the state of the object itself does not change. However, an object is able to get notified when its selected "state" changes, and can also query this at any time.

  When the owning layer draws the object, the selected state is passed as a boolean flag. The object responds by making additional drawing calls to display this state. This will usually involve making use of a DKKnob instance to perform the actual knob drawing. The basic highlight colour for the selection is also supplied by the layer. The involvement of the layer and DKKnob is to provide a consistent appearance for selections, but is also a place that is considered an early target for customisation.

  DKKnob is asked to draw a knob of a given type at a certain point, does so, and returns. The knob "type" is a purely logical classification that DKKnob can use to choose one of several appearances for the knob. It is not in itself a specific appearance - a DKKnob subclass may decide to render all knob types the same for example. Knob types are defined in DKCommonTypes.h As a convenience, you can also pass some extra flags along with the knob type to indicate a locked object, or a disabled object for example. DKKnob is responsible for the interpretation of these flags and turning them into distinct visual renderings.

  Note that clients of DKKnob (drawable objects) should not try to force DKKnob to draw one way or another. The point is to allow DKKnob to provide a consistent selection appearance when given any and all objects. Since DKKnob is drawing UI-related information (it is not part of the data model), it needs to take into account two aspects of the application's UI - the view's zoom scale and the view's window's active state. It does this by querying these via a simple formal protocol implemented by its owner - typically a DKLayer. The view's scale is used to compensate the knob's size for the zoom. By default, DKKnob does not cancel the zoom exactly - it allows a small amount of growth in proportion to the zoom (it grows about a third as fast) which gives better usability - the knobs grow, but not so large as to obliterate the content at large scales.

- ☐ Kinds of Drawables

  All drawn objects inherit from DKDrawableObject. This is a semi-abstract base class that provides an informal protocol that all drawables are expected to comply with. Drawables are responsible for drawing themselves on demand as well as implementing the lowest level of mouse event handling required to edit themselves interactively in a sensible fashion.

  In general, applications can extend the features of DKDrawableObjects in two ways - at compile time or at runtime. Compile time extension is a classic case of subclassing DKDrawableObject or one of its existing subclasses. You would do this if you have particular needs that the standard objects do not cover. Runtime customisation is easier but of course more limited. This involves creating the styles and geometry of the paths and shapes you want using combinations of DrawKit's existing objects.

  Essential properties shared by all drawables are:

  • Its bounds rect. This is a rectangular area that by definition contains all of the drawing done by the object. It will occasionally extend well beyond the obvious visible edge of the object, but for efficiency should not be made excessively large. Drawables are responsible for calculating this bounds rect and not drawing outside it.
  • Its position. A drawable has a definite location within the overall drawing, specified in drawing (Quartz) coordinates. The location can be defined by the object to be anywhere relative to its bounds - for example paths use their top, left point whereas shapes use their centre point, plus some variable offset.
  • Its angle - some types don't have an angle so must always return 0. For types that do, this represents the rotation of the object about some point (typically its location).
  • Its size - a width and height oriented in the direction of the angle.
  • Whether the object is visible (drawn) or not, and whether the object is locked or not (editable).
  • Its geometry - usually specified in terms of an owned NSBezierPath object.
  • Its style - an object's style is responsible for its actual appearance (strokes, fills and other rasterizations).
  • Its metadata - an attached optional dictionary of values. DrawKit is able to use and set some metadata itself but generally this is for application use.

  DKDrawableObject supports two built-in user actions - copy and paste of the attached style.

  DKDrawableObject provides many methods that are of general utility to all concrete subclasses, as well as a number of informal protocols (and stub methods that are part of these). A drawable is always required to draw wholly within the area defined by its -bounds method. It should calculate and return this region taking into account all possible graphical adornment that can be applied to the object. Thus the object's style contributes significantly to this calculation. Note that DrawKit doesn't enforce this region by clipping to it when drawing. This is done for performance reasons, since the need to save and restore the graphics context and text the clipping path can slow things down. As a result, if you do draw outside the bounds, trails of unerased pixels might be left when the object is moved or changed.

  For best performance, the bounds should be kept as tight to the object as possible. When an object needs to be redrawn for any reason, its -notifyVisualChange method is called. This invalidates the object's bounds in all views that are currently displaying it. The resulting areas are repainted on the next event cycle (Cocoa coalesces all such update requests into a single update, and strictly limits drawing to these ares when repainting). For most typical operations on a drawable, -notifyVisualChange is called as necessary - you only need to call it if you need to force an update outside of changes made through the usual methods.

  DrawKit's built-in concrete subclasses of DKDrawableObject fall roughly into two kinds - paths (DKDrawablePath) and shapes (DKDrawableShape).

- ☐ Basic Shapes and Paths
  - ☐ DKDrawablePath

    DKDrawablePath is a concrete subclass of DKDrawableObject that draws a path. Its path is stored at the final size and position in the drawing that it occupies - there is no transformation done on the path at the time it is drawn. Paths are designed to be editable at the path control point level, so when selected these objects display draggable handles for every control point in the path.

    Paths are best for objects that are long and thin, or have an irregular outline. Paths can be created in a number of "modes", which map to different kinds of typical creation tools - for example, bezier curves, straight lines, polygons, ars and wedges and freehand. Once created all these paths are edited in the same way, by dragging their control points. Thus an arc for example doesn't "know" its an arc once created, and will lose its shape if edited.

    Path objects have no concept of overall rotation and always return an angle of zero.

    Paths are drawn by their associated style objects exactly the same as for shapes.
  - ☐ DKDrawableShape

    DKDrawableShape is a concrete subclass of DKDrawableObject that draws a geometric shape defined by a bounding box. A selected shape features (as standard) eight "handles" or "knobs" arranged at the corners and mid-points of the box. Dragging the knobs changes the size of the box and the path of the shape is scaled to fit within.

    Shapes also feature (as standard) a rotation knob which allows the object's angle to be simply dragged to a new value, and a centre position which sets the centre of rotation and origin for the object. This means that you are not required to have a separate "rotate" tool though DrawKit

Topic

would certainly permit this approach if you prefer.

Shapes are best suited for objects that can be defined by simple scaling of a path to fit the bounding box - an irregular path can certainly be set and scaled to fit, but the detail of the path cannot be changed. However, DKDrawablePath and DKDrawableShape are freely interconvertible to each other with no loss of data (except rotation angle), so in fact you can simply convert to the other type and edit how you wish.

Shapes store their paths based on a unit square centred at the origin (i.e. a square 1.0 points wide and high, with its centre point at 0,0) and transform the path at that size to the final position, size and rotation angle when drawn. A shape's path is drawn by its associated style object.

- ☐ DKReshapableShape

  DKReshapableShape is a simple subclass of DKDrawableShape that provides an opportunity for the path to be recomputed whenever the object's size changes. It is still best suited for geometric shapes but provides more flexibility - for example, a round-cornered rectangle will usually want to maintain a constant corner radius even as the overall shape is resized.

  This object uses a helper object, an instance of DKShapeFactory to supply it with a new path on demand.

- ☐ Image Shapes

  DKImageShape is a subclass of DKDrawableShape that displays an image in addition to any stylistic properties it has. The image is displayed within the bounding rectangle of the shape, oriented to its angle. This class provides basic image support within DrawKit, though for another way to use images, you can also add a DKImageAdornment to any object''s style.

  If you drag an image file into a DKObjectDrawingLayer, it will create a DKImageShape to display it by default.

  A DKImageShape has a style as normal, which can be shown in front of or behind the image. For example you could set up a style with a stroke to act as a frame or border for the image, or add text, etc.

- ☐ Image vectorisation

  DrawKit is able, using the third-party potrace code library(available under a separate license), to turn bitmap images into vector paths. This operates at a number of levels, from the very high to lower methods offering finer control. The vectorisation is implemented across a number of classes, the highest level being the DKImageShape+Vectorisation category. With a little set up (or just using th edefaults), a user is able to simply invoke the -vectorize: action and have the image instantly converted to a group of paths in place.

  The settings available at this level include the number of colours or greyscales, and the tracing parameters to pass to the lower level vectorisation code. The results are returned as a DKShapeGroup containing a collection of DKDrawableShape objects having the vectorized paths set. The number of shapes will correspond to the number of colours or greyscales requested, and each is set up with a style matching the original colour in the image.

  Vectorisation results will vary greatly depending on the source image. Very noisy images or those containing ill-defined edges may not work well, whereas line-art and high-contrast images usually will.

  The lower level code that handles vectorisation is of course potrace itself, and the NSImage+Tracing category. This also specifies the DKImageVectorRep class that is used during the conversion. DKColourQuantizer is another small collection of classes that are an important part of the conversion process.

  The conversion process is:

  1. First the image is rendered as a 24-bit bitmap regardless of its original format. This permits all images to be converted by using this common format.
  2. The image is colour quantized. That is, the most important distinct colours in the image up to the number requested are determined. Colour quantizing is quite a complex business, since you want to determine a set which covers the most important colours in an image as perceived by the viewer. DrawKit currently uses an octree quantizer which is widely considered to be one of the best algorithms.
  3. Based on the colours determined in 2, the image is separated into "bitplanes", one for each separate colour. The bitplanes are 1-bit images, having the bit set where the pixel matches the colour, and not set everywhere else.
  4. Any empty bitplanes are thrown away, as they contain nothing which will contribute to the final image.
  5. Each bitplane is stored, along with its associated colour, in a DKImageVectorRep object. A list of these objects is returned as the output from this first phase of the conversion.
  6. The next phase begins by asking each DKImageVectorRep for its vector path. This is created on demand by using potrace to trace the bit image using the given parameters and return it. This is then converted into an NSBezierPath.
  7. If step 6 produced anything, the path is used to initialise a DKDrawableShape object. The original colour is used to create a fill style which is attached to the shape.
  8. All shapes from all the bitplanes are collected into a group object and returned.
  9. The group replaces the original image in the layer - for the user, the conversion has been done "in place".

  Note that the process is complex, and for large images may take time to run. The resulting paths may also be complex and be slow to draw.

- ☐ Text Shapes

  DKTextShape is a DKDrawableShape subclass that is able to display a block of text. The text can have attributes applied to it as a single block either directly or through the attached style. Text can be editied by double-clicking it which creates a temporary editor. Text shapes respond to the font panel and other commands in the Text/Format menu directly (as long as they are not linked to a locked style).

  Another (often more flexible) way to display text in a DrawKit drawing is to use a DKTextAdornment as part of an object's style.

  If you drag text into a DKObjectDrawingLayer, it will create a DKTextShape to display it by default.

- ☐ Groups

  DKShapeGroup is the basis for grouping objects in DrawKit. It inherits from DKDrawableShape so that it gains all the usual interactive editing features such as resizing and rotation, but its content is of course other DKDrawableObjects. A group can contain any other kind of drawable. When objects are grouped, the nested transforms of each group (if more than one) are taken into account in order to render the content correctly. Thus objects can be scaled and rotated within groups and their paths will be transformed as needed. When ungrouped, such changes are made permanent by adjusting the paths of each object.

  Note that the appearance of objects is not necessarily subject to these transformations - so for example a stroke width of 6 points on an object will remain 6 points even if the group is resized dramatically. An alternative is for groups to visually transform their contents, which will scale all the stylistic variation as well. However when ungrouping, work needs to be done to make these style changes permanent.

