

Extract portrait and replace background(image matting) by code, using C# .NET and OpenCVSharp

Abstract: This article introduces a method of using OpenCVSharp to "extract the portrait and replace the background" for the green screen video in the camera in real time, and analyzes the algorithm in the project. This article presents a method to simplify the release of managed resources such as Mat and MatExpr in OpenCVSharp. This article also introduces "efficient camera playing control" and OpenCVSharp performance optimization technology, including efficient reading and writing memory of Mat instances, and how to avoid inefficient code, etc.

1. Why develop image matting software by myself

Because of the need of my work, I need an application that can "subtract the background, replace the background, and display the new image a window" for the portrait in the camera in real time. Many conference live streaming software have this function. For example, Zoom, Microsoft Teams and others all have image matting functions, but these functions can only be used in their applications. I tried several other software, including XSplit Vcam, Douyin live-streaming assistant, and OBS. They all had excellent functions, including many AI portrait matting functions without green screens. They are powerful, but they could not meet my special requirements. So I need to develop such a software by myself.

A typical image matting requires to put green screens behind the object being matted, and then the green screen is removed by computer, so that the portrait will be retained, and then the cut-out portrait will be drawn on the new background. A lot of film and television productions are made on this principle. As shown in Figure 1 [1].



Figure 1

As long as the lighting is good, it is very accurate to matting through the green screen. However, the disadvantage of this way is that the layout requirements for the site are very high. So now the "no green screen matting" is popular, that is, AI can identify the foreground portrait and background. XSplitt Vcam has this function, and it can simulate the result of image cutout into a virtual camera for output, but if the background is more complex, XSplitt Vcam removal background effect is still not ideal. I am not an expert on computer vision, especially AI, therefore, in the case of limited time, I don't think I can write a more powerful than Vcam software, so I decided to still use the traditional green screen form to implement the function, after all, as long as spend a few bucks for a piece of green screen.

Before I get started with the implementation code, let me show you how the software works. Figure 2 is the original image collected by the camera, and you can see a green screen behind, while Figure 3 is the result of image matting, and it is a real-time matting, which can achieve about 20FPS (frames per second).



Figure 2 Original image



Figure 3 The image after matting

2. Architecture

The software uses OpenCV, which is a very mature and feature-rich computer vision library. OpenCV supports C/C++, Python, .NET, Java and other mainstream programming languages. OpenCV development in Python is the most widely used resource. Personally, I don't like Python syntax very much, so I developed this software on the .NET 5 using C#. Because OpenCV is used in much the same way across programming languages, the code implemented here in C# is also very easy to be rewritten in another programming language.

There are two OpenCV binding libraries on .NET: OpenCVSharp and Emgu CV. Because OpenCVSharp has no commercial use restrictions, I use OpenCVSharp in this project. However, even if you are using an Emgu CV, the code in this article can be applied to Emgu CV.

3. How to get source code

Since the function of image matting is only a part of my software, the whole software is not convenient to open source for the time being, so I separated the core code function of green screen replacing background into a new open source project.

Source code repository: <https://github.com/yangzhongke/Zack.OpenCVSharp.Ext>

"GreenScreenRemovalDemo. Cs" is the main code, and you can get the executable binaries at the bottom of the page for all kinds of operating systems, and the GreenScreenRemovalDemo is the entry executable.

In Windows, for example, running GreenScreenRemovalDemo. Exe, the console will appear as shown in Figure 4.

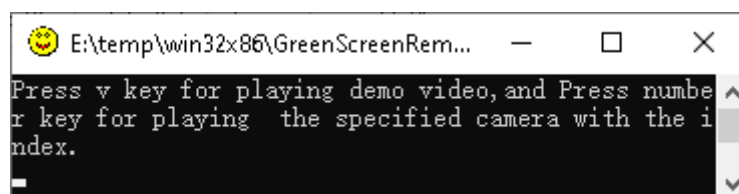


Figure 4

If you enter V, a built-in monster.mp4 green screen video file [2] will automatically play for the ones who don't have green screen. The program will remove the green screen from

the video file and replace it with a background file Bg.png. If you enter a digital in Figure 4, the web camera with the specified number will be used. If you only have one connected camera, please enter 0. You can press any key to exit the program.



