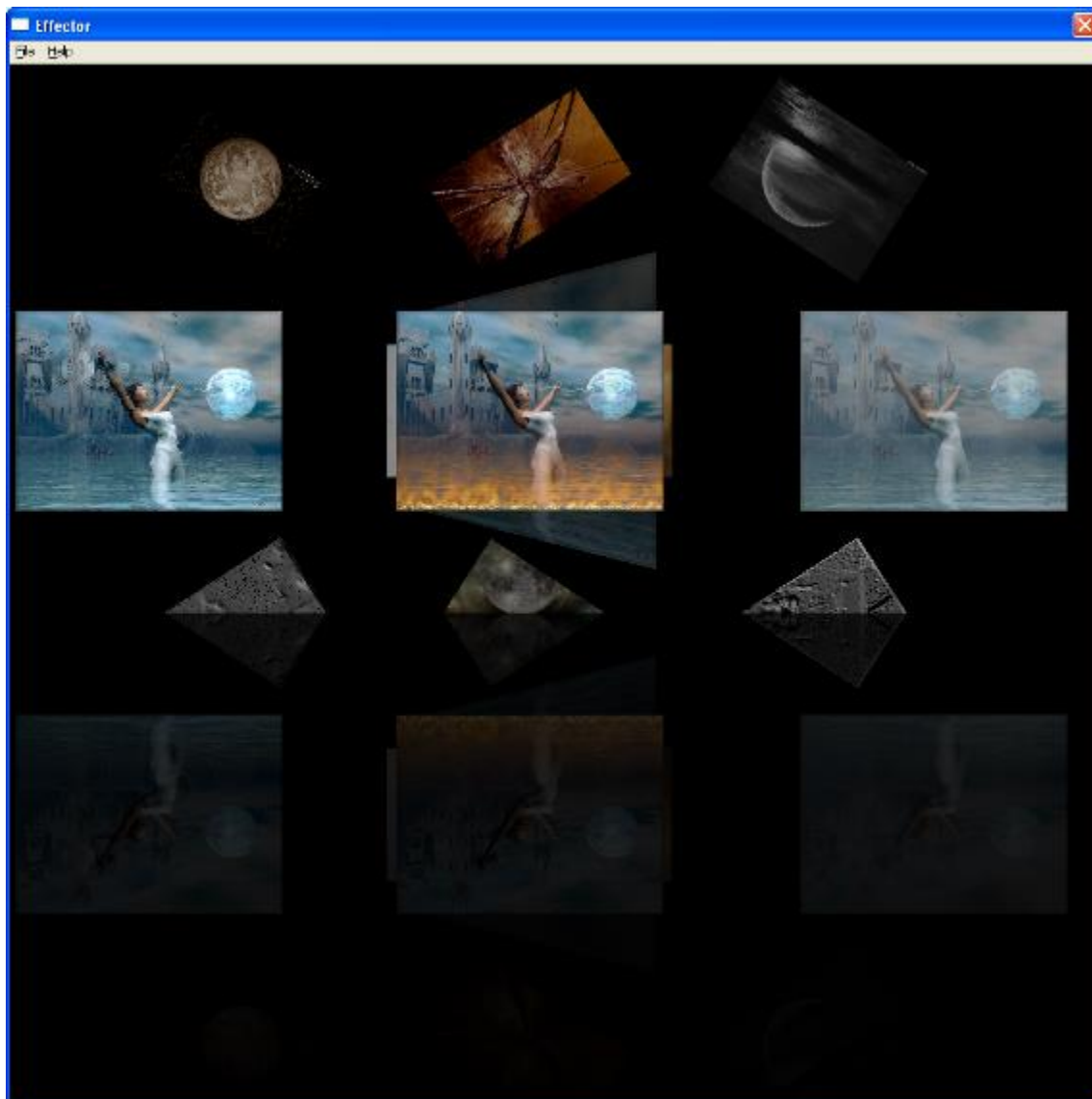


CBitmapEx - Free C++ Bitmap Manipulation Class

An article on a free C++ bitmap manipulation class



Introduction

This article is about the 'simple' C++ bitmap manipulation class called **CBitmapEx**. Many times I have faced myself with the fact that the MFC library offers nothing more to read the bitmap files, but the **CBitmap** class with very limited features. I needed features like scale, rotate, apply different filters, etc. but it was just not there. Another class, the **CDC** class (device context), offered some more options like stretching, transparency and alpha blending, but it was just too slow (proven many times by many different people).

So, my decision was to write another (yes, another) bitmap manipulation class that would not be MFC dependent, and would offer some extended features not present in the original MFC implementation. This class can load any 8, 16, 24 or 32bit bitmap, but internally works with 32bit bitmaps. The results can be saved as the 24bit bitmap on the hard disk, or it can be drawn on the provided device context (DC).

Background

There are many articles here on The Code Project considering this topic, so feel free to browse for them and compare the final results with this implementation.

