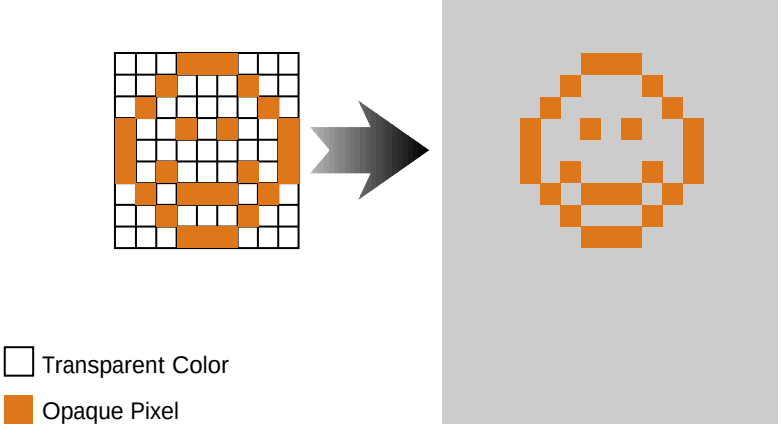
Attention:

Chris Hecker
definition six, incorporated
checker@d6.com
http://www.d6.com/users/checker

PS. The email address at the end of the article is incorrect. Please use checker@d6.com for any correspondence.

Changing The Rules for Transparent BLTs

Figure 1. A Transparent BLT



When I sit down to write an article, the first question I always ask myself is, "Who is going to read this?" No, I don't mean, "Who in their right mind would read this?" I mean who is the audience for this article, and how technical are they?

For this column, I'd like the answer to be "experienced programmers," and I intend to aim the content at just such a readership. My goal is to provide detailed coverage of specific game programming techniques and to present production-quality code, sometimes at the expense of less experienced developers who might want to read the code a few times and step through it in a debugger to see how it works. This is not to say I'll be cryptic, but I'm going to try to move fast enough to keep the advanced people interested, while giving the beginners something they'll need to think about for a bit before grasping all the issues, both explicit and implied. Let me know what you think via the contact information at the end of the article!

Transparency

Transparent block transfers (BLTs—pixel copies) are one of the more useful techniques for game programmers. A transparent BLT can be roughly defined as a block transfer where some pixels are not copied from the source to the destination, leaving destination pixels showing through. The list of effects you can generate with a simple transparent BLT is endless: sprites, floating text or game scores, cursors, shadows, floating maps, and the like. How are transparent BLTs implemented? We'll answer that question with working code, optimize the code, and write a transparent BLT that will handle both WinG DIB orientations as a bonus.

There are a number of ways you can implement transparent BLTs. The most common specifies a single pixel value in the source bitmap (the sprite, if you will) that will not be copied from the source to the destination. The BLT routine examines each pixel and decides whether it is the "transparent color" or whether it's an actual data value that needs to be copied to the destination bitmap, as shown in Figure 1. Other techniques include using a mask to specify which pixels are copied, using raster operations under Windows, and using special bitmap formats (like run length encoding) for the source sprite. When we start optimizing, we'll look into some of these other techniques and how they compare with the base technique.

First, we'll create a relatively naive

transparent BLT. I'm going to write the BLT for use under Windows (my preferred development environment), but it isn't Windows specific and should port to DOS or other platforms without problems. We'll use device independent bitmaps (DIBs), which are just in-memory bitmaps with a header describing their pixel format.

Our naive implementation will read every pixel in the source DIB, check for the transparent color, and optionally write the pixel to the destination DIB. Since we want this code to work well on Windows with WinG, we'll need to deal with the two possible destination "DIB orientations."

Orient Yourself

There are two DIB orientations, top-down and bottom-up. Top-down DIBs are arranged in memory much like the DOS Mode 13h frame buffer or your average DOS bitmap. The pointer to the DIB bits points to the topmost scanline on the DIB, and as the value of the pointer increases, it moves down the DIB surface. On the other hand, bottom-up DIBs are "upside-down," with the pointer referencing the bottom-most scanline, its value increasing as it moves up the DIB surface. Movement across scanlines from left to right is always accompanied by an increase in memory address; only the vertical movement is affected by the orientation. WinG chooses the fastest DIB orientation based on the run-time configuration, so code that expects the best performance must be prepared to deal with either type. This is actually quite easy in practice, and the technique I describe here draws to both orientations without any performance penalty.