Figure 5 video with green screenmonster.mp4

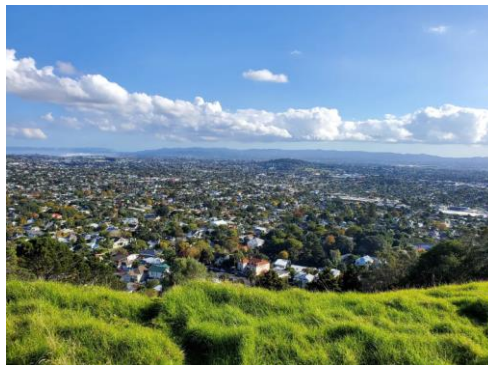


Figure 6 Background Image(Mt. Eden of New Zealand)



Figure 7 Result

4. Core principles



Figure 8 Original frame

The Figure 8 is an original frame from the camera. First, I call the `RenderGreenScreenMask(SRC, matMask)` method I wrote, converting the original frame 'src' to a black and white image 'matMask'. In matMask, the green parts are rendered black, and the rest are rendered white, as shown in Figure 9.

The main code of `RenderGreenScreenMask` is as follows [3] :

```
private unsafe void RenderGreenScreenMask(Mat src, Mat matMask)
{
    int rows = src.Rows;
    int cols = src.Cols;
    for (int x = 0; x < rows; x++)
    {
        Vec3b* srcRow = (Vec3b*)src.Ptr(x);
        byte* maskRow = (byte*)matMask.Ptr(x);
        for (int y = 0; y < cols; y++)
        {
            var pData = srcRow + y;
            byte blue = pData->Item0;
            byte green = pData->Item1;
            byte red = pData->Item2;
            byte max = Math.Max(red, Math.Max(blue, green));
            //if this pixel is some green, render the pixel with the same position on
matMask as black
            if (green == max && green > 30)
            {
                *(maskRow + y) = 0;
            }
            else
            {
                *(maskRow + y) = 255;//render as white
            }
        }
    }
}
```

To accelerate pixel access to the image, pointer operation is used. C# can use pointers to manipulate memory, which can greatly boost the performance. Due to the influence of environmental light, the points on the green screen may have different colors, so I use `green == max (blue,green,red)&& green > 30` to check whether a point has certain green, and 30 is the threshold value. The threshold value can adjust the recognition effect.



Figure 9 green removed

Then, `FindContoursAsArray()` is called to find some of the contours in Figure 9. In order to remove the small area interference caused by some folds or light problems on the green screen, for the contours found, it is necessary to delete the contours with smaller area and only retain the contours with large area. This contours filter can be easily done using the LINQ in C#, as follows:

```
var contoursExternalForeground = Cv2.FindContoursAsArray(matMask,
RetrievalModes.External, ContourApproximationModes.ApproxNone)
    .Select(c => new { contour = c, Area = (int)Cv2.ContourArea(c) })
    .Where(c => c.Area >= minBlockArea)
    .OrderByDescending(c => c.Area).Take(5).Select(c => c.contour);
```

The `minBlockArea` above represents "minimum allowable contour area".

Then, an empty black Mat is created, which is named `matMaskForeground`, and then render the large contours obtained above into the '`matMaskForeground`' and fill it with white inside, the code is as follows:

```
matMaskForeground.DrawContours(contoursExternalForeground, -1, new Scalar(255),
    thickness: -1);
```

The content of the picture corresponding to `matMaskForeground` is shown in Figure 10. In this way, the `matMaskForeground` contains only a certain large area of contours, and all other small areas of noise have been removed.



Figure 10 Find the largest contours and fill them with white

Then, cut out the large hollow areas of the arms, hands, shoulders, and neck shown in Figure 9. Therefore, do an "xor" operation on Figure 9 and Figure 10 to get a hollow area like Figure 11.



