Augmented reality for fun using Nokia Imaging SDK filters

This article explains how to create interesting, fun and useful augmented reality apps with real-time filters using GART (Geo Augmented Reality Toolkit), one of the best open source geo augmented reality libraries out there, with the Nokia Imaging SDK on Windows Phone 8.

06 Apr 2014

Note: This is an entry in the Nokia Imaging and Big UI Wiki Competition 2013Q4.

Introduction

XAML

Augmented Reality takes a live, real-time view of the real world, and overlays (*augments*) it with computer-generated elements, such as graphics and sound, to give it more meaning. The idea of augmented reality is more than a hundred years old. However, to date we have only scratched the surface of augmented reality possibilities. Five years from now, you could be driving your car, while viewing information about road conditions, navigation, and oncoming pedestrians and animals crossing the road in low light conditions, all laid out nicely on your windshield. Or in the medical world, a surgeon could wear special glasses that let her overlay different types of images over a patient's body during surgery, giving her access to up to date and relevant information about the patient at all times.

Writing augmented reality apps

To implement augmented reality in a mobile application, you would generally usually take a view of the world; often from a camera image, and overlay it with information about the contents of the image.

Note: Getting camera frames is easy with Windows Phone APIs, but this topic is beyond the scope of the article and has been described many times before. You can find more details about how to do this in the 'See Also' section.

Once you have a camera image, you overlay it with user interface elements that are related to the image. This might be XAML controls, a simple TextBlock, or even something more complex.

The way we decide what to show and where to show it, and also what to hide, depends on the type of augmented reality app we develop.

Simple augmented reality apps start with a frame from the camera as a background. Next, the location of the device is calculated based on where the person holding the device is standing, and the position in which they're holding the device. Their geographical location is typically represented as latitude and longitude, and the orientation of the device using 3 angles - yaw, pitch and roll. By combining that information, the app can calculate exactly what point of interest is in front of the device at any moment. Information about the points of interest can then be fetched from a simple web service, defined inside the app in XML or JSON file. This type of app is often called Geo Augmented Reality.

Imagine a scenario. You're in Paris, standing quite close to Eiffel Tower. You take your device, turn on your augmented reality app, and point the device at the Eiffel Tower. The app displays frames from the camera that show the Eiffel Tower. It overlays the frames with a UI control that tells us that what we're seeing is, in fact, the famous Eiffel Tower. It also gives a short description of of the Eiffel Tower. Tapping on the UI takes us to another page that contains detailed information about the Eiffel Tower.

An example of a geo augmented reality app is shown on the next picture.

In this article, we'll be building a Geo Augmented Reality application. We'll use:

- the device's location sensor in conjunction with the Windows Phone Runtime Location API to detect the device's position.

(C) Copyright Nokia 2014. All rights reserved.

- the various hardware sensors, including the accelerometer, compass and gyroscope, combined in Windows Phone Motion API, to determine the orientation and motion of the device.
- GART, an open source (MS-LPL license) library which combines all the sensor data and data about points of interest to create the augmented reality portion of the application.

For the purposes of this demonstration, we'll hard-code the augmented reality data that is displayed.

Location sensor

The location data the device provides comes from multiple sources including GPS, Wi-Fi, and cellular towers. There are two different sets of APIs that can be used to incorporate location data into the application. The first is .NET location API which is recommended in case you're targeting both Windows Phone 7 and Windows Phone 8 platforms. The second one is Windows Phone Runtime Location API, used on Windows Phone 8 and convergent with Windows 8. Further differences can be found in the official MSDN documentation. This article targets Windows Phone 8 apps, so the explanation of how to use the Windows Phone Runtime Location API follows. Basically, all we need to do is define the Geolocator object and attach position changed event handler:

```
Geolocator geolocator = new Geolocator();
geolocator.DesiredAccuracy = PositionAccuracy.High;
geolocator.MovementThreshold = 100; // note - this is in meters

geolocator.PositionChanged += geolocator_PositionChanged;
```

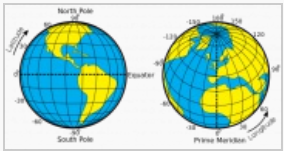
In the position changed event handler, we can easily grab the current latitude and longitude information:

```
void geolocator_PositionChanged(Geolocator sender, PositionChangedEventArgs args)
{
    var latitude = args.Position.Coordinate.Latitude;
    var longitude = args.Position.Coordinate.Longitude;
}
```

Wherever we stand on earth, we have a unique combination of latitude and longitude coordinates.

