Unity and OpenCV – Part one: Install

As a creative developer, sooner rather than later you will require computer vision or image processing in your projects. It allows user interaction through cameras, for instance, but it can also be used to process your rendered frames directly should the need arise. OpenCV is a world class computer vision library, available free of charge – though if you found this tutorial, you are probably already aware of this. Whenever you Google Unity and OpenCV though, you are immediately greeted with a (relatively expensive) paid asset, which I personally find… well, let's not go into what I think about asking money for the work others provided for free.

Needless to say, in this set of tutorials I will show how to integrate C++ OpenCV with Unity, as the native version will always be the fastest and the most memory efficient. Don't let the C++ bit scare you off – OpenCV is set up to be very high level, so don't go thinking you require a C# (or even Java) wrapper. If you've experimented with the most prominent C# wrapper, EmguCV, and Unity before, you will know that the performance leaves something to be desired.

Before we get any work done in Unity though, we need to set up OpenCV itself. If you already have all the .dll's and .lib's ready to go, skip on ahead to the next part. If you don't, and especially if you've never set up OpenCV before, read on – I remember getting started to be quite tricky and not very well documented, so I'd like to make it as straight forward as possible for my readers. I'm on Windows, and I don't have any other OS'es laying around, so that's all the help I can offer.

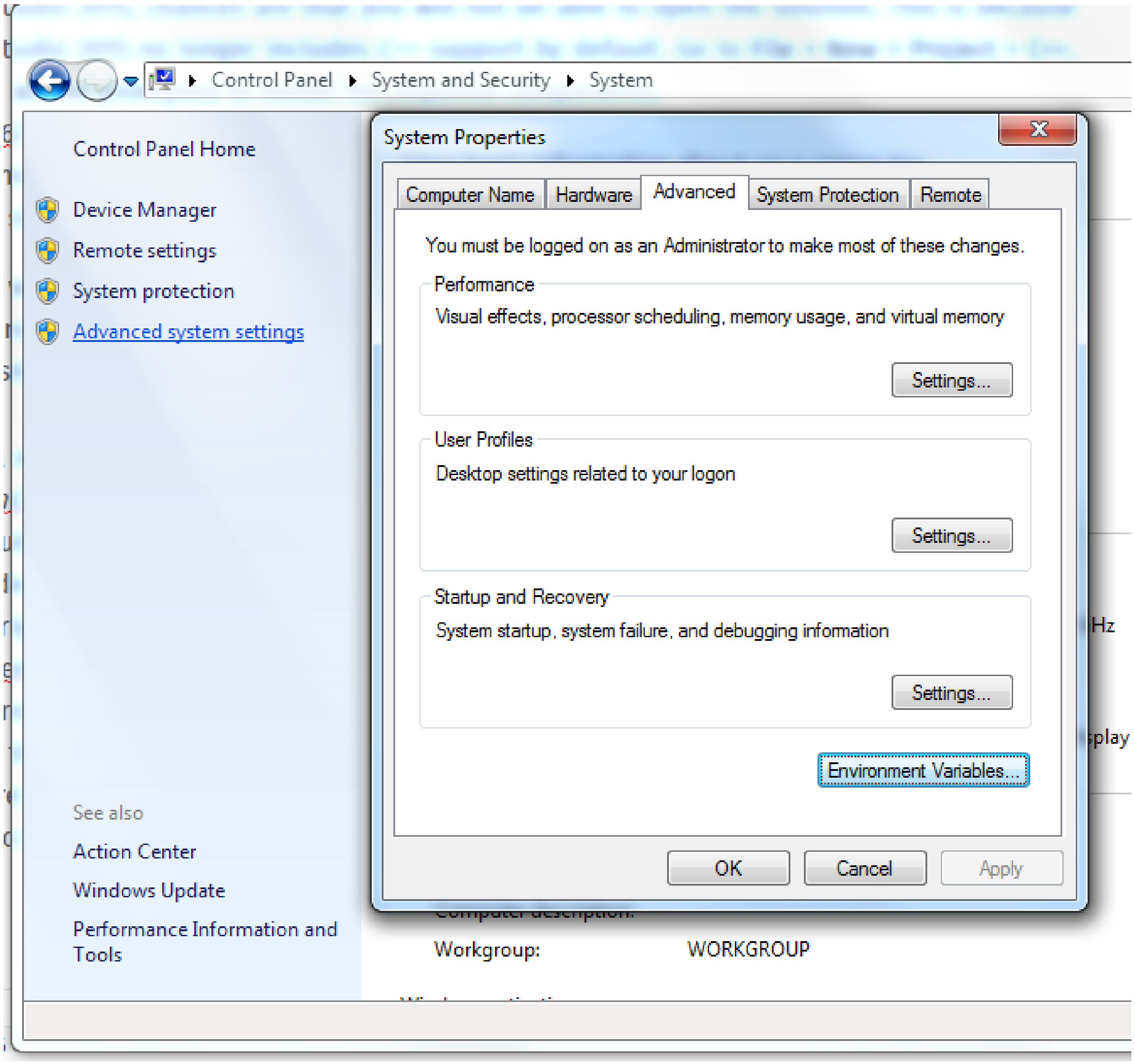
The C++ libraries for OpenCV aren't shipped as direct downloads, but rather need to be built by you. This allows users to more easily manage the vast number of included libraries (and significantly decreases file size), though it makes getting started with the basics a bit more daunting. I'll walk you through it – you'll be ready to go in about 15 minutes.