- ☐ Metadata

  Every drawable object in Drawkit supports the attachment of arbitrary metadata (also known as "user data"). Typically this metadata will be in the form of an NSDictionary, allowing you to store any data you like with a given key, attached to a drawable object. While the metadata object is defined as type id, meaning you can attach anything you like, Drawkit itself does pre-attach some metadata to some objects and it always does so using an NSDictionary, so for maximal compatibility, it is best to do the same - and in fact using a dictionary has numerous advantages. An object's metadata is saved with it in a file, and in general always remains attached and available unless you go out of your way to prevent this.

  Some rasterizers are able to lookup metadata values using a key (thus assuming that metadata is indeed stored in a dictionary). For example, both DKTextAdornment and DKImageAdornment rasterizers are able to look at the object's metadata for specific keyed values (in the first case string data, in the second, images). This feature permits you to define generic styles that nevertheless are able to display data differently for each object they are attached to even when the style is shared. Taking this one step further, these rasterizers (or indeed any client code of the metadata APIs) is able to introspect an object itself using Cocoa's KVC mechanism. To do this, the identifier (key) for a metadata item is prefixed with a '$' symbol. When the API sees this prefix, it knows to strip it off and use the remainder of the identifier as a keypath to a direct property of

Topic

the object. So for example an object can display its own location using an identifier of $location.

To ensure that DKTextAdornment can work with many different possible object values, DrawKit implements -stringValue for many common objects including NSArray, NSSet, NSDictionary, etc. This is very similar in concept to Cocoa's -description method, but formats the result more usefully for user-oriented (rather than programmer-oriented) display. NSObject implements -stringValue to returns its class as a string, so every object will return something displayable.

Some drawables, in particular DKImageShape, may set some metadata items when they are initialised. These pertain to the original path, filename and dimesions of the image used to initialise the object (if these values can indeed be determined). Clients can make use of these metadata values as they wish, or ignore them. Another object that can set metadata is DKTextShape, when it is converted to another kind of drawable. The original text is saved as a metadata item which could permit application code to still find such objects using a text search even though the text is now rendered purely as a graphic.

Build-in metadata items are defined for the following keys:

- ☐ Specialised Layers
  - ☐ The Grid Layer

    The Grid Layer is responsible for several things - a) drawing the squared grid for a drawing, b) establishing "snap to grid" fundamental manipulation of points and rects, c) managing the ruler setup and d) establishing the "real world" coordinate system of a drawing. It is implemented as a layer so that it can be placed anywhere relative to your other content - behind, in front, it's up to you. By default the grid draws a three-part grid designed to simulate common real-world graph paper, using transparency and very fine lines to avoid it "filling up" the display too much. Previously, most drawing programs have had to compromise by drawing unrealistic grids simply because the display resolution was not up to anything better.

    Grids are draw in three parts, corresponding to the three types of "squares" that make up the grid. The fundamental grid unit is called its span. This is divided into a whole number of divisions which are the smallest squares in the grid. A whole number of spans makes up a major interval. Thus on a typical metric grid (in fact the default metric grid) the span is set to 1cm, divided into 5 parts (thus each is 2mm square) and there is a major interval every 10 spans, or 10cm. Each part of the grid can be drawn using a different colour, though usually you will probably set a single "theme" colour from which all three are derived.

    The grid's span establishes the basic conversion between the underlying Quartz drawing coordinates and whatever "real world" coordinate system you define. The units can be whatever you want - centimetres, metres, kilometres, miles, light-years... While DrawKit always works in the underlying coordinate system of Quartz points, the user of an application is most likely to want to work in some real system - DKGridLayer is responsible for that mapping. DKDrawing provides a number of convenient methods to access some of the grid's features, since it helps isolate you from the grid's position and even its existence (which is optional - some drawing systems will prefer to simply do without a grid).

    Snapping points and rectangles to the grid is also performed by fundamental methods in DKGridLayer, but again, you'll probably access these through DKDrawing or even DKDrawableObject itself, which provides a higher-level interface. DKGridLayer's most basic snapping method is -nearestGridIntersectionToPoint:, which given any point in Quartz coordinates, returns the nearest grid point (based on the divisions). You can also convert any length to and from the grid's coordinate system easily.

    DKGridLayer only supports square grids, where the span in the horizontal and vertical dimensions are equal. For alternative grids (polar, for example, or other non-square grid) you would need to subclass DKGridLayer.

  - ☐ Guides

    Guides are implemented using DKGuideLayer. Again, the use of a separate layer allows you to easily place the layer relative to your other content (Z-order) as you wish. You can have any number of guides, positioned anywhere. Guides are typically set up so that they have a slightly stronger "pull" for snapping than the grid does, so guides will tend to have a more obvious snapping action. However, DrawKit will in fact use the nearest grid or guide to a given point, so guides should not prevent you from positioning objects to either.

    DrawKit provides the convenient way to create guides common to many graphics applications of simply dragging a new guide off one of the rulers. This works whether or not the guide layer is active. However, repositioning an existing guide requires that the layer be first made active. You can also snap a guide to the grid when dragging it by holding the shift key, even if the grid snapping is otherwise turned off. To remove a guide, drag it out of the interior area of the drawing and into a margin - that will delete it.

    Each guide can have its own colour, though by default they are initially set up to take their colour from the layer's setting. If this is changed, all guides will be updated to use the new colour.

    Generally DrawKit is designed to have a single guide layer. If you want to change the guide set according to the active layer or some other state change, you will need to arrange this. The -guides and -setGuides: methods allow you to set multiple guides in one call.

  - ☐ Image overlays

    DKImageOverlayLayer is designed to display a single image in a layer, possibly tiled to fill it. It can have its transparency set. Its purpose is to allow images to be imported for tracing for example - it plays a very minor role in DK as a whole.

  - ☐ Drawing Information Layer

    DKDrawingInfoLayer is a layer class which displays a single panel of information in a nominated corner of the drawing. By default it uses some of the metadata of DKDrawing to display a drawing number, the name of the draughtsperson, the date and so forth. If you have a use for this, you'll probably have your own scheme for drawing numbers, etc, and so this layer is really just a demonstration of how that can work.

- ☐ View and Tool Controllers

  DrawKit's controllers sit between a drawing and its views. The controllers are owned by the drawing; the views are owned by their superviews and ultimately their windows. In DrawKit, there is one controller per view.

  The basic controller, DKViewController, provides basic support for forwarding input events from the view to the active layer within the drawing, and for responding to requests from the drawing to update the view. DKToolController, a subclass of DKViewController, adds the concept of a settable tool, allowing the user to interactively create, select and edit objects.

  - ☐ The view controller and the flow of events

    Initially a user's inputs are all directed to the key view, which is the first responder in the responder chain. These user inputs can be mouse clicks and drags, selecting commands from menus, typing on the keyboard, or dragging data items into the view. DKDrawingView forwards all messages it cannot respond to to its controller, including forwarding all mouse and flagsChanged events.

    DKViewController basically forwards everything on again, this time to the active layer as determined by querying the DKDrawing instance that the controller belongs to. However it does intercept the Layer Z-ordering action messages and invokes the relevant methods on DKDrawing to handle them, as well as providing an optional "click to activate" methodology for setting the active layer. DKViewController also implements autoscrolling of a mouse dragged event using a timer.

    Depending on the class of the current active layer, the forwarded messages and mouse events may be handled or forwarded again to "the selection" or a single selected object.

    When a DKToolController is used in place of a DKViewController (which is usually the case), the currently set tool acts as a filter for mouse events such that the basic events are translated into meaningful operations that can be applied to the active layer or objects within it. What happens depends on the class of tool set, and DKToolController merely coordinates the calling of the various methods that the tool protocol provides. The tool itself performs its operations on the active layer according to its design.

Topic

Typically a tool will target a specific object and pass events or messages to it to get its job done. Tools are generally quite lightweight in that they don't do a lot of heavy lifting on behalf of objects - they identify the target object and then mostly simply pass on events to it in the proper sequence.

- ☐ The basics of object editing

  By and large, drawable objects are able to edit themselves. This is sensible as it avoids the need to have lots of different controller classes with overlapping responsibilities just to deal with the object variations. Instead, objects are given fairly low-level events such as mouse dragged, and given the context of where the mouse is within the object, do the expected thing.

  Drawables typically have numerous places or clickable regions within them that are sensitive to the mouse and have specific behaviours when clicked or dragged. For example a path's every control point is draggable, and each one will have a slightly different effect on the path. A shape's handles or knobs each resize the shape in a slightly different way. In order for a drawable object to tell what the user clicked, it assigns a "partcode" to every single separately sensitive place. A partcode is just a number (int) which identifies the point clicked - it is entirely private to the object and is only interpreted by it.

  Two partcodes are reserved and are interpreted outside of an individual drawable instance - that of 0 meaning "no part was hit" and -1 meaning "the entire object was hit" or "no special partcode was hit".

  When the mouse goes down initially in an object, the partcode of the hit is determined by the target object itself, and returned to the caller (the tool or tool controller, for example). If the partcode is anything other than 0 or -1, it is known to be private to the object so all that will happen is that the same value is passed back in subsequent mouse-dragged and mouse-up handlers. 0 or -1 values might be interpreted. For example -1 might trigger a move operation, and 0 might trigger the start of a drag selection operation.

  If the target object is passed back the partcode in a mouse drag call, it alone knows what to do with it. For example, a shape might identify the partcode as meaning the rotation knob was hit, so it carries out a rotation operation based on the current mouse point.

- ☐ Tools and the tool controller

  In a typical drawing application, the user is often presented with tools as a set of buttons in a palette - clicking the button selects that tool, putting the application into a mode where mouse activity achieves certain things. While DrawKit certainly has something like this kind of user interface in mind, its definition of a tool is more abstract and not tied to any particular UI.

  In DrawKit, a tool is an object that converts mouse events into meaningful actions and behaviours targeted at the active layer or objects within it (or perhaps both, as with the select and edit tool). The tool thus translates the general mouse gestures into specific actions. Because drawables are responsible for the nuts and bolts of their own editing, in DrawKit most tools are quite lightweight objects.

  DKToolController is a view controller subclass that manages the tools and coordinates the selecting of the tool and feeding the basic mouse events to them according to the agreed protocol. Only one tool can be set at a time, since this establishes a mode of operation for mouse input, and the user can only use the mouse in one view at a time. Selecting the tool could correspond to a user selecting a button in a palette of tools - in fact implementing this kind of user interface is extremely straightforward, though any other UI that ultimately picks a tool will work equally well.

  DrawKit maintains a simple registry of tools (implemented by the DKDrawingTool class) which allows tools to be associated with a name. You can retrieve a tool by name and set it as the current tool in the tool controller. Convenient action methods in DKToolController allow you to do this very easily by hooking up a user interface element that has a -title property to the -setDrawingToolByName: action, set to the name of the desired tool. This could be a button for example (the title itself might be hidden, as with an icon button, but the title string itself is still readable as a tool name by DK's code).

  Tools might vary in their scope of operation. For example one kind of tool might operate only in one particular layer or kind of layer, another might operate in a general way. To establish this, the tool controller initially asks the current tool whether the active layer is useful to it - by returning NO the tool can prevent any further call to itself. If YES, it will subsequently receive the usual mouse event methods. This allows a tool to decide its own target scope. For example, the zoom tool accepts anything, so it works no matter what type of layer is set as the active layer, even a hidden or locked one. Conversely, object creation tools can only work when an object drawing layer is active. This test is performed at mouseDown time, so having an inappropriate active layer doesn't prevent a tool from being selected, it just stops it from receiving mouse events.

