ShapeGen 2-D Graphics Library User's Guide

Jerry R. VanAken

March 15, 2022

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

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 user of this graphics library calls ShapeGen functions to create arbitrarily complex geometric shapes that are then rendered on a raster display device. The user constructs shape boundaries by connecting geometric primitives such as line segments, spline curves, and circular or elliptic arcs. The library *flattens* the curves and arcs by approximating them with sequences of straight polygonal edges. The resulting shapes are rendered as filled polygons.

ShapeGen additionally supports clipping regions of arbitrary shape and complexity.

The core of the library is the ShapeGen class, which implements a relatively simple but powerful 2-D *polygonal shape generator*.

The polygonal shape generator is the part of a 2-D graphics 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.

The source code for this project includes example renderers that run on Linux and Windows. For either operating system, two example renderers are provided. First, for computers with limited graphics capabilities, a *basic* renderer does simple solid-color fill operations (with no antialiasing). Second, an *enhanced* renderer is provided for use with full-color graphics displays. In addition to solid-color fills, the enhanced renderer does antialiasing, alpha blending, tiled-pattern fills, linear-gradient fills, and radial-gradient fills.

The ShapeGen library is designed to simplify renderer design, and to reduce the amount of platform-specific or device-dependent coding required to port the library. To this end, renderers are *not* responsible for clipping – all shapes that the ShapeGen object passes to the renderer have already been clipped. As a result, only a few lines of code are required to implement a basic renderer.

To reduce the effort required to port enhanced renderers, the library includes *paint generators* that fill shapes with tiled patterns, linear gradients, and radial gradients. During fill operations, these paint generators do most of the heavy lifting for the renderer. And they contain no platform-specific or device-dependent code, although they are, of course, intended for use with full-color graphics displays.

Well-defined interfaces connect the renderer to the library's shape-generator and paint-generator objects.

Programming interface

The programming interface for ShapeGen library users is similar to that of other 2-D graphics software systems, such as PostScript.

The PostScript page description language is the archetypical 2-D graphics interface. For comparison, the following table lists a number of ShapeGen 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.

A lightweight and portable graphics library

The ShapeGen library included in this GitHub project is lightweight and portable. The source code for the ShapeGen class consists of six well-commented C++ source files and two header files, as shown 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 Bezier curves
`edge.cpp`	29K	Manages lists of polygonal edges, does clipping, and drives renderer
`path.cpp`	22K	Performs basic path-management functions
`stroke.cpp`	28K	Converts paths into stroked lines and curves
`thinline.cpp`	8K	Converts paths into *thin* stroked lines and curves
`shapegen.h`	10K	Header file for ShapeGen public interface
`shapepri.h`	15K	Header file for ShapeGen internal interfaces

Also included in this project is example application code, described in the following table, that demonstrates the graphics capabilities of the ShapeGen library.

File name	File size (bytes)	Description
`demo.cpp`	129K	Contains source code for ShapeGen demo program, and for code examples in *ShapeGen User's Guide*
`textapp.cpp`	62K	Implements TextApp graphical text application and provides glyphs for all printing ASCII characters
`bmpfile.cpp`	13K	Rudimentary BMP file reader used by demo program to do pattern fills
`demo.h`	5K	Header file for TextApp interface and demo program parameters

The ShapeGen demo program is designed to run on a full-color graphics display with 24-bit or 32-bit pixels. The demo runs in a 1280x960 window. Every pixel displayed in the demo program – including all geometric shapes and text – is drawn by the ShapeGen graphics library.

This project includes example renderers that run on the Win32 API in Windows, and on SDL2 (Simple DirectMedia Library, version 2) in Linux and Windows. The following table lists the source files for these renderers.

File name	File size (bytes)	Description
`winmain.cpp`	15K	Implements a pair of example renderers to run on Win32 API
`sdlmain.cpp`	13K	Implements a pair of example renderers to run on SDL2 in Linux and Windows

Note that `winmain.cpp` and `sdlmain.cpp` are the only files in this project that contain code that is platform- or device-dependent. All other files have no such dependencies.

Finally, this project also includes paint generators for tiled patterns, linear gradients, and radial gradients. These paint generators simplify the design of enhanced renderers. The following table lists the paint generator source files.

File name	File size (bytes)	Description
`pattern.cpp`	15K	Implements paint generator for tiled-pattern fills
`gradient.cpp`	13K	Implements paint generators for linear-gradient and radial-gradient fills
`renderer.h`	9K	Defines renderer interfaces to ShapeGen library and paint generators

Because nearly all the source code in the ShapeGen library is free of platform or device dependencies, it is easily ported to any computer for which a C++ compiler is available. The ShapeGen source code can be paired with a basic renderer (with no antialiasing) to drive a graphics display on low-cost or compact hardware.

Additionally, ShapeGen can take advantage of computers with full-color displays to do antialiasing, alpha blending, pattern fills, linear gradient fills, and radial gradient fills.

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 easier to modify and build on. 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.

Last but not least, 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.

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. For example, 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.

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. *Stroked* paths are always constructed using the nonzero winding number rule.

ShapeGen's scan conversion rules are different from PostScript's. First, if the ShapeGen library is connected to a basic renderer (with no antialiasing), 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. (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.)

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.

Device clipping rectangle

ShapeGen *x-y* coordinates map directly to pixels on the graphics display. The boundaries between pixels always fall on integer *x-y* coordinates; *x* coordinate values increase to the right, and *y* coordinate values increase in the downward direction.

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

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.

Typically, 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 page). ShapeGen's automatic clipping prevents drawing from occurring outside the window.

The viewer 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 alone is not 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 defined at all times. The ShapeGen class constructor requires a device clipping rectangle as an input parameter, so that an initial clipping region can be set 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.

Creating a ShapeGen object

The ShapeGen programming interface is defined in public header file `shapegen.h`. The internal implementation of the ShapeGen interface is encapsulated by the SGPtr class, which is defined at the bottom of this header file. An instance of the SGPtr class functions as a smart pointer that first creates a ShapeGen object and then provides the user with access to this object's public interface. When the SGPtr object goes out of scope and is automatically deleted, the SGPtr destructor deletes the ShapeGen object.

The constructor for the internal ShapeGen implementation takes two input parameters:

- A pointer to a `Renderer` object
- The device clipping rectangle

The SGPtr constructor takes the same two input parameters, which it passes to the ShapeGen constructor.

The following code example shows how to use an SGPtr object to create a ShapeGen object. In this example, the `MyTest` function is called first. `MyTest` sets the background color in the device clipping rectangle to white. Then it calls the `MySub` function to fill a blue rectangle that's 250 pixels wide and 160 pixels high.

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

void MyTest(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(rend, clip);
    SGRect rect = { 100, 80, 250, 160 };

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

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

The `MyTest` function's first call parameter, `rend`, is a pointer to a basic renderer. The second parameter, `aarend`, points to an enhanced renderer, which is not used in this example. The interface for the `SimpleRenderer` class (see header file `renderer.h`) is derived from the `Renderer` base class (see header file `shapegen.h`), but contains an additional function, `SetColor`, that sets the color to be used for solid-color fills. (The `BasicRenderer` class, discussed in a later section, implements the `SimpleRenderer` interface. The `rend` parameter in this example, in fact, points to a `BasicRenderer` object.)

The MyTest function's third parameter, `clip`, specifies the device clipping rectangle.

As an automatic variable, the SGPtr object, `sg`, resides in the program stack and is deleted when it goes out of scope at the end of the `MyTest` function.