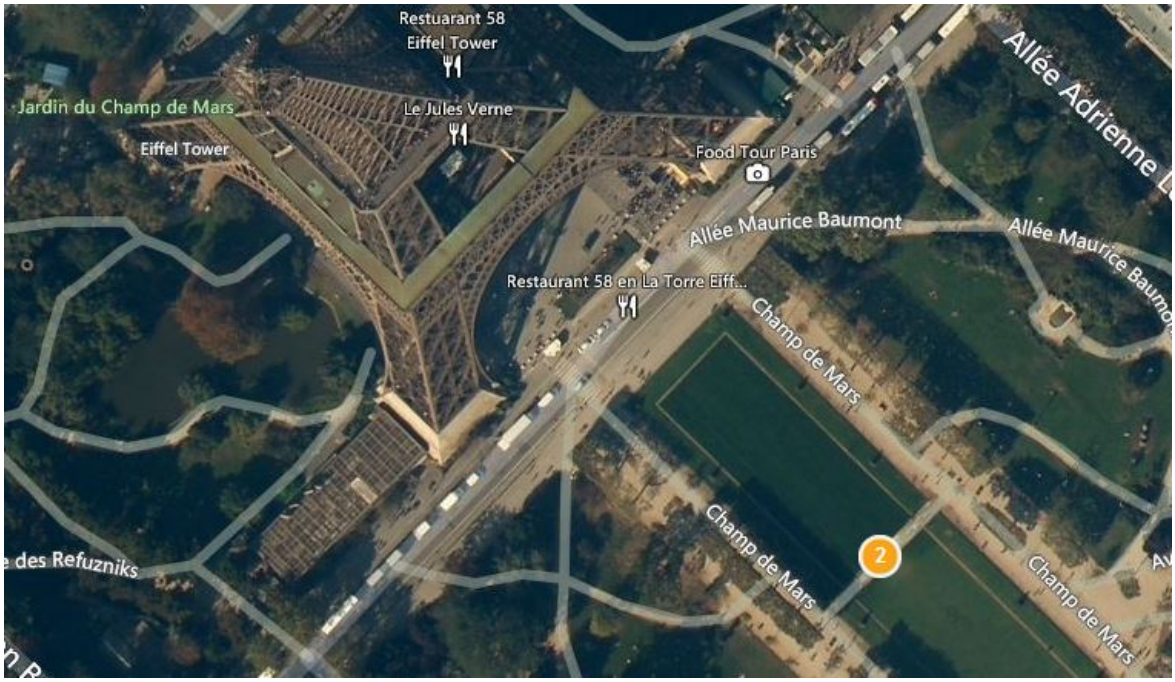
For example, if we're standing close to Eiffel Tower as shown on the next picture, our location is defined by:

- Latitude: 48.857284
- Longitude: 2.296018



Latitude and longitude illustration

(C) Copyright Nokia 2014. All rights reserved.



Note: In some cases, that may not be enough to describe where we are - we might also be elevated, so a third parameter can be used - it's called altitude.

However, having just the location isn't enough. The fact that we are so close to Eiffel Tower does not mean that we're looking right at it. If we turned our back on it and looked the other way, we wouldn't want to show the UI telling us that we're facing the Eiffel Tower. We need information from other sensors in the device to determine the exact orientation of the device.

Sensors

Windows Phones support multiple sensors. These sensors allow apps to determine the orientation and motion of the device. Typical uses of sensors as input include motion-controlled games (think racing games, for example) and augmented reality apps. The sensors which are commonly a part of Windows Phone devices are accelerometer, compass and gyroscope. Every Windows Phone device has an accelerometer (this is a hardware requirement), but other sensors are optional. Besides accelerometer, devices more often have a compass and not a gyroscope, but all the high-end devices have a compass and a gyroscope besides accelerometer. This is important because information from these sensors can be combined to determine the orientation of the device more precisely. It's possible to access all the individual sensors through Windows Phone API, but Microsoft also made it really easy for developers to use this information combined in so called Motion API.

Note: Implementing access to individual sensors is beyond the scope of this article. For detailed information about Windows Phone sensors, check out the 'See Also' section of this article

Let's shortly describe the individual sensors, though, for better understanding of how geo augmented reality works.

Accelerometer

Accelerometer measures the forces applied to the device at a moment in time. These forces can be used to determine in which direction the user is moving the device. When the device is lying steady on the table, the only force that is applied to it is gravitation. The acceleration value is expressed as a 3-dimensional vector representing the acceleration components in the X, Y, and Z axes. Those are expressed in gravitational units. It's often used in racing games to simulate a steering wheel, and it's required in every Windows Phone device.

Compass

Compass essentially tells us the angle by which the device is rotated relative to the Earth's magnetic north pole. Also known as magnetometer. Not required in every device.

Gyroscope

Gyroscope sensor is used to determine the rotational velocity of the device in each axis. Not required in every device.

