ShapeGen 2-D Graphics Library User's Guide

Jerry R. VanAken

November 4, 2023

This user's guide presents an overview of the ShapeGen 2-D graphics library, plus reference pages for the functions in the library and numerous code examples.

Breaking change – September 15, 2023

Minor changes have been made to the ShapeGen API. The following functions are affected: `ShapeGen::FillPath`, `ShapeGen::SetClipRegion`, `ShapeGen::SetMaskRegion`, `ShapeGen::PolyBezier2`, `ShapeGen::PolyBezier3`, `ShapeGen::PolyEllipticSpline`, and `ShapeGen::PolyLine`.

The changes to `FillPath`, `SetClipRegion`, and `SetMaskRegion` enable modifications to the clipping region to be specified as either filled or stroked shapes.

The changes to `PolyBezier2`, `PolyBezier3`, `PolyEllipticSpline`, and `PolyLine` switch the ordering of the two function parameters to conform to common practice.

Table of Contents

ShapeGen overview

Introduction

This GitHub project contains the C++ source code for the ShapeGen 2-D graphics library. ShapeGen draws both filled and stroked shapes. It is lightweight and highly portable. The library's rendering code provides

antialiasing and alpha blending, and can fill shapes with solid colors, patterns, linear gradients, and radial gradients.

Some 2-D graphics libraries achieve portability by sitting atop a portable graphics library such as OpenGL. Or perhaps they run on Cairo, which in turn runs on PixMan. The resulting dependencies complicate the build process and create bloated executables.

The ShapeGen library code has only two dependencies. The library requires

- a C++ compiler
- the Standard C Library

To achieve portability, ShapeGen draws directly to pixel memory, with absolutely no reliance on intermediaries to do the actual rendering.

Of course, an operating system such as Linux or Windows does not allow an application program to draw directly to the on-screen memory for a windowed display that is shared with other applications. To run in these systems, a small amount of platform-dependent code is required to copy an image to on-screen memory after the library has finished rendering the image. All such platform-dependent code in this project is isolated in the `winmain.cpp` and `sdlmain.cpp` files.

Included in this project is the source code for a demo program that shows off the capabilities of the ShapeGen library. Also included is a simple ShapeGen-based SVG file viewer. Make files are included to build and run these example applications in Linux and Windows. Both applications require a true-color graphics display and run in a 1280x960 window.

The core of the library is the ShapeGen class, which implements a relatively simple but powerful 2-D *polygonal shape generator*. Users call ShapeGen functions to create arbitrarily complex geometric shapes that are then rendered and shown on a graphics display. The user constructs shape boundaries by connecting geometric primitives such as line segments, spline curves, and circular or elliptic arcs. The functions implemented by the ShapeGen class are described in detail in the ShapeGen functions reference section in this user's guide.

A 2-D graphics system is naturally partitioned into two parts. The polygonal shape generator is the part of the system that maps mathematically defined shapes onto an *x-y* coordinate grid that represents the positions of pixels on a graphics display. However, the shape generator stops short of actually touching the pixels, which is the job of a separate component, the *renderer*. The shape generator tells the renderer what shapes to draw but leaves all platform-specific and device-dependent operations on pixels to the renderer. Thus, the shape generator remains free of all such dependencies.

But a renderer can also be designed in such a way that the bulk of its code is free of platform and device dependencies. To minimize the amount of platform-dependent code, the two true-color renderers in this project draw shapes directly to a *back buffer*, which is an off-screen, window-backing buffer in memory. After the image in the back buffer has been rendered, it is copied by a BitBlt (bit-block transfer) operation to the target window in the on-screen memory. Although the code that renders shapes to the back buffer is platform-independent, this BitBlt must be implemented by the underlying platform, as previously mentioned.

The most basic task for a renderer is to fill shapes with solid colors. But for graphics systems with true-color displays, the *enhanced renderer* in this project provides antialiasing and alpha blending, and can fill shapes with tiled patterns, linear gradients, and radial gradients. For descriptions of the functions implemented by

the enhanced renderer in this project, see the EnhancedRenderer functions reference section in this user's guide.

Also included in this project is a *simple renderer* that does simple solid-color fill operations, but with no antialiasing. The simple renderer is faster than the enhanced renderer for filling large regions with solid colors. The source code for the simple renderer in this project is quite small (see the `renderer.cpp` file in the project's main directory).

For a system whose graphics display device has limited color capabilities and does not support true-color pixel formats, a standalone simple renderer is sufficient. As shown in Appendix A in this user's guide, just a few lines of code are required to implement this type of renderer. Thus, the ShapeGen library can be ported to such a system with minimal effort.

A ShapeGen-based application program calls the `ShapeGen::SetRenderer` function to switch between the simple renderer and the enhanced renderer.

The ShapeGen library additionally supports clipping regions of arbitrary shape and complexity. To simplify renderer design, renderers are *not* responsible for clipping. All shapes that the ShapeGen object passes to the renderer have already been clipped.

To form a complete, functioning 2-D graphics engine, a well-defined interface connects a renderer to a ShapeGen object. For more information, see the ShapeGen-renderer interface topic in this user's guide.

In comparison with larger and more complex open-source graphics software, the small size and relative simplicity of the source code for the ShapeGen library makes it relatively easy to modify and build upon. The clean separation of the ShapeGen library and renderer implementation enables developers to more easily add new rendering capabilities and to port the library to new platforms and hardware.

Developers may find this GitHub project's ShapeGen library and example renderers to be sufficient, without modification, for many graphics applications.

For a high-level overview of ShapeGen capabilities and internal operation, see the article titled *ShapeGen: A lightweight, open-source 2-D graphics library written in C++* at the ResearchGate website.

Building the demo and svgview example apps

This GitHub project includes the source code for the following two ShapeGen-based example apps:

- The *demo* app first shows off the graphics capabilities of the ShapeGen library. Then it steps through the graphics output from the code examples in this user's guide.
- The *svgview* app is a simple SVG file viewer, which parses and displays a list of SVG files. The user lists the names of the SVG files on the command line, separated by spaces. No SVG files are included in this project, but Appendix B in this user's guide contains links to a number of selected SVG files available at websites such as w3.org and wikipedia.org.

The `demo` and `svgview` example apps can run on the SDL2 (Simple DirectMedia Library, version 2) API in Linux and Windows. Both apps can also run on the Win32/GDI API in Windows.

Follow these steps to build the `demo` and `svgview` apps to run on the SDL2 API in Linux:

- open a Linux terminal window,

- install the SDL2 library (`libsdl2-2.0-0`),
- download (`git clone`) the ShapeGen project from github.com,
- change to the project's `linux-sdl` subdirectory,
- and enter the `make` command.

For more detailed instructions, see the `README.md` file in the `linux-sdl` subdirectory of this project.

Follow these steps to build the `demo.exe` and `svgview.exe` apps to run on the SDL2 API in Windows:

- open a Windows command window,
- install the SDL2 library, download the ShapeGen project from github.com,
- change to the project's `windows-sdl` subdirectory,
- and enter the `nmake` command.

Make sure the `path` environment variable includes the location of the SDL2 library files. For more detailed instructions, see the `README.md` file in the `windows-sdl` subdirectory of this project.

Follow these steps to build the `demo.exe` and `svgview.exe` apps to run on the Win32/GDI API in Windows:

- open a Windows command window,
- download the ShapeGen project from github.com,
- change to the project's `windows-gdi` subdirectory,
- and enter the `nmake` command.

For more detailed instructions, see the `README.md` file in the `windows-gdi` subdirectory of this project.

Programming interface

The ShapeGen library's application programming interface is similar to that of other 2-D graphics software systems, which follow the path-construction model that originated with PostScript in the 1980s.

The PostScript page description language is the archetypical 2-D graphics interface. For comparison, the following table lists a number of ShapeGen library functions and the corresponding PostScript path-construction operators.

ShapeGen function	PostScript operator
`BeginPath`	newpath
`Bezier3`	curveto
`CloseFigure`	closepath
`EllipticArc`	arc
`FillPath`	fill
`GetBoundingBox`	pathbbox
`GetCurrentPoint`	currentpoint
`InitClipRegion`	initclip
`Line`	lineto

ShapeGen function	PostScript operator
`Move`	moveto
`SetClipRegion`	clip
`SetFlatness`	setflat
`SetLineDash`	setdash
`SetLineEnd`	setlinecap
`SetLineJoin`	setlinejoin
`SetLineWidth`	setlinewidth
`SetMiterLimit`	setmiterlimit
`StrokePath`	stroke

The PostScript path-construction operators are described in chapter 8 of the *PostScript Language Reference Manual*, Second Edition, 1990. For detailed descriptions of the functions implemented by the ShapeGen class, see the ShapeGen functions reference section.

A lightweight and portable graphics library

The ShapeGen 2-D graphics library in this GitHub project is lightweight and portable. The platform-independent source code for the ShapeGen class consists of six well-commented C++ source files and two header files, which are listed in the following table.

File name	File size (bytes)	Description
`arc.cpp`	15K	Constructs paths for ellipses, elliptic arcs, and elliptic splines
`curve.cpp`	12K	Constructs paths for cubic and quadratic curves
`edge.cpp`	35K	Manages lists of polygonal edges, does clipping, and drives renderer
`path.cpp`	25K	Performs basic path-management functions
`stroke.cpp`	31K	Converts paths into stroked lines and curves
`thinline.cpp`	7K	Converts paths into *thin* stroked lines and curves
`shapegen.h`	11K	Header file for ShapeGen public interface
`shapepri.h`	12K	Header file for ShapeGen internal interfaces

The files in this table implement all the functions described in the ShapeGen functions reference section.

This project includes example renderers that are used by the ShapeGen demo program and the SVG file viewer. These renderers use a 32-bit pixel format (see the Enhanced renderer pixel format topic) and are augmented with paint generators that fill shapes with solid colors, tiled patterns, linear gradients, and radial gradients. The platform-independent source code for the renderers implemented in this project consists of three well-commented C++ source files and one header file, which are listed in the following table.

File name	File size (bytes)	Description
`gradient.cpp`	26K	Implements paint generators for linear-gradient and radial-gradient fills
`pattern.cpp`	13K	Implements a paint generator for tiled-pattern fills
`renderer.cpp`	32K	Implements a pair of example renderers that are used by the ShapeGen demo program and SVG file viewer
`renderer.h`	11K	Header file for renderer and paint generator interfaces

The files in this table implement all the functions described in the EnhancedRenderer functions reference section.

Also included in this project is example application code that demonstrates the graphics capabilities of the ShapeGen library. The following table lists the source files for the ShapeGen demo program and SVG file viewer.

File name	File size (bytes)	Description
`demo.cpp`	144K	The ShapeGen demo program, plus all the code examples in the *ShapeGen User's Guide*
`textapp.cpp`	62K	A simple graphical text application, with glyphs for all printing ASCII characters
`bmpfile.cpp`	14K	A rudimentary BMP file reader used by the demo program to do pattern fills
`svgview.cpp`	10K	A simple SVG file viewer

nanosvg.h	96K	Modified version of M. Mononen's single-header SVG file parser
alfablur.cpp	12K	An alpha-blurring filter used to render drop shadows
demo.h	7K	Header file for interfaces used in demo and SVG viewer programs

The ShapeGen demo program and SVG file viewer are designed to run on a system with a true-color graphics display with 24-bit or 32-bit pixels, but are otherwise platform-independent. The programs render to a 1280x960 window. Every pixel displayed in the demo program – including all geometric shapes and text – is drawn by the ShapeGen 2-D graphics library.

Although most of the code in this GitHub project is platform-independent, a relatively small amount of platform-dependent code is required to run the ShapeGen demo program and SVG file viewer in Windows and Linux. Specifically, these programs run on the Win32/GDI API in Windows, and on the SDL2 (Simple DirectMedia Library, version 2) API in Linux and Windows. The following table lists all the source files that contain platform-dependent code.

File name	File size (bytes)	Description
winmain.cpp	8K	Implements all the platform-dependent functions needed to run the ShapeGen library on the Win32/GDI API in Windows
sdlmain.cpp	9K	Implements all the platform-dependent functions needed to run the ShapeGen library on the SDL2 API in Linux and Windows

The winmain.cpp file is located in the project's `windows-gdi` subdirectory. The `windows-sdl` and `linux-sdl` subdirectories contain nearly identical copies of `sdlmain.cpp`.

Paths and figures

In both ShapeGen and PostScript, the programmer describes a shape by constructing a *path* to specify the boundary of the shape. For example, a simple path might consist of three points that describe a triangle.

However, a composite path is required to describe a more complex shape. To describe a complex polygonal shape that contains holes and disjoint regions, a path is composed of several plane *figures* (the term used by ShapeGen) or *subpaths* (the PostScript term). Each figure or subpath contains a list of points that describes a sequence of connected boundary segments.

A path is implemented as a simple display list.

In PostScript, a path is a sequence of `lineto` and `curveto` segments. (Circular arcs are approximated with `curveto` segments.) Before a PostScript path can be rendered, it must be *flattened* – that is, each `curveto` segment must be replaced with a sequence of `lineto` segments that approximates the ideal curve.

ShapeGen paths, on the other hand, consist only of line segments. A `ShapeGen::EllipticArc` function call, for example, immediately flattens an arc before adding it to the path.

The ShapeGen approach is simpler but a PostScript path might take less time to transfer over a communications link, or better adapt to a target display device whose resolution is not known in advance. The PostScript scheme is less compelling if the path is to be constructed and then immediately rendered on the same computer.

The PostScript `fill` and `stroke` operators implicitly perform a `newpath` operation after filling or stroking the current path. The path can be preserved across a `fill` or `stroke` operation only by explicitly saving a copy of the path before the operation and then restoring the path afterward. In contrast, ShapeGen paths are reusable: a `ShapeGen::FillPath` or `ShapeGen::StrokePath` function call leaves the path intact. To begin a new path after a `FillPath` or `StrokePath` call, a ShapeGen user calls the `ShapeGen::BeginPath` function.

Current point

A number of ShapeGen path-construction functions rely on the concept of a *current point*. For example, the `ShapeGen::Line` function constructs a line segment that starts at the current point. The line segment ends at the point specified as an input parameter to the function, and this point becomes the new current point when the function returns. PostScript also defines a current point, but with subtle differences.

In ShapeGen, the current point is defined as the point most recently added to the current *figure* (subpath) in the current path. The current point is undefined if the current *figure* is empty. PostScript, on the other hand, defines the current point to be the point most recently added to the current *path*. The current point is undefined if the current *path* is empty.

A ShapeGen user can start a new, empty figure in a composite path that already contains several completed (or *finalized*) figures. Because the new figure is empty, the current point is undefined.

In contrast, a PostScript user encounters an empty subpath only after starting a new, empty path. This is the only time that the current point is undefined.

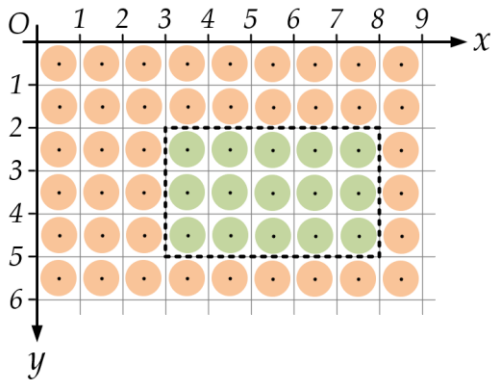
For example, both the `ShapeGen::EllipticArc` function and PostScript `arc` operator can draw a circular arc that starts at a specified angle. These primitives operate under similar rules. Namely, if the current point is defined, then a line segment is constructed from the current point to the starting point of the arc. Otherwise, the arc starting point becomes the first point in the new figure (in ShapeGen) or new path (in PostScript).

To draw a series of unconnected arcs without preceding line segments, a PostScript user must start a new path for each arc. A ShapeGen user, however, can construct all the arcs in a single, composite path.

In addition to the *current point*, ShapeGen defines a *first point*, which is the initial point in the current figure. A user calls the `ShapeGen::GetFirstPoint` and `ShapeGen::GetCurrentPoint` functions to retrieve the first point and current point, respectively. For example, after calling the `EllipticArc` function to add an arc to an empty figure, the user can retrieve the arc starting and ending points by calling the `GetFirstPoint` and `GetCurrentPoint` functions.

ShapeGen coordinates

The ShapeGen coordinate system maps *x*-*y* coordinates directly to pixels on the graphics display. The boundaries between pixels always fall on integer *x*-*y* coordinates. As shown in the following diagram, *x* coordinate values increase to the right and *y* coordinate values increase in the downward direction.



In this diagram, pixel centers (shown as black dots) are located halfway between integer *x*-*y* coordinates. Because the boundaries between pixels always fall on integer coordinates, users don't need to worry about which pixels would get drawn if the boundaries of simple shapes like rectangles were to pass through pixel centers.

For example, in the preceding diagram, a filled green rectangle is rendered over an orange background. The *x*-*y* coordinates at the top-left corner of the rectangle are (3, 2). The width of the rectangle is 5 pixels, and the height is 3 pixels.

By default, application programs use *integer* coordinate values to describe shapes to ShapeGen drawing functions. However, some programs might require subpixel precision in the coordinates they supply to ShapeGen functions. For example, a program that needs to smoothly plot a special curve[1] can specify the curve as a sequence of short line segments. But if these line segments are forced to start and end on integer coordinates, the curve will, on close inspection, appear irregular rather than smooth.

To address this issue, a ShapeGen user can call the `ShapeGen::SetFixedBits` function to switch from the default, integer coordinate format to a fixed-point coordinate format. While the new format is in effect, coordinates supplied by the user to ShapeGen functions are interpreted as fixed-point values.

In a call to `SetFixedBits`, the caller specifies the number of bits to use for the *fractional* part of a coordinate value – that is, the number of bits to the right of the binary point in the fixed-point representation. For example, an argument value of 8 specifies a fixed-bit format with 8 bits of fraction. An argument value of 16 specifies 16 bits of fraction. Calling `SetFixedBits` with an argument value of 0 restores the default, integer coordinate format.

Device clipping rectangle

ShapeGen confines all drawing to the interior of a *device clipping rectangle*. This rectangle is a direct mapping from ShapeGen's *x*-*y* coordinates to the drawing area on the graphics display. The device clipping rectangle

[1] That is, a type of curve that is not directly supported by the ShapeGen library. The library supplies functions for drawing Beziér curves and circular or elliptic arcs. However, a user might need to plot a special mathematical curve with the same geometric fidelity that ShapeGen provides for its built-in curves and arcs. The `SetFixedBits` function fills that need.

typically represents the client drawing area in the target window (or viewport) on the screen. Or it might encompass the entire screen of a dedicated graphics display.

The device clipping rectangle is specified by four integers, (*x*, *y*, *w*, *h*). The *x* and *y* values specify the horizontal and vertical displacements, in pixels, of the rectangle's top-left corner from the ShapeGen coordinate origin. The *w* and *h* values are the width and height, in pixels, of the device clipping rectangle; these are typically the dimensions of the window or viewport on the graphics display. Thus, the device clipping rectangle specifies the region of the ShapeGen coordinate space that the viewer sees on the display.

Frequently, the device clipping rectangle's *x* and *y* coordinates are both zero, in which case the origin of the ShapeGen coordinate system coincides with the top-left corner of the window. However, this rectangle's *x* and *y* coordinates can be specified as nonzero values to enable scrolling and panning through a virtual 2-D image that is larger than the window (see the `ShapeGen::SetScrollPosition` reference topic). ShapeGen's automatic clipping prevents drawing from occurring outside the window.

The user might find this method of scrolling and panning to be convenient for inspecting images that are larger than the drawing area on the screen. But because the image must be redrawn each time the position of the device clipping rectangle changes, this method is not by itself sufficient to provide smooth animation.

The user can call the `ShapeGen::SetClipRegion` and `ShapeGen::SetMaskRegion` functions to modify the shape of the clipping region inside the device clipping rectangle, but no clipping region ever extends beyond the bounds of the device clipping rectangle. To restore the clipping region to the current device clipping rectangle, the user calls the `ShapeGen::ResetClipRegion` function.

The device clipping rectangle is never undefined during the lifetime of a ShapeGen object. The ShapeGen class constructor requires a device clipping rectangle as an input parameter, so that an initial clipping region is defined when a ShapeGen object is created. The width and height of the device clipping rectangle can later be changed, if necessary, by calling the `ShapeGen::InitClipRegion` function. The position in ShapeGen *x*-*y* coordinate space of the top-left corner of the device clipping rectangle can be changed by calling the `ShapeGen::SetScrollPosition` function.

Scan conversion

ShapeGen paths can specify the boundaries of arbitrarily complex polygonal shapes. These shapes can contain holes and disjoint regions. Boundaries can self-intersect. Paths can be filled or stroked.

Each pair of adjacent points in a path specifies a polygonal edge. The edge has a direction. The direction arrow points from the edge's first point to its second point.

ShapeGen supports the same two polygon-fill rules as PostScript:

- nonzero winding number
- even-odd

In both PostScript and ShapeGen, the user can choose either of these rules to construct a *filled* path. No user-supplied fill rule is required for *stroked* paths, but ShapeGen uses the nonzero winding number rule internally to construct stroked paths.

ShapeGen's scan conversion rules are different from PostScript's. First, if the ShapeGen library is connected to a simple renderer (with no antialiasing or alpha blending), a pixel is treated as part of the interior of a

polygon if the center of the pixel lies inside the polygon boundary. If the boundary passes precisely through the center of a pixel, the pixel belongs to the filled region below and to the right of the pixel center.

If the ShapeGen library is instead connected to an enhanced renderer, each pixel is partitioned into subpixels to enable antialiasing. In this case, the scan conversion rules are the same as before, except that they are applied to subpixels rather than pixels.

As previously explained, ShapeGen pixel coordinates are specified, by default, as integers, but an application program can call the `ShapeGen::SetFixedBits` function to switch to using fixed-point coordinates that have up to 16 bits of subpixel precision.

Creating a ShapeGen object

The programming interface for the ShapeGen 2-D graphics library is defined in public header files `shapegen.h` and `renderer.h` in this GitHub project.

An object of the ShapeGen class is created by calling the `CreateShapeGen` function, which is declared in `shapegen.h`, as follows:

```
ShapeGen* CreateShapeGen(Renderer *renderer, const SGRect& cliprect);
```

The second parameter, cliprect, is an `SGRect` structure that specifies the device clipping rectangle, as previously described.

The first parameter is a pointer to a renderer, which is an object that accepts simple drawing commands from the ShapeGen object and then renders shapes into a two-dimensional array of pixels. A renderer is an instance of a class, such as `SimpleRenderer` or `EnhancedRenderer`, that is derived from the `Renderer` base class. For more information, see the Renderers included in this project topic.

For detailed descriptions of the functions implemented by the ShapeGen class, see the ShapeGen functions reference section.

To create a `SimpleRenderer` or `EnhancedRenderer` object, call the `CreateSimpleRenderer` function or `CreateEnhancedRenderer` function. These functions are declared in `renderer.h`, as follows:

```
SimpleRenderer* CreateSimpleRenderer(const PIXEL_BUFFER *pixbuf);
EnhancedRenderer* CreateEnhancedRenderer(const PIXEL_BUFFER *pixbuf);
```

The `pixbuf` parameter points to a `PIXEL_BUFFER` structure that describes the two-dimensional array of pixels into which the renderer will draw shapes. Typically, this pixel array is the backing buffer for a graphics display. The `PIXEL_BUFFER` structure is defined in `renderer.h`, as follows:

```
struct PIXEL_BUFFER
{
    COLOR *pixels;  // pointer to 2-D array of pixels
    int width;      // width of pixel buffer in pixels
    int height;     // height of pixel buffer in pixels
    int depth;      // number of bits per pixel
    int pitch;      // pitch of pixel buffer in bytes
};
```

The SimpleRenderer class is, well, simple — its application interface consists of a single function, SetColor, which specifies a solid color for filling shapes. However, the EnhancedRenderer interface provides a number of additional capabilities. For detailed descriptions of the functions implemented by the EnhancedRenderer class, see the EnhancedRenderer functions reference section.

The CreateShapeGen function requires, as a call parameter, a pointer to an existing Renderer object, which means that a Renderer object must be created before a ShapeGen object can be created.

In the following code example, assume that the MyTest function is called first. This function creates a SimpleRenderer object and a ShapeGen object and uses these objects to draw to the graphics display. MyTest fills the background with white and then calls a second function, MySub, that draws a red rectangle over a portion of the background.

```
void MySub(ShapeGen *sg, SGRect& rect)
{
    sg->BeginPath();
    sg->Rectangle(rect);
    sg->FillPath();
}

void MyTest(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SimpleRenderer *rend = CreateSimpleRenderer(&bkbuf);
    ShapeGen *sg = CreateShapeGen(rend, clip);
    SGRect rect = { 100, 80, 250, 160 };

    sg->BeginPath();
    sg->Rectangle(clip);
    rend->SetColor(RGBX(255,255,255));  // white
    sg->FillPath();

    rend->SetColor(RGBX(255,40,20));  // red
    MySub(sg, rect);

    delete rend;
    delete sg;
}
```

The MyTest function takes two input parameters. The first, bkbuf, is a structure describing the backing buffer for the graphics display and is required as the argument for the CreateSimpleRenderer function call. The second parameter, clip, is the device clipping rectangle that is required for the CreateShapeGen function call. The calling program must ensure that this clipping rectangle fits within the bounds of the backing buffer defined in bkbuf.

Just before the MyTest function returns, it deletes the SimpleRenderer and ShapeGen objects. Neglecting to do so would cause the memory allocated for these objects to leak.

Of course, it's easy enough for a programmer to forget to delete these objects. A straightforward way to reduce the risk of leaks is to use smart pointers. A smart pointer can encapsulate a ShapeGen or Renderer object to make this object available to an application program, and then automatically delete the object when it is no longer needed.

In the following code example, a modified version of the MyTest function, MyTest2, uses smart pointers to encapsulate the SimpleRenderer and ShapeGen objects. The two smart pointer objects, rend and sg, are

defined within the context of the MyTest2 function. Thus, when the MyTest2 function returns and the two smart pointers go out of scope, the smart pointers' destructors are automatically called. These destructors delete the encapsulated Renderer or ShapeGen object, thereby avoiding a potential leak.

```
void MySub(ShapeGen *sg, SGRect& rect)
{
    sg->BeginPath();
    sg->Rectangle(rect);
    sg->FillPath();
}

void MyTest2(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<SimpleRenderer> rend(CreateSimpleRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*rend), clip));
    SGRect rect = { 100, 80, 250, 160 };

    sg->BeginPath();
    sg->Rectangle(clip);
    rend->SetColor(RGBX(255,255,255));  // white
    sg->FillPath();

    rend->SetColor(RGBX(0,120,255));  // blue
    MySub(&(*sg), rect);
}
```

The MyTest2 example uses the SmartPtr class template defined in shapegen.h. Of course, you might prefer to use another smart pointer, such as the unique_ptr class template in the C++ Standard Library. However, the SmartPtr template is used here to avoid introducing additional dependencies into the ShapeGen library build.

Here is the definition of the SmartPtr class template from shapegen.h:

```
template <class T> class SmartPtr {
    T* _ptr;
public:
    explicit SmartPtr(T* ptr = 0) { _ptr = ptr; }
    ~SmartPtr() { delete _ptr; }
    T* operator->() { return _ptr; }
    T& operator*() { return *_ptr; }
};
```

To serve as a smart pointer, the SmartPtr class overloads the **->** operator so that a SmartPtr object (such as rend or sg in the previous code example) can be used as a pointer to the encapsulated (e.g., Renderer or ShapeGen) object. And to enable a pointer to an encapsulated object (such as a ShapeGen object) to be passed as an argument to a function (such as MySub in the MyTest2 code example), the SmartPtr class overloads the ***** operator.

All the code examples in the remainder of this user's guide will use the SmartPtr class template.

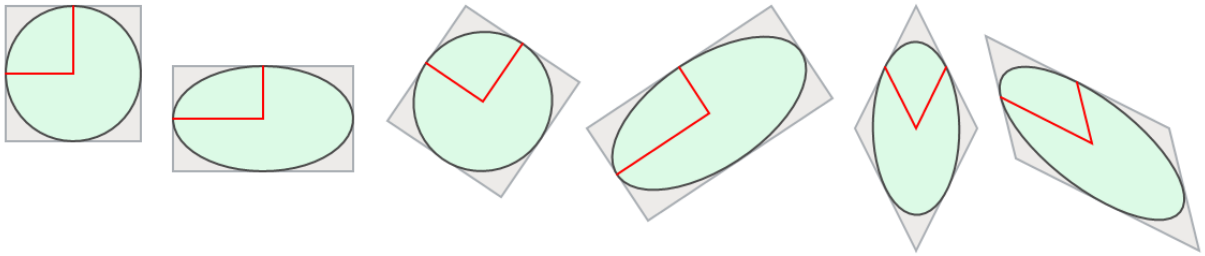
Ellipses and elliptic arcs

The paper titled *A Fast Parametric Ellipse Algorithm* on the `arXiv.org` website describes a fast algorithm for generating ellipses and elliptic arcs (and also, of course, circles and circular arcs).

The ShapeGen library uses this algorithm to construct ellipses and elliptic arcs of any shape and orientation. An ellipse is defined by three points: its center point plus the end points of two *conjugate diameters* of the ellipse. Other 2-D graphics libraries typically do not use conjugate diameters to describe their ellipses, so this brief explanation might be helpful:

> The conjugate diameter end points are simply the midpoints of two adjacent sides of the square, rectangle, or parallelogram in which the ellipse is inscribed.

The following screenshot should clarify things a bit.

