Augmented Reality voor Windows Mobile

Informatie koppelen aan markers in de wereld om je heen

André van der Plas

In de militaire luchtvaart vliegen piloten sinds jaar en dag met head-up displays (HUD's). Op deze displays wordt essentiële informatie getoond waardoor een piloot naar buiten kan kijken en tegelijkertijd informatie kan bestuderen in de vorm van bijvoorbeeld geprojecteerde radargegevens. Dergelijke systemen vallen binnen de categorie Augmented Reality, of kortweg AR.

AR is daarmee oud, maar ondertussen ook weer nieuw en hot, want AR is tegenwoordig niet meer beperkt tot jongensdromen. Wie nu een smartphone heeft, kan ook ervaren hoe het is om de wereld om ons heen vanuit een informatieve context te aanschouwen. In dit artikel worden de basisbenodigdheden voor een AR applicatie beschouwd.

Twee varianten

Maar eerst moet er een onderscheid worden gemaakt tussen twee varianten op het gebied van AR. Zo is er een variant waarbij informatie wordt gekoppeld aan de geolocatie van de mobiele gebruiker, de zogenaamde positiebepaalde variant. Layar is daar het meest bekende voorbeeld van.
De tweede variant maakt geen gebruik van positie, maar van herkenning. De beelden die door de camera op je mobiel worden geregistreerd worden door een tracking algoritme bestudeerd op herkenningspunten, markers genaamd. Is er sprake van herkenning, dan kan er op het beeldscherm iets worden getoond dat gerelateerd is aan hetgeen dat herkend wordt. Hierop focussen we ons in dit artikel.

Drie stappen

Bij AR met behulp van beeldherkenning draait het om een drietal te implementeren stappen:

- Stap 1: Capturing - het capturen van camera frames;
- Stap 2: Rendering - het kunnen tonen en bewerken van die frames in je eigen applicatie;
- Stap 3: Tracking - het analyseren van de frames op herkenningspunten.

FIGUUR1: EEN GECAPTURED FRAME DAT GERENDERED IS NA HERKENNING VAN EEN LOGO.

Zijn deze drie stappen geïmplementeerd, dan zijn we in staat om camerabeelden in onze eigen applicatie te tonen en te voorzien van informatie (in de vorm van tekst, filmpjes, of 3D objecten) op specifieke plekken in het beeld.

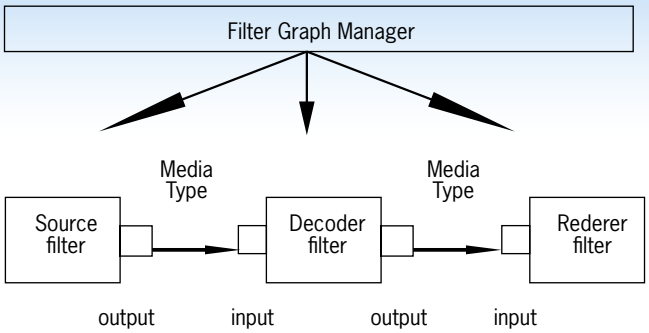
Stap 1: Capturing

Ofschoon we reeds met onze telefooncamera opnames kunnen maken en deze in realtime op het beeldscherm bekijken, is er nog geen sprake van capturing. De camerabeelden worden namelijk vanuit de camera direct doorgestuurd naar het beeldscherm en kunnen niet door onze applicatie worden ondervangen en gemanipuleerd. Onze applicatie moet dan ook op een directe wijze met de camera communiceren. Dit doen we binnen Windows Mobile met behulp van DirectShow.
DirectShow is een mulitmedia framework en API geschreven in C++, waarmee op verschillende mediabronnen (audiofiles/videofiles/camera's /etcetera) kan worden geopereerd. Voor onze AR applicatie is het de manier om binnen alle versies van Windows Mobile op een constistente manier camera-input te verzamelen. Hierbij maakt DirectShow gebruik van filtergraphs om frames van een cameradevice te kunnen capturen.

