编程去除背景绿幕抠图，基于.NET+OpenCVSharp

摘要： 本文介绍了一种使用 OpenCVSharp 对摄像头中的绿幕视频进行实时“抠人像、替换背景”的方式，对于项目中的算法进行了分析。本文中给出了简化 OpenCVSharp 中 Mat、MatExpr 等托管资源释放的方法。本文还介绍了“高效摄像头播放控件”以及和 OpenCVSharp 的性能优化技术，包括高效读写 Mat 数据、如何避免效率低的代码等。

一、为什么自己开发实时抠图软件

由于工作的需要，我需要一个能够对于摄像头中的人像进行实时地“扣除背景、替换背景，并且把替换背景后的图片显示到窗口中”的功能。很多会议直播软件都有类似的功能，比如 Zoom、微软 Teams 等都有人像抠图功能，但是他们的这些功能都只局限于在它们的软件内使用。我又试用了几个软件，包括 XSplit Vcam、抖音直播伴侣、OBS，他们的功能都做的很优秀，包括很多都还有不需要绿幕的智能抠图的功能，非常强大，但是他们都无法满足我的特殊要求。所以我需要自己开发这样一款软件。

典型的人像抠图需要在被抠图的物体之后放上绿幕，然后再通过程序把绿幕扣除掉，这样人像就被保留下来了，再把抠出来的人像绘制到新的背景图上即可。很多影视制作都是用类似这样的原理制作出来的。如图 1 所示 [1]。

图 1

只要环境光线调整好了，通过绿幕进行抠图是非常准确的，不过这种方式的缺点就是对于场地的布置要求非常高。所以现在流行“无绿幕抠图”的功能，也就是用人工智能的方法智能识别前景人像和背景，然后智能的把前景人像识别出来。XSplit Vcam 有这个功能，而且可以把抠图的结果再模拟成一个虚拟摄像头进行输出，属于民用领域中比较强悍的一款软件，但是如果背景比较复杂的话，XSplit Vcam 移除背景的效果仍然不理想。我个人在计算机视觉方面，特别是结合人工智能进行图像的智能处理方面，研究很浅，我不认为在时间有

限的情况下，能写出来一个比 Vcam 还要强大的软件，因此我决定仍然用传统的绿幕形式来实现我想要的功能，毕竟只要花几十块钱买一块绿幕即可。

在开始讲解实现代码之前，先展示一下软件的运行效果。图 2 是相机采集的原始图像，可以看到背后是一张绿幕，而 图 3 则是软件运行后的效果，而且是实时抠图的，目前可以做到大约 20FPS（一秒钟约 20 帧）。

图 2 没有抠绿幕

图 3 抠人像、替换背景

二、软件架构

软件使用了 OpenCV，它是一个非常成熟、功能丰富的计算机视觉库。OpenCV 支持 C/C++、Python、.NET、Java 等主流的编程语言。在互联网上，使用 Python 进行 OpenCV 开发的资料最多。由于个人不是很喜欢 Python 的语法，所以这个软件我使用 C#语言在.NET 5 平台上进行开发。由于 OpenCV 在各个编程语言上用法大同小异，因此这里用 C#实现的代码改用其他编程语言也非常容易。

.NET 平台下，有两个 OpenCV 的绑定库：OpenCVSharp 和 Emgu CV。由于 OpenCVSharp 没有商业使用限制，因此我这里使用 OpenCVSharp。不过，即使您使用的是 Emgu CV，这篇文章里的代码也是简单修改后就可以应用到 Emgu CV 中。

三、如何获得源代码

由于抠绿幕替换背景的功能只是我的软件的一个模块，整个软件暂时不方便开源，所以

我把抠绿幕替换背景这部分核心代码功能剥离到一个单独的开源项目中。

项目开源地址：https://github.com/yangzhongke/Zack.OpenCVSharp.Ext

代码中的“GreenScreenRemovalDemo.cs”就是最核心的代码，也可以在项目页面底部的【GreenScreenRemovalDemo】中下载各个操作系统下的可执行文件，其中的GreenScreenRemovalDemo就是主程序。

以Windows为例，运行GreenScreenRemovalDemo.exe，就会出现如图 4 所示的控制台

图 4 选择用演示视频还是摄像头

如果输入v，就会自动播放一个内置的monster.mp4绿幕视频文件[2]，供没有绿幕环境的朋友进行体验，程序会从视频文件中将绿幕剔除掉替换为自定义背景文件bg.png。如果在图 4 这一步输入数字，则会从指定编号的网络摄像头中读取画面进行抠图，如果您的计算机中只有一个摄像头，那么输入0即可。体验完毕，在图形窗口内按任意键就会退出程序。