Using the Code

Using this class is very simple, see below:

```
#include "BitmapEx.h"

// Load bitmap
CBitmapEx bitmapEx;
bitmapEx.Load(_T("Enter bitmap source file path here..."));

// Do whatever you need to do here

bitmapEx.Rotate(45);
bitmapEx.Sepia();
bitmapEx.Scale(50, 50);

// Draw the results on the screen (get hDC somewhere else)
bitmapEx.Draw(hDC);

// Save bitmap
bitmapEx.Save(_T("Enter bitmap destination file path here..."));
```

There are many **public** methods available. Please read the rest of the text which explains them all.

Create/Load/Save Bitmap

You can use this class to load any 8bpp to 32bpp bitmap from the hard disk or the memory stream. Also, you can load any bitmap from the Windows **HBITMAP** handle, but it has to have more than 8bpp. The class will convert it to the internally used 32bpp bitmap. After saving, the bitmap will be converted again to the 24bpp bitmap and will be saved to the hard disk or the memory stream. You can also save it to the **HBITMAP** handle as 32bpp bitmap. You can also create a blank (empty) bitmap using the methods provided.

To create/load/save bitmap to the hard disk or a memory stream, or the **HBITMAP** handle use the following methods:

```
void Create(long width, long height);
void Create(CBitmapEx& bitmapEx);
void Create(CBitmapEx* pBitmapEx);
void Load(LPTSTR lpszBitmapFile);
void Load(LPBYTE lpBitmapData);
void Load(HBITMAP hBitmap);
void Save(LPTSTR lpszBitmapFile);
void Save(LPBYTE lpBitmapData);
void Save(HBITMAP& hBitmap);
```

Using the first method, you can create the bitmap of almost any size (well, available RAM could be the limiting factor). This is a general 'blank bitmap'. Using the second and the third method, you can create a new bitmap from

the existing one (this is simulating the so called 'copy constructor'). In this way, you actually copy the existing bitmap in the memory. This could be useful when you do some bitmap filtering and you need the original bitmap backup. The next three methods give you an option to load the bitmap from the file or memory stream, or the **HBITMAP**. And finally, the last three methods provide you a simple way to save the bitmap to the file or memory stream or to the **HBITMAP** handle. Loading and saving the bitmap from/to the file stream (on the local or network drive) is a general operation. However, if you prefer having the dynamical images (generated in the memory) to be used by some web-server, loading and saving from/to some memory stream could be an option for you. If you need the results embedded inside the **HBITMAP** handle, then use the appropriate methods.

General Transformation Methods

This class offers some general bitmap transformation methods like scaling, rotation, cropping, flipping and mirroring. During the scaling and rotation operation, the bitmap details could get lost, so the three basic interpolation methods are implemented:

- Nearest neighbour (fastest, no interpolation, low quality)
- Bilinear (fast, better quality)
- Cubic B-spline (slow, best quality)

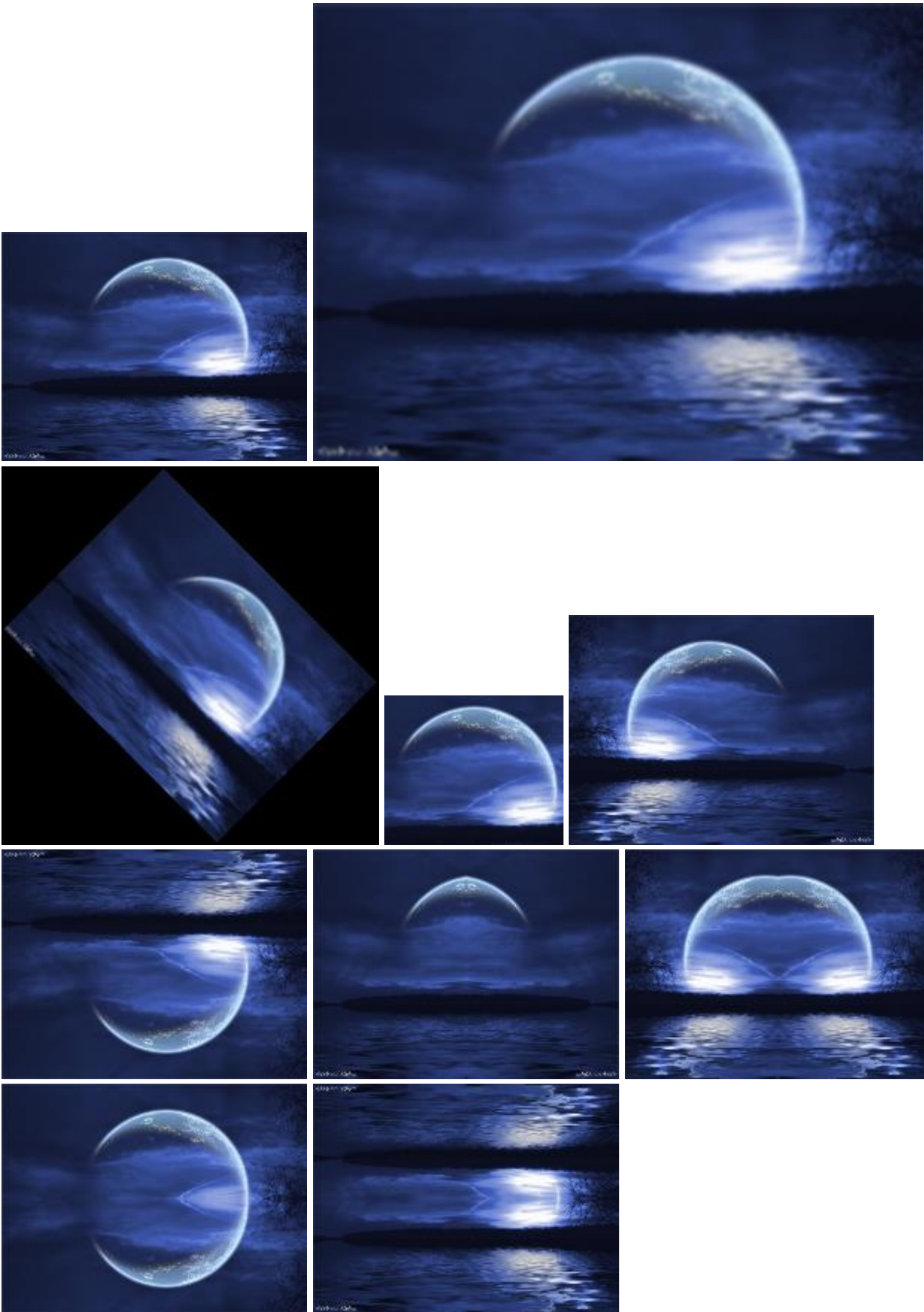
You can set the interpolation method using the **SetResampleMode()** member function. Here is the list of the transformation methods:

```
void Scale(long horizontalPercent=100, long verticalPercent=100);
void Rotate(long degrees=0, _PIXEL bgColor=_RGB(0,0,0));
void Crop(long x, long y, long width, long height);
void FlipHorizontal();
void FlipVertical();
void MirrorLeft();
void MirrorRight();
void MirrorTop();
void MirrorBottom();
```

