ShapeGen User's Guide

Jerry R. VanAken

May 6, 2020

ShapeGen concepts

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

Introduction

This GitHub project contains the C++ source code for the ShapeGen graphics library, which implements a relatively simple but powerful 2-D polygonal shape generator. 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 the 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.

The ShapeGen programming interface 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 library consists of six well-commented C++ source files and two include 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`	11K	Constructs paths for cubic and quadratic Bezier curves
`edge.cpp`	9K	Manages lists of polygonal edges, does clipping, and drives renderer
`path.cpp`	22K	Performs basic path-management functions
`stroke.cpp`	25K	Converts paths into stroked lines and curves
`thinline.cpp`	8K	Converts paths into *thin* stroked lines and curves
`shapegen.h`	9K	Defines ShapeGen public interface
`shapepri.h`	14K	Defines 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`	93K	Contains source code for 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
`demo.h`	4K	Defines TextApp interface and demo program parameters

Finally, 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 example renderers to run on Win32 API
`sdlmain.cpp`	13K	Implements example renderers to run on SDL2

Because 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 very simple renderer to drive a graphics display on low-cost or compact hardware.

Additionally, ShapeGen can take advantage of systems with more advanced graphics capabilities to do antialiasing and alpha blending.

In comparison with larger and more complex open-source graphics software, the small size and relative simplicity of the source code for the ShapeGen implementation makes it easier to build on and modify. The clean separation of the ShapeGen library and renderer implementation enables developers to more easily add new rendering capabilities such as color gradients or pattern fills.

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).

Thus, to draw a series of 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 respective first and current points. 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.)

Second, if the ShapeGen library is instead connected to an antialiasing renderer, each pixel is partitioned into subpixels, and 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 (aka 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 – and also of the window on the graphics display. Thus, the device clipping rectangle specifies the region of the ShapeGen coordinate space that the viewer sees in the window.

Typically, (*x*,*y*) = (0,0), in which case the origin of the ShapeGen coordinate system coincides with the top-left corner of the window. However, the device clipping 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. 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 can incur significant processing overhead. It is not meant 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.

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 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 the 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 operates as a smart pointer that creates a ShapeGen object and provides 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, SimpleRenderer *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 antialiasing renderer, which is not used in this example. The interface for the `SimpleRenderer` class (see header file `demo.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.

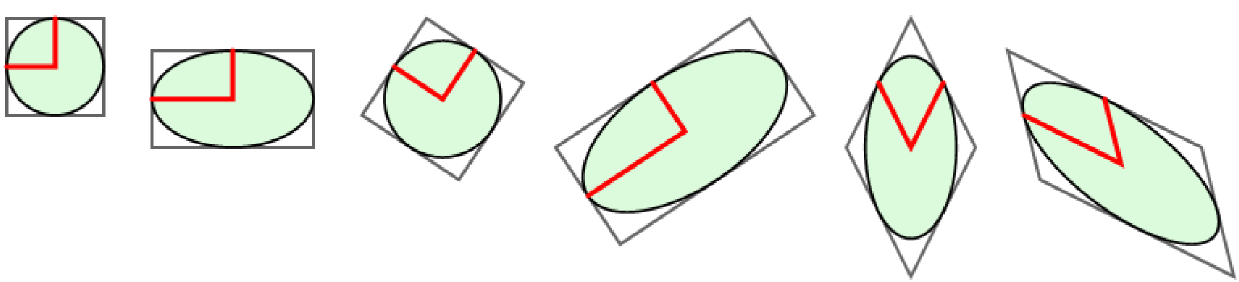
To serve as a smart pointer, the SGPtr class overloads the –> operator so that an SGPtr object, such as `sg`, can be used as a pointer to the encapsulated ShapeGen object. To enable a ShapeGen object pointer to be passed to a function (such as `MySub` in the preceding code example) as a call parameter,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

Ellipses and elliptic arcs

The ShapeGen interface can 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 systems 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.

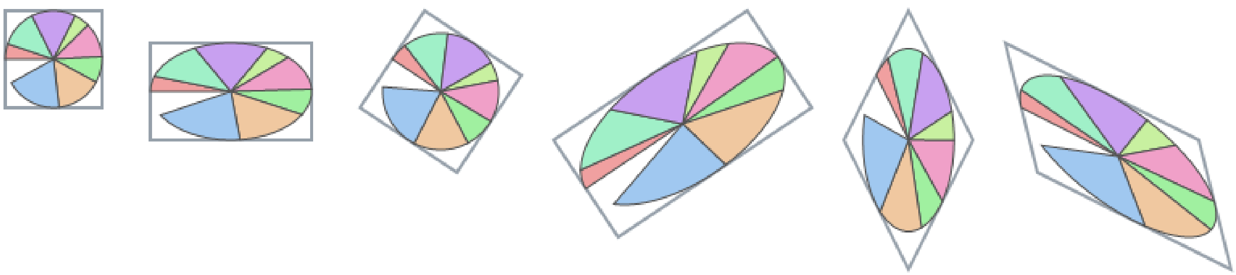
The following screenshot should clarify things a bit.