  - ☐ Creating new objects

    A single tool class - DKObjectCreationTool - is responsible for creating new objects in the active layer. The same tool class is able to make all types of drawable so a typical drawing application might have numerous instances of this registered, one for each kind of shape or path it can natively create. The reason this tool appears to be so versatile is that in fact the majority of the work it does is carried out by the object it creates itself.

    Object creation is based on copying a prototype object attached to the tool. When the tool is brought into use, the prototype object, whatever it is, is copied and becomes the new object. The prototype is copied faithfully, so you can set up the state of this object when you make the tool as part of your application's setup. For instance DKDrawablePath is the same whether it is created as a line, polygon, bezier path, arc or other form - what differs is the initial "creation mode" parameter of the prototype object. When the tool copies the prototype, the creation mode is already set and the tool allows the new object to largely handle its own creation.

    The prototype object also has an initial style, which will be copied also. However, commonly applications may wish to set the style for new objects separately from the tool itself, so DKObjectCreationTool has some handy class methods that allow you to set a style independently which will override the style from the prototype for the new copy.

    This tool works in conjunction with DKObjectOwnerLayer as well as the object it is making to coordinate the creation process. First, the new object is added to the layer only as a "pending" object. This allows it to be drawn normally but doesn't commit the layer to fully taking on the object at this stage - during creation the object is in a "probationary period". The object's class is consulted to determine what partcode should be used for the intial creation of the object - since partcodes are private to the objects only its class can supply this information. For a rectangle for example, the initial partcode is the bottom, right corner, because by convention new shapes are dragged into existence by dragging out the bottom, right corner while anchoring the top, left corner at the initial click point. So once dragging is underway, the creation tool is basically proceeding as for the edit tool, and letting the object do all its own work.

    When the user releases the mouse, the object may then become a full citizen of the target layer, provided that the object is sensible - so for example if the user didn't drag it out so that it has no width or height, it is not committed to the layer but instead merely discarded. This prevents badly-formed objects becoming part of a drawing and potentially causing problems.

    Snap To Grid is honoured for object creation just as for any editing operation.

    Path creation follows much the same sequence as shapes, except that the path "captures" the initial mouse event and for each method of path creation set by the "creation mode", keeps control in its own event loop until the user is done. This approach has the advantage that the exact mouse gestures needed for each type of path are tuned for the best user experience for each path type, and that in many cases these can involve a mouse-up or a drag without the mouse button being down without ending the creation process. Thus the mouse gestures for each type are:

    • Straight lines - click, move, click to end OR click-drag-release.
    • Irregular polygons - click, move, click for each corner, double-click or click on first point to end.
    • Freehand - click, drag, release.
    • Bezier path - click, drag and release to set control points, move end point, repeat. click on first point or double-click to end. (This is far easier to do than to explain, and is a very quick and easy way to drag out bezier curves once learned).
    • Arc and Wedge - click, move to set radius, click, move to set arc, click to finish.

  - ☐ Selection and Editing

    Selection and editing of objects is handled by a single tool, DKSelectAndEditTool. This tool is also able to move and copy objects. This tool appears to do a lot but once again, the objects themselves do most of the hard work. The large number of actions that this tool implements reflects the fact that in most applications, the "select" tool, with the plain arrow cursor, is highly context-sensitive and typically a lot less modal than other kinds of tools.

Topic

The "mode" of operation of the tool is determined from the context of the initial mouse-down click. Depending on whether or not an object was hit, and whether that hit was general in nature or hit a partcode will determine what the subsequent drag and mouse-up phases will do. The mode is set to one of selection-marquee dragging, move/copy of objects or dragging an editable knob with a distinct partcode. There is also an "invalid" mode used to prevent the tool from performing any actions if an attempt is made to use it in an inappropriate layer type.

Selection by dragging is detected by a mouse down in no object, but in the background of a layer. This starts the drag of the selection marquee or rectangle, which is drawn using a nominated style which should be (and by default is) a fairly transparent style. This is because it draws on top of all the objects so you need to be able to see them through it. You can replace the style altogether to change the appearance of the selection marquee, and it isn't required to have a fill at all - you could just have a dotted outline for example. When dragging a selection marquee, all objects touched by the marquee are selected. Drawkit optimises this operation to minimise drawing as much as it can, so as the marquee changes, only objects whose selection state actually changes get redrawn, and then only those intersecting the difference between this marquee rectangle and the previous one. This allows selection of objects to remain fast and smooth, even if they are graphically complex. Holding down the shift key extends the selection as you drag, rather than replacing it.

The other two modes of this tool also select objects initially so that the usual click-selection of objects works in the conventional manner. Thus to select an object one simply clicks it. If the tool is subsequently dragged the object is also dragged with it. The modifier keys are used to alter the selection as suggested by the Humian Interface Guidelines as follows:

+ shift - extends the selection, adding the clicked objects to the existing selection.
+ command - inverts the selected state of the clicked object, adding or removing it from the existing selection.

The tool can drag all objects in the selection or only one, depending on an option flag you set for the tool. The option is set to drag multiple objects by default. In this mode, whichever object you click to select, all others in the current selection are dragged too, maintaining their relative positions. In single object mode, only the single object actually clicked is moved. If objects are not actually moved, on mouse up the single object clicked is exclusively selected so reducing the number of mouse clicks needed to manipulate single objects. In general this tool has a lot of gestural intelligence to separate out the various operations it carries out.

When moving an object, holding down the option key makes a copy of the selection first, allowing you to rapidly duplicate objects.

In object editing mode, the tool mosty just passes on the clicks and drags to the object itself, after first setting it as the single selection. This mode is detected on mouse-down by the object returning a private partcode value to the tool. The tool does not interpret this partcode, it only knows it's not a generic value of 0 or -1, but it will faithfully pass the partcode back to the object in the subsequent drag and mouse-up phases.

- ☐ Object modifying tools

  DrawKit's tool protocol permits tools to do any kind of operation to an object if a special tool is needed. The only built-in one that is provided is the DKPathInsertDeleteTool, which as its name suggests, adds new points to an existing path or deletes existing points. It can only operate on DKDrawablePath objects, attempting to use it on any other type of object is a no-op.

  An application can define other similar types of tools if necessary, by subclassing DKDrawingTool and following the protocol's rules. Tools can be made to modify any object's attributes, copy attributes from one object to another, add objects to layers, or just do things to the user interface.

- ☐ UI Tools

  Often it is convenient for an application to have tools that do not modify the drawing itself but merely control aspects of the user interface. An obvious example is a zoom tool, which merely changes the scale (and possible the scroll position) of the view, it does not affect the content. DrawKit supports this type of tool as well - in fact DKZoomTool already supplies a tool for zooming the view. There is no compelling reason that a tool needs to work with target objects that are supplied by the tool controller - it is free to simply ignore that and do something else, which is how the zoom tool works.

- ☐ Setting the current tool

  Since DKToolController is required to set one tool at a time, at some point your application will need to consider how this is achieved. Ultimately, it's a case of calling the -setDrawingTool: method of the tool controller. The tool controller will take into account the application's setting for whether the tool applies to just this view, the document, or globally for all documents. Your application will set the "scope" setting when it starts up, and is generally expected not to change it during the lifetime of the application (this is not harmful, just potentially confusing to a user). The default scope is per-document, which is likely to be the most typically used case.

  DKToolController and DKDrawingTool together provide a multitude of ways in which a UI can easily interface to the tool selection mechanism. Which you decide to use is up to you - there's no "best" way - it will depend on your UI and application's needs.

  DKDrawingTool has class methods to support a simple registry of tools. This allows you to associate a tool with a name, and retrieve that tool by name. By default, DK pre-registers a "standard" set of tools which you can use, replace, or simply ignore. They are there merely as a convenience. By default, the registered tools use the following names:

  • Select
  • Rectangle
  • Oval
  • Round Rectangle
  • Round End Rectangle
  • Text
  • Ring
  • Path
  • Line
  • Polygon
  • Freehand
  • Arc
  • Wedge
  • Speech Balloon
  • Delete Path Point
  • Insert Path Point
  • Zoom

  Most registered tools create the objects that their name suggests. "Select" is the default select and edit tool, "Zoom" is a view zooming tool, and the path insert/delete tools do what their names suggest.

  You can replace any tool with another of the same name, or just register new tools with new names. DKToolController provides convenient methods for setting the current tool using its registered name. -setDrawingToolWithName: looks up the tool in the registry and sets it if it exists. Even more conveniently perhaps, there is also the -selectDrawingToolByName: action method. This can be targeted by any user interface object that supports the -title method (which includes NSButtons, NSMenuItems and NSCells). The title is set to the name of the registered tool, and simply by targeting that UI element on First Responder with this action, the UI will select the tool. Note - this approach is not ideal if the titles are visible to the user and you need to localise - in that case you need to register the tools using a localised name, or use a different method to select the tool.

  Another approach is to set the tool as the representedObject of the UI object that pertains to it. You can then target First Responder with the action -selectToolByRepresentedObject: and the tool will be set to the sender's representedObject, if it is indeed a tool (if not an exception is raised). This approach has the advantage of not requiring the registry nor needing to use any part of the UI element as a special field, so cannot interfere with localisation.

  If you have the DKDrawingTool object (representedObject or by any other means), a final way to set it is simply to call its -set method. This works by seeing if the current First Responder does indeed respond to the -setDrawingTool: method and passing itself. This can be extremely convenient for

Topic

many kinds of applications. It's also a very easy way to programmatically set a tool without having to worry about the view, controller or anything else - just create the tool and set it. Of course this can fail if there is not a suitable First Responder.

Tools may also be assigned a keyboard equivalent, which can include any modifier flags you wish (except command, since those are detected by Cocoa as being menu shortcuts and are not passed through the usual key handling methods). DKToolController works with DKDrawingTool to look up tools if any of their keyboard equivalents are typed, and select them accordingly. Tools must be registered for the keyboard equivalents to operate.

- ☐ The Graphical Appearance of Objects

  Objects drawn by DrawKit separate their geometry (their path or shape) from their appearance - how the path is painted. This is accomplished using the DKStyle class, which is attached to one or more objects.

  - ☐ The Drawing Pathway

    When an object needs to be drawn, the drawing will start with a call to DKDrawingView's -drawRect: method (as for any NSView). The view will start at the root DKDrawing and ask it to draw itself, which in turn will request each layer to draw itself in the right order using the -drawRect:inView: methods. When a DKObjectDrawingLayer gets a shot at the drawing, it will first weed out any objects that are outside of the update area and then request that those within it draw themselves. If the object is selected, that fact is also passed along so that the object can add the visible selection adornment.

    A DKDrawableObject is generally drawn primarily by its attached style object. The style is passed a -render: message with the object in question. The style applies each rasterizer in its list to the object's path (returned by -renderingPath) in turn. The rasterizers can fill or stroke the path, add text or do anything they like.

    If the drawable has no style at all, a default light gray stroke of the path is performed. This is mainly to prevent any such objects from getting visually "lost" in the drawing, since they are still real objects, merely lacking any other way to make themselves visible. Generally drawables should have a style attached - if you genuinely want to suppress all drawing, you can do so by attaching a style having a single fill rasterizer with clearColor. Such objects will be effectively invisible but can still be selected.

