**PATHS**

*Cocoa provides support for drawing simple or complex geometric SHAPES using* ***PATHS****.   
(****Cocoa*** *cho phép mày vẽ hình bất kì sử dụng* ***Paths.*** *)*

A ***PATH*** is a collection of **POINTS** used to create **PRIMITIVE-SHAPES** such as **LINES**, arcs, and curves. From these **primitives**, you can create more complex **SHAPES**, such as circles, RECTANGLES, polygons, and complex curved SHAPES, and paint them.

\* **as opposed to**: trái ngược  
Because they are composed of **POINTS** (as opposed to a **rasterized bitmap**), PATHS are lightweight, fast, and scale to different resolutions without losing precision or quality.

The following sections focus primarily on the use of the [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) class, which provides the main interface for creating and manipulating PATHS.

Cocoa also provides a handful of functions that offer similar behavior for creating and drawing PATHS but do not require the overhead of creating an OBJECT. Those functions are mentioned where appropriate, but for more INFORMATION, see [*Foundation Framework Reference*](https://developer.apple.com/documentation/foundation) and [*Application Kit Framework Reference*](https://developer.apple.com/documentation/appkit).

**PATH Building Blocks**

Cocoa defines several fundamental data types for manipulating geometric INFORMATION in the drawing environment. These data types include [NSPOINT](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/TypesAndConstants/FoundationTypesConstants/Description.html#//apple_ref/c/tdef/NSPoint), [NSRect](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/TypesAndConstants/FoundationTypesConstants/Description.html#//apple_ref/c/tdef/NSRect), and [NSSize](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/TypesAndConstants/FoundationTypesConstants/Description.html#//apple_ref/c/tdef/NSSize).

You use these data types to specify **LINES**, **RECTANGLES**, and width and height INFORMATION for the SHAPES you want to draw.   
  
Everything from **LINES** and **RECTANGLES** to circles, arcs, and Bezier curves can be specified using one or more of these data structures.  
  
\* **finer precision:** độ chính xác tốt hơn

The coordinate values for **POINT**, **RECTANGLE**, and size data types are all specified using floating-POINT values. Floating-POINT values allow for much **finer precision** as the resolution of the underlying destination device goes up.

The NSPOINT, NSRect, and NSSize data types have equivalents in the Quartz environment: [CGPOINT](https://developer.apple.com/documentation/coregraphics/cgpoint), [CGRect](https://developer.apple.com/documentation/coregraphics/cgrect), and [CGSize](https://developer.apple.com/documentation/coregraphics/cgsize). Because the layout of the Cocoa and Quartz types are identical, you can convert between two types by casting from one type to its counterpart.

**The NSBezierPATH Class**

The [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) class provides the behavior for drawing most PRIMITIVE-SHAPES, and for many complex SHAPES, it is the only tool available in Cocoa.   
  
An NSBezierPATH OBJECT encapsulates the INFORMATION associated with a **PATH**, including the **POINTS** that define the **PATH** and the **ATTRIBUTES** that affect the APPEARANCE of the PATH.  
   
The following sections explain how NSBezierPATH represents **PATH INFORMATION** and also describe the **ATTRIBUTES** that affect a **PATH’s APPEARANCE**.

**PATH ELEMENTS**

An [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) OBJECT uses **PATH ELEMENTS** to build a **PATH**.   
A **PATH ELEMENT** consists of a primitive COMMAND and one or more **POINTS**.   
  
The COMMAND tells the **PATH OBJECT** how to interpret the **associated POINTS**.   
  
When assembled, a set of **PATH ELEMENTS** creates a series of line segments that form the desired SHAPE.

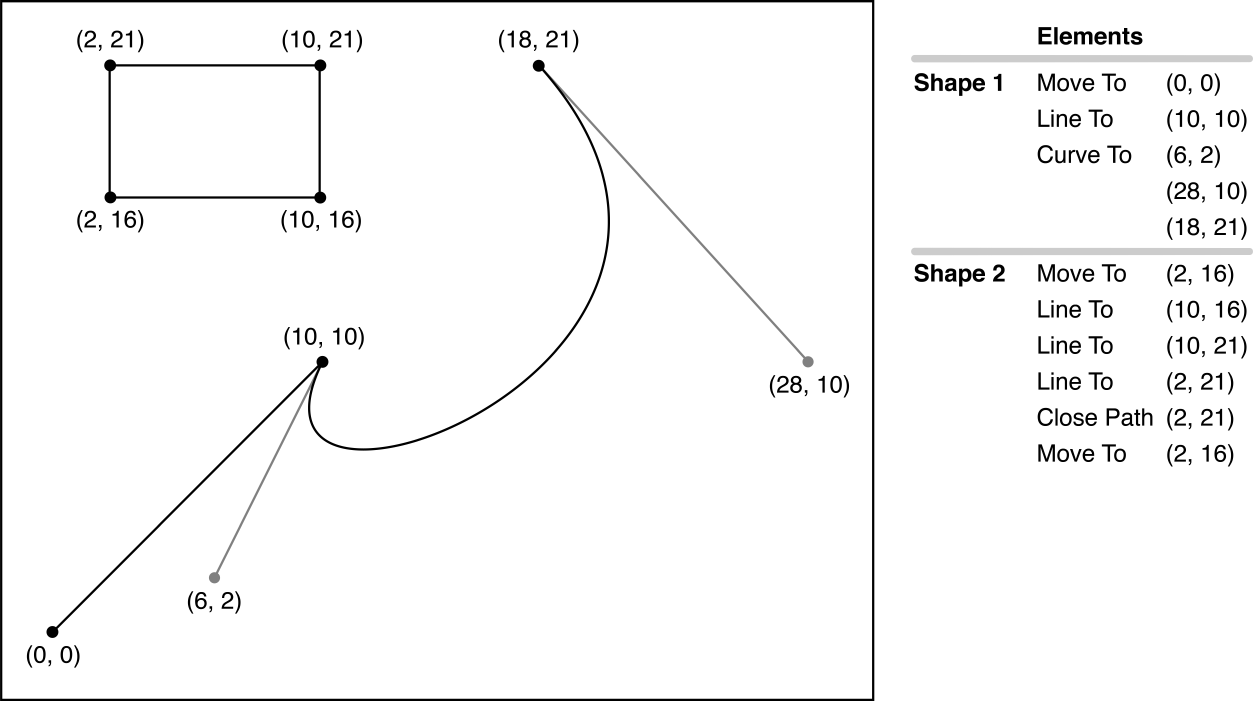
The NSBezierPATH class handles much of the work of creating and organizing **PATH ELEMENTS** initially. Knowing how to manipulate **PATH ELEMENTS** becomes important, however, if you want to make changes to an existing PATH.   
  
If you create a complex PATH based on user input, you might want to give the user the option of changing that PATH later. Although you could create a new PATH OBJECT with the changes, it is far simpler to modify the existing **PATH ELEMENTS**. (For INFORMATION on how to modify PATH ELEMENTS, see [Manipulating Individual PATH ELEMENTS](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BBCICAFC).)

The NSBezierPATH class defines only **4 basic PATH ELEMENT COMMANDS**, which are listed in Table 8-1.

These COMMANDS are enough to define all of the possible **PATH SHAPES**.  
Each **COMMAND** has one or more **POINTS** that contain **INFORMATION** needed to position the **PATH ELEMENT**.   
Most PATH ELEMENTS use the current drawing POINT as the **STARTING POINT** for drawing.

|  |  |  |
| --- | --- | --- |
| **Table 8-1**  PATH ELEMENT COMMANDS | | |
| **COMMAND** | **Number of POINTS** | **Description** |
| [NSMoveToBezierPATHELEMENT](https://developer.apple.com/documentation/appkit/nsmovetobezierpathelement) | 1 | Moves the PATH OBJECT’s current drawing POINT to the specified POINT. This PATH ELEMENT does not result in any drawing. Using this COMMAND in the middle of a PATH results in a disconnected line segment. |
| [NSLineToBezierPATHELEMENT](https://developer.apple.com/documentation/appkit/nslinetobezierpathelement) | 1 | Creates a **straight line** from the current drawing **POINT** to the **specified POINT**. LINES and RECTANGLES are specified using this PATH ELEMENT. |
| [NSCurveToBezierPATHELEMENT](https://developer.apple.com/documentation/appkit/nscurvetobezierpathelement) | 3 | Creates a **curved line segment** from the current POINT to the specified endPOINT using two control POINTS to define the curve. The POINTS are stored in the following order: controlPOINT1, controlPOINT2, endPOINT. Ovals, arcs, and Bezier curves all use curve ELEMENTS to specify their geometry. |
| [NSClosePATHBezierPATHELEMENT](https://developer.apple.com/documentation/appkit/nsclosepathbezierpathelement) | 1 | Marks the end of the current subPATH at the specified POINT. (Note that the POINT specified for the Close PATH ELEMENT is essentially the same as the current POINT. |

When you add a new **SHAPE** to a **PATH**, NSBezierPATH breaks that SHAPE down into one or more component PATH ELEMENTS for storage purposes.   
  
For example, calling [moveToPOINT:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520684-move) or [lineToPOINT:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520742-linetopoint) creates a Move To ELEMENT or Line To ELEMENT respectively. In the case of more complex SHAPES, like RECTANGLES and ovals, several line or curve ELEMENTS may be created. Figure 8-1 shows two SHAPES and the resulting PATH ELEMENTS. For the curved segment, the figure also shows the control POINTS that define the curve.

**Figure 8-1**  PATH ELEMENTS for a complex PATH

Listing 8-1 shows the code that creates the PATH shown in Figure 8-1.