Filters

Een filtergraph is een verzameling aan elkaar gekoppelde filters waar de cameraframes zich doorheen verplaatsen. Elk filter heeft hierbij zijn eigen functie, en je kunt zelf bepalen welke filters je wilt gebruiken. Er zijn drie soorten filters te onderscheiden (zie figuur 2) :

1. Source filters: Dit zijn de filters die gekoppeld zijn aan de datasource (in ons geval de camera) om de input als datastreams naar de andere filters door te sturen.
2. Transform filters: Dit zijn de filters die een transformatie (bijvoorbeeld een frame 90° draaien) op de data uitvoeren.
3. Renderer fitlers: Deze filters renderen de data door bijvoorbeeld de data op het beeldscherm te tonen, of naar een file weg te schrijven.



FIGUUR2: EEN FILTER GRAPH HEEFT MAXIMAAL DRIE VERSCHILLENDE SOORTEN FILTERS.

Voor onze applicatie maken we gebruik van een source filter die aan de camera is gekoppeld, en de data doorstuurt naar een transformfilter. In de transformfilter krijgen we de frame binnen en zijn we in staat om deze te bewerken. Dit doen we echter niet, want we maken gebruik van een callback method om ieder frame door te sturen naar het gedeelte van onze applicatie waar het frame wordt getoond op het scherm. Hierdoor is de laatste filter niet nodig en zullen we dan ook een null renderer gebruiken om het renderen via de filtergraph te voorkomen.
Om de filters aan te maken en te beheren maken we gebruik van een eigen class die we CgraphManager zullen noemen. De stappen die met deze graphmanager gemaakt moeten worden zijn als volgt:

```
graphmanager = new CGraphManager();
graphmanager->Init();
graphmanager->RegisterCallback(OnFrameProcessed);
graphmanager->BuildCaptureGraph();
graphmanager->RunCaptureGraph();
```

Vanwege de continue afname van cameraframes draait de graphmanager binnen een aparte thread die tijdens initialisatie wordt aangemaakt.

BuildCaptureGraph
De BuildCaptureGraph is de functie waarbinnen de filters worden gespecificeerd en aan elkaar worden gekoppeld.

```
CGraphManager::CreateCaptureGraphInternal()
{
    CComVariant                 varCamName;
    CPropertyBag               PropBag;
    WCHAR                       wzDeviceName[ MAX_PATH + 1 ];
    CComPtr<IGraphBuilder>          pFilterGraph;
    CComPtr<IPersistPropertyBag>    pPropertyBag;
    CComPtr<IBaseFilter>            pFrameGrabber;
    CComPtr<IBaseFilter>            pNullRenderer;
    CComPtr<IFrameGrabber>          m_pIFrameGrabber;
   CComPtr<ICaptureGraphBuilder2>  m_pCaptureGraphBuilder;
    CComPtr<IBaseFilter>                    m_pVideoCaptureFilter;

    //
  // Create the capture graph builder and register the
     filtergraph manager.
    //
    m_pCaptureGraphBuilder.CoCreateInstance( CLSID_CaptureGraph
Builder );
    pFilterGraph.CoCreateInstance( CLSID_FilterGraph );
    m_pCaptureGraphBuilder->SetFiltergraph( pFilterGraph );

    //
    // Create and initialize the video capture filter
    //
    m_pVideoCaptureFilter.CoCreateInstance( CLSID_VideoCapture
 );
    m_pVideoCaptureFilter.QueryInterface( &pPropertyBag );

    // We are loading the driver CAM1 in the video capture filter.
    GetFirstCameraDriver( wzDeviceName );
    varCamName = wzDeviceName;
    if( varCamName.vt != VT_BSTR )
    {
        ERR( E_OUTOFMEMORY );
    }

    PropBag.Write( L"VCapName", &varCamName );
    pPropertyBag->Load( &PropBag, NULL );
    // Everything succeeded, the video capture filter is added to
 the filtergraph
    pFilterGraph->AddFilter( m_pVideoCaptureFilter, L"Video
Capture Filter Source" );

    SetCameraResolution();

    // Create and initialize the FrameGrabber filter
    CoCreateInstance( CLSID_FrameGrabber, NULL, CLSCTX_INPROC,
 IID_IBaseFilter, (void**)&pFrameGrabber );
    pFilterGraph->AddFilter( pFrameGrabber, FILTERNAME );

    // Get a pointer to the IFrameGrabber interface
    pFrameGrabber->QueryInterface( IID_IFrameGrabber, (void**)&m_
pIFrameGrabber );
    m_pIFrameGrabber->RegisterCallback( m_Callback);

    pNullRenderer.CoCreateInstance( CLSID_NullRend );
    pFilterGraph->AddFilter( pNullRenderer, L"NullRend" );

    if (ContainsPreviewPin)
    {
    m_pCaptureGraphBuilder->RenderStream(&PIN_CATEGORY_PREVIEW,
 &MEDIATYPE_Video, m_pVideoCaptureFilter, pFrameGrabber, pNull-
Renderer);
    }
    else // no preview pin exists => use capture pin
    {
   m_pCaptureGraphBuilder->RenderStream(&PIN_CATEGORY_CAPTURE,
 &MEDIATYPE_Video, m_pVideoCaptureFilter, pFrameGrabber, pNull-
Renderer);
    }
}
```