Listing 1 shows our first transparent BLT. This code only handles 8 bits-per-pixel DIBs, but could you can easily extend it to other formats. Our initial inner loop looks like this:

```
for(Y = 0;Y < Height;Y++) {
for(X = 0;X < Width;X++) {
  if(*pSourceBits != TransparentColor) {
// not transparent?
          *pDestBits = *pSourceBits;
// copy the pixel
  }
  pDestBits++;
// advance to next pixels
  pSourceBits++;
  }
  pDestBits += DestDeltaScan;
// advance to next dest
  pSourceBits += SourceDeltaScan;
// and source pixels
}
```

We introduce the `DeltaScan` variables (`DestDeltaScan` and `SourceDeltaScan`) to enable top-down and bottom-up drawing. We always start the BLT from the top, and the `DeltaScans` move their respective pointers down the DIB surface from one scanline to the next. We set up the `DeltaScans` to move from the end of one processed span to the beginning of the next span, so we step directly to the next span of pixels to BLT without calculating a new `X` or `Y` offset from the start of the DIB, avoiding multiplies in the loop and other overhead. On top-down DIBs, the `DeltaScan` is positive (the "down" of "top-down" indicates the direction in which a positive pointer increment

Chris Hecker

What's a good concept to follow when you're working with transparent BLTs? If the rules forbid you from getting your images onscreen quickly enough, change the rules!

Listing 1. Simple Transparent BLT

```
#include<windows.h>
#include<assert.h>


void TransparentBlt( BITMAPINFOHEADER *pDestHeader, BYTE *pDestBits,
          int XDest, int YDest, BITMAPINFOHEADER *pSourceHeader,
          BYTE *pSourceBits, BYTE TransparentColor ){
   int DestDeltaScan, DestWidthBytes, DestRealHeight;
   int SourceDeltaScan, XSource = 0, YSource = 0;
   int Width, Height;

   DestWidthBytes = (pDestHeader->biWidth + 3) & ~3;   // dword align

   assert(pDestHeader->biSizeImage);      // insure biSizeImage is set

   if(pDestHeader->biHeight < 0){
      // dest is top-down
      DestRealHeight = -pDestHeader->biHeight;  // get positive height
      DestDeltaScan = DestWidthBytes;        // travel down dest
   }else{
      // dest is bottom-up
      DestRealHeight = pDestHeader->biHeight;
      DestDeltaScan = -DestWidthBytes;       // travel down dest
      // point to top scanline
      pDestBits += pDestHeader->biSizeImage - DestWidthBytes;
   }

   // pDestBits -> top scanline of dest
   // DestDeltaScan -> distance from scan to scan in dest

   // clip source to dest

   assert(pSourceHeader->biHeight < 0);   // assume top-down source DIB
   Width = pSourceHeader->biWidth;
   Height = -pSourceHeader->biHeight;

   if(XDest < 0){
      // left clipped
      Width += XDest;
      XSource = -XDest;
      XDest = 0;
   }

   if((XDest + Width) > pDestHeader->biWidth){
      //right clipped
      Width = pDestHeader->biWidth - XDest;
   }

   if(YDest < 0){
      // top clipped
      Height += YDest;
      YSource = -YDest;
      YDest = 0;
   }
```

Listing 1.

```
if((YDest + Height)>DestRealHeight)
{           // bottom clipped
   Height = DestRealHeight - /
YDest;
}

SourceDeltaScan = /
   (pSourceHeader->biWidth + 3)/
 & ~3;         // dword align

// step to starting source pixel
pSourceBits += /
(YSource * SourceDeltaScan) + /
XSource;

// step to starting dest pixel
pDestBits += /
(YDest * DestDeltaScan) + XDest;

// account for processed span in
// delta scans
SourceDeltaScan -= Width;
DestDeltaScan -= Width;

if((Height > 0) && (Width > 0))
{
   // we have something to BLT
   int X, Y;

   for(Y = 0;Y < Height;Y++) {
      for(X = 0;X < Width;X++) {
         if(*pSourceBits != /
TransparentColor) {
            // not transparent?
            *pDestBits = /
*pSourceBits; // copy the pixel
         }
         pDestBits++;
// advance to next pixels
         pSourceBits++;
      }
      pDestBits += DestDeltaScan;
// advance to next dest
      pSourceBits += /
SourceDeltaScan;
// and source pixels
   }
 }
}
```

