Sigmapi Design Drawing Engine

Siniša Petrić

February 3, 2017

Contents

1 Introduction

1.1 Installation

Sigmapi Design Drawing Engine (spEngine) is simple DLL library written in Embarcadero C++ and is intended for Embarcadero C++/Delphi developers. It can work with any kind of image container with known stride (alignment). Often, Delphi programmers use TBitmap object as image container, however, in this manual we will use TIEBitamp container from ImageEn VCL (www.imageen.com). Installation is straight forward: all you need is to include following files into your main program:

- C++:
  - spEngine.lib: static library that must be linked with your code. 64-bit static library has extension "a" (spEngine.a).
  - spGlobals.h: global spEngine constants (included in spEngineDll.h).
  - spEngineDll.h: structures and API definitions (header must be included in your main program using #include).
- Delphi::
  - spGlobals.pas: global spEngine constants (included in spEngineDll.pas).
  - spEngineDll.pas structures and API definitions (must be included in your main program inside uses block).

Besides that, spEngine.dll must be copied in the directory where your main program's exe resides.

1.2 Library format

spEngine.dll comes in two versions X32 and X64. 32-bit dll version is in OMF format, while 64-bit vestion is in ELF format.

1.3 Concept

spEngine accepts scanline(s) from some user created image container (TBitmap, TIEBitmap, or any other) to perform various types of rendering/drawing onto image. Image scanlines are shared inside speEngin's drawing surface. Besides shared original image, internally, another copy of the same image is created, together with two grayscale masks. All user

images (surfaces) are kept inside Surface container, which is the main part of spEngine. Generally, spEngine has four distinctive modules:

- Surface container: the core part of spEngine. Keeps tracking of various surfaces (images).
- Renderer: rendering engine. In current version, only drawing is implemented.
- Brush: module that takes care of various user created/selected brushes.
- Shape: module used to create/select various shape types (graphics primitives).

Note: speEngine only works with 24-bit images (3 channels RGB) with or without alpha mask (8-bit, one channel). Alpha mask address is passed as separate parameter when surface is constructed, i.e. 32-bit images (RGBA) are not currently supported. User must supply alpha channel as separate gray-scale image.

1.4 Version history

spEnigne 0.6

- Changes in solid brush specific structure/record:
  - **Shape** parameter can have values 0,1,2 (circle, square, triangle).
  - **Filled** parameter added: true by default. If set to false, only shape border will be drawn with desired Thickness.
  - **Angle** parameter added: angle of shape (triangle only). Angle is given in degrees.
  - **RandomAngle** parameter added: *true* by default - a random angle $[0, 360)$ will be picked. If set to *false*, **Angle** value is used.
- New functions added:
  - **spDeleteCloneImage**: delete clone image container.
  - **speCloneSelf**: use image you're drawing on as clone image.
  - **spSetSelection**: set selection mask (8-bit singla channel image).
  - **spDeleteSelection**: delete selection mask container.
  - **spSelectionClip**: set selection clipping on/off.
- Tpx_SelectionParams structure/record removed.
- Change in **Tpx_BrushSpraySpecific** structure/record: **Gravity** parameter type changed from *int (Integer)* to *float (Single)*.
- Brush **median** filter implemented.

spEngine 0.5:

- Corrected bug when drawing on images with alpha channel with capacity $< 100\%$: brush capacity was always 100% on arease with alpha channel > 0.
- Added two new functions for setting clone image.
- Scratch option enabled.

spEngine 0.4:

- Corrected bug in speSetSurface when previously created surface is replaced with new one (program crash).
- Brush (stroke) envelope added: check Envelope... parameters in common brush structure/record. Envelope string is set by two new APIs: speSetSizeEnvelope, speSetCapacityEnvelope.
- speSetScaleColors API added: used to set top and bottom color values for normalization.
- Various arithmetic modes available. Note: modes px_ARITH_BUMP, px_ARITH_BUMP_COLOR are intended for so called "pixels shifting" rendering mode which is not currently implemented.
- Corrected bug when drawing on image with alpha channel.
- New drawing parameter added in DrawParams structure/record - DrawOnAlpha (Boolean): if this parameter is set to **true**, drawing will occure only on areas with alpha > 0. Default value of DrawOnAlpha is **false**.
- Corrected bug when drawing shapes programmatically: current shape (set with speSetShape) was not restored.
- MainMode parameter name in DrawParams structure changed to ArithMode.
- Corrected bug when SizeVary > 0: internal brush memory was not properly cleared.

spEngine 0.3:

- DrawTipOnMouseDown parameter added in RenderParams (default = false). When set to true and "free hand" shape is selected, drawing starts in speHandMouseDown API (brush tip will be drawn). Otherwise, drawing starts in speHandMouseMove API.
- Corrected bug in speHandMouseUp API (internal masks ROIs were not cleared).

spEngine 0.2:

- initial version.

2 Initialization and surface container

2.1 Initialization

Before starting to play with spEngine, initialization API must be invoked: **speInit** function must be called once at the beginning of your main program. speInit API accepts one parameter of type Tpx_RenderData structure(record) and it's called by reference:

```
//C++:
bool __stdcall speInit(Tpx_RenderData &rd);
//Delphi:
function speInit(var rd: Tpx_RenderData): Boolean stdcall;
```

speInit API will return true if everything passed OK and your structure (record) will be filled with default render data values. Tpx_RenderData structure is nested structure that contains following structures (records):

- RenderParams (type Tpx_RenderParams) - contains main rendering parameters.
- DrawParams (type Tpx_DrawParams) - contains main drawing parameters.
- ShadowParams (type Tpx_ShadowParams) - contains parameters that control drop shadow behavior.