You can scale the bitmap along the **x** and/or the **y** axis, rotate it from **0** to **360** percents, or perform flipping and/or mirroring. When you flip or mirror the bitmap, no interpolation is required since you don't change the original bitmap dimensions.

When you perform the bitmap rotation, you can set the background color. The default is **black**.

Here are some samples of the described transformation methods (Internet Explorer only: place mouse pointer above each image to see the applied transformation):



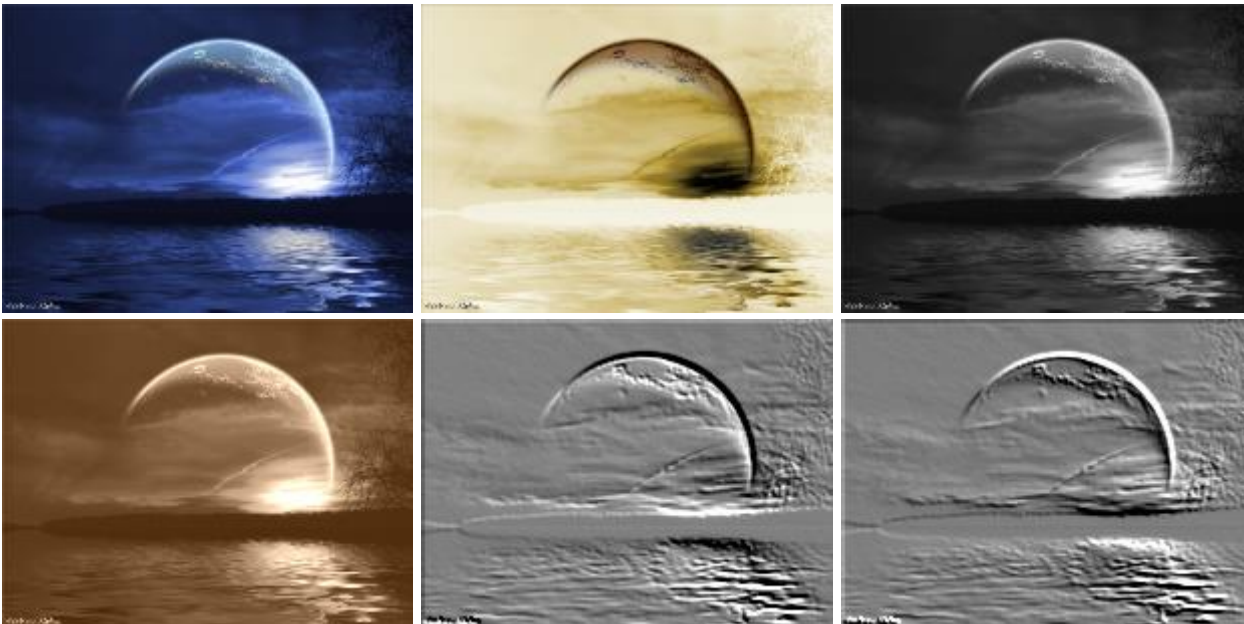
Simple Bitmap Filtering

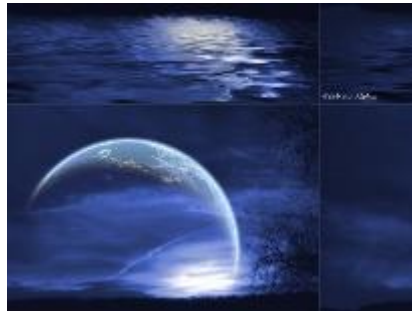
There are 22 simple filters implemented (and more to come) to do a simple bitmap convolution filtering. The **Clear()** methods will fill the bitmap with the color you provide.

```
void Clear(_PIXEL clearColor=_RGB(0,0,0));
void Negative();
void Grayscale();
void Sepia(long depth=34);
void Emboss();
void Engrave();
void Pixelize(long size=4);
void Brightness(long brightness=0);
void Contrast(long contrast=0);
void Blur();
void GaussianBlur();
void Sharp();
void Colorize(_PIXEL color);
void Rank(BOOL bMinimum=TRUE);
void Spread(long distanceX=8, long distanceY=8);
void Offset(long offsetX=16, long offsetY=16);
void BlackAndWhite(long offset=128);
void EdgeDetect();
void GlowingEdges(long threshold=2, long scale=5);
void EqualizeHistogram(long levels=255);
void Median();
void Posterize(long levels=4);
void Solarize(long threshold=128);
```

For some filters, like **Sepia()** and **Pixelize()** and so, you can change the input argument and get the different results on the output.

Here are some samples of the described filtering methods (Internet Explorer only: place mouse pointer above each image to see the applied filtering):





Drawing the Bitmap

The following list of methods make the core of the **CBitmapEx** class:

```
void Draw(HDC hDC);
void Draw(HDC hDC, long dstX, long dstY);
void Draw(long dstX, long dstY, long width, long height,
    CBitmapEx& bitmapEx, long srcX, long srcY);
void Draw(long dstX, long dstY, long width, long height,
    CBitmapEx& bitmapEx, long srcX, long srcY, long alpha);
void Draw(long dstX, long dstY, long dstWidth, long dstHeight,
    CBitmapEx& bitmapEx, long srcX, long srcY, long srcWidth, long srcHeight);
void Draw(long dstX, long dstY, long dstWidth, long dstHeight,
    CBitmapEx& bitmapEx, long srcX, long srcY, long srcWidth,
    long srcHeight, long alpha);
void Draw(_QUAD dstQuad, CBitmapEx& bitmapEx);
void Draw(_QUAD dstQuad, CBitmapEx& bitmapEx, long alpha);
void Draw(_QUAD dstQuad, CBitmapEx& bitmapEx, long srcX, long srcY,
    long srcWidth, long srcHeight);