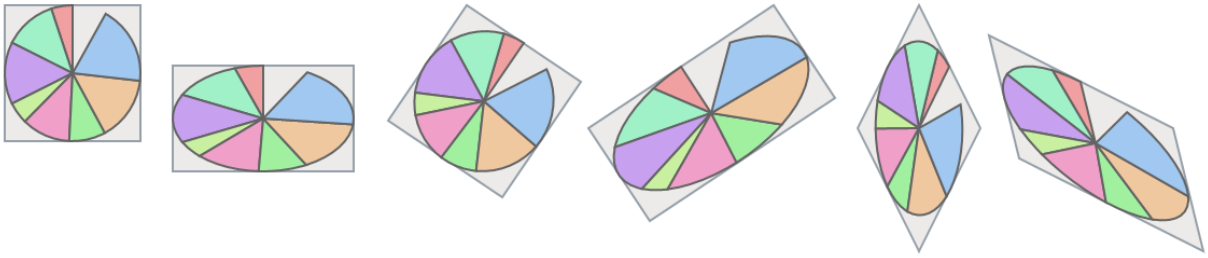


At the left edge of the screenshot, a circle is inscribed in a square. A circle is a special case of an ellipse. Red lines are drawn from the center of this particular ellipse to the end points of two conjugate diameters of the ellipse. (For the special case of a circle, any two perpendicular diameters are conjugate diameters.)

The other figures in the screenshot above are affine transformations of the initial figure at the left. In each case, the end points of the two conjugate diameters coincide with the midpoints of two adjacent sides of an enclosing square, rectangle, or parallelogram[2].

The preceding screenshot was rendered by a program that calls the `ShapeGen::Ellipse` function.

The following screenshot was rendered by a similar program that calls the `ShapeGen::EllipticArc` function to draw six different views of the same pie chart. The two screenshots share the same set of enclosing squares, rectangles, and parallelograms.



[2] You don't actually have to draw a square, rectangle, or parallelogram around your ellipse. That's done here just to help you visualize the underlying geometric principle.

Arcs plotted by the `EllipticArc` function share an important property with Bézier curves: both are *affine-invariant*. For a Bézier curve, applying any affine transformation to the vertices of the control polygon produces the same transformed image as does individually transforming the points on the curve. Similarly, for an arc constructed by the `EllipticArc` function, application of any affine transformation to the ellipse center point and the two conjugate diameter end points has the same effect as individually transforming the points on the arc.

Of course, the `Ellipse` function is similarly affine-invariant.

For more information, see the Wikipedia article on conjugate diameters, or see the paper titled *A rotated ellipse from three points* at the ResearchGate website.

In case you're curious, here's the function that created the pie chart screenshot:

```
void PieToss(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    float percent[] = {
        5.1, 12.5, 14.8, 5.2, 11.6, 8.7, 15.3, 18.7
    };
    COLOR color[] =
    {
        RGBX(240,160,160), RGBX(160,240,200), RGBX(200,160,240), RGBX(200,240,160),
        RGBX(240,160,200), RGBX(160,240,160), RGBX(240,200,160), RGBX(160,200,240),
    };
    // Define three corner points of each square, rectangle,
    // or parallelogram. We'll calculate the fourth point.
    SGPoint xy[][4] = {
        { {  155,   29 }, {   29,   29 }, {   29,  155 }, },
        { {  353,   85 }, {  185,   85 }, {  185,  183 }, },
        { {  564,  100 }, {  458,   29 }, {  385,  136 }, },
        { {  743,   29 }, {  571,  143 }, {  628,  229 }, },
        { {  935,  143 }, {  878,   29 }, {  821,  143 }, },
        { { 1114,  143 }, {  943,   57 }, {  971,  171 }, },
    };

    sg->SetLineJoin(LINEJOIN_MITER);
    for (int i = 0; i < ARRAY_LEN(xy); ++i)
    {
        SGPoint v0, v1, v2;
        float astart = 0;

        // Use symmetry to calculate the fourth point of the
        // square, rectangle, or parallelogram. Then draw it.
        xy[i][3].x = xy[i][0].x - xy[i][1].x + xy[i][2].x;
        xy[i][3].y = xy[i][0].y - xy[i][1].y + xy[i][2].y;
        sg->BeginPath();
        sg->Move(xy[i][0].x, xy[i][0].y);
        sg->PolyLine(&xy[i][1], 3);
        sg->CloseFigure();
        aarend->SetColor(RGBX(237,235,233));
        sg->FillPath();
        aarend->SetColor(RGBX(150,160,170));
        sg->SetLineWidth(2.0);
        sg->StrokePath();

        // The center point v0 of the ellipse is simply the center
        // of the enclosing square, rectangle, or parallelogram
```

```
            v0.x = (xy[i][0].x + xy[i][2].x)/2;
            v0.y = (xy[i][0].y + xy[i][2].y)/2;

            // The conjugate diameter end points are simply the
            // midpoints of two adjacent sides of the enclosing
            // square, rectangle, or parallelogram
            v1.x = (xy[i][0].x + xy[i][1].x)/2;
            v1.y = (xy[i][0].y + xy[i][1].y)/2;
            v2.x = (xy[i][1].x + xy[i][2].x)/2;
            v2.y = (xy[i][1].y + xy[i][2].y)/2;

            // Draw the pie chart inside the square, rectangle, or
            // parallelogram
            for (int j = 0; j < 8; ++j)
            {
                float asweep = 2.0*PI*percent[j]/100.0;  // PI = 3.14159...

                sg->BeginPath();
                sg->EllipticArc(v0, v1, v2, astart, asweep);
                sg->Line(v0.x, v0.y);
                sg->CloseFigure();
                aarend->SetColor(color[j]);
                sg->FillPath();
                aarend->SetColor(RGBX(100,100,100));
                sg->StrokePath();
                astart += asweep;
            }
        }
    }
```

Renderers included in this project

A ShapeGen object generates shapes that can be rendered on a graphics display. This object is paired with a renderer that draws the shapes to the back buffer. The contents of the back buffer can be copied to on-screen memory and shown on the graphics display.

This GitHub project includes the source code for example renderers that are platform-independent and that support true-color graphics displays. The ShapeGen demo program and SVG file viewer use the example renderers, plus a relatively small amount of additional, platform-dependent code, to run in Linux and Windows.

The two example renderers are:

- A *simple renderer* that does solid-color fills with no antialiasing
- An *enhanced renderer* that does antialiasing and alpha blending

These renderers are implemented in the `renderer.cpp` file in the main directory of this project. Both renderers can fill shapes with solid colors. In addition to antialiasing and alpha blending, the enhanced renderer fills shapes with tiled patterns, linear gradients, and radial gradients.

The enhanced renderer works exclusively with true-color graphics displays.

The simple renderer is faster for filling large regions with solid colors.

To construct an image, the two example renderers write directly to a shared window-backing memory buffer, or *back buffer*. This buffer uses a 32-bit BGRA pixel format (that is `0xaarrggbb`). During construction of the

image, a ShapeGen-based application can switch between from one renderer to the other by calling the `ShapeGen::SetRenderer` function. After the image has been rendered, a BitBlt operation copies the back buffer to a window or viewport in the on-screen memory.

The simple renderer is easily modified to run on small computers that have limited graphics capabilities and cannot display true-color images. Just a few lines of code are required to implement a standalone simple renderer, which makes porting the ShapeGen library relatively simple. See Appendix A in this user's guide for code examples of simple renderers that run on the Win32/GDI and SDL2 APIs.

The `EnhancedRenderer` class implementation in this GitHub project can be ported with minimal effort to a computer system with a true-color graphics display. The interface for this renderer implements a number of application-callable functions, which are described in the EnhancedRenderer functions section.

ShapeGen-renderer interface

Instead of passing a shape description directly to a renderer, a ShapeGen object embeds the shape information in a `ShapeFeeder` object. ShapeGen then passes this object to the renderer's `RenderShape` function, which performs the actual rendering. Each `RenderShape` call fills or strokes a shape that was previously specified by a user-constructed path. During this call, the shape feeder acts as an iterator that supplies the renderer with pieces of the shape in line-by-line fashion, from top to bottom.

The shape feeder adapts its behavior to the type of renderer – simple or enhanced – that it's connected to.

For example, to support a *simple* renderer, the shape feeder breaks a shape into a series of rectangles with vertical and horizontal sides. Rectangle coordinates are specified as simple integers. For shapes with sharply curved or slanted sides, most of these rectangles might be just one or two pixels in height. However, a large, rectangular fill region can be specified as a single rectangle.

To support an *enhanced* renderer, the shape feeder breaks a shape into a series of *subpixel spans*[3], which the renderer uses to construct a coverage bitmask for each pixel in the shape. Each of these spans begins and ends on a *subpixel boundary*[4]. The *x* and *y* coordinates of a subpixel are expressed as fixed-point numbers. The integer part of an *x* or *y* coordinate identifies a pixel, and the fractional part identifies the subpixel column or row within the pixel.

For example, the example enhanced renderer in this project uses a 32-bit coverage bitmask (4 rows, and 8 bits per row) for alpha blending and antialiasing. For this renderer, the two fractional bits of a span's *y* coordinate, as supplied by the shape feeder, specify the subpixel row. An *x* coordinate supplied by the shape feeder is always in 16.16 fixed-point format, and the example renderer uses the three most-significant bits of the 16-bit fraction to determine the starting subpixel column.

To enable an enhanced renderer to conserve memory, the shape feeder always supplies spans in *y*-ascending order so that the portion of a shape that falls on a scan line can always be rendered in its entirety before construction of the next line begins.

---

[3] Just as the term *span*, by itself, describes a horizontal row of pixels that spans the distance between the left and right edges of a filled region in a shape, a *subpixel span* is a row of *sub*pixels that spans the region.
[4] Analogous to a *pixel* boundary, which is located halfway between horizontally and vertically adjacent *pixel* centers, a *subpixel* boundary is located halfway between *subpixel* centers. For more information, see the Scan conversion section.

The simplicity of a *simple* renderer makes it easy to port to any processor for which a C++ compiler is available. A simple renderer might be all that's needed to support a graphics display device that has limited color capabilities. The ShapeGen object calls this renderer's RenderShape function to do all of the drawing. For example, a simple renderer that runs on SDL2 can implement the RenderShape function as follows:

```
void BasicRenderer::RenderShape(ShapeFeeder *feeder)
{
    SDL_Rect rect;

    while (feeder->GetNextSDLRect(reinterpret_cast<SGRect*>(&rect)))
        SDL_FillRect(_surface, &rect, _pixel);
}
```

The BasicRenderer class in this code example is derived from the SimpleRenderer class, which defines a simple renderer. Both simple and enhanced renderers derive from the same base class, Renderer, which will be described shortly.

This implementation of the RenderShape function in the preceding example uses the SDL_FillRect function to fill the rectangles and thus runs on platforms, such as Linux and Windows, for which the SDL2 API is available. An implementation that runs on the Win32/GDI API can call the FillRect function instead. For more information, see the simple renderer code examples in Appendix A in this user's guide.

The RenderShape function implemented by an *enhanced* renderer is necessarily more complex than this – it requires the exchange of additional information between the renderer and ShapeGen object. To support antialiasing and alpha blending, the Renderer interface definition includes QueryYResolution and SetMaxWidth functions. A fourth function, SetScrollPosition, supports scrolling of shapes painted with patterns and gradients.

The following Renderer base class definition is taken from the shapegen.h header file:

```
class Renderer
{
public:
    virtual void RenderShape(ShapeFeeder *feeder) = 0;
    virtual int QueryYResolution() { return 0; }
    virtual bool SetMaxWidth(int width) { return true; }
    virtual bool SetScrollPosition(int x, int y) { return true; }
};
```

The four functions in the Renderer class are called exclusively by the ShapeGen object. The last three of these functions are provided specifically to support enhanced renderers. Note that a simple renderer simply uses the versions of these three functions that are implemented just above in the Renderer class definition.

When a new renderer is installed (for example, if the user calls the ShapeGen::SetRenderer function), ShapeGen immediately calls the renderer's QueryYResolution, SetMaxWidth, and SetScrollPosition functions. Additionally, ShapeGen calls SetMaxWidth to notify the renderer when the application program calls the ShapeGen::InitClipRegion function to change the width and height of the device clipping rectangle. ShapeGen calls the renderer's SetScrollPosition function when the application calls the ShapeGen::SetScrollPosition function to change the *x-y* scrolling coordinates. This call enables patterns and gradients to scroll in unison with the shapes they fill.

An enhanced renderer implements a `QueryYResolution` function that returns the number of fractional (subpixel) bits the renderer requires in the fixed-point *y* coordinates for the spans it receives from the `ShapeFeeder`. For example, the example enhanced renderer in `renderer.cpp` requires two fractional bits in its *y* coordinates. (The *x* coordinates supplied by the shape feeder are *always* in 16.16 fixed-point format.)

An enhanced renderer also implements a `SetMaxWidth` function that receives, as an input parameter, the maximum width (in pixels) of any shape it will be asked to draw.

In addition to the four functions in the `Renderer` class above, a renderer is expected to provide functions that can be called by ShapeGen-based applications. Applications call these functions to specify, for example, the solid color, tiled pattern, or color gradient to use to render shapes. Either of the two example renderers in this GitHub project can do solid-color fills and must therefore provide an application-callable `SetColor` function. The example enhanced renderer implements a number of additional application-callable functions, and these are described in detail in the EnhancedRenderer functions section of this user's guide.

Enhanced renderer pixel format

The ShapeGen class generates the geometric outlines of shapes and maps the shapes onto a two-dimensional array of pixels. ShapeGen delegates all processing of pixel color and transparency data to the renderer.

The `EnhancedRenderer` class implementation in this GitHub project deals exclusively with 32-bit pixels that contain 8-bit red, green, blue, and alpha components. `EnhancedRenderer` member functions assume that, by default, ShapeGen-based application programs specify pixel values in RGBA32 format. On the other hand, this renderer draws to pixel buffers that are in BGRA32 format. Translation between the two formats requires swapping the red and blue components.

The terms RGBA32 and BGRA32 mean different things to big-endians and little-endians, so the following figure is necessary to avoid confusion:

Little-endian naming conventions are used here because most readers will compile and run the ShapeGen demo programs on a PC, and the processors in most PCs have x86 or x64 instruction set architectures, which are little-endian. Most ARM processors also run in little-endian mode.

The motivation for using the RGBA32 format for the application programming interface and the BGRA32 format for rendering is to conform to de facto standards that were established early in the history of Microsoft Windows. For example, the following `RGB` macro from the `wingdi.h` header file in Windows allows application programs to construct a 32-bit `COLORREF` (`unsigned int`) value from 8-bit `r`, `g`, and `b` components:

```
#define RGB(r,g,b)  ((COLORREF)(((BYTE)(r)|((WORD)((BYTE)(g))<<8))|(((DWORD)(BYTE)(b))<<16)))
```

Note that this macro places the `r`, `g`, and `b` values in the same byte positions as does the RGBA32 pixel format shown in the preceding figure.

In contrast, the Windows BMP file format, which was borrowed from IBM's OS/2, uses the following RGBQUAD structure (defined in `wingdi.h`) to specify the ordering of color components in the 32-bit pixels that comprise a true-color bitmapped image[5]:

```
typedef struct tagRGBQUAD {
    BYTE    rgbBlue;
    BYTE    rgbGreen;
    BYTE    rgbRed;
    BYTE    rgbReserved;
} RGBQUAD;
```

On a little-endian machine, this structure places the r, g, and b values in the same byte positions as the BGRA32 format shown in the preceding figure.

The C++ compiler for a big-endian machine will, of course, reverse the order of the RGBQUAD members in a 32-bit pixel. The Windows RGB macro has the virtue of being consistent for both big- and little-endian machines.

In the ShapeGen library, the `renderer.h` header file defines the following two macros, which are similar to the Windows RGB macro:

```
#define RGBX(r,g,b)    (COLOR)(((r)&255)|(((g)&255)<<8)|(((b)&255)<<16)|0xff000000)
#define RGBA(r,g,b,a)  (COLOR)(((r)&255)|(((g)&255)<<8)|(((b)&255)<<16)|((a)<<24))
```

These macros build an RGBA32 pixel from three or four 8-bit components. The RGBX macro sets the color components to the caller-specified r, g, and b values, and it sets the alpha (a) component to 255, which makes the pixel fully opaque.

The RGBA macro is similar to RGBX but allows the caller to set the alpha component to a custom value between 0 and 255. For either macro, the alpha component controls the opacity of the pixel. Setting the alpha component to zero makes the pixel completely transparent.

The COLOR type in the RGBX and RGBA macros is an `unsigned int` that the enhanced renderer uses to represent a 32-bit pixel value. By default, the `EnhancedRenderer` interface assumes that a user-supplied COLOR variable is in RGBA32 pixel format. However, the `EnhancedRenderer::SetPattern` function allows the caller to set the `FLAG_IMAGE_BGRA32` flag to indicate that the pixels in the pattern image are in BGRA32 format rather than RGBA32 format.

By default, the enhanced renderer assumes that a COLOR variable supplied by a ShapeGen-based application program is in straight (non-premultiplied) alpha format. However, the `EnhancedRenderer::SetPattern` function allows the caller to set the `FLAG_PREMULTALPHA` flag to indicate that the pattern pixels are in premultiplied-alpha format.

Pixel buffers managed by the enhanced renderer are always in premultiplied-alpha format.

[5] The RGBQUAD structure was initially used to describe 32-bit color palette entries. It was later used to specify the default pixel format for "uncompressed RGB" (true-color images). Later yet, the structure was extended (sort of) to support alpha blending, as described in this MSDN article: Alpha Blending (Windows GDI) - Win32 apps.

Compositing and filtering with layers

ShapeGen application programs typically draw to a back buffer, whose contents are then copied to the on-screen memory and displayed. By making use of the enhanced renderer's alpha-blending capabilities, users can render arbitrarily complex images by compositing layers of graphics elements directly into the back buffer.

However, if a complex image is to be used multiple times, the image can be rendered to a separate pixel buffer and preserved; doing so avoids discarding and then rendering the image again each time it is needed.

A separate pixel buffer can also be used as a scratch buffer in which to construct a series of partial images that are then sequentially composited to the back buffer to form a larger, more complex image. After a partial image has been composited, the scratch buffer is cleared so that the next partial image can be constructed.

This separate pixel buffer is frequently referred to as a transparency layer, or simply *layer*. The pixel buffer for a layer typically has the same 32-bit pixel format as the back buffer. Depending on the application, a layer buffer may or may not have the same dimensions as the back buffer.

To use a pixel buffer as a layer, the buffer is initially filled with a background color of *transparent black*. In a transparent black pixel, the 8-bit red, green, blue, and alpha components are all set to zero. After an image has been rendered over this background, any pixels in the layer buffer that have not been painted remain transparent. Thus, when the completed image in the layer is composited into the back buffer, any part of the back buffer's original background that is covered by a transparent part of the layer remains fully visible.

Additionally, if a shape or image is to be filtered before it is composited to the back buffer, the image must first be rendered to a transparent layer, where the pixels in the image can be filtered in isolation from the background pixels in the back buffer.

An `EnhancedRenderer` object can work equally well with a back buffer or a layer buffer. The renderer doesn't even need to know whether it's drawing to a layer buffer or a back buffer. To use the renderer with a layer, it is only necessary to fill the layer buffer with transparent black before rendering the image[6]. That's in contrast to the back buffer, whose initial background is typically set to be fully opaque.

A ShapeGen app creates separate `EnhancedRenderer` objects to work with the back buffer and layer buffer. Each of these renderer objects is then managed by a dedicated ShapeGen object. Assigning separate renderer and ShapeGen objects to the back buffer and layer buffer avoids having to switch graphics contexts each time an application program's focus changes from the back buffer to the layer buffer, and vice versa.

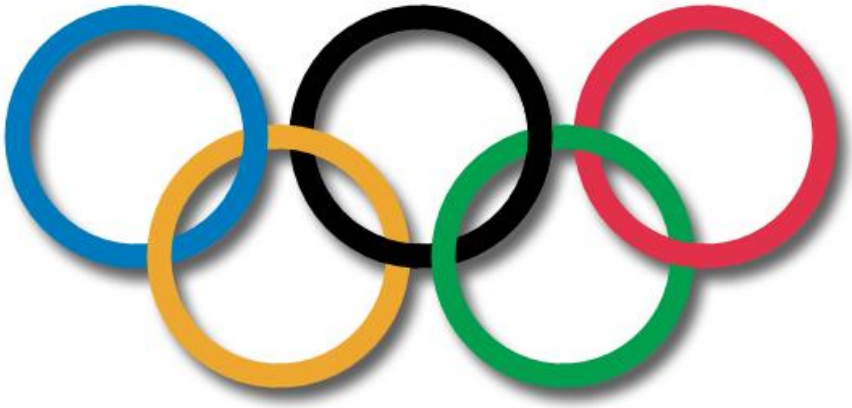
In the following code example, the `DropShadow` function creates a layer buffer, into which the function renders a composite image consisting of five interlinked rings of different colors. Next, the alpha components from this image are Gaussian-blurred to form a drop shadow of the image. Finally, the original image and drop shadow are composited together into the back buffer. A detailed description follows the code example.

```
void DropShadow(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    // 1. Dedicate a renderer and ShapeGen object to the back buffer
    //
```

[6] To clear a pixel buffer or a region within a pixel buffer by setting it to transparent black, set the blend operation to BLENDOP_ALPHA_BLEND and then fill the region with a fully opaque color. For more information, see `EnhancedRenderer::SetBlendOperation`.

```
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));

    // 2. Create dedicated renderer and ShapeGen objects to manage a
    //    layer buffer, which is initialized to transparent black
    //
    SGRect clip2 = { 0, 0, 640, 480 };  // device clipping rect
    PIXEL_BUFFER layerbuf;
    layerbuf.pixels = 0;  // renderer will allocate pixel memory
    layerbuf.width = clip2.w;
    layerbuf.height = clip2.h;
    layerbuf.depth = 32;
    SmartPtr<EnhancedRenderer> aarend2(CreateEnhancedRenderer(&layerbuf));
    SmartPtr<ShapeGen> sg2(CreateShapeGen(&(*aarend2), clip2));

    // 3. Render OlympicRings image into layer, and get image's bbox
    //
    SGRect bbox;
    OlympicRings(&(*sg2), &(*aarend2), &bbox);

    // 4. Get subregion of layer that contains original, unblurred image
    //
    PIXEL_BUFFER subbuf;
    bool status = aarend2->GetPixelBuffer(&layerbuf);
    assert(status);
    DefineSubregion(subbuf, layerbuf, bbox);

    // 5. Apply Gaussian blur to alpha values in OlympicRings image
    //
    float stddev = 4.2f;
    AlphaBlur ablur(stddev);
    ablur.SetColor(RGBA(0,0,0,160));
    status = ablur.BlurImage(subbuf);
    assert(status);

    // 6. Get the expanded bounding box for the blurred image
    //
    SGRect blurbbox;
    ablur.GetBlurredBoundingBox(blurbbox, bbox);

    // 7. Use blurred image as pattern for rendering to back buffer
    //
    blurbbox.x += 6, blurbbox.y += 8;
    aarend->SetPattern(&ablur, blurbbox.x, blurbbox.y,
                       blurbbox.w, blurbbox.h, 0);
    sg->BeginPath();
    sg->Rectangle(blurbbox);
    sg->FillPath();

    // 8. Use the original, unblurred image as the next fill pattern
    //
    aarend->SetPattern(subbuf.pixels, bbox.x, bbox.y,
                       bbox.w, bbox.h, subbuf.pitch/sizeof(COLOR),
                       FLAG_IMAGE_BGRA32 | FLAG_PREMULTALPHA);
    sg->BeginPath();
    sg->Rectangle(bbox);
    sg->FillPath();
}
```

The DropShadow function is taken from the demo.cpp source file in this project. The output produced by this code example is shown in the following screenshot.



The step-by-step description of this code example is as follows:

1. The `DropShadow` function starts by creating an `EnhancedRenderer` object and a ShapeGen object. The two objects can then be accessed through the `sg` and `aarend` pointers, as was done in the Creating a ShapeGen object topic. These objects are dedicated for use with a back buffer that was allocated by the caller. Assume that this buffer has a width of 1280 pixels, a height of 960 pixels, and was initially filled with an opaque background color.
2. Next, the example code creates a second set of `EnhancedRenderer` and ShapeGen objects, which will be dedicated to managing a layer buffer. These objects are accessed through the `aarend2` and `sg2` pointers. In the call to the `CreateEnhancedRenderer` function, the `layerbuf` argument provides a null pointer value for the `pixels` member, which prompts the renderer to allocate a block of memory for the pixels and fill it with zeroes (transparent black). Because the renderer allocated this memory, the renderer's destructor is responsible for deleting it.
3. The example code calls the `OlympicRings` function (in the `demo.cpp` source file in this project) to render an image to the layer buffer. This image is composed of five stroked shapes (interlocked circles) of different colors. The `sg2` and `aarend2` object pointers are passed as arguments in the `OlympicRings` function call. After rendering the image, this function retrieves the bounding box, `bbox`, for the image.
4. The `DefineSubregion` function (in the `renderer.cpp` source file) generates a concise description of the region of the layer buffer that corresponds to the bounding box for the `OlympicRings` image. `DefineSubregion` writes this description to a `PIXEL_BUFFER` structure, which is described in the Creating a ShapeGen object topic.
5. To generate a Gaussian-blurred drop shadow for the OlympicRings image, an instance of the `AlphaBlur` class (defined in `demo.h`; implemented in `alfablur.cpp`) is created. The `stddev` argument passed to the `AlphaBlur` constructor is the standard deviation to use for the Gaussian filter.

   The `AlphaBlur::BlurImage` function call takes the OlympicRings image as input, generates a blurred image (the drop shadow) from the alpha components in the input image, and allocates memory in which to store the blurred image. In this filtering process, which is sometimes referred to as *alpha blurring*, the color components in the input image are discarded and replaced by a fill color in the output image. The preceding call to the `AlphaBlur::SetColor` function specifies the fill color as partially opaque black.

To be suitable for alpha blurring, the source image must reside in a layer buffer so that any surrounding pixels that are not part of the image are transparent. If the source image were instead taken from the back buffer, which contains only fully opaque pixels, the alpha-blurred version of the image would a very boring solid black rectangle with fuzzy edges.

6. The example code calls the `AlphaBlur::GetBlurredBoundingBox` function to retrieve the bounding box for the blurred image.

   The bounding box for a Gaussian-blurred image is an expanded version of the bounding box for the original image. If `r` is the radius of the Gaussian filter kernel, the blurred image's bounding box is calculated by extending the four sides of the original image's bounding box outward by `r = floor(w/2)` pixels, where `w` is the filter kernel width. The `AlphaBlur` constructor allows the caller to specify a custom kernel width (always an odd integer) but this example code simply relies on the constructor to calculate a default width.

7. From here on, the example code will be drawing to the back buffer and will use the `sg` and `aarend` pointers instead of `sg2` and `aarend2`.

   To achieve a shadow effect, the blurred image is composited into the back buffer first, after which (in step 8) the original image will be composited over the blurred image. A small offset is added to the bounding box for the blurred image to push it a few pixels downward and to the right relative to the original image.

   To alpha-blend the blurred image into the back buffer, the example code sets the blurred image to be the current fill pattern and then uses the pattern to fill the rectangle described by the bounding box for the blurred image. In the initial call to the `EnhancedRenderer::SetPattern` function, the first argument is a pointer to the `AlphaBlur` object, which implements an `ImageFeeder` interface, as described in the `EnhancedRenderer::SetPattern` reference topic. The `SetPattern` function uses this interface to load the blurred image to serve as the next pattern.

   At first glance, using the blurred image as a fill pattern might not seem to be the most straightforward way to move the image into the back buffer. But fill patterns have these advantages:

   - They are automatically clipped
   - They use the current constant alpha for blending
   - They use the transformation set by the `EnhancedRenderer::SetTransform` function
   - They can fill shapes other than rectangles
   - They can be automatically converted to premultiplied-alpha format, if needed

8. To blend the original, unblurred image into the back buffer, the example code again calls the `SetPattern` function to designate the original image as the fill pattern and then fills the rectangle described by this image's bounding box. In this version of the overloaded `SetPattern` function, the first argument is a two-dimensional pixel array, as described in the `EnhancedRenderer::SetPattern` topic.

For more information, the following websites have helpful descriptions of layers:

- Apple.com – Transparency Layers and Core Graphics Layer Drawing in *Quartz 2D Programming Guide*
- Wikipedia.org — https://en.wikipedia.org/wiki/Layers_(digital_image_editing)
- Lifewire.com — https://www.lifewire.com/what-is-a-layer-140867

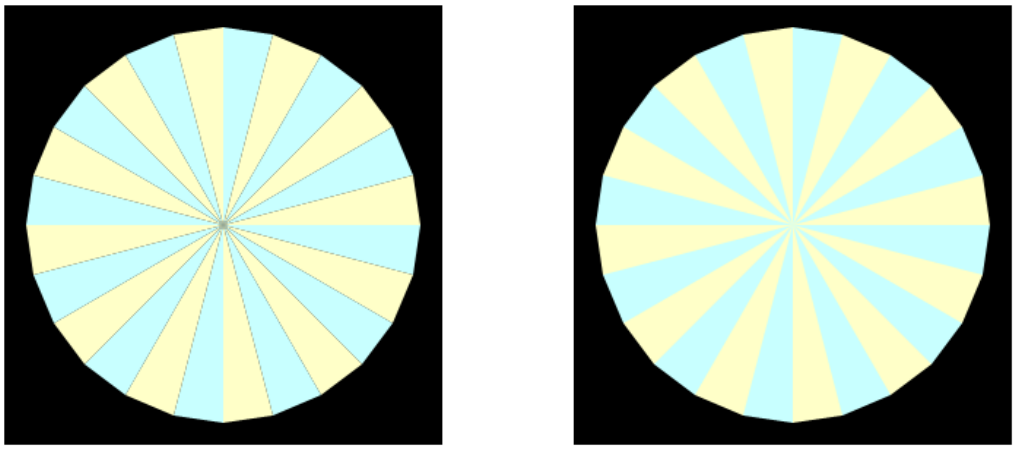
The background-color-leak problem

The `EnhancedRenderer::SetBlendOperation` function allows the user to change the blending operation that the enhanced renderer uses to combine source pixels with destination pixels. During rendering, the *source* pixels belong to the shape that is being rendered. The *destination* pixels belong to the background image in the target pixel buffer in which the shape is being rendered.