To check current version of the engine, use speGetVersion function:

```
//C++:
wchar_t* __stdcall speGetVersion(void);
//Delphi:
function speGetVersion: PWideChar stdcall;
```

2.2 Surface container

Once the initialization is invoked, you must add/create drawing surface from your image container. **speAddSurface** API creates empty slot inside surface container and returns current surface index:

```
//C++:
int __stdcall  speAddSurface(void);
//Delphi:
int __stdcall  speAddSurface(void);
```

To set/create surface related to newly created surface slot, spEngine uses two different approaches:

- **speSetSurface**: if your image container is contiguous (i.e. scanlines are not fragmented):

```
//C++:
bool __stdcall speSetSurface(void *scanOrig, void *scanAlphaOrig,
                        int width, int height,
                        unsigned int scanlineAlignment);
//Delphi:
function speSetSurface(scanOrig: Pointer;
                       scanAlphaOrig: Pointer;
                       width: Integer; height: Integer;
                       scanlineAlignment: Cardinal): Boolean stdcall;
```

- **speSurfaceBegin**/ **speSurfaceAddScanline**/ **speSurfaceFinish** combination: if your image container is not contiguous (i.e. scanlines are fragmented):

```
//C++:
bool __stdcall speSurfaceBegin(int width, int height);
bool __stdcall speSurfaceAddScanline(void *scanOrig, void *scanAlphaOrig);
bool __stdcall speSurfaceFinish(void);
//Delphi:
function speSurfaceBegin(width: Integer;
                         height: Integer): Boolean stdcall;
function speSurfaceAddScanline(scanOrig: Pointer;
                                                  scanAlphaOrig: Pointer):
                                                  Boolean stdcall;
function speSurfaceFinish: Boolean stdcall;
```

So, an example of initialization and adding/setting some contiguous surface in C++ should look like:

```
//C++ example:
// public or private variable
Tpx_RenderData renderData;
// this two lines are usually added in form's constructor:
// all TIEBitmaps are contiguous
IEGlobalSettings()->AutoFragmentBitmap = false;
// initialize spEnigne and fetch default render data
speInit(renderData);
.
.
// Let's assume that some 24-bit RGB map of type TIEBitmap is already created
int h = map->Height;
int w = map->Width;
// add empty surface to surface container
speAddSurface();
// set surface data
speSetSurface(map->ScanLine[h -1], 0, w, h, 4);
// Surface is ready for drawing.
```

In the example above, there is no alpha map so we are setting scanAlphaOrig pointer to 0. Also, we assume that scanlines are aligned to double word boundary and we set parameter scanlineAlignemnt to 4 bytes. An example of adding/setting non-contiguous surface in Delphi should look like:

```
// Delphi example:
//public or private variable
var
  renderData: Tpx_RenderData;
// this two lines are usually added in form's constructor:
// all TIEBitmaps are non-contiguous (fragmented)
IEGlobalSettings().AutoFragmentBitmap := True;
speInit(renderData);
.
.
// Let's assume that some 24-bit RGB map of type TIEBitmap is already created
var h, w, i: Integer;
h := map.Height;
w := map.Width;
// add empty surface to surface container
speAddSurface;
// prepare surface for scanline addition
speSurfaceBegin(w, h);
// add all scanlines to the surface
for i := 0 to h-1 do
   speSurfaceAddScanline(map.ScanLine[i], nil);
// finish scanline addition
speSurfaceFinish;
// Surface is ready for drawing.
```

Again, in the example above, there is no alpha map, so we are passing nil as scanAlphaOrig parameter. Because image is fragmented, we don't need scanlineAlignement information as we are simply storing each scanline pointer into internal surface scanlines array. If we work with more then one surface (bitmap) inside surface container, we need to select surface (set current drawing surface) using speSelectSurface API:

```
//C++:
bool __stdcall speSelectSurface(int sIndex);
//Delphi:
speSelectSurface(sIndex: Integer): Boolean stdcall;
```

To delete some surface from surface container, use speDeleteSurface API:

```
//C++:
bool __stdcall speDeleteSurface(int sIndex);
//Delphi:
speDeleteSurface(sIndex: Integer): Boolean stdcall;
```

Note: when some surface with index sIndex is deleted (removed) from surface container, current surface becomes the surface with index = sIndex + 1. If such surface does not exist, current index is set to 0. If surface container is empty after surface deletion, any call to drawing APIs may result in program crash, so keep that in mind.

3 Shapes, brushes and rendering types

3.1 Shapes

Currently, 13 shapes are supported and are defined as enum type in spGlobals.h (C++) and spGlobals.pas (Delphi) :

- px_shpFreeHand - free hand drawing.
- px_shpLine - straight line.
- px_shpEllipse - ellipse outlined.
- px_shpEllipseFilled - filled ellipse, outlined with current stroke (brush).
- px_shpEllipseFilledNB - filled ellipse without outline (border). NB stands for no border.
- px_shpRectangle - rectangle outlined.
- px_shpRectangleFilled - filled rectangle, outlined with current stroke (brush).
- px_shpRectangleFilledNB - filled rectangle without outline (border). NB stands for no border.
- px_shpPolyline - polyline.
- px_shpPolygon - polygon outlined (closed polyline).
- px_shpPolygonFilled - filled polygon, outlined with current stroke (brush).
- px_shpPolygonFilledNB - filled polygon without outline (border). NB stands for no border.
- px_shpSplineCR - smooth polyline (Catmul-Rom spline).

In demo program, distributed together with spEngine.dll, shape is selected by right mouse click on image view: a pop-up window with available shapes is displayed. Initially, shape is set to free hand. To set current shape use speSetShape API:

```
//C++:
bool __stdcall speSetShape(TspeShapeType shapeType);
//Delphi:
function speSetShape(shapeType: TspeShapeType): Boolean stdcall;
```

When some shape is selected/set it becomes current shape for all surfaces inside surface container.

3.2 Brushes

Currently, 9 brushes (brush tips) are supported:

- px_brshNeon - neon type brush tip with various modes of intensity falloff from the center of the brush tip.
- px_brshSolid - solid brush tip.
- px_brshSpray - spray style brush tip.
- px_brshStar - star style brush tip (a bunch of lines from center).
- px_brshHair - hair style brush tip.
- px_brshMesh - random lines mesh.
- px_brshParallel - brush tip with parallel lines.
- px_brshStamp - brush tip that uses grayscale image as brush tip (alpha channel).
- px_brshCml - coupled map lattice (cellular automata) brush tip.

All brushes have brush common and brush specific parameters (stamp brush does not have specific parameters). Brush common parameters do not depend on brush type and are used to specify brush tip size, capacity, step, colors, stroke rules, etc. Brush specific parameters depend on selected brush type. Brushes are stored in brush bucket. which can hold up to 9 brushes. This comes handy when user performs free hand drawing and wants to change brush tip on the fly. When some brush is selected/set it becomes current brush for all surfaces inside surface container. To get/set brush parameters use following APIs:

```
//C++:
bool __stdcall speGetBrushCommon(Tpx_BrushCommonParams &par);
bool __stdcall speSetBrushCommon(Tpx_BrushCommonParams &par);
//Delphi:
function speGetBrushCommon(var par: Tpx_BrushCommonParams):
                        Boolean stdcall;
function speSetBrushCommon(var par: Tpx_BrushCommonParams):
                        Boolean stdcall;
```