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 preceding screenshot 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, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    const float PI = 3.14159265;
    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] = {
        { {  20,  80 }, {  20,  20 }, {  80,  20 } },
        { { 110, 100 }, { 110,  40 }, { 210,  40 } },
        { { 240,  80 }, { 280,  20 }, { 340,  60 } },
        { { 400, 160 }, { 360, 100 }, { 480,  20 } },
        { { 540, 100 }, { 580,  20 }, { 620, 100 } },
        { { 660, 120 }, { 640,  40 }, { 760, 100 } },
    };

    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(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;

            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(80,80,80));
            sg->SetLineWidth(1.0);
            sg->StrokePath();
            astart += asweep;
        }
    }
}
```

Renderer

A ShapeGen object must be paired with a renderer so that shapes constructed by the ShapeGen library 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*, which does no antialiasing, is paired with an *antialiasing renderer*. The basic renderer is faster, especially for filling large rectangular areas. The antialiasing renderer depends on the alpha-blending capabilities of the underlying platform. The user calls the `ShapeGen::SetRenderer` function to switch between these two renderers.

For Linux, these two 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.

Instead of passing the trapezoids directly to a renderer to be drawn, ShapeGen passes the trapezoid list to a `ShapeFeeder` object and 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 are fed to a basic renderer (with no antialiasing) have integer width and height, and are typically just one pixel in height unless the source trapezoid happens to have vertical sides.

A subpixel span fed to an antialiasing renderer 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.

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, on 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 *antialiasing* 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 two functions specifically to support antialiasing renderers:

```
class Renderer
{
public:
    virtual void RenderShape(ShapeFeeder *feeder) = 0;
    virtual int QueryYResolution() { return 0; }
    virtual bool SetMaxWidth(int width) { 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 defined in the Renderer base class.

All three of the functions in the `Renderer` base class are called exclusively by the ShapeGen object. Additionally, a renderer derived from this class is expected to provide user-callable functions to, for example, specify the solid color, repeating pattern, or color gradient to be used to render shapes. The example renderers included in this GitHub project do solid-color fills and provide a user-callable `SetColor` function.

An antialiasing renderer's `RenderShape` function receives a series of subpixel spans from the `ShapeFeeder` iterator.

Each of the example antialiasing renderers in the ShapeGen Github project partitions a pixel into 32 subpixels (the pixel is four subpixels high, and eight wide). The renderer constructs an 4x8 coverage bit mask 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.

The antialiasing 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 bounds checking before it is drawn. After you finish debugging your graphics program, though, you can improve its performance by first drawing everything to an offscreen buffer and then blitting the completed image to the window. That's how the SDL2 version of the `RenderShape` function in this project works.

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.

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 Device clipping rectangle.

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 Device clipping rectangle.

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++

```cpp
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 Device clipping rectangle.

Header

`shapegen.h`

See also

`SGCoord`

`ShapeGen::SetFixedBits`

ShapeGen functions

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

ShapeGen::BeginPath function

The `BeginPath` function begins a new path.

Syntax

C++

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

Parameters

None

Return value

None

Remarks

This function discards any existing path, starts a new path, and starts a new, empty figure (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 a value of false. If the macro NDEBUG (used by assert.h) is undefined, the function faults before returning false.

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.

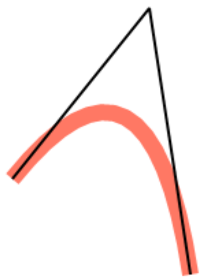
The ShapeGen::PolyBezier2 function constructs a set of connected quadratic Bezier splines in a single function call.

Example

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

```
void example01(SimpleRenderer *rend, SimpleRenderer *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 following screenshot.



In the 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 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 a value of `false`. If the macro NDEBUG (used by `assert.h`) is undefined, the function faults before returning `false`.

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.

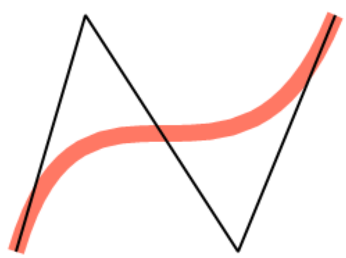
The ShapeGen::PolyBezier3 function constructs a set of connected cubic Bezier splines in a single function call.