Als we stapsgewijs door de BuildCaptureGraph functie heenlopen, dan beginnen we met het aanmaken van een CaptureGraphBuilder die voornamelijk dienst doet als container van de filtergraph. De filtergraph is de daadwerkelijke DirectShow graphmanager, en het is dan ook deze filtergraph waar de filters aan worden toegekend. Zowel de CaptureGraphBuilder als de FilterGraph zijn COM objecten.

VideoCaptureFilter
Vervolgens wordt het eerste filter aangemaakt: de VideoCaptureFilter. Dit filter is, zoals de naam al aangeeft, een sourcefilter dat kan worden gekoppeld aan een cameradevice. Met behulp van een camera GUID en de Windows functie 'FindFirstDevice' kunnen we de naam van de cameradriver vinden die we als property opslaan in de propertybag van de VideoCaptureFilter. Daarna kunnen we het filter aan de filtergraph koppelen.

Resolutie kiezen
Er kan van de default cameraresolutie worden afgeweken door deze expliciet te zetten in 'SetCameraResolution'. In deze functie wordt gebruik gemaakt van de IAMStreamConfig interface die via de 'FindInterface' van de CaptureGraphBuilder kan worden gevonden:

```
CComPtr<IAMStreamConfig> pConfig;
m_pCaptureGraphBuilder->FindInterface(&PIN_CATEGORY_CAPTURE,
 &MEDIATYPE_Video, m_pVideoCaptureFilter, IID_IAMStreamConfig,
 (void**)&pConfig); // if no preview pin available, use capture pin
```

Om vervolgens uit deze configuratie, via GetStreamCaps, de hoogste resolutie te kiezen:

```
AM_MEDIA_TYPE *pmtConfig;
pConfig->GetStreamCaps(iCount - 1, &pmtConfig, (BYTE*)&scc); // use
 highest resolution
pConfig->SetFormat(pmtConfig);
```

Na het aanbrengen van het eerste filter en het zetten van de juiste cameraresolutie, gaan we het tweede filter instantiëren. Dit wordt een transformfilter, en eentje die we zelf maken opdat we via de transformfilter bij de frames kunnen komen die de source filter aan het transform filter doorstuurt.

Transformfilter

De transformfilter, genaamd FrameGrabber, laten we overerven van CTransInPlaceFilter en van IFrameGrabber. De IFrameGrabber is een zelf gedefinieerde interface die een callback functie noteert:

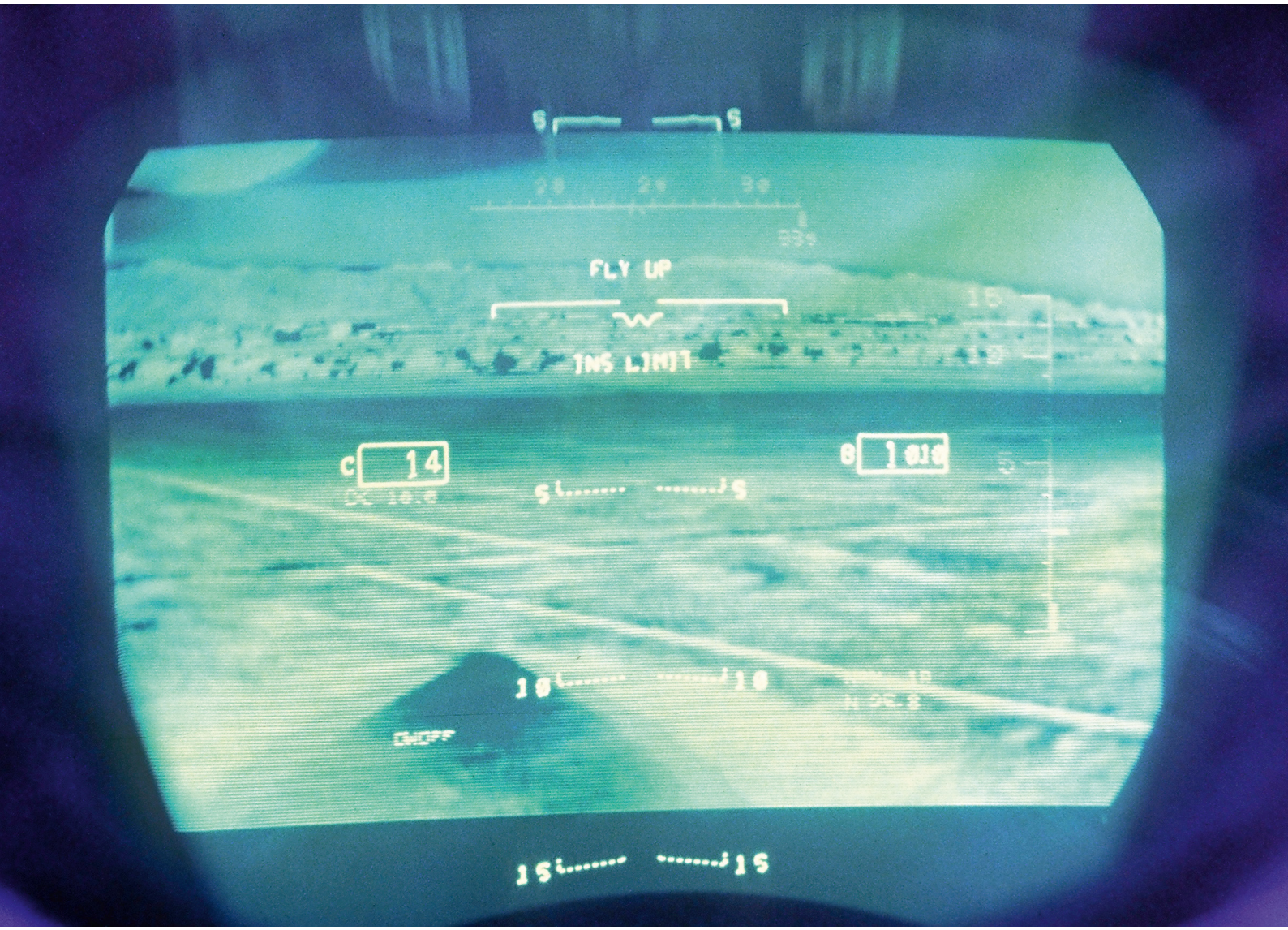
```
DECLARE_INTERFACE_(IFrameGrabber, IUnknown)
{
    STDMETHOD(RegisterCallback)(MANAGED_FRAMEPROCESSEDPROC
callback) PURE;
};
```

De CtransInPlaceFilter is een abstract DirectShow class die aangeeft waar een simpele transformfilter aan moet voldoen. In de Transform functie van deze interface wordt de camera-input meegegeven en is dus de plek waar wij deze input kunnen opvangen en gebruiken voor onze AR bewerking.

```
HRESULT CFrameGrabber::Transform(IMediaSample *pMediaSample)
{
    […]

    BYTE *pCurrentBits;

    CHK( pMediaSample->GetPointer(&pCurrentBits) );
    lSize = pMediaSample->GetSize();

    if ( m_Callback )
    {
m_Callback( pCurrentBits, lSize, m_Height, m_Width, m_Stride
 );
    }

    […]
}
```