To serve as a smart pointer, the SGPtr class overloads the –> operator so that an SGPtr object, such as `sg`, can be used as a pointer to the encapsulated ShapeGen object. To enable a ShapeGen object pointer to be passed to a function (such as `MySub` in the preceding code example) as a call parameter, the SGPtr class overloads the * operator.

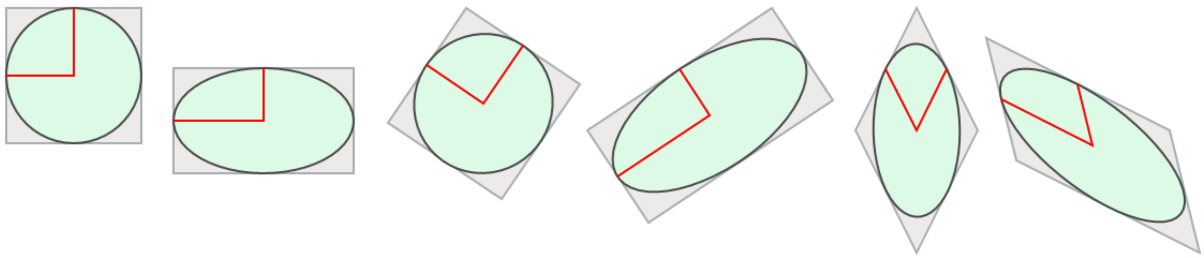
Ellipses and elliptic arcs

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, and 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.

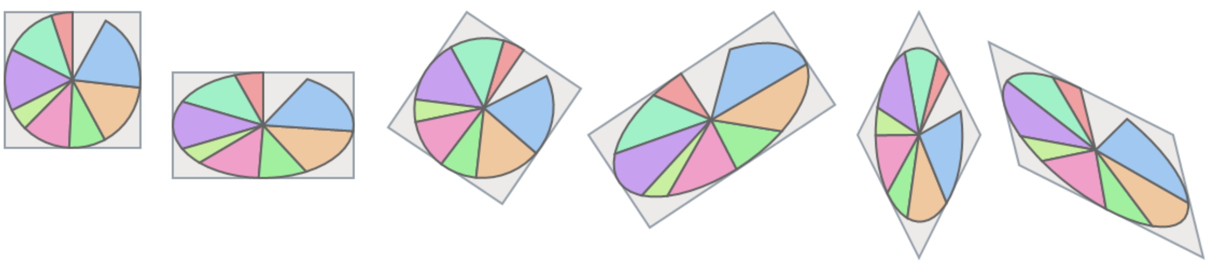


At the left edge of the screenshot, a circle is inscribed in a square. A circle is a special case of an ellipse. Red lines are drawn from the center of this particular ellipse to the end points of two conjugate diameters of the ellipse. (For the special case of a circle, any two perpendicular diameters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

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.



For more information, see the Wikipedia article on conjugate diameters, or see the article 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(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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. 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(3, &xy[i][1]);
        sg->CloseFigure();
        aarend->SetColor(RGBX(237,235,233));
        sg->FillPath(FILLRULE_EVENODD);
        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(FILLRULE_EVENODD);
            aarend->SetColor(RGBX(100,100,100));
            sg->StrokePath();
            astart += asweep;
        }
    }
}
```

Renderer

The ShapeGen library must be paired with a renderer so that shapes constructed by the ShapeGen object can be drawn on a graphics display. This GitHub project includes the source code for example renderers that run in Linux and Windows.

Two types of example renderer are provided for either operating system:

- A *basic* renderer that does solid-color fills (with no antialiasing)
- An *enhanced* renderer that does antialiasing, alpha blending, solid-color fills, tiled-pattern fills, linear-gradient fills, and radial-gradient fills

The basic renderer is faster, especially for filling large rectangular areas, and is suitable for computers with limited graphics capabilities. The enhanced renderer is intended for use with full-color displays. The user calls the `ShapeGen::SetRenderer` function to switch between these two renderers.

For Linux, the basic and enhanced renderers run on SDL2 (Simple DirectMedia Library, version 2). For Windows, one version of the two renderers runs on the Win32 API; the other version runs on SDL2 in Windows, and is essentially identical to the version that runs on SDL2 in Linux.

As previously discussed, the user constructs a path to specify a polygonal shape. To prepare a path to be rendered, ShapeGen subdivides the shape into a list of nonoverlapping trapezoids with horizontal tops and bottoms. The left and right sides of a trapezoid can be at arbitrary angles. A degenerate trapezoid can have a zero-width top or bottom. For a curved shape such as a circle, the trapezoids might shrink in height to a single horizontal span of pixels.

Instead of passing the trapezoids directly to a renderer to be drawn, ShapeGen passes the trapezoid list to a `ShapeFeeder` object and then passes this object to the renderer. The `ShapeFeeder` object operates as an iterator that cuts up each trapezoid into either rectangles (for a basic renderer) or subpixel spans (for an antialiasing renderer) and feeds them, one at a time, to the renderer.

The rectangles that ShapeGen feeds to a basic renderer (with no antialiasing) have integer width and height (measured in pixels). These rectangles are typically just one pixel in height unless the source trapezoid happens to have vertical sides.

Enhanced renderers require shapes to be described at the subpixel level to enable antialiasing. ShapeGen feeds *subpixel spans* to an enhanced renderer. Each span is a horizontal row of subpixels that spans the distance between the left and right sides of a trapezoid. The span is specified by its starting and ending *x* coordinates, and its *y* coordinate. To support subpixel addressing, these coordinates are fixed-point values. An enhanced renderer uses these spans to construct coverage bitmasks to use for antialiasing.

The ShapeGen library calls the renderer's RenderShape function to do all of the drawing. Each RenderShape call fills or strokes a shape specified by a user-constructed path. The input parameter to this function is a ShapeFeeder object.

The simplicity of a *basic* renderer makes it easy to port to any processor for which a C++ compiler is available. For example, the basic renderer that runs on SDL2 implements 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);
}
```

This version of the RenderShape function (see source file `sdlmain.cpp`) uses the `SDL_FillRect` function to fill the rectangles, and therefore runs on platforms, such as Linux, for which SDL2 is available. The version that runs on Windows GDI (see `winmain.cpp`) calls the `FillRect` function instead.

The `RenderShape` functions implemented by the *enhanced* renderers in this project are necessarily more complex than this. In addition to the `RenderShape` function, ShapeGen's `Renderer` interface definition (see source file `shapegen.h`) includes `QueryYResolution` and `SetMaxWidth` functions specifically to support antialiasing renderers. A fourth function, `SetScrollPosition`, supports scrolling.

All four functions in the following `Renderer` base class definition are called exclusively by the ShapeGen object:

```
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; }
};
```

An antialiasing 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`. (The *x* coordinates are always in a 16.16 fixed-point format.) This 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. When the renderer is installed (for example, if the user calls the `ShapeGen::SetRenderer` function), ShapeGen immediately calls the renderer's `QueryYResolution` and `SetMaxWidth` functions. Note that a basic renderer, which does no antialiasing, simply uses the versions of these two functions that are defined in the `Renderer` base class above.

The `SetScrollPosition` function supports horizontal and vertical scrolling of complex paint (that is, patterns and gradients) so that they remain in sync with painted shapes when the window is scrolled. A basic renderer, which does only solid-color fills, simply uses the version of this function that is defined in the `Renderer` base class above.