Example

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

```
void example02(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint v0 = { 100, 200 }, v1 = { 150, 30 },
            v2 = { 260, 200 }, v3 = { 330, 30 };

    // Draw cubic Bezier spline in red
    aarend->SetColor(RGBX(255,120,100));
    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 following screenshot.



In the code example, 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.

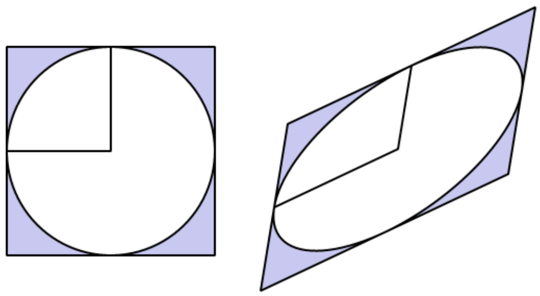
Example

This example uses the `Ellipse` function to draw two ellipses inscribed in parallelograms. (Parameter `rend` points to the basic renderer, `aarend` points to the antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example03(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint u[4] = { { 100, 275 }, { 100, 75 }, { 300, 75 }, };
    SGPoint du[3] = { { 244, 34 }, { 270, 74 }, { 308, -38 }, };
    SGPoint v0, v1, v2;

    sg->SetLineWidth(2.0);
    for (int i = 0; i < 2; ++i)
    {
        u[3].x = u[0].x - u[1].x + u[2].x;
        u[3].y = u[0].y - u[1].y + u[2].y;
        v0.x = (u[0].x + u[2].x)/2;
        v0.y = (u[0].y + u[2].y)/2;
        v1.x = (u[0].x + u[1].x)/2;
        v1.y = (u[0].y + u[1].y)/2;
        v2.x = (u[1].x + u[2].x)/2;
        v2.y = (u[1].y + u[2].y)/2;
        sg->BeginPath();
        sg->Move(u[0].x, u[0].y);
        sg->PolyLine(3, &u[1]);
        sg->CloseFigure();
        sg->Ellipse(v0, v1, v2);
        aarend->SetColor(RGBX(200,200,240));
        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(0,0,0));
        sg->StrokePath();
        for (int j = 0; j < 3; ++j)
        {
```

```
            u[j].x += du[j].x;
            u[j].y += du[j].y;
        }
    }
}
```

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.

In the figure on the right, an affine transformation has been performed on the original points in the figure on the left. The parallelogram on the right is the transformed version of the square on the left. The center point and conjugate diameter end points are also transformed. The resulting ellipse is inscribed in the parallelogram and touches the parallelogram at the midpoint of each side.

Header

`shapegen.h`

See also

`SGPoint`

`ShapeGen::EndFigure`

`ShapeGen::CloseFigure`

ShapeGen::EllipticArc function

The `EllipticArc` function adds an elliptic arc to the current path.

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 swept out 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.

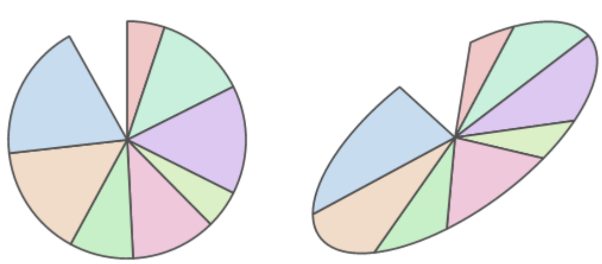
For more information about conjugate diameter end points, see Ellipses and Elliptic Arcs.

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 antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example04(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    const float PI = 3.14159265;
    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 v0[] = { { 200, 175 }, { 476, 173 } };
    SGPoint v1[] = { { 200,  75 }, { 489,  93 } };
    SGPoint v2[] = { { 300, 175 }, { 595, 117 } };

    sg->SetLineWidth(1.7);
    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(v0[i], v1[i], v2[i], astart, asweep);
            sg->Line(v0[i].x, v0[i].y);
            aarend->SetColor(color[j]);
            sg->CloseFigure();
            sg->FillPath(FILLRULE_EVENODD);
            aarend->SetColor(RGBX(80, 80, 80));
            sg->StrokePath();
            astart += asweep;
        }
    }
}
```

The result is shown in the following screenshot.



The two ellipses in the screenshot differ only in the three points – `v0`, `v1`, and `v2` – that are passed to the `EllipticArc` function. The `astart` and `asweep` angles are identical. That's because these angles remain invariant to affine transformations of `v0`, `v1`, and `v2`.

If we needed obtain the x-y coordinates at the starting and ending points of each arc in the preceding code example, we could insert calls to the `ShapeGen::GetFirstPoint` and `ShapeGen::GetCurrentPoint` functions 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 of 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 a value of `false`. If the macro NDEBUG (used by `assert.h`) is undefined, the function faults before returning `false`.

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.

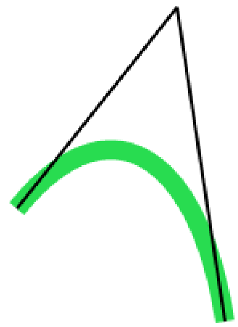
The ShapeGen::PolyEllipticSpline function constructs a set of connected elliptic splines in a single function call.

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 antialiasing renderer, clip specifies the device clipping rectangle, and variable sg is the ShapeGen object pointer.)