1. Download the current release from opencv.org. I will be using OpenCV 3.1 in these tutorials, and I suggest you do the same (at least until this tutorial grows horribly outdated). [2019 edit: OpenCV 4 still works just fine with this tutorial]

2. Run the .exe and extract the files to a temporary location – you won't need these anymore soon enough. I will be extracing them to *G:\Temp\opencv unpack*.
3. Download and install CMake. CMake is a tool for building packages, widely used in the open source C++ world, and it's required to build OpenCV.
4. Run CMake. At the top, click "Browse Source", and navigate to the directory where you unpacked OpenCV in step 2. Under that directory, select /opencv/sources as the source folder for CMake. In my case, that would be *G:\Temp\opencv unpack\opencv\sources*.
5. Select a new temporary location as build directory for CMake. I chose *G:\Temp\OpenCVInstall*.
6. With the two paths set up, go ahead and click "Configure" at the bottom. Once CMake is done, you will see a large number of red items appear in the center area of the application. These are all the build options I mentioned earlier – right now it will suffice to simply stick to the defaults. Remember that this is where you'll have to look if certain features are not included, like OpenNI support for the Kinect, for instance.
7. Click "Generate", select the Visual Studio version you're using (most likely **Visual Studio 14 2015 Win64**) and let CMake do its thing. If you're using MonoDevelop, then maybe now is the time to evaluate certain life choices.
8. Now navigate to the build directory you set up in step 5, and open OpenCV.sln. If you're using Visual Studio 2015, chances are that you will not be able to open the solution. This is because Visual Studio 2015 no longer includes C++ support by default. Go to **File > New > Project > C++**, and you will be prompted to install the required components.
9. Make sure the INSTALL project is set as the active project in the Solution Explorer, then Build (F6) both the Debug and Release **x64** version of OpenCV. You should give Visual Studio some time to initialize the projects and scan all #included files before doing so – wait for the "Ready" status at the bottom of the window.

The builds will take a couple of minutes. Now would be the ideal time to take a breather – stretch your legs and your eyes, grab a coffee, or have a chat at the water cooler. Once the builds are done, we can finish setting everything up.