To getting/set brush specific following APIs are used:

```
//C++:
void* __stdcall speGetBrushSpecific(void);
bool __stdcall speSetBrushSpecific(void *bsp);
//Delphi:
function speGetBrushSpecific: Pointer stdcall;
function speSetBrushSpecific (vBsp: Pointer) : Boolean stdcall;
```

As each brush specific parameters has it's own structure, when getting brush specific parameters, you mast cast void pointer to respective structure (record). Every time a new brush is created, common and specific brush parameters are set to default values. To select some previously created brush from brush bucket use **speSelectBrush** API:

```
//C++:
bool __stdcall speSelectBrush(int bIndex);
//Delphi:
function speSelectBrush(bIndex: Integer): Boolean stdcall;
```

Stamp brush does not have any specific parameters, but has additional function that sets grayscale image (usually alpha channel) as brush stamp:

```
//C++:
bool __stdcall speSetStamp(void *buffer, int width, int height,
                           unsigned int scanlineAlignement);
//Delphi:
function speSetStamp(buffer: Pointer; width: Integer; height:
                           Integer; scanlineAlignement: Cardinal):
                           Boolean stdcall;
```

As you may notice, speSetStamp API currently accepts only contiguous, non-fragmented images. Usually, user passes last scanline as buffer parameter.

3.3 Rendering types

Currently, only one rendering type is supported (px_rndDraw). More rendering types to come (warp, color, filter 3x3, etc..). To set rendering type, use speSetRender API:

```
//C++:
bool __stdcall speSetRender(TspeRenderType renderType);
//Delphi:
function speSetRender(renderType: TspeRenderType): Boolean stdcall;
```

Besides rendering types, there are four different rendering modes defined in spGlobal.h (spGlobal.pas):

- px_RENDER_AUTOMATIC - default mode: will set rendering mode depending on common brush parameters.
- px_RENDER_SLOW - slow rendering: for every stroke, each brush tip is filtered and rendered stepwise.
- px_RENDER_MEDIUM - medium rendering: complete stroke is filtered and then each brush tip is rendered stepwise.
- px_RENDER_FAST - fast rendering: complete stroke is filtered and rendered at once.
- px_RENDER_DUMMY - no rendering is performed.

Rendering mode is specified in Tpx_RenderData structure under Tpx_RenderParams structure (record):

```
//C++ example:
TpxRenderData renderData;
renderData.RenderParams.RenderingMode = px_RENDER_AUTOMATIC;
```

3.4 Prepare everything for drawing

The best way to see how to deal with all previous APIs is to have a look at demo program source code. Usually, after surface is created and set, we will call a sequence of APIs that look like the code below:

```
//C++ example:
//structures - public or private variables
Tpx_RenderData renderData;
Tpx_BrushCommonParams brushCommonParams;
Tpx_BrushNeonSpecific* neonSpecific;
.
// we have already set some surface!!!
.
speSetShape(px_shpFreeHand);
speSetBrush(px_brshNeon);
// fill brush common params structure with default data:
speGetBrushCommon(brushCommonParams);
// change parameters to user defined values:
brushCommonParams.Size = 60;
brushCommonParams.Capacity = 100;
brushCommonParams.ColorVary = 0; // no color variation
brushCommonParams.SizeVary = 0;  // no size variation
brushCommonParams.CapVary = 0;   // no capacity variation
brushCommonParams.Step = 2;
brushCommonParams.ChalkOn = false;
brushCommonParams.BlurOn = true; // blur on
brushCommonParams.BlurValue = 1; // blur radius
// set standard brush colors
speSetStandardColors((unsigned int)clPurple,
                     (unsigned int)clRed,
                         (unsigned int) clGreen);
// set new brush common parameters
speSetBrushCommon(brushCommonParams);
// get neon brush specific parameters:
neonSpecific = static_cast <Tpx_BrushNeonSpecific*>(speGetBrushSpecific());
// change specific parameters to user defined values:
neonSpecific->FalloffIndex = 0; // linear fall-off
neonSpecific->LinearValue = 80; // 20% solid 80% fall-off
neonSpecific->TrigonometricValue = 32;
// set render type (this can be called only once for complete program)
speSetRender(px_rndDraw);
```

```
// change render data parameters to user defined value:
renderData.RenderParams.RenderingMode = px_RENDER_AUTOMATIC;
renderData.ShadowParams.MainMode = 1; // always set to 1
renderData.RenderParams.DoShadow = true; // perform shadow drop
renderData.ShadowParams.Yoffset = 15;
renderData.ShadowParams.Xoffset = 15;
// set new values
speSetRenderData(renderData);
// we're ready for free hand drawing
```

Delphi programmers can check demo program source code to see how to manipulate various parameters. As you may notice, in previous example, there is a call to **speSetStandardColors** API which is a part of color brush settings APIs and we will explain those APIs in following chapter

4 Colors, textures and color scheme

4.1 Standard colors

To get/set standard brush tip colors we use following APIs:

```
//C++:
bool __stdcall speSetStandardColors(unsigned int primColor,
                        unsigned int secColor,
                        unsigned int polyColor);
bool __stdcall speGetStandardColors(unsigned int &primColor,
                        unsigned int &secColor,
                        unsigned int &polyColor);
//Delphi:
function speSetStandardColors(primColor: Cardinal;
                              secColor: Cardinal;
                              polyColor: Cardinal): Boolean stdcall;
function speGetStandardColors(var primColor: Cardinal;
                              var secColor: Cardinal;
                              var polyColor: Cardinal): Boolean stdcall;
```

Function speSetStandardColors API sets three colors in currently selected brush: primary color, secondary color and polygon filling color.

4.2 Palette

Besides drawing with standard colors there is also possibility to draw with palette color, textures and using pixels from auxiliary image (clone brush):

```
//C++:
bool __stdcall speSetPalette3C(unsigned int c1,
                        unsigned int c2,
                        unsigned int c3);
bool __stdcall speSetPalette255C(unsigned int *pal);
//Delphi:
function speSetPalette3C(c1: Cardinal;
                         c2: Cardinal;
                         c3: Cardinal): Boolean stdcall;
function speSetPalette255C(pal: PCardinal) : Boolean stdcall;
```

Function speSelPalette3C API is used to create palette (255 colors) by smoothly blending three user's supplied colors. Function speSetPalette255C accepts unsigned int pointer to preciously defined palette array of size 255 filled by user (or loaded from a file).