```
void example05(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint v0 = { 100, 200 }, v1 = { 200, 75 }, v2 = { 230, 270 };

    // 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 following screenshot.



In the 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 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++

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

Parameters

bbox

A pointer to a caller-supplied `SGRect` structure. The function writes the minimum bounding box coordinates to this structure. This pointer can be null (zero) if the caller simply wants to know whether the current path is empty.

Return value

Returns `true` if the current path is not empty. If the path is empty, the function immediately returns a value of `false` 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.

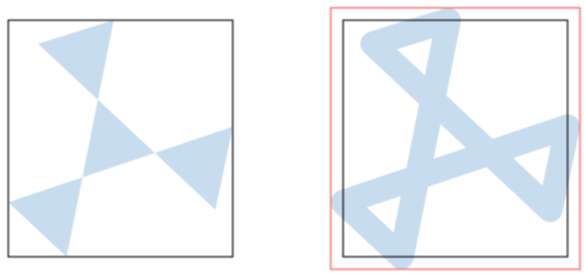
Example

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

```
void example06(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint xy[] = {
        { 104, 62 }, { 247, 196 }, { 261, 129 },
        { 80, 190 }, { 127, 234 }, { 165, 43 }
    };
    float linewidth = 20.0;
    SGRect bbox;

    // Fill shape in blue
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
    sg->PolyLine(5, &xy[1]);
    aarend->SetColor(RGBX(200,220,240));
    sg->FillPath(FILLRULE_EVENODD);

    // Get bounding box and outline it in black
    sg->GetBoundingBox(&bbox);
    sg->SetLineWidth(1.0);
    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 += 270;

    // Stroke the figure in 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(RGBX(200,220,240));
    sg->StrokePath();

    // Get the bounding box and outline it in black
    sg->GetBoundingBox(&bbox);
    sg->SetLineWidth(1.0);
    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 modified 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 joints.

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 latest point appended to the current figure (aka subpath) in the current path.

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 currently open figure (aka subpath) in the current path.

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++

```
void 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

None

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. 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 a value of `false`. If the macro NDEBUG (used by `assert.h`) is undefined, the function faults before returning `false`.

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.

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.

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

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 a value of false. If the macro NDEBUG (used by assert.h) is undefined, the function faults before returning false.

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.

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

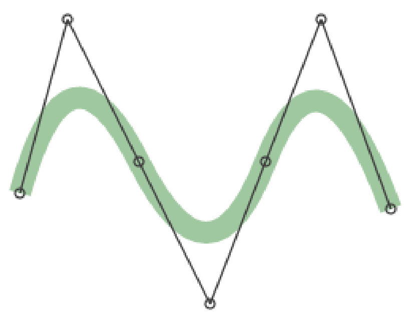
Example

This example uses the PolyBezier2 function to draw three connected quadratic Bezier spline curves. (Parameter rend points to the basic renderer, aarend points to the antialiasing renderer, clip specifies the device clipping rectangle, and variable sg is the ShapeGen object pointer.)

```
void example07(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint xy[] = {
        { 40, 140 }, { 70, 30 }, { 115, 120 }, { 160, 210 },
        { 195, 120 }, { 230, 30 }, { 274, 150 }
    };
    SGPoint v0, v1, v2;

    // Stroke three connected quadratic Bezier splines in green
    aarend->SetColor(RGBX(160,200,160));
    sg->SetLineWidth(14.0);
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
    sg->PolyBezier2(6, &xy[1]);
    sg->StrokePath();
```

```
    // Outline spline skeleton in black; mark knots & control points
    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)
    {
        v0 = v1 = v2 = xy[i];
        v1.x -= 3;
        v2.y -= 3;
        sg->Ellipse(v0, v1, v2);
    }
    sg->StrokePath();
}
```

The result is shown in the following screenshot.



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 a value of `false`. If the macro NDEBUG (used by `assert.h`) is undefined, the function faults before returning `false`.

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.

