应用程序开发报告

写在前面

名称约定

为了对应设计模式 MVVM 中的各个部分，报告中有些情况下将图形用户界面(GUI)称为视图(View)，将底层逻辑称为模型(Model)。此外，在 C#中一般将函数(Function)称为方法(Method).

代码

报告中大部分都是 C#和 XAML 风格的伪代码，不能直接应用与项目，仅供说明。

1 概述

1.1技术路线

此项目使用 WPF (Windows Presentation Foundation) 开发。WPF 运行于微软.Net 平台，供了统一的编程模型、语言和框架，可以使用任意一种.Net 编程语言开发，此项目中使用 C#进行后台开发，使用标记语言 XAML 进行界面设计。

WPF 采用了 MVVM (Model-View-ViewModel) 的设计模式，相对于传统开发模式，该模式最大特征为引入了 ViewModel，实现了图形界面与业务逻辑分离，在之后的内容中将以此项目为例介绍这一设计模式。

1.2项目框架

项目按照功能横向分为 3 个部分：设备控制，图像处理，数据储存。

其中设备控制功能由相机生产方的动态链接库 GxIAPINET.dll 提供，但采集回调函数需要自行编写。

项目按照层次纵向分为 3 个部分，即 MVVM 模式中的 Model, View, ViewModel.

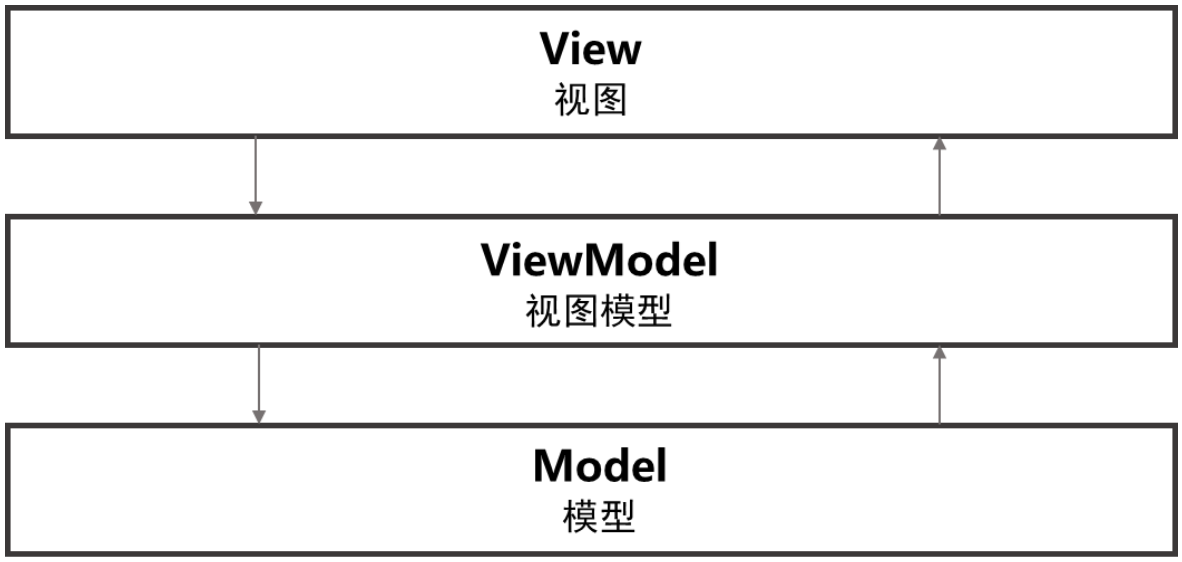

图 MVVM 模式层次

View（视图）负责与用户的交互；ViewModel（视图模型）负责 View 和 Model 间的通信；Model（模型）即此项目中设备控制，图像处理，数据储存这 3 个功能。

其中 View 使用 XAML 编写，ViewModel 和 Model 使用 C#编写。

2 模型 (Model)