1. Create a new directory, which will serve as the permanent home for your OpenCV files. I chose *G:\OpenCV\OpenCV 3.1*
2. In the build directory from step 5, you will now have a *bin* and a *lib* folder. Copy both to this new folder.
3. In the original unpack folder, navigate to opencv/build (or opencv/install), and copy the *include* folder to your final OpenCV directory. You should now have *G:\OpenCV\OpenCV 3.1\bin, G:\OpenCV\OpenCV 3.1\lib* and *G:\OpenCV\OpenCV 3.1\include*, or at least your equivalent of the same.
4. You can go ahead and delete all the temporary files now, as we have everything we need.
5. Now it's time to add the OpenCV directory to Window's Path system variable – this just serves to tell the OS where to look for dll's, so you don't have to copy the dll's to every OpenCV project you make. Go to the **Control panel > System > Advanced system settings** and click **Environment Variables…**.



You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your logon

Startup and Recovery
System startup, system failure, and debugging information

Workgroup: WORKGROUP

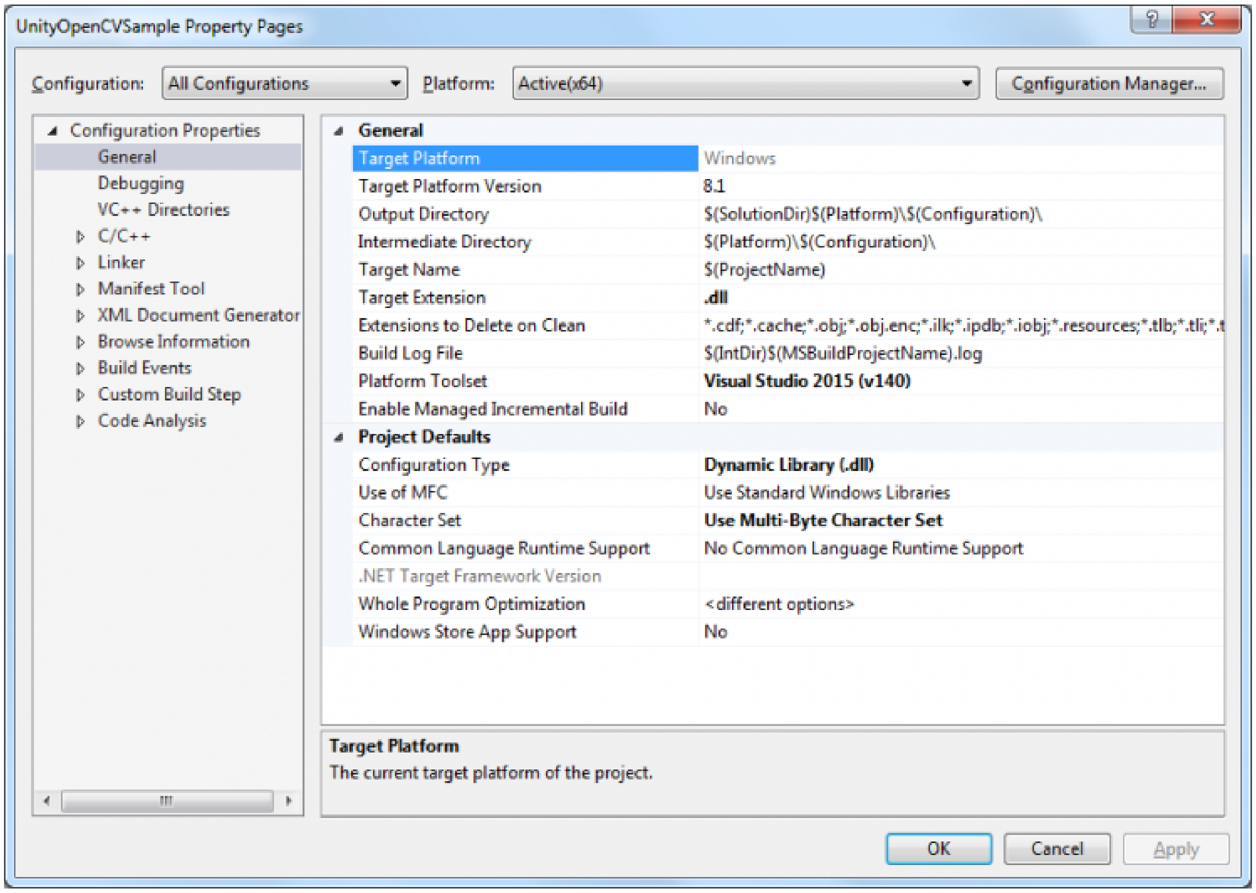
1. Under "System variables", add a new entry called **OPENCV_DIR**, and assign the path defined in step 10.
2. Finally, in the User variables section above the system variables, find the PATH entry, and click edit. At the end, add **%OPENCV_DIR%\bin**.