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

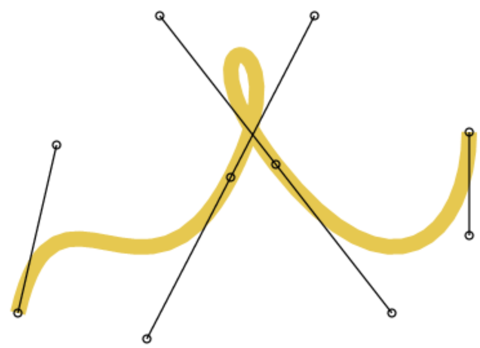
Example

This example uses the Poly`Bezier3` function to draw three connected cubic Bezier spline curves. (Parameter `rend` points to the basic renderer, `aarend` points to the antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example08(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint xy[] = {
        { 30, 250 }, { 60, 120 }, { 130, 270 }, { 195, 145 }, { 260, 20 },
        { 140, 20 }, { 230, 135 }, { 320, 250 }, { 380, 190 }, { 380, 110 }
    };
    SGPoint v0, v1, v2;
    int i;

    // Stroke three connected cubic Bezier splines in yellow
    aarend->SetColor(RGBX(230,200,80));
    sg->SetLineWidth(12.0);
    sg->BeginPath();
    sg->Move(xy[0].x, xy[0].y);
    sg->PolyBezier3(9, &xy[1]);
    sg->StrokePath();
```

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

The result is shown in the following screenshot.



The three connected splines are stroked in yellow, starting from the left. 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 (π/2-radian elliptic arcs), 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 a value of false. If the macro NDEBUG (used by assert.h) is undefined, the function faults before returning false.

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.

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

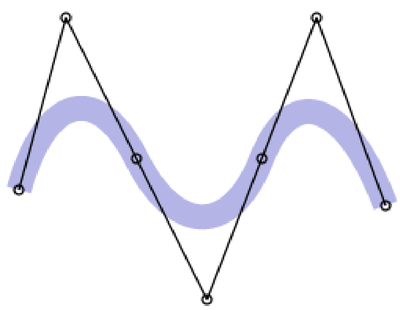
Example

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 antialiasing renderer, clip specifies the device clipping rectangle, and variable sg is the ShapeGen object pointer.)

```
void example09(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint xy[] = {
        { 40, 140 }, { 70, 30 }, { 115, 120 }, { 160, 210 },
        { 195, 120 }, { 230, 30 }, { 274, 150 }
    };
    SGPoint v0, v1, v2;

    // Stroke 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 spline skeleton in black; mark knots & control points
    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)
    {
        v0 = v1 = v2 = xy[i];
        v1.x -= 3;
        v2.y -= 3;
        sg->Ellipse(v0, v1, v2);
    }
    sg->StrokePath();
}
```

The result is shown in the following screenshot.



The three connected splines are stroked in blue, 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::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 a value of `false`. If the macro NDEBUG (used by `assert.h`) is undefined, the function faults before returning `false`.

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.

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:

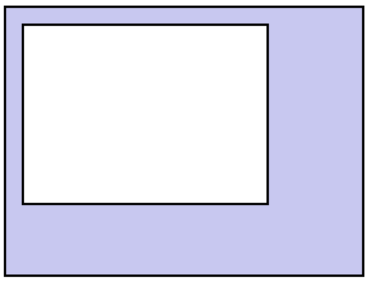
- `rect.w > 0 and rect.h > 0`
- `rect.x is the rectangle's left edge, and rect.y is the top edge`

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 antialiasing renderer, clip specifies the device clipping rectangle, and variable sg is the ShapeGen object pointer.)

```
void example10(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(rend, clip);
    SGRect rect = { 100, 75, 300, 225 };

    sg->BeginPath();
    sg->Rectangle(rect);

    // Make the second rectangle smaller than the first
    rect.x += 15;
    rect.y += 15;
    rect.w -= 95;
    rect.h -= 75;

    // Modify the parameters so that construction
```

```
        // of the second rectangle proceeds in the
        // CCW direction
        rect.y += rect.h;
        rect.h = -rect.h;
        sg->Rectangle(rect);
        rend->SetColor(RGBX(200,200,240));
        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 following screenshot.



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 user can by call the `ShapeGen::InitClipRegion` function to change the width and height of the device clipping rectangle. The user can call the `ShapeGen::SetScrollPosition` function to change the position of the device clipping rectangle in ShapeGen coordinate space. 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 `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 replace the corners of the rectangle with circular arcs, set `round.x` = *r* and `round.y` = *r*, where *r* is 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:

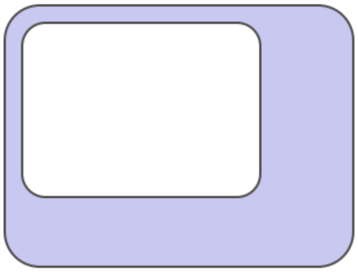
- `rect.w > 0 and rect.h > 0`
- `rect.x is the rectangle's left edge, and rect.y is the top edge`
- `round.x > 0 and round.y > 0`