  - ☐ Styles and Rasterizers

    - ☐ DKStyle and the rasterizer tree

      Actual pixels on screen (or in printed output) are generated by DKRasterizers. These objects take the geometric path supplied by an object and use it to draw things, for example strokes and fills, to give them a visible presence. DKRasterizer is a semi-abstract base class for all rasterizers - concrete subclasses include DKStroke and DKFill among numerous others. Rasterizers are not limited to using the exact path they are given - they can use modified versions of it for special effects, and so forth.

      Rasterizers are organised into a tree, just like layers and drawables. DKRastGroup is a DKRasterizer that can contain other rasterizers. Each rasterizer is applied to the path in turn, so the order of rasterizers gives different effects - those that drawn last draw "on top" of those that drew earlier. Again this is done for maximum flexibility - the decision about drawing order is up to you, simply by arranging the rasterizers in the desired order.

      DKStyle is a subclass of DKRastGroup (just as DKDrawing is a subclass of DKLayerGroup). A style is a high-level object that can be attached directly to DKDrawableObjects to render that object's appearance. Where DKRastGroup is merely a collection of rasterizers, DKStyle brings several additional powerful behaviours into the picture. The main properties of a style are:

      • Its list of rasterizers - these do the actual work of rendering a path.
      • The style's name - this is something generally for a user interface's benefit.
      • Whether the style is sharable or not (see below).
      • Whether the style is locked or not (editable).
      • The style's unique ID - this is used to identify a style across time, space and different documents.
      • Any text attributes
      • Managing undo for all properties of all contained rasterizers.

      The ability to share styles among several objects can be very useful. Some applications might want to do this a lot, for example a GIS application where a style is used to set the appearance of every symbol of a certain kind. Other applications won't want to do this, for example a simple drawing application might prefer to stick to having a 1:1 relationship between an object and its style. DrawKit supports it whichever way you prefer, or allows you to mix the two approaches if you wish. A style is set to be sharable or not - the initial state of this flag is taken from a class variable so you can set the default for all new styles just once.

      A style is always copied when it is attached to an object, but the sharable flag affects copying. If sharable, the object is not really copied, but simply retained. Non-sharable styles are genuinely copied. Thus shared styles, if changed, will change all objects to which they are attached at once, whereas non-shared ones only affect the single object they belong to. Style sharing works even when a style is copied and pasted from one object to another - if a sharable style, it will be shared with the object to which it is pasted. Note that style sharability is a property of the style itself, not the object(s) to which the style is attached.

      All properties of all rasterizers contained by a style are undoable. Because there are so many possible properties that can change and each rasterizer doesn't really want to be bogged down worrying about how to deal with undo, DKStyle makes use of Key-Value Observing (KVO) to implement undo. Each rasterizer advertises the properties that can be undone in a class method, and the style uses that information to observe changes to those properties via KVO. It then saves the old properties to the undo manager so they can be undone.

      Note there is no limit to the number or type of rasterizers that a style can contain. If your style wants twenty different strokes there's no reason (except performance, possibly) why it couldn't. It is the ability to superimpose and reorder rasterizers freely that give styles their tremendous versatility.

    - ☐ DKStyle and Key-Value Observing (KVO)

      The immense variety and sheer number of properties embodied within a DKStyle and all its component rasterizers make managing them for Undo and notifying their clients quite a task. Luckily, Cocoa has a mechanism called Key-Value Observing (KVO) that considerably eases the amount of work we need to do. DKStyle uses KVO to "observe" all of the interesting properties of all of its component rasterizers. This allows it to centralise all of the Undo processing and also updating all of its clients when any property of any component is changed. The beauty of this approach is that the rasterizers themselves need to do very little work to take advantage of this - in fact each class just has to publish a simple list of strings which are the names of the KVC-compliant properties that it wants to be undoable. It can do this because DKRasterizer subclasses GCObservableObject, which simplifies much of the work needed to observe properties using KVO.

      Each rasterizer class has the method + observableKeyPaths which returns an array of strings, each being an undoable property of instances of the class. Subclasses can call super and add their own properties, so a list of all properties for any given class can be built up. When a rasterizer is added to a style, the style uses this list to set itself as an observer for all of the listed properties. When the style is released or a rasterizer removed, the style removes itself as an observer in the same way. As an observer, the style "sees" any changes to any of the published properties.

      To make handling Undo even more user-friendly, each rasterizer class should implement the -registerActionNames: method, and register an action name for each property. DKStyle will use these action names when setting up the Undo menu item for the observed property.

      As anyone who has worked with KVO will know, getting it right can sometimes be tricky. This might matter if you subclass DKRasterizer. In most cases you can implement +observableKeyPaths and -registerActionNames and be done, but if any sub-properties within the rasterizer are themselves observable, you need to also override -setUpKVOForObserver: and -tearDownKVOForObserver: (inherited from GCObservableObject) and set up observation for the properties of the additional sub-property object. (An example of this in DK is DKGradient, which is a property of DKFill but does itself have observable properties of its own). If your sub-property objects subclass GCObservableObject, you can extend the simplified KVO approach of publishing your observable key paths to such objects quite easily.

    - ☐ Strokes

Topic

- ☐ DKStroke
  DKStroke is a concrete subclass of DKRasterizer that provides for a straightforward stroke of a path. Its properties include:

  • its width
  • its colour
  • its shadow, if any
  • its dash, if any (set using a DKLineDash object)
  • line cap and join styles
- ☐ DKArrowStroke
  DKArrowStroke subclasses DKStroke to add arrow heads to a stroked path. A variety of arrow head styles are supported, and this class also supports "smart" arrow heads that can be used on curving paths to good effect.
- ☐ DKRoughStroke
  DKRoughStroke subclasses DKStroke to add random variation to the stroke width along the rendered path. This gives a much more naturalistic and "hand drawn" look to a stroke, which can be invaluable in some kinds of illustration work.
- ☐ DKZigZagStroke
  DKZigZagStroke subclasses DKStroke to provide a path that zig-zags about the nominal path. The wavelength, amplitude and "spread" (roundness of the peaks) can all be adjusted.
- ☐ Dashes
  Dashes are handled by a small helper class called DKLineDash. This simply stores a dash's properties and applies them to a path on demand. The use of an object to store a dash makes life a lot easier when it comes to handling dashes in a user interface, for example.

- ☐ Fills
  - ☐ DKFill
    DKFill provides for a basic colour fill using a solid colour, an image-based pattern or a gradient. Its properties include:

    • its colour (including image-based pattern colours)
    • its shadow, if any
    • its gradient, if any

    Note that image-based patterns are entirely handled by Cocoa's default implementation. They are useful for emulating the patterns used in older graphics software (e.g. MacDraw) but they are not hugely flexible. Their main advantage is good performance. DrawKit also has DKFillPattern for a different approach to patterns, where a "motif" image is placed at regular intervals with many parameters being adjustable.
    - ☐ DKGradient
      DKGradient is an object that implements Quartz's gradient shading with a higher-level API. Here it is used as a property of a DKFill, where it takes priority over any solid colour fill value. DKGradient is compatible with GCGardient as used in the Gradient Panel framework, so you can freely interchange one for the other if you want to use the GP user interface in your application.
  - ☐ DKHatching
    DKHatching is a DKRasterizer subclass that fills a path with a series of straight lines. This class is invaluable for CAD-type drawing applications and is surprisingly versatlie, especially when used in pairs or groups. Properties include:

    • the line width
    • the line colour
    • the line dash (set using a DKLineDash object)
    • the line spacing
    • the line angle
    • line cap and join styles
    • the line phase or lead-in offset
  - ☐ DKFillPattern
    DKFillPattern is a subclass of DKPathDecorator since it leverages the same image caching techniques for performance reasons. However its disposition is rather different. The problem with Quartz's basic pattern support via NSImage and NSColor is that it's hard to control very precisely and alignment is always to the base coordinates not to the object being filled. DKFillPattern works differently - it takes an image (called the "motif") and repeats it at intervals within the path's interior. The image's PDF representation is used wherever possible so a vector image remains a vector image even when used as a pattern motif. The scaling, spacing and angle of the motif is controllable, as is the angle of the pattern as a whole and the alternate row and column offset values. Because this object can do a lot of intensive drawing work at times, it is able to use a low-quality image of the motif during live updates for better performance, then switch to the PDF motif for better quality when the rapid redrawing ceases.

    The motif is always positioned based on the path's centre point so it remains stable as the path is resized and moved in the drawing.
  - ☐ DKZigZagFill
    DKZigZagFill subclasses DKFill to provide a zig-zag outline to the filled region. As with DKZigZagStroke, wavelength, amplitude and spread can be controlled.
- ☐ Adornments
  - ☐ DKImageAdornment
    DKImageAdornment is a DKRasterizer that can display a single image within the path's bounds - typically centred. The image scale, opacity, angle and offset can all be controlled, or the image can be scaled to fill the bounds or fit proportionally within it.

    This approach to displaying images may be more versatile than using a DKImageShape, depending on your needs. If you want to apply a Core Image effect to an image using a DKCIFilterRastGroup, this is currently the only built-in way within DrawKit to do it.
  - ☐ DKTextAdornment
    DKTextAdornment is a DKRasterizer that provides two main ways to add text to an object. It can lay text out in a block in much the same way as a DKTextShape, or it can lay it out along a path (including curved paths). Text attributes can be controlled using the Font Panel or other UI.

    This rasterizer is also able to lay out text by flowing it into a path or shape, using the DKBezierTextContainer class. In this mode, the vertical placement parameter is ignored (always acts as per 'top' alignment), as is the separate text angle. The text can be set to adopt the object's angle or not however.
  - ☐ DKPathDecorator
    DKPathDecorator is a DKRasterizer that can place copies of an image (called the "motif") at linear intervals along a path. Wherever possible it uses the PDF representation of the image to maximise quality, so vector images remain vector images. The spacing and scale of the motif can be controlled, and also whether the image is rotated to the instantaneous slope of the path at the point where it is drawn. This gives a very powerful effect where a path effectively becomes the guideline for the placement of much more complicated objects.

    In addition, you can specify a lead-in and a lead-out scale, where the scale of objects drawn close to the ends of the path are reduced to give a gradual ramping in and out of the motif.

    This object also supports a special "chain mode" of operation where successive links formed by the motif can be positioned accurately so that their nominated end points line up precisely. This has application in mechanical drawing for example, or for drawing linked train cars on a track.

    DKPathDecorator is computationally and graphically expensive, and can cause slow drawing. To alleviate this, it is able to use a cached offscreen version of the motif for faster, low quality updates. It also takes care only to draw those copies of the motif that actually intersect the view's update region. Nevertheless, beware that complex motifs can sap drawing performance considerably.

Topic

- ☐ Rasterizer Groups

  Because styles are organised into a rasterizer tree, rasterizers can be grouped. Normally there is not much advantage to doing this unless you want to enable or disable several rasterizers at once. However, when the group itself is able to perform some drawing work of its own, things start to get more interesting.

  DKCIFilterRastGroup is a rasterizer group that is able to process whatever it contains using a Core Image filter (at the present time, only one filter at a time). Thus the output from whatever other rasterizers it contains is passed through the CI Filter before being displayed, which really opens up some amazing effects.

  DKQuartzBlendRastGroup does a similar thing but applies a different Quartz blending mode to the result before displaying it (Hard Light, Multiply, Difference, etc).