如下的图 5、图 6和图 7分辨就是绿幕视频、背景图以及合成图。

图 5 绿幕视频monster.mp4

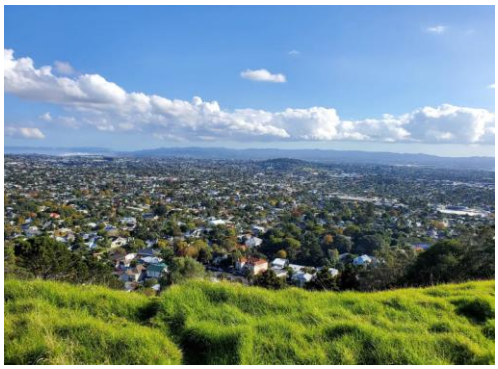

图 6 背景图bg.png(新西兰的伊甸山)

图 7 替换背景后的合成图

四、核心原理

图 8 原始帧图片

图 8 是从摄像头获取的一帧原始图片。首先，调用我编写的 RenderGreenScreenMask(src, matMask)方法，把原始帧 src 转换为一张黑白图 matMask 做为遮罩。matMast 中，绿色部分渲染为黑色，其他部分渲染为白色，如图 9。

RenderGreenScreenMask 方法的主要代码如下 [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
                }
            }
        }
}
```

为了加速图片的像素点访问，这里使用指针来操作。C#中可以使用指针操作内存，这样可以大大加速程序的运行效率。因为环境光照的影响，背景绿幕中的各个点颜色并不完全相同，所以这里使用像素点的 green == max (blue,green,red)&& green > 30 是否为 true 来判断一个点是否是绿色，30 是一个阈值，可以根据情况来调节识别效果，这个阈值选的越大，被认为是绿色的范围越窄。

图 9 去掉绿色

接下来，调用 OpenCV 的 FindContoursAsArray()方法找到 图 9 中的若干个轮廓信息。为了去掉一些绿幕中的褶皱或者光线问题造成的小面积干扰，对于找到的轮廓信息，需要删除掉面积较小的轮廓，只保留面积较大的轮廓。使用 C#中的 LINQ 操作可以轻松的完成这个筛选，代码如下：

```
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);
```

这里的 minBlockArea 代表设定的一个“最小允许轮廓区域的面积”。

接下来新建一个空的黑色 Mat，名字为 matMaskForeground，然后把上面得到的大轮廓区域绘制到这个 matMaskForeground 中，并且内部填充为白色，代码如下：

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

```
                    thickness: -1);
```

matMaskForeground 对应的图片内容如图 10。这样 matMaskForeground 中就只包含若干大面积轮廓了，其他小面积的干扰都被排除了。

图 10 找到最大几个闭合区域，然后填充为白色

接下来，要把图 9 中的手臂、手、肩膀和脖子形成的那些大的镂空区域抠出来。因此把图 9 和 图 10 做“异或”操作，得到 图 11 这样的镂空区域。

图 11 前两张图片做异或操作，得到身体内部的镂空区域

因为眼镜中反射的屏幕中的绿光、或者衣服上的小的绿色可能会被识别为小的镂空区域，，可以看到 图 11 的右下角就有一些小白色区域，因此再次使用 FindContoursAsArray、DrawContours 把 图 11 中的小面积的区域排除掉。然后再把排除掉小面积轮廓的图 11 和图 10 做合并操作，就得到图 12，就是一个白色部分为身体区域，而黑色部分为绿幕背景的的图片。

图 12 把小镂空区域去掉，并和身体遮罩做合并

接下来使用图 12 做为遮罩对原始帧图像 图 8 进行背景透明处理，得到图 13， 这样的图片就是背景透明的图片了。主要代码如下：

```
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);
    }
}
```

其中 src 是原始帧图像，dst 是合并结果，而 alpha 则是图 12 把小镂空区域去掉，并和身体遮罩做合并这个透明遮罩。

最后把背景透明的图 13 绘制到我们自定义的背景图上，就得到替换为背景图的图 14 了。核心代码如下：

```
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;
                }
            }
        }
    }
```

其中参数 bg 就是原始帧图像图 8，而 overlay 则是背景透明的图 13，经过 DrawOverlay 方法绘制后，bg 的内容就变成了图 14，然后就可以输出到界面上了。

图 13 背景透明图

图 14 最终结果

上面讲述的核心代码就位于 GreenScreenRemovalDemo 项目的 ReplaceGreenScreenFilter 类中。下面列出 ReplaceGreenScreenFilter 最主干的代码：

```
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 comparision 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);
            }
        }
```