Figure 11 xor two images, and get the hollow area inside the body

The green light reflected in the screen into your glasses, or the small green on your clothes, can be identified as small hollowed areas. As can be seen in Figure 11, there are some small white areas at the lower right corner, so once again, FindContoursAsArray and DrawContours are used to remove the small areas in Figure 11. Then, combine Figure 11 and Figure 10, and you will get Figure 12, which is a picture with the white part as the body area and the black part as the green screen background.



Figure 12 Remove the small hollows and merge with the body mask

Then, Figure 12 is used as a mask for background transparency processing of the original frame image of Figure 8, and Figure 13 will be output, which is the background transparency image. The main codes are as follows:

```
public static void AddAlphaChannel(Mat src, Mat dst, Mat alpha)
{
    using (ResourceTracker t = new ResourceTracker())
    {
        //split is used for splitting the channels separately
        var bgr = t.T(Cv2.Split(src));
        var bgra = new[] { bgr[0], bgr[1], bgr[2], alpha };
        Cv2.Merge(bgra, dst);
    }
}
```

The 'src' is the original frame image, and 'dst' is the result of merging, and Alpha is the transparent mask shown in Figure 12.

Finally, we will draw the background transparent in Figure 13 onto our custom background image, and we will get the image in Figure 14. The core code is as follows:

```
public unsafe static void DrawOverlay(Mat bg, Mat overlay)
{
    int colsOverlay = overlay.Cols;
    int rowsOverlay = overlay.Rows;

    for (int i = 0; i < rowsOverlay; i++)
    {
        Vec3b* pBg = (Vec3b*)bg.Ptr(i);
        Vec4b* pOverlay = (Vec4b*)overlay.Ptr(i);
        for (int j = 0; j < colsOverlay; j++)
        {
            Vec3b* pointBg = pBg + j;
            Vec4b* pointOverlay = pOverlay + j;
            if (pointOverlay->Item3 != 0)
```



```

    {
        pointBg->Item0 = pointOverlay->Item0;
        pointBg->Item1 = pointOverlay->Item1;
        pointBg->Item2 = pointOverlay->Item2;
    }
}
}
}

```

The parameter 'bg' is the original frame image in Figure 8, and the overlay is the transparent background in Figure 13. After drawing by DrawOverlay, the content of 'bg' will turn into Figure 14, and then it can be output to the user interface.



Figure 13 A portrait with transparent background



Figure 14 Result

The code above is located in the ReplaceGreenScreenFilter.cs of the GreenScreenDemo project. Here's the trunk of the ReplaceGreenScreenFilter:

```

class ReplaceGreenScreenFilter
{
    private byte _greenScale = 30;

```

```

private double _minBlockPercent = 0.01;
private Mat _backgroundImage;
public void SetBackgroundImage(Mat backgroundImage)
{
    this._backgroundImage = backgroundImage;
}

private unsafe void RenderGreenScreenMask(Mat src, Mat matMask)
{
    int rows = src.Rows;
    int cols = src.Cols;
    for (int x = 0; x < rows; x++)
    {
        Vec3b* srcRow = (Vec3b*)src.Ptr(x);
        byte* maskRow = (byte*)matMask.Ptr(x);
        for (int y = 0; y < cols; y++)
        {
            var pData = srcRow + y;
            byte blue = pData->Item0;
            byte green = pData->Item1;
            byte red = pData->Item2;
            byte max = Math.Max(red, Math.Max(blue, green));
            if (green == max && green > this._greenScale)
            {
                *(maskRow + y) = 0;
            }
            else
            {
                *(maskRow + y) = 255;//render as white
            }
        }
    }
}

public void Apply(Mat src)
{
    using (ResourceTracker t = new ResourceTracker())
    {
        Size srcSize = src.Size();
        Mat matMask = t.NewMat(srcSize, MatType.CV_8UC1, new Scalar(0));
        RenderGreenScreenMask(src, matMask);
        //the area is by integer instead of double, so that it can improve the
        performance of comparison of areas
        int minBlockArea = (int)(srcSize.Width * srcSize.Height * this.MinBlockPercent);
    }
}