When the `EnhancedRenderer` object is created, the object's blend-operation attribute is set to its default value, `BLENDOP_SRC_OVER_DST`. This setting configures the renderer to use the Porter & Duff *A*-over-*B* blending operation, where *A* is the source pixel and *B* the destination. For more information, see the `EnhancedRenderer::SetBlendOperation` reference topic.

For many applications, *A*-over-*B* blending is sufficient. It supports antialiasing and transparency. Some users might never feel the need to switch to another blending operation.

However, *A*-over-*B* blending does have drawbacks for some applications. A particularly annoying problem arises when *A*-over-*B* blending is used to render adjoining shapes that share a common edge, as shown on the left in the following screenshot.

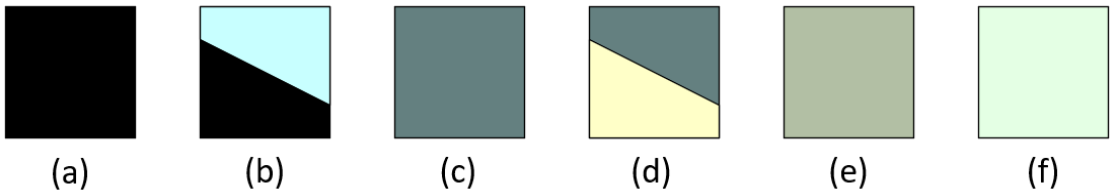


On either side of this screenshot, a circle is formed by 24 light-colored triangles that are rendered over a black background. Look closely at the circle on the left (drawn with *A*-over-*B* blending) and you'll notice that the dark background is leaking through the seams between the triangles. These leaks occur despite the fact that (in terms of the geometry) there should be no gaps between adjacent triangles. Note that the vertex coordinates at the ends of each shared edge are identical for the triangle on either side of the edge.

In contrast, the circle on the right side of the preceding screenshot is drawn using the add-with-saturation blending operation (BLENDOP_ADD_WITH_SAT), as explained in the next topic, Front-to-back rendering, and avoids the color-leak problem.

What causes the background color to leak through the seams? Kilgard and Bolz[7] call these color leaks "conflation artifacts" and point out that the *A*-over-*B* blending operation is to blame. The following figure shows how a color leak develops *within a single pixel* that lies along the edge that separates two of the triangles on the left side of the previous screenshot.

[7] Mark Kilgard, Jeff Bolz (2012). "GPU-accelerated path rendering." *ACM Trans. on Graphics* (TOG) **31**(6), https://developer.download.nvidia.com/devzone/devcenter/gamegraphics/files/opengl/gpupathrender.pdf



In part (a) of this figure, the pixel is filled with the background color, black. In part (b), the edge of a light blue triangle passes through the center of the pixel, and the renderer is informed (through the coverage bitmask, as described in the ShapeGen-renderer interface topic) that this triangle covers half of the pixel. In response, the renderer uses *A*-over-*B* blending in (c) to set the pixel's color to a 50-50 blend of light blue and black (and the coverage bitmask is discarded).

In part (d) of the preceding figure, the renderer is informed that a yellow triangle now covers half of the pixel, and in (e) the renderer uses *A*-over-*B* blending to set the pixel to a 50-50 blend of yellow and the current background color. But remember that this background is now a 50-50 blend of light blue and black. Thus, 25 percent of the rendered pixel color in (e) is due to the black background leaking through the seam between the two light-colored triangles, and the pixel is darker than it should be.

Part (f) of the preceding figure shows what the final color *should* be, which is a 50-50 blend of light blue and yellow. This is the color that results if the add-with-saturation operation is used instead of *A*-over-*B* blending.

The next topic, Front-to-back rendering, describes a technique that could solve the background-color-leak problem for many applications. This technique combines the add-with-saturation blending operation with the `ShapeGen::SetMaskRegion` function to perform front-to-back rendering.

Front-to-back rendering

As explained in the `EnhancedRenderer::SetBlendOperation` reference topic, the enhanced renderer supports the *add-with-saturation* and *alpha-clear* blending operations in addition to the Porter & Duff *A*-over-*B* blending operation.

The previous topic, The background-color-leak problem, described how the *A*-over-*B* blending operation can cause "conflation artifacts" along edges that are shared by two adjacent shapes. The add-with-saturation blending operation can solve this problem for many applications, but it requires somewhat different rendering techniques.

First, before using the add-with-saturation blending operation to compose an image, the background should initially be set to transparent black (all zeroes). For this purpose, the alpha-clear operation can be used to clear (set to zero) a region of the target pixel buffer in which to compose the image.

This image should be composed of non-overlapping shapes to prevent arithmetic overflow in the 8-bit alpha and color components of the destination pixels. Note that fragments of adjacent shapes can safely occupy parts of the same pixel as long as the fragments don't overlap. But even when occasional small (subpixel) overlaps occur due to arithmetic precision errors, the *saturation* feature of the add-with-saturation operation effectively conceals the effects of these overlaps. However, large overlaps are likely to produce visible artifacts.

The requirement that shapes should not overlap might seem to be overly restrictive and thus limit the usefulness of the add-with-saturation blend operation. However, this requirement is easily met by using the

ShapeGen::SetMaskRegion function to mask off each shape as soon as it has been rendered. The part of the clipping region that's masked off by SetMaskRegion will exactly match (to the subpixel level) the filled or stroked shape as long as the same path and the same fill or stroke attributes are used for both. The edge formed where one shape is partially covered by another will be nicely antialiased.

Pairing the add-with-saturation blend operation with the SetMaskRegion function in this way provides a method for rendering shapes in front-to-back order. The foremost shape is rendered first, followed by the shapes behind it, which might be partially covered, and the background is rendered last. In contrast, the *A*-over-*B* blending operation is used with the *painter's algorithm*, which is to say that shapes are rendered in back-to-front order.

With the painter's algorithm, the renderer might end up writing to the same pixel several times as shapes that were rendered earlier in the back-to-front ordering are partially or fully covered by later shapes. On the other hand, with front-to-back rendering, clipping eliminates such hidden shapes (or parts of shapes) so that the renderer is never asked to draw them.

The front-to-back rendering method works well for rendering fully opaque shapes. But a significant limitation of this method is that it cannot handle partially transparent shapes that overlap. After a partially transparent shape has been rendered with the front-to-back method, the shape is masked off from the clipping region, thus preventing any shape behind it (which should be partially visible) from being rendered. For this reason, back-to-front rendering with the *A*-over-*B* blending operation remains the method of choice for composing images that contain partially transparent shapes.

An additional limitation of the front-to-back method is that each time SetMaskRegion is called to remove a newly rendered shape from the clipping region, the clipping region becomes more complex. In the extreme case, constructing an intricately detailed image consisting of a large number of small shapes can eventually make the clipping region so complex that the clipping overhead causes significant delays.

To avoid this problem, a number of partial images can be individually constructed and then composited together (using the Porter & Duff *A*-over-*B* operation) to form a single large image, as was explained in the earlier topic, Compositing and filtering with layers. The complexity of any partial image constructed with front-to-back rendering can then be constrained to avoid creating an overly complex clipping region.

The following code example draws the two circles shown in the screenshot from the previous topic, The backgound-leak-problem. The LeakThru function in this example uses back-to-front and front-to-back rendering to draw the circles on the left and right sides, respectively, of the screenshot.

```
void LeakThru(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));

    SGRect rect = { 60, 60, 300, 300 };
    COLOR color[2] = {
        RGBX(200,255,255),  // light blue
        RGBX(255,255,200),  // yellow
    };

    // Fill the square on the left with opaque black
    aarend->SetColor(RGBX(0,0,0));
    sg->BeginPath();
    sg->Rectangle(rect);
    sg->FillPath();
```

```
    // In the square on the left, use the A-over-B blend operation to
    // render a circular pattern of 24 light-colored triangles
    SGCoord x0 = 210, y0 = 210, r = 135, x1 = 0, y1 = -r, offset = 390;
    float angle = 0;
    for (int i = 0; i < 24; ++i)
    {
        angle += PI/12;
        float sina = sin(angle), cosa = cos(angle);
        aarend->SetColor(color[i&1]);
        sg->BeginPath();
        sg->Move(x0,y0);
        sg->Line(x0+x1,y0+y1);
        x1 = r*sina;
        y1 = -r*cosa;
        sg->Line(x0+x1,y0+y1);
        sg->FillPath();
    }

    // Clear the square on the right to transparent black
    aarend->SetBlendOperation(BLENDOP_ALPHA_CLEAR);
    aarend->SetColor(RGBX(0,0,0));
    sg->BeginPath();
    rect.x += offset;
    sg->Rectangle(rect);
    sg->FillPath();

    // In the square on the right, use add-with-saturation blending to
    // render a circular pattern of 24 light-colored triangles
    angle = 0;
    x0 += offset;
    x1 = 0;
    y1 = -r;
    aarend->SetBlendOperation(BLENDOP_ADD_WITH_SAT);
    for (int i = 0; i < 24; ++i)
    {
        angle += PI/12;
        float sina = sin(angle), cosa = cos(angle);
        aarend->SetColor(color[i&1]);
        sg->BeginPath();
        sg->Move(x0,y0);
        sg->Line(x0+x1,y0+y1);
        x1 = r*sina;
        y1 = -r*cosa;
        sg->Line(x0+x1,y0+y1);
        sg->FillPath();
        sg->SetMaskRegion();
    }

    // Set the background of the square on the right to opaque black
    aarend->SetColor(RGBX(0,0,0));
    sg->BeginPath();
    sg->Rectangle(rect);
    sg->FillPath();
}
```

For the left side of the screenshot, this code example uses the *A*-over-*B* operation to render the circle in back-to-front order, with the background drawn first, followed by the 24 triangles that form the circle. In this case, the background color leaks are clearly visible.

On the right side of the same screenshot, the preceding code example uses the add-with-saturate operation to render the image in front-to-back order. As a result, the color leaks have been eliminated. The 24 triangles that form the circle are drawn first, followed by the background color fill. In the `for`-loop that draws these triangles, each call to the `ShapeGen::FillPath` function is followed by a call to the `ShapeGen::SetMaskRegion` function. As explained earlier, the `SetMaskRegion` calls collectively modify the clipping region to prevent the background fill that follows from overwriting the triangles.

ShapeGen types and structures

The following types and structures are used by the functions in the ShapeGen programming interface. These types and structures are defined in the `shapegen.h` header file included in this GitHub project.

The `SGPoint` and `SGRect` structures are essentially identical to the SDL2 structures SDL_Point and SDL_Rect but are renamed here to enhance portability and to avoid naming conflicts in SDL2-based implementations.

SGCoord type

The `SGCoord` type is used to store an *x* or *y* coordinate value.

Syntax

```
C++

typedef int SGCoord;
```

Remarks

By default, ShapeGen functions treat the user's `SGCoord` values as 32-bit integers. However, the user can call the `ShapeGen::SetFixedBits` function to specify that ShapeGen functions are to treat `SGCoord` values as fixed-point numbers instead.

The `SGPoint` and `SGRect` structures contain coordinate values of type `SGCoord`. The interpretation of the *x*-*y* coordinate values in these structures is affected by calls to the `SetFixedBits` function.

For information about the mapping of ShapeGen *x*-*y* coordinates to the pixels on a graphics display, see Scan conversion.

Header

`shapegen.h`

See also

`ShapeGen::SetFixedBits`

`SGPoint`

`SGRect`

SGPoint structure

The SGPoint structure specifies the position of a point in the *x-y* coordinate space used by the ShapeGen path-construction functions.

Syntax

C++

```
struct SGPoint {
   SGCoord x;
   SGCoord y;
};
```

Members

x

The *x* coordinate value.

y

The *y* coordinate value.

Remarks

Parameters x and y specify horizontal and vertical displacements, in pixels, from the origin of the coordinate space used by the ShapeGen path-construction functions. In the ShapeGen coordinate system, *x* values increase to the right, and *y* values increase in the downward direction.

By default, ShapeGen functions treat the user's SGCoord values as 32-bit integers. However, the user can call the ShapeGen::SetFixedBits function to specify that ShapeGen functions are to treat SGCoord values as fixed-point numbers.

For information about the mapping of ShapeGen *x-y* coordinates to a graphics display, see Scan conversion.

Header

shapegen.h

See also

SGCoord

ShapeGen::SetFixedBits

SGRect structure

The SGRect structure specifies a rectangle in terms of its width and height, in pixels, and the *x-y* coordinates at its top-left corner.

Syntax

C++

```
struct SGRect {
   SGCoord x;
   SGCoord y;
   SGCoord w;
   SGCoord h;
};
```

Members

x

The *x* coordinate at the left edge of the rectangle.

y

The *y* coordinate at the top edge of the rectangle.

w

The width, in pixels, of the rectangle.

h

The height, in pixels, of the rectangle.

Remarks

The top and bottom sides of the rectangle are horizontal. The left and right sides of the rectangle are vertical.

Parameters x and y specify horizontal and vertical displacements, in pixels, from the origin of the coordinate space used by the ShapeGen path-construction functions. In the ShapeGen coordinate system, *x* values increase to the right and *y* values increase in the downward direction. Thus, the minimum *x* and *y* coordinates for a rectangle are located at the rectangle's top-left corner.

By default, SGCoord values are 32-bit integers. However, the user can call the ShapeGen::SetFixedBits function to specify that ShapeGen functions are to treat SGCoord values as fixed-point numbers.

For information about the mapping of ShapeGen *x*-*y* coordinates to a graphics display, see Scan conversion.

Header

shapegen.h

See also

SGCoord

ShapeGen::SetFixedBits

ShapeGen functions

The following reference topics describe the functions that comprise the ShapeGen programming interface. This interface is defined in the `shapegen.h` header file included in this GitHub project.

ShapeGen::BeginPath function

The `BeginPath` function begins a new path.

Syntax

C++

```
void ShapeGen::BeginPath();
```

Parameters

None

Return value

None

Remarks

This function discards any existing path, starts a new path, and starts a new, empty figure (or subpath) in this path.

After a new path is created, it persists until another `BeginPath` function call discards the path and creates a new one. A path is *not* destroyed by calls to `ShapeGen::FillPath`, `ShapeGen::StrokePath`, or the ShapeGen clipping functions.

Header

`shapegen.h`

See also

`ShapeGen::FillPath`

`ShapeGen::StrokePath`

ShapeGen::Bezier2 function

The `Bezier2` function constructs a quadratic Bézier spline curve (a parabolic arc), starting at the current point.

Syntax

C++

```
bool ShapeGen::Bezier2(
   const SGPoint& v1,
   const SGPoint& v2
);
```

Parameters

v1

An SGPoint structure that specifies the *x*-*y* coordinates at the Bézier control point for the spline.

v2

An SGPoint structure that specifies the *x*-*y* coordinates at the end point of the spline.

Return value

Returns true if the function succeeds in constructing the spline. If the current point is undefined (because the current figure is empty), the function fails and immediately returns false. Before returning false, the function faults if the NDEBUG macro (used in assert.h) is undefined.

Remarks

The current point is the starting point for the spline. Parameters v1 and v2 specify the control point and end point of the spline.

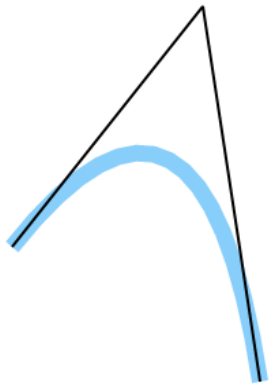
The ShapeGen::PolyBezier2 function constructs a set of connected quadratic Bézier splines in a single function call.

Example

This example uses the Bezier2 function to draw a quadratic Bézier spline (in blue) and its control polygon (in black). (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter clip specifies the device clipping rectangle.)

```
void example01(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint v0 = { 100, 200 }, v1 = { 200, 75 }, v2 = { 230, 270 };

    // Draw quadratic Bezier spline in blue
    aarend->SetColor(RGBX(135,206,250));
    sg->SetLineWidth(12.0);
    sg->BeginPath();
    sg->Move(v0.x, v0.y);
    sg->Bezier2(v1, v2);
    sg->StrokePath();

    // Outline control polygon in black
    aarend->SetColor(RGBX(0,0,0));
    sg->SetLineWidth(2.0);
    sg->BeginPath();
    sg->Move(v0.x, v0.y);
```

```
    sg->Line(v1.x, v1.y);
    sg->Line(v2.x, v2.y);
    sg->StrokePath();
}
```



The output produced by this code example is shown in the screenshot at left.

In the code example above, points v0, v1, and v2 define the control polygon for the spline curve. The starting point, v0, is on the left side of the screenshot, the control point, v1, is at the top, and the end point, v2, is on the bottom right. The curve is tangent to side v0·v1 at the starting point and is tangent to side v1·v2 at the end point.

Header

shapegen.h

See also

SGPoint

ShapeGen::PolyBezier2

ShapeGen::Bezier3 function

The Bezier3 function constructs a cubic Bézier spline curve, starting at the current point.

Syntax

C++

```
bool ShapeGen::Bezier3(
   const SGPoint& v1,
   const SGPoint& v2,
   const SGPoint& v3
);
```

Parameters

v1

An SGPoint structure that specifies the *x*-*y* coordinates at the first Bézier control point for the spline.

v2

An SGPoint structure that specifies the *x*-*y* coordinates at the second Bézier control point for the spline.

v3

An SGPoint structure that specifies the *x*-*y* coordinates at the end point of the spline.

Return value

Returns true if the function succeeds in constructing the spline. If the current point is undefined (because the current figure is empty), the function fails and immediately returns false. Before returning false, the function faults if the NDEBUG macro (used in assert.h) is undefined.

Remarks

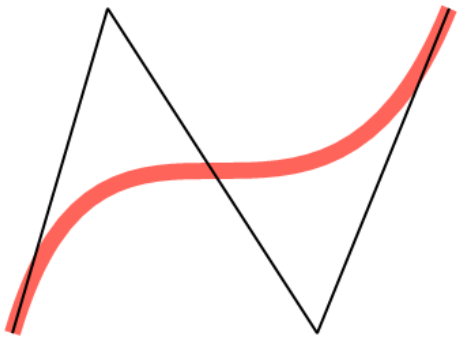
The current point is the starting point for the spline. Parameters v1, v2, and v3 specify the two control points and end point of the spline.

The ShapeGen::PolyBezier3 function constructs a set of connected cubic Bézier splines in a single function call.

Example

This example uses the Bezier3 function to draw a cubic Bézier spline (in orange) and its control polygon (in black). (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter clip specifies the device clipping rectangle.)

```
void example02(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint v0 = { 140, 308 }, v1 = { 210, 70 },
            v2 = { 364, 308 }, v3 = { 461, 70 };

    // Draw cubic Bezier spline in red
    aarend->SetColor(RGBX(255,100,90));
    sg->SetLineWidth(12.0);
    sg->BeginPath();
    sg->Move(v0.x, v0.y);
    sg->Bezier3(v1, v2, v3);
    sg->StrokePath();

    // Outline control polygon in black
    aarend->SetColor(RGBX(0,0,0));
    sg->SetLineWidth(2.0);
    sg->BeginPath();
    sg->Move(v0.x, v0.y);
    sg->Line(v1.x, v1.y);
    sg->Line(v2.x, v2.y);
    sg->Line(v3.x, v3.y);
    sg->StrokePath();
}
```



The output produced by this code example is shown in the screenshot at left.

In the code example above, points v0, v1, v2, and v3 define the control polygon for the spline curve. The starting point, v0, is at the lower-left corner of the screenshot. The end point, v3, is at the top-right corner. In between are the two control points, v1 and v2. The curve is tangent to side v0·v1 at the starting point and is tangent to side v2·v3 at the end point.

Header

shapegen.h

See also

SGPoint

ShapeGen::PolyBezier3

ShapeGen::CloseFigure function

The CloseFigure function closes a figure (or subpath) by adding a line segment connecting the current point to the first point in the figure.

Syntax

C++

```
void ShapeGen::CloseFigure();
```

Parameters

None

Return value

None

Remarks

This function finalizes the current figure and starts a new, empty figure in the same path. After a figure is finalized, it cannot be modified or added to. Any finalized figure not explicitly closed by CloseFigure is open; that is, the first and last points in the figure are not connected.

In constrast to CloseFigure, the ShapeGen::EndFigure function finalizes a figure without closing it – that is, the start and end points are left unconnected.

A CloseFigure call has no effect on a figure that has already been finalized.

CloseFigure affects the appearance of stroked paths drawn by the ShapeGen::StrokePath function but has no effect on the appearance of filled paths. Shapes filled by the ShapeGen::FillPath function are always constructed as though the first and last points in each figure are connected, regardless of any previous calls to CloseFigure. Similarly, clipping regions specified by the ShapeGen::SetClipRegion and ShapeGen::SetMaskRegion functions are always constructed as though the first and last points in each figure are connected.

The StrokePath, FillPath, SetClipRegion, and SetMaskRegion functions finalize the last figure in the path, if it has not already been finalized. When this occurs, the ends of the finalized figure are left open and cannot subsequently be closed.

If CloseFigure is called to finalize a figure that contains a single point, the point is discarded, which leaves the figure empty and ready to receive its first point.

Header

shapegen.h

See also

ShapeGen::EndFigure

ShapeGen::StrokePath

ShapeGen::FillPath

ShapeGen::SetClipRegion

ShapeGen::SetMaskRegion

ShapeGen::Ellipse function

The Ellipse function adds an ellipse to the current path.

Syntax

C++

```
void ShapeGen::Ellipse(
   const SGPoint& v0,
   const SGPoint& v1,
   const SGPoint& v2
);
```

Parameters

v0

An SGPoint structure that specifies the *x*-*y* coordinates at the center of the ellipse.

v1

An SGPoint structure that specifies the *x*-*y* coordinates at an end point of the first of a pair of conjugate diameters of the ellipse.

v2

An SGPoint structure that specifies the *x*-*y* coordinates at an end point of the second of a pair of conjugate diameters of the ellipse.

Return value

None

Remarks

This function can construct an ellipse of arbitrary shape and orientation. The ellipse is defined by its center point and two additional points that lie on the ellipse. The two additional points are the end points of two conjugate diameters of the ellipse.

If the two conjugate diameters are perpendicular and of the same length, the ellipse is a circle. To construct an ellipse in standard position, align the two conjugate diameters to be parallel with the *x* and *y* axes.

An ellipse constructed by the Ellipse function is added to the current path as a complete, closed figure. If, on entry to the Ellipse function, the current figure has not already been finalized, the function finalizes the figure by leaving it open (that is, in the same manner as the ShapeGen::EndFigure function) and then starts a new figure in the same path. After adding the points in the ellipse to the new figure, the Ellipse function finalizes this new figure by closing it (in the same manner as the ShapeGen::CloseFigure function) before starting a newer, empty figure in the same path.

On return from the Ellipse function, the current point is undefined.

For more information about conjugate diameter end points, see Ellipses and Elliptic Arcs.

Example

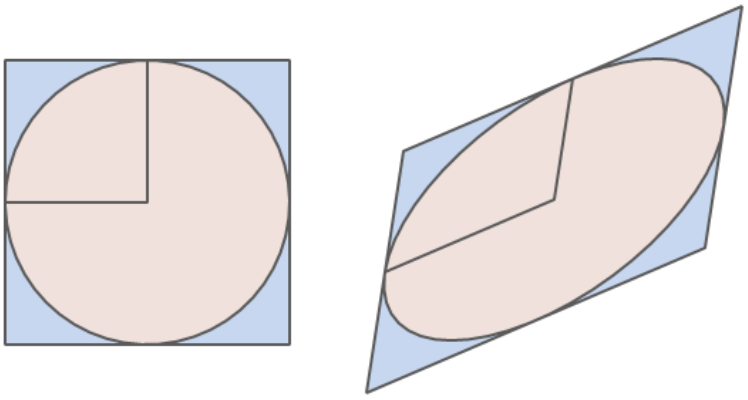
It's straightforward to use the Ellipse function to inscribe an ellipse in a parallelogram so that the ellipse's center coincides with the center of the parallelogram, and the ellipse touches the parallelogram at the midpoint of each of its four sides. If we are given three consecutive corner points, *A*, *B*, and *C*, of the parallelogram, set parameter v0 to the point midway between *A* and *C*, set v1 to the point midway between *A* and *B*, and set v2 to the point midway between *B* and *C*. This technique works for any parallelogram, rhombus, rectangle, or square.

The following example uses the Ellipse function to draw two ellipses inscribed in parallelograms. (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter clip specifies the device clipping rectangle.)

```
void example03(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint xy[2][4] = {
        { { 100, 275 }, { 100,  75 }, { 300,  75 }, },
        { { 354, 309 }, { 380, 139 }, { 618,  37 }, },
    };

    sg->SetLineWidth(2.0);
```

```
    for (int i = 0; i < 2; ++i)
    {
        SGPoint v0, v1, v2;

        // Use symmetry to calculate the fourth vertex of the
        // enclosing square or parallelogram
        xy[i][3].x = xy[i][0].x - xy[i][1].x + xy[i][2].x;
        xy[i][3].y = xy[i][0].y - xy[i][1].y + xy[i][2].y;

        // Calculate ellipse center and two conjugate diameter
        // end points
        v0.x = (xy[i][0].x + xy[i][2].x)/2;
        v0.y = (xy[i][0].y + xy[i][2].y)/2;
        v1.x = (xy[i][0].x + xy[i][1].x)/2;
        v1.y = (xy[i][0].y + xy[i][1].y)/2;
        v2.x = (xy[i][1].x + xy[i][2].x)/2;
        v2.y = (xy[i][1].y + xy[i][2].y)/2;

        // Render parallelogram and inscribed ellipse
        sg->BeginPath();
        sg->Ellipse(v0, v1, v2);
        aarend->SetColor(RGBX(240,225,220));
        sg->FillPath();
        sg->Move(xy[i][0].x, xy[i][0].y);
        sg->PolyLine(&xy[i][1], 3);
        sg->CloseFigure();
        aarend->SetColor(RGBX(200,215,240));
        sg->FillPath();
        sg->Move(v1.x, v1.y);
        sg->Line(v0.x, v0.y);
        sg->Line(v2.x, v2.y);
        aarend->SetColor(RGBX(90,90,90));
        sg->StrokePath();
}
```

The output produced by this code example is shown in the following screenshot.



The figure on the left is a circle (a special kind of ellipse) inscribed in a square (a special kind of parallelogram). The circle touches the square at the midpoint of each side. Lines are drawn from the center of the circle to the end points of a pair of conjugate diameters, which in this case are simply perpendicular radii.

An affine transformation has been applied to the square on the left to produce the parallelogram on the right. The ellipse's center point and conjugate diameter end points have also been transformed. The

resulting ellipse is inscribed in the parallelogram and touches the parallelogram at the midpoint of each of its four sides.

Header

shapegen.h

See also

SGPoint

ShapeGen::EndFigure

ShapeGen::CloseFigure

ShapeGen::EllipticArc function

The EllipticArc function constructs an elliptic arc in the current figure.

Syntax

C++

```
void ShapeGen::EllipticArc(
   const SGPoint& v0,
   const SGPoint& v1,
   const SGPoint& v2,
   float astart,
   float asweep
);
```

Parameters

v0

An SGPoint structure that specifies the *x*-*y* coordinates at the center of the ellipse.

v1

An SGPoint structure that specifies the *x*-*y* coordinates at an end point of the first of a pair of conjugate diameters of the ellipse.

v2

An SGPoint structure that specifies the *x*-*y* coordinates at an end point of the second of a pair of conjugate diameters of the ellipse.

astart

The starting angle, in radians, of the elliptic arc.

asweep

The sweep angle, in radians, of the elliptic arc.

Return value

None

Remarks

Point `v0` is the center of the ellipse, and `v1` and `v2` are the end points of a pair of conjugate diameters of the ellipse. Parameter `astart` is the starting angle of the arc, and parameter `asweep` is the angle traversed by the arc. Both angles are specified in radians of elliptic arc, and both can have positive or negative values.

The starting angle is specified relative to point `v1`, and is positive in the direction of point `v2`. The sweep angle is positive in the same direction as the start angle.

If, on entry to this function, the current point is undefined (because the current figure is empty), the starting point of the arc becomes the first point in the figure. Otherwise, the function inserts a line segment connecting the current point to the starting point of the arc. On return from this function, the current point is set to the end point of the arc.

Arcs plotted by the `EllipticArc` function share an important property with Bézier curves: both are *affine-invariant*. For a Bézier curve, applying any affine transformation to the vertexes of the control polygon produces the same transformed curve as does directly transforming the points on the curve. Similarly, for an arc constructed by the `EllipticArc` function, application of any affine transformation to the ellipse center point `v0` and the two conjugate diameter end points `v1` and `v2` has the same effect as directly transforming the points on the arc. In particular, transformation of the arc does not require modification of the `astart` and `asweep` parameter values supplied to the function (see Example).

For more information about conjugate diameter end points, see Ellipses and Elliptic Arcs.

