Sprite2D SDK for FBSL v3

Introduction

The Ultimate Sprite2D Software Renderer (hereinafter referred to as "the USSR") is intended to create windowed 2D applications suitable for use on Windows XP/Vista/7/8 platforms. As of the date of this writing, the USSR comprises two separate files, CSprite2D.inc and Sprite2DLib.inc, to be included in the user's main script file.

The CSprite2D class loads and stores a colorful 24- or 32-bpp two-dimensional image and features a rich set of properties and methods to transform the image and display it on an FBSL form or any Windows control that has a device context to render to. For brevity, we will hereinafter refer to an instance of the CSprite2D class as "a sprite".

The Sprite2DLib library file is an assortment of FBSL v3 Dynamic Assembler routines compilable into native executable machine code at main script load time. The most time-critical sections of assembly code are written using the MMX instruction set which makes the code extremely fast though narrowing the applicability of USSR down to Intel Pentium MMX+ architectures. The sprites access the routines on their own through their private and public members to perform various transformations of the images loaded in them. However, all the routines are also directly accessible from the main script and its includes thus allowing for greater flexibility of the user's source code. The Sprite2DLib will be hereinafter referred to as "the library".

The USSR **is not based** on OpenGL or DirectX. Neither does it use any hardware acceleration specific to respective GPU makes. It makes the renderer particularly attractive for modern business notebooks which are built around powerful CPU's but usually have very weak video cards. The USSR renders as fast as only the user's CPU permits allowing, at the same time, for a wide variety of per-pixel effects normally available on high-end gaming-style GPU's only.

1. Minimum Deployment

```
#Uses "@|WIN32"
#Option Strict

#DllDeclare ("BeginPaint", "EndPaint", "InvalidateRect")

#Include ".\CSprite2D.inc"
#Include ".\Sprite2DLib.inc"

Resize(ME, 0, 0, 640, 480)

Dim %Width, %Height, $PS * 64 ' fake PAINTSTRUCT
Dim SprTest As New CSprite2D ' create new sprite

GetClientRect(ME, 0, 0, Width, Height)
SprTest.Init(Width, Height) ' assign size equal to form's entire client area
SprTest.ClearBuffer(ARgb(0, 255, 0, 0)) ' fill sprite with red color

Center(ME)
Show(ME)

Begin Events
	Select Case CBMSG
		Case WM_COMMAND ' VK_ESCAPE pressed
			If CBLPARAM = 2 Then
				PostMessage(ME, WM_CLOSE, 0, 0)
				Return 0
			End If
		Case WM_ERASEBKGND ' suppress flicker on form redraw
			Return 1
		Case WM_PAINT
			SprTest.PaintToDevice(BeginPaint(ME, PS))
			EndPaint(ME, PS) ' render sprite onto form
			InvalidateRect(ME, NULL, FALSE) ' initiate next render
			Return 0
		Case WM_CLOSE
			SprTest.Dispose() ' deallocate sprite resources
	End Select
End Events
```

Copy the above code and paste it into the text editor window. Save the script as e.g. Test.fbs and run it to see the entire main form painted in red.

2. Loading and Saving Images

Another way to initialize a sprite is load an image file into it. Currently, CSprite2D supports two image file loaders:

- 24- (RGB) and alpha-channel 32-bit (RGBA) uncompressed TGA files through the LoadFromTGA() method
- Any image file formats that the GDI+ library supports on the user's PC with an alpha-channel where applicable (e.g. BMP, PNG, TIF, GIF and JPG) through the LoadFromFile() method

You can change the following two lines in the above script

```
SprTest.Init(Width, Height) ' assign size equal to form's entire client area
SprTest.ClearBuffer(ARgb(0, 255, 0, 0)) ' fill sprite with red color
```

with

```
SprTest.LoadFromTGA(".\1.tga") ' put 1.tga into main script's directory
```

and the main form will now display the TGA picture. When loading an image file, the sprite readjusts itself automatically to the image size. There's no problem if the image is larger than the main form. In this case, only part of the sprite that fits in the form's client area will be rendered.