**Listing 8-1**  Creating a complex PATH

|  |
| --- |
| NSBezierPATH\* aPATH = [NSBezierPATH bezierPATH]; |
|  |
| [aPATH moveToPOINT:NSMakePOINT(0.0, 0.0)]; |
| [aPATH lineToPOINT:NSMakePOINT(10.0, 10.0)]; |
| [aPATH curveToPOINT:NSMakePOINT(18.0, 21.0) |
| controlPOINT1:NSMakePOINT(6.0, 2.0) |
| controlPOINT2:NSMakePOINT(28.0, 10.0)]; |
|  |
| [aPATH appendBezierPATHWithRect:NSMakeRect(2.0, 16.0, 8.0, 5.0)]; |

**SubPATHS**

A subPATH is a series of connected line and curve segments within an [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) OBJECT. A single PATH OBJECT may contain multiple subPATHS, with each subPATH delineated by a Move To or Close PATH ELEMENT. When you set the initial drawing POINT (typically using the moveToPOINT: method), you set the STARTING POINT of the first subPATH. As you draw, you build the contents of the subPATH until you either close the PATH (using the [closePATH](https://developer.apple.com/documentation/appkit/nsbezierpath/1520640-closepath) method) or add another Move To ELEMENT. At that POINT, the subPATH is considered closed and any new ELEMENTS are added to a new subPATH.

Some methods of NSBezierPATH automatically create a new subPATH for you. For example, creating a RECTANGLE or oval results in the addition of a Move To ELEMENT, several drawing ELEMENTS, and a Close PATH and Move To ELEMENT (see [Figure 8-1](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BBCHIBJC) for an example). The Move To ELEMENT at the end of the list of ELEMENTS ensures that the current drawing POINT is left in a known location, which in this case is at the RECTANGLE’s origin POINT.

SubPATHS exist to help you distinguish different parts of a PATH OBJECT. For example, subPATHS affect the way a PATH is filled; see [WINDING-RULEs](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BAJIJJGD). The division of a PATH into subPATHS also affects methods such as [bezierPATHByReversingPATH](https://developer.apple.com/documentation/appkit/nsbezierpath/1520656-reversed), which reverses the subPATHS one at a time. In other cases, though, subPATHS in an NSBezierPATH OBJECT share the same drawing ATTRIBUTES.

**PATH ATTRIBUTES**

An [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) OBJECT maintains all of the ATTRIBUTES needed to determine the SHAPE of its PATH. These ATTRIBUTES include the **line width**, **curve** **flatness**, line cap style, line join style, and miter limit of the PATH. You set these values using the methods of NSBezierPATH.

PATH ATTRIBUTES do not take effect until you fill or stroke the PATH, so if you change an attribute more than once before drawing the PATH, only the last value is used. The NSBezierPATH class maintains both a custom and default version of each attribute. PATH OBJECTs use custom attribute values if they are set. If no custom attribute value is set for a given PATH OBJECT, the default value is used. The NSBezierPATH class does not use PATH attribute values set using Quartz functions.

**Note:** PATH ATTRIBUTES apply to the entire PATH. If you want to use different ATTRIBUTES for different parts of a PATH, you must create two separate PATH OBJECTs and apply the appropriate ATTRIBUTES to each.

The following sections describe the ATTRIBUTES you can set for a PATH OBJECT and how those ATTRIBUTES affect your rendered PATHS.

**Line Width**

The line width attribute controls the width of the entire PATH. Line width is measured in POINTS and specified as a floating-POINT value. The default width for all LINES is 1. To change the default line width for all NSBezierPATH OBJECTs, you use the setDefaultLineWidth: method. To set the line width for the current PATH OBJECT, you use the [setLineWidth:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520655-linewidth) method of that PATH OBJECT. To set the default line width for SHAPES rendered without an NSBezierPATH OBJECT, you must use the [CGContextSetLineWidth](https://developer.apple.com/documentation/coregraphics/1455270-cgcontextsetlinewidth) function in Quartz.

Fractional line widths are rendered as close as possible to the specified width, subject to the limitations of the destination device, the position of the line, and the current anti-aliasing setting. For example, suppose you want to draw a line whose width is 0.2 POINTS. Multiplying this width by 1/72 POINTS per inch yields a line that is 0.0027778 inches wide. On a 90 dpi screen, the smallest possible line would be 1 pixel wide or 0.0111 inches. To ensure your line is not hidden on the screen, Cocoa nominally draws it at the screen’s larger minimum width (0.0111 inches). In reality, if the line straddles a pixel boundary or anti-aliasing is enabled, the line might affect additional pixels on either side of the PATH. If the output device were a 600 dpi printer instead, Quartz would be able to render the line closer to its true width of 0.0027778 inches.

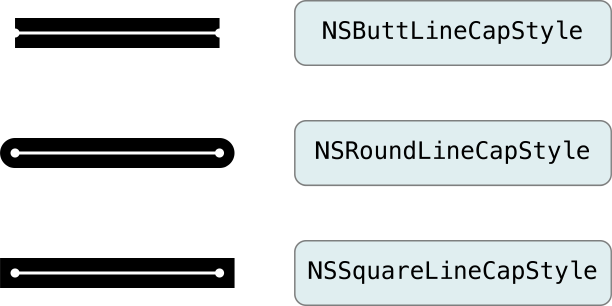
Listing 8-2 draws a few PATHS using different techniques. The [NSFrameRect](https://developer.apple.com/documentation/appkit/1473582-nsframerect) function uses the default line width to draw a RECTANGLE, so that value must be set prior to calling the function. PATH OBJECTs use the default value only if a custom value has not been set. You can even change the line width of a PATH OBJECT and draw again to achieve a different PATH width, although you would also need to move the PATH to see the difference.

**Listing 8-2**  Setting the line width of a PATH

|  |
| --- |
| // Draw a RECTANGLE using the default line width: 2.0. |
| [NSBezierPATH setDefaultLineWidth:2.0]; |
| NSFrameRect(NSMakeRect(20.0, 20.0, 10.0, 10.0)); |
|  |
| // Set the line width for a single NSBezierPATH OBJECT. |
| NSBezierPATH\* thePATH = [NSBezierPATH bezierPATH]; |
| [thePATH setLineWidth:1.0]; // Has no effect. |
| [thePATH moveToPOINT:NSMakePOINT(0.0, 0.0)]; |
| [thePATH lineToPOINT:NSMakePOINT(10.0, 0.0)]; |
| [thePATH setLineWidth:3.0]; |
| [thePATH lineToPOINT:NSMakePOINT(10.0, 10.0)]; |
|  |
| // Because the last value set is 3.0, all LINES are drawn with |
| // a width of 3.0, not just the second line. |
| [thePATH stroke]; |
|  |
| // Changing the width and stroking again draws the same PATH |
| // using the new line width. |
| [thePATH setLineWidth:4.0]; |
| [thePATH stroke]; |
|  |
| // Changing the default line width has no effect because a custom |
| // value already exists. The PATH is rendered with a width of 4.0. |
| [thePATH setDefaultLineWidth:5.0]; |
| [thePATH stroke]; |

**Line Cap Styles**

The current line cap style determines the APPEARANCE of the open end POINTS of a PATH segment. Cocoa supports the line cap styles shown in Figure 8-2.

**Figure 8-2**  Line cap styles

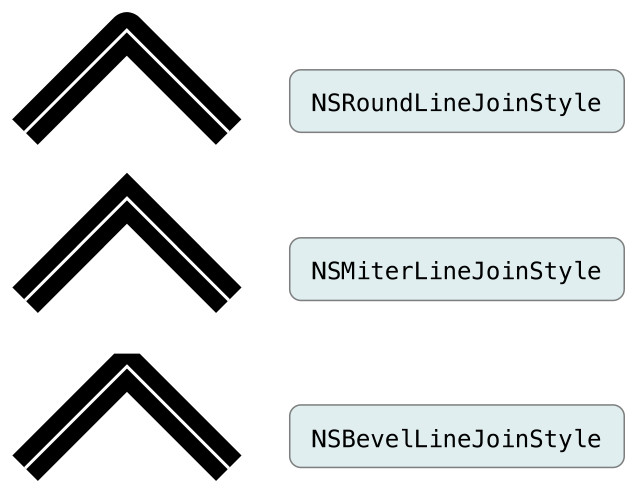
To set the line cap style for a NSBezierPATH OBJECT, use the [setLineCapStyle:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520667-linecapstyle) method. The default line cap style is set to NSButtLineCapStyle. To change the default line cap style, use the setDefaultLineCapStyle: method. Listing 8-3 demonstrates both of these methods:

**Listing 8-3**  Setting the line cap style of a PATH

|  |
| --- |
| [// Set the default line cap style |
| [NSBezierPATH setDefaultLineCapStyle:NSButtLineCapStyle]; |
|  |
| // Customize the line cap style for the new OBJECT. |
| NSBezierPATH\* aPATH = [NSBezierPATH bezierPATH]; |
| [aPATH moveToPOINT:NSMakePOINT(0.0, 0.0)]; |
| [aPATH lineToPOINT:NSMakePOINT(10.0, 10.0)]; |
| [aPATH setLineCapStyle:NSSquareLineCapStyle]; |
| [aPATH stroke]; |

**Line Join Styles**

The current line join style determines how connected LINES in a PATH are joined at the vertices. Cocoa supports the line join styles shown in Figure 8-3.