travels), and the pointer increases through memory as we process the BLT. On bottom-up DIBs, the pointer needs to decrease to move down the surface, so the `DeltaScan` is negative.

Because we always want the BLT to start at the top of the DIBs, we need the pointers to start there, too. For top-down DIBs, this is no problem; the bits pointer already points to the top scanline. For bottom-up DIBs, we need to move the pointer from the bottom scanline to the top using the following expression:

```
pDestBits += pDestHeader->
 biSizeImage - DestWidthBytes;
```

This adds the size of the DIB in bytes to the pointer—bringing it past the top scanline—and subtracts the width of a single scan to bring the pointer back onto the DIB, leaving it pointing at the beginning of the top scanline.

The last bit of code in Listing 1 (before the actual BLT) clips the source to the destination. We step through the extents, adjusting the source and destination offsets and the width and height when necessary. Finally, if we have pixels to draw after the clip, we go into our loop.

Change the Rules

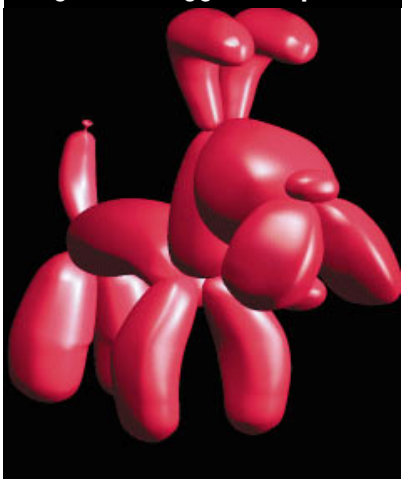
Now that we've got the setup code out of the way, we can try to optimize the inner loop. The first question we must ask is always, "Do I need to optimize the inner loop?" If this code is just supposed to draw a score on top of a bitmap the answer might be no. But if that were the case, this would be a short column, so let's assume this code is our program's bottleneck.

Many people, including myself, make the same mistake over and over again when they start to optimize a piece of code. They usually look at the C version they have working and start rewriting it in assembly language, without taking a step back to ask themselves, "What's this algorithm really doing?"

The key to writing code that runs very fast is *not* to optimize code that obeys the current set of rules and structure you've imposed on it, the key is to *change* the rules. My favorite scene from *Star Trek 2: The Wrath of Khan* is the one where Bones introduces Kirk to a young Starfleet Academy graduate as the only person who has ever aced the final exam, the Kobiashi Maru. When the graduate asks Kirk how it is possible he beat a test that's specifically programmed to be unbeatable, Kirk replies that he sneaked into the testing room the night before his exam and reprogrammed the computer. Kirk would make a great optimizer.

Let's step back and see if we can change the rules. The answer to "What's this algorithm really doing?" is not,

Figure 2. Doggie2.bmp



"Checking every byte for the transparent color and copying it if necessary." That just happens to be the way the current implementation works. The real answer for most sprite-type source bitmaps is, "Skipping a bunch of transparent bytes, copying some data bytes, skipping some more, and then doing it all over again." If we understand this latter answer, a whole range of optimization opportunities open up to us.

We can take advantage of these opportunities by examining the way our current implementation deals with common input data and looking for ways to change it for the better. Most sprites are irregular shapes with transparent areas on the sides of the bitmap and pixel data in the center. Let's take an example scanline from such a sprite. These values are in hex:

```
FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF 01 01 02 02 03 03 03 04 04 04 03
03 03 02 01 FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF
```

If we assume `FF` is our transparent color, the current code will loop through these 46 bytes and skip 31 of them because they're transparent. In other words, it's spending 67% of its time on this scan deciding to do nothing. The other 33% of the time, it's checking for the transparent color when all it needs to do is copy the data. The amount of transparent color per scanline is obviously dependent on your sprite artwork; I'm using the `doggie2.bmp` bitmap (as shown in Figure 2) supplied with the WinG SDK, which is a fairly typical sprite image.