NullRenderer

Omdat we het renderen niet over willen laten aan DirectShow



DE HUD IN DEZE F-15E EAGLE STEUNT DE PILOOT MET BEHULP VAN INFRAROOD LICHT BIJ DE NAVIGATIE WANNEER HIJ LAAG VLIEGT.

(we gaan namelijk zelf renderen in OpenGL ES), voegen we als laatste een NullRenderer toe aan de filtergraph. Deze NullRenderer overerft van CbaseRenderer en doet in de 'DoRenderFrame' functie simpelweg niets.

Run

Nu we de filtergraph hebben voorzien van alle benodigde filters, kunnen we aan de graphbuilder doorgeven in welke volgorde deze filters moeten worden gebruikt, op welke pin ze moeten worden ingezet (capture of preview) en om welk mediatype het gaat.

```
m_pCaptureGraphBuilder->RenderStream(&PIN_CATEGORY_PREVIEW,
 &MEDIATYPE_Video, m_pVideoCaptureFilter, pFrameGrabber, pNull-
Renderer);
```

Daarna kunnen we RunCaptureGraph aanroepen en de graphmanager gaan runnen:

```
CComPtr<IGraphBuilder> pGraphBuilder;
CComPtr<IMediaControl> pMediaControl;

[…]

// Retrieve the filtergraph off the capture graph builder
CHK( m_pCaptureGraphBuilder->GetFiltergraph( &pGraphBuilder ));

// Get the media control interface, and run the graph
CHK( pGraphBuilder->QueryInterface( &pMediaControl ));
CHK( pMediaControl->Run());

[…]
```

Nu de graphmanager runt, levert de camera op onze mobiele telefoon via DirectShow camerabeelden op aan onze applicatie, en dat is een mooie start! Wat we nu willen is deze frames binnen onze applicatie tonen.

Stap 2: Rendering

Om de frames op het beeldscherm te kunnen tonen, kunnen we de frames omzetten naar images binnen een C# applicatie. Deze C# applicatie kan via Pinvoke method calling de graphmanager benaderen en de frames van de callback method binnenkrijgen. Willen we echter 3D objecten gaan aanbrengen in ons beeldmateriaal, dan hebben we een product nodig dat 3D bewerkingen mogelijk maakt. DirectX is voor gebruik op Windows Mobile de meest logische keuze. Voor deze demo is OpenGL ES gebruikt omdat OpenGL de mogelijkheid biedt de applicatie ook op andere platformen te laten draaien.

OpenGL ES

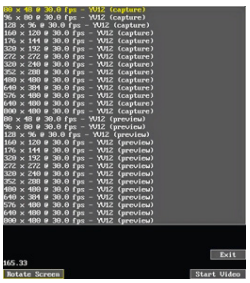
Net als DirectShow is OpenGL geschreven in C++. Met OpenGL kunnen 2D afbeeldingen en 3D objecten worden getoond en gemanipuleerd, OpenGL ES is hierbij de embedded version van OpenGL. Tevens kunnen we in OpenGL onze frames tonen, door deze als textures aan een oppervlakte te koppelen.