The user can also save sprite images to disk files via the sprite's SaveToFile() method. Currently, BMP and PNG formats are supported as well as the JPG format with a user-defined quality that defaults to 80 per cent. To enjoy the full range of file save formats, the corresponding file codecs must be available on the user's PC.

3. Rendering One Sprite into Another One

Rendering one sprite into another one is the main method of game scene creation in the USSR. The engine implements back buffering whereby all the scene sprites are first rendered into the common back buffer sprite which, in its turn, is then transferred into the main form's device context thus completely eliminating any visible flicker.

Let's extend the above sample script in the following way:

```
#Uses "@|WIN32"
#Option Strict

#DllDeclare ("BeginPaint", "EndPaint", "InvalidateRect")

#Include ".\CSprite2D.inc"
#Include ".\Sprite2DLib.inc"

Resize(ME, 0, 0, 640, 480)

Dim %Width, %Height, $PS * 64 ' fake PAINTSTRUCT
Dim X, Y ' image position on main form
Dim SprBack As New CSprite2D ' create backbuffer sprite
Dim SprTest As New CSprite2D ' create image sprite

GetClientRect(ME, 0, 0, Width, Height)
SprBack.Init(Width, Height) ' assign size equal to form's entire client area
SprTest.LoadFromTGA(".\1.tga") ' put 1.tga into main script's directory

Center(ME)
Show(ME)

Begin Events
	Select Case CBMSG
		Case WM_COMMAND ' VK_ESCAPE pressed
			If CBLPARAM = 2 Then
				PostMessage(ME, WM_CLOSE, 0, 0)
				Return 0
			End If
		Case WM_MOUSEMOVE ' image follows mouse
			X = LoWord(CBLPARAM)
			Y = HiWord(CBLPARAM)
```

```
            Case WM_ERASEBKGND ' suppress flicker on form redraw
                  Return 1
            Case WM_PAINT
                  Render(BeginPaint(ME, PS))
                  EndPaint(ME, PS) ' render sprite onto form
                  InvalidateRect(ME, NULL, FALSE) ' initiate next render
                  Return 0
            Case WM_CLOSE
                  SprBack.Dispose() ' deallocate sprite resources
                  SprTest.Dispose() ' ditto
      End Select
End Events
' ------------------------------------------------------------------------

Sub Render(hDC)
      SprBack.ClearBuffer(ARgb(0, 0, 0, 255))
      SprBack.Draw(SprTest, X, Y)
      SprBack.PaintToDevice(hDC)
End Sub
```

Here two sprites are created:

- SprBack which is intended for accumulating the overall scene layout and rendering it onto the main form; and
- SprTest which is used for loading some picture.

On mouse move, SprBack is painted blue and then SprTest is **rendered into** SprBack at the current cursor position, and finally, the entire SprBack bitmap is **transferred** to the main form's device context in one swoop.

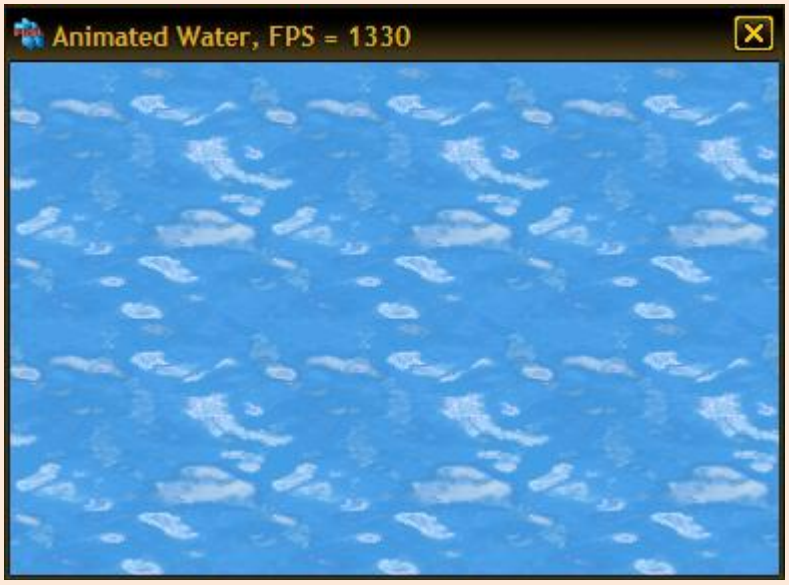
If 1.tga contains an alpha channel, then it will be rendered correctly (translucency is fully supported). Whenever 1.tga has no explicit alpha channel, translucency is also possible through the LoadFromTGA() method's optional ColorKey parameter:

```
SprTest.LoadFromTGA(".\1.tga", ARgba(0, 0, 0, 0))
```