Additionally, a renderer derived from thr `Renderer` class is expected to provide user-callable functions to, for example, specify the solid color, tiled pattern, or color gradient to be used to render shapes. All the example renderers included in this GitHub project can do solid-color fills and therefore must provide a user-callable `SetColor` function. An enhanced renderer provides a number of additional user-callable functions. These functions are described in detail in the EnhancedRenderer functions section of this user's guide.

As previously described, an enhanced renderer's `RenderShape` function receives a series of subpixel spans from the `ShapeFeeder` and then uses these spans to construct coverage bitmasks for antialiasing.

Each of the example enhanced renderers in this GitHub project constructs a 4-by-8 coverage bitmask (four subpixels high, and eight subpixels wide) for each pixel in the scan line that is currently being rendered.

Note that an antialiasing renderer needs only enough scratchpad memory to construct one scan line of pixel data at a time. That's because the subpixel spans that the `ShapeFeeder` iterator supplies to the renderer's `RenderShape` function are always provided in ascending-*y* order. Thus, the renderer completely finishes constructing a shape's contribution to one scan line before starting on the next scan line.

To do antialiasing, the enhanced renderers rely on the alpha-blending capabilities of the underlying platform. The SDL2 version calls the `SDL_BlitSurface` function, and the Windows GDI version calls the `AlphaBlend` function.

The Windows GDI versions of the `RenderShape` functions in this project write "directly" to the window on the screen, which you might find useful if you're debugging a renderer. That's because you can single-step through the `RenderShape` function for a renderer in a debugger, and inspect each rectangle or span as it is filled or blitted to the screen. Of course, Windows doesn't allow a user to write directly to screen memory, and so every rectangle or span undergoes rigorous bounds checking before it is drawn. All this checking noticeably slows drawing operations. After you finish debugging your graphics program, you can improve its performance by first drawing everything to an offscreen buffer and then copying the completed image to the screen in a single bitblt operation. That's how the SDL2 versions of the renderers in this project work, and they run significantly faster.

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

```cpp
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 (aka 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 Bezier 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 Bezier 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

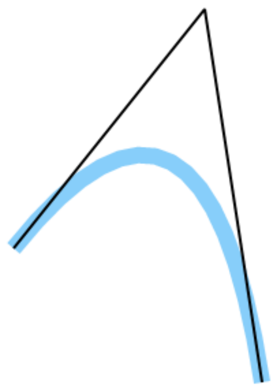
The current point is the starting point for the spline. Parameters v1 and v2 specify the control point and end point of the spline.

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

Example

This example uses the Bezier2 function to draw a quadratic Bezier spline (in blue) and its control polygon (in black). (Parameter rend points to the basic renderer, aarend points to the enhanced renderer, clip specifies the device clipping rectangle, and variable sg is the ShapeGen object pointer.)

```
void example01(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint v0 = { 100, 200 }, v1 = { 200, 75 }, v2 = { 230, 270 };

    // Draw quadratic Bezier spline in red
    aarend->SetColor(RGBX(255,120,100));
    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 result 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 end point, v2, is on the right. The control point, v1, is at the top. 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 Bezier 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 Bezier control point for the spline.

V2

An `SGPoint` structure that specifies the *x-y* coordinates at the second Bezier 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

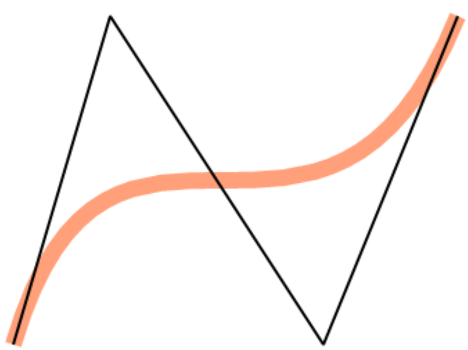
The current point is the starting point for the spline. Parameters `v1`, `v2`, and `v3` specify the two control points and end point of the spline.

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

Example

This example uses the Bezier3 function to draw a cubic Bezier spline (in orange) and its control polygon (in black). (Parameter rend points to the basic renderer, aarend points to the enhanced renderer, clip specifies the device clipping rectangle, and variable sg is the ShapeGen object pointer.)

```
void example02(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint v0 = { 140, 308 }, v1 = { 210, 70 },
            v2 = { 364, 308 }, v3 = { 461, 70 };

    // Draw cubic Bezier spline in red
    aarend->SetColor(RGBX(255,160,122));
    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 result 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 (aka 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. 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.

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

Header

`shapegen.h`

See also

`ShapeGen::StrokePath`

`ShapeGen::FillPath`

`ShapeGen::CloseFigure`

`ShapeGen::SetClipRegion`

`ShapeGen::SetMaskRegion`

`ShapeGen::EndFigure`

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

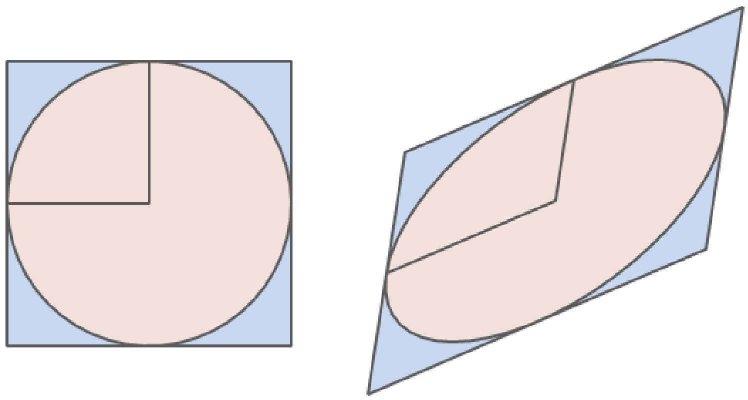
It's straightforward to use the `Ellipse` function to inscribe an ellipse in a parallelogram so that the ellipse's center coincides with the center of the parallelogram, and the ellipse touches the parallelogram at the midpoint of each of its four sides. If we are given three consecutive corner points, *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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example03(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(FILLRULE_EVENODD);
        sg->Move(xy[i][0].x, xy[i][0].y);
        sg->PolyLine(3, &xy[i][1]);
        sg->CloseFigure();
        aarend->SetColor(RGBX(200,215,240));
        sg->FillPath(FILLRULE_EVENODD);
```

```
        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 result 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 Bezier curves: both are *affine-invariant*. For a Bezier 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).

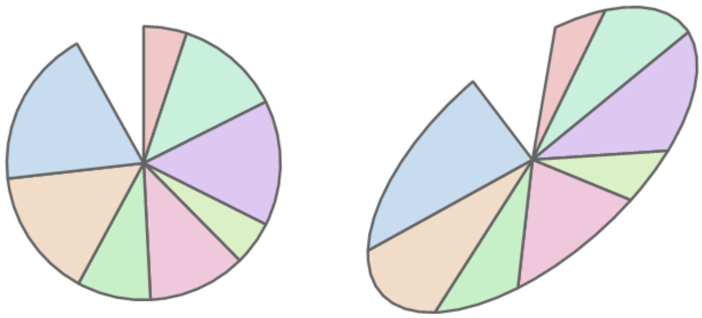
For more information about conjugate diameter end points, see Ellipses and Elliptic Arcs.

Example

This example uses the `EllipticArc` function to draw two affine-transformed views of the same pie chart. (Parameter `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example04(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(FILLRULE_EVENODD);
            aarend->SetColor(RGBX(100,100,100));
            sg->StrokePath();
            astart += asweep;
        }
}
```

The result 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

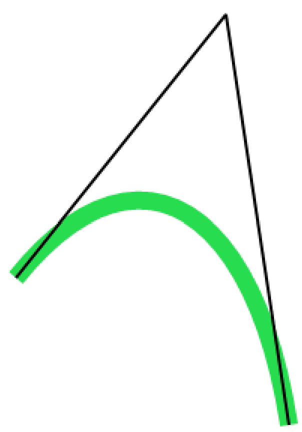
The current point is the starting point for the spline. Parameters v1 and v2 specify the control point and end point of the spline. On return from this function, v2 is the new current point.