2.1设备控制流程

下面为简化版的控制流程图。

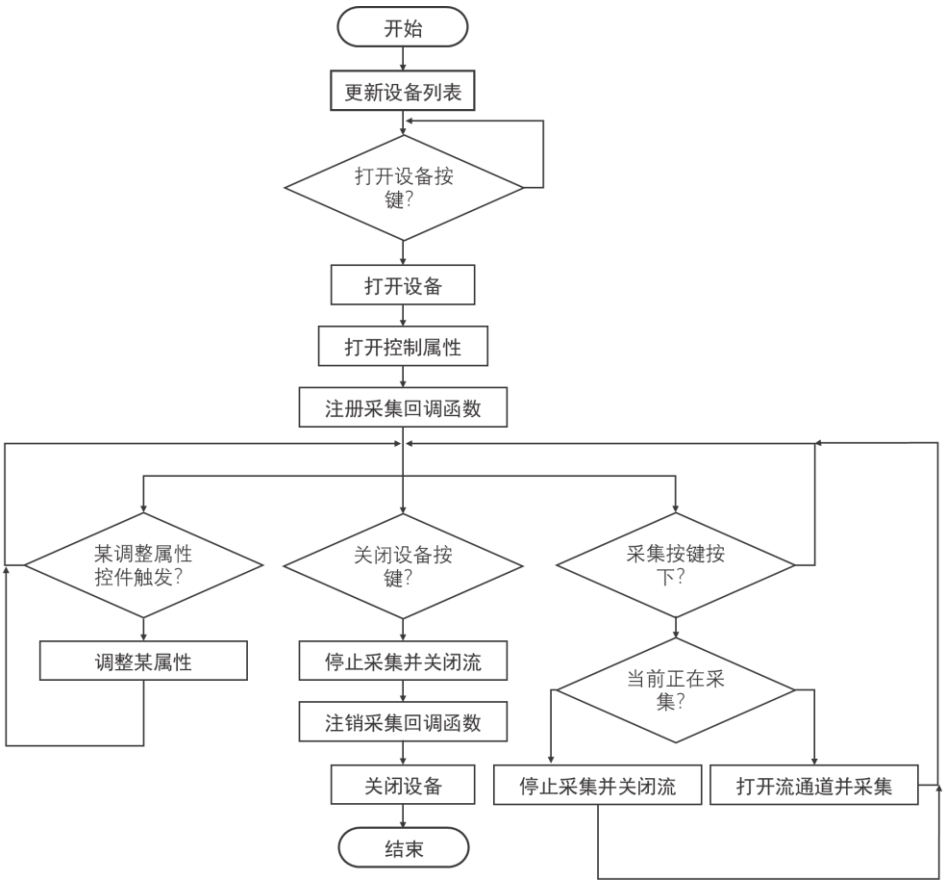

图 设备控制流程

2.2采集回调函数

相机生产方不提供采集回调函数的具体内容，由用户根据需要编写，声明如下：

```
void AcqusitionCallback(object sender,IFrameData frameData);
```

当采集开始后，每采集到一帧画面，程序自动调用此函数一次。函数的第二个参数即采集到的帧数据，在该函数中可以对帧数据进行图像处理并输出到显示界面。

2.3图像处理

图像处理在采集回调函数中进行，通过以下方法获得图像数据的指针：

```
C#:
void AcquisitionCallBack(object sender, IFrameData framedata)
{
    IntPtr buffer = frameData.GetBuffer();
    unsafe
    {
        byte* bufferPtr = (byte*)buffer.ToPointer();
        ......//下面进行图像处理
    }
    ......//下面进行图像输出
```

```
11. }
```

在 unsafe 代码中可以使用 C 语言的方法操作指针，通过循环遍历像素，可对每个像素进行操作。

2.4图像输出

画面输出同样在采集回调函数中，在图像处理之后进行。使用一个可写位图类型 WriteableBitmap 储存当前需要显示的图。WriteableBitmap 类完成一次画面更新的流程为：（1）更新 Backbuffer.（2）锁定图像。（3）指定更新矩形框范围。（4）解锁图像。在这一流程完成后，程序自动将 BackBuffer 中的数据加载到用于显示的 ForeBuffer 中，然后可以进行下一次的画面更新。示例程序：

```
1.  C#:
2.  void AcquisitionCallBack(object sender, IFrameData framedata)
3.  {
4.      ........//以上为图像处理
5.      int length = width * height *3;
6.      IntPtr backBuffer = imageForDisp.BackBuffer;     //获得 backBuffer
7.      CopyMemory(backBuffer,buffer, length);        //复制 buffer 数据到 backBuffer
8.      image.Lock();                                    //锁定图像
9.      image.AddDirtyRect(new Int32Rect(0, 0, width, height)); //指定更新范围为全图
10.     image.Unlock();                                          //解锁图像
11. }
```

目前我们已经知道如何将数据加载到用于显示的 WriteableBitmap 中，那么如何将它显示到屏幕上呢，在之后继续讨论。

3 视图（View）

3.1总览

应用程序的视图采用主窗口+多页面导航的框架。由一个主窗口 Main Window 和 3 个页面 Device Page, Data Page, About Page 构成。

图 总览

红色框中是导航栏，蓝色框中是页面。点击不同导航栏后主窗口将显示不同页面。图中为设备页面。

视图使用标记语言 XAML 编写，其语法与 HTML 有一定相似之处。相比于 C#，使用 XAML 构建视图非常方便，它的使用方式不进行介绍，因为构建视图的重点不在代码编写，而在于设计。

