Technical Notes on Object IFF File Format

Franck Le Ouay (top portion)
Jamie Doornbos (bottom portion)
19- Aug- 99

The IFF files are used to store any information relative to an object (bitmaps, code, description, cost, …).
The information is organized in independent blocks of data. Each block has a header and a data structure.
The header defines the type of data using a 4 bytes id. This can be :
‘SPR#’ , ‘SPR2’ for compressed sprites
‘PALT’ for palette table
‘OBJD’ for object data
The header contains other information such as the length of the following block of data.

Now, to load the bitmaps from an IFF file, you must know the Id of the object, and the number of the texture you want to load.

```
IFFResFile2 file;
char FileName [_MAX_PATH];
int Id= Id of the object;
int Number= Number of the current sprite in the list;
HBITMAP H;

ErrType err = file.iResFile::Open( (FileName, iResFile::kJustOpen );

if (err) {
    MessageBox( NULL, "Couldn’t open that file", "Error:", MB_OK );
    return;
}


H= GetSpritePreview(&file,Id, Number, 0x00000000); // 0x00000000 is the background color
0x00RRGGBB
```

```
The process to read or write on a file is simple.
-      get a handle to the block of data you want : MHandle h = file.GetByID('SPR2', Id,
NULL);
lock the data : MPtr p = Memory::HLock( h );
read/write in the memory allowed
unlock the data : Memory::Hunlock(h);
release the handle : file.Release(h);

Exporting an object is another problem. First you must duplicate a template object so
that you have the z-buffer and all the code needed. Then you create your new sprite,
compress it, create your new palette and copy them into the file. The problem is that I
must include in my project a lot of  files from the game to import and export objects.
All the resources are spread in different files.
Here is an example. You can see that this code needs a lot of resource definitions.
```

```
void ExportPaintingFile(char * InputName, INT CurrentTile, BITMAPINFO * pbmi)
{
    IFFResFile2 file;
    FileName TemplateFileName, FName, objName;
    StubObject newObj;
    char Dir[_MAX_PATH], Name[_MAX_PATH];
    INT CatalogID=2000;

    CCatalogDlg CatalogDlg;
    if (CatalogDlg.DoModal()!=IDOK) return;


    CFileDialog * SaveFile=new CFileDialog( FALSE, ".iff", InputName, OFN_HIDEREADONLY |
OFN_OVERWRITEPROMPT,
                                                                  "Painting Files (*.iff)|
*.iff||", NULL);
```

```
if (SaveFile->DoModal()!=IDOK)
{
    delete SaveFile;
    return;
}
strcpy(Name,(char *) (LPCTSTR)  SaveFile->GetPathName());
DeleteFile(Name);

delete SaveFile;


FName.append(Name);
strcpy(Dir,RootDir);
strcat(Dir, "\\PaintingTmpl.iff");
TemplateFileName.append( Dir);

ExtractFileName(FName, objName);
// Set the name of the object
newObj.SetName(objName);

// Creates a new stub object
ErrType err= newObj.CreateNew(TemplateFileName, FName);
if (err) {
    MessageBox( NULL, "Couldn't open that file", "Error:", MB_OK );
    return;
    }

err = file.iResFile::Open( FName, iResFile::kJustOpen );

if (err) {
    MessageBox( NULL, "Couldn't open that file", "Error:", MB_OK );
    return;
    }

CatalogResource *cr = new CatalogResource();
cr->Load(&file, CatalogID);

for (int i=0; i<300; i++)
{
    if (CatalogDlg.DescriptionBuf[i]==13)
    {
        for (int j=i; j<300; j++)
            CatalogDlg.DescriptionBuf[j]=CatalogDlg.DescriptionBuf[j+1];
    }
}
cr->SetDescription(CatalogDlg.DescriptionBuf);
cr->SetName(CatalogDlg.TitleBuf);

// save the catalog data
cr->Save(&file, CatalogID);
delete cr;


// load the object data and save the price
MHandle objectDef= file.GetByIndex('OBJD', 1, ObjDefinition::Swizzle);
if (!objectDef) {
    MessageBox(NULL, "error", "", MB_OK);
    return;
} else
{
    ObjDefinition* def = (ObjDefinition*)Memory::HLock(objectDef);
    def->price= atoi(CatalogDlg.CostBuf);
    Memory::HUnlock(objectDef);
    file.Write(objectDef);
}

file.Release(objectDef);
delete CatalogDlg;

INT Count;
```

```
    FileName name;
    ResourceName ResName;
    HBITMAP H[12][2];
    CBitmap cBitmap;

    INT Id=100;

    Count=CountSpritesInList( &file, Id);
    if (Count!=12) return;

    for (i=0; i<Count; i++)
    {
        H[i][0]=NULL;
        if ( ((i&3)==1) || ((i&3)==2) ) H[i][0]= GetSpritePreview(&file,Id, i,
0x00FFFFFF);
        H[i][1]=NULL;
        if ( ((i&3)==1) || ((i&3)==2) ) H[i][1]= GetSpritePreview(&file,Id, i, 0x00FFFFFF,
true);
    }

    MHandle aSprite2List = Memory::HAlloc( Sprite2List::HeaderSize() *