Provided 1.tga has its background painted in black, the black pixels of the image will become transparent.

As soon as the image is loaded into the sprite, the AddColorKey() method can be used to add more color keys to the image effectively making more than one color of the image transparent.

The MulAddS2X() method can be used to change the image brightness and contrast. A pixel color component from the source is multiplied by the Mul parameter and the Add value is added to the product. Then 128 is subtracted from the sum as if Add were in fact a signed number, and finally the resultant value is multiplied by 2. The value is transferred to the destination sprite. See Water.fbs for a demonstration of this method:



4. Blend Modes

The Draw() method of a sprite features yet one more optional Op parameter (blend operator) whose acceptable integer values are defined as the Sprite2D_Op enumeration. The parameter determines which particular technique of mixing the source and destination sprite colors is going to be used. There are actually 11 such operator values as described below:

- **OpPaint** replaces the destination sprite's entire data including the alpha channel, with the source sprite's data.

- **OpAlphaTest** replaces the destination pixel color with the source pixel color for all the pixels whose alpha component in the source pixel color is above the medium of 128. Other pixels do not change their colors. This blend mode is useful for stencil sprites.

- **OpAlphaBlend** yields standard alpha blending whereby the resultant color is the product of linear interpolation between the source and destination colors by the alpha component of the source.

- **OpAdd2D** results in a color (including its alpha channel) which is an arithmetic mean of source and destination colors. This blend mode is useful for translucent images.

- **OpAdd** yields a color (including its alpha component) which is an arithmetic sum of source and destination colors. If the sum exceeds 255, it is clamped to that value. This blend mode is useful for translucent light-emitting object such as e.g. fire. See Fire'n'Light.fbs for a use case example:

- **OpMul** yields a resultant color as an arithmetic product of source and destination colors.
- **OpMul2X** produces an arithmetic product of source and destination colors further multiplied by 2 and clamped to 255 if the final product exceeds that value.
- **OpMin** normalizes the destination colors to the corresponding minimum values in the source.
- **OpMax** is the opposite of OpMin effectively saturating the destination colors with the corresponding source colors.
- **OpBlend** implements composite blend modes taking account of the source sprite's BlendFactor() value.
- **OpDefault** is a placeholder which denotes an uninitialized blend mode of the Op() property.

The MaskDraw() method is an analog to Draw() for operating with additional bit masks (see Section 7. Bit Masks for fuller description). MaskDraw() implements full blend mode functionality characteristic of the Draw() method.

If the blend mode operator is not specified explicitly, the source sprite's default operator is used. For example, it will be OpAlphaBlend if the sprite has been loaded with a 32-bit image, or OpPaint for a text or 24-bit image sprite, or OpAlphaTest if the image has been loaded with an explicit ColorKey.

A sprite's default blend mode operator can be changed, e.g. like this:

```
SprTest.Init(640, 480, OpAlphaTest)
```

On executing this command, SprTest will acquire OpAlphaTest as its default blend mode operator. Alternatively, the Op() property can be used to change the sprite's run-time blend mode behavior.

5. Texturing

The TileDraw() method enables the user to tile the source sprite image across a rectangular area of an arbitrary size in the destination sprite. For example, the following command

```
SprBack.TileDraw(SprBrick, 0, 0, SprBack.Width(), SprBack.Height(), X, Y)
```

will cover the entire SprBack with the image of SprBrick. X and Y determine the horizontal and vertical offsets in the SprBrick sprite, respectively. See the Fire & Light snapshot above for an example of brick texture tiled across the wall area in the backbuffer sprite.

