

```

//
// Copyright (C) 2001, Intel Corporation
// All rights reserved.
// Permission is hereby granted to merge this program code with other program
// material to create a derivative work. This derivative work may be distributed
// in compiled object form only. Any other publication of this program, in any form,
// without the explicit permission of the copyright holders is prohibited.
//
// Send questions and comments to John.Hyde@intel.com
//
// Derived, and used with permission, from:
//-----
// File: EZRGB24.cpp
//
// Desc: DirectShow sample code - special effects image filter.
//
// Copyright (c) 1992 - 2000, Microsoft Corporation. All rights reserved.
//-----

//
// Files
//
// ezprop.cpp          A property page to control the video effects
// ezprop.h            Class definition for the property page object
// ezprop.rc           Dialog box template for the property page
// ezrgb24.cpp         Main filter code that does the special effects
// ezrgb24.def         What APIs we import and export from this DLL
// ezrgb24.h           Class definition for the special effects filter
// ezuids.h            Header file containing the filter CLSIDs
// iez.h               Defines the special effects custom interface
// resource.h          Microsoft Visual C++ generated resource file
//
//
// Base classes used
//
// CTransformFilter    A transform filter with one input and output pin
// CPersistStream      Handles the grunge of supporting IPersistStream
//
//
#include <windows.h>
#include <streams.h>
#include <initguid.h>
#if (1100 > _MSC_VER)
#include <olectlid.h>
#else
#include <olectl.h>
#endif
#include "EZuids.h"
#include "IEZ.h"
#include "EZprop.h"
#include "EZrgb24.h"
#include "resource.h"

// Setup information
const AMOVIESETUP_MEDIATYPE sudPinTypes = {
    &MEDIATYPE_Video,          // Major type
    &MEDIASUBTYPE_NULL        // Minor type
};

const AMOVIESETUP_PIN sudpPins[] = {
    { L"Input",                // Pins string name
      FALSE,                   // Is it rendered
      FALSE,                   // Is it an output
      FALSE,                   // Are we allowed none
      FALSE,                   // And allowed many
      &CLSID_NULL,            // Connects to filter
      NULL,                   // Connects to pin
      1,                       // Number of types
      &sudPinTypes            // Pin information
    },
    { L"Output",              // Pins string name
      FALSE,                   // Is it rendered
      TRUE,                    // Is it an output
      FALSE,                   // Are we allowed none
      FALSE,                   // And allowed many
      &CLSID_NULL,            // Connects to filter
      NULL,                   // Connects to pin
      1,                       // Number of types
    }
};

```

```

        &sudPinTypes          // Pin information
    }
};

const AMOVIESETUP_FILTER sudEZrgb24 = {
    &CLSID_EZrgb24,          // Filter CLSID
    L"UDBE Image Effects",  // String name
    MERIT_DO_NOT_USE,       // Filter merit
    2,                       // Number of pins
    sudpPins                 // Pin information
};

// List of class IDs and creator functions for the class factory. This
// provides the link between the OLE entry point in the DLL and an object
// being created. The class factory will call the static CreateInstance