Motion API

Complex geometrical calculations are needed to translate the raw data from these three sensors into the true orientation of the

(C) Copyright Nokia 2014. All rights reserved.

device, which is what's needed in augmented reality apps. The Motion class handles the low-level sensor calculations and allows apps to easily obtain the device's attitude (yaw, pitch, and roll). By knowing yaw, pitch and roll values, we know the orientation of the device in space.

Note: Motion uses accelerometer, compass and gyroscope if it can find all three sensors in a device, but works OK (less precisely, though) even if your device has only accelerometer and compass.

Motion API is really easy to implement. Create a Motion object and attach the `CurrentValueChanged` event handler.

```
Motion motion;

...
if (motion == null)
  {
    motion = new Motion();
    motion.CurrentValueChanged +=
        new
EventHandler<SensorReadingEventArgs<MotionReading>>(motion_CurrentValueChanged);
  }

  // Try to start the Motion API.
  try
  {
    motion.Start();
  }
  catch (Exception ex)
  {
    // do something if Motion not available
  }
```

In the event handler for current value changed we get the yaw, pitch and roll values.

Note: Yaw, pitch and roll angles are given in radians. We can easily convert them to degrees just in case we need to display more meaningful values by using MathHelper.ToDegrees()

```
void motion_CurrentValueChanged(object sender, SensorReadingEventArgs<MotionReading> e)
{
  var yaw = MathHelper.ToDegrees(e.SensorReading.Attitude.Yaw);
  var pitch = MathHelper.ToDegrees(e.SensorReading.Attitude.Pitch);
  var roll = MathHelper.ToDegrees(e.SensorReading.Attitude.Roll);
}
```

Read more about yaw, pitch and roll angles in this Wikipedia article

Having location and orientation is basis for geo augmented reality apps. Now all we need to do is combine them to get meaningful information - when are we close to a certain point of interest? When are we facing it? This is where open source library GART becomes extremely helpful.

GART

GART is an open source (MS-LPL license) library for creating geo augmented reality apps on Windows Phone and Windows 8 platforms. All you need to provide is a collection of objects that have latitudes and longitudes. These can come from pretty much anywhere, but in real life scenarios, they will probably come from a web service. The framework takes care of managing sensors and tracking where the user is in relation to the reference points. It can show where the points are in relation to user from a top-down perspective (on a map) or it can show where the points are as a virtual lens into the real world. This framework combines all that we've covered so far, and even adds the map view, which makes it possible to concentrate on content, design and extension of the framework, without losing too much time doing the calculations behind, which are basically reusable in all sorts of different

(C) Copyright Nokia 2014. All rights reserved.

geo augmented reality projects.

Note: GART, just like any other piece of code written in the history of mankind, has it's bugs and quirks. Still, it remains the best open source library for geo augmented reality and provides a great starting point to develop such apps

GART can be implemented in 6 easy steps.

1. Add an ARDisplay control from the toolkit to your Windows Phone page in XAML
2. Add the views you want as children of the ARDisplay control.
3. GART services need to be started and stopped - using convenient Start and Stop methods when navigating to and from the page. This handles all the sensors for us.
4. Create a collection of ARItem objects (or your own custom type that inherits from ARItem), which must have Geolocation properties.
5. Set items source of the ARDisplay control defined in the first step to the collection you created in fourth step
6. Optionally, style the UI elements which are displayed on top

Note: Knowing how to use GART is a prerequisite for the rest of the article. It's been written about GART before, so check the 'See Also' section to get familiar with GART implementation before continuing

Main GART control is ARDisplay, which you define in XAML. The naming of classes in GART is really self-explanatory - ARDisplay class name stands for *Augmented Reality Display*. However, the ARDisplay control is not enough. It needs to have different views defined inside, such as a map view or world view overlayed with UI elements such as TextBlocks etc. Therefore ARDisplay is like a container for different views. The framework takes care of your location, sensors and even map keeping the abstraction at a high level - when you write an app using GART, you don't have to bother with all that stuff if you don't want to. But the beauty of it being open source is that you can, if you want.

Possible layers/views are:

- Video from the camera (VideoPreview)
- Map (OverheadMap)
- World view, UI elements (WorldView)
- Heading indicator (HeadingIndicator)

You can use none of those (not much point in doing that, though), or just some of them. However, make sure you add them in the exact order shown in all the GART demos - that's very important.

Note: Having a large heading indicator should be reconsidered in real-life scenarios and apps because it adds a slight performance drawback which can be especially problematic when doing real-time filtering

The points of interest are defined as ARItem objects. Again, the naming is obvious - it stands for *Augmented Reality Item*. That class can (and should) be inherited to add more properties because it only has the important ones predefined, including GeoCoordinate to define where exactly each of the points of interest is on Earth.

Simple demo GART app can be made in matter of hours and looks really great. See the following demo video which shows one such demo app. The only difference is that it was taken in Croatia, and not in Paris, France.

