Mit diesen Schritten zur simplen AR-Anwendung

Augmented Reality für Einsteiger

Der vorliegende Artikel vermittelt mit einfachen Codebeispielen erste praktische Umsetzungsschritte für Augmented-Reality-Szenarien mithilfe von Visual Studio 2015 und den Windows-10-Developer-Tools. Damit bietet er eine praktikable Einführung in das spannende Thema der erweiterten Realität durch digitale Einblendungen.

von Sebastian Loose und Peter Brack

Von den einschlägigen Marktbeobachtern schon vor längerer Zeit etwas voreilig prognostiziert, erscheint den Autoren der Einsatz von Augmented-Reality-Technologien für die breite Masse nunmehr tatsächlich vor dem Durchbruch. Unter anderem die Softwareriesen Apple, Google und Microsoft investieren große Summen und bieten mobile Endgeräte an. Darunter auch „Wearables“ – meist in Form von interaktiven Brillen oder Armbändern – sowie erste Programmierschnittstellen, die bereits grundsätzlich tauglich für simple AR-Anwendungsfälle sind.

Das Betriebssystem Windows 10 unterstützt Anwendungen der Universal Windows Platform (UWP) [1]. Gemeint sind damit Applikationen, die auf allen Gerätetypen mit Windows 10 ausgeführt werden können, ob Tablet, Smartphone, Xbox oder PC. Daneben werden Programmierschnittstellen zur Verfügung gestellt, um die für einfache Augmented-Reality-Anwendungsfälle notwendigen Sensoren anzusprechen.

Grundvoraussetzungen zur Entwicklung

Als Entwicklungsumgebung zur Realisierung von UWP-Apps ist Visual Studio 2015 inklusive der optionalen Universal Windows App Development Tools unerlässlich. Es wird empfohlen, auf einem Windows-10-Betriebssystem zu entwickeln; es kann allerdings durchaus auch auf Windows 8.1 programmiert werden. In diesem Fall kann der Code jedoch nur via Emulator oder Remote-Windows-10-Gerät ausgeführt und getestet werden. Da für Augmented-Reality-Anwendungsfälle per definitionem auf diverse Sensoren zugegriffen werden soll, bieten sich insbesondere (Test-)Endgeräte an, die über Kompass und GPS (optional: Beschleunigungssensor und Gyrometer) verfügen. Für die Entwicklung

im Rahmen dieses Tutorials nutzen wir die Sprache C#, alternativ ist die Entwicklung auch in Visual Basic, C++ oder JavaScript möglich.

Einbindung des Kamerasensors

Für die Anzeige von Objekten mit Raumbezug nutzen wir ein einfaches User Control, das das so genannte *CaptureElement* zum Anzeigen des Kamerabilds und ein *Canvas*-Element zum Anzeigen der Objekte integrieren kann. Das *Canvas*-Element soll dabei insbesondere das Kamerabild passgenau überlagern.

In der *Code-Behind*-Datei des eben angelegten User Controls muss nun das *CaptureElement* mit einer Quelle versehen werden. In unserem Fall ist das ein *MediaCapture* aus dem Namespace *Windows.Media.Capture*. Dazu muss das *MediaCapture*-Element zunächst initialisiert und gestartet werden. Hierfür müssen wir zunächst codegesteuert die ID der Kamera des genutzten Endgeräts finden, diese ID der *MediaCaptureInitializationSettings*-Eigenschaft mitteilen und somit dem *MediaCapture*-Element übergeben. Anschließend können wir dem *CaptureElement* die *Source* wiederum zuweisen und die Kamera starten. Listing 1 zeigt dieses Vorgehen exemplarisch ohne Exception Handling.

Einbindung weiterer Sensoren

Um den Status der Sensordaten kontinuierlich abfragen zu können, bietet es sich an, einen *SensorObserver* im Singleton-Entwurfsmuster zu implementierten. Hiermit ist sichergestellt, dass in der App nur eine Instanz jedes Sensors vorhanden ist und die Abfragen darüber hinaus im Hintergrund stattfinden können. Um somit ereignisgesteuert Aktionen in der App ausführen zu können, müssen wir den Dispatcher der *MainPage* verwenden, damit wir wieder in den UI-Thread der Anwendung gelangen können. Essenziell für unsere Anwendung ist beispielhaft der Kompasssensor, der via