CFactoryTemplate g_Templates[] = {
    { L"UDBE Image Effects"
      , &CLSID_EZrgb24
      , CEZrgb24::CreateInstance
      , NULL
      , &sudEZrgb24 }
    ,
    { L"UDBE Special Effects"
      , &CLSID_EZrgb24PropertyPage
      , CEZrgb24Properties::CreateInstance }
    };
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);

// Handle sample registry and unregistry
STDAPI DllRegisterServer() {
    return AMovieDllRegisterServer2( TRUE );
}

STDAPI DllUnregisterServer() {
    return AMovieDllRegisterServer2( FALSE );
}

// Constructor
CEZrgb24::CEZrgb24(TCHAR *tszName, LPUNKNOWN punk, HRESULT *phr) :
    CTransformFilter(tszName, punk, CLSID_EZrgb24),
    m_effect(IDC_MOTION_DETECT),
    m_lBufferRequest(1),
    m_pLastImage(NULL),
    m_nSkippedThreshold(30),
    m_nDifferenceThreshold(500000), // this should really be based on image size
    m_nSkippedCount(0),
    CPersistStream(punk, phr) {
    char sz[60];
    GetProfileStringA("Quartz", "EffectStart", "2.0", sz, 60);
    m_effectStartTime = COARefTime(atof(sz));
    GetProfileStringA("Quartz", "EffectLength", "5.0", sz, 60);
    m_effectTime = COARefTime(atof(sz));
}

// CreateInstance
//
// Provide the way for COM to create a EZrgb24 object
CUnknown *CEZrgb24::CreateInstance(LPUNKNOWN punk, HRESULT *phr) {
    CEZrgb24 *pNewObject = new CEZrgb24(NAME("Image Effects"), punk, phr);
    if (pNewObject == NULL) *phr = E_OUTOFMEMORY;
    return pNewObject;
}

// NonDelegatingQueryInterface
//
// Reveals IIPEffect and ISpecifyPropertyPages
STDMETHODIMP CEZrgb24::NonDelegatingQueryInterface(REFIID riid, void **ppv) {
    CheckPointer(ppv, E_POINTER);
    if (riid == IID_IIPEffect) return GetInterface((IIPEffect *) this, ppv);
    else if (riid == IID_ISpecifyPropertyPages) return GetInterface((ISpecifyPropertyPages *) this, ppv);
    else return CTransformFilter::NonDelegatingQueryInterface(riid, ppv);
}

// Transform
//
// Copy the input sample into the output sample - then transform the output

```

```

// sample 'in place'. If we have all keyframes, then we shouldn't do a copy
// If we have cinepak or indeo and are decompressing frame N it needs frame
// decompressed frame N-1 available to calculate it, unless we are at a
// keyframe. So with keyframed codecs, you can't get away with applying the
// transform to change the frames in place, because you'll mess up the next
// frames decompression. The runtime MPEG decoder does not have keyframes in
// the same way so it can be done in place. We know if a sample is key frame
// as we transform because the sync point property will be set on the sample
//
HRESULT CEZrgb24::Transform(IMediaSample *pIn, IMediaSample *pOut) {
// Copy the properties across
    HRESULT hr = Copy(pIn, pOut);
    if (FAILED(hr)) return hr;

// Check to see if it is time to do the sample
    CRefTime tStart, tStop;
    pIn->GetTime((REFERENCE_TIME *) &tStart, (REFERENCE_TIME *) &tStop);

//     if (tStart >= m_effectStartTime) {
//         if (tStop <= (m_effectStartTime + m_effectTime)) {
//             return Transform(pOut);
//         }
//     }

    return NOERROR;
}

// Copy
//
// Make destination an identical copy of source
HRESULT CEZrgb24::Copy(IMediaSample *pSource, IMediaSample *pDest) const {
// Copy the sample data
    BYTE *pSourceBuffer, *pDestBuffer;
    long lSourceSize = pSource->GetActualDataLength();