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 antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example11(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(rend, clip);
    SGRect rect = { 100, 75, 300, 225 };
    SGPoint round = { 30, 30 };
```

```
      // Construct outer rounded rectangle
      sg->BeginPath();
      sg->RoundedRectangle(rect, round);

      // Make the second rectangle smaller than the first
      rect.x += 15;
      rect.y += 15;
      rect.w -= 95;
      rect.h -= 75;
      round.x = round.y -= 10;

      // Modify the parameters so that construction
      // of the inner rounded rectangle proceeds
      // in the CCW direction
      rect.y += rect.h;
      rect.h = -rect.h;
      round.y = -round.y;
      sg->RoundedRectangle(rect, round);
      rend->SetColor(RGBX(200,200,240));
      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 following screenshot.



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 that is then intersected with the current 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 antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

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

```
    const float PI = 3.14159265;
    const float t = 0.8*PI;
    const float sint = sin(t);
    const float cost = cos(t);
    const int xc = 160, yc = 150;
    int xr = -105, yr = 0;
    SGRect rect = { 50, 50, 200, 200 };

    // Draw a square filled with a light blue color
    sg->BeginPath();
    sg->Rectangle(rect);
    rend->SetColor(RGBX(220,240,255));
    sg->FillPath(FILLRULE_EVENODD);

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

    // 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(238,50,50));
    sg->SetLineWidth(1.0);
    sg->BeginPath();
    for (int y = 50; y < 254; y += 4)
    {
        sg->Move(50, y);
        sg->Line(250, 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 this path to form a new, star-shaped clipping region. Finally, a series of horizontal lines (shown in red) are constructed 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. If the macro NDEBUG (used by `assert.h`) is undefined, the function faults before returning -1.

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 a 32-bit value in which the 16 LSBs are interpreted as a fraction.

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 curve-to-chord error tolerance.

Syntax

C++

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

Parameters

`flatness`

The maximum curve-to-chord distance, measured in pixels. Set this parameter to a value in the range 1.0/16.0 to 16.0.

Return value

Returns the previous flatness setting.

Remarks

ShapeGen approximates curves and arcs with connected straight line segments, or 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, more accurate approximations to 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 1.0/16.0 to 16.0 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++

```
void 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.

`mult`

The dash-length multiplier.

Return value

None

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 15 elements, not counting the terminating zero. A `dash` array longer than this maximum is quietly truncated to 15 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).

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 antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example13(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    const float PI = 3.14159265;
    SGPoint xy[] = {
        {  86, 192 }, {  86, 153 }, {  40, 192 }, { 140, 230 },
        { 140, 184 }, { 140, 153 }, {  71,  92 }, { 117,  32 }
    };
    float linewidth = 6.0;
    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 };

    aarend->SetColor(RGBX(205, 92, 92));
    sg->SetLineWidth(linewidth);
    for (int i = 0; i < 4; ++i)
    {
        sg->SetLineDash(pattern[i], 0, linewidth/2.0);
        sg->BeginPath();
        sg->EllipticArc(xy[0], xy[1], xy[2], 0, PI);
        sg->PolyEllipticSpline(4, &xy[3]);
        sg->Line(xy[7].x, xy[7].y);
        sg->StrokePath();
        for (int j = 0; j < 8; ++j)
            xy[j].x += 150;
    }
}
```

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 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 constructed by the `ShapeGen::StrokePath` function.

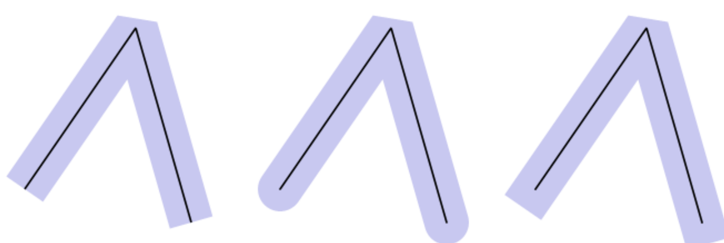
The default value for the line-end attribute is LINEEND_FLAT.

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 antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example14(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    LINEEND cap[] = { LINEEND_FLAT, LINEEND_ROUND, LINEEND_SQUARE };
    SGPoint vert[] = { { 70, 240 }, { 170, 95 }, { 220, 270 } };

    for (int i = 0; i < 3; ++i)
    {
        sg->BeginPath();
        sg->Move(vert[0].x, vert[0].y);
        sg->PolyLine(2, &vert[1]);
        sg->SetLineWidth(40.0);
        sg->SetLineEnd(cap[i]);
        aarend->SetColor(RGBX(200,200,240));
        sg->StrokePath();
        sg->SetLineWidth(1.7);
        aarend->SetColor(RGBX(0,0,0));
        sg->StrokePath();
        for (int j = 0; j < 3; ++j)
            vert[j].x += 230;
    }
}
```

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 values:

LINEJOIN_BEVEL – Beveled joint

LINEJOIN_ROUND – Rounded joint

LINEJOIN_MITER – Mitered joint

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.