Listing 1

```
using Windows.Media.Capture;
using Windows.Devices.Enumeration;

var captureManager = new.MediaCapture();
var devices = await DeviceInformation  .FindAllAsync(DeviceClass.
                                                    VideoCapture);
foreach(var device in devices)
{
  var settings = new MediaCaptureInitializationSettings();
  settings.VideoDeviceId = device.Id;
  await captureManager.InitializeAsync(settings);
  this.PreviewScreen.Source = captureManager;
  await captureManager.StartPreviewAsync();
  return;
}
```

```
var compassSensor = Windows.Devices.Sensors.Compass.GetDefault();
```

initialisiert wird. Anschließend soll das Event *ReadingChanged* abonniert werden, um auf Änderungen des Kompasses zu reagieren:

```
compassSensor.ReadingChanged += CompassReadingChanged;
```

Weiterhin ist beim Umgang mit dem Kompasssensor zu beachten, dass die Ausrichtung nach Norden immer von der Lage des Endgeräts abhängig ist. Je nach Ausrichtung des Endgeräts und somit auch des Sensors muss deshalb ein Offset auf den Kompasswert addiert werden (Listing 2).

Listing 2

```
using Windows.UI.Core;
using Windows.Graphics.Display;

async protected Task CompassReadingChanged(Compass sender,
                                    CompassReadingChangedEventArgs e)
{
  var reading = sender.GetCurrentReading();
  double displayOffset = _viewModel.CompassOffset;

  await _mainPage.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                                                          () =>
  {
    // Calculate the compass heading offset based on
    // the current display orientation.
    var displayInfo = Windows.Graphics.Display.DisplayInformation.
                                                GetForCurrentView();

    switch (displayInfo.CurrentOrientation)
    {
      case DisplayOrientations.Landscape:
        displayOffset += 0;
        break;
      case DisplayOrientations.Portrait:
        displayOffset += 270;
        break;
      case DisplayOrientations.LandscapeFlipped:
        displayOffset += 180;
        break;
      case DisplayOrientations.PortraitFlipped:
        displayOffset += 90;
        break;
    }

    var displayCompensatedHeading = (reading.HeadingMagneticNorth
                                            + displayOffset) % 360;

    _viewModel.UpdateCompass(displayCompensatedHeading);
  });
}
```

Nachdem die Kamera aktiviert wurde und die Aktivitäten der Sensoren überwacht werden, können wir nun beginnen, auf die von den Sensoren erfassten Zustände zu reagieren, Events auszulösen und z. B. eigene Inhalte über das Kamerabild zu legen.

Analog zu den beschriebenen Einbindungen des Kamera- und des Kompasssensors können weitere, für den jeweiligen Anwendungsfall relevante Sensoren eingebunden werden.

Ereignisgesteuerte Darstellung eigener Inhalte

Nachdem die Kamera aktiviert wurde und die Aktivitäten der Sensoren überwacht werden, können wir nun beginnen, auf die von den Sensoren erfassten Zustände zu reagieren, Events auszulösen und z. B. eigene Inhalte über das Kamerabild zu legen. Dazu berechnen wir neben der Distanz den Winkel zwischen unserer Blickrichtung, die wir mithilfe des Kompasses erhalten, und dem anzusprechenden realen Objekt. Hierfür verwenden wir die freigegebenen Spatial Tools Code Samples [2] des Microsoft Tech Ranger Ricky Brundritt (Listing 3).

Ist der Winkel kleiner als der Öffnungswinkel unserer Kamera, dann fügen wir dieses Objekt zur Menge der sichtbaren Objekte hinzu. Anschließend können wir über die Objekte iterieren und abhängig vom Winkel und der Distanz die Position über dem Kamerabild berechnen (Listing 4).

Mögliche Bing-Maps-Integration

Aufbauend auf dem Artikel „Get offline!“ der beiden Autoren zur Entwicklung einer kartenorientierten Windows-10-App [3], bietet sich eine zusätzliche Einbindung der Bing-Maps- Plattform, z. B. für eine Integration von