    long lDestSize = pDest->GetSize();

    ASSERT(lDestSize >= lSourceSize);

    pSource->GetPointer(&pSourceBuffer);
    pDest->GetPointer(&pDestBuffer);

    CopyMemory( (PVOID) pDestBuffer, (PVOID) pSourceBuffer, lSourceSize);

// Copy the sample times
    REFERENCE_TIME TimeStart, TimeEnd;
    if (NOERROR == pSource->GetTime(&TimeStart, &TimeEnd)) pDest->SetTime(&TimeStart, &TimeEnd);

    LONGLONG MediaStart, MediaEnd;
    if (pSource->GetMediaTime(&MediaStart, &MediaEnd) == NOERROR) pDest->SetMediaTime(&MediaStart, &MediaEnd);

// Copy the Sync point property
    HRESULT hr = pSource->IsSyncPoint();
    if (hr == S_OK) pDest->SetSyncPoint(TRUE);
    else if (hr == S_FALSE) pDest->SetSyncPoint(FALSE);
    else return E_UNEXPECTED;

// Copy the media type
    AM_MEDIA_TYPE *pMediaType;
    pSource->GetMediaType(&pMediaType);
    pDest->SetMediaType(pMediaType);
    DeleteMediaType(pMediaType);

// Copy the preroll property
    hr = pSource->IsPreroll();
    if (hr == S_OK) pDest->SetPreroll(TRUE);
    else if (hr == S_FALSE) pDest->SetPreroll(FALSE);
    else return E_UNEXPECTED;

// Copy the discontinuity property
    hr = pSource->IsDiscontinuity();
    if (hr == S_OK) pDest->SetDiscontinuity(TRUE);
    else if (hr == S_FALSE) pDest->SetDiscontinuity(FALSE);
    else return E_UNEXPECTED;

// Copy the actual data length
    long lDataLength = pSource->GetActualDataLength();

```

```

pDest->SetActualDataLength(lDataLength);

return NOERROR;
}

// Transform (in place)
//
// 'In place' apply the image effect to this sample
HRESULT CEZrgb24::Transform(IMediaSample *pMediaSample) {
    AM_MEDIA_TYPE* pType = &m_pInput->CurrentMediaType();
    VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *) pType->pbFormat;
    BYTE *pData;           // Pointer to the actual image buffer
    long lDataLen;         // Holds length of any given sample
    unsigned int grey;     // Used when applying greying effects
    int iPixel;           // Used to loop through the image pixels
    RGBTRIPLE *prgb;      // Holds a pointer to the current pixel
    HRESULT hr = NOERROR;

    BYTE *pStartOfScreen, bitmap;
    int dx, dy, Index;
    SYSTEMTIME CurrentTime;
    // The time and date display is sized for a 320 x 240 video window
    // Change CharacterOffset and CharacterPosition to place the display line anywhere in the video window
    int CharacterOffset = 20;
    const int CharacterWidth[] = { 8, 8, 3, 8, 8, 3, 8, 8, 8, 8, 3, 8, 8, 3, 8, 8 };
    const int CharacterPosition[] = { 20, 31, 37, 48, 59, 65, 76, 87, 105, 116, 122, 133, 144, 150, 161, 172 };
    int Character[16];

    pMediaSample->GetPointer(&pData);
    lDataLen = pMediaSample->GetSize();

    // Get the image properties from the BITMAPINFOHEADER
    int iPixelSize = pvi->bmiHeader.biBitCount / 8;
    int cxImage = pvi->bmiHeader.biWidth;
    int cyImage = pvi->bmiHeader.biHeight;
    int cbImage = cyImage * cxImage * iPixelSize;
    int numPixels = cxImage * cyImage;
    prgb = (RGBTRIPLE*) pData;

    switch (m_effect) {
        case IDC_NONE: break;
        case IDC_RED: for (iPixel=0; iPixel < numPixels; iPixel++, prgb++) prgb->rgbtGreen = prgb->rgbtBlue = 0;
break;
        case IDC_GREEN: for (iPixel=0; iPixel < numPixels; iPixel++, prgb++) prgb->rgbtRed = prgb->rgbtBlue = 0;
break;