3.2设计

接下来将从平面设计的几点原则介绍此项目视图的设计思想。

对齐：

视图中的各个元素应与页面上某个内容存在某种视觉联系。即使这些项并不靠近，但它们属于同一组——以一条看不见的线将彼此连在一起。

图 对齐示例 1

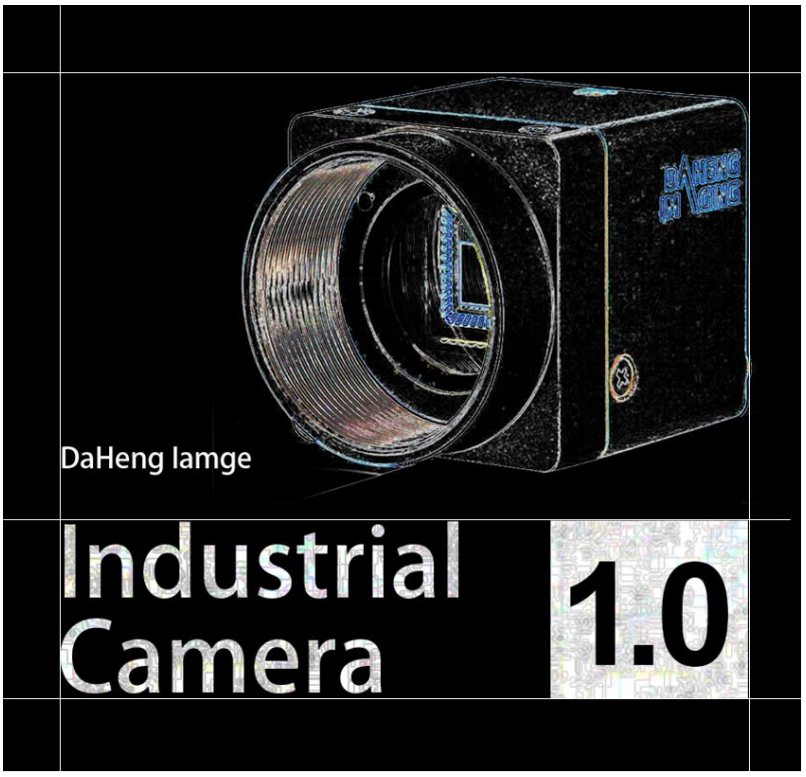

图 对齐示例 2

亲密：

把相关项组织在一起：物理位置的接近就意味着存在关联，运用亲密性原则可以使页面变得有条理、阅读逻辑清晰、页面留白变得有组织感。

图 关联的项距离更近

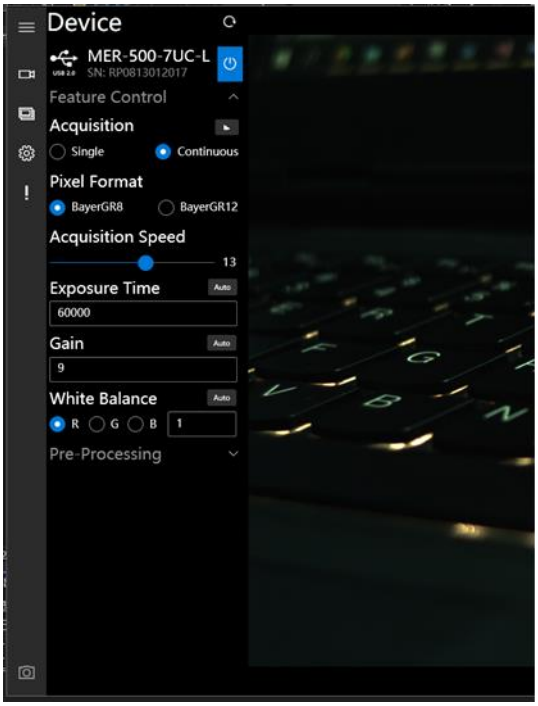

图 设备页与导航栏及图片过近

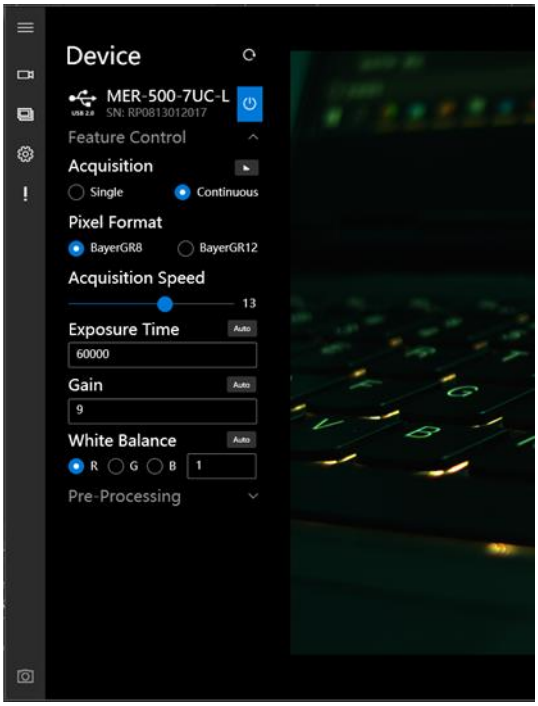

图 正确示范

对比：

页面上的不同元素之间的对比效果达到吸引读者的效果。如果两项完全不同，那就让其截然不同。运用对比原则的效果：组织信息；清晰层级；指引读者；制造焦点。例如在界面中，面积最大的部分是用户最关心的内容：相机的输出画面。