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 rend points to the basic renderer, aarend points to the enhanced renderer, clip specifies the device clipping rectangle, and variable sg is the ShapeGen object pointer.)

```
void example05(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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 result 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 end point, `v2`, is on the bottom right. The control point, `v1`, is at the top. 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 (aka 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.

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.

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

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 specified by the caller.

Syntax

C++

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

Parameters

fillrule

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

FILLRULE_EVENODD – Even-odd (aka parity) fill rule

FILLRULE_WINDING – Nonzero winding number fill rule

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

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::EndFigure`

`ShapeGen::CloseFigure`

`ShapeGen::StrokePath`

ShapeGen::GetBoundingBox function

The `GetBoundingBox` function retrieves the minimum bounding box for the points in the current path.

Syntax

C++

```
int ShapeGen::GetBoundingBox(
   SGRect *bbox
);
```

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. This pointer can be null (zero) if the caller simply wants a count of the number of points in the path.

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.

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 (aka subpaths), the bounding box takes into account the points in all the figures.

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.

If the path contains a single point, the width and height values calculated for the bounding box are small but not necessarily zero.

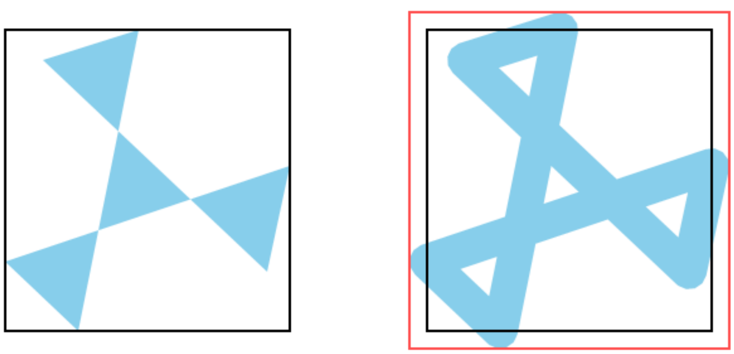
Example

This example uses the `GetBoundingBox` function to get the minimum bounding boxes for two identical shapes. One is subsequently filled, and the other is stroked. (Parameter `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example06(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint xy[] = {
        { 130, 97 }, { 308, 265 }, { 326, 181 },
        { 100, 257 }, { 158, 312 }, { 206, 73 }
    };
    float linewidth = 28.0;
    COLOR ltblue = RGBX(135, 206, 235);
    SGRect bbox;

    // Fill the shape with solid light blue
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
    sg->PolyLine(5, &xy[1]);
    aarend->SetColor(ltblue);
    sg->FillPath(FILLRULE_EVENODD);

    // Get bounding box and outline it in black
    sg->GetBoundingBox(&bbox);
    sg->SetLineWidth(2.0);
    sg->SetLineJoin(LINEJOIN_MITER);
    sg->BeginPath();
    sg->Rectangle(bbox);
    aarend->SetColor(RGBX(0,0,0));
    sg->StrokePath();

    // Move the shape to the right
    for (int i = 0; i < 6; ++i)
        xy[i].x += 335;
```

```
    // Stroke the shape in light blue
    sg->SetLineJoin(LINEJOIN_ROUND);
    sg->SetLineWidth(linewidth);
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
    sg->PolyLine(5, &xy[1]);
    sg->CloseFigure();
    aarend->SetColor(ltblue);
    sg->StrokePath();

    // Get the bounding box and outline it in black
    sg->GetBoundingBox(&bbox);
    sg->SetLineWidth(2.0);
    sg->SetLineJoin(LINEJOIN_MITER);
    sg->BeginPath();
    sg->Rectangle(bbox);
    aarend->SetColor(RGBX(0,0,0));
    sg->StrokePath();

    // Expand each side of the bounding box by half the line width
    bbox.x -= linewidth/2;
    bbox.y -= linewidth/2;
    bbox.w += linewidth;
    bbox.h += linewidth;

    // Outline the expanded bounding box in red
    sg->BeginPath();
    sg->Rectangle(bbox);
    aarend->SetColor(RGBX(255,80,80));
    sg->StrokePath();
}
```

The result is shown in the following screenshot.



First, the code example constructs a path and fills it to produce the shape shown on the left side of the screenshot. The GetBoundingBox function is called on the path, and the resulting bounding box is outlined in black.

Next, the code example constructs a similar path, shifted to the right, and strokes it to produce the shape on the right side of the preceding screenshot. The GetBoundingBox function is called on the path, and the resulting bounding box is outlined in black. In this case, the edges of the stroked path extend beyond the bounding box. To address this problem, the code example expands the original bounding box by half the stroked line width on all four sides. This expanded bounding box (outlined in red) successfully encloses the entire shape. Of course, this trick might not work as well for mitered joins.

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 (aka 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 (aka 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 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 (aka 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 Bezier spline curves, starting at the current point.

Syntax

C++

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

Parameters

`npts`

The number of points in the xy array.

`xy`

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

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.

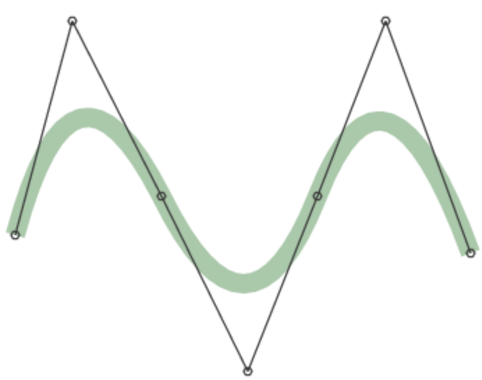
Whereas the `ShapeGen::Bezier2` function constructs a single quadratic Bezier curve, the `PolyBezier2` function can construct multiple quadratic Bezier curves in a single call.

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 Poly`Bezier2` function to draw three connected quadratic Bezier spline curves. (Parameter `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example07(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(6, &xy[1]);
    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(6, &xy[1]);
    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 result 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 Bezier spline curves, starting at the current point.

Syntax

C++

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

Parameters

npts

The number of points in the xy array.

xy

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

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 Bezier curve, the `PolyBezier3` function can construct multiple cubic Bezier curves in a single call.

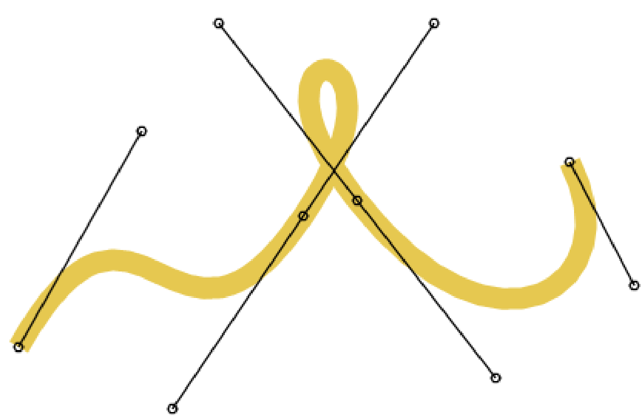
On return from this function, the end point of the final spline in the array is the new current point.