        case IDC_BLUE: for (iPixel=0; iPixel < numPixels; iPixel++, prgb++) prgb->rgbtRed = prgb->rgbtGreen = 0;
break;
        case IDC_DARKEN:for (iPixel=0; iPixel < numPixels; iPixel++, prgb++) {
// Bitwise shift to the right results in the image getting much darker
prgb->rgbtRed = prgb->rgbtRed >> 1;
prgb->rgbtGreen = prgb->rgbtGreen >> 1;
prgb->rgbtBlue = prgb->rgbtBlue >> 1;
}
break;
        case IDC_XOR: for (iPixel=0; iPixel < numPixels; iPixel++, prgb++) {
// Toggle each bit - this gives a sort of X-ray effect
prgb->rgbtRed = prgb->rgbtRed ^ 0xff;
prgb->rgbtGreen = prgb->rgbtGreen ^ 0xff;
prgb->rgbtBlue = prgb->rgbtBlue ^ 0xff;
}
break;
        case IDC_POSTERIZE: for (iPixel=0; iPixel < numPixels; iPixel++, prgb++) {
// Zero out the five LSB per each component
prgb->rgbtRed = prgb->rgbtRed & 0xe0;
prgb->rgbtGreen = prgb->rgbtGreen & 0xe0;
prgb->rgbtBlue = prgb->rgbtBlue & 0xe0;
}
break;
        case IDC_GREY: for (iPixel=0; iPixel < numPixels ; iPixel++, prgb++) {
// An excellent greyscale calculation is: grey = (30 * red + 59 * green + 11 * blue) / 100
// This is a bit too slow so a faster calculation is: grey = (red + green) / 2
grey = (prgb->rgbtRed + prgb->rgbtGreen) >> 1;
prgb->rgbtRed = prgb->rgbtGreen = prgb->rgbtBlue = (BYTE) grey;
}
break;
        case IDC_TIMESTAMP:
// Set top of display line
pStartOfScreen = pData;

```

```

        GetLocalTime(&CurrentTime);
// Format for my display
Character[0] = CurrentTime.wHour / 10; Character[1] = CurrentTime.wHour % 10;
Character[2] = Character[5] = 10; Character[10] = Character[13] = 11; // delimiters
Character[3] = CurrentTime.wMinute / 10; Character[4] = CurrentTime.wMinute % 10;
Character[6] = CurrentTime.wSecond / 10; Character[7] = CurrentTime.wSecond % 10;
Character[8] = CurrentTime.wMonth / 10; Character[9] = CurrentTime.wMonth % 10;
Character[11] = CurrentTime.wDay / 10; Character[12] = CurrentTime.wDay % 10;
Character[14] = (CurrentTime.wYear - 2000) / 10; Character[15] = (CurrentTime.wYear - 2000) % 10;
// Now update the video image
for (Index = 0; Index < 16; Index++) {
    for (dy = 0; dy < 14; dy++) {
        bitmap = Lookup(Character[Index], dy);
        for (dx = 0; dx < CharacterWidth[Index]; dx++) {
            pData = pStartOfScreen + (((CharacterOffset - dy)*320) + CharacterPosition[Index] - dx) *
3);
// Write a white background for the time and date. Removed, it looked better without
// prgb->rgbtRed = 0; prgb->rgbtGreen = prgb->rgbtBlue = 0;
prgb = (RGBTRIPLE*) pData;
// Overwrite the screen color with red characters
if ((bitmap >> dx) & 1) {prgb->rgbtRed = 255; prgb->rgbtGreen = prgb->rgbtBlue = 0;}
        }
    }
    break;
case IDC_MOTION_DETECT:
    long nDifference;
// first frame only
if (m_pLastImage == NULL) {
    m_pLastImage = new BYTE[cbImage];
    memcpy(m_pLastImage, pData, cbImage);
}
// Output a frame if even if there was no movement for x seconds
if (m_nSkippedCount > m_nSkippedThreshold) {
    memcpy(m_pLastImage, pData, cbImage);
    m_nSkippedCount = 0;
    hr = NOERROR;
}
else {
    nDifference = FrameDifference(pData, m_pLastImage, cbImage);
    memcpy(m_pLastImage, pData, cbImage);
    if (nDifference > m_nDifferenceThreshold) {
        m_nSkippedCount = 0;
        hr = NOERROR;
    }
    else {
        m_nSkippedCount++;
        hr = S_FALSE;
    }
}
    break;
} // Switch()
return hr;

} // Transform (in place)