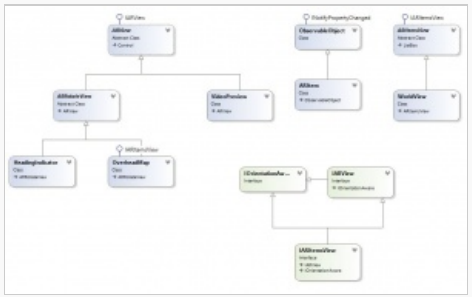
Extending GART with real-time filtering

(C) Copyright Nokia 2014. All rights reserved.

GART is easy to use, but is a lot more fun if we incorporate Nokia Imaging SDK in it. It's released under Microsoft Limited Permissive License (MS-LPL) on Codeplex, which means that the following modifications made to GART are published under the same license. Unfortunately, it was impossible to fork it directly on Codeplex, but I forked it to create a WP8 version called AR.ForFun (Augmented Reality For Fun - the name was inspired by Microsoft's Coding4Fun, but the library serves for much more than just simple fun) and uploaded all the code to GitHub.

Note: The code for the modified GART library including a sample app which demonstrates how to use it is available for download from GitHub repository

The most important part of GART class diagram is shown on the following picture



GART class diagram

You probably recognize some of the classes mentioned in the previous chapter. We will be adding a few new classes to the project. Most of the editing will be done in ARDisplay class.

Let's start by opening GART source code which can be downloaded from Codeplex.

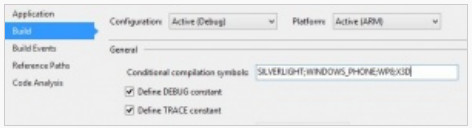
Note: Before unpacking, it may be necessary that you right-click on the zip file, click on properties, and unblock the zipped file

When unpacked, the WP8 project is usually in the following location:

location-where-you-unpacked-gart-zip\gart-VERSION-NUMBER\GeoARToolkit\Lib\WP8\GART

Open it in Visual Studio 2012 or above, but be careful - there's a different project file created just for VS2013. So if you're using VS2013, use that project file. Define the following "Conditional compilation symbols" in the project properties:

SILVERLIGHT;WINDOWS_PHONE;WP8;X3D



Conditional compilation symbols for WP8 build

These are necessary because GART library was written to support WP7, WP8 and Windows 8 (you'll notice a lot of #IFs to handle different platforms in one code base) and we want it to build for Windows Phone 8 only because of the Nokia Imaging SDK.

Next step is adding the Nokia Imaging SDK via Nuget. This effectively means that the library now needs to be built for either ARM or X86, for people to use depending on whether they test it on a real device or on emulator.

Warning: Make sure that the right build configuration is set, especially if you download the complete final solution for this article from GitHub

Our goal is to implement real-time filtering of the camera frames using Nokia Imaging SDK 1.0. We will do it in 4 major steps

1. Replace PhotoCamera with PhotoCaptureDevice in ARDisplay class
2. Implement LiveFilters class and Filters property in ARDisplay
3. Implement FrameStreamSource class to take and render filtered frames
4. Enable circular browsing through defined list of filters

In the end, we'll show how to use it in a demo app, and how it all works with a short demo video.

Replacing PhotoCamera with PhotoCaptureDevice

Let's start by pushing out PhotoCamera class which was used in WP7 and introducing PhotoCaptureDevice instead, which offers

(C) Copyright Nokia 2014. All rights reserved.

more possibilities than PhotoCamera and let's us work more easily with preview frames coming from camera. We use preview frames because they are much smaller in size so they're much easier to process with different filters in real-time. Even though they are smaller in size, they are in good enough quality for the phone screen size.

Replace PhotoCamera variable and property in `ARDisplay` class using PhotoCaptureDevice variable and property:

```
private PhotoCaptureDevice photoCaptureDevice;

/// <summary>
/// Gets the PhotoCaptureDevice used by the ARDisplay.
/// </summary>
#if WINDOWS_PHONE
[Category("AR")]
public PhotoCaptureDevice PhotoCaptureDevice
{
    get
    {
            return photoCaptureDevice;
    }
 }
#endif
```

The method for creating a VideoBrush object needs to use a different value for Rotation parameter, called SensorRotationInDegrees, because we replaced PhotoCamera with PhotoCaptureDevice:

```
#if WINDOWS_PHONE
private void CreateVideoBrush()
{
    VideoBrush vb = new VideoBrush();
    vb.RelativeTransform = new CompositeTransform { CenterX = 0.5, CenterY = 0.5,
Rotation = photoCaptureDevice.SensorRotationInDegrees };
    VideoSource = vb;
}
#endif
```