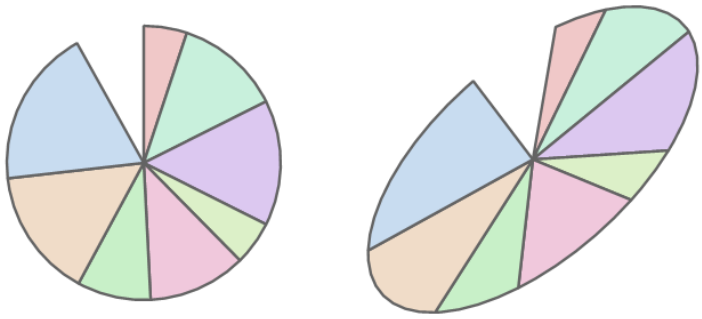
Example

This example uses the `EllipticArc` function to draw two affine-transformed views of the same pie chart. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example04(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    float percent[] = {
        5.1, 12.5, 14.8, 5.2, 11.6, 8.7, 15.3, 18.7
    };
    COLOR color[] = {
        RGBX(240,200,200), RGBX(200,240,220), RGBX(220,200,240), RGBX(220,240,200),
        RGBX(240,200,220), RGBX(200,240,200), RGBX(240,220,200), RGBX(200,220,240),
    };
    SGPoint v[2][4] = {
        { { 240, 210 }, { 240,  90 }, { 360, 210 }, },
        { { 581, 207 }, { 601,  91 }, { 724, 140 }, },
    };

    sg->SetLineWidth(2.4);
    for (int i = 0; i < 2; ++i)
    {
        float astart = 0;
```

```
        for (int j = 0; j < 8; ++j)
        {
            float asweep = 2.0*PI*percent[j]/100.0;

            sg->BeginPath();
            sg->EllipticArc(v[i][0], v[i][1], v[i][2], astart, asweep);
            sg->Line(v[i][0].x, v[i][0].y);
            aarend->SetColor(color[j]);
            sg->CloseFigure();
            sg->FillPath();
            aarend->SetColor(RGBX(100,100,100));
            sg->StrokePath();
            astart += asweep;
        }
}
```

The output produced by this code example is shown in the following screenshot.



The two pie charts in this screenshot differ only in values of the parameters v0, v1, and v2 that are passed to the EllipticArc function. The astart and asweep parameter values that delimit the arcs in the two pie charts are identical. In essence, an affine transformation is applied to the v0, v1, and v2 parameter values for the pie chart on the left to produce the pie chart on the right.

Should the caller need to obtain the *x-y* coordinates at the starting and ending points of each arc in the preceding code example, calls to the ShapeGen::GetFirstPoint and ShapeGen::GetCurrentPoint functions could be inserted immediately after the EllipticArc call.

Header

shapegen.h

See also

SGPoint

ShapeGen::GetFirstPoint

ShapeGen::GetCurrentPoint

ShapeGen::EllipticSpline function

The EllipticSpline function constructs an elliptic spline curve (an elliptic arc spanning π/2 radians), starting at the current point.

Syntax

C++

```
bool ShapeGen::EllipticSpline(
   const SGPoint& v1,
   const SGPoint& v2
);
```

Parameters

v1

An SGPoint structure that specifies the *x*-*y* coordinates at the control point for the spline.

v2

An SGPoint structure that specifies the *x*-*y* coordinates at the end point of the spline.

Return value

Returns `true` if the function succeeds in constructing the spline. If the current point is undefined (because the current figure is empty), the function fails and immediately returns `false`. Before returning `false`, the function faults if the NDEBUG macro (used in `assert.h`) is undefined.

Remarks

The current point is the starting point for the spline. Parameters v1 and v2 specify the control point and end point of the spline. On return from this function, v2 is the new current point.

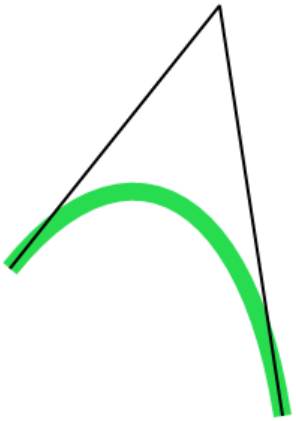
The ShapeGen::PolyEllipticSpline function constructs a set of connected elliptic splines in a single function call.

Example

This example uses the `EllipticSpline` function to draw an elliptic spline curve (in green). The spline's control polygon is outlined in black. (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example05(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint v0 = { 140, 250 }, v1 = { 280, 75 }, v2 = { 322, 348 };

    // Draw elliptic spline in green
    aarend->SetColor(RGBX(40,220,80));
    sg->SetLineWidth(12.0);
    sg->BeginPath();
    sg->Move(v0.x, v0.y);
    sg->EllipticSpline(v1, v2);
    sg->StrokePath();
```

```
    // Outline control polygon in black
    aarend->SetColor(RGBX(0,0,0));
    sg->SetLineWidth(2.0);
    sg->BeginPath();
    sg->Move(v0.x, v0.y);
    sg->Line(v1.x, v1.y);
    sg->Line(v2.x, v2.y);
    sg->StrokePath();
}
```



The output produced by this code example is shown in the screenshot at left.

In the preceding code example, points v0, v1, and v2 define the control polygon for the spline curve. The starting point, v0, is on the left side of the screenshot, the control point, v1, is at the top, and the end point, v2, is on the bottom right. The curve is tangent to side v0·v1 at the starting point and is tangent to side v1·v2 at the end point.

Header

shapegen.h

See also

SGPoint

ShapeGen::PolyEllipticSpline

ShapeGen::EndFigure function

The EndFigure function finalizes a figure (or subpath) by leaving the figure open; that is, the starting and ending points of the figure are left unconnected.

Syntax

C++

```
void ShapeGen::EndFigure();
```

Parameters

None

Return value

None

Remarks

This function finalizes the current figure and starts a new, empty figure in the same path. After a figure is finalized, it cannot be modified or added to. A figure that is finalized by the `EndFigure` function is open and cannot subsequently be closed. If this figure is later stroked by the `ShapeGen::StrokePath` function, no line segment is added to connect the starting and ending points of the figure.

Calls to `EndFigure` have no effect on a figure that has already been finalized.

The `EndFigure` and `ShapeGen::CloseFigure` functions affect the appearance of stroked paths but have no effect on the appearance of filled paths. Shapes filled by the `ShapeGen::FillPath` function are always constructed as though the first and last points in each figure are connected, regardless of any previous calls to `EndFigure` or `CloseFigure`. Similarly, clipping regions specified by the `ShapeGen::SetClipRegion` and `ShapeGen::SetMaskRegion` functions are always constructed as though the first and last points in each figure are connected.

If `EndFigure` is called for a figure that contains a single point, the point is discarded, which leaves the figure empty and ready to receive its first point.

Header

`shapegen.h`

See also

`ShapeGen::StrokePath`

`ShapeGen::CloseFigure`

`ShapeGen::FillPath`

`ShapeGen::SetClipRegion`

`ShapeGen::SetMaskRegion`

ShapeGen::FillPath function

The `FillPath` function fills the area enclosed by the current path according to the *fill rule* that is currently in effect.

Syntax

C++

```
bool ShapeGen::FillPath();
```

Parameters

none

Return value

Returns `true` if the path, after being clipped, is not empty – in this case, the function has sent a description of the clipped path to the renderer to be filled. Otherwise, the function returns `false` to indicate that the clipped path was empty and that nothing has been sent to the renderer.

Remarks

This function fills the current path according to the current fill rule, which is set to either *even-odd* or *nonzero-winding-number*. The `ShapeGen::SetFillRule` function can change the current fill rule to be either `FILLRULE_EVENODD` or `FILLRULE_WINDING`. During the creation of the ShapeGen object, the fill-rule attribute is set to its default value, `FILLRULE_EVENODD`.

The `ShapeGen::EndFigure` and `ShapeGen::CloseFigure` functions affect the appearance of stroked paths but have no effect on the appearance of filled paths. Shapes filled by the `FillPath` function are always constructed as though the first and last points in each figure are connected, regardless of any previous calls to `EndFigure` or `CloseFigure`.

If the final figure in the path has not already been finalized, `FillPath` finalizes this figure in the same manner as the `EndFigure` function. If, for example, a path is to be filled first and then stroked, and the final figure in the path needs to be closed for the `ShapeGen::StrokePath` call, be sure to call `CloseFigure` before calling `FillPath`.

Header

`shapegen.h`

See also

`ShapeGen::SetFillRule`

`ShapeGen::EndFigure`

`ShapeGen::CloseFigure`

`ShapeGen::StrokePath`

ShapeGen::GetBoundingBox function

The `GetBoundingBox` function retrieves the minimum bounding box for the shape defined by the current path.

Syntax

C++

```
int ShapeGen::GetBoundingBox(
   SGRect *bbox,
   int flags = 0
);
```

Parameters

bbox

A pointer to a caller-supplied `SGRect` structure. The function writes the *x*-*y* coordinates, width, and height of the minimum bounding box to this structure. The bbox argument can be null (zero) if the caller simply wants a count of the number of points in the path.

flags

The following flag bits are defined for this function:

- FLAG_BBOX_STROKE – Take the current stroked-line attributes into account
- FLAG_BBOX_CLIP – Clip the bounding box to the device clipping rectangle
- FLAG_BBOX_ACCUM – Accumulate bounding box data from multiple paths

Two or more of these flag bits can be used in combination in the same function call by bitwise-ORing them together.

If the caller does not explicitly supply a `flags` argument, the default value, 0, is automatically assigned to the `flags` parameter.

Return value

Returns a count of the number of points in the current path. If the path is empty, the function immediately returns a value of zero without writing to the structure pointed to by bbox. If the FLAG_BBOX_CLIP flag is set, and the bounding box lies completely outside the device clipping rectangle, the function returns a value of zero without writing to the bbox structure.

Remarks

The bounding box is determined by all the points in the current path. The path can be empty, or can contain one or more points. If the path contains multiple figures (or subpaths), the bounding box takes into account the points in all the figures.

Unless the FLAG_BBOX_STROKE flag bit is set, the `GetBoundingBox` function calculates the bounding box with the assumption that the path is to be filled by the `ShapeGen::FillPath` function.

If the FLAG_BBOX_STROKE flag bit is set, the `GetBoundingBox` function calculates the bounding box with the assumption that the path is to be stroked by the `ShapeGen::StrokePath` function. In this case, the `GetBoundingBox` function takes into account the current set of stroked-path attributes. These attributes are the line width, the line-end-cap style, the line-join style, and the miter limit. `GetBoundingBox` does not actually construct the stroked path. Instead, it first calculates the simple bounding box for the points in the path and then expands the box to encompass the features of the stroked path. For example, if rounded end caps and joins are currently selected, the function extends the simple bounding box outward by half the linewidth on each of its four sides.

The `GetBoundingBox` function's handling of the `FLAG_BBOX_STROKE` flag is conservative and can lead to an oversized bounding box if the current miter limit is excessively long. For example, if the joins in the stroked path are all formed by pairs of lines that meet at obtuse angles, a smaller miter limit would suffice without affecting the appearance of the rendered shape.

If the FLAG_BBOX_CLIP flag bit is set, the GetBoundingBox function clips the bounding box to the device clipping rectangle. If only part of the shape defined by the path lies inside the clipping rectangle, the function still returns the full count of points in the path. But if the shape lies entirely outside the clipping rectangle, the function returns zero and does not write to the bbox structure.

Unless the FLAG_BBOX_ACCUM flag bit is set, the GetBoundingBox function ignores the original contents of the bbox structure and overwrites them with the bounding box data for the current path.

The FLAG_BBOX_ACCUM flag enables a bounding box to be cumulatively assembled from multiple paths. If this flag is set, then instead of simply overwriting the initial contents of the bbox structure, the GetBoundingBox function treats these contents as valid bounding box data accumulated from previous paths. In this case, after constructing the bounding box for the new path, the function merges this bounding box with the existing contents of bbox. The result is a bounding box that encompasses multiple paths.

However, the GetBoundingBox function ignores the FLAG_BBOX_ACCUM flag if the original bbox contents specify a width or height that is less than or equal to zero and is therefore invalid. This behavior provides a convenient way to initialize a bbox structure before making the first in a series of GetBoundingBox function calls that will all use the FLAG_BBOX_ACCUM flag. For example, initialize the bbox width and height to zero.

The GetBoundingBox function does not alter the path in any way.

The bounding-box coordinates retrieved by this function are converted to the user's SGCoord format. By default, SGCoord values are integers, but the user can call the ShapeGen::SetFixedBits function to switch to a fixed-point format.

The four sides of the bounding box generated by the GetBoundingBox function are always adjusted to fall on the boundaries between pixels. These boundaries align with integer *x* and *y* coordinates in the coordinate system used by the ShapeGen drawing functions.

If the path contains a single point, the width and height values calculated for the bounding box can be small but are always greater than zero.

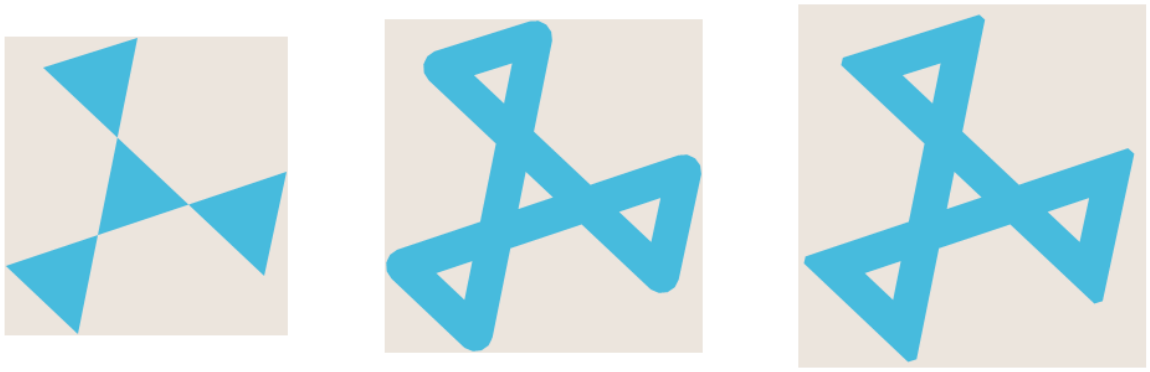
Example

This example uses the GetBoundingBox function to get the minimum bounding boxes for three similar shapes. The first shape is filled, the second is stroked with round joins, and the third is stroked with miter joins. (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter clip specifies the device clipping rectangle.)

```
void example06(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint xy[] = {
        { 130, 97 }, { 308, 265 }, { 326, 181 },
        { 100, 257 }, { 158, 312 }, { 206, 73 }
    };
    SGRect bbox;

    // Fill the shape with solid blue
    aarend->SetColor(RGBX(0,190,255));  // blue
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
```

```
    sg->PolyLine(&xy[1], 5);
    sg->GetBoundingBox(&bbox, 0);
    sg->FillPath();

    // Overlay the bounding box on the filled shape
    aarend->SetColor(RGBA(200,180,160,90));  // beige
    sg->BeginPath();
    sg->Rectangle(bbox);
    sg->FillPath();

    // Move the shape to the right
    for (int i = 0; i < 6; ++i)
        xy[i].x += 320;

    // Stroke the shape with round joins
    aarend->SetColor(RGBX(0,190,255));  // blue
    sg->SetLineJoin(LINEJOIN_ROUND);
    sg->SetLineWidth(28.0);
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
    sg->PolyLine(&xy[1], 5);
    sg->CloseFigure();
    sg->GetBoundingBox(&bbox, FLAG_BBOX_STROKE);
    sg->StrokePath();

    // Overlay the bounding box on the stroked shape
    aarend->SetColor(RGBA(200,180,160,90));  // beige
    sg->BeginPath();
    sg->Rectangle(bbox);
    sg->FillPath();

    // Move the shape to the right
    for (int i = 0; i < 6; ++i)
        xy[i].x += 345;

    // Stroke the shape with mitered joins
    aarend->SetColor(RGBX(0,190,255));  // blue
    sg->SetLineJoin(LINEJOIN_MITER);
    sg->SetMiterLimit(1.6);
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
    sg->PolyLine(&xy[1], 5);
    sg->CloseFigure();
    sg->GetBoundingBox(&bbox, FLAG_BBOX_STROKE);
    sg->StrokePath();

    // Overlay the bounding box on the stroked shape
    aarend->SetColor(RGBA(200,180,160,90));  // beige
    sg->BeginPath();
    sg->Rectangle(bbox);
    sg->FillPath();
}
```

The output produced by this code example is shown in the following screenshot.



In this screenshot, three shapes are filled or stroked in blue. The bounding box that `GetBoundingBox` calculates for each shape is shown in beige. The path coordinates for the three shapes are identical, except for being shifted horizontally.

The shape on the left is filled by the `ShapeGen::FillPath` function. The middle shape is stroked by the `ShapeGen::StrokePath` function with round line joins and the line width set to 28 pixels. The shape on the right is stroked by the `StrokePath` function with the same line width, but with miter joins and a miter limit of 1.8.

The bounding boxes at the left and middle of the screenshot tightly circumscribe the enclosed shapes.

However, the bounding box on the right more loosely circumscribes the shape that is stroked with miter joins. That's because `GetBoundingBox` has to make worst-case assumptions about the extent of stroked shapes with miter joins. In the worst case, the two line segments that connect at a join are parallel or nearly parallel but point in opposite directions. Clipping this join to the miter limit leaves two roughly square corners that push the sides of the bounding box further outward as the line width increases.

Header

`shapegen.h`

See also

`SGRect`

`SGCoord`

`ShapeGen::SetFixedBits`

ShapeGen::GetCurrentPoint function

The `GetCurrentPoint` function retrieves the current point.

Syntax

```
C++
```

```
bool ShapeGen::GetCurrentPoint(
   SGPoint *cpoint
);
```

Parameters

cpoint

A pointer to a caller-supplied SGPoint structure. The function writes the current point's *x*-*y* coordinates to this structure. This pointer can be null (zero) if the caller simply wants to know whether the current point is defined.

Return value

Returns `true` if the current point is defined. If the current point is undefined (because the current figure is empty), the function immediately returns a value of `false` without writing to the structure pointed to by `cpoint`.

Remarks

The current point is the point most recently added to the current figure (or subpath).

The *x*-*y* coordinates retrieved by this function are converted to the user's SGCoord format – integer or fixed-point – and rounded off as appropriate. By default, SGCoord values are integers, but the user can call the ShapeGen::SetFixedBits function to switch to a fixed-point format.

Header

`shapegen.h`

See also

SGPoint

SGCoord

ShapeGen::SetFixedBits

ShapeGen::GetFirstPoint function

The `GetFirstPoint` function retrieves the first point in the current figure.

Syntax

C++

```
bool ShapeGen::GetFirstPoint(
   SGPoint *fpoint
);
```

Parameters

fpoint

A pointer to a caller-supplied SGPoint structure. The function writes the first point's *x-y* coordinates to this structure. This pointer can be null (zero) if the caller simply wants to know whether the first point is defined.

Return value

Returns `true` if the first point is defined. If the first point is undefined (because the current figure is empty), the function immediately returns a value of `false` without writing to the structure pointed to by `fpoint`.

Remarks

The first point is the initial point in the current figure (or subpath).

This function can be used to retrieve the starting point of an elliptic arc with a nonzero starting angle, as constructed by the ShapeGen::EllipticArc function.

The *x-y* coordinates retrieved by this function are converted to the user's SGCoord format – integer or fixed-point – and rounded off as appropriate. By default, SGCoord values are integers, but the user can call the ShapeGen::SetFixedBits function to switch to a fixed-point format.

Header

`shapegen.h`

See also

SGPoint

ShapeGen::EllipticArc

SGCoord

ShapeGen::SetFixedBits

ShapeGen::InitClipRegion function

The `InitClipRegion` function sets the device clipping rectangle to the specified width and height.

Syntax

C++

```
bool ShapeGen::InitClipRegion(
   int width,
   int height
);
```

Parameters

`width`

The width, in pixels, of the new device clipping rectangle.

`height`

The height, in pixels, of the new device clipping rectangle.

Return value

Returns `true` if the `width` and `height` parameters are both greater than zero. Otherwise, the function fails and immediately returns `false`. Before returning `false`, the function faults if the `NDEBUG` macro (used in `assert.h`) is undefined.

Remarks

In addition to changing the dimensions of the device clipping rectangle, this function sets the current clipping region to the updated device clipping rectangle.

ShapeGen always interprets the `width` and `height` parameter values in this function call as integers and never as fixed-point numbers. Only parameters of type `SGCoord` are affected by `ShapeGen::SetFixedBits` function calls.

The `InitClipRegion` function changes only the width and height of the device clipping rectangle – it has no effect on the position of the top-left corner of the device clipping rectangle relative to the ShapeGen coordinate origin. To change this position, call the `ShapeGen::SetScrollPosition` function.

The `ShapeGen::SetClipRegion` and `ShapeGen::SetMaskRegion` functions can modify the clipping region inside the device clipping rectangle. However, when an `InitClipRegion` or `SetScrollPosition` function call changes the size or position of the device clipping rectangle, the current clipping region is replaced by the new device clipping rectangle, and any previous clipping region set by the `SetClipRegion` and `SetMaskRegion` functions is discarded.

An `InitClipRegion` or `SetScrollPosition` function call discards any copy of a clipping region that was previously saved by the `ShapeGen::SaveClipRegion` function or swapped out by the `ShapeGen::SwapClipRegion` function.

The current path is not altered in any way by an `InitClipRegion` or `SetScrollPosition` function call.

Header

`shapegen.h`

See also

`SGCoord`

`ShapeGen::SetFixedBits`

`ShapeGen::SetScrollPosition`

`ShapeGen::SetClipRegion`

`ShapeGen::SetMaskRegion`

`ShapeGen::SaveClipRegion`

ShapeGen::SwapClipRegion

ShapeGen::Line function

The Line function constructs a straight line from the current point to the specified end point.

Syntax

```
C++

bool ShapeGen::Line(
   SGCoord x,
   SGCoord y
);
```

Parameters

x

The *x* coordinate of the end point for the line.

y

The *y* coordinate of the end point for the line.

Return value

Returns `true` if the function succeeds in constructing the line. If the current point is undefined (because the current figure is empty), the function fails and immediately returns `false`. Before returning `false`, the function faults if the NDEBUG macro (used in `assert.h`) is undefined.

Remarks

The current point is the starting point for the line. Parameters x and y specify the end point of the line.

On return from a Line call, the current point is set to the coordinates specified by parameters x and y.

The ShapeGen::PolyLine function can construct a list of connected line segments in a single function call.

Header

shapegen.h

See also

ShapeGen::PolyLine

ShapeGen::Move function

The Move function lifts the pen and moves it to a new starting point.

Syntax

C++

```
void ShapeGen::Move(
   SGCoord x,
   SGCoord y
);
```

Parameters

x

The *x* coordinate of the first point in the new figure.

y

The *y* coordinate of the first point in the new figure.

Return value

None

Remarks

This function starts a new figure (or subpath) and adds the first point to this figure. On return from a Move call, the current point is set to the coordinates specified by parameters x and y.

If, on entry to the Move function, the current figure has not already been finalized, the function finalizes the figure in the same manner as the ShapeGen::EndFigure function before starting the new figure.

If a Move call is followed by another Move call, with no intervening path-construction calls, the second Move call overwrites (that is, discards and replaces) the point specified by the first Move call.

Header

shapegen.h

See also

ShapeGen::EndFigure

ShapeGen::PolyBezier2 function

The PolyBezier2 function constructs one or more connected quadratic Bézier spline curves, starting at the current point.

Syntax

C++

```
bool ShapeGen::PolyBezier2(
   const SGPoint xy[],
   int npts
);
```

Parameters

xy

An SGPoint array containing two points for each quadratic Bézier spline. For example, an array of length npts = 10 describes five splines.

npts

The number of points in the xy array.

Return value

Returns true if the function succeeds in constructing the splines. If the current point is undefined (because the current figure is empty), the function fails and immediately returns false. Before returning false, the function faults if the NDEBUG macro (used in assert.h) is undefined.

Remarks

The current point is the starting point for the first spline. The first two elements in array xy specify the control point and end point of the first spline. If the array contains more than two points, the end point of the first spline becomes the starting point for the second spline, and so on.

Whereas the ShapeGen::Bezier2 function constructs a single quadratic Bézier curve, the PolyBezier2 function can construct multiple quadratic Bézier curves in a single call.

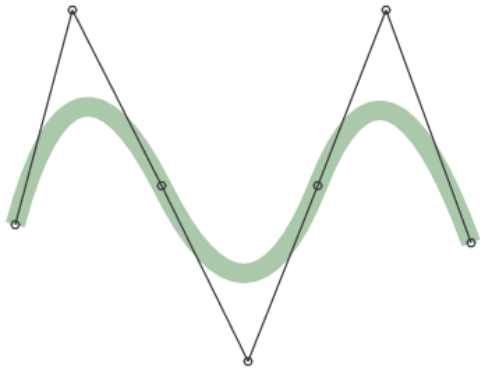
On return from this function, the end point of the final spline in the array is the new current point.

Example

This example uses the PolyBezier2 function to draw three connected quadratic Bézier spline curves. (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter clip specifies the device clipping rectangle.)

```
void example07(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint xy[] = {
        { 84, 224 }, { 125, 70 }, { 189, 196 }, { 251, 322 },
        { 301, 196 }, { 350, 70 }, { 411, 237 },
    };

    // Stroke the three connected quadratic Bezier splines in green
    aarend->SetColor(RGBX(170,200,170));
    sg->SetLineWidth(14.0);
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
```

```
    sg->PolyBezier2(&xy[1], 6);
    sg->StrokePath();

    // Outline the spline skeleton in black
    aarend->SetColor(RGBX(60,60,60));
    sg->SetLineWidth(1.25);
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
    sg->PolyLine(&xy[1], 6);
    for (int i = 0; i < 7; ++i)
    {
        // Mark the knots and control points
        SGPoint v0 = xy[i], v1 = v0, v2 = v0;
        v1.x += 3;
        v2.y += 3;
        sg->Ellipse(v0, v1, v2);
    }
    sg->StrokePath();
}
```



The output produced by this code example is shown in the screenshot at left.

The three connected splines are stroked in green, starting from the left. The spline skeleton is outlined in black, and the knots and control points are marked.

Header

shapegen.h

See also

SGPoint

ShapeGen::Bezier2

ShapeGen::PolyBezier3 function

The PolyBezier3 function constructs one or more connected cubic Bézier spline curves, starting at the current point.

Syntax

C++

```
bool ShapeGen::PolyBezier3(
   const SGPoint xy[],
   int npts
);
```

Parameters

xy

An SGPoint array containing three points for each quadratic Bézier spline. For example, an array of length npts = 6 describes two splines.

npts

The number of points in the xy array.

Return value

Returns true if the function succeeds in constructing the splines. If the current point is undefined (because the current figure is empty), the function fails and immediately returns false. Before returning false, the function faults if the NDEBUG macro (used in assert.h) is undefined.

Remarks

The current point is the starting point for the first spline. The first three elements in array xy specify the two control points and end point of the first spline. If the array contains more than three points, the end point of the first spline becomes the starting point for the second spline, and so on.

Whereas the ShapeGen::Bezier3 function constructs a single cubic Bézier curve, the PolyBezier3 function can construct multiple cubic Bézier curves in a single call.

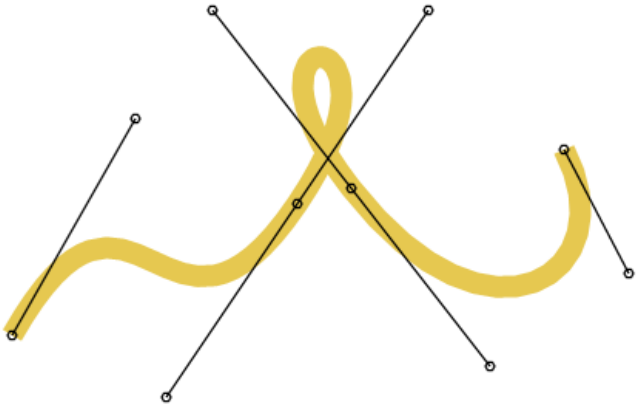
On return from this function, the end point of the final spline in the array is the new current point.

Example

This example uses the PolyBezier3 function to draw three connected cubic Bézier spline curves. (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter clip specifies the device clipping rectangle.)

```
void example08(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint xy[] = {
        { 50, 270 }, { 130, 130 }, { 150, 310 }, { 235, 185 }, { 320, 60 },
        { 180, 60 }, { 270, 175 }, { 360, 290 }, { 450, 230 }, { 408, 150 }
    };

    // Stroke the three connected cubic Bezier splines in yellow
    aarend->SetColor(RGBX(230,200,80));
    sg->SetLineWidth(14.0);
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
```

```
    sg->PolyBezier3(&xy[1], 9);
    sg->StrokePath();

    // Draw the spline handles in black
    aarend->SetColor(RGBX(0,0,0));
    sg->SetLineWidth(1.25);
    sg->BeginPath();
    for (int i = 0; i < 9; i += 3)
    {
        sg->Move(xy[i].x, xy[i].y);
        sg->Line(xy[i+1].x, xy[i+1].y);
        sg->Move(xy[i+2].x, xy[i+2].y);
        sg->Line(xy[i+3].x, xy[i+3].y);
    }
    for (int j = 0; j < 10; ++j)
    {
        SGPoint v0 = xy[j], v1 = v0, v2 = v0;
        v1.x -= 3;
        v2.y -= 3;
        sg->Ellipse(v0, v1, v2);
    }
    sg->StrokePath();

}
```



The output produced by this code example is shown in the screenshot at left.

The three connected splines are stroked in yellow, starting from the left edge of the screenshot. The spline handles are drawn in black.