- ☐ Styles and Text

  Given how useful the ability to share styles among multiple objects can be for many applications, DKStyle also supports text attributes that can be shared among text objects in exactly the same way. All styles support text attributes independently of any rasterizers they might also have, but by default the text attributes are not set. The category DKStyle+Text provides numerous additional methods on a DKStyle for setting attributes such as the font, paragraph style, alignment and so forth.

  Objects such as DKTextShape can use their attached style's text attributes or their own local attributes - again it's up to the application design to decide how such objects should typically operate. The text attributes of rasterizers such as DKTextAdornment are independent of the style's attributes, if it has any, except for the case of dragging text onto an object, where the style's text attributes, if any, are used for the adornment's initial attributes.

  It is possible to define a style with only text attributes - in fact DKStyle+Text returns such a style by default, having 18pt Helvetica centred text. Styles that only have text attributes are also usually named after the font for ease of easily identifying them, but this is up to your application - there is a class method to generate a suitable name from the font info.

  The DKTextShape object is also able to set its style's text attributes if the style is not locked, by directly responding to standard Text menu commands such as Bold, Italic and so forth, and to the Font Panel. If the style is shared all objects sharing the style will be updated as well. Currently Drawkit requires that the text attributes apply to all of the text at once (because where a style's attributes can be shared by different pieces of text, applying the text in ranges is not really feasible as each piece of text is generally different). However if you elect to disconnect a DKTextShape from its style and allow it to use its local attributes, different attributes can be applied to different ranges of the text if required. The disadvantage of course is that this "style" is then unique to that particular DKTextShape object.

- ☐ The Style Registry

  The Style registry is an optional component in DrawKit that helps an application manage styles that it reuses across different documents. If your application is relatively simple this component is probably of little interest, but something like a GIS application is much more likely to use it, where many documents are likely to contain identical styles.

  The purpose of the registry (the DKStyleRegistry class) is to store styles in a persistent way so that they can be reused efficiently, and to manage changes to the styles even if used in multiple documents. As a convenience it also stores styles in named categories and has methods to help support a user interface.

  Styles can come from several sources - the user can create them on the fly as they use an application, they can be read in with existing documents, read in from user defaults or external files, or generated by code. The style registry aims to consolidate all of these sources into one coherent repository for styles. A user typically identifies a style by its name, but this is just a convenience for user interfaces - internally a style is identified strongly by its uniqueKey. This is a UUID string that the registry uses to say whether a style is the same as another one - they are if their uniqueKey values match, even if their actual content might be quite different.

  A style's uniqueKey (or ID) is set once for all time the first time the style is created. Even if the style is subsequently edited, this key never changes. When a style is registered (and remember, you are not obliged to register any style, or use the registry at all) it is stored against this key. The style's name doesn't come into it (though names are automatically disambiguated when registered, for the user's convenience). Registered styles should generally avoid being edited, and so the registry locks styles that are added to it by default.

  When a style is read in from an external source and it is flagged as a registered style, the style needs to be reconsolidated with the current registry. The same style may have changed since the document was saved for example. What should be done? Well, it depends on the application. Some applications might want to keep documents up to date with the latest version of the style, others might want the document's styles to prevail, other might want to maintain them as two separate styles... there is no one rule that applies to all. Thus the registry provides a mechanism for handling this, but the application itself must decide what to do. DKDrawingDocument provides a default template for doing this, that can be overridden or reimplemented in your application as required.

  GIven a set of styles read in that are flagged as previously registered, you can ask the registry to test each one and determine whether the style is known, and if so whether it is newer, the same or older than the registered version. You can then reconsolidate the styles as you need to based on this information - and if as a result styles in the file need to be updated, a simple call will do this for you. All this happens as a document or other file is opened.

  The registry can build a menu of styles divided into categories if you wish, this can be handy to help support a user interface. Each style contributes an icon-sized swatch to the menu.

- ☐ Style Scripts

  Style scripting was originally developed because the rasterizer tree was considered to be a complex structure that had many properties and parameters, and building a complex style from all of these components was likely to be somewhat involved. The script approach allows you to specify the style's structure and properties by using a specially formed string, which is parsed and used to make the relevant rasterizer objects and set their properties.

  Unfortunately scripting does not really simplify the process - it merely makes it just as complicated in another form. Thus while scripting is still supported and works, its benefits have proved limited and it is likely to be removed from DrawKit at some future point. To this end there seems little purpose in documenting the scripting extensively since investment in learning this approach is likely to be wasted. In addition, some newer rasterizers have not been written to include scripting support, and scripting isn't a good fit with the modern behaviour of the style registry.

  The alternative to scripting is either to: a) build styles the good-old fashioned way by instantiating rasterizers, setting their properties and adding them to a style; b) by implementing a user interface that does it for you; or c) using the existing UI within e.g. the DrawKit demo application to build styles and saving them as a file that you can read into your own application.

  If your application only has basic needs, say the traditional one stroke and fill per object, building styles is not especially onerous - in fact there are simple convenience class methods that will make this type of style for you. You can also use these styles as starting points for constructing more complex ones. In truth, making a style-editing UI is far less difficult that it might seem, and for many apps will be necessary regardless, since users will expect a way to change the appearance of objects in some way.

  Reading in a file built elsewhere is also worth considering, especially if your application uses the style registry, because it is able to help you manage styles from external files very effectively.

- ☐ Views

  The architecture of DrawKit is designed to support any number of views "into" a drawing. These views might actually display the drawing itself, or just present some aspect of it to the user in some fashion - an example could be a table view showing the arrangement of the layers. By using DKViewController as a basis for your views' controllers, you can easily make any sort of view or user interface for the DrawKit drawing.

Topic

For views such as the layer UI mentioned, it is worth using a DKViewController because it's much easier to get informed about events and so forth that involve objects deep within the drawing without having to dig down through the drawing structure yourself. For example when any layer or drawable object changes state the view controller is informed directly, so any interface attached to it is able to quickly get the update triggers it needs.

Of course there are numerous notifications that you can also make use of, if that makes more sense. For example a global "inspector" type interface may prefer to rely on notifications since it can get them from all documents/drawings at once. The DKDrawKitInspectorBase works this way, for example. For a complete list of available notifications, see xxxx.

- ☐ General views

  For the case of actually viewing and interacting with the drawing, DrawKit provides the DKDrawingView class. This provides some useful features that most drawing programs are likely to want to take advantage of, such as rulers, zooming in and out of the drawing, and of course acting as an initial first responder for all actions and events that ultimately end up targeted at the active layer and any objects it contains. DKDrawingView interfaces to DKViewController for handling all the usual mouse events, and provides a few other conveniences, such as drawing page breaks for the current printer setup. In general for actually viewing the drawing, you should use DKDrawingView, or possibly a subclass of it, though there's relatively little functionality that should be part of the view itself - mostly subclassing the controller is likely to be the better bet for customising interactivity with DKDrawing and its layers.

  DKDrawing owns its controllers but it does not own or keep a reference to any view. Views are, as normal, owned by their superviews and ultimately their windows. There is also no limit on the number of view/controller pairs that a DKDrawing can support - you can easily set up a split view of the same drawing, or have views in different windows viewing the same drawing. All that is required is that each view has an associated DKViewController added to the drawing. Each view that views the same drawing will be automatically updated when necessary - if you are working with objects in one view, any others will display changes live as you work.

  DKDrawingView is generally used in a "flipped" state, meaning that the Y coordinate increases in a downward direction. This is common to many drawing programs and is familiar. However you can use DrawKit in an "unflipped" state where Y coordinates increase in an upward direction. Because this needs to be consistent across all views associated with a drawing, this is actually set in DKDrawing - DKDrawingView queries this value and returns it in the -isFlipped method.

- ☐ Automatic Drawing construction

  DrawKit suppports a special mode of operation that is implemented by DKDrawingView. This convenient feature constructs all of the necessary drawing "back-end" for a view if you don't bother to make one yourself. This is meant to be analogous to Cocoa's NSTextView class - if you do not set up the text "back-end" yourself it does it for you, so you can go right ahead and start editing text immediately. DrawKit provides the same approach to drawings - drop a DKDrawingView into a nib and you will get a complete working drawing editor with no set up at all. The drawing size is set to the view's bounds. In this case the DKDrawing is owned by the view, which is fine because the weak references within DKViewController ensure that there is no retain cycle created.

  This mechanism works when DKDrawingView's drawRect: method is called - if at that point there is no drawing, DKDrawingView creates one, adds layers to it and then carries on as normal. The upshot is a very simple and easy way to drop a drawing editor into your application.

  The drawback of this approach is the same as the one for NSTextView - it might not be what you want. While it's a very typical setup, possible drawing editors are likely to be even more varied than possible text editors, so chances are you'll want something else. For ultimate control you'll want to set up the "back-end" by hand, just as you would for NSTextView - for other cases you can let the automatic setup be done and then modify it, or override methods in DKDrawingView that make the automatic DKDrawing and supply the set-up you want.

  Functionally, there is no difference in terms of what DK can do when it's set up this way, though because you are not using DKDrawingDocument you would have to handle file opening and saving yourself.

- ☐ Handling Drag and Drop

  In a Cocoa application, NSView is naturally the receiver of dragged data. DrawKit allows the general dragging model (NSDraggingDestination) to be extended down into its layers and objects within layers, much as it does for menu commands etc. The default behaviour is for dragged objects to target the active layer, and then as the drag location changes, to allow objects under the mouse to become selected if they are able to receive the dragged data. This allows you for example to drop image files from the Finder on top of an individual object, and have it adopt that image in a meaningful manner. In fact, DrawKit's standard objects are able to receive drags of strings, images, and colours, and layers are also able to receive drags of images, creating DKImageShapes, and text, creating DKTextShapes. In addition, DrawKit's native objects are also draggable between different views and documents.

  The first object that needs to deal with a drag is DKDrawingView, because a view must be the initial destination of any drag. When the active layer changes, DKViewController gathers the drag types that the layer and its objects are able to respond to, and sets these types as the registered drag types for its view. Thus any drags matching the given types are first handled by the view. Then the same NSDraggingDestination methods are called on the active layer - so any layer is able to implement the NSDraggingDestination protocol exactly as if it were a view.

  DKObjectDrawingLayer takes this one step further and targets the drawable object under the mouse (drag location). For simplicity a drawable object does not implement the full NSDraggingDestination protocol - instead the layer handles the majority of it and the object is required only to do two things:- 1. supply the drag flavours it knows how to respond to, and 2. handle the drop when the user drops the item. The layer will not pass a drop to an object that can't receive the data, is locked or hidden, nor allow it to be selected. You can disable the passing on of drags to objects by using the -setAllowsObjectsToBeTargetedByDrags: method. If NO, the layer can still receive drags.

  Standard objects in DrawKit respond to dragged data in the following ways:

  • Any object can receive a colour item. If the object has one or more fills, the topmost fill is set to the colour, otherwise if it has one or more strokes the topmost stroke is set to the colour. If it has neither, a fill is added with the given colour.

  • Any object is able to receive an image. If the object already has an image or path adornment or pattern fill, the image sets the adornment' or pattern's image. Otherwise if the object is a shape, an image is added as a DKImageAdornment. If the object is a path, a DKPathDecorator is added having the dragged image. If the object is a DKImageShape, the dragged image sets the shape's image.