Next places where we won't be using PhotoCamera are methods for stopping and starting the camera. Stop method needs to be rewritten to use the new photoCaptureDevice object

```
private void StopCamera()
{
    if (photoCaptureDevice != null)
    {
        photoCaptureDevice.Dispose();
        photoCaptureDevice = null;
    }
}
#endif
```

and the Start method needs to be a bit more complicated than before, also adding some changes to support real-time filtering:

```
private async void StartCamera()
{
    // this gets the lowest preview resolution
    var resolution =
PhotoCaptureDevice.GetAvailablePreviewResolutions(CameraSensorLocation.Back).Last();
```


(C) Copyright Nokia 2014. All rights reserved.

```
    // we set the photoCaptureDevice to use the back camera
    photoCaptureDevice = await PhotoCaptureDevice.OpenAsync(CameraSensorLocation.Back,
resolution);

    await photoCaptureDevice.SetPreviewResolutionAsync(resolution);

    // If our video brush hasn't been created yet, create it
    VideoBrush vb = VideoSource as VideoBrush;
    if (vb == null)
    {
        CreateVideoBrush();
        vb = VideoSource as VideoBrush;
    }

    // initialize the FrameStreamSource object
    frameStreamSource = new FrameStreamSource(photoCaptureDevice, Filters);

    // we will use the MediaElement as a source for videobrush, because that's where we
will be rendering our frames after applying real time filtering
    mediaElement = new MediaElement();
    Canvas.SetZIndex(mediaElement, 1);
    mediaElement.Stretch = Stretch.UniformToFill;
    mediaElement.BufferingTime = new TimeSpan(0);
    mediaElement.SetSource(frameStreamSource);

    vb.SetSource(mediaElement);
}
```

You'll notice two main differences in this method other than just replacing PhotoCamera with PhotoCaptureDevice. We're using `FrameStreamSource` class which inherits from MediaStreamSource. MediaStreamSource gives developers direct access to APIs for manipulating encoded elementary video streams. This is a great way to render a stream of images as we apply filters on them. The implementation will be shown later on in the article. MediaElement makes it easy to show the rendered frames and therefore serves as a source for the VideoBrush object.

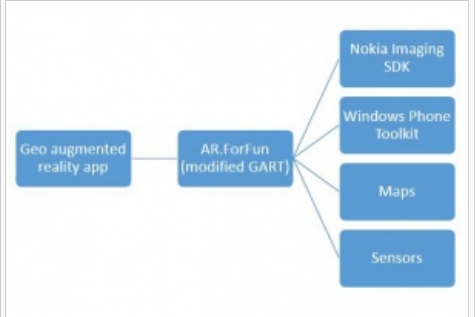
The two variables (FrameStreamSource and MediaElement objects) are defined in the ARDisplay class, among all the other private variables.

```
#if WINDOWS_PHONE
private GeoCoordinateWatcher locationService;
private Motion motion;
....
private PhotoCaptureDevice photoCaptureDevice;
private FrameStreamSource frameStreamSource;
private MediaElement mediaElement;
#endif
```

Implementing LiveFilters and adding Filters property to ARDisplay

You may have noticed earlier that the Nokia Imaging SDK has been added to the GART library, so we want to define an enumeration of filters to enforce abstraction between the main app that will use this modified GART library (which will not be using Nokia Imaging SDK reference) and the modified GART library. In other words, we don't want to tamper with Nokia Imaging SDK objects and methods in the app which consumes the modified GART library to keep the same level of abstraction that is typical for GART. The following picture explains the abstractions we're trying to achieve:

(C) Copyright Nokia 2014. All rights reserved.



Library/app abstraction

We want to simplify GART usage for developer using the library, so we create a simple enum of filters that can be used to define which real-time filters will be enabled. For this article, 15 filters have been added to enum definition, but you can add as many as you like, even your own custom ones, because after all this is just an enum, which means it's important how we interpret it on the library side.

Add a **LiveFilter.cs** class to Data folder in GART library.

```
public enum LiveFilter
{
    None,
    Antique,
    AutoEnhance,
    AutoLevels,
    Cartoon,
    ColorBoost,
    Grayscale,
    Lomo,
    MagicPen,
    Negative,
    Noise,
    Oily,
    Paint,
    Sketch,
    Vignetting,
    WhiteboardEnhancement
};
```

Notice the None value. We want to be able to always make sure that one of the filters is a default view, the view when no filters are applied - raw frames coming from the camera. The filters are set through a dependency property in ARDisplay class called Filters:

```
static public readonly DependencyProperty FiltersProperty =
DependencyProperty.Register("Filters", typeof(ObservableCollection<LiveFilter>),
typeof(ARDisplay), new PropertyMetadata(new ObservableCollection<LiveFilter>(),
OnFiltersChanged));

private static void OnFiltersChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
{
    ((ARDisplay)d).OnFiltersChanged(e);
}

protected virtual void OnFiltersChanged(DependencyPropertyChangedEventArgs e)
{
    Filters = (ObservableCollection<LiveFilter>)e.NewValue;
}
```


(C) Copyright Nokia 2014. All rights reserved.