The "rule change" we need so we can take advantage of the source redundancy is a change to the source bitmap format. Instead of storing each pixel separately (and processing each pixel separately in the BLT), let's use a compression technique to encode pixel spans compactly. This technique is called run length encoding (RLE).

There are many forms of RLE, but most use a few different "token" types to compress bitmaps. Common tokens include `Run` records, which have a value and a number of pixels to copy that value in the destination; `Copy` records, which tell the decompressor to copy a series of pixels from the source like a normal BLT; and `Skip` (or `Jump`) records, which give a number of pixels to skip in the destination. (You can find documentation for one type of RLE format in the Windows SDK documentation under BITMAPINFO. The PCX file format is another RLE format commonly used on PCs.)

I've defined a simple RLE format for compressing our source bitmap, with the tokens shown in Table 1. Each record is a `DWORD` in the source bitmap, with the high word specifying the type of the token, and the low word specifying the run length for each token.

Table 1. RLE Tokens

NEWLINE	0000NNNN	NNNN=number of bytes to next NEWLINE record
SKIPRUN	0001NNNN	NNNN=number of pixels to skip in destination
COPYRUN	0002NNNN	NNNN=number of pixels to copy from source to destination

Here is the same scanline encoded with this RLE format:

```
0000001F 0001000F 0002000F 01 01 02 02
03 03 03 04 04 04 03 03 03 02 01
00010010
```

Now, instead of checking every byte as it transfers, the code can look at each record. If it's a `SKIPRUN`, the decompressor just increments the destination pointer, skipping over the pixels that wouldn't be drawn anyway (they're transparent in the source), and if it's a `COPYRUN`, the decompressor copies the pixels without checking for the transparent color. Plus, although we're not concerned with size compression right now, this encoding is only 31 bytes long, compared to 46 bytes for the raw scanline.

Instead of using `SKIPRUN`s to compress transparent runs, an alternative encoding would use another record type, the `COLORRUN`. This record encodes a strip of pixels with the same value. If we used `COLORRUN`s, we'd be able to change the transparent color on-the-fly to make new parts of the source bitmap invisible, but our decompressor would need to treat `COLORRUN`s differently depending on whether they encoded the transparent color or not.

In an RLE bitmap, each scanline is a different length in memory, so it's sometimes hard to find a certain line. The `NEWLINE` record makes clipping and subrectangle BLTing much easier. If we want to skip to a certain line, we start at the first scanline and move from `NEWLINE` to `NEWLINE` until we get to the one we want.

Listing 2 shows the new transparent BLT, `TransparentBltRLE`. The setup code for the destination and the clipping calculations stay the same, and both were copied from Listing 1. The actual inner loop looks a lot different from Listing 1 because we need to parse the source RLE. Clipping an RLE bitmap in the X-axis gets interesting; we need to loop over the records until we find one that intersects our BLT rectangle, process the "active" portion (the portion that actually intersects), then start the BLT loop on the next record.

Listing 3 is the RLE compressor, `CompressSprite`. It's a fairly simple state machine that writes out records on state transitions from `SKIPRUN`s to `COPYRUN`s or vice versa. This code could use a bit of work. It doesn't shrink the allocated memory after compressing the sprite, for example. We'll discuss other optimizations to the format below.

Numbers

Listing 2 is significantly faster than Listing 1 when BLTing the doggie. Table 2 contains some performance numbers (for 1,000 iterations). Listing 2 is two times faster than Listing 1. More interesting still, Listing 2 is almost twice as fast as fast32.asm, the assembly language transparent BLT we shipped with the WinG SDK! Fast32.asm is basically an optimized 386 assembly language version of Listing 1, and it uses some special techniques to increase speed, but it's clear that changing the rules gives a much bigger payoff than just brute force optimization or assembly language.