**Figure 8-3**  Line join styles

To set the line join style for an NSBezierPATH OBJECT, use the [setLineJoinStyle:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520726-linejoinstyle) method. The default line join style is set to NSMiterLineJoinStyle. To change the default line join style, use the setDefaultLineJoinStyle: method. Listing 8-4 demonstrates both of these methods:

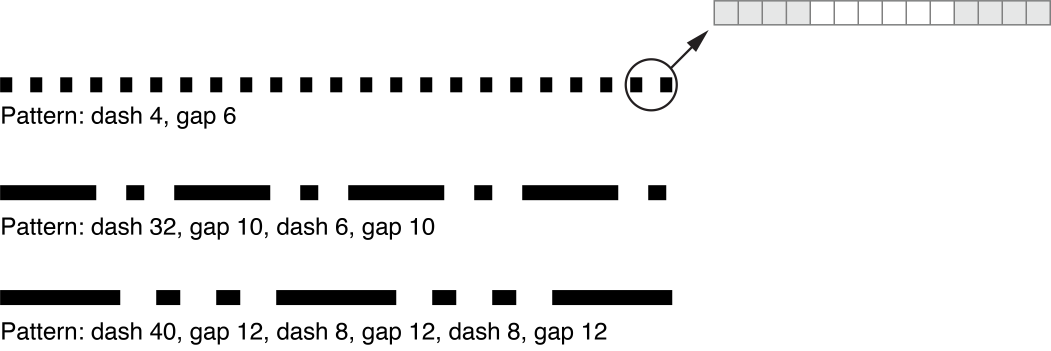
**Listing 8-4**  Setting the line join style of a PATH

|  |
| --- |
| [// Set the default line join style |
| [NSBezierPATH setDefaultLineJoinStyle:NSMiterLineJoinStyle]; |
|  |
| // Customize the line join style for a new PATH. |
| NSBezierPATH\* aPATH = [NSBezierPATH bezierPATH]; |
| [aPATH moveToPOINT:NSMakePOINT(0.0, 0.0)]; |
| [aPATH lineToPOINT:NSMakePOINT(10.0, 10.0)]; |
| [aPATH lineToPOINT:NSMakePOINT(10.0, 0.0)]; |
| [aPATH setLineJoinStyle:NSRoundLineJoinStyle]; |
| [aPATH stroke]; |

**Line Dash Style**

The line dash style determines the pattern used to stroke a PATH. By default, stroked PATHS appear solid. Using a line-dash pattern, you can specify an alternating group of solid and transparent swatches. When setting a line dash pattern, you specify the width (in POINTS) of each successive solid or transparent swatch. The widths you specify are then repeated over the entire length of the PATH.

Figure 8-4 shows some sample line dash patterns, along with the values used to create each pattern.

**Figure 8-4**  Line dash patterns

The NSBezierPATH class does not support the concept of a default line dash style. If you want a line dash style, you must apply it to a PATH explicitly using the [setLineDash:count:phase:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520730-setlinedash) method as shown in Listing 8-5, which renders the last pattern from the preceding figure.

**Listing 8-5**  Adding a dash style to a PATH

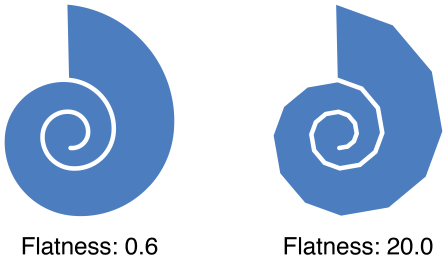
|  |
| --- |
| void AddDashStyleToPATH(NSBezierPATH\* thePATH) |
| { |
| // Set the line dash pattern. |
| float lineDash[6]; |
|  |
| lineDash[0] = 40.0; |
| lineDash[1] = 12.0; |
| lineDash[2] = 8.0; |
| lineDash[3] = 12.0; |
| lineDash[4] = 8.0; |
| lineDash[5] = 12.0; |
|  |
| [thePATH setLineDash:lineDash count:6 phase:0.0]; |
| } |

Line Flatness

The line flatness attribute determines the rendering accuracy for curved segments. The flatness value measures the maximum error tolerance (in pixels) to use during rendering. Smaller values result in smoother curves but require more computation time. Larger values result in more jagged curves but are rendered much faster.

Line flatness is one parameter you can tweak when you want to render a large number of curves quickly and do not care about accuracy. For example, you might increase this value during a live resize or scrolling operation when accuracy is not as crucial. Regardless, you should always measure performance to make sure such a modification actually saves time.

Figure 8-5 shows how changing the default flatness affects curved surfaces. The figure on the left shows a group of curved surfaces rendered with the flatness value set to 0.6 (its default value). The figure on the right shows the same curved surfaces rendered with the flatness value set to 20. The curvature of each surface is lost and now appears to be a set of connected line segments.

**Figure 8-5**  Flatness effects on curves

To set the flatness for a specific NSBezierPATH OBJECT, use the [setFlatness:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520676-flatness) method. To set the default flatness value, use setDefaultFlatness:, as shown in Listing 8-6:

**Listing 8-6**  Setting the flatness of a PATH

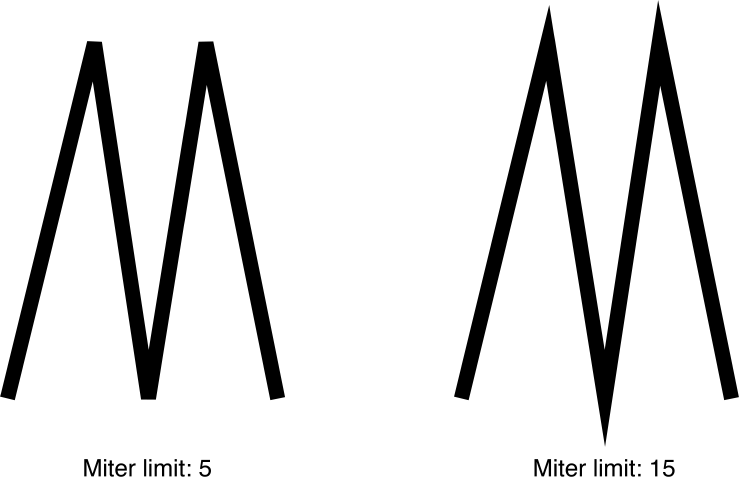
|  |
| --- |
| [- (void) drawRect:(NSRect)rect |
| { |
| if ([self inLiveResize]) |
| { |
| // Adjust the default flatness upward to reduce |
| // the number of required computations. |
| [NSBezierPATH setDefaultFlatness:10.0]; |
|  |
| // Draw live resize content. |
| } |
| // ... |
|  |
| } |

Miter Limits

Miter limits help you avoid spikes that occur when you join two line segments at a sharp angle. If the ratio of the miter length—the diagonal length of the miter—to the line thickness exceeds the miter limit, the corner is drawn using a bevel join instead of a miter join.

**Note:** Miter limits apply only to PATHS rendered using the miter join style.

Figure 8-6 shows an example of how different miter limits affect the same PATH. This PATH consists of several 10-POINT wide LINES connected by miter joins. In the figure on the left, the miter limit is set to 5. Because the miter lengths exceed the miter limit, the line joins are changed to bevel joins. By increasing the miter limit to 16, as shown in the figure on the right, the miter joins are restored but extend far beyond the POINT where the two LINES meet.

**Figure 8-6**  Miter limit effects

To set the miter limits for a specific NSBezierPATH OBJECT, use the [setMiterLimit:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520740-miterlimit) method. To set the default miter limit for newly created NSBezierPATH OBJECTs, use setDefaultMiterLimit:. Listing 8-7 demonstrates both of these methods:

**Listing 8-7**  Setting the miter limit for a PATH

|  |
| --- |
| // Increase the default limit |
| [NSBezierPATH setDefaultMiterLimit:20.0]; |
|  |
| // Customize the limit for a specific PATH with sharp angles. |
| NSBezierPATH\* aPATH = [NSBezierPATH bezierPATH]; |
| [aPATH moveToPOINT:NSMakePOINT(0.0, 0.0)]; |
| [aPATH lineToPOINT:NSMakePOINT(8.0, 100.0)]; |
| [aPATH lineToPOINT:NSMakePOINT(16.0, 0.0)]; |
| [aPATH setLineWidth:5.0]; |
| [aPATH setMiterLimit:5.0]; |
| [aPATH stroke]; |