Implementing FrameStreamSource class

Now let's take a look at the implementation of FrameStreamSource class. It inherits the class MediaStreamSource, which means it has to override some of it's methods. Let's take a look at the class, piece by piece, starting from the variables defined on top.

```
public class FrameStreamSource : MediaStreamSource
{
    private readonly Dictionary<MediaSampleAttributeKeys, string> emptyAttributes = new
Dictionary<MediaSampleAttributeKeys, string>();

    private MediaStreamDescription videoStreamDescription = null;
    private MemoryStream frameStream = null;

    private Size frameSize = new Size(0, 0);
    private int frameBufferSize = 0;
    private byte[] frameBuffer = null;
    private PhotoCaptureDevice photoCaptureDevice;
    private CameraPreviewImageSource cameraPreviewImageSource;
    private ObservableCollection<LiveFilter> liveFilters;

    private IFilter activeFilter;
    private int currentFilterIndex = 0;
```

The most important objects are the MemoryStream which will hold our filtered frame. frameSize is the size of the preview image. photoCaptureDevice is the PhotoCaptureDevice object that we defined earlier when starting the camera (we pass it in through constructor). CameraPreviewImageSource implements IImageProvider which means that it can be used as a source in FilterEffect class. The ObservableCollection of LiveFilters keeps a list of chosen live filters. The activeFilter is the currently active filter which we apply on every frame.

Next is the constructor:

```
public FrameStreamSource(PhotoCaptureDevice pcd, ObservableCollection<LiveFilter>
liveFilters)
{
    this.photoCaptureDevice = pcd;
    var smallestPreview = this.photoCaptureDevice.PreviewResolution;
    this.liveFilters = liveFilters;
    if (this.liveFilters == null || this.liveFilters.Count == 0)
    {
        this.liveFilters = new ObservableCollection<LiveFilter>();
    }

    this.liveFilters.Insert(0, LiveFilter.None);
    currentFilterIndex = 0;
    ConvertFilterEnumToIFilter(this.liveFilters[currentFilterIndex]);

    cameraPreviewImageSource = new CameraPreviewImageSource(photoCaptureDevice);
    frameSize = new Size(smallestPreview.Width / 2, smallestPreview.Height / 2);
    frameBufferSize = (int)frameSize.Width * (int)frameSize.Height * 4; // RGBA
    frameBuffer = new byte[frameBufferSize];
    frameStream = new MemoryStream(frameBuffer);
}
```

The constructor takes the PhotoCaptureDevice object and ObservableCollection<LiveFilter> collection of filters as parameters. It sets the current active filter to None, because we don't want any filters applied when the app is first run, and sets the size of the

(C) Copyright Nokia 2014. All rights reserved.

preview image to one half of the smallest size. This is to ensure better performance because the high quality image is of less importance in these kind of scenarios, especially when we apply fun filters such as Cartoon filter which distort the image anyway. In the end of the constructor is initialization of the buffers and MemoryStream.

Next we override OpenMediaAsync() method:

```
protected override void OpenMediaAsync()
{
    var mediaStreamAttributes = new Dictionary<MediaStreamAttributeKeys, string>();

    mediaStreamAttributes[MediaStreamAttributeKeys.VideoFourCC] = "RGBA";
    mediaStreamAttributes[MediaStreamAttributeKeys.Width] =
((int)frameSize.Width).ToString();
    mediaStreamAttributes[MediaStreamAttributeKeys.Height] =
((int)frameSize.Height).ToString();

    videoStreamDescription = new MediaStreamDescription(MediaStreamType.Video,
mediaStreamAttributes);

    var mediaStreamDescriptions = new List<MediaStreamDescription>();
    mediaStreamDescriptions.Add(videoStreamDescription);

    var mediaSourceAttributes = new Dictionary<MediaSourceAttributesKeys, string>();
    mediaSourceAttributes[MediaSourceAttributesKeys.Duration] =
TimeSpan.FromSeconds(0).Ticks.ToString(CultureInfo.InvariantCulture);
    mediaSourceAttributes[MediaSourceAttributesKeys.CanSeek] = false.ToString();

    ReportOpenMediaCompleted(mediaSourceAttributes, mediaStreamDescriptions);
}
```

This is done to inform the MediaElement that the MediaStreamSource has been opened and to supply information about the streams it contains. We are using RGBA streams (red, green, blue, alpha to describe every pixel of a frame).