Listing 3

```
using Windows.Devices.Geolocation;

double fieldOfView = 22.5;  // ergibt 45° absolut
VisibleItems = new ObservableCollection<GeoItem>();
foreach (var item in Items)
{
  var posAsGeopoint = new Geopoint(item.Location.Position);
  var itemHeading = SpatialTools.CalculateHeading(CurrentPosition,
                                                  posAsGeopoint);

  double angle = CurrentHeading - itemHeading;
  if (angle > 180)
    angle = CurrentHeading - (itemHeading + 360);
  else if (angle < -180)
    angle = CurrentHeading + 360 - itemHeading;

  if (Math.Abs(angle) <= fieldOfView)
  {
    var distance = SpatialTools.HaversineDistance(CurrentPosition,
                        posAsGeopoint, SpatialTools.DistanceUnits.KM);
    if (distance <= this.Range)
    {
      item.Distance = distance;
      item.Angle = angle;
      VisibleItems.Add(item);
    }
  }
}
```

Listing 4

```
var visibleItems = viewModel.CalculateItemsInView();
this.ItemCanvas.Children.Clear();

foreach (var item in visibleItems)
{
  double left = 0;
  if (item.Angle > 0)
    left = ItemCanvas.ActualWidth / 2 * ((foV - item.Angle) / foV);
  else
    left = ItemCanvas.ActualWidth / 2 * (1 + -item.Angle / foV);

  double baseline = (ItemCanvas.ActualHeight - 60);
  double top = (baseline) - (baseline * item.Distance / this.Range);

  var metaInfoBlock = new TextBlock() {
    Text = item.Name + " : " + (String.Format(
      "{0:n}", item.Distance * 1000) + "m"),
    Width = 128, Height = 50,
    TextWrapping =TextWrapping.Wrap };

  StackPanel aRItem = new StackPanel { Orientation = Orientation.
                                                    Horizontal,
    Background = new SolidColorBrush(Colors.White), Opacity = 0.5 };
  arItem.Children.Add(metaInfoBlock);

  Canvas.SetLeft(aRItem, left);
  Canvas.SetTop(aRItem, top);
  this.ItemCanvas.Children.Add(arItem);
}
```

Routingfunktionalitäten, innerhalb der Augmented-Reality-Applikation an. Über Adresseingaben für Start und Ziel könnten Fußgänger- und Autoroutings ausgelöst werden, um letztendlich ganz real vom aktuellen Standort zum avisierten Ziel zu gelangen. Die ebenfalls für Windows-10-Betriebssysteme zur Verfügung gestellten 3-D-Ansichten und Offlinekartenfunktionalitäten können die Augmented-Reality-Applikation darüber hinaus eindrucksvoll und sinnvoll ergänzen.

Ausblick

Just zum Zeitpunkt des Verfassens dieses Artikels erschien die Ankündigung zum Release der interaktiven Microsoft-HoloLens-Brille für Entwickler. Die Entwicklung so genannter „Holographic-Apps“ [4] mittels Unity wird das anhaltende Interesse an dieser Technologie weiterhin steigern und vermutlich größere Verbreitung finden. Im Prinzip sind alle Anwendungsfälle, in denen räumlich orientierte Zusatzinformationen zur Realität sinnvoll sein können, für den Einsatz solcher Applikationen prädestiniert. Vor allem bei Anwendungsfällen rund um Sicherheitsthemen (z. B. Brand-/Katastrophenschutz) erscheinen den Autoren die Einsatzzwecke für AR-Technologien besonders vielfältig und sinnhaft.

Peter Brack ist Geschäftsfeldleiter Business Geo Intelligence (BGI) bei der Fichtner IT Consulting AG (FIT) und Microsoft MVP mit der Kompetenz Bing Maps Development. Er beschäftigt sich seit den späten 90er-Jahren mit Businesskartenapplikationen und ist Mitautor des Buchs „Aufbruch in die Geoinformationsgesellschaft mit Microsoft Bing Maps“. Er berichtet regelmäßig über Neuigkeiten rund um das Thema Bing Maps z. B. via Twitter.

@bgisolutions

Sebastian Loose ist Diplom-Ingenieur für Computervisualistik und als Software Engineer bei der Fichtner IT Consulting AG (FIT) tätig. Der Microsoft Certified Professional realisiert seit Jahren komplexe IT-Lösungen auf Basis der Bing Maps for Enterprise Platform für internationale Großunternehmen.

Links & Literatur

[1] Develop apps for the Universal Windows Platform: https://msdn.microsoft.com/en-us/library/dn975273.aspx

[2] Location Intelligence for Windows Store apps: http://bit.ly/1SOiRFp

[3] Brack, Peter; Loose, Sebastian: „Get offline! Bing Maps in einer Windows-10-Universal-App“, in: Windows Developer 9.2015

[4] Start building holographic apps: www.microsoft.com/microsoft-hololens/en-us/developers