  • Any object is able to accept a dragged string. If the object already has a DKTextAdornment, the strings sets the adornment's text. If the object is a DKTextShape, the string sets the object's text. If it doesn't have any text, a DKTextAdornment is added, having the block layout for a shape, and the path layout for a path, and having the style's text attributes, if any (otherwise Helvetica 12, left aligned).

  • If an image is dropped into the layer (and not an object within it), a new DKImageShape is created with the image and added at the drop location.

  • If text is dropped into the layer a new DKTextShape is created with the text and added at the drop location.

  The choice of these behaviours is to do the most useful, expected thing with dragged-in data. Naturally you can customise this in many ways including disabling it altogether. Each class of drawable object supplies a list of the drag data types it is willing to respond to, so your subclass can modify this list how it likes. Its implementation of -performDragOperation can do whatever makes sense for the data being dropped. The layer itself returns the union of all of the drag types from all drawable classes when it registers the drag types, and this code is forward-compatible with any new classes of DKDrawableObject you might create.

  Note that the above default implementation of handling dragged data works by modifying a copy of the receiver object's current style. Thus even if the object is using a locked and/or registered style, it still responds to the drag and ends up with a new style assigned. Style's are never modified "in place" though if the object is the sole client of a style, it will be replaced and the old one discarded, so it is effectively equivalent. The point is that the drag/drop behaviour does not change the style of any object other than the one receiving the drag.

Topic

- ☐ Specialised views

  Because DrawKit provides no user interface of its own except the interactivity with objects, it is designed to make as few assumptions about how your user interface might work as possible. However because many typical drawing-type applications often have quite similar user interfaces, DK does provide some support, particularly for inspector-type views.

  DKDrawkitInspectorBase is a simple class that subclasses NSWindowController, and can be used by your own inspector controllers if you wish. All it does it to provide some standard hooks for most of the useful state change notifications coming out of drawkit, such as the active document changing, the active layer changing, and the selection changing. It leaves the handling and display of the information entirely up to your application.

- ☐ Creating other kinds of views of the model

- ☐ Using DrawKit as the basis for an application

  DrawKit isn't much use on its own - it needs to be made into part of an application. It might play the major role in providing your application's data model, or it may only find a minor home somewhere on the periphery. This section deals with a few typical scenarios and explains how DrawKit is designed to address them.

  - ☐ Document-based applications

    If your application is a document-based drawing application, DrawKit can play a major role in providing the core "engine" that you need. In fact, this is its main design focus. In such an application, a document (NSDocument subclass) will typically own a DKDrawing instance and arrange for one or more DKDrawingViews to display and interact with it. To make this approach even easier, you could base your document class on DKDrawingDocument.

    DKDrawingDocument is an NSDocument subclass that owns a DKDrawing and arranges for it to be created when the document is initialised or opened from a file on disk. It also provides an outlet called m_mainView which you can hook up to a DKDrawingView in Interface Builder (of course if you have other arrangements in mind you can easily extend this approach). DKDrawingDocument automatically connects a valid DKDrawingView on that outlet to the DKDrawing instance it creates via a suitable controller, so it very much "just works" for this typical case.

    The important thing to remember is the M-V-C design - you need all three: model, view and controller for the Drawkit system to hang together and function. DKDrawingDocument makes these connections if you use it, but if you prefer to go it alone, you need to make these connections yourself. Recall that DKDrawing owns its controllers (DKViewController or subclass), the view is owned by its superview or window, and the drawing itself must be owned by some other object - typically the document.

  - ☐ Data handling

    Drawkit is able to archive its entire state and return it to you as NSData. Conversely, it is able to be instantiated complete from a suitably formatted NSData. By saving the data to disk you can save and open a complete drawing. Because DrawKit uses the NSKeyedArchiver and NSKeyedUnarchiver, its format is fairly well insulated from future changes and features added to DrawKit - and it's easy to accommodate older forms automatically when new versions of DK are released, and older versions will ignore new keys added by more recent versions. So in general DK's forrmat is robust and easily adapted and extended without impacting on compatibility.

    You can of course simply add DK's data to some other data you are reading and writing, or just rely on supporting NSCoding for new objects that you add within DK.

    DKDrawingDocument simply reads and writes DrawKit's data as its complete file type, with a .drawing extension and a UTI of net.apptree.drawing

  - ☐ How Undo works

    DrawKit implements undo extensively. Almost any activity is undoable if it changes the data content. Because DrawKit is highly interactive, many operations can generate many nearly identical undo tasks. Because of that, a simple NSUndoManager subclass is recommended to coalesce identical consecutive tasks into one undoable task, thus the undo doesn't replay all the intermediate states the user dragged an object through, it just jumps back to where it started from. This saves a great deal of memory and unnecessary retention of tasks.

    DrawKit uses whatever undo manager object you give it, so the coalescing is NOT automatic. DKDrawingDocument does set up a coalescing undo manager however, so you only need to give this any thought if you are not using it.

    The undo manager is typically set per-document, so each is individually undoable, but you might want to consider sharing an undo manager across all your documents. Consider that a style's properties are generally undoable, yet styles can be shared by multiple objects, including those in different documents. If the undo manager is per-document, the shared style's undoable changes will only be recorded by the undo manager for the most recent object that the style was attached to. For unshared styles this is fine, but for styles shared across more than one document, this can lead to a situation where a user can't undo a style's change if the "wrong" document is active. By using a global undo manager, this is avoided, but has the downside that as a series of tasks are undone, the document they apply to will change, and may not be the active one. Your application design will need to take a view on which approach is better, given that undo is not a frequent operation and in most real-world cases will do the expected thing.

    DKDrawingDocument has a compile-time setting as to whether to use a global or a per-document undo manager.

  - ☐ Reconciling registered styles

    When a drawing is saved, it saves all of the styles it is using with it, as you would expect. Some or all of these styles may be being managed by the Style Registry, if your application is taking advantage of that feature. When such a file is opened again at some later time, it may contain styles that also exist in the current registry - possibly in a different state. Thus a way is needed to reconcile such styles so that the registry and the document agree, or agree to disagree!

    The point of the Style Registry is to keep, in a database-like fashion, a set of styles that can be reused across many documents. However if a style is ever changed, which one is true - the document, the registry, or neither? DrawKit provides a mechanism for handling this problem easily, but the decisions about which style is the "one true version" will rest with your application, or its user, perhaps. This mechanism kicks in when a drawing is first unarchived from a file on disk. If you are using DKDrawingDocument, you can override one high-level method to deal with this, if you don't like the default behaviour. If not, you'll need to drop down to lower levels. Of course if you are not using the Style Registry, you can simply ignore the problem altogether.

    After unarchiving, styles that were originally saved with the drawing are reinitialised and attached to their original objects. Thus after opening, the drawing always "looks right", as it was saved. However if any styles were registered at the time the document was saved, they may have the same unique key as a style in the current registry. Such styles are flagged as "formerly registered" after unarchiving, and can be collected into a set by the document. The set can be checked against the current registry, which will identify any that do match a registered style, and which of the two is newer (or the same, if they haven't been changed). Using this information an application can decide what to do - possibly deferring to the user. The registry then performs a "remerge" operation, which either updates the registry from the document, updates the document from the registry, or re-registers the document's styles anew as additional versions. When actually replacing styles, it calls a nominated delegate to make the final decision about which stays and which goes. If any of the document's styles are to be updated from the current registry, this set of styles is returned to it and the final step is to switch out the document's styles for those matching in the returned set.

    DKDrawingDocument handles all of this processing but by default it allows the registry to overrule the document for any styles that match. If your application doesn't want this, it should override -remergeStyles:readFromURL: and either pass a different option to the remerge method of the style registry, or implement a delegate method so it can check each style as it goes, or preflight the styles and take a decision based on that. It can also of course present a dialog or other UI to the user and ask for their opinion.

- ☐ Ad-hoc deployment of DrawKit

  DrawKit may find a good home in many applications that are not complete drawing applications. For example you might just have a dialog that pops up and allows you to define some quick graphic that you can use elsewhere. DrawKit is a good basis for these types of editors - you may not be using much of its power but it doesn't greatly add to your application's waistline or complexity, while still offering a lot of features "for free" if you want them.

  Deploying DrawKit is basically always the same process - create a DKDrawing, a DKDrawingView or some other interface, and bind them together with a DKViewController or subclass. You can do this manually whenever and wherever you want.

  There is another way however - you can let DKDrawingView do all the work. If a DKDrawingView is instantiated (from a nib, for example) and it gets as far

Topic

as having its drawRect: method called with no drawing in place, it will create one, add a controller and stitch it all together. The view owns the drawing, in this case (the design of the controller prevents any awkward retain cycles). The drawing's initial size is set to the view's bounds, and is complete with a grid layer, a guide layer, and one drawable object layer ready to go. It has a tool controller set with the select/edit tool and works immediately. Of course you'll probably want to tweak this setup, but at least the majority of the initial configuration is done. You can also override DKDrawingView's createAutoDrawing method to make the drawing differently, or modify the default one. This mechanism is intended to mirror the functional concept of NSTextView - you add the view and it creates a fully operational text editor. Likewise, DKDrawingView implements a fully operational drawing editor, just by adding the view.

- ☐ Supporting Inspectors
  - ☐ Drawkit notifications
- ☐ Bullt-in commands and features

Many objects in DrawKit are able to respond directly and immediately to commands (in the sense of menu commands, typically) if you set up the user interface items to send the appropriate actions, targeting first responder. Unless you have done away with using DKDrawingView altogether for some reason, your application will get all of these features for free. If you don't want them or need them, just ignore them.

The way DrawKit handles this is using automatic message forwarding and its hierarchical structure. Thus when First Responder is a DKDrawingView, commands (indeed all messages) pass in order:

• The View
• The View's Controller
*• If the controller is a Tool Controller, the currently set tool**
• The Active Layer
• If the layer owns objects, the "current selection"
• If only one object is selected, the selected object

**note that tools only handle mouse input events, not other messages such as those originating from menu items, other user interface elements, drag and drop and so forth. All these "events" pass directly to the Active Layer.*

Generally objects at each level implement the commands they are interested in and are able to handle, and pass on the rest. If the commands originate from a menu item, each level implements the -validateMenuItem: informal protocol and will manage the menu state automatically. Most objects also support contextual menus consisting of a subset of the most useful commands. Some action methods are intended to be targeted by several items with different tags to differentiate the operation.

The following is a complete list of all action methods implemented at each level by each class. They can be set as the action of any user interface element that supports target/action, and the target should be First Responder (i.e. nil).