Give Me More

If we want to max out Listing 2, there are a number of other techniques to consider. You'll notice that if a source scanline looks like this:

```
FF 01 FF 01 FF 01 FF 01 FF
```

`CompressSprite` will generate this:

```
0000002C 00010001 00020001 01 00010001
00020001 01 00010001 00020001 01
00010001 00020001 01 00010001
```

This is definitely a waste of space and almost certainly a speed loss, too. To fix this case, we could extend our RLE format to contain a transparent color, and instead of simply copying the `COPYRUN` data bytes with `memcpy`, as we did in Listing 2, we could run the equivalent of Listing 1's inner loop on them. This gives us the benefits of both techniques. We could go even farther and make a new record type, `TRANSCOPYRUN`, for runs that contain pixels with interspersed transparent colors and keep `COPYRUN` for plain copies so we don't slow down the normal nontransparent runs. Our compressor would have to be smarter, too. It would look at the data and make a decision about whether it is better to compress a run of transparency with a `SKIPRUN` or to simply embed the transparent pixels in a `TRANSCOPYRUN`.

Obviously, well-written assembly language code would make things faster as well, but we could probably optimize the C code without resorting to assembly language and still get some more performance. For example, we could `DWORD` align our copies, we could unroll once or twice (although on a Pentium especially, this probably wouldn't be a big win and it would make our code bigger and less cacheable), and we could redesign the RLE format so we read less (using `DWORD` tokens is wasteful in most circumstances). Another option to consider is compiling code to do the transparent BLT, so instead of our sources being bitmaps, they'd be blocks of code that draw the sprite directly. Fast32.asm uses hysteresis to speed things up, and we could put that in our RLE decompressor as well.

Hysteresis is basically "stickiness," or a tendency to stay the same. For example, when I'm awake, I tend to stay

Table 2. Timing Numbers

Listing 1	7,100 ms
Listing 2	2,414 ms
fast32.asm	6,950 ms

[Fast32.asm is from the WinG SDK doggie sample application.]

awake for a long time, and when I'm asleep in bed, I stay there, too. You can use hysteresis in transparent BLTing by recognizing that when you're in a transparent run you'll probably be there for a while, and similarly, when you're copying pixels, you'll do that for a bit rather than switching between the two. Of course, our RLE format takes advantage of a lot of this redundancy, so hysteresis might not make much sense for our decompressor.

The only way to know is to understand your data and truthfully answer the question, "What's this algorithm really doing?"

Actually, you need to answer this question in two parts. The first, as we discussed, is understanding what the algorithm is supposed to do, not what the current implementation does. The second part comes in when you've decided on an optimization strategy, and is best summarized by Michael Abrash's quote from *Zen of Code Optimization* (Coriolis, 1994), "Assume nothing!" Time your algorithms, don't assume certain performance. I use the `timeGetTime` API on Windows, which returns millisecond-accurate timings, and Michael uses the Zen Timer, but whatever you do, time your results.

One Last Word

In the future, I plan to cover (in a technical way, naturally) digital wave audio mixing, perspective texture mapping, animated cursors, and maybe some wacky 32-bit programming hacks under 16-bit Windows. Write and let me know what you think or, better yet, post to rec.games.programmer or the CompuServe GAMDEV forum so everyone can join in. I also hang out on BIX in Michael Abrash's ibm.pc/fast.code conference, simply the best place to discuss optimization I've ever seen. ■

Chris Hecker works for a large software company in the Pacific Northwest. He can't mention the name because then he'll need all sorts of disclaimers. It's just a coincidence that he can be reached at checker@microsoft.com or through Game Developer *magazine.*