6. Draw Area Clipping

On default, the destination sprite's entire area is accessible for painting or overlaying with the source sprite image. This behavior can however be overridden by specifying a rectangular area beyond which graphical output is inhibited (clipped). For example, the following line:

```
SprBack.ClipRect(50, 250, 50, 250) ' left/right/top/bottom
```

will clip the output into SprBack to a square area within the specified borders. Invalid border values (e.g. Left > Right) will be ignored and previous borders will be kept instead. On loading an image from the disk file into the sprite or initializing the sprite via its Init() method, the clipping rectangle is set to the sprite's entire area.

7. Bit Masks

Alongside Draw(), the MaskDraw() method is yet another technique of projecting one sprite onto another. But in contrast to Draw(), it operates with **three** masks rather than two of them. The third sprite is interpreted as a bit mask. The syntax of the method is as follows:

```
Dest.MaskDraw(Src, SrcMask, sX, sY, mX, mY, BitMask, NotMask, Op)
```

whereby sX, sY is the position to draw the source sprite in the destination sprite and mX, mY is the position where to overlay the source mask sprite image on the destination. A color value is taken from the source mask sprite and compared against the BitMask selector which is an integer value. If at least one bit in the color value and BitMask is the same, the Src sprite pixel is drawn in the destination sprite literally, just as it would using simple Draw().

A typical example would be

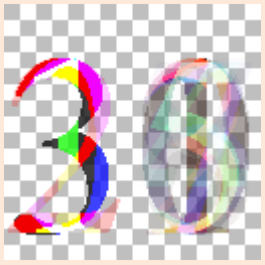
```
Dest.MaskDraw(Src, SrcMask, sX, sY, mX, mY, &H80000000, OpPaint)
```

whereby the Src sprite is drawn to the Dest sprite as if an OpAlphaTest operator were specified, except that the alpha channel is taken from the SrcMask sprite rather than from Src. This method can be used e.g. for overlaying a light spot on another sprite's image which has an alpha channel (the OpMul2X operator is especially useful for light spot blending), all this being drawn against a distant background which shouldn't naturally be lit by the nearby light source.

All the sprites in the USSR are always stored in the 32-bit format, i.e. any sprite can store up to 32 bit masks. Moreover, the user can select not only any single one of them but also any number of them and in any combination. For example, a call to

```
BitMask(0) BOr BitMask(13) ' similar to FBSL's "1<<numbits" expression
```

would select and combine masks 0 and 13. When set to TRUE, boolean NotMask inverts the mask opacity. This is how an example sprite hosting 32 bit masks may look (see Mask.fbs for a use case implementation):



8. Bump Mapping

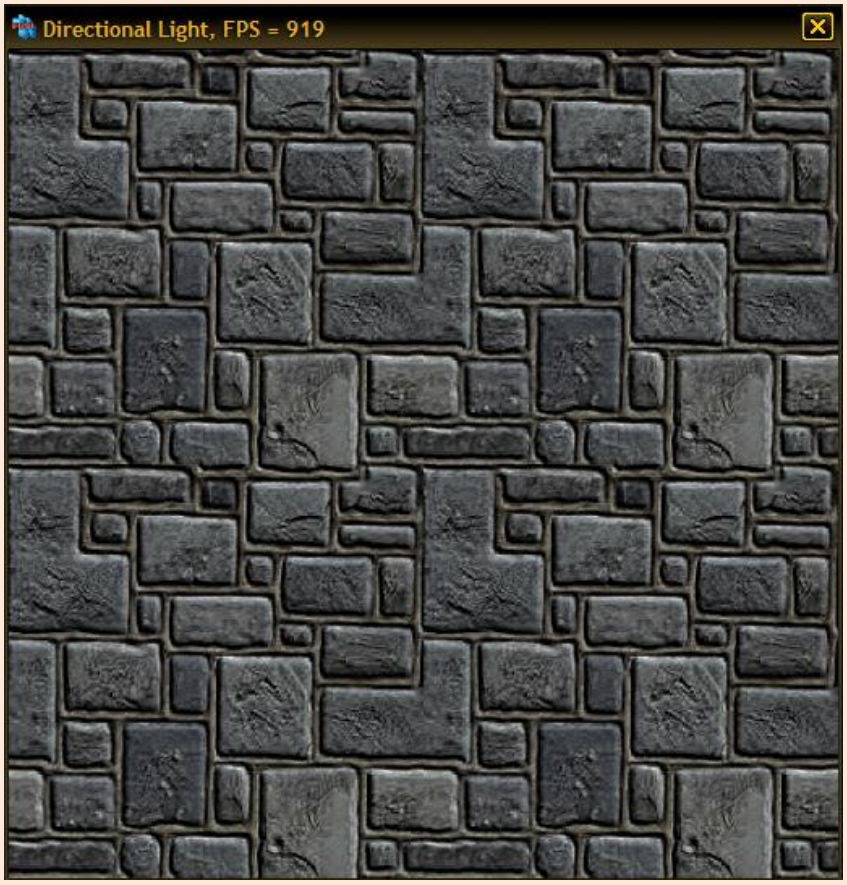
The USSR implements two methods of per-pixel bump mapping:

- Environment Bump Mapping (EBM) with an infinite distance light source; and
- Dot Product Bump Mapping (DPBM) with a finite distance light source.

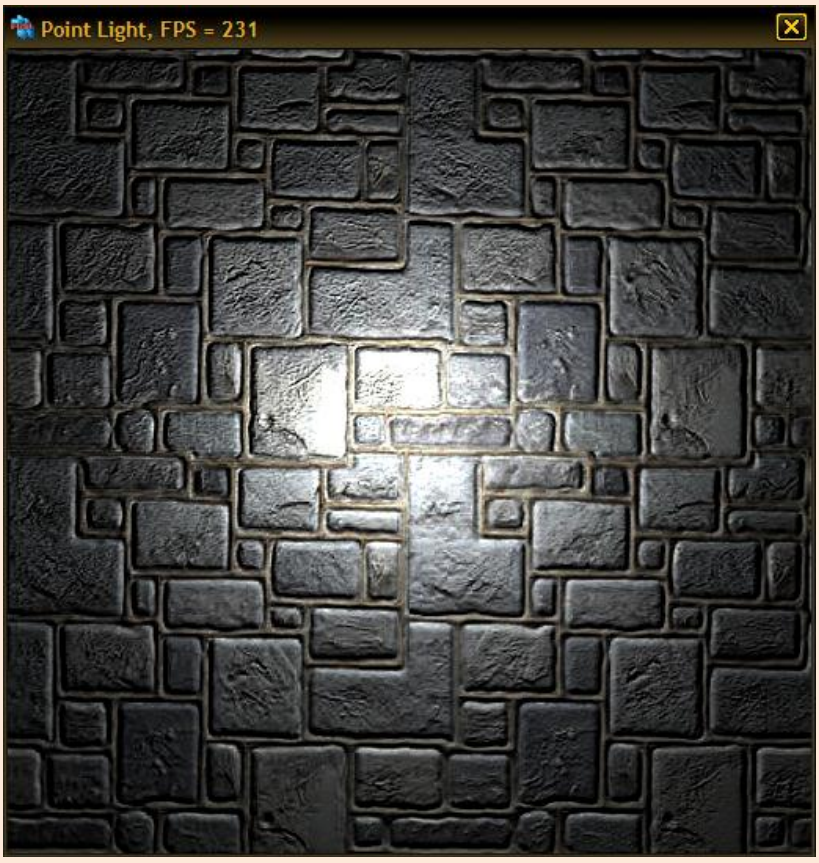
Both methods are intended for surface relief emulation. DrawEBM() emulates global reflective bump mapping effect while DrawDPBM() emulates surfaces partially lit by a finite-distance (optionally point-) light source. The methods have their bitmask counterparts, MaskDrawEBM() and MaskDrawDPBM(), with essentially the same functionality.