                                                                4 * (12) );
    XASSERT( aSprite2List != NULL, "HAlloc failed!");

    Sprite2List *aList = (Sprite2List *) Memory::HLock( aSprite2List );
    XASSERT( aList != NULL, "HLock failed!");
    aList->mVersion = cRenderer::kEndOfOldVersions + 1;
    aList->mCount = Count;
    aList->mPaletteID =0;
    Memory::HUnlock( aSprite2List );

    MHandle aSprite2;

    for (i=0;i<Count; ++i)

    {
        HBITMAP target;

        if ( ((i&3)==1) || ((i&3)==2) ) target = hPaintings[CurrentTile-
(MAXTILES+1+MAXWALLS+1)][(i&3)-1+(i/4)*2];
        else target = H[i][0];

        if ( (target) && (H[i][1]) )
        {
            cOffscreen pixels(target);
            cOffscreen z(H[i][1]);

            Rect TempRect; //, TempRect2;
            BITMAP bm;

            GetObject(target, sizeof(bm), &bm);      // get size of bitmap
            TempRect.left= TempRect.top=0;
            TempRect.right=bm.bmWidth;
            TempRect.bottom=bm.bmHeight;


            aSprite2 =
                CompressZPixelsToSprite2(
                                pixels,
                                z,
                                TempRect,
                                TempRect);

            UInt32 oldSize = Memory::HGetSize( aSprite2List );
            UInt32 newSize = oldSize + Memory::HGetSize(aSprite2);

            XASSERT( (oldSize & 1) == 0, "old size should always be even");
            XASSERT( (newSize & 1) == 0, "new size should always be even");

            // scribble offset into Sprite2List
            MPtr p = Memory::HLock( aSprite2List );
            ((Sprite2List *) p)->mOffsetTable[i] = oldSize;
            Memory::HUnlock( aSprite2List );

            // resize Sprite2List to new correct size
            ErrType err = Memory::HSetSize( aSprite2List, newSize );
            XASSERT( err == 0, "HSetSize failed!");


            // copy latest Sprite2 into Sprite2List
            p = Memory::HLock( aSprite2List );
            MPtr q = Memory::HLock( aSprite2 );

            memcpy( (char *) p + oldSize, q, newSize-oldSize);

            Memory::HUnlock( aSprite2List );
            Memory::HUnlock( aSprite2 );
            // give back the Sprite2 the compressor created
            //Memory::HFree( aSprite2 );
            Memory::HFree(aSprite2);
        } else
        {

            MHandle aSprite2 = Memory::HAlloc(sizeof (MHandle));
            MPtr p = Memory::HLock(aSprite2);
            (( Sprite2 *) p)->mHeight=0;
            (( Sprite2 *) p)->mWidth=0;
            Memory::HUnlock(aSprite2);
            UInt32 oldSize = Memory::HGetSize( aSprite2List );
            UInt32 newSize = oldSize + Memory::HGetSize(aSprite2);

            XASSERT( (oldSize & 1) == 0, "old size should always be even");
            XASSERT( (newSize & 1) == 0, "new size should always be even");

            // scribble offset into Sprite2List
            p = Memory::HLock( aSprite2List );
            ((Sprite2List *) p)->mOffsetTable[i] = oldSize;
            Memory::HUnlock( aSprite2List );

            // resize Sprite2List to new correct size
            ErrType err = Memory::HSetSize( aSprite2List, newSize );
            XASSERT( err == 0, "HSetSize failed!");


            // copy latest Sprite2 into Sprite2List
            p = Memory::HLock( aSprite2List );
            MPtr q = Memory::HLock( aSprite2 );

            memcpy( (char *) p + oldSize, q, newSize-oldSize);

            Memory::HUnlock( aSprite2List );
            Memory::HUnlock( aSprite2 );
            // give back the Sprite2 the compressor created
            //Memory::HFree( aSprite2 );
            Memory::HFree(aSprite2);

        }

    }

    file.Add(aSprite2List, 'SPR2', Id, "User Created Painting", false);
    file.Update();


    file.Release(aSprite2List);

    int palette_id = 0;
    MHandle h= file.GetByID('PALT', palette_id, NULL);
    if (!h) return;

    PaletteResource *pal = (PaletteResource *) Memory::HLock(h);
    for (int j=0; j<pal->mColorsUsed; j++)
    {
        pal->mPalette[j][0]=pbmi->bmiColors[j].rgbRed;
        pal->mPalette[j][1]=pbmi->bmiColors[j].rgbGreen;
        pal->mPalette[j][2]=pbmi->bmiColors[j].rgbBlue;
    }

    Memory::HUnlock(h);

    file.Write(h);
    file.Release(h);

    for (i=0; i<Count; i++)
    {
        if (H[i][0]) { DeleteObject(H[i][0]); H[i][0]=NULL; }
        if (H[i][1]) { DeleteObject(H[i][1]); H[i][1]=NULL; }
    }

}
```