Listing 2. RLE Transparent BLT (Continued on p. 20)

```
#include<windows.h>
#include<windowsx.h>
#include<string.h>
#include<assert.h>



#define ISSKIPRUN( Record ) (int)((((DWORD)(Record)) & 0xFFFF0000) ==
0x00010000)
#define ISCOPYRUN( Record ) (int)((((DWORD)(Record)) & 0xFFFF0000) ==
0x00020000)

#define RUNLENGTH( Record ) (int)(((DWORD)(Record)) & 0xFFFF)


void TransparentBltRLE( BITMAPINFOHEADER *pDestHeader, BYTE *pDestBits,
          int XDest, int YDest, BITMAPINFOHEADER *pSourceHeader,
          BYTE *pSourceBits, BYTE TransparentColor ){
   int DestDeltaScan, DestWidthBytes, DestRealHeight;
   int XSource = 0, YSource = 0;
   int Width, Height;

   DestWidthBytes = (pDestHeader->biWidth + 3) & ~3;   // dword align

   assert(pDestHeader->biSizeImage);      // insure biSizeImage is set

   if(pDestHeader->biHeight < 0){
      // dest is top-down
      DestRealHeight = -pDestHeader->biHeight;  // get positive height
      DestDeltaScan = DestWidthBytes;        // travel down dest
   }else{
      // dest is bottom-up
      DestRealHeight = pDestHeader->biHeight;
      DestDeltaScan = -DestWidthBytes;       // travel down dest
      // point to top scanline
      pDestBits += pDestHeader->biSizeImage - DestWidthBytes;
   }

   // pDestBits -> top scanline of dest
   // DestDeltaScan -> distance from scan to scan in dest

   // clip source to dest

   assert(pSourceHeader->biHeight < 0);      // assume top-down source DIB
   Width = pSourceHeader->biWidth;
   Height = -pSourceHeader->biHeight;

   if(XDest < 0){
      // left clipped
      Width += XDest;
      XSource = -XDest;
      XDest = 0;
   }

   if((XDest + Width) > pDestHeader->biWidth){
      //right clipped
      Width = pDestHeader->biWidth - XDest;
```

Listing 2. (Continued on p. 21)

```
}

if(YDest < 0){
   // top clipped
   Height += YDest;
   YSource = -YDest;
   YDest = 0;
}

if((YDest + Height) > DestRealHeight){
   // bottom clipped
   Height = DestRealHeight - YDest;
}

// step to starting dest pixel
pDestBits += (YDest * DestDeltaScan) + XDest;

// account for span in delta scans
DestDeltaScan -= Width;

if((Height > 0) && (Width > 0)){
   // we have something to BLT
   int X, Y;
   DWORD *pCurrentSourceScan = (DWORD *)pSourceBits;

   // prestep to starting source Y

   for(Y = 0;Y < YSource;Y++){
      pCurrentSourceScan = (DWORD *)((BYTE *)pCurrentSourceScan +
                           RUNLENGTH(*pCurrentSourceScan));
   }

   for(Y = 0;Y < Height;Y++){
      DWORD *pCurrentSourceRecord = pCurrentSourceScan + 1;

      // prestep to starting source X

      X = 0;

      while(X < XSource){
         X += RUNLENGTH(*pCurrentSourceRecord);

         if(X > XSource){
            // we need to partially process the current record

            int Overlap = X - XSource;
            int ActiveOverlap = (Overlap > Width) ? Width : Overlap;

            if(ISCOPYRUN(*pCurrentSourceRecord)){
               // copy overlap pixels to destination

               // get pointer to data
               BYTE *pCopyRun = (BYTE *)pCurrentSourceRecord + 4;

               // prestep to desired pixels
               pCopyRun += RUNLENGTH(*pCurrentSourceRecord) - Overlap;

               memcpy(pDestBits,pCopyRun,ActiveOverlap);
```

Listing 2. (Continued from p. 20)

```
            }

            // skip to next dest pixel
            pDestBits += ActiveOverlap;
         }

         // skip to next record

         if(ISCOPYRUN(*pCurrentSourceRecord)){
            // skip any data bytes
            pCurrentSourceRecord =
               (DWORD *)((BYTE *)pCurrentSourceRecord +
                  RUNLENGTH(*pCurrentSourceRecord));
         }

         pCurrentSourceRecord++;             // skip record itself
      }

      X = X - XSource;

      while(X < Width){
         int RunLength = RUNLENGTH(*pCurrentSourceRecord);
         int RemainingWidth = Width - X;
         int ActivePixels = (RunLength > RemainingWidth) ?
                            RemainingWidth : RunLength;

         if(ISCOPYRUN(*pCurrentSourceRecord)){
            // copy pixels to destination

            // get pointer to data
            BYTE *pCopyRun = (BYTE *)pCurrentSourceRecord + 4;

            memcpy(pDestBits,pCopyRun,ActivePixels);
         }

         // skip to next dest pixel
         pDestBits += ActivePixels;

         // skip to next record

         if(ISCOPYRUN(*pCurrentSourceRecord)){
            // skip any data bytes
            pCurrentSourceRecord =
               (DWORD *)((BYTE *)pCurrentSourceRecord + RunLength);
         }

         pCurrentSourceRecord++;             // skip record itself

         X += RunLength;
      }

      pDestBits += DestDeltaScan;

      pCurrentSourceScan = (DWORD *)((BYTE *)pCurrentSourceScan +
                           RUNLENGTH(*pCurrentSourceScan));
   }
  }
}
```