void Draw(_QUAD dstQuad, CBitmapEx& bitmapEx, long srcX, long srcY,
    long srcWidth, long srcHeight, long alpha);
void DrawTransparent(long dstX, long dstY, long width, long height,
    CBitmapEx& bitmapEx, long srcX, long srcY, _PIXEL transparentColor=_RGB(0,0,0));
void DrawTransparent(long dstX, long dstY, long width, long height,
    CBitmapEx& bitmapEx, long srcX, long srcY, long alpha,
    _PIXEL transparentColor=_RGB(0,0,0));
void DrawTransparent(long dstX, long dstY, long dstWidth, long dstHeight,
    CBitmapEx& bitmapEx, long srcX, long srcY, long srcWidth,
    long srcHeight, _PIXEL transparentColor=_RGB(0,0,0));
void DrawTransparent(long dstX, long dstY, long dstWidth, long dstHeight,
    CBitmapEx& bitmapEx, long srcX, long srcY, long srcWidth,
    long srcHeight, long alpha, _PIXEL transparentColor=_RGB(0,0,0));
void DrawTransparent(_QUAD dstQuad, CBitmapEx& bitmapEx,
    _PIXEL transparentColor=_RGB(0,0,0));
void DrawTransparent(_QUAD dstQuad, CBitmapEx& bitmapEx, long alpha,
    _PIXEL transparentColor=_RGB(0,0,0));
void DrawTransparent(_QUAD dstQuad, CBitmapEx& bitmapEx, long srcX,
    long srcY, long srcWidth, long srcHeight, _PIXEL transparentColor=_RGB(0,0,0));
void DrawTransparent(_QUAD dstQuad, CBitmapEx& bitmapEx, long srcX,
    long srcY, long srcWidth, long srcHeight, long alpha,
    _PIXEL transparentColor=_RGB(0,0,0));
void DrawBlended(long dstX, long dstY, long width, long height,
    CBitmapEx& bitmapEx, long srcX, long srcY, long startAlpha,
    long endAlpha, DWORD mode=GM_NONE);
void DrawBlended(long dstX, long dstY, long dstWidth, long dstHeight,
    CBitmapEx& bitmapEx, long srcX, long srcY, long srcWidth,
    long srcHeight, long startAlpha, long endAlpha, DWORD mode=GM_NONE);
void DrawMasked(long dstX, long dstY, long width, long height,
    CBitmapEx& bitmapEx, long srcX, long srcY,
    _PIXEL transparentColor=_RGB(255,255,255));
void DrawAlpha(long dstX, long dstY, long width, long height,
    CBitmapEx& bitmapEx, long srcX, long srcY, long alpha,
    _PIXEL alphaColor=_RGB(0,0,0));
void DrawCombined(long dstX, long dstY, long width, long height,
    CBitmapEx& bitmapEx, long srcX, long srcY, DWORD mode=CM_SRC_AND_DST);
void DrawCombined(long dstX, long dstY, long dstWidth, long dstHeight,
    CBitmapEx& bitmapEx, long srcX, long srcY, long srcWidth,
    long srcHeight, DWORD mode=CM_SRC_AND_DST);
void DrawTextA(long dstX, long dstY, LPSTR lpszText, _PIXEL textColor,
    long textAlpha, LPTSTR lpszFontName, long fontSize,
    BOOL bBold=FALSE, BOOL bItalic=FALSE);
void DrawTextW(long dstX, long dstY, LPWSTR lpszText,
    _PIXEL textColor, long textAlpha, LPTSTR lpszFontName,
    long fontSize, BOOL bBold=FALSE, BOOL bItalic=FALSE);
```