4.3 Textures

Instead of using single color or palette colors, you can relate brush tip to some 24-bit texture:

```
//C++:
bool __stdcall speSetTexture(void *buffer, int width, int height,
                        unsigned int scanlineAlignement);
bool __stdcall speSetPolyTexture(void *buffer, int width, int height,
                        unsigned int scanlineAlignement);
bool __stdcall speDeleteTexture(void);
bool __stdcall speDeletePolyTexture(void);
//Delphi:
function speSetTexture(buffer: Pointer; width: Integer; height: Integer;
                        scanlineAlignement: Cardinal):
                        Boolean stdcall;
function speSetPolyTexture(buffer: Pointer; width: Integer; height: Integer;
                        scanlineAlignement: Cardinal):
                        Boolean stdcall;
function speDeleteTexture: Boolean stdcall;
function speDeletePolyTexture: Boolean stdcall;
```

Again, as with stamp brush, APIs currently accept only contiguous, non-fragmented images. Usually, user passes last scanline as buffer parameter. First function (speSetTexture) sets brush tip texture, while second one (speSetPolyTexture) sets polygon filling texture. Functions speDeleteTexture and speDeletePolyTexture are used to delete texture/polytexture from selected brush.

4.4 Background tiles

There is another type of texture used to simulate various canvases and is called background tile, usually some grayscale seamless texture that can be set/deleted using following APIs:

```
//C++:
bool __stdcall speSetBackgroundTile(void *buffer, int width, int height,
                        unsigned int scanlineAlignement);
bool __stdcall speDeleteBackgroundTile(void);
//Delphi:
function speSetBackgroundTile(buffer: Pointer; width: Integer; height: Integer;
                        scanlineAlignement: Cardinal):
                        Boolean stdcall;
function speDeleteBackgroundTile: Boolean stdcall;
```

Again, as with stamp brush, speSetBackgroundTile function accepts only contiguous, non-fragmented images (bitmaps). User passes last scanline as buffer parameter.

4.5 Color scheme

All APIs described in previous sections are closely related to color scheme which is used to set the way colors are combined/rendered.To set desired color scheme, use following API:

```
//C++:
bool __stdcall speSetColorScheme(const int cScheme);
// Delphi:
function speSetColorScheme(const cScheme: Integer): Boolean stdcall;
```

Color scheme types (indexes) are defined in spGlobals.h (spGlobals.pas):

- px_CTYPE_PRIMARY - draw stroke with primary color.
- px_CTYPE_SECONDARY - draw stroke with secondary color.
- px_CTYPE_MIXED - draw stroke mixing (blending) primary and secondary with respect to brush capacity at given point (alpha blending).
- px_CTYPE_SOLIDMIXED - draw stroke mixing (blending) primary and secondary color with respect to brush capacity at given point, but render resulting color with full capacity.
- px_CTYPE_PALETTE - draw stroke using brush capacity at given pint as palette index [0..255] and perform alpha blending.
- px_CTYPE_SOLIDPALETTE - draw stroke using brush capacity at given point as palette index [0..255] and render resulting palette color with full capacity.
- px_CTYPE_TEXTUREALIGNED - draw stroke using loaded 24-bit texture (usually seamless) and take care of alignment (wrapping) .
- px_CTYPE_TEXTUREFIXED - draw stroke using loaded 24-bit texture (usually seamless) and always set brush tip center to texture center.
- px_CTYPE_CLONEALIGNED - draw stroke using pixels from some auxiliary image (clone brush at selected point) and take care of alignment (wrapping).
- px_CTYPE_CLONEREPOS - as above, but after stroke is completed, reposition clone position: set brush tip center to selected point.
- px_CTYPE_CLONEFIXED - always align brush tip center with selected point.
- px_CTYPE_PALETTE_STEP_C - (step circular) draw stroke using palette, where palette index is increased for each stroke step. When palette index reached maximum (255), index is set to 0.
- px_CTYPE_PALETTE_STEP_FB - (forward-backward) draw stroke using palette as above, but when palette index reaches maximum (255), for each subsequent step, palette index is decreased.

- px_CTYPE_PALETTE_STEP_STOP - (step stop) as above, but when palette index reaches maximum value (255), the rest of stroke is drawn with last color entry (index = 255).
- px_CTYPE_MOUSE_DOWN - color is picked from current image/surface at mouse-down position (x,y) and is used as primary color.
- px_CTYPE_IMAGE_STEP - similar to px_CTYPE_MOUSE_DOWN, but the new color is picked at each stroke step.
- px_CTYPE_ABSQUE - no color, sine colore: this mode has sense when brush bumping is set, otherwise nothing will be rendered.
- px_CTYPE_SIDEKICK - draw stroke using colors from sidekick image. This option is not yet implemented.
- px_CTYPE_GRADIENT_STEP_C - (step circular) similar to px_CTYPE_PALETTE_STEP_C, but instead of palette, primary color-secondary color gradient is used.
- px_CTYPE_GRADIENT_STEP_FB - (forward-backward) similar to px_CTYPE_PALETTE_STEP_FB, but instead of palette, primary color-secondary color gradient is used.
- px_CTYPE_GRADIENT_STEP_STOP - (step stop) similar to px_CTYPE_PALETTE_STEP_STOP, but instead of palette, primary color-secondary color gradient is used.

Note: if you select px_CTYPE_TEXTUREALIGNED color scheme and texture is not set, px_CTYPE_PRIMARY color scheme will be used.

4.6 Shadow color

Last API in this chapter sets shadow color. Drop shadow is a render option that can cast/drop shadow under the last shape drawn:

```
//C++:
bool __stdcall speSetShadowColor(unsigned int sColor);
//Delphi:
function speSetShadowColor(sColor: Cardinal): Boolean stdcall;
```

Here is a Delphi example (see demo program) which shows how to set render parameters to perform shadow casting under drawn shape:

```
//Delphi example:
procedure TForm1.shadowChange(Sender: TObject);
var
  sc: Cardinal;
begin
```

```
renderData.RenderParams.DoShadow := True;
// MainMode parameter value sets the shadow mode.
// Currently, only one mode is available
// and MainMode must be set to 1
renderData.ShadowParams.MainMode := 1;
// x and y offest of the shadow
renderData.ShadowParams.Xoffset := 10;
renderData.ShadowParams.Yoffset := 10;
// blur shadow
renderData.ShadowParams.BlurOn := True;
// blur radius
renderData.ShadowParams.BlurValue  := 3;
// shadow capacity
renderData.ShadowParams.Capacity   := 60;
// set/modify render data
speSetRenderData(renderData);
// set shadow color to black
sc := Cardinal(clBlack);
speSetShadowColor(sc);
end;
```

When ShadowParams structure (record) in render data record is set according to example above, after shape is drawn, the shadow of the shape will be casted/drawn with given color, offset ,blur radius and capacity.