Here is the PaletteResource structure definition. In practice it's very easy to use. You can check it in the code above.

```
struct PaletteResource
{
    typedef unsigned char     PalRsrcColor[3];              // r, g, b

    PaletteResource()
    {
        mVersion = 1;
        mColorsUsed = 256;
        mUnused1 = 0;
        mUnused2 = 0;
        ::memset(&mPalette,0,sizeof(mPalette));
    }

    PaletteResource(PalWrap *inPal)
    {
        mVersion = 1;
        mColorsUsed = 256;
        mUnused1 = 0;
        mUnused2 = 0;

        const PalWrap::Color *pal = inPal->GetPaletteEntries();
        for (int i=0;i<256;++i) {
            mPalette[i][0] = pal[i].peRed;
            mPalette[i][1] = pal[i].peGreen;
            mPalette[i][2] = pal[i].peBlue;
        }
    }

    UInt32         mVersion;
    UInt32         mColorsUsed;
    UInt32         mUnused1;
    UInt32         mUnused2;
    PalRsrcColor   mPalette[256];
};
```

Jamie's portion begins here:

Here's an excerpt from the old TDRs on draw groups and sprites:

> **3.2.7 SPR# : Sprite List**
> A single sprite is identified by its file, its resource id, and its index. An SPR# resource contains version info and an offset table. An individual sprite is found using this table to add to the address of the beginning of a resource. Each sprite has header info and compression data. The header has a bit mask denoting which types of tokens it contains, and a bounding box. The compression data

is tokenized to do run length, clear space, patterns, and shades. (see also TDSBuildObject in 17.4.1 and Blitters in 10.2).

3.2.8 DGRP : Draw Groups

A draw group contains enough information to render a single frame of an object in any of 4 rotations and 3 zooms. This is done with a list of draw lists. Each draw list has a bit mask denoting which rotation the frame is valid for, and an integer value denoting the zoom. To select a draw list, a rotation and zoom are specified. When drawing an object, the object's rotation is composed with the view rotation, and then, with the current viewing zoom, the appropriate draw list is selected. Within each draw list is a list of tokenized "draw items". The different types of tokens that can occur are:

1. Sprite: id and index with x, y pixel offset and flipping flags.
2. Top Object: x, y offset of where to draw the target object. Not used anymore because it does not encode a depth offset for the sprite. This has been replaced by slots, which do store depth information.
3. Body Token: draw this object's old-style body (not used anymore)
4. Slot Object Token : (x, y, alt) 3D fixed-point coordinate (see Slot Coordinates in Chapter 2 ???), in object's frame of reference. A slot index denoting which slot object to draw there.