You can see a number of different bitmap drawing methods. The one that has the source and destination size as input arguments will perform the bitmap drawing with scaling. The final quality and speed will depend on the interpolation method you have selected. In most cases, the **bilinear interpolation** is a good choice between the speed and the quality. This is similar to the **StretchBlt()** GDI method.

Besides the general drawing methods, there are some special ones. Using the **Draw()** function with **_QUAD** as an input argument, you can draw the bitmap in any polygon defined with four points. It is similar to the **PlgBlt()** GDI method, only offers some more freedom. That is, it can do 'projective' and not just 'affine' bitmap transformations. Some will find this very useful.

Another group of available bitmap drawing methods is the **DrawTransparent()** group of methods. It can do all that is described above; even a transparent color in the source bitmap can be specified. Additionally, one could perform a bitmap 'projective' transformation.

The next group is the **DrawBlended()** methods. They provide you the way to draw the source bitmap 'gradiently' over the destination bitmap. There are four types of gradients supported: horizontal, vertical, diagonal and radial.

Two special methods **DrawMasked()** and **DrawAlpha()** are used in special effects like shadows, etc.

Methods called **DrawCombined()** are used to perform logical operations between source and destination bitmap (similar to raster operation codes in GDI).

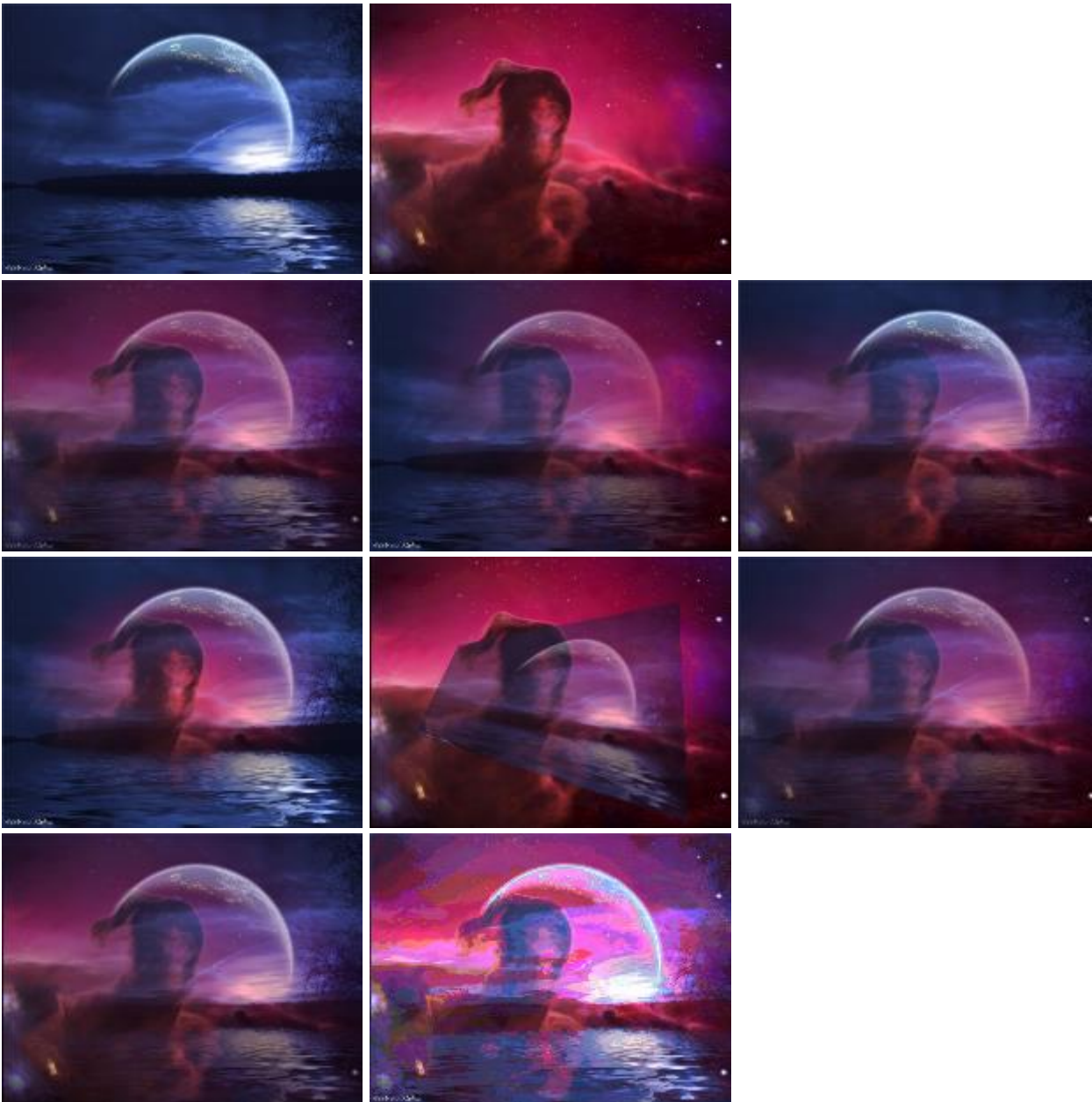
Methods called **DrawText()** are used to output ANSI and UNICODE text on the bitmap. They also internally perform the text antialiasing using glyph raster bitmaps.