Header

shapegen.h

See also

SGPoint

ShapeGen::Bezier3

ShapeGen::PolyEllipticSpline function

The PolyEllipticSpline function constructs one or more elliptic spline curves, starting at the current point.

Syntax

C++

```
bool ShapeGen::PolyEllipticSpline(
   const SGPoint xy[],
   int npts
);
```

Parameters

xy

An SGPoint array containing two points for each elliptic spline. For example, an array of length npts = 4 describes two splines.

npts

The number of points in the xy array.

Return value

Returns true if the function succeeds in constructing the splines. If the current point is undefined (because the current figure is empty), the function fails and immediately returns false. Before returning false, the function faults if the NDEBUG macro (used in assert.h) is undefined.

Remarks

The current point is the starting point for the first spline. The first two elements in array xy specify the control point and end point of the first spline. If the array contains more than three points, the end point of the first spline becomes the starting point for the second spline, and so on.

Whereas the ShapeGen::EllipticSpline function constructs a single elliptic spline, the PolyEllipticSpline function can construct multiple elliptic splines in a single call.

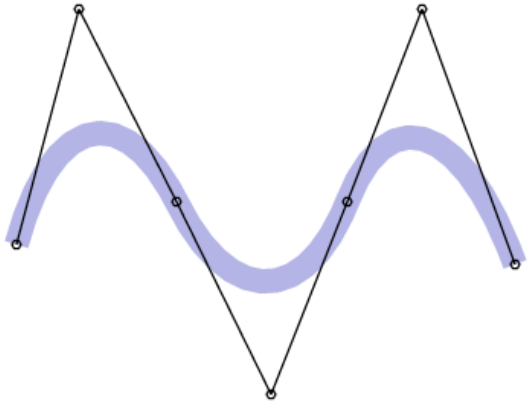
On return from this function, the end point of the final spline in the array is the new current point.

Example

This example uses the PolyEllipticSpline function to draw a series of connected elliptic spline curves. (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter clip specifies the device clipping rectangle.)

```
void example09(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint xy[] = {
        { 84, 224 }, { 125, 70 }, { 189, 196 }, { 251, 322 },
        { 301, 196 }, { 350, 70 }, { 411, 237 },
    };

    // Stroke the three connected elliptic splines in blue
```

```
    aarend->SetColor(RGBX(180,180,230));
    sg->SetLineWidth(16.0);
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
    sg->PolyEllipticSpline(&xy[1], 6);
    sg->StrokePath();

    // Outline the spline skeleton in black
    aarend->SetColor(RGBX(0,0,0));
    sg->SetLineWidth(1.25);
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
    sg->PolyLine(&xy[1], 6);
    for (int i = 0; i < 7; ++i)
    {
        // Mark the knots and control points
        SGPoint v0 = xy[i], v1 = v0, v2 = v0;
        v1.x -= 3;
        v2.y -= 3;
        sg->Ellipse(v0, v1, v2);
    }
    sg->StrokePath();
}
```



The output produced by this code example is shown in the screenshot at left.

The three connected splines are stroked in blue, starting from the left edge of the screenshot. The spline skeleton is outlined in black, and the knots and control points are marked.

Header

shapegen.h

See also

SGPoint

ShapeGen::EllipticSpline

ShapeGen::PolyLine function

The PolyLine function constructs one or more connected line segments, starting at the current point.

Syntax

C++

```
bool ShapeGen::PolyLine(
   const SGPoint xy[],
   int npts
);
```

Parameters

xy

An SGPoint array containing a point for each line segment. For example, an array of length npts = 5 describes five lines.

npts

The number of points in the xy array.

Return value

Returns true if the function succeeds in constructing the lines. If the current point is undefined (because the current figure is empty), the function fails and immediately returns false. Before returning false, the function faults if the NDEBUG macro (used in assert.h) is undefined.

Remarks

The current point is the starting point for the first line segment. The first element in the xy array specifies the end point of this line segment. If the array contains more than one point, the end point of the first line segment becomes the starting point for the second line segment, and so on.

Whereas the ShapeGen::Line function constructs a single line, the PolyLine function can construct multiple lines in a single call.

On return from this function, the end point of the last line segment is the new current point.

Header

shapegen.h

See also

SGPoint

ShapeGen::Line

ShapeGen::Rectangle function

The Rectangle function adds a rectangle to the current path.

Syntax

C++

```
bool ShapeGen::Rectangle(
   const SGRect& rect
);
```

Parameters

rect

An SGRect structure that specifies the rectangle to add to the path.

Return value

None

Remarks

A rectangle constructed by the Rectangle function is added to the current path as a complete, closed figure. If, on entry to the Rectangle function, the current figure has not already been finalized, the function finalizes the figure by leaving it open (that is, in the same manner as the ShapeGen::EndFigure function) and then starts a new figure in the same path. After adding the points in the rectangle to the new figure, the Rectangle function finalizes this new figure by closing it (in the same manner as the ShapeGen::CloseFigure function) before starting a newer, empty figure in the same path.

On return from the Rectangle function, the current point is undefined.

Example

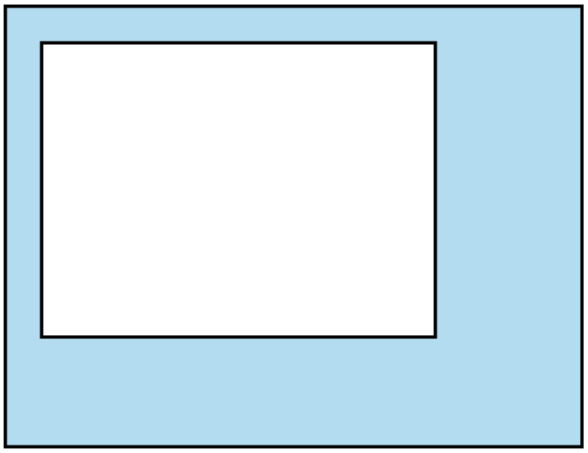
Construction of a rectangle by the Rectangle function proceeds in a clockwise direction if we assume the following:

- rect.w > 0 and rect.h > 0
- rect.x is the rectangle's left edge, and rect.y is the top edge

However, it's possible to modify the input parameters to the function so that the direction is reversed. In the following code example, the first (outer) rectangle is constructed in the CW direction, and the second in the CCW direction. (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter clip specifies the device clipping rectangle.)

```
void example10(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<SimpleRenderer> rend(CreateSimpleRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*rend), clip));
    SGRect rect = { 100, 75, 350, 265 };

    // Construction of the outer rectangle proceeds in the
    // clockwise direction (as seen on the display)
    sg->BeginPath();
    sg->Rectangle(rect);

    // Make the second rectangle smaller than the first
    rect.x += 22;
    rect.y += 22;
    rect.w -= 111;
```

```
    rect.h -= 88;

    // Modify the second rectangle's parameters so that its
    // construction proceeds in the counterclockwise direction
    rect.y += rect.h;
    rect.h = -rect.h;
    sg->Rectangle(rect);
    rend->SetColor(RGBX(180,220,240));
    sg->SetFillRule(FILLRULE_WINDING);  // <-- winding number fill rule!
    sg->FillPath();
    sg->SetLineWidth(2.0);
    sg->SetLineJoin(LINEJOIN_MITER);
    rend->SetColor(RGBX(0,0,0));
    sg->StrokePath();
}
```



The output produced by this code example is shown in the screenshot at left.

Header

shapegen.h

See also

SGRect

ShapeGen::EndFigure

ShapeGen::CloseFigure

ShapeGen::ResetClipRegion function

The ResetClipRegion function sets the current clipping region to the device clipping rectangle.

Syntax

C++

```
void ShapeGen::ResetClipRegion();
```

Parameters

None

Return value

None

Remarks

The device clipping rectangle is never undefined. When a ShapeGen object is created, the constructor sets the initial clipping region to the device clipping rectangle it receives as an input parameter. Thereafter, any changes made by the `ShapeGen::SetClipRegion` and `ShapeGen::SetMaskRegion` functions to the shape of the clipping region are always confined to the interior of the device clipping rectangle.

The `ResetClipRegion` function discards any changes to the clipping region that were made by previous calls to the `SetClipRegion` and `SetMaskRegion` functions.

Any changes made by previous calls to the `ShapeGen::InitClipRegion` and `ShapeGen::SetScrollPosition` functions are retained in the device clipping rectangle that is restored by the `ResetClipRegion` function. The `InitClipRegion` function changes the width and height of the device clipping rectangle. The `SetScrollPosition` function changes the position of the top-left corner of the device clipping rectangle in ShapeGen coordinate space.

The `ResetClipRegion` function preserves any copy of a clipping region that was previously saved by the `ShapeGen::SaveClipRegion` function or that was previously swapped out by the `ShapeGen::SwapClipRegion` function.

Header

`shapegen.h`

See also

`ShapeGen::SetClipRegion`

`ShapeGen::SetMaskRegion`

`ShapeGen::InitClipRegion`

`ShapeGen::SetScrollPosition`

`ShapeGen::SaveClipRegion`

`ShapeGen::SwapClipRegion`

ShapeGen::RoundedRectangle function

The `RoundedRectangle` function adds a rectangle with rounded corners to the current path.

Syntax

C++

```
bool ShapeGen::RoundedRectangle(
   const SGRect& rect
   const SGPoint& round
);
```

Parameters

rect

An SGRect structure that specifies the rectangle to add to the path.

round

An SGPoint structure that specifies the *x* (horizontal) and *y* (vertical) displacements of the elliptical arc starting and ending points from each corner of the rectangle.

Return value

None

Remarks

A rounded rectangle is a rectangle with rounded corners. The corners of the rectangle specified by the `rect` parameter are replaced with elliptic arcs. The `round` parameter specifies the horizontal and vertical dimensions of each arc.

To use circular arcs for the corners of the rectangle, set both components (that is, `x` and `y`) in the `round` parameter to the circle radius.

A rounded rectangle constructed by the `RoundedRectangle` function is added to the current path as a complete, closed figure. If, on entry to the `RoundedRectangle` function, the current figure has not already been finalized, the function finalizes the figure by leaving it open (that is, in the same manner as the `ShapeGen::EndFigure` function) and then starts a new figure in the same path. After adding the points in the rounded rectangle to the new figure, the `RoundedRectangle` function finalizes this new figure by closing it (in the same manner as the `ShapeGen::CloseFigure` function) and starting a newer, empty figure in the same path.

On return from the `RoundedRectangle` function, the current point is undefined.

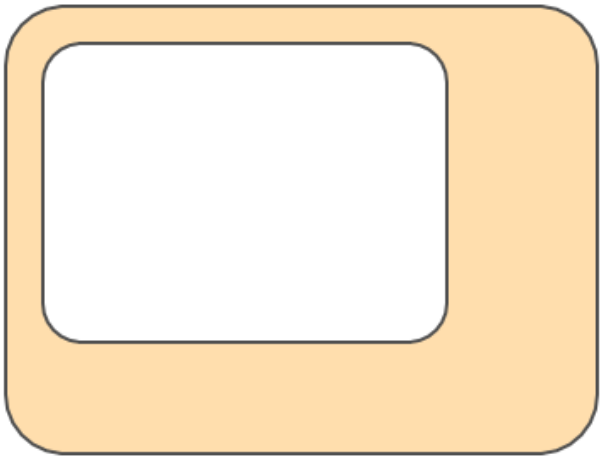
Example

Construction of a rounded rectangle by the `RoundedRectangle` function proceeds in a clockwise direction if we assume the following:

- `rect.w` > 0 and `rect.h` > 0
- `rect.x` is the rectangle's left edge, and `rect.y` is the top edge
- `round.x` > 0 and `round.y` > 0

However, it's possible to modify the input parameters to the function so that the direction is reversed. In the following code example, the first (outer) rounded rectangle is constructed in the CW direction, and the second in the CCW direction. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example11(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<SimpleRenderer> rend(CreateSimpleRenderer(&bkbuf));
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*rend), clip));
    SGRect rect = { 100, 75, 350, 265 };
    SGPoint round = { 35, 35 };

    // Construction of the outer rounded rectangle proceeds
    // in the clockwise direction (as seen on the display)
    sg->BeginPath();
    sg->RoundedRectangle(rect, round);

    // Make the second rectangle smaller than the first
    rect.x += 22;
    rect.y += 22;
    rect.w -= 111;
    rect.h -= 88;
    round.x = round.y -= 12;

    // Modify the second rectangle's parameters so that its
    // construction proceeds in the counterclockwise direction
    rect.y += rect.h;
    rect.h = -rect.h;
    round.y = -round.y;
    sg->RoundedRectangle(rect, round);
    rend->SetColor(RGBX(255,222,173));
    sg->SetFillRule(FILLRULE_WINDING);  // <-- winding number fill rule!
    sg->FillPath();

    // Switch to antialiasing renderer and stroke boundaries
    sg->SetRenderer(aarend);
    aarend->SetColor(RGBX(80,80,80));
    sg->SetLineWidth(2.0);
    sg->StrokePath();
}
```



The output produced by this code example is shown in the screenshot at left.

Header

shapegen.h

See also

SGRect

SGPoint

ShapeGen::EndFigure

ShapeGen::CloseFigure

ShapeGen::SaveClipRegion function

The SaveClipRegion function saves a copy of the current clipping region.

Syntax

C++

```
bool ShapeGen:: SaveClipRegion();
```

Parameters

None

Return value

Returns true if the current clipping region is not empty, in which case the saved copy of this clipping region is also not empty. Otherwise, the function returns false.

Remarks

A clipping region that is copied and saved by this function can be restored at a later time by calling the ShapeGen::SwapClipRegion function. Only one such copy exists at a time. Any previously existing copy of a clipping region is overwritten by a call to SaveClipRegion, or is swapped in by a SwapClipRegion call.

The saved copy of a clipping region is preserved through calls to the ShapeGen::ResetClipRegion, ShapeGen::SetClipRegion, and ShapeGen::SetMaskRegion functions.

A call to the ShapeGen::InitClipRegion, ShapeGen::SetScrollPosition, or ShapeGen::SetRenderer function causes any saved copy of a clipping region to be discarded and replaced with an empty clipping region.

ShapeGen clips all shapes, before they are rendered, to the interior of the current clipping region. An empty clipping region, which has no interior, effectively disables all drawing.

For example, if a SetClipRegion function call intersects the current clipping region with a path whose interior lies entirely outside the region, the resulting clipping region is empty.

Immediately after the ShapeGen object is created, the saved clipping region is, by default, empty.

Header

shapegen.h

See also

ShapeGen::SwapClipRegion

ShapeGen::ResetClipRegion

ShapeGen::SetClipRegion

ShapeGen::SetMaskRegion

ShapeGen::InitClipRegion

ShapeGen::SetScrollPosition

ShapeGen::SetRenderer

ShapeGen::SetClipRegion function

The SetClipRegion function sets the new clipping region to the intersection of the interior of the current clipping region with a shape constructed from the current path. This shape is constructed either by filling or by stroking the current path, as determined by the clipmode parameter.

Syntax

C++

```
bool ShapeGen::SetClipRegion(
   CLIPMODE clipmode = CLIPMODE_FILLPATH
);
```

Parameters

clipmode

Determines whether the shape that intersects the clipping region is to be constructed by filling or by stroking the current path. Specify one of the following values for this parameter:

`CLIPMODE_FILLPATH` – Construct the intersecting shape by *filling* the current path.

`CLIPMODE_STROKEPATH` – Construct the intersecting shape by *stroking* the current path.

If the caller does not explicitly specify a value for this parameter, it is automatically set to the default value, `CLIPMODE_FILLPATH`.

Return value

Returns `true` if the new clipping region is not empty; otherwise, returns `false`. Drawing occurs only in the interior of the clipping region. Thus, if the clipping region is empty, it has no interior and no drawing can occur.

Remarks

This function confines all drawing to the interior of an arbitrarily shaped area.

In contrast to the `ShapeGen::SetMaskRegion` function, which constructs a new clipping region that is the intersection of the current clipping region with the *exterior* of a filled or stroked shape, the `SetClipRegion` function constructs a new clipping region that is the intersection of the current clipping region with the *interior* of the shape.

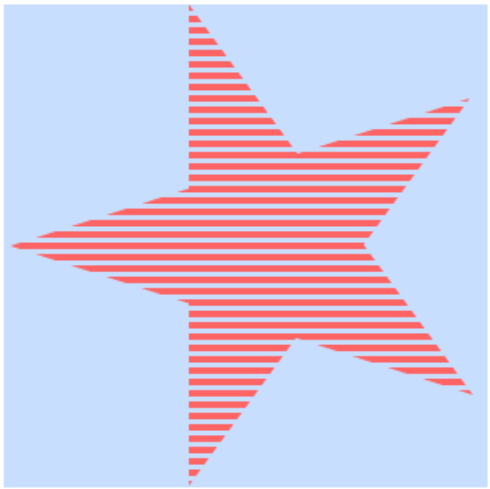
The `SetClipRegion` and `SetMaskRegion` functions can modify the clipping region inside the device clipping rectangle but cannot expand the clipping region beyond the bounds of the device clipping rectangle.

Example

This example uses the `SetClipRegion` function to set the clipping region to the interior of a star-shaped path. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example12(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    const float t = 0.8*PI;
    const float sint = sin(t);
    const float cost = cos(t);
    const int xc = 212, yc = 199;
    const SGRect rect = { 50, 50, 298, 298 };
    int xr = -158, yr = 0;
```

```
    // Set the clipping region to a 298x298-pixel square
    sg->BeginPath();
    sg->Rectangle(rect);
    sg->SetClipRegion(CLIPMODE_FILLPATH);  // use default fill rule

    // Do background fill with solid light blue
    sg->BeginPath();
    sg->Rectangle(clip);
    aarend->SetColor(RGBX(200, 222, 255));
    sg->FillPath();  // use default fill rule

    // Set the clipping region to a star-shaped area inside the square
    sg->BeginPath();
    sg->Move(xc + xr, yc + yr);
    for (int i = 0; i < 4; ++i)
    {
        int xtmp = xr*cost + yr*sint;
        yr = -xr*sint + yr*cost;
        xr = xtmp;
        sg->Line(xc + xr, yc + yr);
    }
    sg->SetFillRule(FILLRULE_WINDING);
    sg->SetClipRegion();  // use default clip mode

    // Draw a series of horizontal red lines through the square
    aarend->SetColor(RGBX(255,100,100));
    sg->SetLineWidth(4.0);
    sg->BeginPath();
    for (int y = rect.y+2; y <= rect.y+rect.h; y += 7)
    {
        sg->Move(rect.x, y);
        sg->Line(rect.x+rect.w, y);
    }
    sg->StrokePath();
}
```



The output produced by this code example is shown in the screenshot at left.

The first `SetClipRegion` call in this example sets the new clipping region to the intersection of the previous clipping region and the shape formed by filling the 298x298 square that was just added the current path. The background is then filled with solid blue, but the new clipping region restricts the fill to just this square.

Next, a new path containing a five-pointed star is constructed. A second `SetClipRegion` call further restricts the clipping region to the star shape, which is filled using the nonzero-winding-number fill rule.

Finally, a series of horizontal red lines is drawn through the blue square, but only the part of each line that lies inside the new, star-shaped clipping region gets drawn.

Header

shapegen.h

See also

ShapeGen::SetMaskRegion

ShapeGen::SetFillRule function

The SetFillRule function changes the ShapeGen object's fill-rule setting. This setting determines whether the *even-odd* fill rule or the *nonzero-winding-number* fill rule is to be used to convert the current path to a filled shape.

Syntax

C++

```
FILLRULE ShapeGen::SetFillRule(
   FILLRULE fillrule = FILLRULE_EVENODD
);
```

Parameters

fillrule

The fill rule to use for filling the path. Specify one of the following values for this parameter:

- FILLRULE_EVENODD – Even-odd (or parity) fill rule
- FILLRULE_WINDING – Nonzero-winding-number fill rule

If the caller does not explicitly supply a fillrule argument, the default value, FILLRULE_EVENODD, is automatically assigned to the fillrule parameter.

Return value

Returns the previous fill-rule setting.

Remarks

The current fill-rule setting is used by the ShapeGen::FillPath, ShapeGen::SetClipRegion, and ShapeGen::SetMaskRegion functions.

The FillPath function always uses the current fill rule to determine whether to construct a filled shape according to the even-odd rule or the nonzero-winding-number rule.

The SetClipRegion and SetMaskRegion functions use the current fill rule when their clipmode parameter is set to CLIPMODE_FILLPATH. In this case, SetClipRegion sets the new clipping region to the intersection of the previous clipping region with the *interior* of the filled shape that is constructed according to the fill rule. For SetMaskRegion, the new clipping region is set to the intersection of the previous clipping region with the *exterior* of the filled shape.

During the creation of the ShapeGen object, this object's fill-rule setting is automatically set to its default value, FILLRULE_EVENODD.

Header

shapegen.h

See also

ShapeGen::FillPath

ShapeGen::SetClipRegion

ShapeGen::SetMaskRegion

ShapeGen::SetFixedBits function

The SetFixedBits function specifies the new fixed-point format that the caller will use for SGCoord values in subsequent calls to ShapeGen functions.

Syntax

C++

```
int ShapeGen::SetFixedBits(
   int nbits = 0
);
```

Parameters

nbits

The number of bits of fraction in the fixed-point format for the caller's coordinate values. To specify that coordinate values are integers rather than fixed-point numbers, set this parameter to zero. Values for this parameter should be in the range 0 to 16.

If the caller does not explicitly supply an nbits argument, the default value, 0, is automatically assigned to the nbits parameter.

Return value

Returns the previous nbits value, if the function succeeds. If the new nbits parameter value is outside the range 0 to 16, the function fails and immediately returns a value of -1. Before returning -1, the function faults if the NDEBUG macro (used in assert.h) is undefined.

Remarks

By default, ShapeGen functions assume that all SGCoord values supplied by the caller are integers. The caller can opt to use fixed-point coordinates by calling the SetFixedBits function. At any time, the caller can switch back to using integer coordinates by calling SetFixedBits with nbits = 0.

The SGPoint and SGRect structures contain SGCoord members whose interpretation by ShapeGen is affected by SetFixedBits function calls.

To improve accuracy, the ShapeGen object uses 16.16 fixed-point coordinates rather than integer coordinates for its internal calculations. A 16.16 fixed-point number is stored as a 32-bit signed integer, but the 16 least-significant bits are assumed to lie to the right of the binary point and represent a fractional value.

Example

For example, the parameter value nbits = 16 specifies that the caller's SGCoord values are to be interpreted as 16.16 fixed-point numbers.

Header

shapegen.h

See also

SGCoord

SGPoint

SGRect

ShapeGen::SetFlatness function

The SetFlatness function sets the ShapeGen flatness attribute, which specifies the maximum the chord-to-curve error tolerance.

Syntax

C++

```
float ShapeGen::SetFlatness(
   float flatness = FLATNESS_DEFAULT
);
```

Parameters

flatness

The maximum chord-to-curve distance (in pixels) that can be tolerated. Set this parameter to a value in the range 0.2 to 100.0.

If the caller does not explicitly supply a flatness argument, the default value, 0.5, is automatically assigned to the flatness parameter.

Return value

Returns the previous flatness setting.

Remarks

ShapeGen approximates curves and arcs with connected straight line segments; that is, with chords. The `flatness` parameter specifies how flat a curve segment must be before it can be satisfactorily approximated with a chord. Smaller `flatness` values result in smoother-looking curves and arcs but do so at the cost of shorter and more numerous chords.

The default flatness attribute value is 0.5 pixels.

If the caller specifies a `flatness` parameter value that is outside the range 0.2 to 100.0 pixels, the function quietly clamps the value to this range.

Header

`shapegen.h`

See also

`ShapeGen::FillPath`

`ShapeGen::StrokePath`

`ShapeGen::SetClipRegion`

`ShapeGen::SetMaskRegion`

ShapeGen::SetLineDash function

The `SetLineDash` function specifies the dash pattern to use for stroked paths.

Syntax

C++

```
bool ShapeGen::SetLineDash(
   char *dash = 0,
   int offset = 0,
   float mult = 1.0
);
```

Parameters

dash

A zero-terminated byte array that specifies, in alternating fashion, the lengths of the dashes and of the gaps between dashes in the pattern. The first array element specifies a dash length, the second specifies a gap length, and so on. The effective length of a dash or gap, in pixels, is the product of the corresponding `dash` array element value and the dash-length multiplier, `mult`.

If the caller does not explicitly supply a `dash` argument, the default value, 0, is automatically assigned to the `dash` parameter.

offset

The starting offset into the dash pattern. The effective offset, in pixels, is the product of the offset and mult parameters. Should be greater than or equal to zero or the function fails.

If the caller does not explicitly supply an offset argument, the default value, 0, is automatically assigned to the offset parameter.

mult

The dash-length multiplier. Should be greater than zero or the function fails.

If the caller does not explicitly supply a mult argument, the default value, 1.0, is automatically assigned to the mult parameter.

Return value

Returns true if the function successfully updates the dash pattern. Otherwise, it returns false. Before returning false, the function faults if the NDEBUG macro (used in assert.h) is undefined.

Remarks

The dash pattern affects the appearance of stroked paths constructed by the ShapeGen::StrokePath function.

For each figure constructed by the StrokePath function, the function begins at the specified offset into the pattern and repeats the pattern as many times as needed to reach the end of the figure.

The maximum length of the dash array is 32 elements, not counting the terminating zero. A dash array longer than this maximum is quietly truncated to 32 elements.

The SetLineDash function treats each element of the dash array as an unsigned, 8-bit integer regardless of whether the compiler defines the char type to be signed or unsigned.

By default, stroked paths are constructed as solid lines (that is, with no dash pattern). To restore this default, call SetLineDash with dash = 0 (that is, a null pointer value). In this case, the offset and mult parameters are ignored.

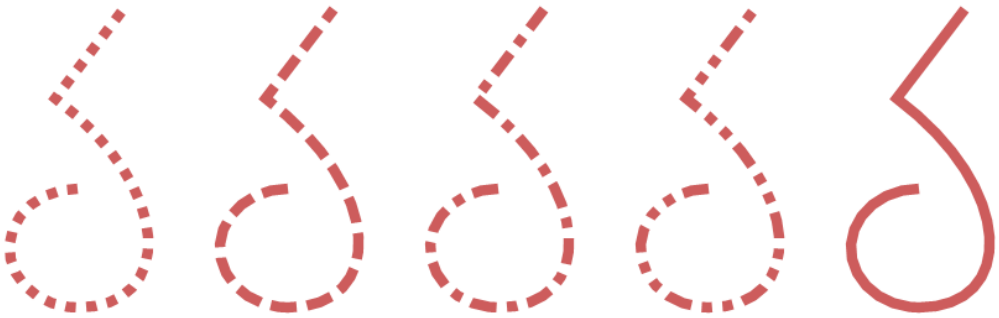
Example

This example uses the SetLineDash function to construct stroked paths with four different line dash patterns. (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter clip specifies the device clipping rectangle.)

```
void example13(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint xy[] = {
        { 127, 251 }, { 127, 203 }, { 72, 251 }, { 206, 299 },
        { 206, 203 }, { 109, 130 }, { 164, 58 },
    };
    float linewidth = 8.43;
    char dot[] = { 2, 0 };
    char dash[] = { 5, 2, 0 };
    char dashdot[] = { 5, 2, 2, 2, 0 };
    char dashdotdot[] = { 5, 2, 2, 2, 2, 2, 0 };
```

```
    char *pattern[] = { dot, dash, dashdot, dashdotdot, 0 };

    aarend->SetColor(RGBX(205, 92, 92));
    sg->SetLineWidth(linewidth);
    sg->SetLineJoin(LINEJOIN_MITER);
    for (int i = 0; i < 5; ++i)
    {
        sg->SetLineDash(pattern[i], 0, linewidth/2.0);
        sg->BeginPath();
        sg->EllipticArc(xy[0], xy[1], xy[2], 0, PI);  // PI = 3.14159...
        sg->PolyBezier3(&xy[3], 3);
        sg->Line(xy[6].x, xy[6].y);
        sg->StrokePath();
        for (int j = 0; j < 7; ++j)
            xy[j].x += 170;
    }
}
```

The output produced by this code example is shown in the following screenshot.



Header

shapegen.h

See also

ShapeGen::StrokePath

ShapeGen::SetLineEnd function

The SetLineEnd function sets the ShapeGen line-end attribute, which specifies how to cap the ends of stroked paths.

Syntax

C++

```
LINEEND ShapeGen::SetLineEnd(
   LINEEND capstyle = LINEEND_FLAT
);
```

Parameters

`capstyle`

The type of cap to use at the ends of stroked lines and curves. This parameter should be set to one of the following line-end attribute values:

- LINEEND_FLAT – Flat line end (or butt cap)
- LINEEND_ROUND – Rounded line end (or round cap)
- LINEEND_SQUARE – Squared line end (or projecting cap)

If the caller does not explicitly supply a `capstyle` argument, the default value, LINEEND_FLAT, is automatically assigned to the `capstyle` parameter.

Return value

Returns the previous line-end attribute value.

Remarks

The line-end attribute affects the appearance of stroked paths subsequently constructed by the `ShapeGen::StrokePath` function.

The default value for the line-end attribute is LINEEND_FLAT.