Example

This example uses the `PolyBezier3` function to draw three connected cubic Bezier spline curves. (Parameter `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example08(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(9, &xy[1]);
    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 result 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(
   int npts,
   const SGPoint xy[]
);
```

Parameters

npts

The number of points in the xy array.

xy

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

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.

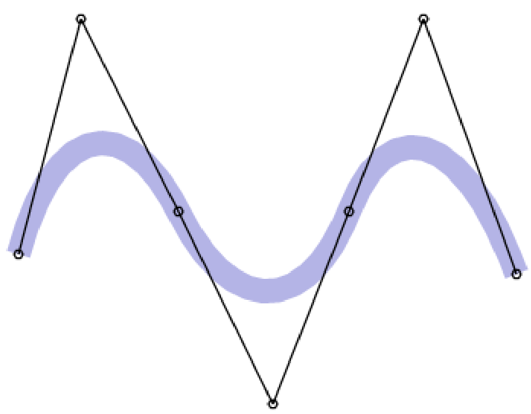
On return from this function, the end point of the final spline in the array is the new current point.

Example

This example uses the `PolyEllipticSpline` function to draw a series of connected elliptic spline curves. (Parameter `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example09(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(6, &xy[1]);
    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(6, &xy[1]);
    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 result 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(
   int npts,
   const SGPoint xy[]
);
```

Parameters

npts

The number of points in the xy array.

xy

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

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

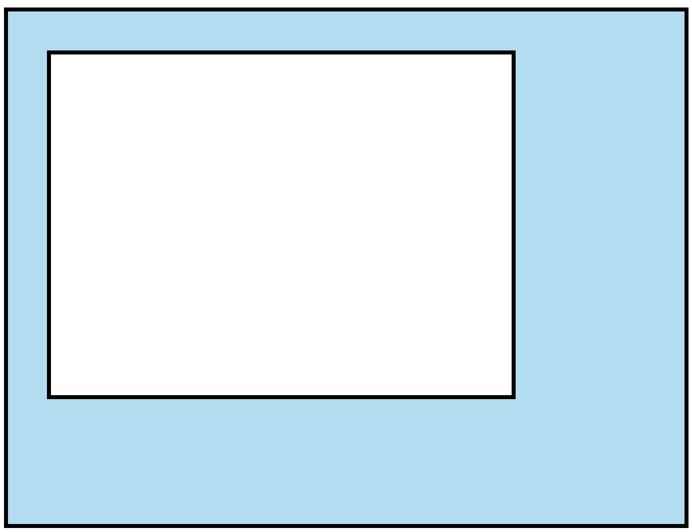
Construction of a rectangle by the Rectangle function proceeds in a clockwise direction if we assume the following:

- 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 rend points to the basic renderer, aarend points to the enhanced renderer, clip specifies the device clipping rectangle, and variable sg is the ShapeGen object pointer.)

```
void example10(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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->FillPath(FILLRULE_WINDING);  // <-- winding number fill rule!
    sg->SetLineWidth(2.0);
    sg->SetLineJoin(LINEJOIN_MITER);
    rend->SetColor(RGBX(0,0,0));
    sg->StrokePath();
}
```

	The result 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.

The device clipping rectangle that is restored by the `ResetClipRegion` function reflects any changes made by previous calls to the `InitClipRegion` and `SetScrollPosition` functions. The `ShapeGen::InitClipRegion` function changes the width and height of the device clipping rectangle. The `ShapeGen::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

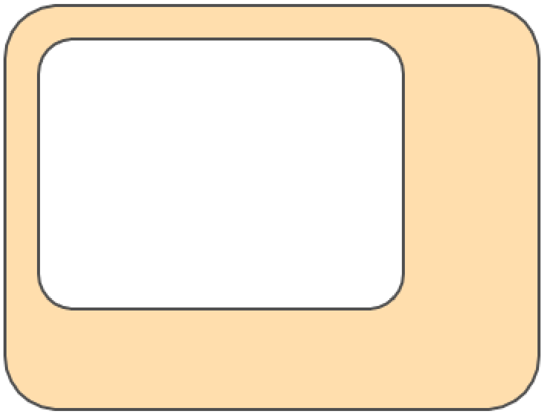
Construction of a rounded rectangle by the `RoundedRectangle` function proceeds in a clockwise direction if we assume the following:

- `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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example11(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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->FillPath(FILLRULE_WINDING);  // <-- winding number fill rule!

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



The result 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 current clipping region and the interior of the current path.

Syntax

C++

```
bool ShapeGen::SetClipRegion(
  FILLRULE fillrule
);
```

Parameters

`fillrule`

The fill rule to use for converting the path to a filled region, which is then intersected with the current clipping region to form the new clipping region. Specify one of the following values for this parameter:

FILLRULE_EVENODD – Even-odd (aka parity) fill rule

FILLRULE_WINDING – Nonzero winding number fill rule

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 confines drawing to the interior of an arbitrarily shaped area.

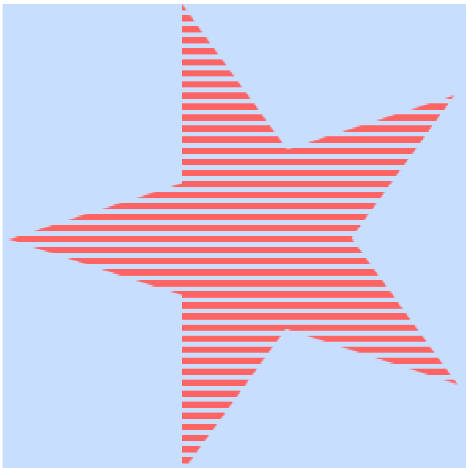
In contrast to the `ShapeGen::SetMaskRegion` function, which constructs a new clipping region that is the intersection of the current clipping region with the *exterior* of the current path, the `SetClipRegion` function constructs a new clipping region that is the intersection of the current clipping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

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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example12(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(FILLRULE_EVENODD);

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

    // 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->SetClipRegion(FILLRULE_WINDING);

    // 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 result is shown in the screenshot at left.

The code example starts by filling a blue square that lies entirely within the current clipping region. Next, a star-shaped path is constructed, and the clipping region is intersected with this path to form a new, star-shaped clipping region. 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 clipping region is drawn.

Header

`shapegen.h`

See also

`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
);
```

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.

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
);
```

Parameters

`flatness`

The maximum chord-to-curve distance, measured in pixels. Set this parameter to a value in the range 0.2 to 100.0.