Listing3. RLE Compressor

```
#include<windows.h>
#include<windowsx.h>
#include<string.h>
#include<assert.h>

#define NEWLINE( Length ) /
((DWORD)(0x00000000 | /
(short unsigned)(Length)))

#define SKIPRUN( Length ) /
((DWORD)(0x00010000 | /
(short unsigned)(Length)))

#define COPYRUN( Length ) /
((DWORD)(0x00020000 | /
(short unsigned)(Length)))

BYTE *CompressSprite(
   BITMAPINFOHEADER *pSourceHeader,
   BYTE *pSourceBits,
   BYTE TransparentColor ){
   int SourceWidthBytes = /
(pSourceHeader->biWidth + 3) & ~3;
   void *pOutputBuffer = /
GlobalAllocPtr/
(GHND,pSourceHeader->biSizeImage);
   DWORD *pOutputRecord = /
(DWORD *)pOutputBuffer;
   BYTE *pOutputByte;
   int X, Y;

   assert(pOutputBuffer);

   for(Y = 0;/
Y < pSourceHeader->biHeight;Y++){
      int Width = /
pSourceHeader->biWidth;
      enum state { InSkipRun, /
InCopyRun } State;
      BYTE *pSourceByte = /
pSourceBits;
      DWORD *pNewlineRecord = /
pOutputRecord++;
      int LineLength = 4;
      int CurrentRunLength = 1;

      pOutputByte = /
(BYTE *)(pOutputRecord + 1);

      if(*pSourceByte ==/
TransparentColor){
      // we´re starting a skip run
         State = InSkipRun;
         LineLength += 4;
      }else{
      // source is data
         // we´re starting a copy
  run
         State = InCopyRun;
         *pOutputByte++ = *pSourceByte;
         LineLength += 5;
      }

      pSourceByte++;

      for(X = 1;X < Width;X++){
         if(*pSourceByte == TransparentColor){
            if(State == InSkipRun){            // still in skip run
               CurrentRunLength++;
            }else{                        // changing to skip run
               // write out copy record
               *pOutputRecord = COPYRUN(CurrentRunLength);
               pOutputRecord = (DWORD *)pOutputByte;

               CurrentRunLength = 1;
               State = InSkipRun;
               LineLength += 4;
            }
         }else{    // source is data
            if(State == InCopyRun){            // still in copy run
               CurrentRunLength++;
               *pOutputByte++ = *pSourceByte;
               LineLength++;
            }else{                        // changing to copy run
               // write out skip record
               *pOutputRecord = SKIPRUN(CurrentRunLength);
               pOutputRecord++;
               pOutputByte = (BYTE *)(pOutputRecord + 1);

               CurrentRunLength = 1;
               State = InCopyRun;
               *pOutputByte++ = *pSourceByte;
               LineLength += 5;
            }
         }

         pSourceByte++;
      }

      // finish off current record

      if(State == InSkipRun){
         *pOutputRecord = SKIPRUN(CurrentRunLength);
         pOutputRecord++;
      }else{ // InCopyRun
         *pOutputRecord = COPYRUN(CurrentRunLength);
         pOutputRecord = (DWORD *)pOutputByte;
      }

      *pNewlineRecord = NEWLINE(LineLength);

      pSourceBits += SourceWidthBytes;
   }

   return (BYTE *)pOutputBuffer;
}
```

Listing3.