Example

This example uses the `SetLineEnd` function to set different line-end attributes for three stroked paths. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example14(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    LINEEND cap[] = { LINEEND_FLAT, LINEEND_ROUND, LINEEND_SQUARE };
    SGPoint vert[] = { { 84, 288 }, { 204, 114 }, { 264, 324 } };

    for (int i = 0; i < 3; ++i)
    {
        sg->BeginPath();
        sg->Move(vert[0].x, vert[0].y);
        sg->PolyLine(&vert[1], 2);
        sg->SetLineWidth(48.0);
        sg->SetLineEnd(cap[i]);
        aarend->SetColor(RGBX(135,206,235));
        sg->StrokePath();
        sg->SetLineWidth(2.0);
        aarend->SetColor(RGBX(0,0,0));
        sg->StrokePath();
        for (int j = 0; j < 3; ++j)
            vert[j].x += 295;
    }
}
```

The output produced by this code example is shown in the following screenshot.

From left to right, the stroked paths are drawn with line-end attributes of LINEEND_FLAT, LINEEND_ROUND, LINEEND_SQUARE. The path skeletons are outlined in black.

Header

`shapegen.h`

See also

`ShapeGen::StrokePath`

ShapeGen::SetLineJoin function

The `SetLineJoin` function sets the ShapeGen line-join attribute, which specifies how two connecting line segments in a stroked path are to be joined.

Syntax

C++

```
LINEJOIN ShapeGen::SetLineJoin(
   LINEJOIN joinstyle = LINEJOIN_BEVEL
);
```

Parameters

`joinstyle`

The way in which connecting line segments are to be joined. Set this parameter to one of the following join-style attribute values:

- LINEJOIN_BEVEL – Beveled join style
- LINEJOIN_ROUND – Rounded join style
- LINEJOIN_MITER – Mitered join style
- LINEJOIN_SVG_MITER – SVG default join style

If the caller does not explicitly supply a `joinstyle` argument, the default value, LINEJOIN_BEVEL, is automatically assigned to the `joinstyle` parameter.

Return value

Returns the previous line-join attribute value.

Remarks

The line-join attribute affects the appearance of stroked paths constructed by the `ShapeGen::StrokePath` function.

The default value for the line-join attribute is LINEJOIN_BEVEL.

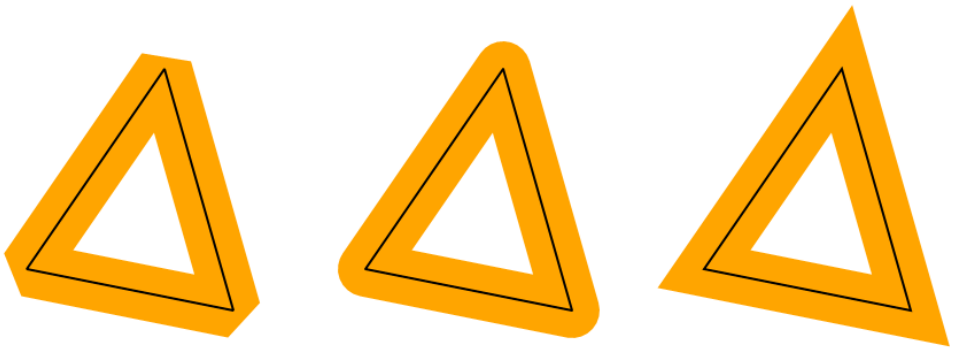
The LINEJOIN_SVG_MITER line-join attribute value is provided for compatibility with the SVG 2-D graphics standard.

Example

This example uses the `SetLineJoin` function to set different line-join attributes for three stroked paths. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example15(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    LINEJOIN join[] = { LINEJOIN_BEVEL, LINEJOIN_ROUND, LINEJOIN_MITER };
    SGPoint vert[] = { { 84, 288 }, { 204, 114 }, { 264, 324 } };

    for (int i = 0; i < 3; ++i)
    {
        sg->BeginPath();
        sg->Move(vert[0].x, vert[0].y);
        sg->PolyLine(&vert[1], 2);
        sg->SetLineWidth(48.0);
        sg->SetLineJoin(join[i]);
        sg->CloseFigure();
        aarend->SetColor(RGBX(255,165,0));
        sg->StrokePath();
        sg->SetLineWidth(2.0);
        aarend->SetColor(RGBX(0,0,0));
        sg->StrokePath();
        for (int j = 0; j < 3; ++j)
            vert[j].x += 295;
    }
}
```

The output produced by this code example is shown in the following screenshot.



From left to right, the stroked paths are drawn with line-join attributes of LINEJOIN_BEVEL, LINEJOIN_ROUND, and LINEJOIN_MITER. The path skeletons are outlined in black.

Header

`shapegen.h`

See also

`ShapeGen::StrokePath`

ShapeGen::SetLineWidth function

The `SetLineWidth` function sets the width of stroked paths.

Syntax

C++

```
float ShapeGen::SetLineWidth(
   float width = LINEWIDTH_DEFAULT
);
```

Parameters

`width`

The line width (in pixels) of a stroked path.

If the caller does not explicitly supply a `width` argument, the default value, 4.0, is automatically assigned to the `width` parameter.

Return value

Returns the previous line-width attribute value.

Remarks

The line-width setting determines the width of stroked paths constructed by the `ShapeGen::StrokePath` function.

The default line-width setting is 4.0 pixels.

In addition to the line-width setting, the appearance of a stroked path is affected by the following attributes:

- Dashed-line pattern
- Line-join style
- Line-end cap style
- Miter limit

However, these attributes do not apply to a stroked path constructed with a line-width setting of zero, which is a special value that is typically used in conjunction with a simple renderer (no antialiasing).

If the line width is zero, a stroked line is constructed as a thinly connected string of pixels that mimic the appearance of a line drawn by the Bresenham line algorithm. With this special line-width setting, stroked paths are always appear as solid lines (that is, with no dashed-line pattern). These stroked paths have beveled joins and triangular line-end caps, although these features might be difficult to discern due to their small size.

Header

`shapegen.h`

See also

`ShapeGen::StrokePath`

ShapeGen::SetMaskRegion function

The `SetMaskRegion` function sets the new clipping region to the intersection of the current clipping region and the *exterior* of a shape constructed from the current path. This shape is constructed either by filling or by stroking the current path, as determined by the `clipmode` parameter.

Syntax

C++

```
bool ShapeGen::SetMaskRegion(
   CLIPMODE clipmode = CLIPMODE_FILLPATH
);
```

Parameters

`fillrule`

Determines whether the shape that is to be excluded from the clipping region is to be constructed by filling or by stroking the current path. Specify one of the following values for this parameter:

`CLIPMODE_FILLPATH` – Construct the excluded shape by *filling* the current path.

`CLIPMODE_STROKEPATH` – Construct the excluded shape by *stroking* the current path.

If the caller does not explicitly specify a value for this parameter, it is automatically set to the default value, `CLIPMODE_FILLPATH`.

Return value

Returns `true` if the new clipping region is not empty; otherwise, returns `false`. Drawing occurs only in the interior of the clipping region. Thus, if a clipping region is empty, it has no interior and no drawing can occur.

Remarks

This function masks off an arbitrarily shaped area so that drawing can occur only outside this area.

In contrast to the `ShapeGen::SetMaskRegion` function, which constructs a new clipping region that is the intersection of the current clipping region with the *exterior* of a filled or stroked shape, the `SetClipRegion` function constructs a new clipping region that is the intersection of the current clipping region with the *interior* of the shape.

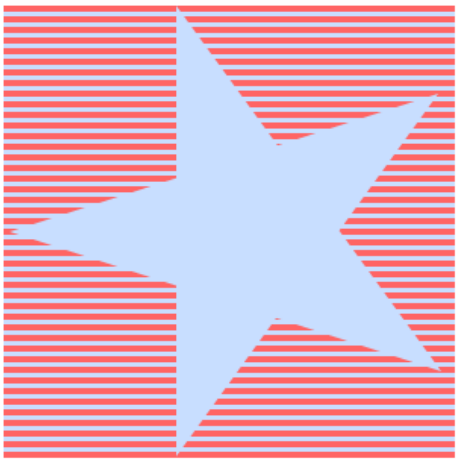
The `SetClipRegion` and `SetMaskRegion` functions can modify the clipping region inside the device clipping rectangle but cannot expand the clipping region beyond the bounds of the device clipping rectangle.

Example

This example uses the `SetMaskRegion` function to exclude a star-shaped path from the interior of the clipping region. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example16(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    const float t = 0.8*PI;
    const float sint = sin(t);
    const float cost = cos(t);
    const int xc = 212, yc = 199;
    const SGRect rect = { 50, 50, 298, 298 };
    int xr = -158, yr = 0;

    // Fill the rectangle with solid light blue
    sg->BeginPath();
    sg->Rectangle(rect);
    aarend->SetColor(RGBX(200, 222, 255));
    sg->FillPath();  // use default fill mode

    // Mask off a star-shaped area from the clipping region
    sg->BeginPath();
    sg->Move(xc + xr, yc + yr);
    for (int i = 0; i < 4; ++i)
    {
        int xtmp = xr*cost + yr*sint;
        yr = -xr*sint + yr*cost;
        xr = xtmp;
        sg->Line(xc + xr, yc + yr);
    }
    sg->SetFillRule(FILLRULE_WINDING);
    sg->SetMaskRegion(CLIPMODE_FILLPATH);
```

```
    // Draw a series of horizontal, red lines through the square
    aarend->SetColor(RGBX(255,100,100));
    sg->SetLineWidth(4.0);
    sg->BeginPath();
    for (int y = rect.y+2; y <= rect.y+rect.h; y += 7)
    {
        sg->Move(rect.x, y);
        sg->Line(rect.x+rect.w, y);
    }
    sg->StrokePath();
}
```



The output produced by this code example is shown in the screenshot at left.

First, a 298x298 square is filled with solid blue.

Next, a new path containing a five-pointed star is constructed. A `SetMaskRegion` call then sets the interior of the new clipping region to the intersection of the previous clipping region and the *exterior* of the star shape, which is filled using the nonzero-winding-number fill rule.

Finally, a series of horizontal red lines is constructed through the blue square, but only the part of each line that lies inside the new clipping region (which lies *outside* the masked-off area) gets drawn.

Header

`shapegen.h`

See also

`ShapeGen::SetClipRegion`

ShapeGen::SetMiterLimit function

The `SetMiterLimit` function sets the value of the ShapeGen miter-limit attribute, which specifies the maximum length that a mitered join can reach before the point is automatically beveled off.

Syntax

C++

```
float ShapeGen::SetMiterLimit(
   float mlim = MITERLIMIT_DEFAULT
);
```

Parameters

`mlim`

The new miter-limit setting. Set this parameter to a value greater than or equal to 1.0.

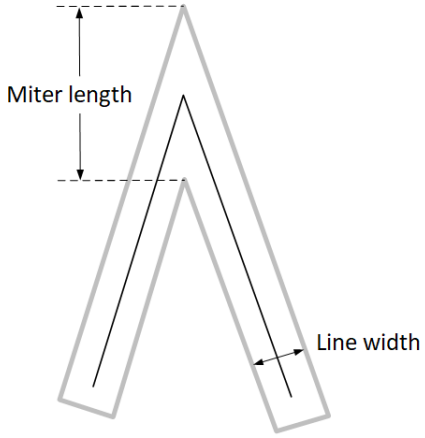
If the caller does not explicitly supply an `mlim` argument, the default value, 10.0, is automatically assigned to the `mlim` parameter.

Return value

Returns the previous miter-limit attribute value.

Remarks

This function specifies the miter limit, which determines the maximum length of a mitered join in a stroked path. This length, the *miter length*, is shown in the following figure.



For a given miter limit value, `mlim`, the maximum miter length is calculated as

max_miter_length = `mlim` * line_width

The `ShapeGen::StrokePath` function automatically snips off the sharp point of a mitered join that exceeds this limit, turning it into a beveled join whose length matches the max_miter_length value calculated above.

The default miter-limit setting is 10.0.

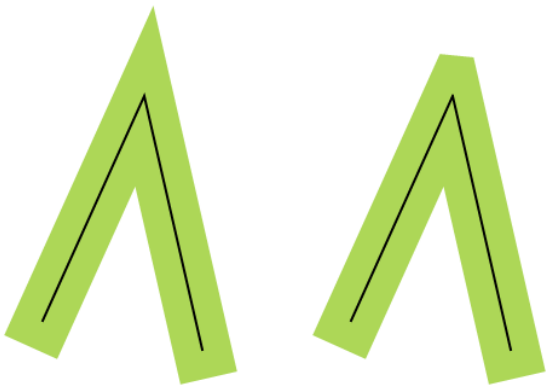
The minimum miter-limit setting is 1.0. A miter limit of 1.0 specifies that mitered joins at all angles are to be beveled. If the caller specifies an mlim parameter value less than 1.0, the function quietly clamps the miter limit to 1.0.

To specify that stroked paths are to be constructed with mitered joins, call the `ShapeGen::SetLineJoin` function with `joinstyle` = LINEJOIN_MITER. By default, stroked paths are constructed with beveled joins.

Example

This example uses the `SetMiterLimit` function to set different miter limits for two mitered joins. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example17(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint vert[] = { { 120, 300 }, { 204, 114 }, { 252, 324 } };

    sg->SetLineEnd(LINEEND_SQUARE);
    sg->SetLineJoin(LINEJOIN_MITER);
    sg->SetMiterLimit(4.0);
    for (int i = 0; i < 2; ++i)
    {
        sg->BeginPath();
        sg->Move(vert[0].x, vert[0].y);
        sg->PolyLine(&vert[1], 2);
        sg->SetLineWidth(48.0);
        aarend->SetColor(RGBX(173,215,87));
        sg->StrokePath();
        sg->SetLineWidth(2.0);
        aarend->SetColor(RGBX(0,0,0));
        sg->StrokePath();
        sg->SetMiterLimit(1.4);
        for (int j = 0; j < 3; ++j)
            vert[j].x += 255;
    }
}
```



The output produced by this code example is shown in the screenshot at left.

The stroked path on the left side of the screenshot is drawn with a miter-limit setting of 4.0. The stroked path on the right side is drawn with a miter-limit setting of 1.4.

Header

shapegen.h

See also

ShapeGen::StrokePath

ShapeGen::SetLineJoin

ShapeGen::SetRenderer function

The SetRenderer function sets the Renderer object that ShapeGen uses to render filled and stroked shapes on the display device.

Syntax

C++

```
bool ShapeGen::SetRenderer(
   Renderer *rend
);
```

Parameters

rend

A pointer to a `Renderer` object.

Return value

Returns `true` if the function call is successful. If `rend` = 0 (null pointer), the function fails and returns `false`. Before returning `false`, the function faults if `NDEBUG` (used in `assert.h`) is undefined.

Remarks

A ShapeGen object is always paired with a `Renderer` object. A nonnull `Renderer` object pointer is a required ShapeGen constructor parameter. At any time, the user can call `SetRenderer` to change the `Renderer` object associated with the ShapeGen object.

The `SetRenderer` call has these side effects:

- The current clipping region is reset to the device clipping rectangle. The effect is the same as a call to the `ShapeGen::ResetClipRegion` function.
- Any clipping region previously saved by the `ShapeGen::SaveClipRegion` function or swapped out by the `ShapeGen::SwapClipRegion` function is discarded.

For more information, see `Renderer`.

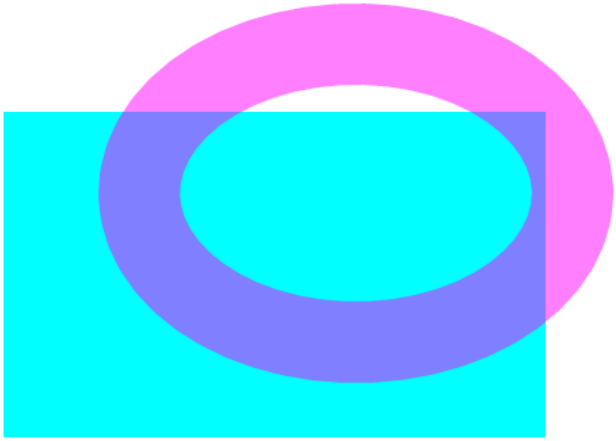
Example

This example uses the `SetRenderer` function to switch from one renderer to another. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example18(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<SimpleRenderer> rend(CreateSimpleRenderer(&bkbuf));
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*rend), clip));
    SGRect rect = { 100, 110, 400, 240 };
    SGPoint v0 = { 360, 170 }, v1 = { 360+160, 170 }, v2 = { 360, 170+110 };

    // Use the simple renderer to fill a cyan rectangle
    sg->BeginPath();
    sg->Rectangle(rect);
    rend->SetColor(RGBX(0,255,255));  // cyan (100% opaque)
    sg->FillPath();
```

```
    // Switch to the enhanced renderer
    sg->SetRenderer(aarend);

    // Alpha-blend a stroked magenta ellipse over the rectangle
    sg->BeginPath();
    sg->Ellipse(v0, v1, v2);
    aarend->SetColor(RGBA(255,0,255,128));  // magenta (50% opaque)
    sg->SetLineWidth(60.0);
    sg->StrokePath();
}
```

The output produced by this code example is shown in the following screenshot.



This example uses smart pointers to create a simple renderer, `rend`, and an enhanced renderer, `aarend`. Next, `rend` is installed as the initial renderer for the ShapeGen object, which is made accessible through another smart pointer, `sg`. The ShapeGen object then uses `rend` to draw a cyan rectangle. This rectangle is completely opaque, as are all shapes painted by a simple renderer.

Next, a `SetRenderer` function call installs the enhanced renderer, `aarend`, in place of the simple renderer. Using the new renderer, a partially transparent ellipse is stroked in magenta over the cyan rectangle. This example uses the RGBA macro (defined in `renderer.h`) to construct a 32-bit RGBA pixel value with an 8-bit alpha component value of 128, which makes the pixel 50.2 percent opaque.

Header

`shapegen.h`

See also

`ShapeGen::ResetClipRegion`

`ShapeGen::SaveClipRegion`

`ShapeGen::SwapClipRegion`

ShapeGen::SetScrollPosition function

The `SetScrollPosition` function enables a viewer to scroll and pan through a "virtual" 2-D image that is larger than the available drawing area on the screen.

Syntax

C++

```
void ShapeGen:: SetScrollPosition(
   int x,
   int y
);
```

Parameters

x

The horizontal scrolling displacement, in pixels, from the origin of the ShapeGen x-y cooordinate space.

y

The vertical scrolling displacement, in pixels, from the origin of the ShapeGen x-y cooordinate space.

Return value

None

Remarks

The `SetScrollPosition` function changes the position of the top-left corner of the device clipping rectangle relative to the ShapeGen *x-y* coordinate origin. The device clipping rectangle is a mapping of a rectangular portion of ShapeGen *x-y* coordinate space to a window (or viewport) on the graphics display device. Thus, if the user program constructs a path containing a virtual 2-D image that is larger than the window, the `SetScrollPosition` function can be used to scroll and pan through the image. Clipping prevents drawing from occurring outside the window.

If input parameters x and y are both zero, the top-left corner of the target window on the graphics display coincides with the ShapeGen *x-y* coordinate origin. Increasing the value of *x* causes the window to pan to the right. Increasing the value of *y* causes the window to scroll downward.

ShapeGen always interprets parameter values x and y as integers. Note that only parameters of type `SGCoord` are affected by `ShapeGen::SetFixedBits` function calls.

The `SetScrollPosition` function changes only the *position* of the device clipping rectangle – it has no effect on its width or height. To change the width and height of the device clipping rectangle, call the `ShapeGen::InitClipRegion` function.

The `ShapeGen::SetClipRegion` and `ShapeGen::SetMaskRegion` functions can modify the clipping region inside the device clipping rectangle. However, when a `SetScrollPosition` or `InitClipRegion` call changes the position or size of the device clipping rectangle, the current clipping region is replaced by the new device clipping rectangle, and any previous clipping region set by the `SetClipRegion` and `SetMaskRegion` functions is discarded.

Also, a `SetScrollPosition` or `InitClipRegion` function call discards any copy of a clipping region that was previously saved by the `ShapeGen::SaveClipRegion` function or swapped out by the `ShapeGen::SwapClipRegion` function.

The current path is not altered in any way by a `SetScrollPosition` or `InitClipRegion` function call.

A ShapeGen user program can construct a path containing a shape that is larger than the available drawing area on the screen. In this case, a viewer can use the `SetScrollPosition` function to inspect all parts of the shape by scrolling and panning through it. However, this function is not meant to provide smooth animation. Note that the shape must be redrawn after each `SetScrollPosition` call. Additionally, using this function with a shape that is much larger than the device clipping rectangle might incur significant clipping overhead.

To observe the effect of the `SetScrollPosition` function, run either the ShapeGen demo program or the SVG file viewer included in this GitHub project. Then, while holding the control key down, press any arrow key to generate calls to this function.

Header

`shapegen.h`

See also

`SGCoord`

`ShapeGen::SetFixedBits`

`ShapeGen::InitClipRegion`

`ShapeGen::SetClipRegion`

`ShapeGen::SetMaskRegion`

`ShapeGen::SaveClipRegion`

`ShapeGen::SwapClipRegion`

ShapeGen::StrokePath function

The `StrokePath` function strokes the current path.

Syntax

C++

```
bool ShapeGen::StrokePath();
```

Parameters

None

Return value

Returns `true` if the path, after being stroked and clipped, was not empty – in this case, the function sent a description of the resulting path to the renderer to be filled. Otherwise, the function returns `false` to indicate that the resulting path was empty and that nothing was sent to the renderer.

Remarks

The appearance of a stroked path is affected by several stroked-path attributes. The following table contains a list of stroked-path attributes, the default settings of these attributes, and the functions that change the attribute values.

Attribute	Default setting	Function
Line dash pattern	No dash pattern (solid line)	ShapeGen::SetLineDash
Line end cap style	Flat (or butt) cap	ShapeGen::SetLineEnd
Line join style	Beveled join	ShapeGen::SetLineJoin
Line width	4.0	ShapeGen::SetLineWidth
Miter limit	10.0	ShapeGen::SetMiterLimit

The values held by these attributes during the time the path is being constructed are irrelevant. The appearance of a stroked path is affected only by the attribute values at the time of the StrokePath call.

The ShapeGen::EndFigure and ShapeGen::CloseFigure functions affect the appearance of stroked paths but have no effect on the appearance of filled paths. Shapes filled by the ShapeGen::FillPath function are always constructed as though the first and last points in each figure are connected, regardless of any previous calls to EndFigure or CloseFigure.

Header

shapegen.h

See also

ShapeGen::SetLineDash

ShapeGen::SetLineEnd

ShapeGen::SetLineJoin

ShapeGen::SetLineWidth

ShapeGen::SetMiterLimit

ShapeGen::EndFigure

ShapeGen::CloseFigure

ShapeGen::FillPath

ShapeGen::SwapClipRegion function

The SwapClipRegion function swaps the current clipping region with a previously saved copy of a clipping region.

Syntax

C++

```
bool ShapeGen:: SwapClipRegion();
```

Parameters

None

Return value

The function returns `true` if the new clipping region is not empty. Otherwise, it returns `false`.

Remarks

This function exchanges the current clipping region with the copy of a clipping region that was previously saved or swapped out. Only one such copy exists at a time. This copy was either swapped out by an earlier call to SwapClipRegion, or was previously saved by a `ShapeGen::SaveClipRegion` function call.

The saved copy of a clipping region is preserved through calls to the `ShapeGen::ResetClipRegion`, `ShapeGen::SetClipRegion`, and `ShapeGen::SetMaskRegion` functions.

A call to the `ShapeGen::InitClipRegion`, `ShapeGen::SetScrollPosition`, or ShapeGen::SetRenderer function causes any saved copy of a clipping region to be discarded and replaced with an empty clipping region.

ShapeGen clips all shapes, before they are rendered, to the interior of the current clipping region. An empty clipping region, which has no interior, effectively disables all drawing.

For example, if a `SetClipRegion` function call intersects the current clipping region with a path whose interior lies entirely outside the region, the resulting clipping region is empty.

Immediately after the ShapeGen object is created, the saved clipping region is, by default, empty.

Header

`shapegen.h`

See also

`ShapeGen::SaveClipRegion`

`ShapeGen::ResetClipRegion`

`ShapeGen::SetClipRegion`

`ShapeGen::SetMaskRegion`

`ShapeGen::InitClipRegion`

`ShapeGen::SetScrollPosition`

ShapeGen::SetRenderer

EnhancedRenderer functions

The following reference topics describe the functions that comprise the `EnhancedRenderer` programming interface. This interface is defined in the `renderer.h` header file included in this GitHub project.

Some of the functions in this section take pixel values as input parameters. These pixel values are specified to be of type COLOR, which is defined in renderer.h to be an unsigned 32-bit integer. The function descriptions that follow contain more information about pixel formats.

EnhancedRenderer::AddColorStop function

The AddColorStop function adds a new color stop to the renderer's internal color stop table.

Syntax

C++

```
void EnhancedRenderer::AddColorStop(float offset, COLOR color);
```

Parameters

offset

A float value in the range 0 to 1.0 that specifies the offset of the new color stop in the color stop table.

color

A COLOR value that specifies the color at the new color stop.

Return value

None

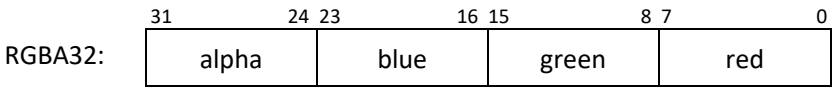
Remarks

The renderer's color stop table contains the list of color stops that are used to generate linear gradients and radial gradients.

To create a new color stop table, first call the EnhancedRenderer::ResetColorStops function to delete any previously added color stops. Next, through a series of calls to AddColorStop, add two or more color stops to the table. Color stops should be added in order of increasing offsets, starting with offset = 0 and ending with offset = 1.0.

To enable gradients to have abrupt changes in color, two successive color stops are allowed to have the same offset. If the abruptness of the color change results in objectionable aliasing, you can move the two offsets slightly apart to achieve a smoother transition.

The color parameter is a COLOR (unsigned 32-bit integer) value that contains 8-bit red, green, blue, and alpha components. These components are specified in RGBA32 pixel format, as follows:



To supply a color stop table for a linear gradient or radial gradient, construct the table *before* calling the EnhancedRenderer::SetLinearGradient or EnhancedRenderer::SetRadialGradient function.

Immediately after the EnhancedRenderer object is created, the renderer's color stop table is empty.

Example

This example uses the `AddColorStop` function to add three color stops to an initially empty color stop table. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example19(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    int i0, j0, i1, j1;
    float x0 = 300.0, y0 = 190.0;
    float x1 = 400.0, y1 = 190.0;
    char dash[] = { 1, 0 };

    // Add three color stops
    aarend->AddColorStop(0, RGBX(0,255,255));      // cyan
    aarend->AddColorStop(0.33, RGBX(240,255,22));  // yellow
    aarend->AddColorStop(1.0, RGBX(255,100,255));  // magenta

    // Set up the linear gradient
    aarend->SetLinearGradient(x0,y0, x1,y1, SPREAD_REPEAT,
                              FLAG_EXTEND_START | FLAG_EXTEND_END);

    // Use the gradient to fill a wide, stroked horizontal line
    i0 = x0 - 150, j0 = y0;
    i1 = x1 + 150, j1 = y0;
    sg->SetLineWidth(100.0);
    sg->SetLineEnd(LINEEND_ROUND);
    sg->BeginPath();
    sg->Move(i0, j0);
    sg->Line(i1, j1);
    sg->StrokePath();

    // Draw a dashed vertical black line through the
    // linear gradient's starting point at (x0,y0)
    sg->SetLineDash(dash, 0, 8.07);
    sg->SetLineWidth(2.0);
    aarend->SetColor(RGBX(0,0,0));
    i0 = x0, j0 = y0 - 85;
    i1 = x0, j1 = y0 + 85;
    sg->BeginPath();
    sg->Move(i0, j0);
    sg->Line(i1, j1);
    sg->StrokePath();

    // Draw a dashed vertical red line through the
    // linear gradient's ending point at (x1,y1)
    aarend->SetColor(RGBX(240,40,40));
    i0 = x1, j0 = y1 - 85;
    i1 = x1, j1 = y1 + 85;
    sg->BeginPath();
    sg->Move(i0, j0);
    sg->Line(i1, j1);
    sg->StrokePath();
}
```

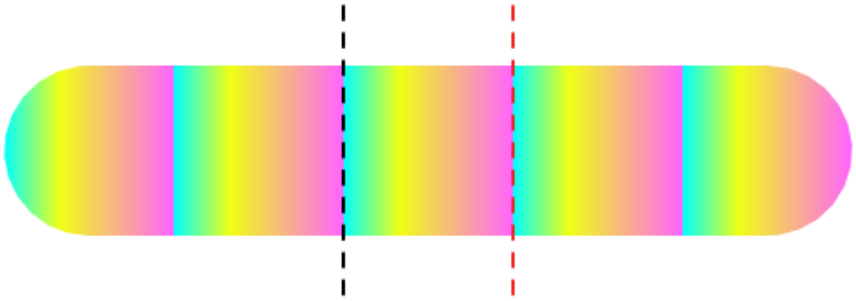
The first color stop in this example is set to cyan, and the last color stop is set to magenta. In between these two color stops, a third color stop, at `offset` = 0.33, is set to yellow. As required, the three color stops are added in order of increasing offsets, with the first and last offsets set to 0 and 1.0, respectively.

Next, an `EnhancedRenderer::SetLinearGradient` function call sets up the renderer to do linear gradient fills in *repeat* mode (`spread` = `SPREAD_REPEAT`). These gradient fills will use the new color stop table.

A wide horizontal line with rounded end caps is then stroked with the gradient pattern.

Finally, a dashed vertical black line is drawn through the linear gradient starting point (x0,y0), and a dashed vertical red line is drawn through the linear gradient ending point (x1,y1).

The output produced by this code example is shown in the following screenshot.



Header

`renderer.h`

See also

`EnhancedRenderer::ResetColorStops`

`EnhancedRenderer::SetLinearGradient`

`EnhancedRenderer::SetRadialGradient`

EnhancedRenderer::GetPixelBuffer

The `GetPixelBuffer` function retrieves a description of the pixel buffer that is managed by this renderer.