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 0.6 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,
   int offset,
   float mult
);
```

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`.

`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.

`mult`

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

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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example13(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(3, &xy[3]);
        sg->Line(xy[6].x, xy[6].y);
        sg->StrokePath();
        for (int j = 0; j < 7; ++j)
            xy[j].x += 170;
    }
}
```

The result 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++

```
bool ShapeGen::SetLineEnd(
   LINEEND capstyle
);
```

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 (aka butt cap)
LINEEND_ROUND – Rounded line end (aka round cap)
LINEEND_SQUARE – Squared line end (aka projecting cap)

Return value

None

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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example14(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(2, &vert[1]);
        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 result 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++

```
void ShapeGen::SetLineJoin(
   LINEJOIN joinstyle
);
```

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
LINEJOIN_ROUND – Rounded join
LINEJOIN_MITER – Mitered join

Return value

None

Remarks

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

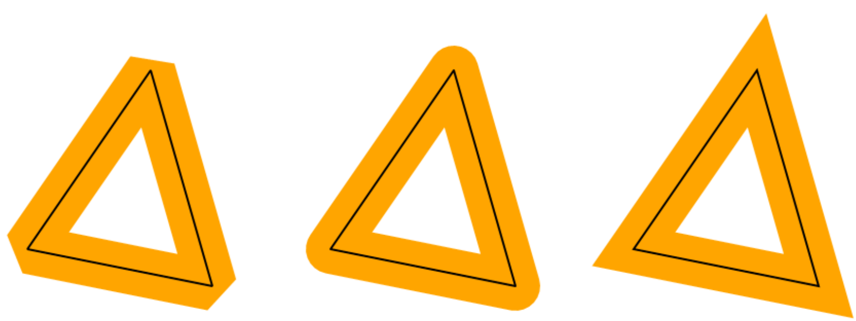
The default value for the line-join attribute is LINEJOIN_BEVEL.

Example

This example uses the `SetLineJoin` function to set different line-join attributes for three stroked paths. (Parameter `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example15(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(2, &vert[1]);
        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 result 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
);
```

Parameters

`width`

The line-width, in pixels, of a stroked path.

Return value

Returns the previous line-width setting.

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 basic 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 the current path.

Syntax

C++

```
bool ShapeGen::SetMaskRegion(
   FILLRULE fillrule
);
```

Parameters

`fillrule`

The fill rule to use for converting the path to a filled region, which is then masked off from the current clipping region to form the new clipping region. Specify one of the following values for this parameter:

FILLRULE_EVENODD – Even-odd (aka parity) fill rule

FILLRULE_WINDING – Nonzero winding number fill rule

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::SetClipRegion` function, which constructs a new clipping region that is the intersection of the current clipping region with the *interior* of the current path, the `SetMaskRegion` function constructs a new clipping region that is the intersection of the current clipping region with the *exterior* of the current path

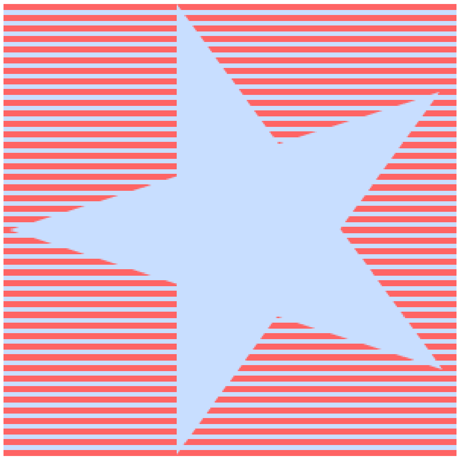
The `SetMaskRegion` and `SetClipRegion` functions can modify the clipping region inside the device clipping rectangle, but cannot expand the clipping region beyond the device clipping rectangle.

Example

This example uses the `SetMaskRegion` function to exclude a star-shaped path from the interior of the clipping region. (Parameter `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example16(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(FILLRULE_EVENODD);

    // 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->SetMaskRegion(FILLRULE_WINDING);

    // 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 result is shown in the following screenshot.



The code example starts by filling a blue square that lies entirely within the current clipping region. Next, a star-shaped path is constructed, and the clipping region is intersected with the *exterior* of this path to form a new clipping region. 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 (outside the masked-off area) is 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
);
```

Parameters

`mlim`

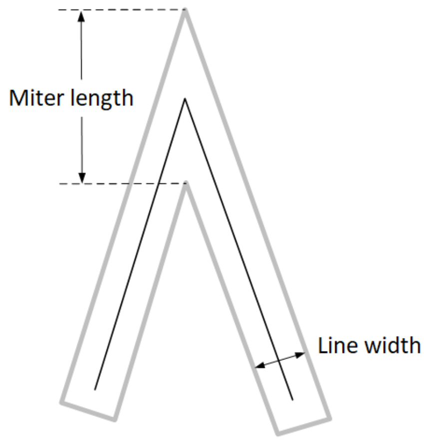
The new miter-limit setting. Set this parameter to a value greater than or equal to 1.0.

Return value

Returns the previous miter-limit setting.

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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example17(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(2, &vert[1]);
            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 result 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:

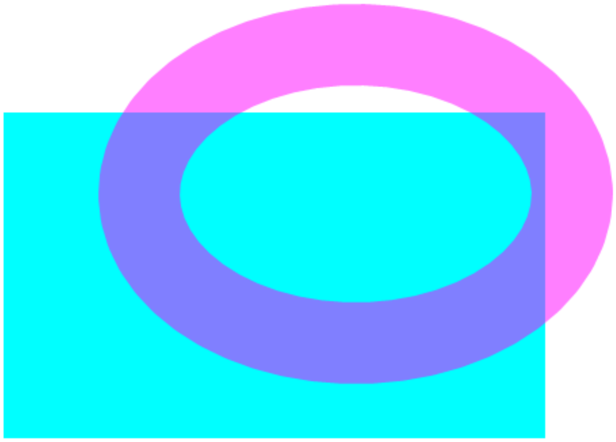
- The current clipping region is reset to the device clipping rectangle. The effect is the same as a call to the ShapeGen::ResetClipRegion function.
- Any clipping region previously saved by the ShapeGen::SaveClipRegion function or swapped out by the ShapeGen::SwapClipRegion function is discarded.

Example

This example uses the SetRenderer function to switch from one renderer to another. (Parameter rend points to the basic renderer, aarend points to the enhanced renderer, clip specifies the device clipping rectangle, and variable sg is the ShapeGen object pointer.)

```
void example18(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(rend, clip);
    SGRect rect = { 100, 110, 400, 240 };
    SGPoint v0 = { 360, 170 }, v1 = { 360+160, 170 }, v2 = { 360, 170+110 };

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

    // 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 result is shown in the following screenshot.



In this example, the basic renderer, `rend`, is passed as an input parameter to the `SGPtr` constructor, which creates a ShapeGen object and installs `rend` as this object's initial renderer. Using this renderer, a rectangle is painted cyan. This rectangle is completely opaque, as are all shapes painted by the basic renderer.

Next, a `SetRenderer` function call installs the antialiasing renderer, `aarend`, in place of the basic 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 (aka 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.

The ShapeGen demo program in this GitHub project enables the viewer to observe the effect of the `SetScrollPosition` function. For the SDL2 version of the demo, when the control key is held down, the arrow keys generate calls to this function. For the Win32 version, moving the thumb tabs on the window scroll bars generates `SetScrollPosition` function calls.

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++

```cpp
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.