5 Drawing

Finally, how to draw a stroke defined by shape, brush and color onto our image/surface/-canvas? We will first show how to perform free hand drawing.

5.1 Free hand

To perform free hand drawing we must engage mouse events, namely: OnMouseDown, OnMouseMove and OnMouseUp. Each event is connected to respective API:

```
//C++:
bool __stdcall speHandMouseDown(TShiftState Shift, int X, int Y);
bool __stdcall speHandMouseMove(TShiftState Shift, int X, int Y);
bool __stdcall speHandMouseUp(TShiftState Shift, int X, int Y);
//Delphi:
function speHandMouseDown(Shift: TShiftState;
                          X: Integer;
                          Y: Integer): Boolean stdcall;
function speHandMouseMove(Shift: TShiftState;
                          X: Integer;
                          Y: Integer): Boolean stdcall;
function speHandMouseUp(Shift: TShiftState;
                        X: Integer;
                        Y: Integer): Boolean stdcall;
```

Although speHandMouseXXX function keeps track if left-mouse button is down or not, in order to make life easier (especially for debugging purposes) we can add a Boolean variable which will tell us if left mouse button is down or not, so we don't need to call speHandMouseMove function unnecessary. So here is an example of free-hand drawing using OnMouseXXX events:

```
//C++ example:
// PaintView is of type TImageEnVect and surface is created from it's IEBitmap.
// TImageEnVect onMouseDown event:
void __fastcall TForm1::PaintViewMouseDown(TObject *Sender,
                        TMouseButton Button, TShiftState Shift,
                        int X, int Y)
 {
 if (Button != mbLeft)
    return;
 _mouseDown = true;
 X = PaintView->CurrentLayer->ConvXScr2Bmp(X);
 Y = PaintView->CurrentLayer->ConvYScr2Bmp(Y);
```

```
 speHandMouseDown(Shift, X, Y);
 }
// TImageEnVect onMouseMove event:
void __fastcall TForm1::PaintViewMouseMove(TObject *Sender,
                        TShiftState Shift, int X, int Y)
 {
 if (!_mouseDown)
    return;
 X = PaintView->CurrentLayer->ConvXScr2Bmp(X);
 Y = PaintView->CurrentLayer->ConvYScr2Bmp(Y);
 speHandMouseMove(Shift, X, Y);
 PaintView->Update();
 }
// TImageEnVect onMouseUp event:
void __fastcall TForm1::PaintViewMouseUp(TObject *Sender,
                        TMouseButton Button,
                        TShiftState Shift, int X, int Y)
 {
 if (!_mouseDown)
    return;
 X = PaintView->CurrentLayer->ConvXScr2Bmp(X);
 Y = PaintView->CurrentLayer->ConvYScr2Bmp(Y);
 speHandMouseUp(Shift, X, Y);
 }
```

That's it! All events are very similar. Delphi programmers can check demo source code to see a Delphi version of previous example.

5.2 Other shapes

To draw other shapes (ellipses, rectangles, etc...) instead of speHandMouseXXX functions we use speObjectMouseXXX functions:

```
//C++:
bool __stdcall speObjectMouseDown(TShiftState Shift, int X, int Y);
bool __stdcall speObjectMouseMove(TShiftState Shift, int X, int Y);
bool __stdcall speObjectMouseUp(TShiftState Shift, int X, int Y);
//Delphi:
function speObjectMouseDown(Shift: TShiftState;
                          X: Integer;
                          Y: Integer): Boolean stdcall;
function speObjectMouseMove(Shift: TShiftState;
                          X: Integer;
                          Y: Integer): Boolean stdcall;
function speObjectMouseUp(Shift: TShiftState;
                        X: Integer;
                        Y: Integer): Boolean stdcall;
```

So, for instance if you want to draw an outlined ellipse, here is a sequance of events (now this time in Delphi):

```
//Delphi example:
// on mouse down event
procedure TForm1.PaintViewMouseDown(Sender: TObject;
                        Button: TMouseButton;
                        Shift: TShiftState;
                        X, Y: Integer);
begin
if (Button <> mbLeft) then
   exit;
_mouseDown := True;
X := PaintView.CurrentLayer.ConvXScr2Bmp(X);
Y := PaintView.CurrentLayer.ConvYScr2Bmp(Y);
speObjectMouseDown(Shift, X, Y);
_hobj := PaintView.AddNewObject();
PaintView.ObjLayer[_hobj] := PaintView.LayersCurrent;
PaintView.ObjKind[_hobj] := iekELLIPSE
PaintView.ObjLeft[_hobj] := X;
PaintView.ObjTop[_hobj] := Y;
PaintView.ObjWidth[_hobj] := 0;
PaintView.ObjHeight[_hobj] := 0;
PaintView.ObjBrushStyle[_hobj] := bsClear;
PaintView.ObjPenColor[_hobj]   :=  clRed;
PaintView.ObjPenWidth[_hobj] := trackSize.Position;
PaintView.ObjStyle[_hobj] := [ievsVisible];
end;
// on mouse move event
procedure TForm1.PaintViewMouseMove(Sender: TObject;
                        Shift: TShiftState; X,   Y: Integer);
var
  rect: TRect;
begin
if not _mouseDown then
   exit;
X := PaintView.CurrentLayer.ConvXScr2Bmp(X);
Y := PaintView.CurrentLayer.ConvYScr2Bmp(Y);
speObjectMouseMove(Shift, X, Y);
speGetMouseData(MouseData);
rect.left := MouseData.X_down;
rect.top  := MouseData.Y_down;
rect.right := MouseData.X_move_to;
rect.bottom := MouseData.Y_move_to;
rect.NormalizeRect;
PaintView.ObjLeft[_hobj] := rect.left;
PaintView.ObjTop[_hobj] := rect.top;
PaintView.ObjWidth[_hobj] := rect.Width;
PaintView.ObjHeight[_hobj] := rect.Height;
end;
// on mouse up event
procedure TForm1.PaintViewMouseUp(Sender: TObject;
```

```
                          Button: TMouseButton;
                          Shift: TShiftState;
                          X, Y: Integer);
begin
if not _mouseDown then
   exit; // following lines do not affect drawing as they
_mouseDown := False;
PaintView.RemoveAllObjects;
X := PaintView.CurrentLayer.ConvXScr2Bmp(X);
Y := PaintView.CurrentLayer.ConvYScr2Bmp(Y);
speObjectMouseUp(Shift, X, Y);
end;
```