```

```

        var contoursExternalForeground = Cv2.FindContoursAsArray(matMask,
RetrievalModes.External, ContourApproximationModes.ApproxNone)
        .Select(c => new { contour = c, Area = (int)Cv2.ContourArea(c) })
        .Where(c => c.Area >= minBlockArea)
        .OrderByDescending(c => c.Area).Take(5).Select(c => c.contour);

//a new Mat used for rendering the selected Contours
var matMaskForeground = t.NewMat(srcSize, MatType.CV_8UC1, new
Scalar(0));

//thickness: -1 means filling the inner space
matMaskForeground.DrawContours(contoursExternalForeground, -1, new
Scalar(255),
    thickness: -1);
//matInternalHollow is the inner Hollow parts of body part.
var matInternalHollow = t.NewMat(srcSize, MatType.CV_8UC1, new Scalar(0));
Cv2.BitwiseXor(matMaskForeground, matMask, matInternalHollow);

int minHollowArea = (int)(minBlockArea * 0.01);//the lower size limitation of
InternalHollow is less than minBlockArea, because InternalHollows are smaller
//find the Contours of Internal Hollow
var contoursInternalHollow = Cv2.FindContoursAsArray(matInternalHollow,
RetrievalModes.External, ContourApproximationModes.ApproxNone)
    .Select(c => new { contour = c, Area = Cv2.ContourArea(c) })
    .Where(c => c.Area >= minHollowArea)
    .OrderByDescending(c => c.Area).Take(10).Select(c => c.contour);
//draw hollows
foreach (var c in contoursInternalHollow)
{
    matMaskForeground.FillConvexPoly(c, new Scalar(0));
}

var element = t.T(Cv2.GetStructuringElement(MorphShapes.Cross, new Size(3,
3)));

//smooth the edge of matMaskForeground
Cv2.MorphologyEx(matMaskForeground, matMaskForeground,
MorphTypes.Close,
    element, iterations: 6);

var foreground = t.NewMat(src.Size(), MatType.CV_8UC4, new Scalar(0));
ZackCVHelper.AddAlphaChannel(src, foreground, matMaskForeground);
//resize the _backgroundImage to the same size of src
Cv2.Resize(_backgroundImage, src, src.Size());
//draw foreground(people) on the backgroundimage
ZackCVHelper.DrawOverlay(src, foreground);

```

```

    }
}
}

```

5. Highlights

Limited by space, the basic knowledge of OpenCV is not explained here, but only some key technologies in the project and some matters needing attention during the use of OpenCVSharp are explained here. Since I have only used OpenCVSharp for a few days, please correct me if there are any problems.

1) Simplification of OpenCVSharp objects release

In OpenCVSharp, objects of classes such as Mat and MatExpr have unmanaged resources and need to be manually released by calling the Dispose() method. Worst of all, the +, -, *, and other operators create a new object each time, and these objects need to be disposed, or there will be a memory leak. But the code that these objects release looks very verbose.

For example, the following code is accessing Opencv in Python:

```

mat1 = np.empty([100,100])
mat3 = 255-mat1*0.8
mats1 = cv2.split(mat3)
mat4=cv2.merge(mats1[0],mats1[2],mats1[2])

```

In contrast, the equivalent code in C# is as verbose as the following

```

using (Mat mat1 = new Mat(new Size(100, 100), MatType.CV_8UC3))
using (Mat mat2 = mat1 * 0.8)
using (Mat mat3 = 255-mat2)
{
    Mat[] mats1 = mat3.Split();
    using (Mat mat4 = new Mat())
    {
        Cv2.Merge(new Mat[] { mats1[0], mats1[1], mats1[2] }, mat4);
    }
    foreach(var m in mats1)
    {
        m.Dispose();
    }
}