Slot Object Tokens can currently be left out of the draw group entirely for slots that do not change positions from frame to frame. If no slot tokens are in the draw group , all the object slots will be drawn using the coordinates in the slot itself after the draw group has been drawn. (See also Slots, 3.5.1)

Note that sprite tokes are now the only ones used in draw groups. Also, the SPR# format has been extended to include SPR2.

Here are some excerpts from DrawGroup.h and DrawGroup.cpp which specify the DGRP resource type. It is saved and loaded using the DoStream routines (can't cast the resource to the class directly!). DoContainerStream is a function which streams the size of the container, then calls DoStream on all contained objects.

```
class DrawGroupItem {
    Int     fSpriteID;
    Int     fSpriteNum;
    Int     fPixelXOff;
    Int     fPixelYOff;
    float   fZOff; // In alt units
    float   fXOff; // In fractional tiles.
    float   fYOff; // In fractional tiles.
    Int     fFlags;
}

class DrawGroupItemList : private STD::vector<DrawGroupItem> {
    Int     fDirFlags;
    Int     fZoom;
}

class DrawGroup : private STD::vector<DrawGroupItemList> {
}

void DrawGroup::DoStream(ReconBuffer *r)
{
    SInt16 version = 20004;
    r->Recon16(&version);

    if  (version>=20003) {
        // Cast to super class to unprotect the private inheritance.
        STD::vector<DrawGroupItemList>& items = *this;
        DoContainerStream(items, r, version);
    }
    else {
        ReadOldVersion(r, version);
    }
}

void DrawGroupItemList::DoStream(ReconBuffer *r, SInt32 version)
{
    XASSERT(version >= 20003, "low version");

    r->ReconInt(&fDirFlags);
    r->ReconInt(&fZoom);

    // Cast to super class to unprotect the private inheritance.
    STD::vector<DrawGroupItem> &items = *this;
    DoContainerStream(items, r, version);
}

void DrawGroupItem::DoStream(ReconBuffer *r, SInt32 version)
{
    XASSERT(version >= 20003, "low version");

    r->ReconInt(&fSpriteID);
    r->ReconInt(&fSpriteNum);
    r->ReconInt(&fPixelXOff);
    r->ReconInt(&fPixelYOff);
    r->ReconFloat(&fZOff);
    r->ReconInt(&fFlags);
    if (version>=20004) {
        r->ReconFloat(&fXOff);
        r->ReconFloat(&fYOff);
    }
}
```

Here is an excerpt from xrender.h of the SPR# format. The pointer to the resource may be cast directly, after swizzling. SPR# are always stored big-endan.

```
    struct NewSPRList
    {
        SInt32  version;
        UInt32  numGraphics;
        SInt32  mPaletteID;
        UInt32  offsets[1];
    };
```

Versions increase and 1000 marks the beginning of the SPR2 formats. Currently SPR# is only used for certain special sprite types, such as walls and floors. All object sprites use SPR2.

Here is the struct for SPR2:

```
struct Sprite2List
{
    UInt32  mVersion;
    Int32   mCount;
    Int32   mPaletteID;
    Int32   mOffsetTable[1];   // contains mCount entries; each one holds of
                                      // offset (in bytes) from pointer to Sprite2List
                                      // and pointer to Sprite2 embedded therein.
    static int HeaderSize()
    {
        return int( &(((Sprite2List *) NULL)->mOffsetTable[0]) );
    }
};
```

The Sprite2List offset table points to the individual sprites. Here is the struct for that:

```
// internal format for a sprite.  Encoded data starts at mData[0]
// header size = (&aSpr2.mData[0] - &aSpr2.mWidth) = 16 bytes
struct Sprite2
{
    UInt16  mWidth;
    UInt16  mHeight;
    UInt32  mFlags;            // 0 for now, but room to wiggle
    Int16   mPaletteID;        // may be negative...
    Int16   mClearColor;       // between 0 and 255...probably only palette index never
encoded
    Point16 mOrigin;           // position of (top,left) in some coord system
    UInt8   mData[1];          // begin of encoded stream; see below

    static int HeaderSize()
    {
        return int( &(((Sprite2 *) NULL)->mData[0]) );
    }
};
```