```
}
```

五、重要技术

受限于篇幅，这里不讲解 OpenCV 的基础知识，这里只讲解项目中的一些重点技术以及 OpenCVSharp 使用过程中的一些需要注意的事项。由于我也是刚接触 OpenCVSharp 几天时间，所以如果存在有问题的地方，请各位指正。

1) 简化 OpenCVSharp 对象的释放

在 OpenCVSharp 中，Mat 和 MatExpr 等类的对象拥有非托管资源，因此需要调用 Dispose()方法手动释放。更糟糕的是，+、-、*等运算符每次都会创建一个新的对象，这些对象都需要释放，否则就会有内存泄露。但是这些对象释放的代码看起来非常啰嗦。

假设有如下 Python 中访问 opencv 的代码：

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

而在 C#中同样的代码则像下面这样啰嗦：

```
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();
    }
}
```

因此我创建了一个 ResourceTracker 类用来管理 OpenCV 的资源。ResourceTracker 类的 T()方法用于把 OpenCV 对象加入跟踪记录。T()方法的实现很简单，就是把被包裹的对象加入跟踪记录，然后再把对象返回。T()方法的核心代码如下：

```
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;
}
```

ResourceTracker 实现了 IDisposable 接口，当 ResourceTracker 类的 Dispose()方法被调用后，ResourceTracker 跟踪的所有资源都会被释放。T()方法可以跟踪一个对象或者一个对象数组。而 NewMat() 这个方法是 T(new Mat(...)) 的一个简化。因为+、-、*等运算符每次都会创建一个新的对象，所以每步运算得到的对象都需要释放，他们可以使用 T()进行包裹。例如：t.T(255 - t.T(picMat * 0.8))

因此，上面的啰嗦的 C#代码可以简化成如下的样子：