**WINDING-RULEs (Quy tắc quanh co 😐)**

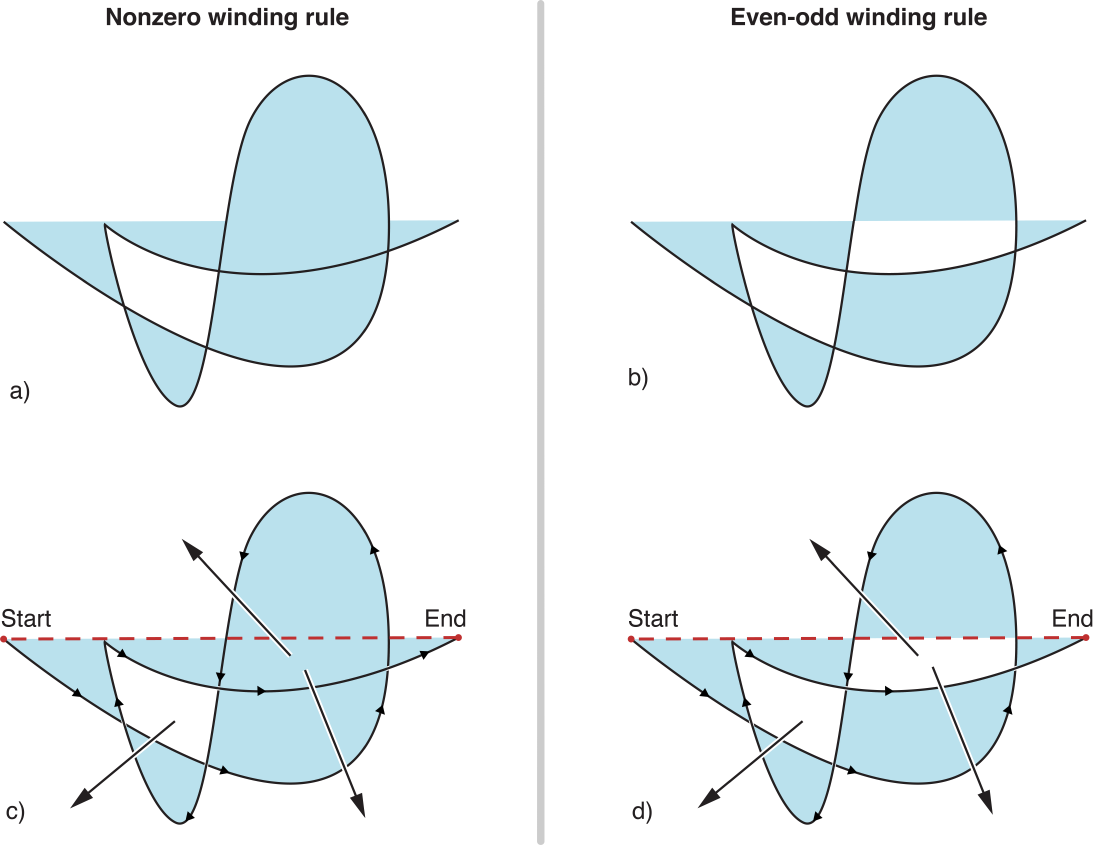
When you fill the area **encompassed (bao trùm)** by a PATH, [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) applies the current **WINDING-RULE** to determine which areas of the screen to fill.   
*(****NSBezierPath*** *sẽ xác định vùng nào sẽ được fill màu)*

A ***WINDING-RULE***  is simply an algorithm that tracks INFORMATION about each ***contiguous region (vùng liên tục)*** that makes up the PATH's overall fill area.   
  
A **ray (tia)** is drawn from a **POINT** inside a given region to any POINT outside the PATH bounds.   
  
The total number of **crossed PATH LINES** (including implicit LINES) and the direction of each PATH line are then interpreted using the rules in Table 8-2, which determine if the region should be filled.

|  |  |
| --- | --- |
| **Table 8-2**  WINDING-RULEs | |
| **WINDING-RULE** | **Description** |
| [NSNonZeroWindingRule](https://developer.apple.com/documentation/appkit/nsnonzerowindingrule) | Count each left-to-right PATH as +1 and each right-to-left PATH as -1. If the sum of all crossings is 0, the POINT is outside the PATH. If the sum is nonzero, the POINT is inside the PATH and the region containing it is filled. This is the default WINDING-RULE. |
| [NSEvenOddWindingRule](https://developer.apple.com/documentation/appkit/nsevenoddwindingrule) | Count the total number of PATH crossings. If the number of crossings is even, the POINT is outside the PATH. If the number of crossings is odd, the POINT is inside the PATH and the region containing it should be filled. |

**Fill operations** are suitable for use with both open and closed subPATHS. A closed subPATH is a sequence of drawing calls that ends with a Close PATH PATH ELEMENT. An open subPATH ends with a Move To PATH ELEMENT. When you fill a partial subPATH, NSBezierPATH closes it for you automatically by creating an implicit (non-rendered) line from the first to the last POINT of the subPATH.

Figure 8-7 shows how the WINDING-RULEs are applied to a particular PATH. Subfigure a shows the PATH rendered using the nonzero rule and subfigure b shows it rendered using the even-odd rule. Subfigures c and d add direction marks and the hidden PATH line that closes the figure to help you see how the rules are applied to two of the PATH’s regions.

**Figure 8-7**  Applying WINDING-RULEs to a PATH

To set the WINDING-RULE for an NSBezierPATH OBJECT, use the [setWindingRule:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520657-windingrule) method. The default WINDING-RULE is NSNonZeroWindingRule. To change the default WINDING-RULE for all NSBezierPATH OBJECTs, use the setDefaultWindingRule: method.

Manipulating Geometric Types

The Foundation framework includes numerous functions for manipulating geometric values and for performing various calculations using those values. In addition to basic equality checks, you can perform more complex operations, such as the union and intersection of RECTANGLES or the inclusion of a POINT in a RECTANGLE’s boundaries.

Table 8-3 lists some of the more commonly used functions and their behaviors. The function syntax is provided in a shorthand notation, with parameter types omitted to demonstrate the calling convention. For a complete list of available functions, and their full syntax, see the Functions section in [*Foundation Framework Reference*](https://developer.apple.com/documentation/foundation).

|  |  |  |
| --- | --- | --- |
| **Table 8-3**  Commonly used geometry functions | | |
| **Operation** | **Function** | **Description** |
| Creation | NSPOINT [NSMakePOINT](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSMakePoint)(x, y) | Returns a properly formatted NSPOINT data structure with the specified x and y values. |
| NSSize [NSMakeSize](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSMakeSize)(w, h) | Returns a properly formatted NSSize data structure with the specified width and height. |
| NSRect [NSMakeRect](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSMakeRect)(x, y, w, h) | Returns a properly formatted NSRect data structure with the specified origin (x, y) and size (width, height). |
| Equality | BOOL [NSEqualPOINTS](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSEqualPoints)(p1, p2) | Returns YES if the two POINTS are the same. |
| BOOL [NSEqualSizes](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSEqualSizes)(s1, s2) | Returns YES if the two size types have identical widths and heights. |
| BOOL [NSEqualRects](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSEqualRects)(r1, r2) | Returns YES, if the two RECTANGLES have the same origins and the same widths and heights. |
| RECTANGLE manipulations | BOOL [NSContainsRect](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSContainsRect)(r1, r2) | Returns YES if RECTANGLE 1 completely encloses RECTANGLE 2. |
| NSRect [NSInsetRect](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSInsetRect)(r, dX, dY) | Returns a copy of the specified RECTANGLE with its sides moved inward by the specified delta values. Negative delta values move the sides outward. Does not modify the original RECTANGLE. |
| NSRect [NSIntersectionRect](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSIntersectionRect)(r1, r2) | Returns the intersection of the two RECTANGLES. |
| NSRect [NSUnionRect](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSUnionRect)(r1, r2) | Returns the union of the two RECTANGLES. |
| BOOL [NSMouseInRect](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSMouseInRect)(p, r, flipped) | Tests whether the POINT lies within the specified view RECTANGLE. Adjusts the hit-detection algorithm to provide consistent behavior from the user’s perspective. |
| BOOL [NSPOINTInRect](https://developer.apple.com/library/archive/documentation/LegacyTechnologies/WebObjects/WebObjects_3.5/Reference/Frameworks/ObjC/Foundation/Functions/FoundationFunctions/Description.html#//apple_ref/c/func/NSPointInRect)(p, r) | Tests whether the POINT lies within the specified RECTANGLE. This is a basic mathematical comparison. |

Drawing Fundamental SHAPES

For many types of content, PATH-based drawing has several advantages over image-based drawing:

* Because PATHS are specified mathematically, they scale easily to different resolutions. Thus, the same PATH OBJECTs can be used for screen and print-based drawing.
* The geometry INFORMATION associated with a PATH requires much less storage space than most image data formats.
* Rendering PATHS is often faster than compositing a comparable image. It takes less time to transfer PATH data to the graphics hardware than it takes to transfer the texture data associated with an image.

The following sections provide INFORMATION about the PRIMITIVE-SHAPES you can draw using PATHS. You can combine one or more of these SHAPES to create a more complex PATH and then stroke or fill the PATH as described in [Drawing the SHAPES in a PATH](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BBCBGDFI). For some SHAPES, there may be more than one way to add the SHAPE to a PATH, or there may be alternate ways to draw the SHAPE immediately. Wherever possible, the benefits and disadvantages of each technique are listed to help you decide which technique is most appropriate in specific situations.

Adding POINTS

An NSPOINT structure by itself represents a location on the screen; it has no weight and cannot be drawn as such. To draw the equivalent of a POINT on the screen, you would need to create a small RECTANGLE at the desired location, as shown in Listing 8-8.

**Listing 8-8**  Drawing a POINT

|  |
| --- |
| void DrawPOINT(NSPOINT aPOINT) |
| { |
| NSRect aRect = NSMakeRect(aPOINT.x, aPOINT.y, 1.0, 1.0); |
|  |
| NSRectFill(aRect); |
| } |

Of course, a more common use for POINTS is to specify the position of other SHAPES. Many SHAPES require you to specify the current POINT before actually creating the SHAPE. You set the current POINT using the [moveToPOINT:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520684-move) or [relativeMoveToPOINT:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520724-relativemovetopoint) methods. Some SHAPES, like RECTANGLES and ovals, already contain location INFORMATION and do not require a separate call to moveToPOINT:.

**Important:**  You must specify a STARTING POINT before drawing individual line, arc, curve, and glyph PATHS. If you do not, NSBezierPATH raises an exception.