DKDrawingView	
toggleRuler:	*shows or hides the view's rulers*
toggleShowPageBreaks:	*shows or hides the page break lines in the view - the print info is obtained from the view's window's document.*
DKViewController	
layerBringToFront:	*brings the active layer in front of all other layers*
layerBringForward:	*brings the active layer one place forward*
layerSendToBack:	*moves the active layer behind all other layers*
layerSendBackward:	*moves the active layer one place backward*
toggleSnapToGrid:	*turns snap to grid on or off*
toggleSnapToGuides:	*turns snap to guides on or off*
toggleGridVisible:	*shows or hides the grid layer*
toggleGuidesVisible:	*shows or hides the guide layer*
copyDrawing:	*copies the entire drawing to the clipboard as a PDF*
DKToolController	
selectDrawingToolByName:	*if [sender title] is the name of a registered tool, the tool is set as the current tool*
selectDrawingToolByRepresentedObject:	*if [sender representedObject] is a valid DKDrawingTool subclass, it is set as the current tool*
toggleAutoRevertAction:	*turns automatic "springback" of tools to the selection tool on or off*
DKLayer	
lockLayer:	*locks the layer - locked layers cannot be edited or moved and do not respond to any commands except unlock*
unlockLayer:	*unlocks the layer*
toggleLayerLock:	*locks or unlocks the layer*
showLayer:	*makes the layer visible*
hideLayer:	*hides the layer - hidden layers generally cannot respond to any commands except showLayer:*
toggleLayerVisible:	*shows or hides the layer*
DKGridLayer	
copy:	*copies the drawing grid to the clipboard as a PDF*
setMeasurementSystemAction:	*[sender tag] is interpreted as a measurement system constant and the grid is set to the default values for that system. Of fairly limited use.*
DKGuideLayer	
clearGuides:	*removes all guides from the layer*
DKObjectOwnerLayer	
toggleSnapToObjects:	*turns snap to other objects on or off*
DKObjectDrawingLayer	
cut:	*invokes copy, then delete*
copy:	*copies the selection to the clipboard as native types and as various image types including PDF*
paste:	*pastes clipboard data into the layer. Native objects are pasteable as is any sort of image which is pasted as a DKImageShape.*
delete:	*deletes the selection*
deleteBackward:	*deletes the selection when the backspace key is pressed*
duplicate:	*makes a copy of the selection, adds it to the same layer and selects it*
selectAll:	*selects every object in the layer*
selectNone:	*deselects all objects in the layer*
objectBringForward:	*if there's only one object selected, brings it in front of all other objects*
objectSendBackward:	*if there's only one object selected, moves it one position backward in the layer*
objectBringToFront:	*if there's only one object selected, moves it one position forward in the layer*
objectSendToBack:	*if there's only one object selected, moves it behind all other objects*
lockObject:	*locks all objects in the selection*
unlockObject:	*unlocks all objects in the selection*
showObject:	*makes all objects in the selection visible*
hideObject:	*makes all objects in the selection hidden*

Topic

revealHiddenObjects:	*makes all hidden objects visible, whether they are selected or not*
groupObjects:	*places all of the selected objects into a group and replaces the selection with the group*
moveLeft:	*moves all selected objects one grid increment to the left*
moveRight:	*moves all selected objects one grid increment to the right*
moveUp:	*moves all selected objects one grid increment upwards*
moveDown:	*moves all selected objects one grid increment downwards*
selectMatchingStyle:	*selects all objects that have the same style as the single object already selected*
joinPaths:	*joins the endpoints of two or more selected path objects if they are placed close enough, turning them into one object.*
+Alignment	
alignLeftEdges:	*aligns the left hand edges of the objects in the selection*
alignRightEdges:	*aligns the right hand edges of the objects in the selection*
alignHorizontalCentres:	*aligns the horizontal centres of the objects in the selection*
alignTopEdges:	*aligns the top edges of the objects in the selection*
alignBottomEdges:	*aligns the bottom edges of the objects in the selection*
alignVerticalCentres:	*aligns the vertical centres of the objects in the selection*
distributeVerticalCentres:	*for three or more selected objects, equally spaces out the vertical centres of objects between top and bottom*
distributeVerticalSpace:	*for three or more selected objects, makes the space between the top and bottom edges of each equal*
distributeHorizontalCentres:	*for three or more selected objects, equally spaces out the horizontal centres of objects between left and right*
distributeHorizontalSpace:	*for three or more selected objects, makes the space between the left and right edges of each equal*
alignEdgesToGrid:	*adjusts the sizes and positions of objects in the selection minimally so their edges align to the grid*
alignLocationToGrid:	*adjusts the positions of objects so they are located at the nearest grid intersection*
+BooleanOps	
unionSelectedObjects:	*for two or more selected objects, forms the union of their paths and replaces them with a new object having that path*
diffSelectedObjects:	*for exactly two selected objects, the path of the upper is subtracted from the lower and a new object having the resulting path replaces the original objects*
intersectionSelectedObjects:	*for exactly two selected objects that intersect, they are replaced by a new object having the intersecting path.*
xorSelectedObjects:	*for exactly two selected objects that intersect, they are replaced by a new object having the XOR of the two paths.*
combineSelectedObjects:	*for two or more selected objects, the paths are all appended into one new path. The result is liek a union or XOR operation depending on other factors.*
setBooleanOpsFittingPolicy:	*[sender tag] supplies the curve-fitting policy to use for boolean operations - always, never or auto.*
DKDrawableObject	
copyDrawingStyle:	*copies the object's drawing style as a private type on the clipboard*
pasteDrawingStyle:	*pastes a drawing style private type, replacing the object's style with it*
DKDrawablePath	
convertToShape:	*replaces the object with a DKDrawableShape having the identical path , style and other attributes*
addRandomNoise:	*adds a small random offset to each control point on the object's path*
convertToOutline:	*replaces the object's path with another formed by taking the outline of its widest stroke. The style is modified to keep the appearance similar.*
breakApart:	*converts each subpath of the object's path (if more than one) to a new separate object.*
roughenPath:	*takes the path's stroke outline, adds many extra points to it and then offsets each by a small random amount.*
smoothPath:	*curve-fits a vector path*
smoothPathMore:	*curve-fits a vector path with a much larger acceptable error value*
parallelCopy:	*creates a copy of the path by offsetting each control point of the original in the direction of the normal of the path at each point. This is not a true parallel to the original, but quite useful in practical terms.*
toggleHorizontalFlip:	*flips the path horizontally (and back)*
toggleVerticalFlip:	*flips the path vertically (and back)*
DKDrawableShape	
convertToPath:	*replaces the object with a DKDrawablePath having the identical path, style and other attributes*
unrotate:	*sets the object's rotation angle to zero*
rotate:	*[sender floatValue] is interpreted as a value in degrees, and the object's angle is set to it*
setDistortMode:	*[sender tag] is used to set one of the distortion operation modes - free, shear, perspective.*
resetBoundingBox:	*leaves the shape's appearance exactly the same, but re-orients the bounding box to the orthogonal drawing coordinates and sets the angle to 0.*
toggleHorizontalFlip:	*flips the shape horizontally (and back)*
toggleVerticalFlip:	*flips the shape vertically (and back)*
pastePath:	*pastes the path of a private clipboard object into the shape, replacing its path*
DKTextShape	
changeFont:	*changes the font of the object's text (conditional on other object settings)*
changeFontSize:	*changes the font size of the object's text*
changeAttributes:	*changes the attributes of the object's text*
editText:	*sets up the editor in the first responding view to edit the object's text*
toggleIgnoreStyleAttributes:	*toggle whether the object gets its text attributes from its style or not*
alignLeft:	*set the text to have a left aligned paragraph style*
alignRight:	*set the text to have a right aligned paragraph style*
alignCenter:	*set the text to have a centre aligned paragraph style*
alignJustified:	*set the text to have a fully justified paragraph style*
underline:	*underline the text*
fitToText:	*resize the object to just enclose the current text*
verticalAlign:	*[sender tag] is interpreted as a vertical alignment constant - top, middle or bottom*
convertToShape:	*converts the object's text to a single new shape object which replaces the original*
convertToShapeGroup:	*converts each glyph of the text to a shape object, groups them and replaces the original with the group.*
paste:	*pastes any text on the clipboard into the object, replacing its text.*
DKImageShape	
selectCropOrScaleAction:	*[sender tag] is interpreted as a general dispaly mode - crop or scale*
toggleImageAboveAction:	*sets whether the image is displayed above or below the inherited drawn content of the shape*
pasteImage:	*pastes any image on the clipboard into the shape as its image*
fitToImage:	*resizes the shape to exactly encompass the image*
DKShapeGroup	
ungroupObjects:	*replaces itself in its owner layer by its own contents.*

Topic

- ☐ Class Reference
  - ☐ DKArrowStroke

    Class name: *DKArrowStroke*
    Inherits from: *DKStroke*
    Purpose: *Rasterizer*
    Outline: *Draws an arrowed stroke along an object's path*
    Implements: *DKRasterizerProtocol, NSCoding, NSCopying*
    See also: *DKStyle, DKRasterizer, GCObservableObject, DKLineDash*
    Description: *Subclasses DKStroke to add arrow heads of various types to the ends of the paths. A variety of arrow head styles are easily set, and the size is independently variable or can be linked to the stroke width. This class is also able to optionally show a dimension value. A DKArrowStroke is added to a DKStyle object in order to render arrowed paths.*
    Principal properties: *Type of arrow head for each end of the line, the size of the arrow heads, dimension label attributes, the colour and width of the path outline (if any).*

  - ☐ DKBezierTextContainer
  - ☐ DKCategoryManager
  - ☐ DKCIFilterRastGroup
  - ☐ DKColourQuantizer
  - ☐ DKDistortionTransform
  - ☐ DKDrawableObject
  - ☐ DKDrawablePath
  - ☐ DKDrawableShape
  - ☐ DKDrawing
  - ☐ DKDrawingDocument
  - ☐ DKDrawingInfoLayer
  - ☐ DKDrawingTool
  - ☐ DKDrawingView
  - ☐ DKDrawkitInspectorBase
  - ☐ DKFill

    Class name: *DKFill*
    Inherits from: *DKRasterizer*
    Purpose: *Rasterizer*
    Outline: *Fills the interior of a path with a solid colour, gradient or image-based pattern*
    Implements: *DKRasterizerProtocol, NSCoding, NSCopying*
    See also: *DKStyle, DKRasterizer, GCObservableObject, DKGradient*
    Description: *Subclasses DKRasterizer to implement the general ordinary type of fill.*
    Principal properties: *Fill colour, gradient or image*

  - ☐ DKFillPattern

    Class name: *DKFillPattern*
    Inherits from: *DKPathDecorator*
    Purpose: *Rasterizer*
    Outline: *Fills the interior of a path with a repeating motif image*
    Implements: *DKRasterizerProtocol, NSCoding, NSCopying*
    See also: *DKStyle, DKPathDecorator, GCObservableObject*
    Description: *Draws a repeating motif at a given spacing, scale, angle and offset. Unlike the ordinary image-based colour fill, this maintains the vector qualities of the motif, and allows greater flexibility in the adjustment of the arrangement..*
    Principal properties: *x and y offset, motif angle*

  - ☐ DKGradient
  - ☐ DKGridLayer
  - ☐ DKGuideLayer
  - ☐ DKHatching
  - ☐ DKDrawableShape+Hotspots
  - ☐ DKImageAdornment
  - ☐ DKImageOverlayLayer
  - ☐ DKImageShape
  - ☐ NSImage+Tracing
  - ☐ DKKnob
  - ☐ DKLayer
  - ☐ DKLayerGroup
  - ☐ DKLineDash
  - ☐ DKObjectCreationTool
  - ☐ DKObjectDrawingLayer
  - ☐ DKObjectOwnerLayer
  - ☐ DKPathDecorator
  - ☐ DKPathInsertDeleteTool
  - ☐ DKPrintDrawingView
  - ☐ DKQuartzBlendRastGroup
  - ☐ DKRandom
  - ☐ DKRasterizer
  - ☐ DKRastGroup
  - ☐ DKReshapableShape
  - ☐ DKRoughStroke
  - ☐ DKRuntimeHelper
  - ☐ DKSelectAndEditTool
  - ☐ DKSelectionPDFView
  - ☐ DKShapeCluster
  - ☐ DKShapeFactory
  - ☐ DKShapeGroup

Topic

- ☐ DKStroke
- ☐ DKStyle
- ☐ DKStyleRegistry
- ☐ DKSweptAngleGradient
- ☐ DKTextAdornment
- ☐ DKTextShape
- ☐ DKToolController
- ☐ DKUndoManager
- ☐ DKUniqueID
- ☐ DKViewController
- ☐ DKZigZagFill
- ☐ DKZigZagStroke
- ☐ DKZigZagStroke
- ☐ GCInfoFloater
- ☐ GCObservableObject
- ☐ GCOneShotEffectTimer
- ☐ GCThreadQueue
- ☐ GCZoomView
- ☐ DKDrawingToolProtocol
- ☐ DKCommonTypes
- ☐ DKRasterizerProtocol

- ☐ Using DrawKit Demo

The DrawKit Demo application is a DrawKit-based applicaiton that adds a basic user interface around DrawKit. Its purpose is twofold - it allows a potential user of DrawKit to explore its features from a user's perspective, and it provides a test harness in which DrawKit itself is developed and tested. The Demo is not a user-oriented drawing program and should not be judged as one (though you can of course create complex graphics with it). The user interface is intended to be the simplest possible that accesses DrawKit's features - it is not intended to be an example of the best or only approach to implementing a UI to DrawKit.