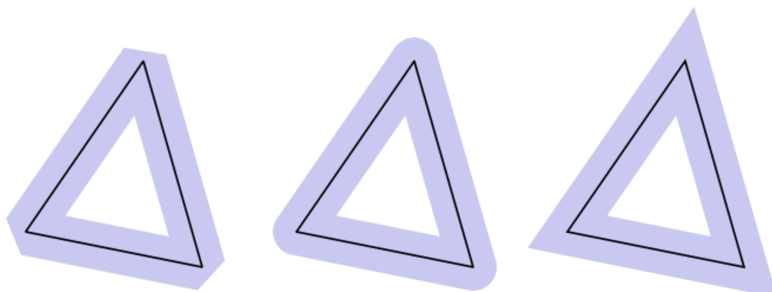
Example

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 antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example15(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    LINEJOIN join[] = { LINEJOIN_BEVEL, LINEJOIN_ROUND, LINEJOIN_MITER };
    SGPoint vert[] = { { 70, 240 }, { 170, 95 }, { 220, 270 } };

    for (int i = 0; i < 3; ++i)
    {
        sg->BeginPath();
        sg->Move(vert[0].x, vert[0].y);
        sg->PolyLine(2, &vert[1]);
        sg->SetLineWidth(40.0);
```

```
        sg->SetLineJoin(join[i]);
        sg->CloseFigure();
        aarend->SetColor(RGBX(200,200,240));
        sg->StrokePath();
        sg->SetLineWidth(1.7);
        aarend->SetColor(RGBX(0,0,0));
        sg->StrokePath();
        for (int j = 0; j < 3; ++j)
            vert[j].x += 230;
    }
}
```

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 joint 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 joints 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 that is masked off from the current 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.

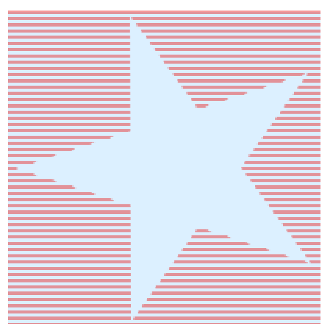
Example

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 antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example16(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(rend, clip);
    const float PI = 3.14159265;
    const float t = 0.8*PI;
    const float sint = sin(t);
    const float cost = cos(t);
    const int xc = 160, yc = 150;
    int xr = -105, yr = 0;
    SGRect rect = { 50, 50, 200, 200 };

    // Draw a square filled with a light blue color
    sg->BeginPath();
    sg->Rectangle(rect);
    rend->SetColor(RGBX(220,240,255));
    sg->FillPath(FILLRULE_EVENODD);

    // Switch to antialiasing renderer
    sg->SetRenderer(aarend);

    // Mask off 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->SetMaskRegion(FILLRULE_WINDING);

        // Draw a series of horizontal, red lines through the square
        aarend->SetColor(RGBX(234,50,50));
        sg->SetLineWidth(1.0);
        sg->BeginPath();
        for (int y = 50; y < 254; y += 4)
        {
            sg->Move(50, y);
            sg->Line(250, 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 lines (shown in red) are constructed through the blue square, but only the part of each line that lies 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 joint can reach before the point is automatically beveled off.

Syntax

C++

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

Parameters

`mlim`

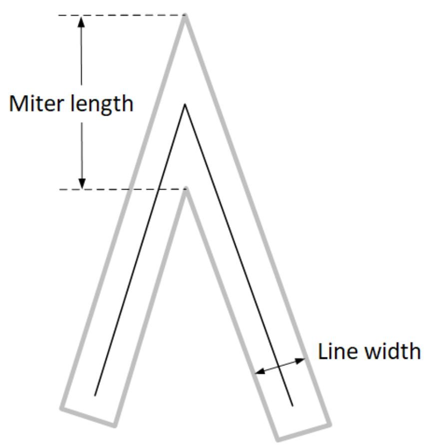
The new miter-limit setting. Set this parameter to a value in the range 1.0 to 100.0.

Return value

Returns the previous miter-limit setting.

Remarks

This function specifies the miter limit, which determines the maximum length of a mitered joint 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 joint that exceeds this limit, turning it into a beveled joint whose length matches the max_miter_length value calculated above.

The default miter-limit setting is 10.0.

A miter-limit setting of 1.0 specifies that miter joints at all angles are to be beveled.

If the caller specifies an `mlim` parameter value that is outside the range 1.0 to 100.0, the function quietly clamps the value to this range.

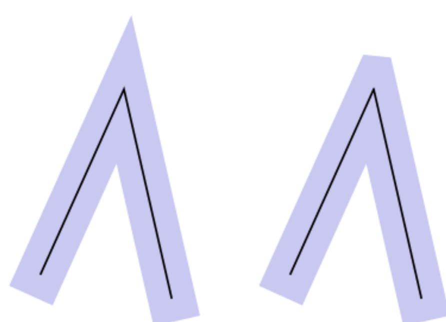
To specify that stroked paths are to be constructed with miter joints, call the `ShapeGen::SetLineJoin` function with `joinstyle` = LINEJOIN_MITER. By default, stroked paths are constructed with beveled joints.

Example

This example uses the `SetMiterLimit` function to set different miter limits for two mitered joints. (Parameter `rend` points to the basic renderer, `aarend` points to the antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