Main

In onze main windows functie zien we al direct het een en ander samen komen.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPWSTR lpCmdLine, int nCmdShow)
{
    […]
    LoadOpenGLES()
    […]
    graphmanager = new CGraphManager();
    graphmanager->Init();
    graphmanager->RegisterCallback(OnFrameProcessed);
    graphmanager->BuildCaptureGraph();
    graphmanager->RunCaptureGraph();
    […]
    glGenTextures(2, &textures[0]);
    glBindTexture(GL_TEXTURE_2D, textures[0]);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texturesizeWidth,
texturesizeHeight, 0, GL_RGB, GL_UNSIGNED_SHORT_5_6_5,
NULL);
    glVertexPointer(3, GL_FLOAT, 0, backgroundSurface);
    glTexCoordPointer(2, GL_FLOAT, 0, textureCoordinates);
    […]
    // Main message loop:
    while (running)
    {
 […]
 UpdateApp();
    }
}
```

Als eerste wordt in LoadOpenGLES een opengl display aangemaakt waarop we onze output kunnen renderen. Hiervoor wordt de EGL interface gebruikt die met standaar methodes een koppeling kan realiseren tussen de OpenGL display en het onderliggende Windows systeem. Na de display aangemaakt te hebben wordt ook onze graphmanager gestart en geven we de callback method OnFrameProcessed op als callback method.
Vervolgens declareren we de texture waarop we onze camerabeelden gaan plaatsen.
In glTexImage2D geven we aan dat we met 2D textures gaan werken, en geven de widht en height op van de texture. Dit is de width en height van het cameraframe dat we gaan displayen. Belangrijk is dat deze waardes een macht van 2 moeten zijn en groter dan de width en height van het cameraframe. Als een frame een breedte heeft van bijvoorbeeld 150 px, dan moet een texture width genomen worden van 256 px.
Ook geven we in glTexImage2D de kleurindeling van de camera op. In dit geval maakt de camera gebruik van een RGB indeling verdeeld over 2 bytes: R – 5, G – 6, B – 5. Om te achterhalen welke configuratie jouw camera gebruikt, kun je gebruiken van een handig tooltje genaamd CamTest (zie figuur 3).



FIGUUR 3: MET DE CAMTEST APPLICATIE ZIJN DE CAMERASETTINGS EENVOUDIG TE ACHTERHALEN.

De oppervlakte waarop de texture wordt getoond, wordt aangegeven in glVertexPointer door middel van de backgroundSurface parameter. De positie en ruimte van onze camera frame texture binnen dat oppervlakte wordt aangegeven in glTexCoordPointer middels de textureCoordinates parameter.
Omdat we de cameraframe ons gehele scherm willen laten opvullen, gebruiken we de volgende coordinaten:

```
static const GLfloat backgroundSurface[] =
{
    // use 0.0 as most left bottom point of projectionscreen
    0.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    1.0, 0.0, 0.0,
    1.0, 1.0, 0.0
};

static const GLfloat textureCoordinates[] =
{
    0.0, 0.0,
    0.0, 1.0,
```

```
    1.0, 0.0,
    1.0, 1.0
};
```

Als laatste starten we de applicatieloop waarin we iedere iteratie een call doen naar UpdateApp().

Update
Het cameraframe wordt asynchroon via de callback ontvangen en opgeslagen in een globale variabele:

```
void CALLBACK OnFrameProcessed(BYTE* pData, long len, int height,
 int width, int stride)
{
    videotexture = pData;
}
```

In de UpdateApp methode wordt dit frame als texture gekoppeld aan onze 2D target 'GL_TEXTURE_2D'.

```
void UpdateApp()
{
    // lock on shared resource 'videotexture'
    HANDLE hMutex = CreateMutex(NULL, true, NULL);
    WaitForSingleObject(hMutex, INFINITE);

    glTexSubImage2D(GL _ TEXTURE _ 2D, 0, 0, 0, videoFrameWidth,
videoFrameHeight, GL _ RGB, GL _ UNSIGNED _ SHORT _ 5 _ 6 _ 5,
videotexture);
    InvalidateRect(hWnd, NULL, FALSE);
    UpdateWindow(hWnd);

    ReleaseMutex(hMutex);
}
```

Om te voorkomen dat de globale variabele 'videotexture' wordt overschreven terwijl de texturebinding en displaying nog bezig is, gebruiken we een mutex voor exclusieve toegang tot de resource.

Paint
Wanneer de UpdateWindow in de UpdateApp methode wordt aangeroepen, wordt in WndProc een WM_PAINT message opgevangen, waarna er een paint plaatsvindt van het nieuwe scherm.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
LPARAM lParam)
{
   […]
    glViewport(0, 0, windowWidth, windowHeight);
    glMatrixMode(GL _ MODELVIEW);
    glLoadIdentity();
    glEnableClientState(GL _ VERTEX _ ARRAY);
    glVertexPointer(3, GL _ FLOAT, 0, backgroundSurface);
    glEnable(GL _ TEXTURE _ 2D);
    glDrawArrays(GL _ TRIANGLE _ STRIP, 0, 4);
    […]
}
```