This section is a brief guide to the DK Demo - it is not a complete or comprehensive user manual.

The Demo provides the following:

• A document-based application built around DrawKit as its entire data model.
• A user interface for selecting one of several drawing tools (The Tools Palette)
• A large set of menu commands that access most of DrawKit's built-in features directly.
• A way to add, remove, rename and reorder layers (The Layers Palette)
• A user interface to examine and set the basic geometric and metadata properties of any object (The Object Inspector)
• A user interface to examine and extensively edit any style (The Style Inspector)
• A user interface to manage the Style Registry (The Style Registry dialog)
• A user interfaces to set up the grid
• Other user interfaces for lesser features, such as linear and polar duplication of objects
• Opening and saving of documents in DrawKit's native archive format

The source code of the demo is available, but the implementation of the various interfaces has often been done in a quick and dirty manner that would not be advisable to follow for production code. It may help you learn how to interface with DrawKit, but it should not be taken as the best code example possible.

Using the Demo is straightforward - use the Layers Palette to set the current active layer (the highlighted one) and then choose a tool and use it to draw by clicking and dragging in the main drawing area. Newly created objects are selected by default, and the inspector interfaces will update to reflect the current selection. The Demo does not do anything special with the main drawing view - everything that happens in the view is the default standard DrawKit behaviour.

- ☐ Editing Styles

The Style Inspector is the most complex user interface in the Demo. It can be used to modify existing styles or create new ones. Most (but not all) of the properties of all rasterizers are available in the Style Inspector's interface. Because there are so many properties distributed across a number of different rasterizer classes, the Style Inspector uses a master-detail interface to help keep the UI manageable in both size and usability.

The Style Inspector reflects the current selection automatically. If there is no selection, or more than one object with differing styles selected, the Style Inspector is effectively disabled. If there is just one object selected, or if all selected objects share the same style, the Style Inspector displays the attached style. The top part of the interface (the 'master' list) displays the hierarchical make-up of the style, starting with the style itself at the top (root) and an indented list of contained rasterizers below. As each new object in the drawing is selected, the root (style) is preselected and the lower part of the interface (the 'detail') shows the controles that operate on the style as a whole. The largest element is an image showing the style preview.

Selecting any rasterizer in the master list will display the detail for that rasterizer. Naturally what is displayed depends on the kind of rasterizer selected. For example a DKFill has controls to edit the colour and shadow of the fill, a DKStroke has controls for the stroke width and dash, among others. Note that the master list displays the actual class name of the rasterizer. This is deliberate - it allows a programmer to quickly see which classes of rasterizer have which properties and how they affect the graphical output. All properties are updated live and are undoable (this is largely because of DrawKit's design, the Style Inspector does little to make this happen). Note that although DrawKit allows each rasterizer to be named, this feature is not used nor is accessible in the Demo.

The Style's main detail interface contains controls that pertain to the style itself, and also a basic interface to the Style Registry. This has two main functions - it allows a style you create to be added to the registry, and it allows any of the already registered styles to be picked from the menu and immediately applied to the selected object(s) (which in turn brings it into the Style Inspector for possible editing). You can also set whether the style is locked or can be shared, and give it a user-friendly (i.e. descriptive) name.

To edit a style, it must be unlocked. Locked styles grey out the master list so the contained rasterizers can't be selected. If a style is unlocked that is in the registry, you'll get a warning about the potential unwanted consequences, but it won't prevent you from editing that style. Once a style is unlocked, editing it is a case of selecting rasterizers in the master list and changing their properties using the controls presented in the detail area. You can also use the checkbox next to the rasterizer in the master list to enable or disable the rasterizer. Disabled rasterizers are still present but they do not draw anything - this is a quick way to see what a particular rasterizer is contributing to the result.

Creating a new style is much the same as editing an existing one. In the Demo, you start by copying (using the "Clone style" button) an existing style. Typically you might clone one of the defaults, but you can clone any other style - maybe one that's already close to what you want. When cloned, the style is identical in appearance to the original, but won't be registered (even if the original was) and it won't have a name. It will be unlocked ready for editing. If the rasterizers it has are what you want, you can just edit them as usual. If you want to add or delete rasterizers, you can do so using the + and - buttons between the master and detail areas. The + button has a drop-down menu that lists all of the available rasterizer types - choose one to add it to the style (initially new rasterizers are added to the end of the master list which draws last and thus on top of any previous ones - in this sense the list is upside down). To delete a rasterizer, select it and press the - button. Rasterizers can be reordered (and grouped) simply by dragging them into the desired order in the master list. You can also copy and paste rasterizers or duplicate them using the menu linked to the "commands" button (with the gear icon). Thus you can simply 'borrow' useful rasterizers from other styles if you want.

Topic

If you want to register the style you created, you can do so simply by clicking the + button next to the Style Library pop-up at the bottom of the main style detail area. If you didn't give it a name, one will be assigned to it (typically "untitled 1" or similar). Registering a style also locks it. To delete the displayed style from the registry, click the - button. To name a style, type a name into the Name field and press return (unless you press return the name isn't applied).

Don't forget that styles can be cut and pasted between objects as well, so that is also a quick way to apply a style to several different objects (a convenient way to do this is to right-click the object and choose "Copy Drawing Style" and "Paste Drawing Style"). Another way to apply a style to several objects at once that don't currently share a style is to group them, and to apply the style to the group - as a convenience a group propagates a style to all of its contained items. However you can't edit a group's style using the Style Inspector, because the group object itself has no style.

- ☐ Text Style Editing

  Editing the text attributes of a style is not done using the Style Inspector directly. This is because additional user interfaces in the form of the standard Font Panel and menu commands in the standard Text menu are used which is conventional for a Mac application.

  Typically text style attributes are edited via a DKTextShape object which is selected. The style must be unlocked as usual for editing. Changes in the Font Panel and the Text menu thus directly change the text attributes of the selected object's style. These changes are reflected in the Style Inspector and any other objects sharing the style as usual. For this to work, ensure that the DKTextShape is not set to ignore the style's attributes, otherwise the text changes will only be applied to the text object locally, and not to its style. This setting can be toggled using the Object -> Text ->Use Style Text Attributes menu.

- ☐ Editing object metadata

  The Object Inspector provides a simple inspector that displays the attributes of the current selection. If that is a single object, it provides a way to set its basic geometric parameters (position, size and angle) and also the user data or metadata.

  User data is free-form data attached to an object, using a key (name) to identify it. While DrawKit allows this data to be anything you like, the Demo only provides a more limited set of types, these being strings, or real or integer number values. To add a user data item, click the + button below the metadata list and choose the type from the menu. The item is added to the list and the item's name and value can be edited directly there by double-clicking it and typing in the desired name and value. Items are listed alphabetically by their names.

  Certain style components are able to make use of an object's metadata dynamically. One of these is DKTextAdornment. To try this out, add a metadata item to several objects having the same name but different values. Then use the Style Inspector to create a style having a DKTextAdornment component. Set the label to be blank, and put the name of the metadata item in the "Identifier" field. When this style is shared among the various objects, the text adornment will display the value of the metadata item given by the identifier (strings or numeric data are all valid).

- ☐ Managing the Style Registry

  The Demo provides a user interface to the Style Registry that allows you to manage its categories and the style that each contains. The main benefit of doing this is that it can allow the Demo to operate as an editor for external files of pre-organised styles, that can be used in another DrawKit-based application.

  To open the Style Registry Manager dialog, choose it from the Windows menu. The dialog consists of three areas. On the left is the list of categories. In the centre is a list or grid showing the styles present in the selected category, and on the right there is a preview of the style plus some command buttons. Note that all categories are editable except the "All Items" category which can be selected but not edited (it's shown greyed out).

  To change a category's name, double-click its name and type a new one. To add a category, click the + button, and to delete the selected one, click the - button.

  Note: at the time of writing this interface is terribly buggy and probably not worth documenting until these are fixed.

- ☐ Manipulating Layers

  The Layers Palette shows what layers the current document's drawing has. The active layer is shown as the main highlighted layer. The order of the layers follows the stacking (Z) order in the drawing, with the top layer in the list being the top layer in the drawing.

  Layers can be reordered by dragging them into the desired order. To edit a layer's name, double-click it and type a new name. The checkboxes can be used to lock or hide a layer. Note that in the Demo, you can also do many of these things using the menu commands in the "Layers'"menu.

  To add a new layer, click the + button. To delete the selected (active) layer, click the - button. All layer-related actions are undoable.

  To set the selection colour for a layer, click the colour swatch and choose a new colour from the pop-up menu. The selection colour may be used in different ways by different layer types. If there is no colour swatch, it means that the layer does not support a selection colour (e.g. the grid layer doesn't).

  Using the menu command "New Drawing Layer With Selection" you can create a new layer and move the current selected objects to it in one easy call.

- ☐ Menu Commands

  The Demo has many menu commands. In most cases these are linked directly to the action methods implemented at various levels within DrawKit. The current context (object and layer selections) will allow some menu commands and disallow others that are inappropriate. If a menu command is unavailable, check the current selection - menu commands are highly context sensitive.

  The Demo provides top level menus which are arranged to go from the most general to the most specific left to right (while still following the Human Interface Guidelines). Thus we have File, Edit, View, Layer, Object, Text and Window. Menus work in the conventional manner and meaning generally speaking.

  In addition to the menu bar, contextual menus are available using ctrl-click or right-click on layers and objects. The commands these menus contain will vary according to the object type clicked and other contextual cues.

- ☐ File and Clipboard Operations

  The Demo saves and opens files in DrawKit's native archive format, which has an extension of .drawing (UTI is net.apptree.drawing). The Demo simply uses the default file operations provided by DKDrawingDocument and does not implement anything special in this respect.

  Currently the Demo doesn't export drawings in other formats directly as files, but it is possible to do this using Cut and Paste from the Demo application. For example, you can Copy the entire drawing as a PDF using command-option-shift Copy (Copy Drawing). A normal Copy copies the selection of the active layer as a PDF and as a TIFF image, or you can copy the grid as a PDF by making it active and invoking Copy.

- ☐ Index