That's it for now! In the next part, we'll set up an OpenCV project that will be usable from within Unity.

Unity and OpenCV – Part two: Project setup

After installing OpenCV in the previous part, we can now set up the first C++ project – the steps listed below apply to all OpenCV projects, not just to use within Unity. Let's dive right in!

1. Create a new empty C++ project in Visual Studio.
2. Add a first .cpp to the project – nothing needs to be in there yet, it's just necessary to get all the project properties.
3. At the top of the window, change the platform to x64 – we compiled the 64 bit OpenCV libraries earlier after all.
4. Open up the project properties and switch the configuration to *All Configurations* so both Release and Debug are changed simultaneously. Change the target extension to .dll, and the configuration type to the same – we need a .dll to include in our Unity project.


UnityOpenCVSample Property Pages

Configuration: All Configurations | Platform: Active(x64)

General	
Target Platform	Windows
Target Platform Version	8.1
Output Directory	$(SolutionDir)$(Platform)\$(Configuration)\
Intermediate Directory	$(Platform)\$(Configuration)\
Target Name	$(ProjectName)
Target Extension	**.dll**
Extensions to Delete on Clean	*.cdf;*.cache;*.obj;*.obj.enc;*.ilk;*.ipdb;*.iobj;*.resources;*.tlb;*.tli;*.t
Build Log File	$(IntDir)$(MSBuildProjectName).log
Platform Toolset	**Visual Studio 2015 (v140)**
Enable Managed Incremental Build	No
Project Defaults	
Configuration Type	**Dynamic Library (.dll)**
Use of MFC	Use Standard Windows Libraries
Character Set	**Use Multi-Byte Character Set**
Common Language Runtime Support	No Common Language Runtime Support
.NET Target Framework Version	
Whole Program Optimization	<different options>
Windows Store App Support	No

Target Platform
The current target platform of the project.

1. Under *C/C++ > General*, add **$(OPENCV_DIR)\include** to the “Additional Include Directories” field. If you recall the previous part, we defined **OPENCV_DIR** as a system user variable, pointing to where we copied the OpenCV .dll’s, .lib’s and .hpp’s (includes). In case you are not yet familiar with C++ and wonder what these includes are about, for now it suffices to know that includes are what’s necessary to tell a .cpp file what classes exist and which members they have – this has to be defined manually, unlike C#, which knows all about other classes in the same project, or projects that are referenced.

2. In the *Linker > General* options, change the “Additional Library Directories” property to **$(OPENCV_DIR)\lib\Debug** and **$(OPENCV_DIR)\lib\Release** for the Debug and Release configuration respectively. We built both the debug and release versions in the previous part, and this is where we need ’em.

3. Finally, under *Linker > Input*, edit the “Additional Dependencies”. For the Debug configuration, add **opencv_core310d.lib** and **opencv_highgui310d.lib**. Notice the

"d" at the end of the name – it signifies the debug version. The "310" is the version number. For the Release configuration, add the same two files, only without the "d".

To test that everything is set up correctly, add the following two lines to the .cpp file you created in step 2:

```
#include <opencv2\core.hpp>
#include <opencv2\highgui.hpp>
```

Build (F6) in both the Debug and Release configurations. If you encounter an error, go through the steps again and double check spelling. Also check the directories in your OpenCV install directory from part one, make sure all the required folders and files are there, and you defined the system variables correctly. If any are missing, go through part one once more.