图 通过字体大小和颜色深度制造对比

重复：

设计的某些方面需要在整个作品中重复来体现风格一致性。比如：颜色、字体、控件元素。运用重复原则有助于组织信息，利于单独的部分统一起来；体现专业和权威性。

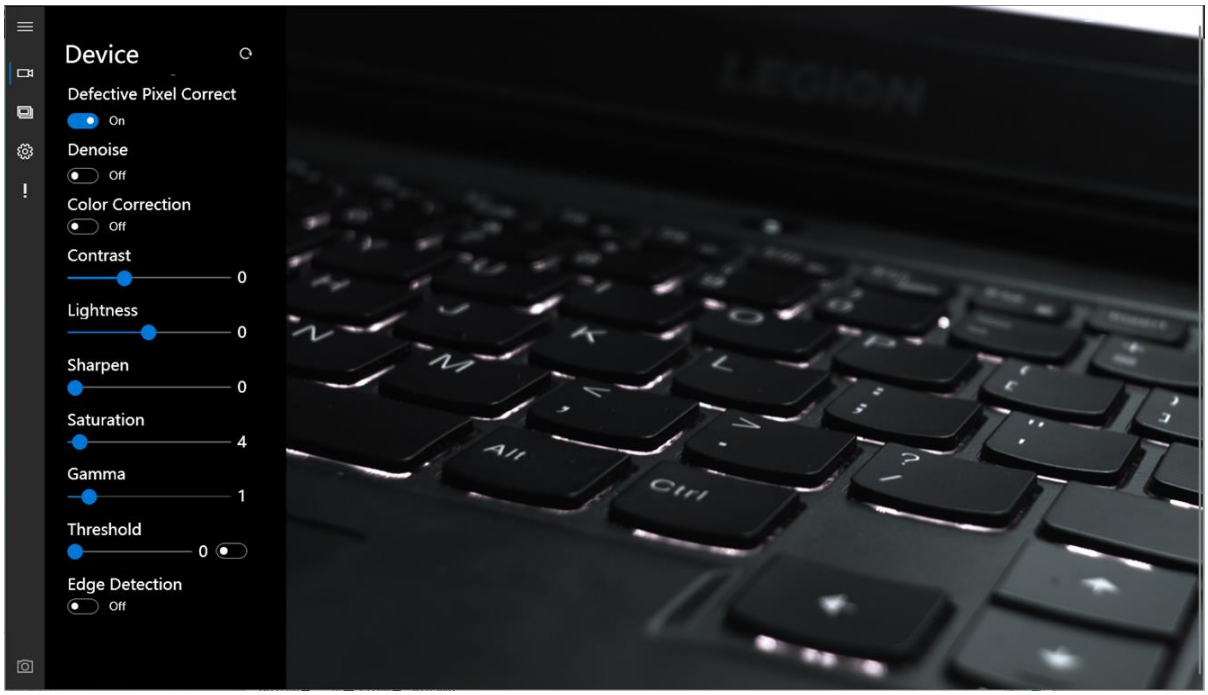

图 通过重复统一视图风格

4 视图模型（ViewModel）

视图模型负责视图层和模型层的通信。

4.1C#属性

在介绍视图模型之前，需要先了解前置知识属性。在 C#中，属性(Property)是类(Class) 中的一个数据项的成员。
它的访问方式和类中的一个普通成员变量相同：

```
C#:
int MyValue1 = 3;
int MyValue2;
myClass.MyProperty = MyValue1;     //给属性赋值
myClass.MyVar = MyValue1;          //给普通成员变量赋值
MyValue2 = myClass.MyProperty;     //读取属性
MyValue2 = myClass.MyVar;          //读取普通成员变量
```

属性的定义方式更接近于方法：

```
C#:
class MyClass
{
    private int MyValue;     //定义一个成员变量
    public int MyProperty    //定义一个属性
    {
        get
        {
```

```
            return MyValue      //这里返回读取属性时得到的值
        }
        set                     //在这里形参(value)被隐式给出
        {
            MyValue = value  //这里定义为给属性赋值时的操作
        }
    }
    public int GetValue()    //定义一个成员方法
    {
        return value;
    }
    public int SetValue(int value)  //定义另一个成员方法
    {
        MyValue = value;
    }
}
```

以上代码中，属性的功能与其之后的两个方法的功能相同。此外，可以通过只设置 set 或 get 访问器来将某一成员变量设置为只写或只读。

总的来说：属性的功能与用于访问的函数完全相同，访问它的语法与成员变量相同。实际上这是一个语法糖衣。

属性并不一定是只用于访问类中的某一成员变量，下面通过一个例子说明属性的另一种应用场景：现在我们有一个灯的实例 Light，它有两个成员方法用于开关，一个成员方法用于获得灯的状态，我们想通过属性简化对灯的操作。

定义属性

```
C#:
public bool IsLightOn    //定义灯是否打开的属性
{
    get
    {
        if(Light.GetStatus() == "ON")
            return true;
        if(Light.GetStatus() == "OFF")
            return false;
    }
    set
    {
        if(value)
            Light.TurnOn();
        else
            Light.TurnOff();
    }
}
```

访问属性

```
C#:
IsLightOn = true;          //打开灯
IsLightOn = false;         //关闭灯
bool status = IsLightOn;   //获得灯的状态
```

等价于

```
C#:
Light.TurnOn();
Light.TurnOff();
if(Light.GetStatus() == "ON")
    bool status = true;
if(Light.GetStatus() == "OFF")
    bool status = false;
```

通过这种方式，我们可以把一系列繁琐的函数调用和状态判断等简化为赋值和取值操作。它非常重要，在此项目的很多地方的属性都是这种用法。

4.2属性绑定

属性除了可以简化操作和限制访问外，在 WPF 中，它还可以通过绑定，将模型和视图连接起来。需要注意，绑定是 WPF 中特有的功能。具体方法为：

- 在视图模型中定义属性，属性中为模型的各种操作或数据。
- 在视图中设置绑定，将控件(如按键)的某一个状态（如开关状态）与视图模型中的属性绑定。
- 设置数据上下文(Data Context)，让视图得知绑定的属性来源于哪里。