```

So I wrote a ResourceTracker class to manage OpenCV's resources. The T() method of the ResourceTracker class is used to add an OpenCV object to the tracing records and then return the object. The T() method is simply a matter of adding the wrapped object to the tracing records and returning it. The core code of the T() method is as follows:

```

public Mat T(Mat obj)
{
    if (obj == null)

```

```

    {
        return obj;
    }
    trackedObjects.Add(obj);
    return obj;
}

public Mat[] T(Mat[] objs)
{
    foreach (var obj in objs)
    {
        T(obj);
    }
    return objs;
}

```

The ResourceTracker implements the IDisposable interface, and when the Dispose() method is called, all resources tracked by the ResourceTracker are disposed. The T() method can trace an object or an array of objects. And this method NewMat() is like T(new Mat(...). Because the +, -, *, and other operators creates a new object each time, the resulting object at each step needs to be disposed, and they can be wrapped with T(). For example: t.T(255 - t.T(picMat * 0.8))

Therefore, the verbose C# code above can be reduced to the following:

```

using (ResourceTracker t = new ResourceTracker())
{
    Mat mat1 = t.NewMat(new Size(100, 100), MatType.CV_8UC3, new Scalar(0));
    Mat mat3 = t.T(255 - t.T(mat1 * 0.8));
    Mat[] mats1 = t.T(mat3.Split());
    Mat mat4 = t.NewMat();
    Cv2.Merge(new Mat[] { mats1[0], mats1[1], mats1[2] }, mat4);
}

```

After leaving the scope of using block of the ResourceTracker, the resources of all the objects managed by the ResourceTracker will be released.

I placed the ResourceTracker class in the NuGet package Zack.OpenCVSharp.Ext, so it can be installed on your project with the following NuGet command:

```
Install-Package Zack.OpenCVSharp.Ext
```

Source code repository: <https://github.com/yangzhongke/Zack.OpenCVSharp.Ext>

2) Efficient way to access data in Mat

OpenCVSharp provides many methods to access the data in Mat. After testing, I found that the At() method is the slowest, and GetGenericIndexer is also slow, because they are all conducted entirely through managed code, the performance is bound to be compromised. GetUnsafeGenericIndexer, which accesses memory directly, is much faster, but the fastest way is to use mat.ptr(x) and pointer, because it reads and writes Mat's memory directly. Please

mark the code using pointer with unsafe, and enable the "allow unsafe code" of the project. Because this way is to read and write memory directly, so please be sure to pay attention to your code, so as not to cause incorrect memory access or AccessViolation.

The reference code for this pointer approach refer to the above RenderGreenScreenMask() and DrawOverlay() methods. The 'where()' method of the np class in the open source project Zack.OpenCVSharp.Ext also demonstrates the use of a combination of C# generics, pointer manipulation, and lambda.

In OpenCVSharp, Vec4b, Vec3b, byte and other memory units with different byte lengths must be selected according to the channel number of Mat objects used. Improper use of Vec4b, Vec3b, byte and other memory units will not only affect performance, but also may cause data chaos. The most direct manifestation of data chaos is disordered picture display.

3) CameraPlayer

My software needs to collect images from the camera and display them on the user interface, and before displaying them on the interface, I also need to "cut the portrait and replace the background" on the image. At the beginning, I used AForge.NET to complete the image collection and display of the camera, but its performance was very low. Because I need to convert the Bitmap collected by AForge.NET into an OpenCVSharp Mat instance first, and then convert the Mat back to the Bitmap and display it on the user interface after the matting processing. Therefore, I directly used VideoCapture of OpenCVSharp to complete the camera image collection. Since the collected frame images are in the forms of Mat, the conversion between Mat and BitMap is eliminated, so the performance is improved significantly.

I created a CameraPlayer control to fetch data from the camera and display it on the interface. Meanwhile, I provide .NET Core and .NET Framework version of WinForm control, which can be used directly. In addition, SetFrameFilter(Action<Mat> frameFilterFunc) method is provided to allow setting a method, so as to use OpenCVSharp for processing before drawing the Mat of frame image to the user interface.