Next we override the GetSampleAsync method. The MediaElement calls this method to ask the MediaStreamSource to prepare the next MediaStreamSample of the requested stream type for the media pipeline.

```
protected override void GetSampleAsync(MediaStreamType mediaStreamType)
{
    var task = GetNewFrameAndApplyChosenEffectAsync(frameBuffer.AsBuffer());

    task.ContinueWith((action) =>
    {
        if (frameStream != null)
        {
            frameStream.Position = 0;

            var sample = new MediaStreamSample(videoStreamDescription, frameStream, 0,
frameBufferSize, 0, emptyAttributes);

            ReportGetSampleCompleted(sample);

        }
    });
}
```

This method calls the GetNewFrameAndApplyChosenEffectAsync with the byte array as a parameter. That method gets the new

(C) Copyright Nokia 2014. All rights reserved.

frame and applies the chosen filter:

```
private async Task GetNewFrameAndApplyChosenEffectAsync(IBuffer buffer)
{
    var lineSize = (uint)frameSize.Width * 4;
    var bitmap = new Bitmap(frameSize, ColorMode.Bgra8888, lineSize, buffer);

    IFilter[] filters;

    if (activeFilter == null)
    {
        filters = new IFilter[0];
    }
    else
    {
        filters = new IFilter[]
        {
            activeFilter
        };
    }

    using (FilterEffect fe = new FilterEffect(cameraPreviewImageSource)
    {
        Filters = filters
    })
    using (BitmapRenderer renderer = new BitmapRenderer(fe, bitmap))
    {
        await renderer.RenderAsync();
    }
}
```

It creates a new FilterEffect using the cameraPreviewImageSource IImageProvider with Filters array which is basically either an empty array (meaning that we don't do real-time filtering - render raw camera frame) or an array with one element - currently active filter. Then we use a BitmapRenderer to render the image to Bitmap object. After the GetNewFrameAndApplyChosenEffectAsync is done, GetSampleAsync continues with creating MediaStreamSample based on the image which was rendered to the memory stream defined earlier. After we're done, we call

```
ReportGetSampleCompleted(sample);
```

with the newly created MediaStreamSample.

It's also necessary to override the SeekAsync method by just calling the ReportSeekCompleted. We are not doing any seeking on the frames.

```
protected override void SeekAsync(long seekToTime)
{
    ReportSeekCompleted(seekToTime);
}
```

To convert the LiveFilter enum values to a real IFilter implementation, the following method is used:

```
private void ConvertFilterEnumToIFilter(LiveFilter liveFilter)
{
    switch (liveFilter)
    {
```


(C) Copyright Nokia 2014. All rights reserved.

```
        case LiveFilter.None:
            activeFilter = null;
            break;
        case LiveFilter.Antique:
            activeFilter = new AntiqueFilter();
            break;
        case LiveFilter.AutoEnhance:
            activeFilter = new AutoEnhanceFilter();
            break;
        case LiveFilter.AutoLevels:
            activeFilter = new AutoLevelsFilter();
            break;
        case LiveFilter.Cartoon:
            activeFilter = new CartoonFilter();
            break;
        case LiveFilter.ColorBoost:
            activeFilter = new ColorBoostFilter();
            break;
        case LiveFilter.Grayscale:
            activeFilter = new GrayscaleFilter();
            break;
        case LiveFilter.Lomo:
            activeFilter = new LomoFilter();
            break;
        case LiveFilter.MagicPen:
            activeFilter = new MagicPenFilter();
            break;
        case LiveFilter.Negative:
            activeFilter = new NegativeFilter();
            break;
        case LiveFilter.Noise:
            activeFilter = new NoiseFilter();
            break;
        case LiveFilter.Oily:
            activeFilter = new OilyFilter();
            break;
        case LiveFilter.Paint:
            activeFilter = new PaintFilter();
            break;
        case LiveFilter.Sketch:
            activeFilter = new SketchFilter();
            break;
        case LiveFilter.Vignetting:
            activeFilter = new VignettingFilter();
            break;
        case LiveFilter.WhiteboardEnhancement:
            activeFilter = new WhiteboardEnhancementFilter();
            break;
        default:
            activeFilter = null;
            break;
    }
}
```

Note: These filters are all defined with their default parameters. One possible improvement in your own implementation

(C) Copyright Nokia 2014. All rights reserved.

could be to allow defining the filter parameters.