如果把绑定比作数据的桥梁，那么它的两端分别是源(Source)和目标(Target)。数据从哪里来哪里就是源，到哪里去哪里就是目标。一般情况下，绑定的源是模型层的对象，绑定的目标是视图层的控件对象。这样数据就会源源不断的通过绑定送达视图，被视图层展现，这就完成了模型中的数据驱动视图的过程。

现在终于可以解决前面的问题：如何将 WriteableBitmap 显示到屏幕上：

在视图模型定义 WriteableBitmap 实例与其属性

```
C#:
public class Device
{
    ......//其他事件
    public event PropertyChangedEventHandler PropertyChanged;  //声明属性变化事件
    private WriteableBitmap imageForDisp;   //定义私有成员变量
    ......//其他变量
    public WriteableBitmap ImageForDisp   //定义公共属性用于访问上述变量
    {
        get
        {
            return imageForDisp;
```

```
        }
        set
        {
            imageForDisp= value;
            if (PropertyChanged != null)    //属性改变时触发属性变化事件
        PropertyChanged.Invoke(this, new PropertyChangedEventArgs("ImageForDisp"));
        }
    }
    ......//其他属性
    ......//其他方法
}
```

说明：当模型的值发生变化（即调用了一次 set 访问器）时，将会触发属性变化事件，自动通知视图更新，也就是说当我们设置好绑定以后，数据变化时程序将自动更新视图，不再需要手动更新。这里涉及到 C#中的委托和事件，这里不详细说明；当用户的操作使视图中的元素改变时，程序将自动通过属性的 set 访问器，改变模型的数据。

在视图绑定属性

```
XAML:
<Image Source = "{Binding ImageForDisp}"/>
```

说明：定义了一个用于显示的图片控件，Source 是图片的源，通过绑定将它的源设置为 ImageForDisp 属性。

设置数据上下文

```
C#:
Device device = new Device();
view.DataContext = device;
```

说明：将视图的数据上下文设置为一个 Device 类的实例 device，即先前定义的视图模型，让视图知道从 device 中去获取 ImageForDisp 属性，而不是其他地方。

另一个例子：绑定设备连接

定义属性

```
public bool IsConnected
{
    set
    {
        if(value)
        {
            instance = IGXFactory.GetInstance().OpenDeviceBySN(SN, GX_ACCESS_MODE.GX_ACCESS_CONTROL); //打开设备
            stream = instance.OpenStream(0);  //打开流通道
```

```
            featureControl = instance.GetRemoteFeatureControl(); //打开参数设置
            stream.RegisterCaptureCallback(instance,AcquisitionHandler);//注册回调
            imageProcessConfig=instance.CreateImageProcessConfig();//打开图像处理
            image = new WriteableBitmap(2592,1944,96,96,PixelFormats.Rgb24,null);
        }
        else
        {
            AcquisitionStop();              //停止采集
            stream.Close();                 //关闭流通道
            imageProcessConfig.Destory();   //销毁图像处理设置
            instance.Close();               //关闭设备
            //重置成员变量
            stream = null;
            instance = null;
            imageProcessConfig = null;
            featureControl = null;
            backBuffer = IntPtr.Zero;
            image = null;
        }
    }
    get { return instance != null; }
}
```

说明：这里体现了在前一节中介绍的属性的优点：把一系列繁琐的函数调用等操作简化。只需要对 IsConnected 赋值，即可完成上述一系列流程。这里的 set 访问器里不再需要属性变化事件，因为所有设备连接操作都由视图方发起，不用通知自己更新。

设置绑定

```
XAML:
<ToggleButton Name="IsConnectedButton" IsChecked="{Binding IsConnected}" />
```

当连接按键按下后，控件访问 set 访问器并执行其中的程序，完成打开设备需要的所有流程，关闭设备同理。

5 其他问题

5.1限制输入

在开发程序是，需要注意到用户可能会有不合法的输入，例如在设备未连接时调整设备的参数，或在设备采集时调整采集模式。通过绑定可以解决这一问题。除了前面介绍的视图与视图模型的属性绑定之外，在视图的各个元素之间也可以进行绑定。

例如，在开启自动曝光时，禁用手动输入曝光时间：

```
XAML:
<ToggleButton Name ="IsExposureAutoButton" IsChecked ={"Binding IsExposureAuto"} />
<TextBox Name = "ExposureTime" IsEnable = {"Binding IsChecked, ElementName =
IsExposureAutoButton ", Converter={StaticResource InvBoolConverter}}/>
```

首先定义一个名为 IsExposureAutoButton 翻转按键用于控制是否开启自动曝光，其是否按下绑定视图模型中的 IsExposureAuto 属性，这里与之前提到的绑定方式一致。

重点在第三行：定义名为 ExposureTime 的文本框用于在手动模式下输入曝光时间，它的 IsEnable（是否能够访问）属性通过一个非运算转换器与自动曝光按键的 IsChecked（是否按下）属性绑定（不需要视图模型层，因为这是视图元素之间的绑定）。当自动曝光按键按下时，曝光时间文本框变成不可访问状态，如下图：