```
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);
}
```

在离开 ResourceTracker 的 using 代码块之后，所有 ResourceTracker 对象管理的 Mat、MatExpr 等对象的资源都会被释放。

这个 ResourceTracker 类我放到了 Zack.OpenCVSharp.Ext 这个 NuGet 包中，可以通过如下 NuGet 命令安装：

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

项目的源代码地址：https://github.com/yangzhongke/Zack.OpenCVSharp.Ext

2) 访问 Mat 中数据的高效方式

OpenCVSharp 中提供了很多访问 Mat 中数据的方法，经过测试，我发现，At()方式最慢，GetGenericIndexer 也很慢，因为他们都是完全通过托管代码的方式进行的，性能必然打折扣。而直接访问内存的 GetUnsafeGenericIndexer 方式快了很多，但是最快的方式还是使用 mat.Ptr(x)并使用指针这种方式速度最快，因为这种方式直接通过指针读写 Mat 的内存。使用这种方式的方法需要标记为 unsafe，并且项目要启用“允许不安全代码”。由于这种方式是直接读写内存，所以一定要注意你的代码，以免造成不正确的内存访问或者 AccessViolation，对指针操作不熟悉的读者，可以阅读我出版的图书《零基础趣学 C 语言》（作者：杨中科，人民邮电出版社），因为 C#中指针操作和 C 语言几乎一模一样。

这种指针方式的参考代码请参考上面的 RenderGreenScreenMask()、DrawOverlay()两个方法，

Zack.OpenCVSharp.Ext 这个开源项目中 np 类的 where 方法还演示了 C#泛型、指针操作以及 lambda 的结合使用。

OpenCVSharp 中，Vec4b、Vec3b、byte 等代表不同字节长度的内存单元，一定要根据使用的 Mat 对象的通道数等来选择使用 Vec4b、Vec3b、byte 等，使用不当不仅会影响性能，而且还可能会造成数据混乱，数据混乱的最直接的表现就是图片显示错乱、花屏。

3) CameraPlayer

我的软件需要从摄像头采集图像，并且显示到界面上，而且在显示到界面上之前，还要对图像进行“抠人像、替换背景”的操作。在最开始的时候，我使用 AForge.NET 完成摄像头的图像采集和显示，不过性能非常低。因为需要先把 AForge.NET 采集到的 Bitmap 转换为 OpenCVSharp 的 Mat，抠图处理完成后再把 Mat 转换回 Bitmap，显示到界面上。所以我就直接使用 OpenCVSharp 的 VideoCapture 类来完成摄像头图像的采集，由于它采集到的帧图像直接用 Mat 表示，省去了转换环节，速度得到了很大的提升。

我把从摄像头取数据以及显示到界面上的操作封装了一个 CameraPlayer 控件中，同时提供了.NET Core 和.NET Framework 版的 WinForm 控件，可以直接拿来用，而且提供了 SetFrameFilter(Action<Mat> frameFilterFunc)方法来允许设定一个委托，从而在把帧图像的 Mat 绘制到界面前使用 OpenCVSharp 进行处理。

CameraPlayer 控件中图像采集、图像的处理和图像的显示是由不同线程负责，各自并行处理，所以性能非常高。

我把这个 CameraPlayer 控件开源了，具体用法请参考项目的文档。

项目地址：https://github.com/yangzhongke/Zack.CameraLib

在开发 CameraPlayer 的时候，我发现如果不设定 VideoCapture 的 FourCC 属性（也就是视频的编码），取一帧需要 100ms，而把 FourCC 属性设置为"MJPG"之后，取一帧只要 50ms。我不知道这是否和摄像头相关。因此，如果你因为 FourCC 属性设置为"MJPG"之后，读取图像的速度反而变慢了，可以尝试修改一个不同的 FourCC 值。

4) 谨慎使用可能造成性能问题的玩意儿

在实现 RenderGreenScreenMask()这个方法的时候，其中有一步是用来“取 blue、green、red 三个值中的最大值”，最开始的时候，我使用.NET 中的 LINQ 扩展方法实现 new byte[]{blue,green,red}.Max(); 但是发现改成 byte max1 = blue > green ? blue : green; byte max = max1>red?max1:red;这种简单的方法计算之后，每一帧的处理时间减少了 50%。

由于 LINQ 操作涉及到“创建集合对象、把数据放入集合对象、获取数据”这样的过程，速度会比常规算法慢一些，在普通的数据处理中这点性能差距可以忽略不计，特别是在使用 LINQ 对数据库等进行操作的时候，相对于耗时的 IO 操作来讲，这点性能差别更是可以忽略不计。但是由于这里是在双层循环中使用，而且执行的操作的速度非常快的内存读写，所以就把性能差距放大了。

因此，在使用 OpenCVSharp 对图像进行处理的时候，要谨慎使用这些可能会造成性能问题的高级玩意儿。

5) Mat 内存的初始化

在创建空的 Mat 对象的时候，最好初始化 Mat 对象的内存数据，就像在 C 语言中对于 malloc 拿到的内存空间最好用 memset 重置一样，以免造成内存中旧的残留数据干扰我们的操作。比如 new Mat(srcSize,MatType.CV_8UC1)这样创建的空白 Mat 中的内存可能是复用之前被释放的其他对象的内存，数据是脏的，除非你的下一步操作是把 Mat 的每一位都重

新填充，否则请使用 Mat 构造函数的 Scalar 类型的参数来初始化内存，参考代码如下：new Mat(srcSize,MatType.CV_8UC1,new Scalar(0))

六、未来工作

在以后有时间的时候，我可能会做如下这些工作。

1) 提升从摄像头取一帧的速度。因为我目前用的摄像头“罗技 C920”标称的是 FPS=30，所以理论上来讲，取一帧的速度是 33ms，而目前我取一帧的速度是 50ms，我要研究一下是否能进一步提升取一帧图像的速度。
2) 考虑增加美颜、瘦脸、亮肤等功能，目前的人像抠图算法处理一帧需要大约 20ms，而从摄像头取一帧的速度是 50ms，因此还有 30ms 的额外时间可以用来做这些美化工作。
3) 用人工智能算法实现“无绿幕抠人像、去除背景”。完全自己实现这个无疑是比较难的。我发现一个很强大的开源项目 MODNet，它是一个 python+torch 实现的使用神经网络做智能人像识别的库，包含已经训练完成模型。而 torch 也有对应的.NET 移植版，所以理论上这是可以做到的。

七、结论

使用 OpenCVSharp 的时候，只要注意使用本文中介绍的高效访问内存的方式，并且合理调用相关的函数，可以非常高性能的进行图像的处理，因此我开发的软件可以做到每一帧图像处理仅需大约 20ms。借助于我开发的 Zack.OpenCVSharp.Ext 这个包中的 ResourceTracker 类，可以让 OpenCVSharp 中的资源释放变得非常简单，在几乎不用修改表达式、代码的基础上，让资源能够及时得到释放，避免内存泄漏。

参考资料

[1] “BMPCC4K – Green Screen,” 26 7 2019. [联机]. Available: https://australianimage.com.au/bmpcc4k-green-screen/.

[2] “Green Screen Monster 2 - monster attacks / feeds / eats,” [联机]. Available: https://www.youtube.com/watch?v=GEhPo1n15H4.

[3] B. Friederichs, “Processing an image to extract green-screen mask,” [联机]. Available: https://codereview.stackexchange.com/questions/184044/processing-an-image-to-extract-green-screen-mask.