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. The result is an empty color stop table. 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, beginning 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 32-bit value that contains 8-bit red, green, blue, and alpha components. These components are specified in RGBA32 pixel format, as follows:

	31 … 24	23 … 16	15 … 8	7 … 0
RGBA32:	alpha	blue	green	red

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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example19(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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();
}
```

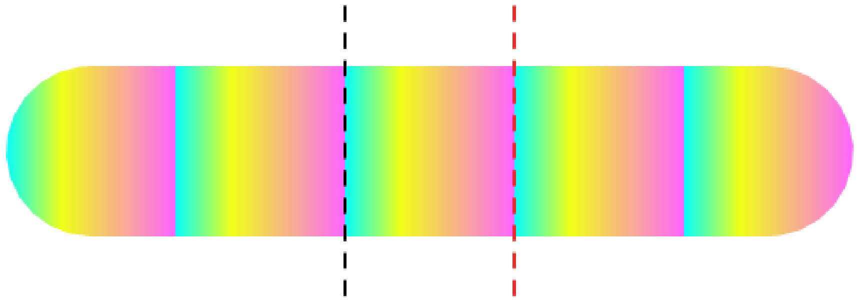
The first color stop in this example is set to cyan, and the last color stop is set to magenta. In between these two color stops, a third color stop, at `offset` = 0.33, is set to yellow. As required, the three color stops are added in order of increasing offsets, with the first and last offsets set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



Header

`renderer.h`

See also

EnhancedRenderer::ResetColorStops

EnhancedRenderer::SetLinearGradient

EnhancedRenderer::SetRadialGradient

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::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 32-bit value that contains 8-bit red, green, blue, and alpha components. These components are specified in RGBA32 pixel format, as follows:

	31 … 24	23 … 16	15 … 8	7 … 0
RGBA32:	alpha	blue	green	red

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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example20(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(FILLRULE_EVENODD);
    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(FILLRULE_EVENODD);
```

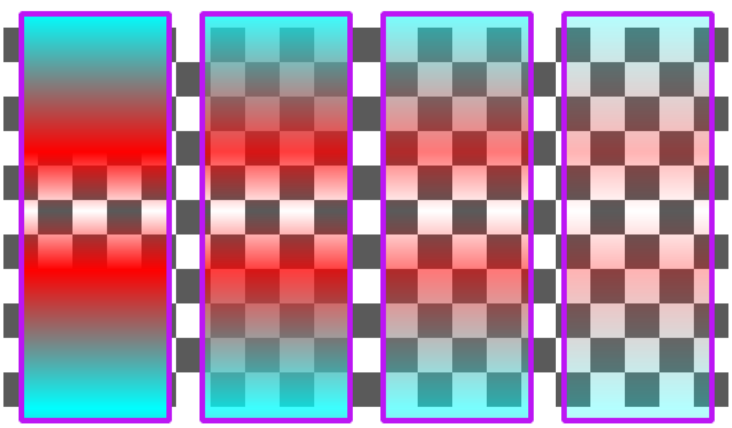
```
        aarend->SetConstantAlpha(255);
        aarend->SetColor(RGBX(188,22,244));
        sg->StrokePath();
        rect.x += 131;
    }
}
```

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 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 result 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`.

During the `SetLinearGradient` function call, the linear-gradient paint generator also takes snapshots of the renderer's affine transform matrix and source constant alpha. For more information, see `EnhancedRenderer::SetTransform` and `EnhancedRenderer::SetConstantAlpha`.

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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example21(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(FILLRULE_EVENODD);

    // 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(FILLRULE_EVENODD);

    // 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(FILLRULE_EVENODD);
}
```

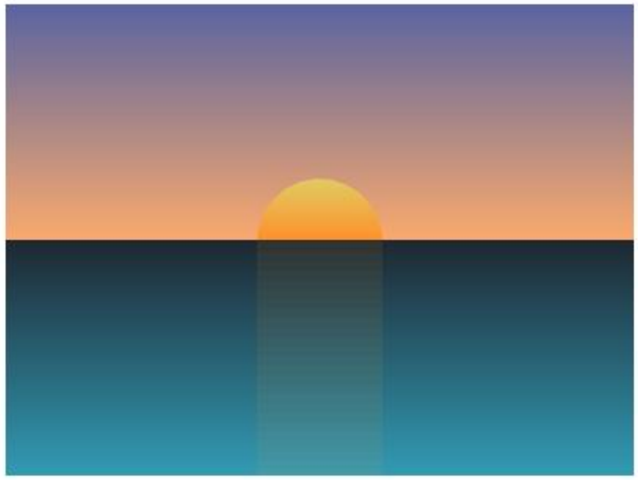
In this example, the color gradients are all oriented vertically. In the starting and ending points passed as arguments to each `SetLinearGradient` function call, the two *x*-coordinates are identical, which means that the two *y*-coordinates wholly determine the slope of the gradient.

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 result is shown in the following screenshot.



Header

`renderer.h`

See also

`EnhancedRenderer::AddColorStop`

`EnhancedRenderer::ResetColorStops`

`EnhancedRenderer::SetTransform`

`EnhancedRenderer::SetConstantAlpha`

EnhancedRenderer::SetPattern function

The two `SetPattern` functions create 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 section 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 section 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 section 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 section above.

Return value

None

Remarks

The two versions of the `SetPattern` function listed in the Syntax section 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 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. 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:

	31 … 24	23 … 16	15 … 8	7 … 0
RGBA32:	alpha	blue	green	red
BGRA32:	alpha	red	green	blue

If the `FLAG_PREMULTALPHA` flag is *not* set, the `SetPattern` function converts each pixel in the source image to premultiplied-alpha format; that is, each pixel's red, green, and blue color components are multiplied by the pixel's alpha value.

Tiled-pattern fill operations use the renderer's source constant alpha value (set by the `EnhancedRenderer::SetConstantAlpha` function) and affine transform matrix (set by the `EnhancedRenderer::SetTransform` function). A tiled-pattern paint generator takes a snapshot of these settings during the `SetPattern` function call and then continues to use these settings 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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example22(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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(FILLRULE_EVENODD);
}
```

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 result is shown in the following screenshot.

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`.

During the `SetRadialGradient` function call, the radial-gradient paint generator also takes snapshots of the renderer's current affine transform matrix and source constant alpha. For more information, see `EnhancedRenderer::SetTransform` and `EnhancedRenderer::SetConstantAlpha`.

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 reduces 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 $\lfloor t \rfloor$ is even) or reflect (if $\lfloor t \rfloor$ 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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

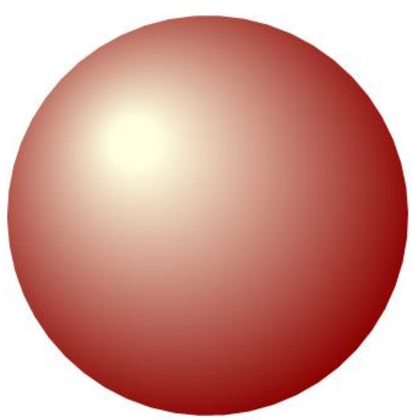
```
void example23(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint v0 = { 200, 200 }, v1 = { 200-162, 200 }, v2 = { 200, 200-162 };

    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(FILLRULE_EVENODD);
}
```

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 result 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://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/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 defined 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 defined as follows:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \mathrm{a} & \mathrm{c} & \mathrm{e} \\ \mathrm{b} & \mathrm{d} & \mathrm{f} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

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 the `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 example shows how the `SetTransform` function can be used to take the same 2-D affine transformation that is applied to a painted shape and apply it to the complex paint (gradient or pattern) that fills the shape. (Parameter `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example24(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGRect rect = { 40, 40, 400, 300 };
    SGPoint u[3] = { { 200, 160 }, { 200-70, 160 }, { 200, 160-70 } };
    SGPoint v[3] = { { 290, 215 }, { 290-100, 215 }, { 290, 215-100 } };
    float xform[6] = { 0.375, -0.650, 1.083, 0.625, 364.3, 227.1 };
    char dash[] = { 1, 0 };

    // Fill the two rectangles with background color gray
    aarend->SetColor(RGBX(180,180,180));
    sg->BeginPath();
    sg->Rectangle(rect);
    sg->FillPath(FILLRULE_WINDING);

    // 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(FILLRULE_WINDING);