图 自动曝光前后

5.2多设备控制

WPF 为我们提供了列表控件 ItemsControl，它可以直接以 C#中的 List 类作为源，List 中有多少元素，ItemsControl 就会在视图中生成多少设备控制界面。它的具体用法非常复杂，涉及到模板和其控件的绑定，这里省略。

5.3线程冲突

在 WPF 程序中，视图层更新和模型层的线程彼此独立，当这两个层都访问同一实例时会产生线程冲突。让我们回到 2.4 节：图像输出。

图像输出程序：

```
C#:
int length = width * height *3;
IntPtr backBuffer = imageForDisp.BackBuffer; //获得 backBuffer
CopyMemory(backBuffer,buffer, length);       //复制 buffer 数据到 backBuffer
imageForDisp.Lock();                         //锁定图像
imageForDisp.AddDirtyRect(new Int32Rect(0, 0, width, height)); //指定更新范围为全图
imageForDisp.Unlock();                       //解锁图像
```

这个程序独立地在模型层中运行没有任何问题，但是当绑定完成后，会产生异常提示：调用线程无法访问此对象，因为另一个线程拥有该对象。

其原因为：采集时该采集回调函数会不断被调用，不断访问 imageForDisp，而视图层也通过绑定在始终监听 imageForDisp 是否发生改变，此时两个线程同时访问了同一实例。

改进后的图像输出程序：

```
C#:
int length = width * height *3;
IntPtr backBuffer = imageForDisp.BackBuffer; //获得 backBuffer
CopyMemory(backBuffer,buffer, length);       //复制 buffer 数据到 backBuffer
Application.Current.Dispatcher.Invoke(() =>
{
    imageForDisp.Lock();                     //锁定图像
    imageForDisp.AddDirtyRect(new Int32Rect(0, 0, width, height));//指定更新范围
    imageForDisp.Unlock();                   //解锁图像
});
```

说明：区别在第五行，这一语句的作用可简单理解为：将以下任务委托给视图线程进行操作。以此避免了线程冲突。

6 使用

此程序完全支持型号 MER-500-7UC-L，部分支持 MER-130-30UM-L，其他型号未验证。可同时控制多个相机。

插入相机后，需要点击刷新按钮获得设备列表，之后点击列表中的设备即可展开相机控制“Feature Control”和图像处理“Image Processing”，其中图像处理包含图像的基本属性调整和可用于实际工业场景的检测：表面缺陷(划痕)检测和尺寸检测。

点击储存“Save”可将当前帧图像、基本信息和检测结果储存在数据“Data”页面。数据页面中，点击删除按键“×”可删除当前图像和数据，双击图像可查看大图。

需要注意：

尺寸检测和表面缺陷检测将自动开启阈值功能，阈值大小可由用户根据需要调节。

尺寸检测时必须要使阈值分割后图像的背景为黑色，待检测物体为白色，当背景为白色且物体为黑色时需要在阈值一栏选择反转。在画面中可同时显示多个物体的尺寸，但在数据库中只储存面积最大的物体的尺寸，单位为像素。

表面缺陷检测时会将图像中会将一切与背景不同的部分识别为缺陷，因此如果待检测物件表面有文字或花纹等，需要采取手段将它们过滤。

7 结果演示

这部分只展示尺寸检测和表面缺陷检测，因为只有这两部分具有明确的检测结果，其他图像处理手段只是改变画面显示，不存在结果。此外，下面两个实验仅需要白色光源即可完成，使用彩色光源是因为器材限制。