```
void example17(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(aarend, clip);
    SGPoint vert[] = { { 100, 250 }, { 170, 95 }, { 210, 270 } };

    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(40.0);
        aarend->SetColor(RGBX(200, 200, 240));
        sg->StrokePath();
        sg->SetLineWidth(1.7);
        aarend->SetColor(RGBX(0,0,0));
        sg->StrokePath();
        sg->SetMiterLimit(1.4);
        for (int j = 0; j < 3; ++j)
            vert[j].x += 210;
    }
}
```

The result is shown in the following screenshot.



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

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

Parameters

rend

A pointer to a `Renderer` object. This parameter must be nonnull. If the `rend` parameter is zero and the macro NDEBUG (used by `assert.h`) is undefined, the function faults.

Return value

None

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 antialiasing renderer, `clip` specifies the device clipping rectangle, and variable `sg` is the ShapeGen object pointer.)

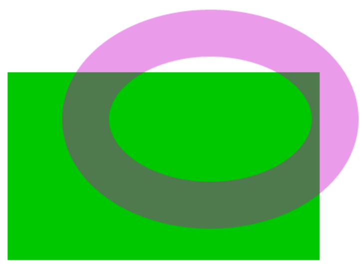
```
void example18(SimpleRenderer *rend, SimpleRenderer *aarend, const SGRect& clip)
{
    SGPtr sg(rend, clip);
    SGRect rect = { 100, 80, 400, 240 };
    SGPoint v0 = { 360, 140 }, v1 = { 360+160, 140 }, v2 = { 360, 140+110 };
```

```
        // Use the basic renderer to fill a green rectangle
        sg->BeginPath();
        sg->Rectangle(rect);
        rend->SetColor(RGBX(0,200,0));  // green (opaque)
        sg->FillPath(FILLRULE_EVENODD);

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

        // Alpha-blend a magenta, stroked ellipse with the rectangle
        sg->BeginPath();
        sg->Ellipse(v0, v1, v2);
        aarend->SetColor(RGBA(200,0,200,100));  // magenta + alpha
        sg->SetLineWidth(60.0);
        sg->StrokePath();
    }
```

In this example, the basic renderer, `rend`, is passed as an input parameter to SGPtr constructor, which creates a ShapeGen object and installs `rend` as this object's initial renderer. Using this renderer, a rectangle is painted green. 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 green rectangle.

The result is shown in the following screenshot.



This example uses the RGBA macro (defined in demo.h) to construct a 32-bit RGBA pixel value with an alpha channel value of 100.

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 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. 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.

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, the arrow keys generate calls to this function. For the Win32 version, the thumb tabs on the window scroll bars generate `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 joint style	Beveled joint	`ShapeGen::SetLineJoin`
Line width	4.0	`ShapeGen::SetLineWidth`
Miter limit	10.0	`ShapeGen::SetMiterLimit`

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

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

Header

`shapegen.h`

See also

`ShapeGen::SetLineDash`

`ShapeGen::SetLineEnd`

`ShapeGen::SetLineJoin`

`ShapeGen::SetLineWidth`

`ShapeGen::SetMiterLimit`

`ShapeGen::EndFigure`

`ShapeGen::CloseFigure`

`ShapeGen::FillPath`

ShapeGen::SwapClipRegion function

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

Syntax

C++

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

Parameters

None

Return value

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

Remarks

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

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

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

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

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

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

Header

`shapegen.h`

See also

`ShapeGen::SaveClipRegion`

`ShapeGen::ResetClipRegion`

`ShapeGen::SetClipRegion`

`ShapeGen::SetMaskRegion`

`ShapeGen::InitClipRegion`

`ShapeGen::SetScrollPosition`

ShapeGen::SetRenderer