The image collection, image processing and image display in the CameraPlayer are performed in different threads and processed in parallel, so the performance is very high.

I have made the CameraPlayer control open source. Please check out the project documentation for usage.

Project repository: <https://github.com/yangzhongke/Zack.CameraLib>

When developing CameraPlayer, I found that if I did not set VideoCapture's attribute FourCC (video encoding), It would take 100ms to fetch each frame, while if I set the attribute FourCC to "MJPG", it would only take 50ms. I don't know if it's a camera-specific issue. So, if you read the image slower, you can try to use a different FourCC value.

4) Pay attention to the code that may lead to performance problems

When implementing the RenderGreenScreenMask() method, there is a process "calculate the maximum value of blue, green, and red". At the beginning, I used the LINQ extension to implement: `'new byte[] {blue, green, red}.Max();'` But I found that `'byte max1 = blue > green ? blue : green; byte max = max1 > red ? max1 : red;'` is more performant, and it reduces the processing time per frame by 50%.

Because LINQ operation involves the "create a collection object, put the data into the collection objects, and then get the data", performance will be slower than the conventional algorithm, in the ordinary scenario of data processing, the performance gap can be neglected,

especially when using LINQ to manipulate database, relative to the time consuming IO operations, this performance difference is negligible. But because this is used in two nested loops, the performance gap is huge.

Therefore, be careful with the operations that can lead to performance problems when manipulating images using OpenCVSharp

5) Initialization of Mat memory

When creating an empty Mat object, it is better to initialize the memory data of the Mat object, just as it is better to reset the memory space obtained by malloc with memset() in C language, so as not to cause the old residual data in memory to interfere with our operation. For example, the memory of instance created by 'new Mat(srcSize,MatType.CV_8UC1)' may reuse the memory of other objects that have been released previously. The data will be dirty. Unless your next step is to replace every item of Mat, please use the Scalar object parameters of Mat's constructor to initialize the memory, the code is like: new Mat(srcSize,MatType.CV_8UC1,new Scalar(0))

6. Future work

When I have time in the future, I may do the following work:

- 1) Improve the speed of reading images from the camera. Since my Logitech C920 camera is labeled as FPS=30, in theory, the time of fetching a frame is at 33ms, but I'm currently picking up a frame at 50ms, so I'm going to see if I can further speed up the process.
- 2) I consider adding features such as beauty, face thinning and skin brightening. The current image matting algorithm needs about 20ms to process a frame, while the speed of taking a frame from the camera is 50ms, so there is still 30ms extra time available for these beautification work.
- 3) AI algorithm will be used to achieve "port matting without green screen". It's hard to do this on my own. I found a very powerful open source project called MODNet, which is a library implemented by Python + Torch that uses neural networks for smart portrait recognition, including models that have been pre-trained. There is also a .NET ported version for Torch, so in theory this could be done.

7. Conclusion

When using OpenCVSharp, as long as you pay attention to the efficient access to memory introduced in this article and properly call functions, you can carry out image processing with very high performance. Therefore, the software I developed can achieve that each frame of image processing only needs about 20ms. With the help of the ResourceTracker class in the package Zack.OpenCVSharp.Ext developed by me, it is very easy to release resources of OpenCVSharp, and resources can be released in time without modifying the expression and code, so as to avoid memory leak.

Reference

- [1] "BMPCC4K – Green Screen," 26 7 2019. [Online]. Available: <https://australianimage.com.au/bmpcc4k-green-screen/>.

- [2] "Green Screen Monster 2 - monster attacks / feeds / eats," [Online]. Available: <https://www.youtube.com/watch?v=GEhPo1n15H4>.
- [3] B. Friederichs, "Processing an image to extract green-screen mask," [Online]. Available: <https://codereview.stackexchange.com/questions/184044/processing-an-image-to-extract-green-screen-mask>.