Adding LINES and Polygons

Cocoa provides a couple of options for adding LINES to a PATH, with each technique offering different tradeoffs between efficiency and correctness. You can draw LINES in the following ways:

* Create single horizontal and vertical LINES by filling a RECTANGLE using [NSRectFill](https://developer.apple.com/documentation/appkit/1473652-nsrectfill). This technique is less precise but is often a little faster than creating an [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) OBJECT. To create diagonal LINES using this technique, you must apply a rotation transform before drawing. This technique is not appropriate for creating connected line segments.
* Use the [lineToPOINT:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520742-linetopoint), [relativeLineToPOINT:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520709-relativeline), or [strokeLineFromPOINT:toPOINT:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520626-strokelinefrompoint) methods of NSBezierPATH to create individual or connected line segments. This technique is fast and is the most precise option for creating LINES and complex polygons.
* Use the [appendBezierPATHWithPOINTS:count:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520749-appendbezierpathwithpoints) method to create a series of connected LINES quickly. This technique is faster than adding individual LINES.

Polygons are composed of multiple connected LINES and should be created using an NSBezierPATH OBJECT. The simplest way to create a four-sided nonrectangular SHAPE, like a parallelogram, rhombus, or trapezoid, is using line segments. You could also create these SHAPES using transforms, but calculating the correct skew factors would require a lot more work.

Listing 8-9 shows code to draw a parallelogram using NSBezierPATH. The method in this example inscribes the parallelogram inside the specified RECTANGLE. The withShift parameter specifies the horizontal shift applied to the top left and bottom right corners of the rectangular area.

**Listing 8-9**  Using LINES to draw a polygon

|  |
| --- |
| void DrawParallelogramInRect(NSRect rect, float withShift) |
| { |
| NSBezierPATH\* thePATH = [NSBezierPATH bezierPATH]; |
|  |
| [thePATH moveToPOINT:rect.origin]; |
| [thePATH lineToPOINT:NSMakePOINT(rect.origin.x + withShift, NSMaxY(rect))]; |
| [thePATH lineToPOINT:NSMakePOINT(NSMaxX(rect), NSMaxY(rect))]; |
| [thePATH lineToPOINT:NSMakePOINT(NSMaxX(rect) - withShift, rect.origin.y)]; |
| [thePATH closePATH]; |
|  |
| [thePATH stroke]; |
| } |

Adding RECTANGLES

Because RECTANGLES are used frequently, there are several options for drawing them.

* Use the methods of NSBezierPATH to create your RECTANGLE. The following methods are reasonably fast and offer the best precision:
  + [strokeRect:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520741-stroke)
  + [fillRect:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520747-fillrect)
  + [bezierPATHWithRect:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520664-bezierpathwithrect)
  + [appendBezierPATHWithRect:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520670-appendbezierpathwithrect)
* Create RECTANGLES using the Cocoa functions described in [Drawing RECTANGLES](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BBCFABIE). These functions draw RECTANGLES faster than, but with less precision than, the methods of NSBezierPATH.
* Create a RECTANGLE using individual LINES as described in [Adding LINES and Polygons](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BBCHIDGH). You could use this technique to create diagonally oriented RECTANGLES—that is, RECTANGLES whose sides are not parallel to the x and y axes—without using a rotation transform.

Listing 8-10 shows a simple function that fills and strokes the same RECTANGLE using two different techniques. The current fill and stroke colors are used when drawing the RECTANGLE, along with the default compositing operation. In both cases, the RECTANGLES are drawn immediately; there is no need to send a separate [fill](https://developer.apple.com/documentation/appkit/nsbezierpath/1520700-fill) or [stroke](https://developer.apple.com/documentation/appkit/nsbezierpath/1520739-stroke) message.

**Listing 8-10**  Drawing a RECTANGLE

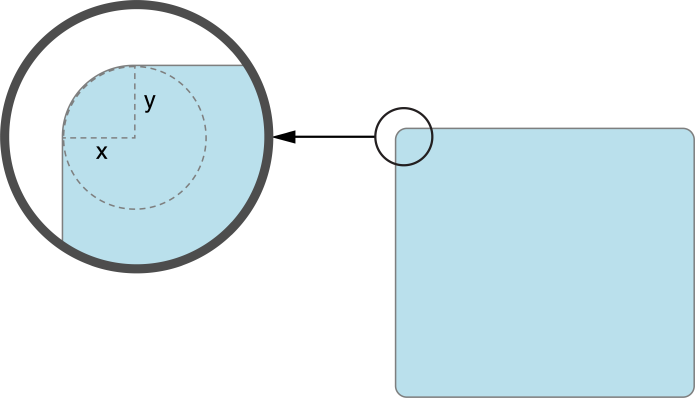
|  |
| --- |
| void DrawRECTANGLE(NSRect aRect) |
| { |
| NSRectFill(aRect); |
| [NSBezierPATH strokeRect:aRect]; |
| } |

Adding Rounded RECTANGLES

In OS X v10.5 and later, the [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) class includes the following methods for creating rounded-RECTANGLES:

* [bezierPATHWithRoundedRect:xRadius:yRadius:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520638-bezierpathwithroundedrect)
* [appendBezierPATHWithRoundedRect:xRadius:yRadius:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520705-appendbezierpathwithroundedrect)

These methods create RECTANGLES whose corners are curved according to the specified radius values. The radii describe the width and height of the oval to use at each corner of the RECTANGLE. Figure 8-8 shows how this inscribed oval is used to define the PATH of the RECTANGLE’s corner segments.

**Figure 8-8**  Inscribing the corner of a rounded RECTANGLE

Listing 8-11 shows a code snippet that creates and draws a PATH with a rounded RECTANGLE.

**Listing 8-11**  Drawing a rounded RECTANGLE

|  |
| --- |
| void DrawRoundedRect(NSRect rect, CGFloat x, CGFloat y) |
| { |
| NSBezierPATH\* thePATH = [NSBezierPATH bezierPATH]; |
|  |
| [thePATH appendBezierPATHWithRoundedRect:rect xRadius:x yRadius:y]; |
| [thePATH stroke]; |
| } |

Adding Ovals and Circles

To draw ovals and circles, use the following methods of NSBezierPATH:

* [bezierPATHWithOvalInRect:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520647-bezierpathwithovalinrect)
* [appendBezierPATHWithOvalInRect:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520718-appendbezierpathwithovalinrect)

Both methods inscribe an oval inside the RECTANGLE you specify. You must then fill or stroke the PATH OBJECT to draw the oval in the current context. The following example creates an oval from the specified RECTANGLE and strokes its PATH.

|  |
| --- |
| void DrawOvalInRect(NSRect ovalRect) |
| { |
| NSBezierPATH\* thePATH = [NSBezierPATH bezierPATH]; |
|  |
| [thePATH appendBezierPATHWithOvalInRect:ovalRect]; |
| [thePATH stroke]; |
| } |

You could also create an oval using arcs, but doing so would duplicate what the preceding methods do internally and would be a little slower. The only reason to add individual arcs is to create a partial (non-closed) oval PATH. For more INFORMATION, see [Adding Arcs](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BBCBGFBH).

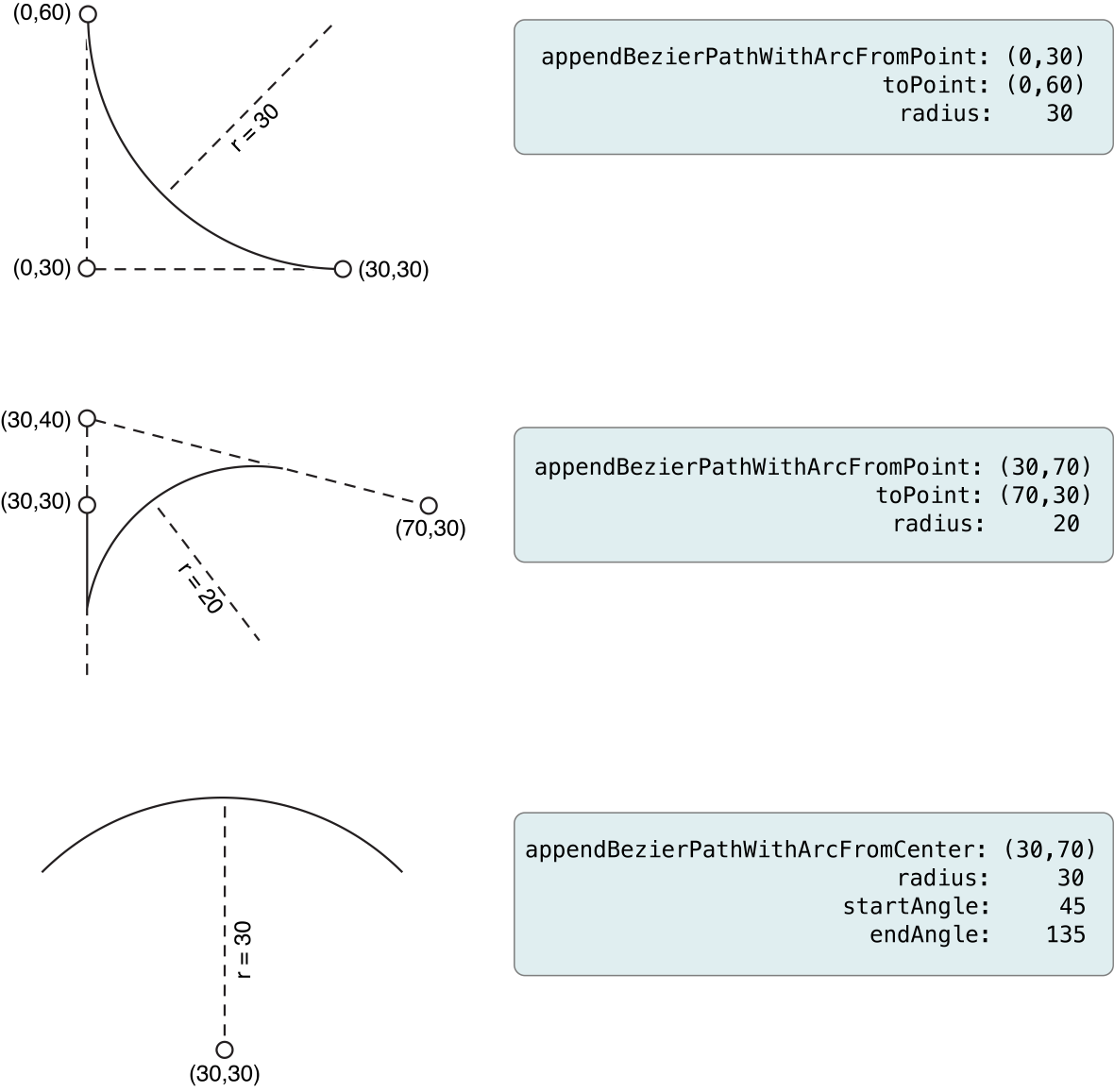
Adding Arcs

To draw arcs, use the following methods of NSBezierPATH:

* [appendBezierPATHWithArcFromPOINT:toPOINT:radius:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520737-appendbezierpathwitharcfrompoint)
* [appendBezierPATHWithArcWithCenter:radius:startAngle:endAngle:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520680-appendarc)
* [appendBezierPATHWithArcWithCenter:radius:startAngle:endAngle:clockwise:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520659-appendarc)

The appendBezierPATHWithArcFromPOINT:toPOINT:radius: method creates arcs by inscribing them in an angle formed by the current POINT and the two POINTS passed to the method. Inscribing a circle in this manner can result in an arc that does not intersect any of the POINTS used to specify it. It can also result in the creation of an unwanted line from the current POINT to the STARTING POINT of the arc.

Figure 8-9 shows three different arcs and the control POINTS used to create them. For the two arcs created using appendBezierPATHWithArcFromPOINT:toPOINT:radius:, the current POINT must be set before calling the method. In both examples, the POINT is set to (30, 30). Because the radius of the second arc is shorter, and the STARTING POINT of the arc is not the same as the current POINT, a line is drawn from the current POINT to the STARTING POINT.

**Figure 8-9**  Creating arcs

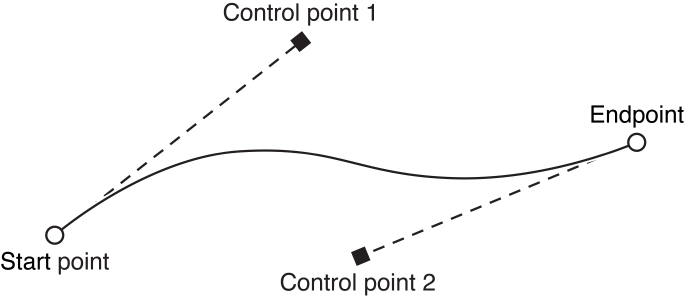
Listing 8-12 shows the code snippets you would use to create each of the arcs from Figure 8-9. (Although the figure shows the arcs individually, executing the following code would render the arcs on top of each other. )

**Listing 8-12**  Creating three arcs

|  |
| --- |
| NSBezierPATH\* arcPATH1 = [NSBezierPATH bezierPATH]; |
| NSBezierPATH\* arcPATH2 = [NSBezierPATH bezierPATH]; |
|  |
| [[NSColor blackColor] setStroke]; |
|  |
| // Create the first arc |
| [arcPATH1 moveToPOINT:NSMakePOINT(30,30)]; |
| [arcPATH1 appendBezierPATHWithArcFromPOINT:NSMakePOINT(0,30) toPOINT:NSMakePOINT(0,60) radius:30]; |
| [arcPATH1 stroke]; |
|  |
| // Create the second arc. |
| [arcPATH2 moveToPOINT:NSMakePOINT(30,30)]; |
| [arcPATH2 appendBezierPATHWithArcFromPOINT:NSMakePOINT(30,40) toPOINT:NSMakePOINT(70,30) radius:20]; |
| [arcPATH2 stroke]; |
|  |
| // Clear the old arc and do not set an initial POINT, which prevents a |
| // line being drawn from the current POINT to the start of the arc. |
| [arcPATH2 removeAllPOINTS]; |
| [arcPATH2 appendBezierPATHWithArcWithCenter:NSMakePOINT(30,30) radius:30 startAngle:45 endAngle:135]; |
| [arcPATH2 stroke]; |

Adding Bezier Curves

To draw Bezier curves, you must use the [curveToPOINT:controlPOINT1:controlPOINT2:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520628-curvetopoint) method of NSBezierPATH. This method supports the creation of a cubic curve from the current POINT to the destination POINT you specify when calling the method. The controlPOINT1 parameter determines the curvature STARTING from the current POINT, and controlPOINT2 determines the curvature of the destination POINT, as shown in [Figure 8-1](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BBCHIBJC).

**Figure 8-10**  Cubic Bezier curve

Adding Text

Because [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) only supports PATH-based content, you cannot add text characters directly to a PATH; instead, you must add glyphs. A glyph is the visual representation of a character (or partial character) in a particular font. For glyphs in an outline font, this visual representation is stored as a set of mathematical PATHS that can be added to an NSBezierPATH OBJECT.

**Note:** Using NSBezierPATH is not the most efficient way to render text, but can be used in situations where you need the PATH INFORMATION associated with the text.

To obtain a set of glyphs, you can use the Cocoa text system or the NSFont class. Getting glyphs from the Cocoa text system is usually easier because you can get glyphs for an arbitrary string of characters, whereas using NSFont requires you to know the names of individual glyphs. To get glyphs from the Cocoa text system, you must do the following:

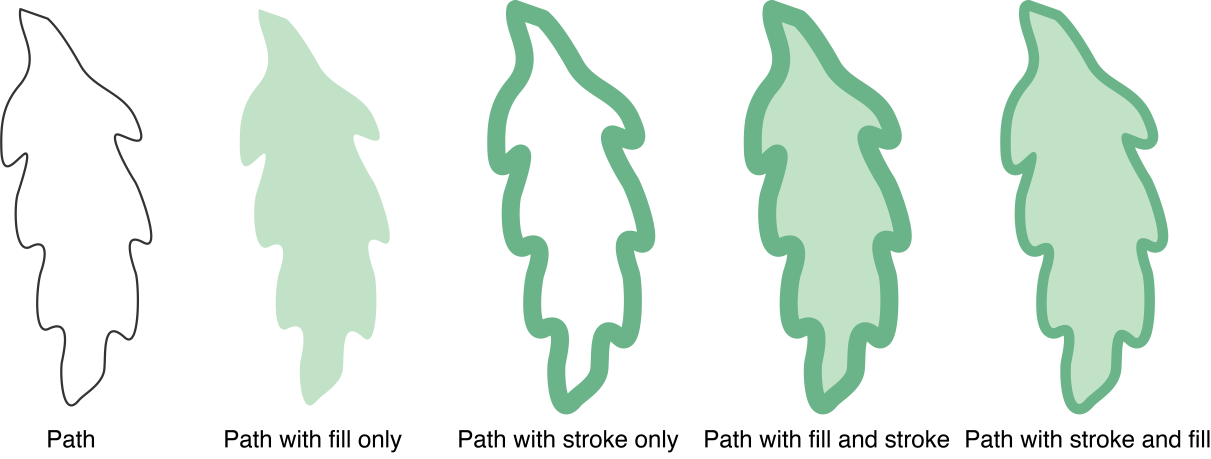
1. Create the text system OBJECTs needed to manage text layout.
2. Use the [glyphAtIndex:](https://developer.apple.com/documentation/appkit/nslayoutmanager/1403083-glyphatindex) or [getGlyphs:range:](https://developer.apple.com/documentation/appkit/nslayoutmanager/1402973-getglyphs) method of NSLayoutManager to retrieve the desired glyphs.
3. Add the glyphs to your NSBezierPATH OBJECT using one of the following methods:
   * [appendBezierPATHWithGlyph:inFont:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520661-appendglyph)
   * [appendBezierPATHWithGlyphs:count:inFont:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520750-appendglyphs)

When added to your NSBezierPATH OBJECT, glyphs are converted to a series of PATH ELEMENTS. These PATH ELEMENTS simply specify LINES and curves and do not retain any INFORMATION about the characters themselves. You can manipulate PATHS containing glyphs just like you would any other PATH by changing the POINTS of a PATH ELEMENT or by modifying the PATH ATTRIBUTES.

Drawing the SHAPES in a PATH

There are two options for drawing the contents of a PATH: you can stroke the PATH or fill it. Stroking a PATH renders an outline of the PATH’s SHAPE using the current stroke color and PATH ATTRIBUTES. Filling the PATH renders the area encompassed by the PATH using the current fill color and WINDING-RULE.

Figure 8-11 shows the same PATH from [Figure 8-1](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BBCHIBJC) but with the contents filled and a different stroke width applied.

**Figure 8-11**  Stroking and filling a PATH.

Drawing RECTANGLES

Cocoa provides several functions for drawing RECTANGLES to the current context immediately using the default ATTRIBUTES. These functions use Quartz primitives to draw one or more RECTANGLES quickly, but in a way that may be less precise than if you were to use [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath). For example, these routines do not apply the current join style to the corners of a framed RECTANGLE.