All the bitmap drawing methods described offer an opacity parameter (alpha) that can be applied during the drawing operation. This is similar to using the **AlphaBlend()** GDI method.

By combining the different bitmap drawing methods, some really cool looking effects can be done.

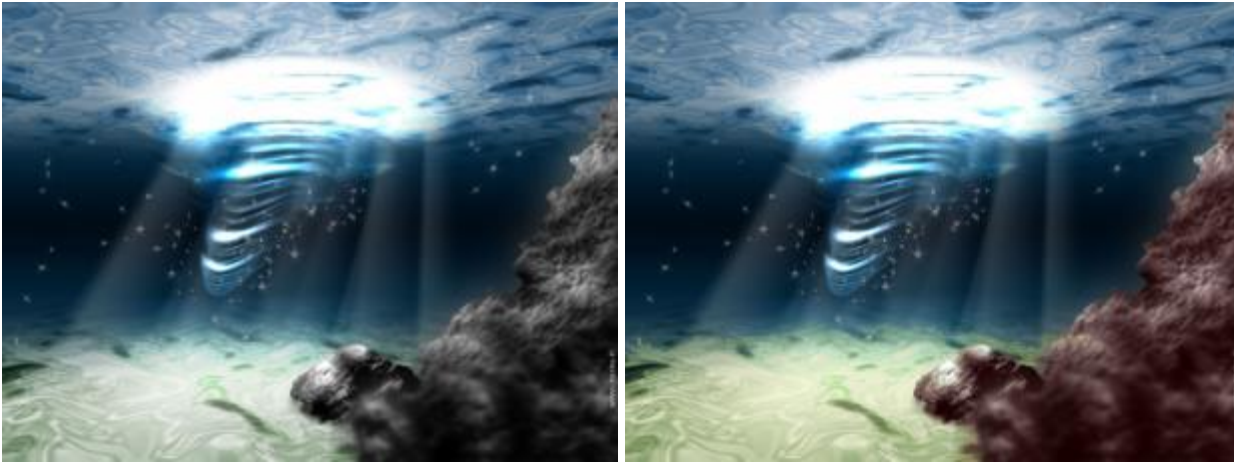
To output the bitmap on the screen (or any other device context) use the first two methods, with the **HDC** as an input argument.

Here are some samples of the described drawing methods (Internet Explorer only: place mouse pointer above each image to see the applied drawing):



Changing Colors in the Bitmap

This is a special method defined in the **CBitmapEx** class. It allows the developer to change the existing color in the bitmap with another one. The transparency and the 'strength' of the new color can be specified. Also, this operation can be done locally (on some region), or in the whole image. Please see the images below:



The following code snippet shows how this is done:

```
CBitmapEx bitmap;
bitmap.Load(_T("Original3.bmp"));
bitmap.ReplaceColor(345, 275, _RGB(255,0,0), 10, 150, FALSE); // Sea rock
bitmap.ReplaceColor(105, 305, _RGB(255,255,0), 10, 90, FALSE); // Bottom of the sea
bitmap.Save(_T("Original3_r.bmp"));
```

This processing could take some time, so please don't expect the final result at the moment. Also, to get the best output bitmap, it could request some experimenting with different options. Here is the definition on the `ReplaceColor()` method:

```
void ReplaceColor(long x, long y, _PIXEL newColor,
                 long alpha=20, long error=100, BOOL bImage=TRUE);
```

Actually, you take the pixel from the location `x` and `y` and change its color with `newColor`. Also, you specify the transparency level of the new color that is to be applied `alpha`, as also the 'strength', that is the `error` value. The last parameter `bImage` is applying the re-coloring of the whole image or just the region which is defined by the 'strength'. You have to experiment to see how this method is working on different images. This could be useful if you want to change the background color of the image, or to re-color the grayscale images.