Syntax

C++

```
bool GetPixelBuffer(PIXEL_BUFFER *pixbuf);
```

Parameters

`pixbuf`

A pointer to a caller-supplied `PIXEL_BUFFER` structure into which the function writes a description of the renderer's pixel buffer.

Return value

If the renderer has a valid pixel buffer, the function returns `true` after writing a description of the buffer to the caller-supplied `PIXEL_BUFFER` structure. Otherwise, the function immediately returns `false`.

Remarks

A pixel buffer that is described by the `PIXEL_BUFFER` structure is a two-dimensional array of 32-bit pixels. These pixels are typically in either RGBA32 or BGRA32 format. These two formats are described in the Enhanced renderer pixel format topic. The `GetPixelBuffer` function enables the caller to retrieve a description of the pixel buffer that is managed by the renderer.

For more information about the `PIXEL_BUFFER` structure, see the Creating a ShapeGen object topic.

Code example

For a code example that uses the `GetPixelBuffer` function, see the Compositing and filtering with layers topic.

Header

`renderer.h`

EnhancedRenderer::ResetColorStops function

The `ResetColorStops` function deletes all color stops from the renderer's internal color stop table.

Syntax

C++

```
void EnhancedRenderer::ResetColorStops();
```

Parameters

None

Return value

None

Remarks

Before using the `EnhancedRenderer::AddColorStop` function to construct a new color stop table, call the `ResetColorStops` function to delete any previously added color stops from the table.

Header

`renderer.h`

See also

`EnhancedRenderer::AddColorStop`

EnhancedRenderer::SetBlendOperation function

The `SetBlendOperation` function sets the blending operation to use for rendering filled and stroked shapes.

Syntax

C++

```
void EnhancedRenderer::SetBlendOperation(
    BLENDOP blendop = BLENDOP_SRC_OVER_DST
);
```

Parameters

`blendop`

A `BLENDOP` enum value that specifies the blending operation to use for rendering shapes. Set this parameter to one of the following `BLENDOP` values:

- `BLENDOP_SRC_OVER_DST` – Source over destination
- `BLENDOP_ADD_WITH_SAT` – Add with saturation
- `BLENDOP_ALPHA_CLEAR` – Alpha clear

If the caller does not explicitly supply a `blendop` argument, the default value, `BLENDOP_SRC_OVER_DST`, is automatically assigned to the `blendop` parameter.

Return value

None

Remarks

The blendop parameter specifies the blending operation that the ShapeGen object will use to render filled and stroked shapes. The choice of blending operation determines how each source pixel in a shape is to be blended with the corresponding destination pixel in the target pixel buffer.

During creation of the `EnhancedRenderer` object, the object's blend-operation attribute is set to its default value, `BLENDOP_SRC_OVER_DST`. The current blend operation can be changed by calling the `SetBlendOperation` function.

The `BLENDOP_SRC_OVER_DST` blending operation is an implementation of the *A*-over-*B* operation described in the classic paper on alpha compositing by Porter & Duff.[8] *A*-over-*B* is the most widely used of the blending operations described in this paper, and, in common usage, the term *alpha blending*, without further qualification, typically implies the *A*-over-*B* operation. With this operation, an artist uses the *painter's algorithm* (back-to-front rendering) to compose images. A key strength of the *A*-over-*B* operation is its ability to render overlapping semi-transparent (semi-opaque) shapes.

[8] Thomas Porter, Tom Duff (1984). "Compositing Digital Images", *Computer Graphics*, **18**(3), https://graphics.pixar.com/library/Compositing/paper.pdf.

`BLENDOP_ADD_WITH_SAT` specifies an *add-with-saturation* operation in which the 8-bit alpha and color components in a source pixel are added to their respective components in the corresponding destination pixel. If this operation causes arithmetic overflow in one or more of the resulting 8-bit destination components, the component *saturates* at the maximum value for that component; that is, its value is clamped at 255. To use this operation, non-overlapping shapes are rendered to a target pixel buffer whose background is initially set to *transparent black* (all zeroes). The saturation feature protects against overflow when adjacent shapes accidentally overlap by a small (subpixel) amount due to arithmetic precision errors. Larger overlaps can result in visible artifacts.

`BLENDOP_ALPHA_CLEAR` specifies an *alpha-clear* operation. Alpha-clear can be used to set a region of a destination pixel buffer to transparent black in the same way as the *clear* operation in the Porter & Duff paper, but alpha-clear is more versatile. An alpha-clear operation multiplies the 8-bit alpha and color components in a destination pixel by the bitwise inverse of the 8-bit alpha component of the corresponding source pixel; the source color components are not used in this operation. Thus, if a source alpha component is 255, the destination pixel is set to transparent black. However, an alpha-clear operation can reduce a destination pixel's opacity without driving it all the way down to zero. For example, with a source alpha component equal to 128, the destination components are multiplied by (255-128), which decreases the destination pixel's opacity roughly by half. A source alpha value of zero leaves the destination pixel unchanged.

The add-with-saturation and alpha-clear operations are related to the *A*-over-*B* operation in an interesting way. Namely, if a shape is first rendered with an alpha-clear operation and the same shape is then rendered with an add-with-saturation operation, the result is the same as if the shape were rendered with the *A*-over-*B* operation.

For more information about these blending operations, see the topics titled The background-color-leak problem and Front-to-back rendering.

Header

`renderer.h`

See also

`ShapeGen::SetMaskRegion`

EnhancedRenderer::SetColor function

The `SetColor` function creates a solid-color paint generator for the renderer to use for subsequent fill and stroke operations. The new paint generator replaces any previously created paint generator.

Syntax

C++

```
void EnhancedRenderer::SetColor(COLOR color);
```

Parameters

`color`

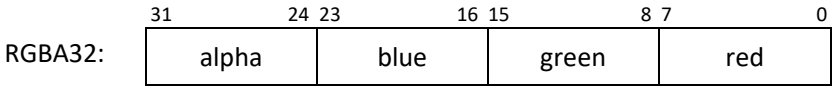
A COLOR value that specifies the solid color to use for subsequent fill operations.

Return value

None

Remarks

The `color` parameter is a COLOR (unsigned 32-bit integer) value that contains 8-bit red, green, blue, and alpha components. These components are specified in RGBA32 pixel format, as follows:



Immediately after the `EnhancedRenderer` object is created, the renderer has been configured to fill with solid opaque black.

Header

`renderer.h`

See also

None

EnhancedRenderer::SetConstantAlpha function

The `SetConstantAlpha` function sets the *source constant alpha* value that the renderer is to use for subsequent fill and stroke operations.

Syntax

C++

```
void EnhancedRenderer::SetConstantAlpha(COLOR alpha);
```

Parameters

`alpha`

A COLOR value in the range 0 (fully transparent) to 255 (fully opaque) that specifies the source constant alpha value to use for subsequent fill operations. Place this value in the 8 least-significant bits of the 32-bit `alpha` parameter; set the other 24 bits to zero.

Return value

None

Remarks

During fill or stroke operations, the renderer's source constant alpha is mixed with the paint generated for solid-color, pattern, and gradient fills. That is, the renderer combines the source constant alpha with the per-pixel alpha values from the current paint generator.

When a solid-color, pattern, or gradient paint generator is created, it takes a snapshot of the current source constant alpha value. For example, when an `EnhancedRenderer::SetLinearGradient` function call creates a linear-gradient paint generator, the generator captures the source constant alpha value that is in effect when the call occurs and then continues to use this value for fill operations until the paint generator is replaced.

Immediately after the `EnhancedRenderer` object is created, the source constant alpha has been set to its default value, which is 255 (fully opaque).

Example

This example uses the `SetConstantAlpha` function to change the opacity of four rectangles filled with a linear gradient. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

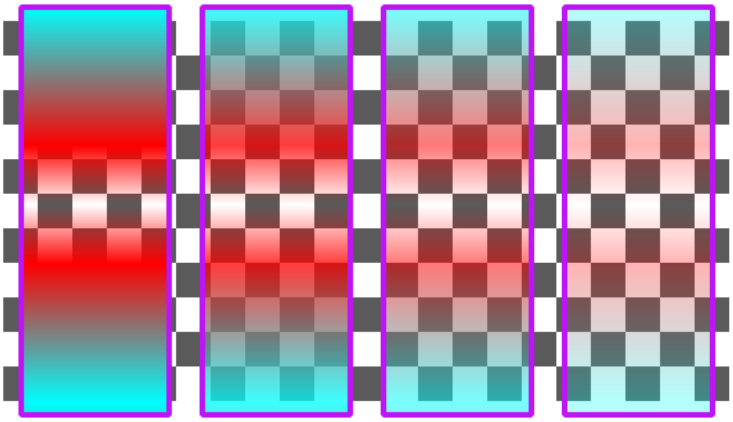
```
void example20(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    COLOR checker[4] = {
        RGBX(90,90,90), RGBX(255,255,255),
        RGBX(255,255,255), RGBX(90,90,90),
    };
    SGRect bkgd = { 40, 40, 525, 275 };
    SGRect rect = { 53, 30, 107, 295 };
    float xform[6] = { 0.040, 0, 0, 0.040, 0, 0 };

    // Draw checkerboard background pattern
    aarend->SetTransform(xform);
    aarend->SetPattern(checker, 1.6,1.6, 2,2, 2, 0);
    sg->BeginPath();
    sg->Rectangle(bkgd);
    sg->FillPath();
    aarend->SetTransform(0);

    // Set up color stop table
    aarend->AddColorStop(0, RGBX(0,255,255));  // cyan
    aarend->AddColorStop(0.7, RGBX(255,0,0));  // red
    aarend->AddColorStop(1.0, RGBA(0,0,0,0));  // transparent

    // Fill four rectangles with linear gradient
    for (int alpha = 255; alpha > 60; alpha -= 60)
    {
        aarend->SetConstantAlpha(alpha);
        aarend->SetLinearGradient(0,30, 0,173, SPREAD_REFLECT,
                                  FLAG_EXTEND_START | FLAG_EXTEND_END);
        sg->BeginPath();
        sg->Rectangle(rect);
        sg->FillPath();
        aarend->SetConstantAlpha(255);
        aarend->SetColor(RGBX(188,22,244));
        sg->StrokePath();
        rect.x += 131;
    }
}
```

In this example, the background is first filled with a black-and-white checkerboard pattern.

Next, a three-entry color stop table is set up with a gradient-fill pattern consisting of cyan, red, and fully transparent color stops.

Four rectangles (outlined in purple) are then filled with identical linear gradients, all of which use the same color stop table. The gradient is inherently opaque at the top and bottom of each rectangle and is transparent in the middle. The only difference among the four rectangles is the source constant alpha value that is applied to the gradient fill. For the rectangle on the left, the source constant alpha value is 255 (fully opaque), which means the only non-opaque part of the rectangle is due to the transparent pixel value in the color stop table. The source constant alpha values in the other three rectangles are, from left to right, 195, 135, and 75.

The output produced by this code example is shown in the following screenshot.



In the rectangle furthest to the right, even the fully opaque parts of the gradient pattern are nearly transparent when combined with the source constant alpha value.

Header

`renderer.h`

See also

`EnhancedRenderer::SetLinearGradient`

EnhancedRenderer::SetLinearGradient function

The `SetLinearGradient` function creates a linear-gradient paint generator for the renderer to use for subsequent fill and stroke operations. The new paint generator replaces any previously created paint generator.

Syntax

```
C++
```

```
void EnhancedRenderer:: SetLinearGradient(
   float x0,
   float y0,
   float x1,
   float y1,
   SPREAD_METHOD spread,
   int flags
);
```

Parameters

x0

A `float` value that specifies the *x*-coordinate at the starting point.

y0

A `float` value that specifies the *y*-coordinate at the starting point.

x1

A `float` value that specifies the *x*-coordinate at the ending point.

y1

A `float` value that specifies the *y*-coordinate at the ending point.

spread

A `SPREAD_METHOD enum` value that specifies whether the linear gradient is to spread according to the *repeat*, *reflect*, or *pad* method. The following `enum` values are defined for this parameter:

- `SPREAD_PAD` – Use the padding colors to fill pixels that lie outside the region between the starting and ending isolines (the pair of gradient isolines that pass through the starting and ending points; see Remarks below).
- `SPREAD_REPEAT` – Repeat the gradient pattern to fill pixels that lie outside the region between the starting and ending isolines.
- `SPREAD_REFLECT` – Alternately repeat and reflect the gradient pattern to fill pixels that lie outside the region between the starting and ending isolines.

flags

The following flag bits are defined for this parameter:

- `FLAG_EXTEND_START` – Extend the gradient fill beyond the starting isoline (the gradient isoline that passes through the starting point; see Remarks below).
- `FLAG_EXTEND_END` – Extend the gradient fill beyond the ending isoline.

If neither of these flags is set, gradient fills are confined to the region between the starting isoline and ending isoline. In this case, pixels outside this region are not filled, and, thus, the `spread` parameter has no effect.

Return value

None

Remarks

The color intervals in the linear-gradient fill pattern are specified in the renderer's color stop table. A linear-gradient paint generator takes a snapshot of the current color stop table during the `SetLinearGradient` function call and then uses these color stops for gradient-fill operations until the paint generator is replaced. For more information, see `EnhancedRenderer::AddColorStop` and `EnhancedRenderer::ResetColorStops`.

Linear-gradient fill operations use the renderer's current constant alpha, which is set by the `EnhancedRenderer::SetConstantAlpha` function.

Linear-gradient fill operations use the renderer's affine transformation matrix (set by the `EnhancedRenderer::SetTransform` function). A linear-gradient paint generator takes a snapshot of the matrix during the `SetLinearGradient` function call and then continues to use this matrix for linear-gradient fill operations until the paint generator is replaced.

Internally, the linear-gradient paint generator uses a parameter *t* to designate the gradient at a point in the gradient pattern. Parameter *t* has the value 0 at starting point (x0,y0) and has the value 1.0 at ending point (x1,y1). Isolines of constant t = 0 and t = 1.0 pass through the starting and ending points, respectively, and are normal to the vector from (x0,y0) to (x1,y1). Gradient *t* increases linearly from 0 to 1.0 over the interval from the starting isoline to the ending isoline, and the color stop table specifies the color changes over this interval.

Outside this interval, the `spread` and `flags` parameters specify the gradient fill behavior. If the FLAG_EXTEND_START flag is set, the gradient fill extends beyond the starting isoline, into the region for which $t < 0$. If the FLAG_EXTEND_END flag is set, the gradient fill extends beyond the ending isoline, into the region for which $t > 1.0$.

The SPREAD_PAD spread method uses solid colors to pad areas outside the region between the starting and ending isolines. The padding color for pixels that lie beyond the starting isoline is taken from the first entry in the color stop table (at `offset` = 0) . The padding color for pixels that lie beyond the ending isoline is taken from the last entry in the color stop table (at `offset` = 1.0).

The SPREAD_REPEAT and SPREAD_REFLECT spread methods use the fractional part of gradient *t* to look up colors in the color stop table. The SPREAD_REFLECT method additionally uses the integer part of *t* to determine whether to repeat (if $\lfloor t \rfloor$ is even) or reflect (if $\lfloor t \rfloor$ is odd) the gradient pattern.

Example

This example uses the `SetLinearGradient` function to imitate the color gradients of sunrise over the ocean. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example21(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGRect bkgd = { 40, 40, 400, 300 };
    SGRect light = { 200, 190, 80, 150 };
```

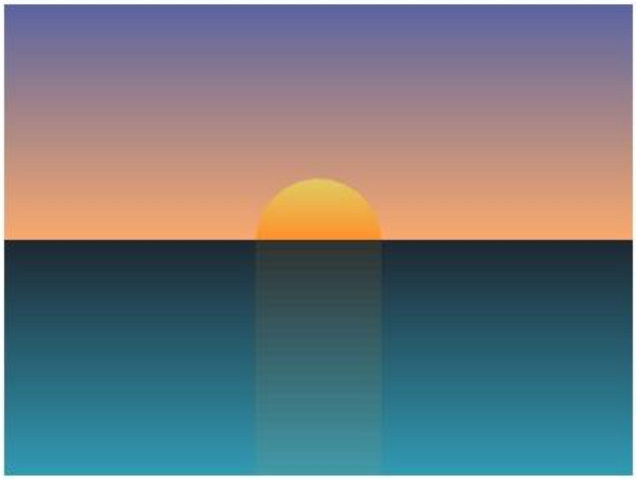
```
    SGPoint v0 = { 240, 191 }, v1 = { 200, 191 }, v2 = { 240, 151 };

    // Fill background with ocean horizon gradient colors
    aarend->AddColorStop(0, RGBX(90,100,160));
    aarend->AddColorStop(0.5, RGBX(250,170,110));
    aarend->AddColorStop(0.5, RGBX(30,40,50));
    aarend->AddColorStop(1.0, RGBX(50,155,180));
    aarend->SetLinearGradient(0,40, 0,340, SPREAD_PAD, 0);
    sg->BeginPath();
    sg->Rectangle(bkgd);
    sg->FillPath();

    // Show sun rising on horizon
    aarend->ResetColorStops();
    aarend->AddColorStop(0, RGBX(225,205,100));
    aarend->AddColorStop(1.0, RGBX(255,145,44));
    aarend->SetLinearGradient(0,150, 0,190, SPREAD_PAD, 0);
    sg->BeginPath();
    sg->EllipticArc(v0, v1, v2, 0, PI);  // PI = 3.14159265...
    sg->FillPath();

    // Show hazy reflection of sun on water
    aarend->ResetColorStops();
    aarend->AddColorStop(0, RGBA(255,160,44,16));
    aarend->AddColorStop(1.0, RGBA(255,160,44,24));
    aarend->SetLinearGradient(0,190, 0,193, SPREAD_REFLECT, FLAG_EXTEND_END);
    sg->BeginPath();
    sg->Rectangle(light);
    sg->FillPath();
}
```

In this example, the color gradients are all oriented vertically. In the starting and ending points passed as arguments to each `SetLinearGradient` function call, the two *x*-coordinates are identical, which means that the two *y*-coordinates wholly determine the slope of the gradient.

The color stop table for the background is half sky colors and half ocean colors, with the abrupt color transition at the horizon occurring halfway through the table, at `offset` = 0.5. The first `SetLinearGradient` function call generates the fill colors for the background rectangle, with the gradient starting point set to the top of the rectangle and the ending point to the bottom.

Next, a new two-entry color stop table is constructed to represent the color gradient in the rising sun. The second `SetLinearGradient` function call generates the fill colors for the filled half circle, with the gradient starting point aligned with the top of the top of the circular arc and the ending point aligned with the center of the circle.

Finally, a new two-entry color stop table is constructed to represent the sun's reflection on the water. The RGB values in the two colors stops are identical, but their alpha values are slightly different. These alpha values are small (16 and 24) to make the reflection appear hazy. In the third `SetLinearGradient` function call, the `FLAG_EXTEND_END` flag is set to enable the gradient to extend beyond the region between the starting and ending points. The `spread` argument is set to `SPREAD_REFLECT` to give the reflection an undulating appearance.



The output produced by this code example is shown in the screenshot at left.

Header

renderer.h

See also

EnhancedRenderer::AddColorStop

EnhancedRenderer::ResetColorStops

EnhancedRenderer::SetTransform

EnhancedRenderer::SetConstantAlpha

EnhancedRenderer::SetPattern function

Either version of the SetPattern function creates a tiled-pattern paint generator for the renderer to use for subsequent fill and stroke operations. The new paint generator replaces any previously created paint generator.

Syntax

C++

```
void EnhancedRenderer::SetPattern(
   const COLOR *pattern,
   float u0,
   float v0,
   int w,
   int h,
   int stride,
   int flags
);
```

```
void EnhancedRenderer:: SetPattern(
   ImageReader *imgrdr,
   float u0,
   float v0,
   int w,
   int h,
   int flags
);
```

Parameters

pattern

A pointer to a COLOR array that contains the image to use for tiled pattern fills. This parameter is used only by the first version of the `SetPattern` function listed in the Syntax block above.

imgrdr

A pointer to an `ImageReader` object (see Remarks below) that supplies the image to use for tiled pattern fills. This parameter is used only by the second version of the `SetPattern` function listed in the Syntax block above.

u0

A `float` value that specifies the horizontal offset of the left edge (minimum *u*-coordinate) of the pattern image from the origin of the *u*-*v* coordinate system in pattern space.

v0

A `float` value that specifies the vertical offset of the top edge (minimum *v*-coordinate) of the pattern image from the origin of the *u*-*v* coordinate system in pattern space.

w

An `int` value that specifies the width (in pixels) of the pattern image.

h

An `int` value that specifies the height (in pixels) of the pattern image.

stride

An `int` value that specifies the stride (in pixels) from the start of one row to the start of the next row in the image array pointed to by the `pattern` parameter. If successive rows in the image are separated in memory by padding (unused storage for one or more COLOR values), then `stride` > `w`. But if the rows are contiguous in memory, then `stride` = `w`. This parameter is used only by the first version of the `SetPattern` function listed in the Syntax block above.

flags

The following flag bits are defined for the `flags` parameter:

- FLAG_IMAGE_BOTTOMUP – The rows in the pattern image are stored in bottom-up rather than top-down order.
- FLAG_IMAGE_BGRA32 – The pixels in the pattern image use BGRA32 format rather than RGBA32 format.
- FLAG_PREMULTALPHA – The pixels in the pattern image are already in premultiplied-alpha format.

These flags are used by both versions of the SetPattern function listed in the Syntax block above.

Return value

None

Remarks

The two versions of the SetPattern function listed in the Syntax block above are similar but differ in how the caller supplies the pattern image to the tiled-pattern object. In both versions, the image consists of 32-bit pixel values.

The first version supplies a pattern image in the form of an array of pixel values. Such an array is convenient for specifying simple patterns – for example, a checkerboard pattern specified by a four-pixel array. The pattern parameter points to an array of pixels, each of which is a COLOR (unsigned 32-bit integer) value. By default, each pixel is assumed to be in RGBA32 format, but the FLAG_IMAGE_BGRA32 flag bit can be set to indicate that the pixels are in BGRA32 format, as will be described shortly.

The second version of the SetPattern function receives a pointer to an ImageReader object, which supplies the pattern image as a stream of pixel values. This version is convenient for reading a pattern image from an image file – for example, a BMP, PNG, or JPEG file. The SetPattern function requires the ImageReader object to support the following interface (see the renderer.h header file):

```
class ImageReader
{
public:
    virtual int ReadPixels(COLOR *buffer, int count) = 0;
};
```

Each successive ReadPixels function call copies the number of pixels specified by the count parameter to the location pointed to by the buffer parameter. The function returns the number of pixels copied. When the supply of pixels is exhausted, the function continues to return zero. Two example implementations of the ImageReader class can be found in the source code for the ShapeGen demo program.

Three flag bits are defined for the flags parameter. The FLAG_IMAGE_BOTTOMUP flag indicates that the rows of the pattern image are stored in bottom-up order. By default, the SetPattern function expects images to be stored in top-down order. For example, images in BMP files are usually stored in bottom-up order and would appear upside down if used without the FLAG_IMAGE_BOTTOMUP flag.

The FLAG_IMAGE_BGRA32 flag indicates that the 32-bit pixels in the pattern image are in BGRA32 format. By default, the SetPattern function expects the pixels to be in RGBA32 format. BGRA32 format is similar to RGBA32 format, except that the 8-bit red and blue fields are swapped, as shown in the following figure:

The SetPattern function requires its internal copy of the source image to be in *premultiplied-alpha format*, which means that each pixel's red, green, and blue color components have been multiplied by the pixel's alpha value. By default, the SetPattern function converts the pixels in the image to premultiplied-alpha format immediately after copying them. However, the caller can set the FLAG_PREMULTALPHA flag to indicate that the pixels in the input image are already in premultiplied-alpha format (so that no conversion is required).

Tiled-pattern fill operations use the renderer's current constant alpha, which is set by the EnhancedRenderer::SetConstantAlpha function.

Tiled-pattern fill operations use the renderer's affine transformation matrix (set by the EnhancedRenderer::SetTransform function). A tiled-pattern paint generator takes a snapshot of the matrix during the SetPattern function call and then continues to use this matrix for pattern-fill operations until the paint generator is replaced.

Example

This example uses the SetPattern function to fill an ellipse with a tartan pattern. (Parameter bkbuf is a PIXEL_BUFFER structure that contains a description of the backing buffer for the graphics display, and parameter clip specifies the device clipping rectangle.)

```
void example22(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    COLOR yel = RGBX(231,213,168), gry = RGBX(163,147,128),
          red = RGBX(211,76,73), mar = RGBX(146,74,77),
          blu = RGBX(79,78,88);
    COLOR tartan[7*7] = {
        yel, gry, gry, yel, gry, gry, gry,
        gry, red, red, gry, mar, mar, mar,
        gry, red, red, gry, mar, mar, mar,
        yel, gry, gry, yel, gry, gry, gry,
        gry, blu, blu, gry, blu, blu, blu,
        gry, blu, blu, gry, blu, blu, blu,
        gry, blu, blu, gry, blu, blu, blu,
    };
    SGPoint v0 = { 260, 194 }, v1 = { 40, 194 }, v2 = { 260, 40 };
    float xform[6] = { 0.055, 0.055, -0.055, 0.055, 0, 0 };

    aarend->SetTransform(xform);
    aarend->SetPattern(tartan, 0,0, 7,7,7, 0);
    sg->BeginPath();
    sg->Ellipse(v0, v1, v2);
    sg->FillPath();
}
```

In this example, the source image for the pattern is contained in a 49-element array consisting of 32-bit pixels. The image is 7 pixels wide and 7 pixels high.

The transform matrix, as specified by the `xform` array, scales up the image by a factor of 13 and then rotates the image by 45 degrees. This 2-D affine transformation can be factored into two 3x3 matrices, as follows:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} 0.71 & 0.71 & 0 \\ -0.71 & 0.71 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/13 & 0 & 0 \\ 0 & 1/13 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

When the two 3x3 matrices above are multiplied together, the result is the `xform` array in the code example. As explained on the `EnhancedRenderer::SetTransform` reference page, the last row of a 3x3 transform matrix is always [0 0 1] and does not need to be explicitly included in the `xform` array.

The output produced by this code example is shown in the screenshot at left.

Header

`renderer.h`

See also

`EnhancedRenderer::SetConstantAlpha`

`EnhancedRenderer::SetTransform`

EnhancedRenderer::SetRadialGradient function

The `SetRadialGradient` function creates a radial-gradient paint generator for the renderer to use for subsequent fill and stroke operations. The new paint generator replaces any previously created paint generator.

Syntax

C++

```
void EnhancedRenderer:: SetRadialGradient(
   float x0,
   float y0,
   float r0,
   float x1,
   float y1,
   float r1,
   SPREAD_METHOD spread,
   int flags
);
```

Parameters

x0

A `float` value that specifies the *x*-coordinate at the center of the starting circle.

y0

A `float` value that specifies the *y*-coordinate at the center of the starting circle.

r0

A `float` value that specifies the radius of the starting circle.

x1

A `float` value that specifies the *x*-coordinate at the center of the ending circle.

y1

A `float` value that specifies the *y*-coordinate at the center of the ending circle.

r1

A `float` value that specifies the radius of the ending circle.

spread

A `SPREAD_METHOD enum` value that specifies whether the linear gradient is to spread according to the *repeat*, *reflect*, or *pad* method. The following `enum` values are defined for this parameter:

- `SPREAD_PAD` – Use the padding colors to fill pixels that lie outside the region between the starting and ending circles.
- `SPREAD_REPEAT` – Repeat the gradient pattern to fill pixels that lie outside the region between the starting and ending circles.

- SPREAD_REFLECT – Alternately repeat and reflect the gradient pattern to fill pixels that lie outside the region between the starting and ending circles.

flags

The following flag bits are defined for the flags parameter:

- FLAG_EXTEND_START – Extend the gradient fill beyond the starting circle.
- FLAG_EXTEND_END – Extend the gradient fill beyond the ending circle.

If neither of these flags is set, the gradient fill is confined to the region between the starting circle and ending circle. In this case, pixels outside this region are not filled, and, thus, the spread parameter has no effect.

Return value

None

Remarks

The color intervals in the radial-gradient fill pattern are specified in the renderer's color stop table. A radial-gradient paint generator takes a snapshot of the current color stop table during the SetRadialGradient function call and then continues to use these color stops for gradient-fill operations until the paint generator is replaced. For more information, see EnhancedRenderer::AddColorStop and EnhancedRenderer::ResetColorStops.

Radial-gradient fill operations use the renderer's current constant alpha, which is set by the EnhancedRenderer::SetConstantAlpha function.

Radial-gradient fill operations use the renderer's affine transformation matrix (set by the EnhancedRenderer::SetTransform function). A radial-gradient paint generator takes a snapshot of the matrix during the SetRadialGradient function call and then continues to use this matrix for radial-gradient fill operations until the paint generator is replaced.

Internally, the radial-gradient paint generator uses a parameter *t* to designate the gradient at a point in the gradient pattern. Parameter *t* is 0 at the edge of the starting circle and is 1.0 at the edge of the ending circle. Either radius – r0 or r1 – can be zero, in which case the starting or ending circle shrinks to a point. The two radii cannot both be zero.

Along a line connecting corresponding points on the two circles, gradient *t* increases from 0 to 1.0, and the color stop table specifies the color changes over this interval.

Outside this interval, the spread and flags parameters specify the gradient fill behavior. If the FLAG_EXTEND_START flag is set, the gradient fill extends beyond the starting circle, into the region for which $t < 0$. If the FLAG_EXTEND_END flag is set, the gradient fill extends beyond the ending circle, into the region for which $t > 1.0$.

The SPREAD_PAD spread method uses solid colors to pad areas outside the region between the starting and ending circle. The padding color for pixels that lie beyond the starting circle is taken from the first entry in the color stop table (at offset = 0) . The padding color for pixels that lie beyond the ending circle is taken from the last entry in the color stop table (at offset = 1.0).

The SPREAD_REPEAT and SPREAD_REFLECT spread methods use the fractional part of gradient *t* to look up colors in the color stop table. The SPREAD_REFLECT method additionally uses the integer part of *t* to determine whether to repeat (if ⌊*t*⌋ is even) or reflect (if ⌊*t*⌋ is odd) the gradient pattern.

The two EXTEND flags defined for the `flags` parameter are fashioned after the Extend array defined for Type 3 (Radial) Shadings in the PDF specification*. The radial gradients defined in the SVG 2 standard encompass the subset of this functionality for which, in effect, the two EXTEND flags are always set.

Example

This example uses the `SetRadialGradient` function to fill a circle with a gradient that ranges in color from light yellow to dark red. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example23(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    SGPoint v0 = { 200, 200 }, v1 = { 200-162, 200 }, v2 = { 200, 200-162 };

    aarend->AddColorStop(0, RGBX(255,255,224));  // light yellow
    aarend->AddColorStop(1.0, RGBX(139,0,0));  // dark red
    aarend->SetRadialGradient(140,140,19, 182,182,188, SPREAD_PAD, FLAG_EXTEND_START);
    sg->BeginPath();
    sg->Ellipse(v0, v1, v2);
    sg->FillPath();
}
```

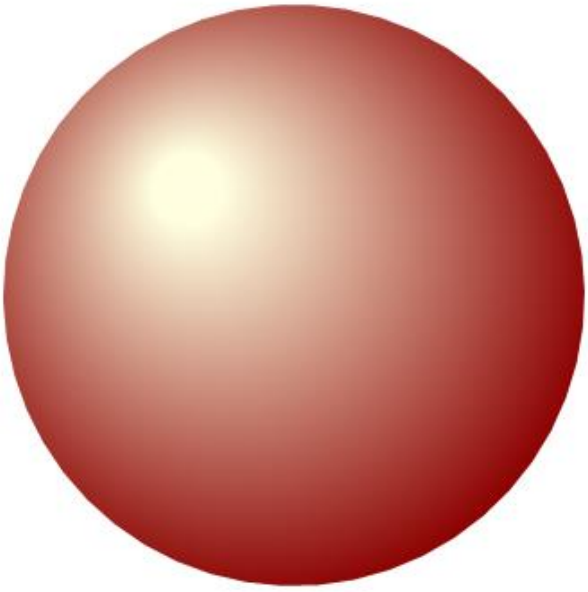
The goal in this example is to give the radial gradient an appearance similar to that of a glossy red sphere reflecting a yellowish light source.