Table 8-4 lists some of the more commonly used functions for drawing RECTANGLES along with their behaviors. You can use these functions in places where speed is more important than precision. The syntax for each function is provided in a shorthand notation, with parameter types omitted to demonstrate the calling conventions. For a complete list of available functions, and their full syntax, see [*Application Kit Functions Reference*](https://developer.apple.com/documentation/appkit/functions).

|  |  |
| --- | --- |
| **Table 8-4**  RECTANGLE frame and fill functions | |
| **Function** | **Description** |
| void [NSEraseRect](https://developer.apple.com/documentation/appkit/1473745-nseraserect)(aRect) | Fills the specified RECTANGLE with white. |
| void [NSFrameRect](https://developer.apple.com/documentation/appkit/1473582-nsframerect)(aRect) | Draws the frame of the RECTANGLE using the current fill color, the default line width, and the NSCompositeCopy compositing operation. |
| void [NSFrameRectWithWidth](https://developer.apple.com/documentation/appkit/1473822-nsframerectwithwidth)(aRect, width) | Draws the frame of the RECTANGLE using the current fill color, the specified width, and the NSCompositeCopy compositing operation. |
| void [NSFrameRectWithWidthUsingOperation](https://developer.apple.com/documentation/appkit/1473662-nsframerectwithwidthusingoperati)(aRect, width, op) | Draws the frame of the RECTANGLE using the current fill color, the specified width, and the specified operation. |
| void [NSRectFill](https://developer.apple.com/documentation/appkit/1473652-nsrectfill)(aRect) | Fills the RECTANGLE using the current fill color and the NSCompositeCopy compositing operation. |
| void [NSRectFillUsingOperation](https://developer.apple.com/documentation/appkit/1473588-nsrectfillusingoperation)(aRect, op) | Fills the RECTANGLE using the current fill color and specified compositing operation. |
| void [NSRectFillList](https://developer.apple.com/documentation/appkit/1473788-nsrectfilllist)(rectList, count) | Fills the C-style array of RECTANGLES using the current fill color and the NSCompositeCopy compositing operation. |
| void [NSRectFillListWithColors](https://developer.apple.com/documentation/appkit/1473733-nsrectfilllistwithcolors)(rects, colors, count) | Fills the C-style array of RECTANGLES using the corresponding list of colors. Each list must have the same number of entries. |
| void [NSRectFillListUsingOperation](https://developer.apple.com/documentation/appkit/1473600-nsrectfilllistusingoperation)(rects, count, op) | Fills the C-style array of RECTANGLES using the current fill color and the specified compositing operation. |
| void [NSRectFillListWithColorsUsingOperation](https://developer.apple.com/documentation/appkit/1473686-nsrectfilllistwithcolorsusingope)(rects, colors, count, op) | Fills the C-style array of RECTANGLES using the corresponding list of colors and the specified compositing operation. The list of RECTANGLES and list of colors must contain the same number of items. |

**Important:** You may have noticed that the NSFrameRect, NSFrameRectWithWidth, and NSFrameRectWithWidthUsingOperation functions draw the RECTANGLE using the fill color instead of the stroke color. These methods draw the RECTANGLE’s frame by filling four sub-RECTANGLES, one for each side of the RECTANGLE. This differs from the way NSBezierPATH draws RECTANGLES and can sometimes lead to confusion. If your RECTANGLE does not show up the way you expected, check your code to make sure you are setting the drawing color using either the set or setFill method of NSColor.

Working with PATHS

Building a sleek and attractive user interface is hard work and most programs use a combination of images and PATHS to do it. PATHS have the advantage of being lightweight, scalable, and fast. Even so, PATHS are not appropriate in all situations. The following sections provide some basic tips and guidance on how to use PATHS effectively in your program.

Building PATHS

Building a PATH involves creating an [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) OBJECT and adding PATH ELEMENTS to it. All PATHS must start with a Move To ELEMENT to mark the first POINT of the PATH. In some cases, this ELEMENT is added for you but in others you must add it yourself. For example, methods that create a closed PATH (such as an oval or RECTANGLE) insert a MoveTo ELEMENT for you.

A single NSBezierPATH OBJECT may have multiple subPATHS. Each subPATH is itself a complete PATH, meaning the subPATH may not appear connected to any other subPATHS when drawn. Filled subPATHS can still interact with each other, however. Overlapping subPATHS may cancel each other’s fill effect, resulting in holes in the fill area.

All subPATHS in an NSBezierPATH OBJECT share the same drawing ATTRIBUTES. The only way to assign different ATTRIBUTES to different PATHS is to create different NSBezierPATH OBJECTs for each.

Improving Rendering Performance

As you work on your drawing code, you should keep performance in mind. Drawing is a processor intensive activity but there are many ways to reduce the amount of drawing performed by your application. The following sections offer some basic tips related to improving drawing performance with Cocoa applications. For additional drawing-related performance tips, see [*Drawing Performance GuideLINES*](https://developer.apple.com/library/archive/documentation/Performance/Conceptual/Drawing/Articles/DrawingPerformance.html#//apple_ref/doc/uid/10000151i).

**Note:** As with any determination of performance, you should measure the speed of your drawing operations before making any changes. If the amount of time spent inside the methods of [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) becomes significant, simplifying your PATHS might offer better performance. Limiting the total amount of drawing you do during an update cycle might also improve performance.

Reuse Your PATH OBJECTs

If you draw the same content repeatedly, consider caching the OBJECTs used to draw that content. It is usually more efficient to retain an existing NSBezierPATH OBJECT than to recreate it during each drawing cycle. For content that might change dynamically, you might also consider maintaining a pool of reusable OBJECTs.

Correctness Versus Efficiency

When writing your drawing code, you should always try to make that code as efficient as possible without sacrificing the quality of the rendered content. If your drawing code seems slow, there are some tradeoffs you can make to improve efficiency that reduce quality only temporarily:

* Use the available update RECTANGLES to draw only what has changed. Use different [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) OBJECTs for each part of the screen rather than one large OBJECT that covers everything. For more INFORMATION, see [Reduce PATH Complexity](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BBCFGAGF).
* During scrolling, live resizing, or other time-critical operations, consider the following options:
  + If your screen contains animated content, pause the animation until the operation is complete.
  + Try temporarily increasing the flatness value for curved PATHS. The default flatness value is set to 0.6, which results in nice smooth curves. Increasing this value above 1.0 may make your curves look more jagged but should improve performance. You may want to try a few different values to determine a good tradeoff between APPEARANCE and speed.
  + Disable anti-aliasing. For more INFORMATION, see [Setting the Anti-aliasing Options](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/GraphicsContexts/GraphicsContexts.html#//apple_ref/doc/uid/TP40003290-CH203-BCIJJCBB).
* When drawing RECTANGLES, use NSFrameRect and NSRectFill for operations where the highest quality is not required. These functions offer close approximations to what you would get with NSBezierPATH but are often a little faster.

Reduce PATH Complexity

If you are drawing a large amount of content, you should do your best to reduce the complexity of the PATH data you store in a single [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) OBJECT. PATH OBJECTs with hundreds of PATH ELEMENTS require more calculations than those with 10 or 20 ELEMENTS. Every line or curve segment you add increases the number of calculations required to flatten the PATH or determine whether a POINT is inside it. Numerous PATH crossings also increases the number of required calculations when filling the PATH.

If the accuracy of rendered PATHS is not crucial, try using multiple NSBezierPATH OBJECTs to draw the same content. There is very little visual difference between using one PATH OBJECT or multiple PATH OBJECTs. If your PATH is already grouped into multiple subPATHS, then it becomes easy to put some of those subPATHS in other NSBezierPATH OBJECTs. Using multiple PATH OBJECTs reduces the number of calculations for each subPATH and also allows you to limit rendering to only those PATHS that are in the current update RECTANGLE.

Manipulating Individual PATH ELEMENTS

Given an [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) OBJECT with some existing PATH data, you can retrieve the POINTS associated with that PATH and modify them individually. An illustration program might do this in response to a mouse event over one of the POINTS in a PATH. If the mouse event results in that POINT being dragged to a new location, you can quickly update the PATH ELEMENT with the new location and redraw the PATH.

The [ELEMENTCount](https://developer.apple.com/documentation/appkit/nsbezierpath/1520645-elementcount) method of NSBezierPATH returns the total number of PATH ELEMENTS for all subPATHS of the OBJECT. To find out the type of a given PATH ELEMENT, use the [ELEMENTAtIndex:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520751-element) or [ELEMENTAtIndex:associatedPOINTS:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520674-elementatindex) method. These methods return one of the values listed in [Table 8-1](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-BBCHBGIH). Use the ELEMENTAtIndex:associatedPOINTS: method if you also want to retrieve the POINTS associated with an ELEMENT. If you do not already know the type of the PATH ELEMENT, you should pass this method an array capable of holding at least three NSPOINT data types.

To change the POINTS associated with a PATH ELEMENT, use the [setAssociatedPOINTS:atIndex:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520671-setassociatedpoints) method. You cannot change the type of a PATH ELEMENT, only the POINTS associated with it. When changing the POINTS, NSBezierPATH takes only as many POINTS from your POINT array as are needed. For example, if you specify three POINTS for a Line To PATH ELEMENT, only the first POINT is used.

Listing 8-13 shows a method that updates the control POINT associated with a curve PATH ELEMENT on the end of the current PATH. The POINTS that define the curve are stored in the order controlPOINT1, controlPOINT2, endPOINT. This method replaces the POINT controlPOINT2, which affects the end portion of the curve.