It seams complicated at a first glance!? Not so. We are using TImageEnVect ellipse object to draw temporary vectorial object (iekEllipse), which we can stretch around to accommodate our ellipse. It's more some kind of preview. After left mouse button is release, vectorial object (iekEllipse) is removed and real elliptical stroke with selected brush, color and color scheme is rendered. In this example we used TImageEnVect image container and viewer. If you use some other kind of image viewer, you must accommodate it's mouse events, but you'll always have a call to respective speObjectMouseXXX APIs. As you may notice in previous example, there was a call to speGetMouseData API. This function fills a structure (record) of type Tpx_MouseData which contains a bunch of useful information such as stroke beginning coordinate, ending coordinates, min, max, etc.. Here is a function prototype:

```
//C++:
bool __stdcall speGetMouseData(Tpx_MouseData &md);
//Delphi:
function speGetMouseData(var md: Tpx_MouseData) : Boolean stdcall;
```

5.3 Drawing shapes programmatically

If you want to draw some shape programmatically without using mouse events, there is a class of APIs that can perform such task:

```
//C++:
bool __stdcall speDrawRectangleRect(TRect *objRect,
                          Tpx_Geometry &geometry);
bool __stdcall speDrawEllipseRect(TRect *objRect,
                          Tpx_Geometry &geometry);
bool __stdcall speDrawRectangle(int x1, int y1, int x2, int y2,
                          Tpx_Geometry &geometry);
bool __stdcall speDrawEllipse(int x1, int y1, int x2, int y2,
                          Tpx_Geometry &geometry);
bool __stdcall speDrawPoly(TPoint *pts, int ptsCount,
                          Tpx_Geometry &geometry);
//Delphi:
function speDrawRectangleRect(var objRect: PTRect;
```

```
                        var geometry: Tpx_Geometry): Boolean stdcall;
function speDrawRectangle(x1, y1, x2, y2: Integer;
                        var geometry: Tpx_Geometry): Boolean stdcall;
function speDrawEllipseRect(var objRect: PTRect;
                        var geometry: Tpx_Geometry): Boolean stdcall;
function speDrawEllipse(x1, y1, x2, y2: Integer;
                        var geometry: Tpx_Geometry): Boolean stdcall;
function speDrawPoly(var pts: PTPoint; ptsCount: Integer;
                        var geometry: TPx_Geometry): Boolean stdcall;
```

Here we introduce a new structure (record) of type Tpx_Geometry which is used to control output of our object. Here are an explanation of structure's entries:

- SimplifyPolyline - (boolean) perform polyline simplification.
- SimplifyAlgorithm - (integer) a type of simplification algorithm to use. Currently only Douglass-Poucker algorithm is available (must be set to 0)
- SimplifyDPTolerance - (integer) tolerance in pixels
- SimplifyMaxPts - (integer) maximum number of polyline points. If this number is higher then zero, an iteration is performed until max number is reached. If this number is 0 (default), only one iteration is performed.
- PolygonModification - (integer): 0 - no modification, 1 - simplify polygon, 2 - convert polygon to splinegon.
- RotationAngle - (double) rotate an object by given angle (degrees).
- RotationOrigin - (TPoint) origin of rotation. If RotationOrigin.x and RotationOrigin.y are set to -1, rotation is perform around object's center.
- Filled - (Boolean) fill the object interior (if object is closed).
- Border - (Boolean) draw the stroke over object's border.
- PolyClosed - (Boolean) close polyline (create polygon).

Here is an example (C++) on how to draw ellipse rotated around the center:

```
//C++ example:
Tpx_Geometry geo;
// rotation angle in degrees.
geo.RotationAngle = 23;
geo.Filled = false;
geo.Border = true;
// following two lines indicate that rotation
// will be performed around ellipse's center
geo.RotationOrigin.x = -1;
geo.RotationOrigin.y = -1;
// draw ellipse with given brush and color sceheme
speDrawEllipse(100, 100, 600, 400, geo);
```

At this point, we have covered all basic APIs available in spEngine, except one:

```
//C++:s
bool __stdcall speUpdateSurface(void);
//Delphi:
function speUpdateSurface : Boolean stdcall;
```

Whenever the surface original image is modified externally (not with speEngine APIs) we need to tell this to surface container. As mentioned previously, surface consists of original image (shared) and another copy of the same image which is not shared. When original image is modified by some other means that do not include spEngine APIs call, **speUpdateSurface** must be called to ensure that non-shared copy of original image does not differ from shared original image. Here is an example. Let's say we have saved a copy of original image before drawing is performed (some kind of undo map) and we want to clear (undo) all our drawings:

```
//Delphi example:
//let's say we have saved original image in _undoMap
//now, let's clear all drawings:
procedure TfrmMain.btnUndoClick(Sender: TObject);
begin
 _undoMap.DrawToTIEBitmap(PaintView.IEBitmap, 0, 0);
 speUpdateSurface;
 PaintView.Update;
end;
```

As content of undo map is drawn onto original image, we must synchronize current surface with new image state, so we call **speUpdateSurface** function to do that.

6 Advanced drawing

6.1 Stroke envelope

So far, we have covered simple drawing techniques, without calling any additional API. This section covers drawing using stroke envelope. Stroke envelope is used to vary brush tip size (or capacity) while stroke is drawn. You may notice that inside BrushCommon structure/records there are two parameters that can change brush size/capacity while stroke is drawn (SizeVary and CapVary). Both parameters range from 0 to 100 and give percentage of size/capacity variations of brush tip inside stroke. Brush tip variations can not exceed initial size, i.e. variations are always smaller then initial value.

However, variations using SizeVary and CapVary parameters are random variations. On the other hand, stroke envelope uses predefined patter (string) which tells spEngine how to change size/capacity inside the stroke. When envelope is on, it will override random variations of respective type (size/capacity).

```
//C++:
bool __stdcall speSetSizeEnvelope(wchar_t* str);
bool __stdcall speSetCapacityEnvelope(wchar_t* str);
//Delphi:
function speSetSizeEnvelope(str: PWideChar): Boolean stdcall;
function speSetCapacityEnvelope(str: PWideChar): Boolean stdcall;
```

Here is an example how to set some simple pattern for stroke envelope:

```
//C++ example:
UnicodeString env = "10,40,80,100";
speSetSizeEnvelope(env.c_str());
brushCommonParams.EnvelopeSizeOn = true;
brushCommonParams.EnvelopeSizeRepeat = true;
brushCommonParams.EnvelopeResolution = 4;
brushCommonParams.EnvelopeResolutionManual = true;
speSetBrushCommon(brushCommonParams);
```

In above example envelope pattern is set to "10,40,80,100" which means that stroke will begin with 10% of original brush size, then it will go to 40% of the brush size and so on. How fast this change occurs depends on **EnveloperResolution** parameter which gives number of steps between each subsequent pattern value. By default, resolution is set to 10 and can be changed by user. In the example above, resolution is set to 4, which means that change from 10% to 40% will be executed in 4 steps. On each step new value is calculated (interpolation between 10 and 40). Higher the resolution value, slower the change in size. If **EnvelopSizeRepeat** is false (by default), after last size change is

achieved (lastSize), stroke continues with brush size = lastSize. If EnvelopeSizeRepeat is set to **true**, pattern will be repeated when internal counter reaches the last pattern value.