    // 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(u[0], u[1], u[2]);
    sg->StrokePath();

    // Stroke the outline of the ending circle in red
    aarend->SetColor(RGBX(255,0,0));
    sg->BeginPath();
    sg->Ellipse(v[0], v[1], v[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->FillPath(FILLRULE_WINDING);

    // 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(FILLRULE_WINDING);

    // Transform the coordinates of the starting and ending circles.
    // To improve resolution, save the transformed coordinates in
    // 16.16 fixed-point format.
    for (int i = 0; i < 3; ++i)
    {
        float xtmp = 65536*(xform[0]*u[i].x + xform[2]*u[i].y + xform[4]);
        u[i].y = 65536*(xform[1]*u[i].x + xform[3]*u[i].y + xform[5]);
        u[i].x = xtmp;

        xtmp = 65536*(xform[0]*v[i].x + xform[2]*v[i].y + xform[4]);
        v[i].y = 65536*(xform[1]*v[i].x + xform[3]*v[i].y + xform[5]);
        v[i].x = xtmp;
    }

    // Tell ShapeGen that 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(u[0], u[1], u[2]);
    sg->StrokePath();

    // Outline the transformed ending circle in red
    aarend->SetColor(RGBX(255,0,0));
    sg->BeginPath();
    sg->Ellipse(v[0], v[1], v[2]);
    sg->StrokePath();
}
```

In this example, the shapes to be painted are the starting and ending circles for the radial gradient. The original, untransformed radial gradient is used to fill the rectangle on the left. The `ShapeGen::Ellipse` and `ShapeGen::StrokePath` functions are used to outline the starting and ending circles for the radial gradient with black and red dashed lines.

Next, the affine transformation specified in the `xform` array is applied to both the radial gradient and the two shapes (the circles). The transformed radial gradient is then used to fill the rectangle on the right. (Note that we're transforming only the circles, and not the rectangle.) The two circles, which have been transformed into ellipses, are again outlined with black and red dashed lines.

The `xform` array embodies the following transformation:

1. Translate the original figure from the center coordinates (240, 190) of the rectangle on the left to the *x*-*y* origin.

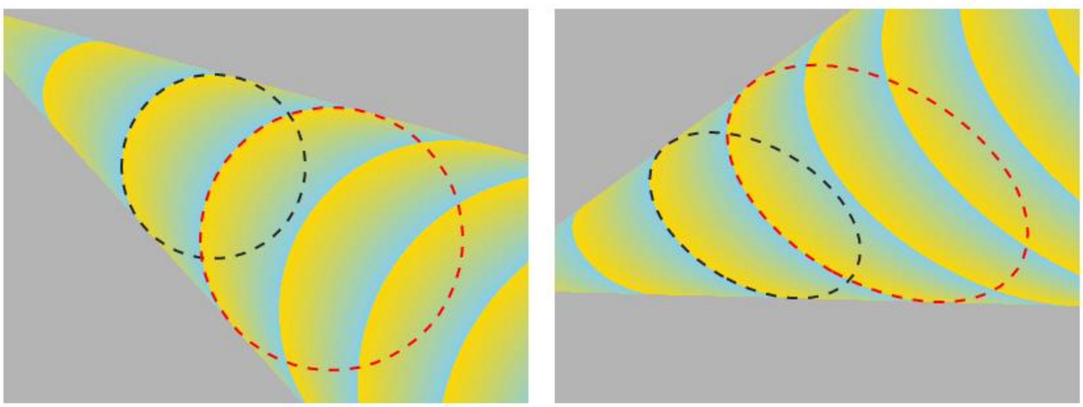
2. To squash the circles into ellipses, scale the figure by 0.75 in the *x* dimension, and by 1.25 in the *y* dimension.
3. Rotate the figure by π/3 radians in the counterclockwise direction.
4. Translate the figure from the *x*-*y* origin to the center coordinates (660, 190) of the rectangle on the right.

The transformation can be expressed as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\frac{\pi}{3} & \sin\frac{\pi}{3} & 660 \\ -\sin\frac{\pi}{3} & \cos\frac{\pi}{3} & 190 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.75 & 0 & 0 \\ 0 & 1.25 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -240 \\ 0 & 1 & -190 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Applying this transformation to the two shapes (the circles) is straightforward – the required matrix multiplications are performed in the `for`-loop in the code example. However, applying the transformation to the radial gradient is a bit tricky, and the `SetTransform` function takes care of all the messy details.

The result is shown in the following screenshot.



Example 2

This next 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 `rend` points to the basic renderer, `aarend` points to the enhanced renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example25(SimpleRenderer *rend, EnhancedRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(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 xy[][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(xy); ++i)
    {
        // Use symmetry to calculate the 4th vertex of the square,
        // rectangle, or parallelogram that is to be filled
        float x0 = xy[i][0].x, y0 = xy[i][0].y;
        float x1 = xy[i][1].x, y1 = xy[i][1].y;
        float x2 = xy[i][2].x, y2 = xy[i][2].y;

        xy[i][3].x = x0 - x1 + x2;
        xy[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(xy[i][0].x, xy[i][0].y);
        sg->PolyLine(3, &xy[i][1]);
        sg->CloseFigure();
        aarend->SetPattern(image, 0,0, w,h,w, 0);
        sg->FillPath(FILLRULE_EVENODD);

        // Outline the square, rectangle, or parallelogram with a
        // black dashed line
        aarend->SetColor(RGBX(0,0,0));
        sg->StrokePath();
    }
}
```

In this example, the `image` array contains a simple 7-by-7 test image.

The initializer for the xy 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 xy array. In preparation for this call, a 2-D affine transformation matrix is constructed and loaded into the xform array. This matrix transforms 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 can be expressed as follows:

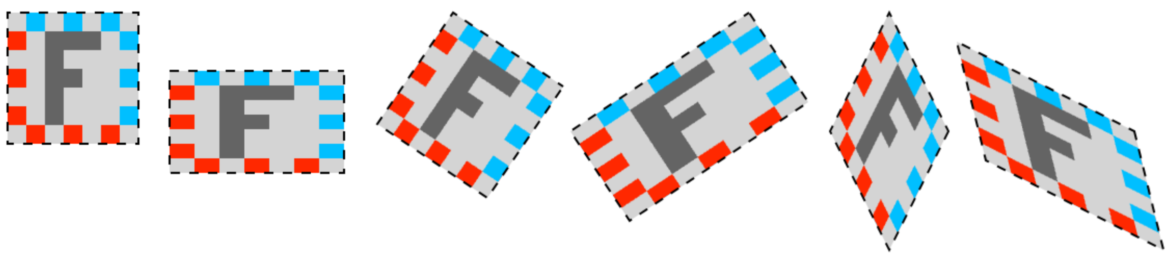
$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{x_P y_Q - x_Q y_P} \begin{bmatrix} w & 0 & 0 \\ 0 & h & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_Q & -x_Q & 0 \\ -y_P & x_P & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

When the rightmost 3x3 matrix above translates the vertex at (x_1, y_1) to the *x-y* origin, the first and third vertices are translated to coordinates (x_P, y_P) = (x_0-x_1, y_0-y_1) and (x_Q, y_Q) = (x_2-x_1, y_2-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 xy array, the ShapeGen::StrokePath function call outlines the shape boundary with a black dashed line.

The result is shown in the following screenshot.



Header

renderer.h

See also

EnhancedRenderer::SetPattern

EnhancedRenderer::SetLinearGradient

EnhancedRenderer::SetRadialGradient