If both builds succeeded, then congratulations! We'll meet again in the next part, when we actually have the project do something useful and integrate it with Unity.

Unity and OpenCV – Part three: Passing detection data to Unity

We'll now integrate OpenCV face detection into Unity. In this part, the camera stream and pixel processing will be done within OpenCV, and we will only send the location and size of the detected faces to Unity. This approach is used for applications which don't need to overlay any visuals onto the camera stream, but only require the OpenCV data as a form of input.

Let's start on the C++ side. First, add these files to the dependencies (as shown in the previous part):

opencv_core310.lib

opencv_highgui310.lib

opencv_objdetect310.lib

opencv_videoio310.lib

opencv_imgproc310.lib

Here's the full Source.cpp which is used to track faces and send their location to Unity. I will not cover the actual OpenCV code – it's mostly just sample code, and the scope of this tutorial is purely to introduce you to a way of making OpenCV and Unity communicate in an optimized way.

```
#include "opencv2/objdetect.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace std;
using namespace cv;

// Declare structure to be used to pass data from C++ to Mono.
struct Circle
{
  Circle(int x, int y, int radius) : X(x), Y(y), Radius(radius) {}
  int X, Y, Radius;
};

CascadeClassifier _faceCascade;
String _windowName = "Unity OpenCV Interop Sample";
VideoCapture _capture;
int _scale = 1;

extern "C" int __declspec(dllexport) __stdcall  Init(int& outCameraWidth, int& outCameraHeight)
{
  // Load LBP face cascade.
  if (!_faceCascade.load("lbpcascade_frontalface.xml"))
    return -1;

  // Open the stream.
  _capture.open(0);
  if (!_capture.isOpened())
    return -2;

  outCameraWidth = _capture.get(CAP_PROP_FRAME_WIDTH);
  outCameraHeight = _capture.get(CAP_PROP_FRAME_HEIGHT);

  return 0;
}

extern "C" void __declspec(dllexport) __stdcall  Close()
{
  _capture.release();
}

extern "C" void __declspec(dllexport) __stdcall SetScale(int scale)
{
  _scale = scale;
}
```

```
extern "C" void __declspec(dllexport) __stdcall Detect(Circle* outFaces, int maxOutFacesCount, int& outDetectedFacesCount)
{
  Mat frame;
  _capture >> frame;
  if (frame.empty())
    return;

  std::vector<Rect> faces;
  // Convert the frame to grayscale for cascade detection.
  Mat grayscaleFrame;
  cvtColor(frame, grayscaleFrame, COLOR_BGR2GRAY);
  Mat resizedGray;
  // Scale down for better performance.
  resize(grayscaleFrame, resizedGray, Size(frame.cols / _scale, frame.rows / _scale));
  equalizeHist(resizedGray, resizedGray);

  // Detect faces.
  _faceCascade.detectMultiScale(resizedGray, faces);

  // Draw faces.
  for (size_t i = 0; i < faces.size(); i++)
  {
    Point center(_scale * (faces[i].x + faces[i].width / 2), _scale * (faces[i].y + faces[i].height / 2));
    ellipse(frame, center, Size(_scale * faces[i].width / 2, _scale * faces[i].height / 2), 0, 0, 360, Scalar(0, 0, 255), 4, 8, 0);

    // Send to application.
    outFaces[i] = Circle(faces[i].x, faces[i].y, faces[i].width / 2);
    outDetectedFacesCount++;

    if (outDetectedFacesCount == maxOutFacesCount)
      break;
  }

  // Display debug output.
  imshow(_windowName, frame);
}
```

We obviously start with a couple of imports and namespace using statements. Then, we declare a struct: this structure will be used to pass data directly from the unmanaged C++ code into the managed Unity scripts. This will be covered in more detail once we get to the Unity side of things. The structure is made to suit the application’s needs – you are free to change this as required.

```
struct Circle
{
  Circle(int x, int y, int radius) : X(x), Y(y), Radius(radius) {}
  int X, Y, Radius;
};
```