7.1尺寸检测

按照以下方式搭建试验台：

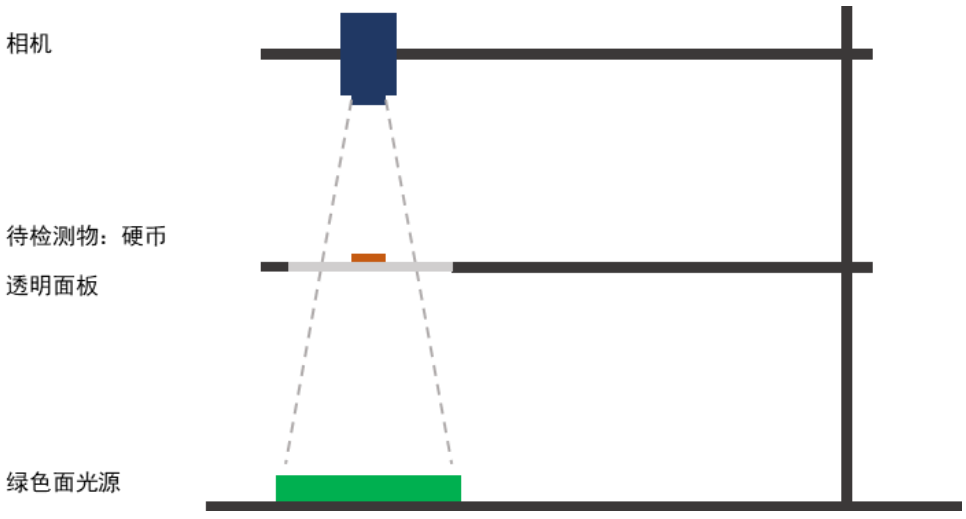

图 尺寸检测演示示意

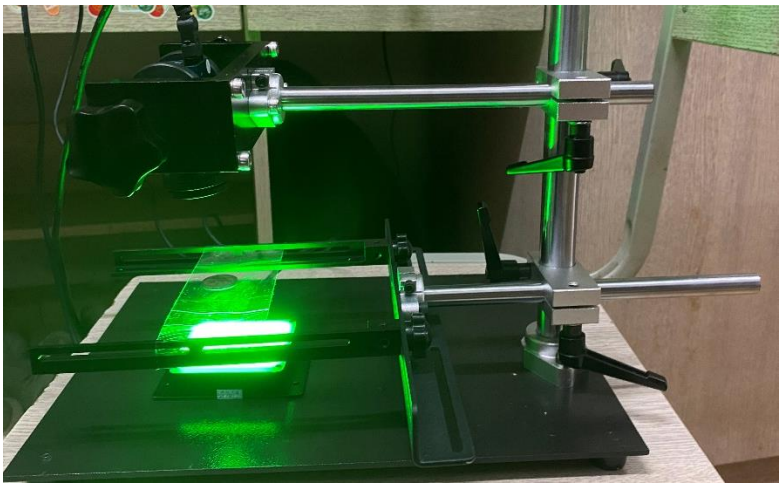

图 尺寸检测演示实物

检测结果：

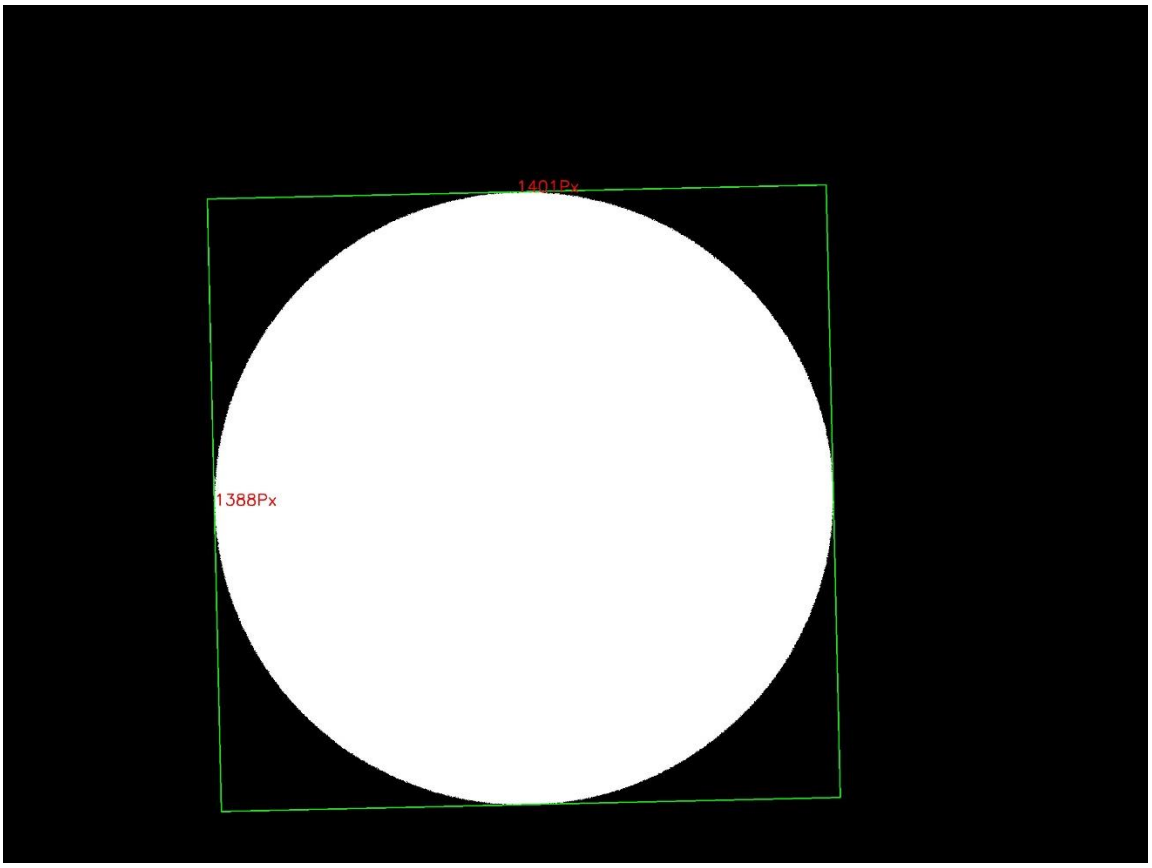

图 尺寸检测结果

图 数据页面储存的结果

结果分析：图中两个直径大小有细微差别，这是因为条件有限，使用胶带代替透明面板，造成放置面不平整，硬币倾斜。