Here is the enum for the flags:

```
// masks for Sprite2::mFlags
enum Spr2Flags {
    kColorData     = 0x01,
    kZData         = 0x02,
    kAlphaData     = 0x04
};
```

Here are the tokens that go into it. Notice the token space is only 3 bits:

```
// The mData of a Sprite2 struct points to an encoded stream,
// comprised of the following tokens.
// (don't change these; they are encoded in SPR2 resources.
// it's ok to add more (up to 0x07)
enum Spr2Tokens
{
    kNL        = 0x00 << 13,          // "new line", start of sprite, eol
    kRunZP     = 0x01 << 13,          // no alpha blending
    kRunZPA    = 0x02 << 13,          // yes alpha blending
    kClear     = 0x03 << 13,          // completely transparent pixels
    kSL        = 0x04 << 13,          // "skip lines"; start of sprite or eol
    kEOS       = 0x05 << 13,          // end of sprite
    kRunP      = 0x06 << 13           // no z, no alpha, just color
};
```

Here is an excerpt from the 565 decoder:

```
        Spr2Tokens token = GETTOKEN(streamPtr);
        int data  = GETDATA(streamPtr);
        streamPtr += 2;

        switch (token) {

        case kNL:  // new line
            curDestPx = nextPxLine;
            curDestZ = nextZLine;
            nextPxLine += destPx.mStride;     // creating local strides might
            nextZLine += destZ.mStride;               // have better cache coherency
            break;

        case kSL:  // skip "data" lines
            // might be faster to use a lookup table
            // to do the multiplication stride*data,
            //      nextPxLine += lookup_px_stride[ data ];
            //      nextZLine += lookup_z_stride[ data ];
            // over even to do the multiply!
            XASSERT( data > 1, "kSL(1) and kSL(0) not allowed!" );          // kSL(1) not
allowed!
            for ( i = 0 ; i < data ; ++i ) {
                curDestPx = nextPxLine;
                curDestZ = nextZLine;
                nextPxLine += destPx.mStride;
                nextZLine += destZ.mStride;
            }
            break;
        case kRunZP:
            // [z][px] [z][px] ...
            {
                for ( i = 0 ; i < data ; ++i ) {
                    z = z_origin + z_lookup[ streamPtr[0] ];         // read z
                    if ( z <= int(*curDestZ) ) {      // do z test
                        *curDestZ = z;                        // write z immediately
                        *curDestPx = px_lookup[ streamPtr[1] ];      // lookup pixel
                    }
                    streamPtr += 2;    // incr streamPtr to next [z]
                    ++curDestPx;       // next dest px
                    ++curDestZ;        // next dest z
                }
                // streamPtr is guaranteed to be 2-byte aligned
                XASSERT( ( int(streamPtr) & 1) == 0, "streamPtr went odd" );
            }
            break;

        case kRunZPA:
            // [z][px][a] [z][px][a] ...
            {
                for ( i = 0 ; i < data ; ++i ) {
                    z = z_origin + z_lookup[ streamPtr[0] ]; // decode z
                    if ( z <= int(*curDestZ) ) {                            // if z-test
 passes,
                        ALPHA_ZWRITE(*curDestZ = z);  // free up register holding z, I hope
                                                      // blend like we've never blended
 before
                        *curDestPx = Blend565( *curDestPx,
                                                                px_lookup[ streamPtr[1] ],
                                                                streamPtr[2] );
                    }
                    streamPtr += 3;           // incr stream to next [z]
                    ++curDestPx;              // next dest px
                    ++curDestZ;               // next dest z
                }
                // round up to even addr...note we avoid a branch to do it
                streamPtr += (int(streamPtr) & 1);
            }
            break;

 case kClear:   // "data" transparent pixels
            curDestPx += data;
            curDestZ += data;
            break;

 case kEOS:     // end-of-sprite
            // "data" is undefined
            goto done;
 } // switch (token)
```