int CEZrgb24::Lookup(int Char, int Line) {
// Rather than use a large font table this routine uses a 7-segment display model
//
// Define segments on/off    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, :, /
const BYTE SegA[] = { 255, 0, 255, 255, 0, 255, 255, 255, 255, 255, 0, 0 };
const BYTE SegB[] = { 255, 255, 255, 255, 255, 0, 0, 255, 255, 255, 0, 0 };
const BYTE SegC[] = { 255, 255, 0, 255, 255, 255, 255, 255, 255, 255, 0, 255 };
const BYTE SegD[] = { 255, 0, 255, 255, 0, 255, 255, 0, 255, 255, 0, 0 };
const BYTE SegE[] = { 255, 0, 255, 0, 0, 0, 255, 0, 255, 0, 0, 0 };
const BYTE SegF[] = { 255, 0, 0, 0, 255, 255, 255, 0, 255, 255, 0, 0 };
const BYTE SegG[] = { 0, 0, 255, 255, 255, 255, 255, 0, 255, 255, 0, 0 };
const BYTE Dots[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0 };

switch(Line) {
case 0: return (0x3C & SegA[Char]);
case 1: return (0x7E & SegA[Char]) | (0x40 & SegF[Char]) | (0x02 & SegB[Char]);
case 2: return (0x3C & SegA[Char]) | (0xE0 & SegF[Char]) | (0x07 & SegB[Char]);
case 3: return (0x02 & Dots[Char]) | (0xE0 & SegF[Char]) | (0x07 & SegB[Char]);
case 4: return (0x07 & Dots[Char]) | (0xE0 & SegF[Char]) | (0x07 & SegB[Char]);
case 5: return (0x07 & Dots[Char]) | (0xE0 & SegF[Char]) | (0x07 & SegB[Char]) | (0x3C & SegG[Char]);
case 6: return (0x02 & Dots[Char]) | (0x40 & SegF[Char]) | (0x02 & SegB[Char]) | (0x7E & SegG[Char]) |
(0x40 & SegE[Char]) | (0x02 & SegC[Char]);

```

```

        case 7: return (0x02 & Dots[Char]) | (0xE0 & SegE[Char]) | (0x07 & SegC[Char]) | (0x3C & SegG[Char]);
        case 8: return (0x07 & Dots[Char]) | (0xE0 & SegE[Char]) | (0x07 & SegC[Char]);
        case 9: return (0x07 & Dots[Char]) | (0xE0 & SegE[Char]) | (0x07 & SegC[Char]);
        case 10: return (0x02 & Dots[Char]) | (0x3C & SegD[Char]) | (0xE0 & SegE[Char]) | (0x07 & SegC[Char]);
        case 11: return (0x7E & SegD[Char]) | (0x40 & SegE[Char]) | (0x02 & SegC[Char]);
        case 12: return (0x3C & SegD[Char]);
        default: return 0;
    }
}

long CEZrgb24::FrameDifference(BYTE *pCurrent, BYTE *pLast, int nBytes) {
    long diff = 0;
    long d;

    // compute absolute pel difference
    while (nBytes-- > 0) {
        d = (long)(*pCurrent++ - *pLast++);
        if (d < 0) d = -d;
        diff += d;
    }
    return diff;
}

// Check the input type is OK - return an error otherwise
HRESULT CEZrgb24::CheckInputType(const CMediaType *mtIn) {
    // check this is a VIDEOINFOHEADER type
    if (*mtIn->FormatType() != FORMAT_VideoInfo) return E_INVALIDARG;

    // Can we transform this type
    if (!CanPerformEZrgb24(mtIn)) return E_FAIL;
    return NOERROR;
}

// Check a transform can be done between these formats
HRESULT CEZrgb24::CheckTransform(const CMediaType *mtIn, const CMediaType *mtOut) {
    if (CanPerformEZrgb24(mtIn)) {
        if (*mtIn != *mtOut) return E_FAIL;
    }
    return NOERROR;
}

// Tell the output pin's allocator what size buffers we
// require. Can only do this when the input is connected
HRESULT CEZrgb24::DecideBufferSize(IMemAllocator *pAlloc, ALLOCATOR_PROPERTIES *pProperties) {
    // Is the input pin connected
    if (m_pInput->IsConnected() == FALSE) return E_UNEXPECTED;

    ASSERT(pAlloc);
    ASSERT(pProperties);
    HRESULT hr = NOERROR;

    pProperties->cBuffers = 1;
    pProperties->cbBuffer = m_pInput->CurrentMediaType().GetSampleSize();