The same logic applies to brush capacity envelope. Size and capacity envelope changes may be simultaneously in single stroke. Note: size and capacity envelope patterns can be different.

Parameter **EnvelopeResolutionManual** does not effect "free-hand" drawing shape, but effects all other shapes (ellipse, rectangle, etc...). When set to false (default), envelope resolution of a stroke going through shape's points is calculated internally: pattern is adjusted to shape's points number and EnvelopeSizeRepeat (EnvelopeCapacityRepeat) value does not have any effect. When set to **true** (as in example above), shape is treated as "free-hand" drawing: envelope follows predefined resolution and repeating value.

6.2 Clone brush

To use clone brush, you must first set some clone image. It can be the same image you're drawing on (**speSetCloneSelf**), or some other external image (**speSetCloneImage**). Besides that, to begin drawing with clone brush, you need to set the clone image start position:

```
//C++:
bool __stdcall speSetCloneImage(void *buffer, int width, int height,
        unsigned int scanlineAlignement, bool shared);
bool __stdcall speDeleteCloneImage(void);
bool __stdcall speCloneSelf(TspeCloneType clType);
bool __stdcall speSetClonePosition(int X, int Y);
//Delphi:
function speSetCloneImage(buffer: Pointer;
        width: integer; height: Integer;
    scanlineAlignement: Cardinal; shared: Boolean): Boolean stdcall;
function speDeleteCloneImage: Boolean stdcall;
function speCloneSelf(clType: TspeCloneType): Boolean stdcall;
function speSetClonePosition(X: Integer; Y: Integer): Boolean stdcall;
```

In order to enable clone brush drawing, you must set color scheme to: px_CTYPE_CLONEALIGNED, px_CTYPE_CLONEREPOS or px_CTYPE_CLONEFIXED. Here is an example:

```
//C++ example:
//load clone image
TIEBitmap cloneMap;
cloneMap = new TIEBitmap();
TImageEnIO *io = new TImageEnIO(this);
io->AttachedIEBitmap = cloneMap;
io->LoadFromFile("C:\\art\\flowers.jpg");
io->AttachedIEBitmap = 0;
delete io;
if (cloneMap)
```

```
{
int h = cloneMap->Height;
int w = cloneMap->Width;
// set non-shared clone map (new copy) shared = false
speSetCloneImage(cloneMap->Scanline[h-1], w, h, 4, false);
// set some starting point or use mouse to pick the coordinates
speSetClonePosition(w/2, h/2);
speSetRenderData(renderData);
// set clone reposition mode
speSetColorScheme(px_CTYPE_CLONEREPOS);
//we can delete orignal TIEBitamp, because it's not shared
delete cloneMap;
//ready for drawing
}
```

When some external image is used as clone image (**speSetCloneImage**), you must be careful if you use shared option, because if image is deleted, internal clone surface will reference unexisting data. When clone image is deleted, you need to call **speDeleteCloneImage.** When you want to use image you're drawing on as a clone image, use **speCloneSelf** function. With speCloneSelf function you don't have to worry about deleting image/surface as it's done internally. Threee options are available for **clType** parameter:

- **px_cltExternal**: use external clone image if available (default).
- **px_cltDirect**: use original image as clone image. Clone image is updated on each brush step.
- **px_cltBuffered**: use internal copy of original image. Clone image is updated after stroke is finished (*usually in MouseUp event*).

When some clone color scheme is set, the engine first looks if self clonning is set (*px_ cltDirect or px_ cltBufferd*). If not, it looks if there is external clone image. If clone image is not set, it will draw strokes with primary color.

6.3 Output color scaling

Normally, when we manipulate pixels, each output color channel (RGB) is scaled to [0,255]. This scaling range can be changed using **speSetScaleColors** API:

```
//C++:
bool __stdcall speSetScaleColors(unsigned int topVal,
        unsigned int botVal);
//Delphi:
function speSetScaleColors(topVal: Cardinal;
        botVal: Cardinal): Boolean stdcall;
```

Any valid color can be used for new scale. **topVal** value is used to set upper scale boundary while **botVal** value is used to set lower scale boundary. By default topVal = RGB(255,255,255) and botVal = RGB(0,0,0).

6.4 Pixels arithmetics

By default, strokes drawing mode (ArithMode in DrawParams) is set to px_ARITH_NORM value: pixel value of input image at position X,Y and brush tip color are blended according to brush mask intensity. If ArithMode is set to, let's say px_ARITH_XOR, input image pixel value is XOR-ed with current brush color value and resulting value is blended with original input image pixel value according to brush mask intensity. All possible modes are described in **spEngine structures and constants** chapter.

7 spEngine structures and constants

Structures/records parameters (variables) will be explained in 4 column tables:

1. Parameter (variable) name.
2. Parameter type and it's domain (range).
3. Default parameter value.
4. Description

7.1 Tpx_RenderData

As previously stated structure/record Tpx_RenderData is nested structure and controls global rendering parameters, i.e. once set, it will reflect rendering style on all images inside surface container.

7.1.1 Tpx_RenderParams

Parameter	Type-Domain	Def.	Description
ClearMask	bool	true	Clear internal mask after drawing.
HardMask	bool	false	Perform hard masking after drawing.
UpdateImage	bool	true	Synchronize original image and it's internal copy.
AddUndo	bool	true	Add undo slot after drawing.
RenderingMode	int - [0,4]	0	Fore each mode, there is associated constant defined in spGlobals unit: px_RENDER_AUTOMATIC = 0 px_RENDER_SLOW = 1 px_RENDER_MEDIUM = 2 px_RENDER_FAST = 3 px_RENDER_DUMMY = 4
DoShadow	bool	false	Drop shadow after drawing.
DrawPolyInterior	bool	false	Draw polygon interior
DrawTipOnMouseDown	bool	true	Draw brush tip on mouse down event.

Note: to do.