Creating a Drop Shadow Effect

This is a very popular effect and can be achieved using the `CBitmapEx` class easily. Please see the images below:



The following code snippet shows how this is done:

```
CBitmapEx bitmap1, bitmap2, bitmap3;
bitmap1.Load(_T("Original4.bmp"));
bitmap2.Load(_T("Original5.bmp"));
bitmap3.Create(bitmap2.GetWidth(), bitmap2.GetHeight());
bitmap3.DrawMasked(0, 0, bitmap2.GetWidth(), bitmap2.GetHeight(),
    bitmap2, 0, 0, bitmap2.GetPixel(0, 0));
bitmap3.Blur();
bitmap3.Blur();
bitmap3.Blur();
bitmap1.DrawAlpha(4, 4, bitmap3.GetWidth(), bitmap3.GetHeight(),
    bitmap3, 0, 0, 25, _RGB(0,0,0));
bitmap1.DrawTransparent(0, 0, bitmap2.GetWidth(),
    bitmap2.GetHeight(), bitmap2, 0, 0, 50, bitmap2.GetPixel(0, 0));
bitmap1.Save(_T("Original4_s.bmp"));
```

This processing is quite quick so you would not have to wait a long time for the final results. The bottleneck of this process is the number of **Blur()** methods you call. Also, using the special method **DrawMasked()**, you can create a monochromatic mask of just any image with the transparent background. Another special method called **DrawAlpha()** renders the created mask in just any color (here we need the shadow to be black, right) and with the selected transparency level (the shadows are mainly semi-transparent).

Drawing Text on the Bitmap

You can use the **CBitmapEx** class to output some text on the bitmap. Please see the image below:



The following code snippet shows how this is done:

```
CBitmapEx bitmap;  
bitmap.Load(_T("Original4.bmp"));  
bitmap.DrawTextW(0, 0, _T("Enter your text here..."), _RGB(255,0,0),  
    50, _T("Times New Roman"), 12);  
bitmap.Save(_T("Original4_t.bmp"));
```

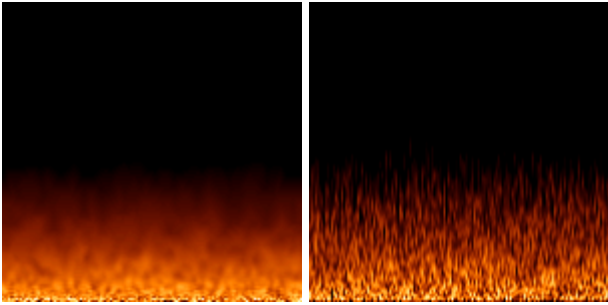
You can draw ANSI or UNICODE text, set text color and opacity, select different font and its size, and also make text **bold** or *italic*. You can use the **MeasureText()** method to obtain the text width and height.

Creating a Fire Effect

The **CBitmapEx** class can produce the 'burning fire' effect. Please see the code below:

```
CBitmapEx bitmap;  
bitmap.Create(300, 300);  
bitmap.Clear();  
bitmap.CreateFireEffect();  
// Call the following method for some time (i.e. a few seconds)  
bitmap.UpdateFireEffect();
```

See the results below:



Here is the definition of the methods used:

```
void CreateFireEffect();
void UpdateFireEffect(BOOL bLarge=TRUE, long iteration=5, long height=16);
```

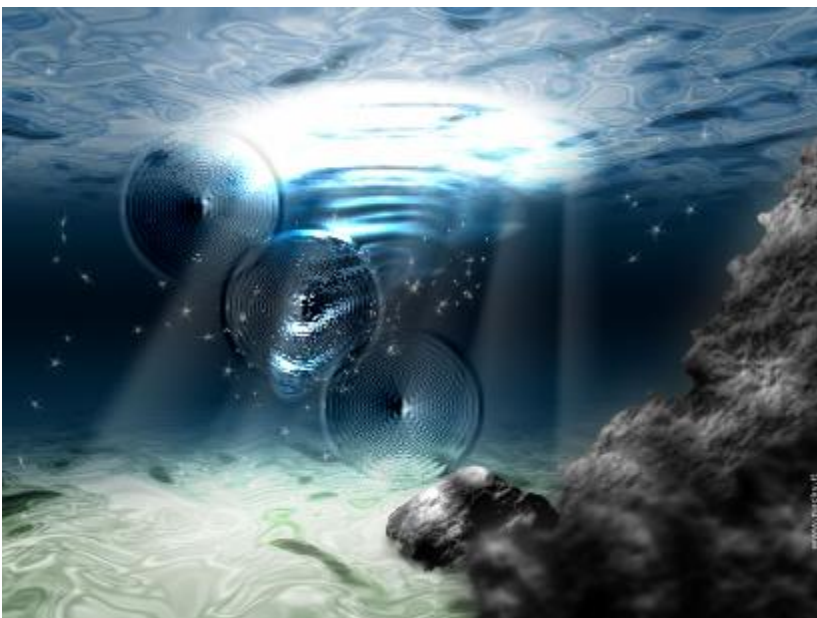
You control the fire (speed and size), by changing the params in the `UpdateFireEffect()` method. This method should be called in some thread or loop to get the 'burning fire' animation.

Creating a Water Effect

The `CBitmapEx` class can produce the 'water' effect. Please see the code below:

```
CBitmapEx bitmap;
bitmap.Load(_T("Undersea.bmp"));
bitmap.CreateWaterEffect();
bitmap.MakeWaterBlob(100, 100, 20, -500);
bitmap.MakeWaterBlob(150, 150, 20, 500);
bitmap.MakeWaterBlob(200, 200, 20, -500);
// Call the following method for some time (i.e. a few seconds)
bitmap.UpdateWaterEffect();
```

See the results below:



Here is the definition of the methods used:

```
void CreateWaterEffect();  
void UpdateWaterEffect(long iteration=5);  
void MakeWaterBlob(long x, long y, long size, long height);
```

You control the water (drops, speed, size), by changing the params in the `UpdateWaterEffect()` method and `MakeWaterBlob()` method. This method should be called in some thread or loop to get the 'water' animation.