7. 2表面缺陷检测

按照以下方式搭建试验台：

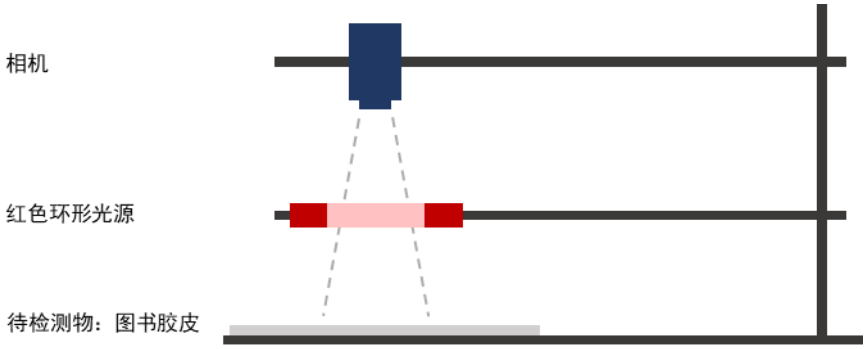

图 表面缺陷检测示意

图 表面缺陷检测实物

检测结果：

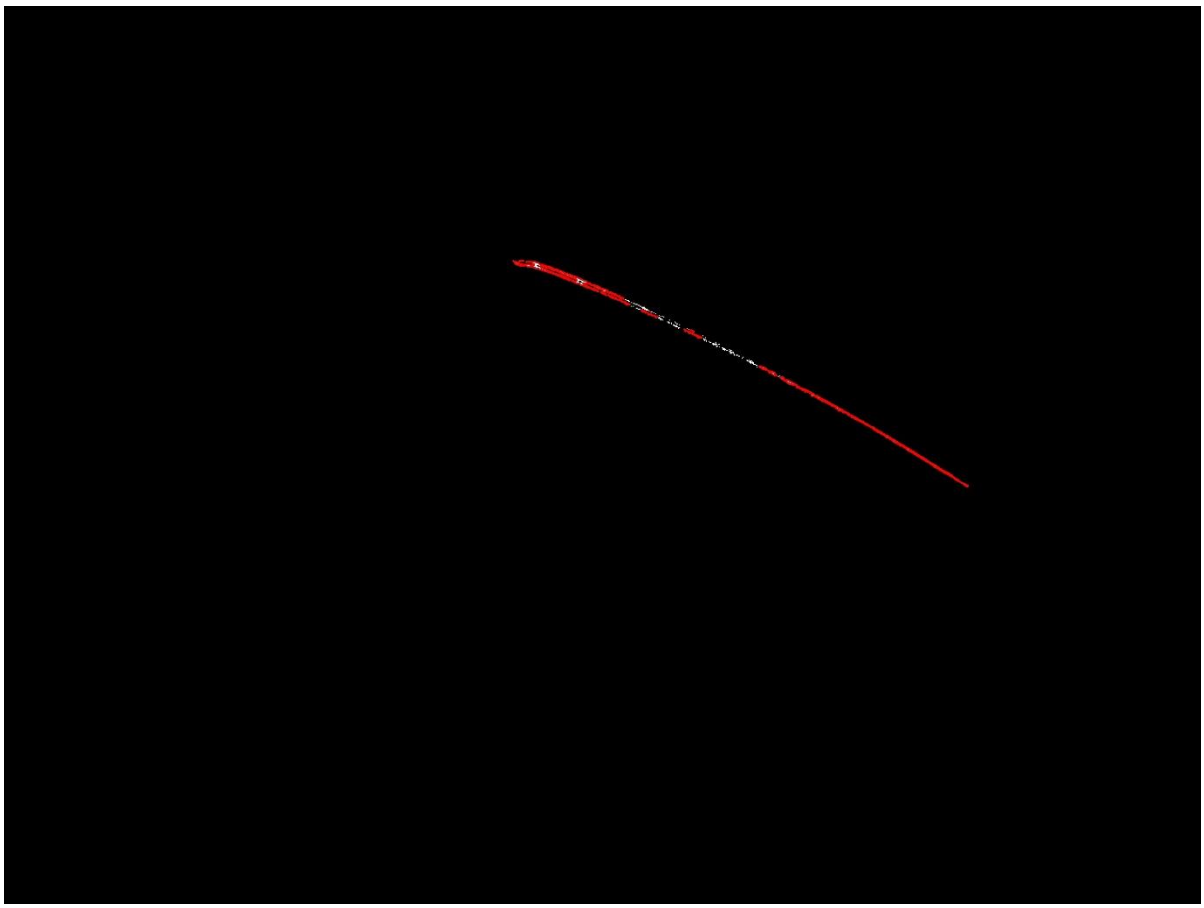

图 表面缺陷检测结果

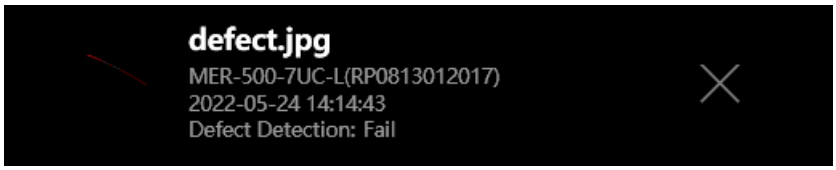

图 数据页面储存的结果

结果分析：图书胶皮表面的缺陷使用红色在图中标出，且数据页面储存了检测结果：未通过。