7.1.2 Tpx_DrawParams

Parameter	Type-Domain	Def.	Description
ArithMode	int - [0, 27]	0	Arithmetic mode - sets how image pixels are combined with brush color scheme. For each mode, there is a constant in spGlobal unit: px_ARITH_**mode_name**.
SaveMode	int - [0, 27]	0	Used to save current mode.
Shape	int - [0, 12]	0	Obsolete
InvertSource	bool	false	Inverts source image (internal image negative).
BitShift	int - [-8,8]	0	Shifts source image (internal) by desired amount of bits: left if value is positive, right if negative.
UpdateAlpha	bool	true	Update alpha channel after drawing.
StrokesNumber	int	0	Number of strokes that will be drawn (used for ornamental drawing). Currently not implemented.
EraseMatchAll	bool	?	Erase all pixels if ArithMode = px_ARITH_ERASE
EraseMatchColorTolerance	int - [0, 255]	?	Color tolerance when EraseMatchAll = false

Note: to do.

7.1.3 Tpx_ShadowParams

Parameter	Type-Domain	Def.	Description
MainMode	int - [0, ?]	0	Shadow blending mode - currently only mode 0 is supported.
Xoffset	int	4	Shadow displacement in X directon.
Yoffset	int	4	Shadow displacement in Y directon.
BlurOn	bool	false	Apply blur filter on shadow.
BlurValue	int - [1, n]	1	Blur filter radius.
Capacity	int [0, 100]	100	Shadow capacity/opacity percentage.

Note: shadow color is controlled/set using **speSetshadowColor** function.

7.2 Tpx_BrushCommonParams

This structure/record contains various brush parameters (common to all brushes) and will be explained by sections.

7.2.0.1 Brush tip parameters

Parameter	Type-Domain	Def.	Description
ListIndex	int - [0, 9]	0	Currently not used- will return brush bucket index
Size	int - [1, n]	20	Brush tip size
Capacity	int - [0, 100]	100	Brush capacity/opacity percentage.
ColorVary	int - [0, 100]	0	Max. color variation percentage (randomized).
SizeVary	int - [0, 100]	0	Max. size variation percentage (randomized).
CapVary	int - [0, 100]	0	Capacity/opacity variation percentage (randomized).
Step	int - [1, n]	2	Brush tip step.

Note: to do.

7.2.1 Brush filter parameters

Parameter	Type-Domain	Def.	Description
ChalkOn	bool	false	Perform chalk effect
InvertOn	bool	false	Invert brush intensity
InvertValue	int - [1, 255]	16	Invert threshold: only intensities above this value will be inverted.
CloseOn	bool	false	Perform close morphological filter. Currently not implemented.
CloseValue	int - [1, n]	1	Number of close filter iterations. Currently not implemented.
BlurOn	bool	false	Perform blur filter.
BlurValue	int - [1, n]	1	Blur filter radius.
MedianOn	bool	false	Perform median filter.
MedianValue	int - [1, n]	1	Median filter radius.

Note: to do.

7.2.2 Brush bumping parameters

Parameter	Type-Domain	Def.	Description
BumpOn	bool	false	Perform brush bumping effect.
BumpSoft	bool	false	Perform soft bumping (if BumpOn = true).
BumpScaleValue	float - [0, n]	1	Bumping scale.
BumpThresholdValue	int - [0, 255]	0	Bump threshold value: only intensities above this value will be bumped.
BumpRoughnessValue	int - [0, 255]	0	Bump roughness (randomized).
BumpXoffsetValue	int	1	Bump displacement in X direction ("light" position).
BumpYoffsetValue	int	-1	Bump displacement in Y direction ("light" position).
BumpInvertThreshold	bool	false	Invert threshold value (255 - BumpThresholdValue).

Note: to do.

7.2.3 Brush envelope parameters

Parameter	Type-Domain	Def.	Description
EnvelopeSizeOn	bool	false	Apply stroke size envelope.
EnvelopeCapacityOn	bool	false	Apply stroke capacity/opacity envelope.
EnvelopeSizeRepeat	bool	true	Repeat size envelope while stroke is drawn.
EnvelopeCapacityRepeat	bool	true	Repeat capacity/opacity envelope while stroke is drawn.
EnvelopeResolution	int - [0, n]	10	Envelope resolution.
EnvelopeResolutionManual	bool	false	Set manual resolution (used for "delayed" shapes).

Note: to do.

7.2.4 Brush tip jittering

Parameter	Type-Domain	Def.	Description
JitterOn	bool	false	Perform stroke (brush tip) jittering.
JitterConnectedOn	bool	true	Perform connected jittering.
JitterRangeValue	int - [1, n]	1	Max. jittering distance (randomized).
JitterLoopValue	int - [1, n]	1	Number of jitterings per step.

Note: to do.

7.2.5 Polygon filling parameters

Parameter	Type-Domain	Def.	Description
PolyFillOver	bool	false	Polygon is filled over stroke borders.
PolyExcludeFilters	bool	true	Do not apply brush filters on polygon interior.
PolyExcludeBump	bool	false	Do not apply bumping on polygon interior.
PolyColorIndex	int - [0, 1]	0	Polygon filling color scheme. Currently supported: 0 - single color(0), 1 - texture
PolyOverrideCapacityOn	bool	false	Override current brush capacity/opacity.
PolyCapacityValue	int - [0, 100]	100	Capacity/opacity percentage (if override is true).
PolyBlurOn	bool	false	Apply blur filter on polygon interior.
PolyBlurValue	int - [1, n]	1	Blur radius.
PolyShrinkOn	bool	false	Shrink polygon interior.
PolyShrinkValue	int - [1, n]	1	Shrinking percentage (uses current brush size as reference). Currently not implemented.

Note: to do.

7.2.6 Strokes scratching

Parameter	Type-Domain	Def.	Description
ScratchType	int - [0, 3]	0	Scratch type. For each type there is associated constant defined in spGlobals unit: px_SCRATCH_NONE = 0 px_SCRATCH_CIRCLE = 1 px_SCRATCH_SQUARE = 1 px_SCRATCH_SHORT_LINE = 2 px_SCRATCH_LONG_LINE = 3
ScratchThicknessValue	int - [1, n]	1	Scratch thickness.
ScratchNumberValue	int - [1, 100]	10	Number of scrathes per 100x100 pixels.

Note: to do.

7.3 Brush specific parameters

This section describes specific brush parameters, i.e. parameters related exclusively to selected brush type...to do...

8 Acknowledgments

Many thanks to **Bill Miller** from **Adirondack Software & Graphics** for testing, suggestions and Delphi demo with standard VCLs.