Creating a Smoke (or Clouds) Effect

The `CBitmapEx` class can produce the 'smoke' (or 'clouds') effect. Please see the code below:

```
CBitmapEx bitmap;  
bitmap.Load(_T("Original4.bmp"));  
bitmap.CreateSmokeEffect();  
// Call the following method for some time (i.e. a few seconds)  
bitmap.UpdateSmokeEffect();
```

See the results below:



Here is the definition of the methods used:

```
void CreateSmokeEffect();  
void UpdateSmokeEffect(long offsetX=0, long offsetY=0, long offsetZ=0);
```

You control the smoke (or clouds) (speed, size), by changing the params in the `UpdateSmokeEffect()` method. This method should be called in some thread or loop to get the 'smoke' (or 'clouds') animation.

Bitmap from RGB to HSV and Back

Here is the list of methods that can be useful when converting bitmap from **RGB** color-space to **HSV** and vice-versa:

```

_PIXEL _RGB2HSV(_PIXEL rgbPixel);
_PIXEL _HSV2RGB(_PIXEL hsvPixel);
void ConvertToHSV();
void ConvertToRGB();
_COLOR_MODE GetColorMode() {return m_ColorMode;}

```

This color transformations were very useful when implementing the `ReplaceColor()` method described above.

Bitmap Information Methods

Various bitmap information methods are available, see the list below:

```

LPBITMAPFILEHEADER GetFileInfo() {return &m_bfh;}
LPBITMAPINFOHEADER GetInfo() {return &m_bih;}
long GetWidth() {return m_bih.biWidth;}
long GetHeight() {return m_bih.biHeight;}
long GetPitch() {return m_iPitch;}
long GetBpp() {return m_iBpp;}
long GetPaletteEntries() {return m_iPaletteEntries;}
LPRGBQUAD GetPalette() {return m_lpPalette;}
DWORD GetSize() {return m_dwSize;}
LPBYTE GetData() {return m_lpData;}
void SetResampleMode(RESAMPLE_MODE mode=RM_NEARESTNEIGHBOUR)
    {m_ResampleMode = mode;}
RESAMPLE_MODE GetResampleMode() {return m_ResampleMode;}
BOOL IsValid() {return (m_lpData != NULL);}
void GetRedChannel(LPBYTE lpBuffer);
void GetGreenChannel(LPBYTE lpBuffer);
void GetBlueChannel(LPBYTE lpBuffer);
void GetRedChannelHistogram(long lpBuffer[256], BOOL bPercent=FALSE);
void GetGreenChannelHistogram(long lpBuffer[256], BOOL bPercent=FALSE);
void GetBlueChannelHistogram(long lpBuffer[256], BOOL bPercent=FALSE);

```

Direct Pixel Access Methods

These important methods are also provided, look below:

```

_PIXEL GetPixel(long x, long y);
void SetPixel(long x, long y, _PIXEL pixel);

```

These methods are, however, slower than a direct memory access. You can get the bitmap memory buffer using the `GetData()` member function. Use this memory buffer to do any 'real-time' bitmap processing you plan. For testing purposes, it is safer to use direct pixel access methods.

Notice on Using the CBitmapEx Class

Using all these methods, you should be able to do any type of drawing and transformation you need. Well, almost any type. There are surely many more filters to add, and also many more transformations to apply.

Also, the comments of the readers were very useful while updating the original source code. New methods, like loading/saving from/to memory stream have been added, as well as another method for drawing on the custom device context (DC) using the insertion point, or scaling using pixels and not just percents.

Many of the original methods for drawing are now written in the 'inline assembler' to obtain the speed increase. The original source code is left commented for better understanding. All optimizations considering the ASM code (since I am not an expert on this) are welcome.

About the Demo Project

There is a special class, written just for the demo project, called **CEffectorBuilder** that is used to create animations. This class uses the original **CBitmapEx** class to load the bitmap files and then apply different transformations on them. The screenshot above shows the demo project screen output. There are many drawing and transformation effects available in this class so feel free to experiment. The final results could look just great.

Points of Interest

Working on this class, I have found the perfect way to speed-up any graphical operation many times. It is done using the fixed point arithmetic. It is applied everywhere in the **CBitmapEx** class, almost in every single operation that has divisions and multiplications, with floating point numbers. So, don't be surprised with the speed of the real-time animation you can achieve using only this class. Take a look at the demo project for an example of the real-time animation project that uses the **CBitmapEx** class.

License


This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

Download CBitmapEx source - 37.22 KB **Download Effector source - 70.66 KB**

About the Author



darkoman

Software Developer (Senior) Elektromehanika d.o.o. Nis
Serbia 

He has a master degree in Computer Science at Faculty of Electronics in Nis (Serbia), and works as a C++/C# application developer for Windows platforms since 2001. He likes traveling, reading and meeting new people and cultures.