    ASSERT(pProperties->cbBuffer);

    // Ask the allocator to reserve us some sample memory, NOTE the function
    // can succeed (that is return NOERROR) but still not have allocated the
    // memory that we requested, so we must check we got whatever we wanted
    ALLOCATOR_PROPERTIES Actual;
    hr = pAlloc->SetProperties(pProperties, &Actual);
    if (FAILED(hr)) return hr;

    ASSERT( Actual.cBuffers == 1 );

    if (pProperties->cBuffers > Actual.cBuffers || pProperties->cbBuffer > Actual.cbBuffer) {
        return E_FAIL;
    }
    return NOERROR;
}

// I support one type, namely the type of the input pin
// This type is only available if my input is connected
HRESULT CEZrgb24::GetMediaType(int iPosition, CMediaType *pMediaType) {
    // Is the input pin connected
    if (m_pInput->IsConnected() == FALSE) return E_UNEXPECTED;

    // This should never happen
    if (iPosition < 0) return E_INVALIDARG;
}

```

```

// Do we have more items to offer
if (iPosition > 0) return VFW_S_NO_MORE_ITEMS;

    *pMediaType = m_pInput->CurrentMediaType();
    return NOERROR;
}

// Check if this is a RGB24 true colour format
BOOL CEZrgb24::CanPerformEZrgb24(const CMediaType *pMediaType) const {
    if (IsEqualGUID(*pMediaType->Type(), MEDIATYPE_Video)) {
        if (IsEqualGUID(*pMediaType->Subtype(), MEDIASUBTYPE_RGB24)) {
            VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *) pMediaType->Format();
            return (pvi->bmiHeader.biBitCount == 24);
        }
    }
    return FALSE;
}

#define WRITEOUT(var) hr = pStream->Write(&var, sizeof(var), NULL); \
    if (FAILED(hr)) return hr;

#define READIN(var) hr = pStream->Read(&var, sizeof(var), NULL); \
    if (FAILED(hr)) return hr;

// This is the only method of IPersist
STDMETHODIMP CEZrgb24::GetClassID(CLSID *pClsid) {
    return CBaseFilter::GetClassID(pClsid);
}

// Overriden to write our state into a stream
HRESULT CEZrgb24::ScribbleToStream(IStream *pStream) {
    HRESULT hr;
    WRITEOUT(m_effect);
    WRITEOUT(m_effectStartTime);
    WRITEOUT(m_effectTime);
    return NOERROR;
}

// Likewise overriden to restore our state from a stream
HRESULT CEZrgb24::ReadFromStream(IStream *pStream) {
    HRESULT hr;
    READIN(m_effect);
    READIN(m_effectStartTime);
    READIN(m_effectTime);
    return NOERROR;
}

// Returns the clsid's of the property pages we support
STDMETHODIMP CEZrgb24::GetPages(CAUUID *pPages) {
    pPages->cElems = 1;
    pPages->pElems = (GUID *) CoTaskMemAlloc(sizeof(GUID));
    if (pPages->pElems == NULL) return E_OUTOFMEMORY;
    *(pPages->pElems) = CLSID_EZrgb24PropertyPage;
    return NOERROR;
}

// Return the current effect selected
STDMETHODIMP CEZrgb24::get_IPEffect(int *IPEffect, REFTIME *start, REFTIME *length) {
    CAutoLock cAutolock(&m_EZrgb24Lock);
    CheckPointer(IPEffect, E_POINTER);
    CheckPointer(start, E_POINTER);
    CheckPointer(length, E_POINTER);

    *IPEffect = m_effect;
    *start = COARefTime(m_effectStartTime);
    *length = COARefTime(m_effectTime);

    return NOERROR;
}

// Set the required video effect
STDMETHODIMP CEZrgb24::put_IPEffect(int IPEffect, REFTIME start, REFTIME length) {

```

```
CAutoLock cAutolock(&m_EZrgb24Lock);  
  
m_effect = IPEffect;  
m_effectStartTime = COARefTime(start);  
m_effectTime = COARefTime(length);  
  
SetDirty(TRUE);  
return NOERROR;  
}
```