Next up, we have all the methods which can be called from within Unity. Because we are using C++, we need to explicitly tell the compiler how to expose these methods. Normally, the C++ compiler will mangle the method names when packaging them into a .dll. Therefore, we instruct it to use the classic “C” style of signatures, which leaves the method names just as you wrote them. You will always have to use this syntax when exposing C++ methods to a managed application.

```
extern "C" void __declspec(dllexport) __stdcall Detect(Circle* outFaces, int maxOutFacesCount, int& outDetectedFacesCount)
```

Important to note here are the parameters **Circle* outFaces** and **int& outDetectedFacesCount**. The first one is a pointer to a Circle struct, indicating here that we are sending an array of Circles to Detect(). The latter indicates that outDetectedFacesCount is sent by reference.

Compile the project as x64 Release, and copy the resulting .dll to the Assets/Plugins folder in your Unity project. You will also need to copy the OpenCV .dlls to that same folder, as our own .dll depends on those. The OpenCV .dll’s were compiled in the first part, and can be found in \OpenCV 3.1\bin\Release.

It can be a bit tricky to know exactly which .dll’s you need – copying just the ones declared in the #include statements isn’t enough, as these in turn are dependent on other .dll’s. You can use Dependency Walker on our .dll to figure out exactly which .dll’s are required, or if you’re feeling a bit lazy, you can just copy all of the OpenCV .dll’s. If Unity tells you our .dll can’t be loaded even though it’s in the Plugins folder, it’s because dependencies are missing.

A final thing you will need to copy is the cascade classifier .xml. In this sample, I'm using the lbp frontal face cascade – lbp cascades are the significantly faster than haar cascades, though slightly less accurate. You will need to copy it from your OpenCV directory into the working directory of your Unity application – when you're within the editor, this is the root project directory.

With all the files in place, we can get to the Unity scripts. Create a new script called OpenCVFaceDetection, and copy this underneath the generated class:

```
// Define the functions which can be called from the .dll.
internal static class OpenCVInterop
{
    [DllImport("UnityOpenCVSample")]
    internal static extern int Init(ref int outCameraWidth, ref int outCameraHeight);

    [DllImport("UnityOpenCVSample")]
    internal static extern int Close();

    [DllImport("UnityOpenCVSample")]
    internal static extern int SetScale(int downscale);

    [DllImport("UnityOpenCVSample")]
    internal unsafe static extern void Detect(CvCircle* outFaces, int maxOutFacesCount, ref int outDetectedFacesCount);
}
```

The static OpenCVInterop class exposes to C# all the C++ methods we just marked as dllexport. Note that the method signatures have to match. The DllImport attribute takes the file name of your dll.

Underneath that, add this structure declaration. It needs to have the same exact fields as the one declared in C++, in the same order, and it must be marked to have a sequential layout. This way we'll be able to read the struct data coming from the unmanaged environment.

```
// Define the structure to be sequential and with the correct byte size (3 ints = 4 bytes * 3 = 12 bytes)
[StructLayout(LayoutKind.Sequential, Size = 12)]
public struct CvCircle
{
    public int X, Y, Radius;
}
```