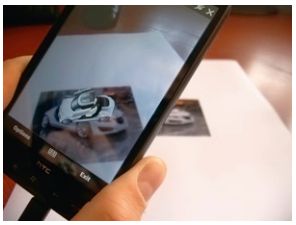
Het is hier dat we, met behulp van de aanroep naar glDrawArrays, onze texture op onze backgroundsurface tekenen en we ons gecapturede cameraframe voor het eerst kunnnen aanschouwen. Omdat we nu 'in control' zijn over de camerabeelden, kunnen we nu ieder frame voorzien van aanvullend materiaal. We kunnen bijvoorbeeld een tekstoverlay creëren, of leuker, een hijskraan bovenop iedere frame renderen. Willen we die hijskraan koppelen aan een marker, om zo die hijskraan een gecontroleerde positie binnen dat frame te geven, dan moeten we eerst die marker zien te localiseren.

Stap 3: Tracking

De basis van tracking bestaat eigenlijk uit het vinden van objecten aan de hand van een kleurenhistogram. Het kleurenhistogram toont de distributie van kleuren in ons camerabeeld en kunnen we met behulp van wiskundige vergelijkingen proberen te matchen met een target kleurenhistogram. Hiervoor is één histogram niet voldoende, omdat de marker die we proberen te vinden zowel dichtbij als ver weg kan zijn. We kunnen tevens loodrecht op de marker kijken, of onder een hoek, etcetera. Hoe meer histogrammen we gebruiken om mee te vergelijken, hoe preciezer de berekening maar natuurlijk ook met meer benodigde rekenkracht als gevolg.
De truuk is nu om het zogenaamde Meanshift Tracking algoritme toe te passen waarin je na het vinden van een object in een frame, bij ieder volgend frame in de buurt van het in het vorige frame gevonden object gaat zoeken. Je zult dan ook merken dat de eerste keer vinden meer tijd kost dan het vervolgens 'vasthouden' van de marker.
Heb je een object gevonden, dan heb je de positie gevonden die we in de vorige paragraaf zochten. Je wilt dan de videotexture gaan bewerken, en dit doe je voordat je deze texture gaat tonen, namelijk in de UpdateApp() method:

```
UpdateApp()
{
    …
    m _ pObjectTracker->ObjeckTrackerHandler(videotexture);
// start tracking
    glTexSubImage2D(GL _ TEXTURE _ 2D, 0, 0, 0, videoFrameWidth,
videoFrameHeight, GL _ RGB, GL _ UNSIGNED _ SHORT _ 5 _ 6 _ 5,
videotexture);
    …
}
```

FIGUUR 4: TRACKING VAN EEN PICTURE MARKER.



Wil je meer weten over tracking en AR in het algemeen, kijk dan eens naar de documentatie van de NyARToolkit of bekijk eens de demo van Metaio. Met de NyARToolkit kun je ook de afstand en hoek tot een bepaald object bepalen om met de juiste translatie van het object deze nog realistischer op de marker te kunnen plaatsen. Tevens kun je daar direct aan de slag met code die het mogelijk maakt om snel je eigen herkenning, tracking, en render functionaliteit te implementeren.

Links

Zie http://www.khronos.org/egl/
http://studierstube.icg.tu-graz.ac.at/handheld_ar/download/CamTest2_20100112.zip
http://www.codeproject.com/KB/GDI-plus/MeanshiftTracking.aspx
http://nyatla.jp/nyartoolkit/wiki/index.php?FrontPage.en
http://www.metaio.com/lizenzen-mobile-demos/lizenzbestaetigung-winmobile-demo/

André van der Plas, is technical lead bij Macaw.