The color stop table starts with light yellow (at `offset` = 0) and ends with dark red (at `offset` = 1.0). The ending circle for the radial gradient has a radius that's just a bit larger than that of the circular shape that's being filled, and its center is located 20 pixels to the left and above the shape's center. The effect is to lighten the top-left edge of the circular shape by shaving off the darkest part of the gradient there.

The starting circle for the radial gradient has a radius of 19 pixels and is padded with light yellow to give the appearance of a reflection from a light source. The center of the starting circle is offset from that of the ending circle by 42 pixels in both *x* and *y*.

The output produced by this code example is shown in the following screenshot.

* See section 8.7.4.5.4 of the PDF specification (ISO 32000-1 standard, first edition, 2008) at https://opensource.adobe.com/dc-acrobat-sdk-docs/standards/pdfstandards/pdf/PDF32000_2008.pdf.



Header

renderer.h

See also

EnhancedRenderer::AddColorStop

EnhancedRenderer::ResetColorStops

EnhancedRenderer::SetConstantAlpha

EnhancedRenderer::SetTransform

EnhancedRenderer::SetTransform function

The SetTransform function sets the affine transform matrix that the renderer will use for subsequent fill and stroke operations. Both pattern fills and gradient fills are transformed by this matrix.

Syntax

C++

```
void EnhancedRenderer::SetTransform(const float xform[]);
```

Parameters

xform

A six-element float array that specifies the affine transformation to use for subsequent pattern and gradient fill operations. Setting xform to zero (null array pointer) indicates that no transformation is to be performed on fill patterns or gradients (equivalent to specifying the identity matrix).

Return value

None

Remarks

The renderer's affine transform matrix specifies the mapping of the *x*-*y* coordinates of a pixel on the screen to the corresponding *u*-*v* coordinates in pattern (or texture) space. The renderer uses this mapping to copy the color of the pattern or gradient at coordinates (*u*,*v*) to the pixel at coordinates (*x*,*y*).

If the `xform` array is specified as

```
float xform[6] = { a, b, c, d, e, f };
```

then the resulting affine transformation of a point (*x*,*y*) on the display to a point (*u*,v) in pattern space is as follows:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The bottom row of the matrix is implicitly always [0 0 1] and does not need to be explicitly included in the `xform` array supplied to the `SetTransform` function. The format of the `xform` array is the same as that of the 2-D affine transform array defined in the SVG 2 standard.

To apply an affine transformation to a tiled pattern, linear gradient, or radial gradient, call `SetTransform` to set the transformation matrix *before* calling one of the following:

- `EnhancedRenderer::SetPattern`
- `EnhancedRenderer::SetLinearGradient`
- `EnhancedRenderer::SetRadialGradient`

Immediately after an `EnhancedRenderer` object is created, the renderer's affine transform matrix has been set to its default value, which is the identity matrix (that is, no transformation).

Example 1

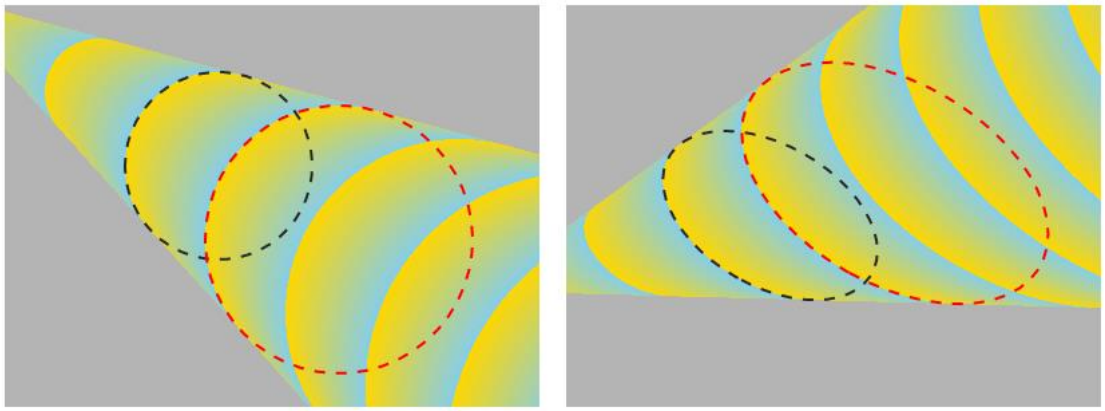
This first code example shows how to use the `SetTransform` function to apply a 2-D affine transformation to a radial gradient. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example24(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    float xform[6] = { 0.375, -0.650, 1.083, 0.625, 364.3, 227.1 };
    SGPoint sc[3] = { { 200, 160 }, { 200-70, 160 }, { 200, 160-70 } };
    SGPoint ec[3] = { { 290, 215 }, { 290-100, 215 }, { 290, 215-100 } };
    SGRect rect = { 40, 40, 400, 300 };
    char dash[] = { 1, 0 };

    // Fill the left rectangle with background color gray
    aarend->SetColor(RGBX(180,180,180));
    sg->BeginPath();
    sg->Rectangle(rect);
    sg->SetFillRule(FILLRULE_WINDING);
```

```
    sg->FillPath();

    // Fill the left rectangle with a radial gradient
    aarend->AddColorStop(0, RGBX(255, 215, 0));  // gold
    aarend->AddColorStop(1.0, RGBX(135, 206, 235));  // skyblue
    aarend->SetRadialGradient(200,160,70, 290,215,100, SPREAD_REPEAT,
                              FLAG_EXTEND_START | FLAG_EXTEND_END);
    sg->FillPath();

    // Stroke the outline of the starting circle in black
    aarend->SetColor(RGBX(40,40,40));
    sg->SetLineDash(dash, 0, 8.07);
    sg->SetLineWidth(2);
    sg->BeginPath();
    sg->Ellipse(sc[0], sc[1], sc[2]);
    sg->StrokePath();

    // Stroke the outline of the ending circle in red
    aarend->SetColor(RGBX(255,0,0));
    sg->BeginPath();
    sg->Ellipse(ec[0], ec[1], ec[2]);
    sg->StrokePath();

    // Fill the right rectangle with background color gray
    rect.x += 420;
    aarend->SetColor(RGBX(180,180,180));
    sg->BeginPath();
    sg->Rectangle(rect);
    sg->SetFillRule(FILLRULE_WINDING);
    sg->FillPath();

    // Set up a new transform for the radial gradient
    aarend->SetTransform(xform);

    // Fill the right rectangle with the transformed radial gradient
    aarend->SetRadialGradient(200,160,70, 290,215,100, SPREAD_REPEAT,
                              FLAG_EXTEND_START | FLAG_EXTEND_END);
    sg->FillPath();

    // Transform the coordinates of the starting and ending circles.
    // To improve resolution, convert the transformed coordinates to
    // 16.16 fixed-point format.
    for (int i = 0; i < 3; ++i)
    {
        float xtmp = 65536*(xform[0]*sc[i].x + xform[2]*sc[i].y + xform[4]);
        sc[i].y = 65536*(xform[1]*sc[i].x + xform[3]*sc[i].y + xform[5]);
        sc[i].x = xtmp;

        xtmp = 65536*(xform[0]*ec[i].x + xform[2]*ec[i].y + xform[4]);
        ec[i].y = 65536*(xform[1]*ec[i].x + xform[3]*ec[i].y + xform[5]);
        ec[i].x = xtmp;
    }

    // Tell ShapeGen the coordinates are in 16.16 fixed-point format
    sg->SetFixedBits(16);

    // Outline the transformed starting circle in black
    aarend->SetColor(RGBX(40,40,40));
    sg->BeginPath();
    sg->Ellipse(sc[0], sc[1], sc[2]);
    sg->StrokePath();
```

```
    // Outline the transformed ending circle in red
    aarend->SetColor(RGBX(255,0,0));
    sg->BeginPath();
    sg->Ellipse(ec[0], ec[1], ec[2]);
    sg->StrokePath();
}
```

The output produced by this code example is shown in the following screenshot.



The rectangle on the left is filled with the original, untransformed radial gradient, after which the starting and ending circles that define this gradient are outlined by black and red dashed lines, respectively.

At the start of the code example, the `xform` array defines the 2-D affine transformation that is to be applied to the radial gradient. The transformed gradient will be used to fill the rectangle on the right side of the screenshot.

To transform the radial gradient, the `xform` array is first passed to the `SetTransform` function. The `SetTransform` call is followed by an `EnhancedRenderer::SetRadialGradient` function call, which loads the transformation matrix and radial gradient parameters into the radial-gradient paint generator. The `ShapeGen::FillPath` function call then fills the rectangle on the right with the transformed radial gradient.

To verify that the radial gradient has been precisely transformed, this same transformation is applied to the coordinates in the `sc` and `ec` arrays, which specify the starting and ending circles. The transformed circles, which are now ellipses, are stroked with black and red dashed lines over the transformed gradient pattern on the right side of the screenshot. Applying the transformation to the circle coordinates is straightforward – the required matrix multiplications are performed in the `for`-loop in the code example.

However, applying the transformation to the radial gradient parameters is a bit tricky. By handing off the transformation matrix to the `SetTransform` function, the programmer can ignore all the messy details.

The `xform` array in this example embodies the following transformation:

1. Translate the original, untransformed radial gradient from center coordinates (240, 190) of the rectangle on the left to the *x*-*y* origin.
2. To squash the circles into ellipses, scale the figure by a factor of 0.75 in the *x* dimension and by 1.25 in the *y* dimension.
3. Rotate the figure by π/3 radians in the counterclockwise direction.

4. Translate the figure from the *x-y* origin to center coordinates (660, 190) of the rectangle on the right.

The steps in this transformation are summarized in the following matrix equation:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\frac{\pi}{3} & \sin\frac{\pi}{3} & 660 \\ -\sin\frac{\pi}{3} & \cos\frac{\pi}{3} & 190 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.75 & 0 & 0 \\ 0 & 1.25 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -240 \\ 0 & 1 & -190 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

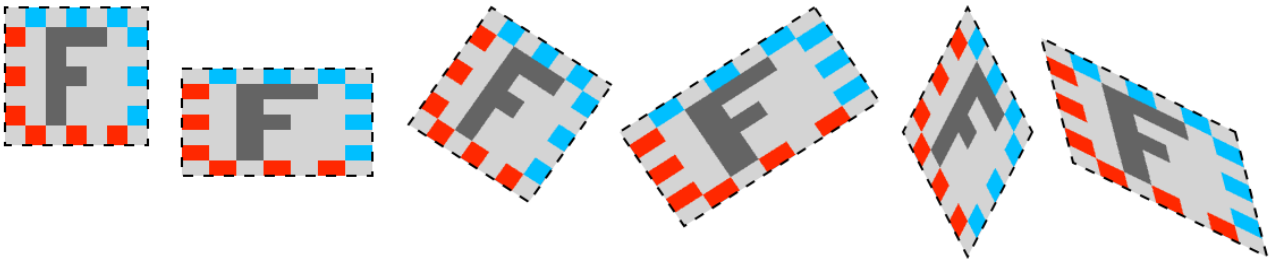
Example 2

This next code example shows how to use the `SetTransform` function to map a rectangular image to a target shape that is a square, rectangle, or parallelogram. (Parameter `bkbuf` is a `PIXEL_BUFFER` structure that contains a description of the backing buffer for the graphics display, and parameter `clip` specifies the device clipping rectangle.)

```
void example25(const PIXEL_BUFFER& bkbuf, const SGRect& clip)
{
    SmartPtr<EnhancedRenderer> aarend(CreateEnhancedRenderer(&bkbuf));
    SmartPtr<ShapeGen> sg(CreateShapeGen(&(*aarend), clip));
    int w = 7, h = 7;  // width, height of test image
    COLOR c0 = RGBX(212,212,212), c1 = RGBX(255,40,0),
          c2 = RGBX(0,191,255), c3 = RGBX(100,100,100);
    COLOR image[] = {
        c0,c2,c0,c2,c0,c2,c0,  // a 7x7 test image
        c1,c0,c3,c3,c3,c0,c2,
        c0,c0,c3,c0,c0,c0,c0,
        c1,c0,c3,c3,c0,c0,c2,
        c0,c0,c3,c0,c0,c0,c0,
        c1,c0,c3,c0,c0,c0,c2,
        c0,c1,c0,c1,c0,c1,c0,
    };
    SGPoint vert[][4] = {  // 3 of 4 vertices
        { {  155,   29 }, {   29,   29 }, {   29,  155 }, },
        { {  353,   85 }, {  185,   85 }, {  185,  183 }, },
        { {  564,  100 }, {  458,   29 }, {  385,  136 }, },
        { {  743,   29 }, {  571,  143 }, {  628,  229 }, },
        { {  935,  143 }, {  878,   29 }, {  821,  143 }, },
        { { 1114,  143 }, {  943,   57 }, {  971,  171 }, },
    };
    char dash[] = { 9, 0 };

    sg->SetLineWidth(2.0);
    sg->SetLineJoin(LINEJOIN_MITER);
    sg->SetLineDash(dash, 3, 1.0);
    for (int i = 0; i < ARRAY_LEN(vert); ++i)
    {
        // Use symmetry to calculate the 4th vertex of the square,
        // rectangle, or parallelogram that is to be filled
        float x0 = vert[i][0].x, y0 = vert[i][0].y;
        float x1 = vert[i][1].x, y1 = vert[i][1].y;
        float x2 = vert[i][2].x, y2 = vert[i][2].y;

        vert[i][3].x = x0 - x1 + x2;
        vert[i][3].y = y0 - y1 + y2;

        // Construct the matrix that will transform points in the
```

```
            // square, rectangle, or parallelogram to the test image
            float xP = x0 - x1, yP = y0 - y1;
            float xQ = x2 - x1, yQ = y2 - y1;
            float det = xP*yQ - xQ*yP;
            float wdet = w/det, hdet = h/det;
            float xform[6];

            xform[0] = wdet*yQ, xform[2] = -wdet*xQ;
            xform[1] = -hdet*yP, xform[3] = hdet*xP;
            xform[4] = wdet*(xQ*y1 - yQ*x1), xform[5] = hdet*(yP*x1 - xP*y1);
            aarend->SetTransform(xform);

            // Map the test image onto the square, rectangle, or
            // parallelogram that is to be filled
            sg->BeginPath();
            sg->Move(vert[i][0].x, vert[i][0].y);
            sg->PolyLine(&vert[i][1], 3);
            sg->CloseFigure();
            aarend->SetPattern(image, 0,0, w,h,w, 0);
            sg->FillPath();

            // Outline the square, rectangle, or parallelogram with a
            // black dashed line
            aarend->SetColor(RGBX(0,0,0));
            sg->StrokePath();
        }
    }
```

The output produced by this code example is shown in the following screenshot.



The `image` array, which is defined near the start of the code example, contains a simple 7-by-7 test image.

The initializer for the `vert` array defines the first three vertices for each shape in a series of squares, rectangles, and parallelograms oriented at various angles. The fourth vertex for each of these shapes is later calculated using symmetry.

Each iteration of the `for`-loop in the code example calls the `EnhancedRenderer::SetPattern` function to specify the mapping of the test image to the next shape (square, rectangle, or parallelogram) in the `vert` array. In the code that precedes this call, a 2-D affine transformation matrix is constructed and loaded into the `xform` array. This matrix specifies the transformation from the *x*-*y* coordinates of a pixel in the shape to the corresponding u-v coordinates in the image. Later, during the pattern-fill operation, the image will be sampled at these *u*-*v* coordinates to determine the color value that is to be written to the pixel.

This transformation matrix is constructed from the first three vertices – (x_0, y_0), (x_1, y_1), (x_2, y_2) – of the shape and then scaled to the width w and height h of the test image. The transformation is described by the following matrix equation:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{x_P y_Q - x_Q y_P} \begin{bmatrix} w & 0 & 0 \\ 0 & h & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_Q & -x_Q & 0 \\ -y_P & x_P & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The rightmost 3x3 matrix above translates the vertex at (x_1, y_1) to the *x*-*y* origin and translates the first and third vertices to coordinates $(x_P, y_P) = (x_0\text{-}x_1, y_0\text{-}y_1)$ and $(x_Q, y_Q) = (x_2\text{-}x_1, y_2\text{-}y_1)$.

Before each `SetPattern` call, the `SetTransform` function is called to specify the new transformation matrix. The `SetPattern` function then loads the test image and transformation matrix into the pattern-fill paint generator. Finally, during the `ShapeGen::FillPath` function call, the pattern-fill paint generator fills the shape with the transformed test image.

To verify that the test image has been mapped precisely to the designated shape in the `vert` array, the `ShapeGen::StrokePath` function call outlines the shape boundary with a black dashed line.

Header

`renderer.h`

See also

`EnhancedRenderer::SetPattern`

`EnhancedRenderer::SetLinearGradient`

`EnhancedRenderer::SetRadialGradient`

Appendices

Appendix A: Simple renderer code examples

This GitHub project's `renderer.cpp` file implements both a simple renderer and an enhanced renderer, as explained in the Renderer section. These two example renderers can cooperatively render an image in a back buffer that has a 32-bit BGRA pixel format (that is `0xaarrggbb`).

However, an enhanced renderer works exclusively with true-color graphics display devices. For devices that have more limited color capabilities, a standalone simple renderer is sufficient.

To draw a shape, the ShapeGen object embeds the description of the shape in a `ShapeFeeder` object. The shape feeder is then passed to the renderer's `RenderShape` function, which performs all rendering.

To support a simple renderer, the shape feeder breaks a shape into a series of rectangles with vertical and horizontal sides. Rectangle coordinates are specified as simple integers. For shapes with sharply curved or slanted sides, most of these rectangles might be just one or two pixels in height. However, a large, rectangular fill region can be specified as a single rectangle.

A simple renderer provides a `SetColor` function that a ShapeGen-based application calls to specify the color to use for filling the next shape. The application specifies the color as an RGB value, and the `SetColor` function maps this value as closely as possible to the pixel format in which the shape is to be rendered.

A simple way to design a standalone simple renderer that will handle an arbitrary pixel format is to allow the underlying platform do the required color mapping and rectangle filling.

This appendix provides complete C++ code examples for standalone simple renderers that run on the SDL2 (Simple DirectMedia Library, version 2) and Win32/GDI APIs. Modifying these examples to run on other platforms should be straightforward.

SDL2 API code example

The following C++ code example is the complete implementation of a simple renderer that runs on the SDL2 API (for example, in Linux or Windows):

```
//------------------------------------------------------------------------
//
// BasicRenderer class: Fills a shape with a solid color but does _NOT_
// do antialiasing. This class is derived from the SimpleRenderer class
// in demo.h, which, in turn, is derived from the Renderer base class
// in shapegen.h. This BasicRenderer class implementation is platform-
// dependent -- it uses the SDL_FillRect function in SDL2 to fill the
// horizontal spans that comprise the shape.
//
//------------------------------------------------------------------------

class BasicRenderer : public SimpleRenderer
{
    friend ShapeGen;

    SDL_Surface* _surface;
    Uint32 _pixel;

protected:
    void RenderShape(ShapeFeeder *feeder);

public:
    BasicRenderer(SDL_Surface* screenSurface) : _surface(screenSurface), _pixel(0)
    {
        SetColor(RGBX(0,0,0));
    }
    ~BasicRenderer()
    {
    }
    void SetColor(COLOR color);
};

// Fills a series of horizontal spans that comprise a shape
void BasicRenderer::RenderShape(ShapeFeeder *feeder)
{
    SDL_Rect rect;

    while (feeder->GetNextSDLRect(reinterpret_cast<SGRect*>(&rect)))
        SDL_FillRect(_surface, &rect, _pixel);
}

// Selects the color to be used for subsequent fills and strokes
void BasicRenderer::SetColor(COLOR color)
{
    int r = color & 0xff;
    int g = (color >> 8) & 0xff;
    int b = (color >> 16) & 0xff;
```

```
    _pixel = SDL_MapRGB(_surface->format, r, g, b);
}
```

This renderer paints shapes with solid colors and performs no antialiasing. The `BasicRenderer` constructor receives, as an input parameter, the back buffer pointer that the calling program obtained from the `SDL_GetWindowSurface` function. The `RenderShape` member function performs all of the rendering. It iteratively calls the shape feeder's `GetNextSDLRect` function to receive the next rectangle (an `SDL_Rect` structure) and then calls the `SDL_FillRect` function to draw the filled rectangle into the back buffer. The `SetColor` member function calls the `SDL_MapRGB` function to map the caller's RGB color parameter to the nearest color value in the pixel format of the back buffer The COLOR type and RGBX macro are defined in the `renderer.h` file in the ShapeGen project directory. After the rendered image in the back buffer is complete, the calling program calls the `SDL_UpdateWindowSurface` function to BitBlt the back buffer to the window.

Win32/GDI API code example

The following C++ code example is the complete implementation of a simple renderer that runs on the Win32/GDI API in Windows:

```
//-----------------------------------------------------------------------
//
// BasicRenderer class: Fills a shape with a solid color but does _NOT_
// do antialiasing. This class is derived from the SimpleRenderer class
// in demo.h, which, in turn, is derived from the Renderer base class
// in shapegen.h. This BasicRenderer class implementation is platform-
// dependent -- it uses the FillRect function in Windows GDI to fill
// the horizontal spans that comprise the shape.
//
//-----------------------------------------------------------------------

class BasicRenderer : public SimpleRenderer
{
    friend ShapeGen;

    HDC _hdc;
    HBRUSH _hBrush;

protected:
    void RenderShape(ShapeFeeder *feeder);

public:
    BasicRenderer(HDC hdc) : _hdc(hdc), _hBrush(0)
    {
        SetColor(RGBX(0,0,0));
    }
    ~BasicRenderer()
    {
        DeleteObject(_hBrush);
    }
    void SetColor(COLOR color);
};

// Fills a series of horizontal spans that comprise a shape
void BasicRenderer::RenderShape(ShapeFeeder *feeder)
{
    RECT rect;

    while (feeder->GetNextGDIRect(reinterpret_cast<SGRect*>(&rect)))
        FillRect(_hdc, &rect, _hBrush);
```

```
}

// Selects the color to be used for subsequent fills and strokes
void BasicRenderer::SetColor(COLOR color)
{
    DeleteObject(_hBrush);
    _hBrush = CreateSolidBrush(color & 0x00ffffff);
    SelectObject(_hdc, _hBrush);
}
```

Unlike the SDL2 implementation previously discussed, this renderer can draw directly to a window in on-screen memory – *if* the input parameter is the handle to a display device context for the window. The calling program obtains this handle from the Win32/GDI `BeginPaint` function. Drawing directly to the window is slower than using a back buffer but is useful for debugging graphics software. By stepping through the rendering code on a debugger, you can watch as each shape is constructed in piecemeal fashion. When debugging is complete, the code can be sped up by rendering a complete image to a back buffer that is then copied to the window in a single Win32/GDI `BitBlt` operation.

The `RenderShape` member function performs all of the rendering. It iteratively calls the shape feeder's `GetNextGDIRect` function to receive the next rectangle (a Win32/GDI RECT structure) and then calls the Win32/GDI `FillRect` function to draw the filled rectangle into the back buffer. The `SetColor` member function calls the Win32/GDI `CreateSolidBrush` function to map the caller's RGB color parameter to the nearest color value in the target buffer's pixel format.

Appendix B: SVG files for the ShapeGen-based svgview app

This GitHub project includes the source code for the `svgview` example app, which is a ShapeGen-based SVG file viewer. For build instructions, see Building the demo and svgview example apps.

The `svgview` app incorporates a modified version of M. Mononen's single-header-file SVG parser, `nanosvg.h`. This simple parser supports a limited subset of the W3C SVG2 standard. This subset does not support, for example, text or Gaussian blur, and has only rudimentary support for `defs` elements. In spite of these limitations, the parser can handle a surprisingly large number of SVG files, including those listed later in this appendix. If an SVG file contains unsupported features, the parser is robust in skipping over the elements that it cannot display while displaying the ones that it can.

To test the `svgview` app on Linux, open a terminal window and enter the list of SVG files you wish to view. For example, to view three files named bozo.svg, tiger.svg, and heart.svg, enter the following command:

```
./svgview bozo.svg tiger.svg heart.svg
```

This example assumes that the three SVG files are located in the same directory as the `svgview` executable. Similarly, to view the files on Windows, open a command window and enter the same command (but without the "`./`"). While the `svgview` app is running, press the space key to step through the SVG files in sequence, one at a time.

This project does not include any SVG files, but many such files are available on the Web. To help you get started, following tables contain links to selected SVG files on the following websites: w3.org, wikimedia.org, openclipart.org, and freesvg.org.

First, the following page on the w3.org site contains links to dozens of SVG files:

https://dev.w3.org/SVG/tools/svgweb/samples/svg-files/

The following table contains a list of SVG files selected from this page. You can use these files to compare the graphics output of the `svgview` app to that produced by your favorite browser. These SVG files vary in size from 192 bytes to 329K bytes.

Description	File size	File name
Bozo clown face	268 bytes	`bozo.svg`
Ford Focus	329K bytes	`compuserver_msn_Ford_Focus.svg`
Four glyphs	797 bytes	`dojo.svg`
Heart symbol	192 bytes	`heart.svg`
Question mark	277 bytes	`whatwg.svg`
Staircase	4K bytes	`penrose-staircase.svg`
Ghostscript tiger	97K bytes	`tiger.svg`

Another source for SVG files is the Wikimedia Commons website at

https://commons.wikimedia.org/wiki/Category:SVG_files

The following table contains a list of SVG files selected from Wikimedia Commons to try out on the svgview app:

Description	File size	File name
BMW logo	12K bytes	`BMW.svg`
Coca-Cola logo	11K bytes	`Coca-Cola_logo.svg`
F15 Eagle jet	950K bytes	`McDonnell_Douglas_F-15_Eagle_cutaway_drawing.svg`
Five-point star	1.3K bytes	`Golden_star.svg`
Treble clef	5K bytes	`Treble_clef.svg`

The following table lists some SVG files that are available on the openclipart.org website. This site is described in the Wikipedia article at https://en.wikipedia.org/wiki/Openclipart.

Description	File size	Download page
Cartoon parrot	432K bytes	`Parrot (#2)`
Log cabin	123K bytes	`Cabin Hebrews 3 4 Remix`
Macaw in flight	201K bytes	`Macaw`

The following table lists some SVG files that are available on the freesvg.com website. This site is briefly mentioned in the Wikipedia article at https://en.wikipedia.org/wiki/Openclipart. Note that some of the SVG files in this table are quite large.

Description	File size	Download page
Cup of espresso	13K bytes	Cup of espresso vector
Floral delight	7M bytes	Floral Delight 2
Flowery paisley	1.2M bytes	Flowery Paisley Design By Paul Brennan Prismatic
Lamborgini	546K bytes	car lamborgini
Shark	49K bytes	Shark (#7)
Summer kimono	1.8M bytes	Summer kimono (#7)
Yellow school bus	1.9M bytes	American School Bus