This is the class itself:

```
public class OpenCVFaceDetection : MonoBehaviour
{
    public static List<Vector2> NormalizedFacePositions { get; private set; }
    public static Vector2 CameraResolution;

    /// <summary>
    /// Downscale factor to speed up detection.
    /// </summary>
    private const int DetectionDownScale = 1;

    private bool _ready;
    private int _maxFaceDetectCount = 5;
    private CvCircle[] _faces;

    void Start()
    {
        int camWidth = 0, camHeight = 0;
        int result = OpenCVInterop.Init(ref camWidth, ref camHeight);
        if(result < 0)
        {
            if(result == -1)
            {
                Debug.LogWarningFormat("[{0}] Failed to find cascades definition.", GetType());
            }
            else if(result == -2)
            {
                Debug.LogWarningFormat("[{0}] Failed to open camera stream.", GetType());
            }

            return;
        }
        CameraResolution = new Vector2(camWidth, camHeight);
        _faces = new CvCircle[_maxFaceDetectCount];
        NormalizedFacePositions = new List<Vector2>();
        OpenCVInterop.SetScale(DetectionDownScale);
        _ready = true;
    }

    void OnApplicationQuit()
    {
        if(_ready)
        {
            OpenCVInterop.Close();
        }
    }

    void Update()
    {
        if (!_ready)
            return;

        int detectedFaceCount = 0;
        unsafe
        {
            fixed(CvCircle* outFaces = _faces)
            {
                OpenCVInterop.Detect(outFaces, _maxFaceDetectCount, ref detectedFaceCount);
            }
        }

        NormalizedFacePositions.Clear();
        for(int i = 0; i < detectedFaceCount; i++)
        {
            NormalizedFacePositions.Add(new Vector2((_faces[i].X * DetectionDownScale) / CameraResolution.x, 1f - ((_faces[i].Y * Detection
        }
    }
}
```

The important bit happens in Update(): in an **unsafe** block, we call OpenCVInterop.Detect(), and pass the **fixed** pointer of an array of CvCircle. This means that the C++ OpenCV code will write the detected faces directly into this struct array we defined in C#, without the need for performance heavy copies from unmanaged space into managed space. This is a good trick to know for any C++ interop you may have to do in the future.

Because we don't know how many faces will be detected, we create the array at a predefined size, and ask our C++ code to tell us how many faces were actually detected using a by ref integer. We also pass the array size to C++ to prevent buffer overflows.

In case you are not familiar with the two above keywords, **unsafe** simply allows you to use pointers in C#, and **fixed** tells the compiler that the given variable has to stay at its assigned position in memory, and is not allowed to be moved around by the garbage collector – otherwise the C++ code could inadvertently be writing to a different bit of memory entirely, corrupting the application.

This same procedure can be used to pass an array of pixels between OpenCV and Unity without having to copy it, allowing you to display video footage from OpenCV within Unity, or passing a WebcamTexture stream to OpenCV for processing. That is beyond the scope of this part, however.

[2020 edit] To use `unsafe`, in recent Unity versions (starting from at least 2018) you can now tick *Allow unsafe code* in the player preferences under "Configuration", and you can ignore the below info.

[Original post] In order to be able to use **unsafe**, we need to add a file called "mcs.rsp" to the root asset folder, and add the line "-unsafe" to it. (in versions before 5.5 you may need to use either smcs.rsp for .NET 2.0 subset, or gmcs.rsp for the full .NET 2.0). This file is an instruction to the compiler to allow unsafe code.

While this will let Unity compile your scripts, Visual Studio will still complain when you try to debug with an **unsafe** block – normally you add a flag in the project properties, but Visual Studio Tools for Unity blocks access to those. To be able to debug, you will have edit the .csproj (root project folder) manually, and set the two **<AllowUnsafeBlocks>false</AllowUnsafeBlocks>** lines to true. You will have to do this after every script change since the .csproj is recreated by Unity after every compile, so it'll be useful to comment out the **unsafe** lines when you're working on something else in the project.

In this sample, I'm sending the face viewport positions to a list, to be consumed by other scripts such as this one, which simply moves an object at that position:

```
using UnityEngine;

public class PositionAtFaceScreenSpace : MonoBehaviour
{
    private float _camDistance;

    void Start()
    {
        _camDistance = Vector3.Distance(Camera.main.transform.position, transform.position);
    }

    void Update()
    {
        if (OpenCVFaceDetection.NormalizedFacePositions.Count == 0)
            return;

        transform.position = Camera.main.ViewportToWorldPoint(new Vector3(OpenCVFaceDetection.NormalizedFacePositions[0].x, OpenCVFaceDetec
    }
}
```

That wraps up this part – hopefully I've taught you enough to set you on the path. Good luck!

[2020 Edit] Community member Iseta has set up a full Github repo with everything from this tutorial, which should help you get set up even faster!