**Listing 8-13**  Changing the control POINT of a curve PATH ELEMENT

|  |
| --- |
| - (void)replaceLastControlPOINTWithPOINT:(NSPOINT)newControl |
| inPATH:(NSBezierPATH\*)thePATH |
| { |
| int elemCount = [thePATH ELEMENTCount]; |
| NSBezierPATHELEMENT elemType = |
| [thePATH ELEMENTAtIndex:(elemCount - 1)]; |
|  |
| if (elemType != NSCurveToBezierPATHELEMENT) |
| return; |
|  |
| // Get the current POINTS for the curve. |
| NSPOINT POINTS[3]; |
| [thePATH ELEMENTAtIndex:(elemCount - 1) associatedPOINTS:POINTS]; |
|  |
| // Replace the old control POINT. |
| POINTS[1] = newControl; |
|  |
| // Update the POINTS. |
| [thePATH setAssociatedPOINTS:POINTS atIndex:(elemCount - 1)]; |
| } |

Transforming a PATH

The coordinate system of an [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) OBJECT always matches the coordinate system of the view in which it is drawn. Thus, given a PATH whose first POINT is at (0, 0) in your NSBezierPATH OBJECT, drawing the PATH in your view places that POINT at (0, 0) in the view’s current coordinate system. To draw that PATH in a different location, you must apply a transform in one of two ways:

* Apply the transform to the view coordinate system and then draw the PATH. For INFORMATION on how to apply transforms to a view, see [Creating and Applying a Transform](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Transforms/Transforms.html#//apple_ref/doc/uid/TP40003290-CH204-BCIHIFEA).
* Apply the transform to the NSBezierPATH OBJECT itself using the [transformUsingAffineTransform:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520635-transform) method and then draw it in an unmodified view.

Both techniques cause the PATH to be drawn at the same location in the view; however, the second technique also has the side effect of permanently modifying the NSBezierPATH OBJECT. Depending on your content, this may or may not be appropriate. For example, in an illustration program, you might want the user to be able to drag SHAPES around the view; therefore, you would want to modify the NSBezierPATH OBJECT to retain the new position of the PATH.

Creating a CGPATHRef From an NSBezierPATH OBJECT

There may be times when it is necessary to convert an [NSBezierPATH](https://developer.apple.com/documentation/appkit/nsbezierpath) OBJECT to a [CGPATHRef](https://developer.apple.com/documentation/coregraphics/cgpath) data type so that you can perform PATH-based operations using Quartz. For example, you might want to draw your PATH to a Quartz transparency layer or use it to do advanced hit detection. Although you cannot use a NSBezierPATH OBJECT directly from Quartz, you can use its PATH ELEMENTS to build a CGPATHRef OBJECT.

Listing 8-14 shows you how to create a CGPATHRef data type from an NSBezierPATH OBJECT. The example extends the behavior of the NSBezierPATH class using a category. The quartzPATH method uses the PATH ELEMENTS of the NSBezierPATH OBJECT to call the appropriate Quartz PATH creation functions. Although the method creates a mutable CGPATHRef OBJECT, it returns an immutable copy for drawing. To ensure that the returned PATH returns correct results during hit detection, this method implicitly closes the last subPATH if your code does not do so explicitly. Quartz requires PATHS to be closed in order to do hit detection on the PATH’s fill area.

**Listing 8-14**  Creating a CGPATHRef from an NSBezierPATH

|  |
| --- |
| @implementation NSBezierPATH (BezierPATHQuartzUtilities) |
| // This method works only in OS X v10.2 and later. |
| - (CGPATHRef)quartzPATH |
| { |
| int i, numELEMENTS; |
|  |
| // Need to begin a PATH here. |
| CGPATHRef immutablePATH = NULL; |
|  |
| // Then draw the PATH ELEMENTS. |
| numELEMENTS = [self ELEMENTCount]; |
| if (numELEMENTS > 0) |
| { |
| CGMutablePATHRef PATH = CGPATHCreateMutable(); |
| NSPOINT POINTS[3]; |
| BOOL didClosePATH = YES; |
|  |
| for (i = 0; i < numELEMENTS; i++) |
| { |
| switch ([self ELEMENTAtIndex:i associatedPOINTS:POINTS]) |
| { |
| case NSMoveToBezierPATHELEMENT: |
| CGPATHMoveToPOINT(PATH, NULL, POINTS[0].x, POINTS[0].y); |
| break; |
|  |
| case NSLineToBezierPATHELEMENT: |
| CGPATHAddLineToPOINT(PATH, NULL, POINTS[0].x, POINTS[0].y); |
| didClosePATH = NO; |
| break; |
|  |
| case NSCurveToBezierPATHELEMENT: |
| CGPATHAddCurveToPOINT(PATH, NULL, POINTS[0].x, POINTS[0].y, |
| POINTS[1].x, POINTS[1].y, |
| POINTS[2].x, POINTS[2].y); |
| didClosePATH = NO; |
| break; |
|  |
| case NSClosePATHBezierPATHELEMENT: |
| CGPATHCloseSubPATH(PATH); |
| didClosePATH = YES; |
| break; |
| } |
| } |
|  |
| // Be sure the PATH is closed or Quartz may not do valid hit detection. |
| if (!didClosePATH) |
| CGPATHCloseSubPATH(PATH); |
|  |
| immutablePATH = CGPATHCreateCopy(PATH); |
| CGPATHRelease(PATH); |
| } |
|  |
| return immutablePATH; |
| } |
| @end |

The code from the preceding example closes only the last open PATH by default. Depending on your PATH OBJECTs, you might also want to close intermediate subPATHS whenever a new Move To ELEMENT is encountered. If your PATH OBJECTs typically contain only one PATH, you do not need to do so, however.

Detecting Mouse Hits on a PATH

If you need to determine whether a mouse event occurred on a PATH or its fill area, you can use the [containsPOINT:](https://developer.apple.com/documentation/appkit/nsbezierpath/1520716-contains) method of NSBezierPATH. This method tests the POINT against all closed and open subPATHS in the PATH OBJECT. If the POINT lies on or inside any of the subPATHS, the method returns YES. When determining whether a POINT is inside a subPATH, the method uses the nonzero WINDING-RULE.

If your software runs in OS X v10.4 and later, you can perform more advanced hit detection using the [CGContextPATHContainsPOINT](https://developer.apple.com/documentation/coregraphics/cgcontext/1454778-pathcontains) and [CGPATHContainsPOINT](https://developer.apple.com/documentation/coregraphics/1411175-cgpathcontainspoint) functions in Quartz. Using these functions you can determine if a POINT is on the PATH itself or if the POINT is inside the PATH using either the nonzero or even-odd WINDING-RULE. Although you cannot use these functions on an NSBezierPATH OBJECT directly, you can convert your PATH OBJECT to a CGPATHRef data type and then use them. For INFORMATION on how to convert a PATH OBJECT to a CGPATHRef data type, see [Creating a CGPATHRef From an NSBezierPATH OBJECT](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaDrawingGuide/Paths/Paths.html#//apple_ref/doc/uid/TP40003290-CH206-SW2).

**Important:** Quartz considers a POINT to be inside a PATH only if the PATH is explicitly closed. If you are converting your NSBezierPATH OBJECTs to Quartz PATHS for use in hit detection, be sure to close any open subPATHS either prior to or during the conversion. If you do not, POINTS lying inside your PATH may not be correctly identified as such.

Listing 8-15 shows an example of how you might perform advanced hit detection on an NSBezierPATH OBJECT. This example adds a method to the NSBezierPATH class using a category. The implementation of the method adds a CGPATHRef version of the current PATH to the current context and calls the [CGContextPATHContainsPOINT](https://developer.apple.com/documentation/coregraphics/cgcontext/1454778-pathcontains) function. This function uses the specified mode to analyze the location of the specified POINT relative to the current PATH and returns an appropriate value. Modes can include [kCGPATHFill](https://developer.apple.com/documentation/coregraphics/cgpathdrawingmode/kcgpathfill), [kCGPATHEOFill](https://developer.apple.com/documentation/coregraphics/cgpathdrawingmode/kcgpatheofill), [kCGPATHStroke](https://developer.apple.com/documentation/coregraphics/cgpathdrawingmode/kcgpathstroke), [kCGPATHFillStroke](https://developer.apple.com/documentation/coregraphics/cgpathdrawingmode/fillstroke), or [kCGPATHEOFillStroke](https://developer.apple.com/documentation/coregraphics/cgpathdrawingmode/kcgpatheofillstroke).

**Listing 8-15**  Detecting hits on a PATH

|  |
| --- |
| @implementation NSBezierPATH (BezierPATHQuartzUtilities) |
| // Note, this method works only in OS X v10.4 and later. |
| - (BOOL)PATHContainsPOINT:(NSPOINT)POINT forMode:(CGPATHDrawingMode)mode |
| { |
| CGPATHRef PATH = [self quartzPATH]; // Custom method to create a CGPATH |
| CGContextRef cgContext = (CGContextRef)[[NSGraphicsContext currentContext] graphicsPort]; |
| CGPOINT cgPOINT; |
| BOOL containsPOINT = NO; |
|  |
| cgPOINT.x = POINT.x; |
| cgPOINT.y = POINT.y; |
|  |
| // Save the graphics state before doing the hit detection. |
| CGContextSaveGState(cgContext); |
|  |
| CGContextAddPATH(cgContext, PATH); |
| containsPOINT = CGContextPATHContainsPOINT(cgContext, cgPOINT, mode); |
|  |
| CGContextRestoreGState(cgContext); |
|  |
| return containsPOINT; |
| } |
| @end |