Enabling circular browsing through defined list of filters

We want to be able to browse through predefined set of filters from a WP8 app. We want it to be as simple as possible on the app side, so whenever a user taps on a button, the TakeNextFilter from ARDisplay class is called, which takes and applies the next filter from the collection:

```
public void TakeNextFilter() // this is a part of ARDisplay class
{
    if (frameStreamSource != null)
    {
        frameStreamSource.NextFilter();
    }
}
```

The TakeNextFilter method calls the NextFilter method defined in frameStreamSource, which just iterates through the collection of filters in a circular manner:

```
public void NextFilter() // this is a part of FrameStreamSource class
{
    if (currentFilterIndex == liveFilters.Count - 1)
    {
        currentFilterIndex = 0;
    }
    else
    {
        currentFilterIndex++;
    }
    ConvertFilterEnumToIFilter(liveFilters[currentFilterIndex]);
}
```

Testing the changes with a sample project

Sample project consists of the built library, either through referencing a .dll or adding the enhanced GART (AR.ForFun) source code project to the solution and referencing it from there.

There are a couple of changes needed compared to the basic sample version showed in some of the GART articles from 'See Also' section. First of all, let's make sure we have the proper app bar buttons. On the MainPage of your WP8 app, add the following app bar buttons:

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="False">
        <shell:ApplicationBarIconButton x:Name="nextButton" IconUri="/Images/arrow.png"
Text="next filter" Click="next_Click"/>
        <shell:ApplicationBarIconButton x:Name="mapButton" IconUri="/Images/map2.png"
Text="map" Click="mapButton_Click"/>
        <shell:ApplicationBarIconButton x:Name="worldButton" IconUri="/Images/eye.png"
Text="world" Click="world_on_off_Click"/>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

The next button will shift through the defined filters. The map button will show or hide the map. The world button will toggle the visibility of the world view. The event handlers are:

```
private void next_Click(object sender, EventArgs e)
```


(C) Copyright Nokia 2014. All rights reserved.

```
{
    ardisplay.TakeNextFilter(); // applying the next filter from the collection is as
simple as this
}

private void mapButton_Click(object sender, EventArgs e)
{
    UIHelper.ToggleVisibility(overheadMap);
}

private void world_on_off_Click(object sender, EventArgs e)
{
    UIHelper.ToggleVisibility(worldView);
}
```

We're calling the TakeNextFilter() method from ARDisplay class which is a part of the modified GART framework.

Next, other than starting the services on navigating to MainPage, it's also necessary to define the filters we wish to be able to browse:

```
protected override void OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e)
{
    ardisplay.Filters = new ObservableCollection<LiveFilter>
    {
        LiveFilter.Grayscale,
        LiveFilter.Lomo,
        LiveFilter.Cartoon,
        LiveFilter.Paint,
        LiveFilter.AutoLevels,
        LiveFilter.AutoEnhance,
        LiveFilter.WhiteboardEnhancement
    };

    ardisplay.StartServices();
    base.OnNavigatedTo(e);
}
```

If you don't want to add any ARItems yourself, you can simply generate dummy 'Lorem ipsum' points of interest on a click of a button or when the page is loaded, like this:

```
var currentLocation = ardisplay.Location; // make sure to not do it immediately on
navigating to page because your location may not have yet been properly fetched
Random rand = new Random();
for (int i = 0; i < 5; i++) // adding 5 items
{

    GeoCoordinate coordinate = new GeoCoordinate()
    {
        Latitude = currentLocation.Latitude + ((double)rand.Next(-90, 90)) / 100000,
        Longitude = currentLocation.Longitude + ((double)rand.Next(-90, 90)) / 100000,
    };

    locationsTvrda.Add(new CityPlace()
    {
        GeoLocation = coordinate,
        Content = "Lorem ipsum " + i,
```


(C) Copyright Nokia 2014. All rights reserved.

```
        Description = "Quisque tincidunt lorem vitae porta gravida. Nullam vitae suscipit risus, sit amet sagittis arcu. In elementum lacinia turpis, vel commodo eros consequat nec. Nam vitae tristique metus, a sagittis nibh. Etiam id purus vel elit egestas semper ac ac ipsum. Etiam id commodo ligula, quis ultrices nibh. "
    });
}
```

When the app is started, it's very easy to browse through different filters applied to our world view in real time by tapping on the button in AppBar. Makes augmented reality even more exciting! This can also be a great start for many fun and useful scenarios, such as games, utility apps etc.

Check out the video below which shows the demo app in use:

Summary

In this article, we explained what augmented reality is and what's the difference between the most common types of augmented reality apps. Then we focused on geo augmented reality apps, explaining how they work and what's the best way to get started with such apps using open source libraries. By incorporating Nokia Imaging SDK real-time filtering in the most popular library, GART, we make augmented reality more exciting and give more possibilities and scenarios to developers. Code samples are shown throughout the article, along with the two videos. The first one demonstrates usage of simple GART sample, and the second one shows GART modified to use Nokia Imaging SDK real-time filtering. The whole solution with the source code of both the modified GART (so called AR.ForFun) and the sample app which uses it are available on GitHub.

(C) Copyright Nokia 2014. All rights reserved.