DrawEBM() operates with two source sprites, SrcBump and SrcCol. SrcBump hosts a bump map (a.k.a. "normal map") while SrcCol hosts a diffuse map (a.k.a. "color map"). The bump map's red color component corresponds to the X coordinate, the green one, to the Y coordinate. First SrcBump is sampled for a color and then SrcCol is sampled using the same coordinates as the former sample. If the optional DestSpace parameter is set to TRUE, the latter sample will also account for the destination sprite coordinates. See Water.fbs above for a demonstration of EBM effects, see Dot3Light.fbs for DPBM.

DrawDPBM() requires the light direction to be specified on the third, Z, coordinate. The light is directed towards the viewer. Brightness should be clamped to within [0, 1]. The light vector is normalized and multiplied pixel-per-pixel (scalar multiplication used) by the source sprite values. The red color component in the bump map corresponds to the X coordinate, the green one, to Y, the blue one, to Z. The result is written into all components of the destination sprite including its alpha channel (OpMul2X adds extra gloss):



By multiplying the DPBM result by the diffuse map color values with the OpMul or OpMul2X blend mode operators, the user can obtain perfect per-pixel point lighting effects:



9. Text Sprites

Text sprites are a special brand of sprites that can display formatted text. A text sprite cannot host an additional image but the text can be drawn transparently, in which case it overlays seamlessly on any desired background.

The text is entered into the sprite via its LoadText() method. The size of the sprite is calculated automatically given the length of text string as well as the metrics of the font chosen to display the text. The user can specify the following font parameters:

- foreground and background colors (optionally a color key)
- point size
- bold, italic and underline flags
- font family name

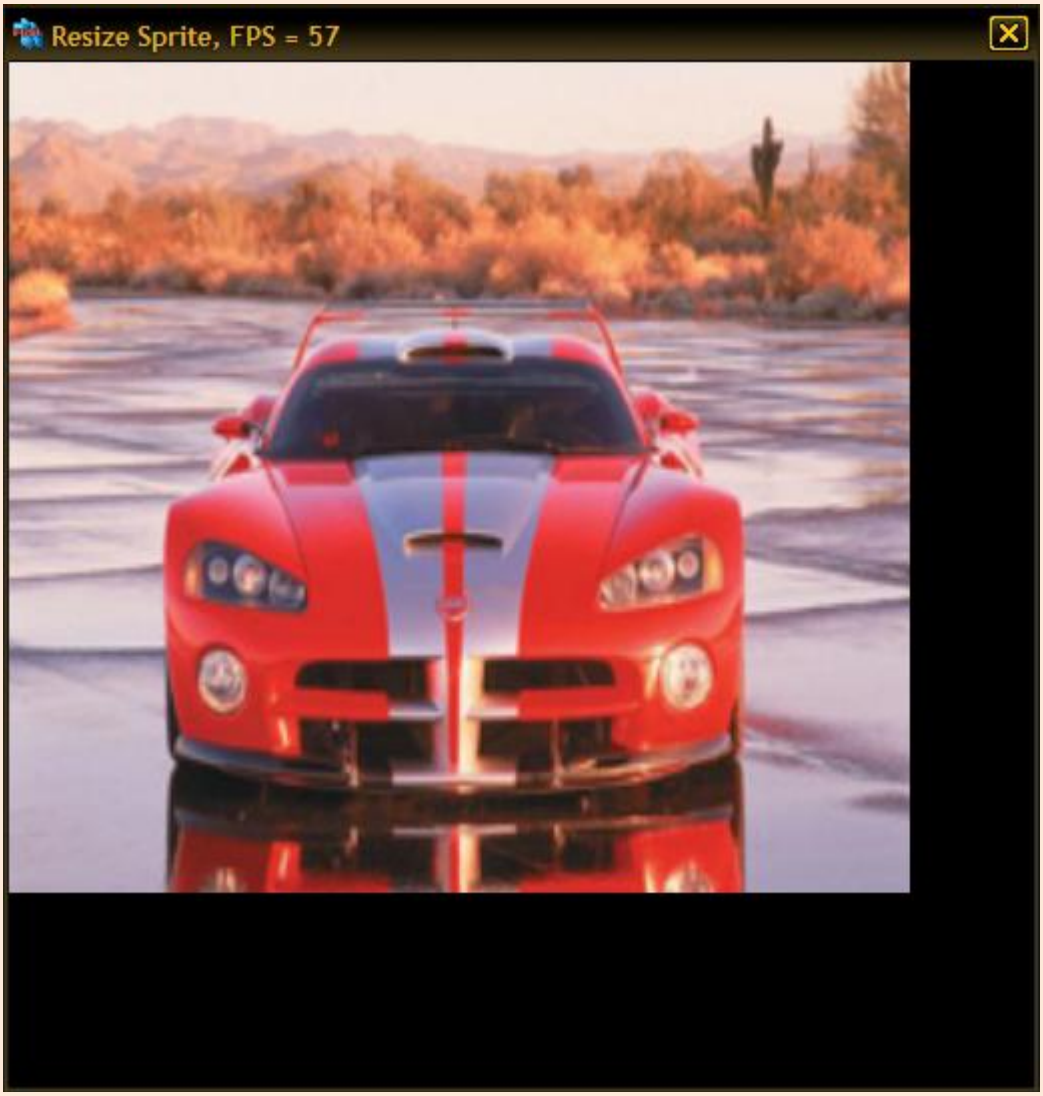
The text line may contain optional line breaks in which case the text is wrapped into a multiline message. The text is always centered within its sprite both horizontally and vertically. See Font.fbs for an example use case:

10. Image Transforms

The USSR implements various image transforms including top-down flipping, left-right mirroring and 90-degree CW and CCW rotation.

The image can also be rotated at an arbitrary angle using the DrawRotate() method. The rotation comes in two flavors, ordinary (fast) and anti-aliased (slow). The method uses bilinear filtering for its anti-aliased rotation mode.

The image can also be resized to fit the given sprite bounds. This can be achieved by loading one sprite's image into another sprite with the LoadFromSprite() method. If the sizes of the source and destination sprites don't match, the source image is stretched or shrunk to fit the destination bounds. While being resized, the image can be optionally bilinearly filtered for better visual results. See Resize.fbs for an example of the quality of bilinear filtering:



11. Pixel-Perfect Collision Detection

The MaskIntersector() method returns the area of sprite intersection with a given source sprite by comparing the masks of the two sprites. Based on the given threshold value entered via the Mask parameter, the masks are compared against each other pixel-by-pixel yielding pixel perfect results.

Considering a zero return value of the method as "Collision = FALSE" and any other value as "Collision = TRUE", the user can introduce a pixel-perfect sprite collision detection system into his/her application. See MaskedISect.fbs for a use case of this method:

12. Drawing Primitives

Individual pixels within the sprite image may be freely accessed/colorized using the GetPixel() and SetPixel() methods.

The DrawLine() method enables the user to draw colorful lines. The optional DotStep parameter can be modified to draw arbitrarily dotted lines. The optional IsXor parameter, if set to TRUE, has the same effect as mixing the colors by "Dest BXor Color" so that if Color = &H808080 then the resultant destination color will be inverted. In this case, if DrawLine() is used with the same parameters again, the destination colors are restored exactly to their original values.

The entire sprite rectangle can be colored using the ClearBuffer() and MaskClearBuffer() methods. Arbitrarily sized color rectangles can be drawn using the ClearRect() method.

13. Extending Sprite2D Engine Usability

The Water.fbs and Fire'n'Light.fbs scripts mentioned above extend applicability of the USSR by introducing very simple composite sprite classes based on the available base CSprite2D class.

Yet another simple sample extends this applicability still further by providing a CAnime2D class which exemplifies a simple sprite animation facility that allows for CPU-speed-independent performance of the animated character in the game scene environment.

Acknowledgements

The Ultimate Sprite2D Software Engine is built around a similar project by **Mikle**, found at the Russian VB-oriented site VBStreets.ru.

Another beautiful VB6 masterpiece called "Sunrise" allegedly by the same very creative and talented person, **Mikle**, can be found at the DemoScene board here.