CMPE2800 – Review Lab – ImagePress (Term 1242)

In this review exercise, you will complete a project that does not have any bouncing balls of any kind.

The spirit of this activity will focus on a problem that is relatively easy to describe declaratively. You will need to assess what needs to be done, given some guidance, and implement a solution using the techniques you learned in previous courses, while following best practices.

The goal of the activity is to reduce the number of unique colours in an image by collapsing colours that are *similar*.

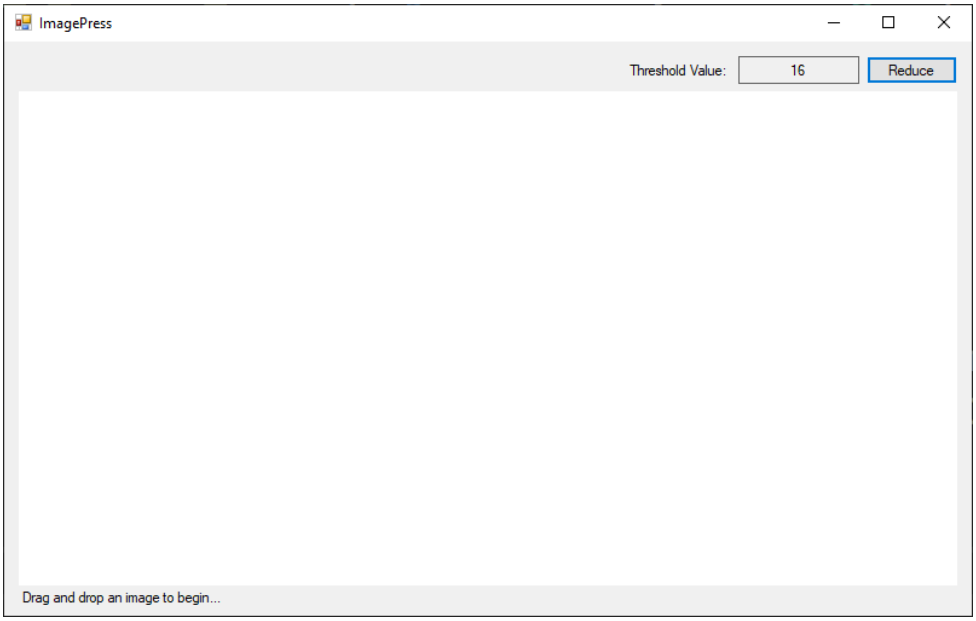
A user-determined threshold value will be used as the criteria for colours that are considered *the same*. Colours that have a sum of absolute value differences in RGB components less than or equal to the threshold will be considered the same.

The following partially completed base class will be used, intact, as a starting point for your derived class. It is your derived class that will perform the actual manipulation (the reduction) of the image. A version of this class will be provided on Moodle, but you can chop and paste from here.

```
public abstract class BaseBitmapManip
{
    // constructor requires a source Bitmap, and an error handler
    // the error handler will be used for error notification
    //  when operating the provided bitmap (load, NULL, etc.)
    public BaseBitmapManip(Bitmap bm, Action<string> error)
    {
        try
        {
            // other error handling here?

            // save a copy of the image
            BitmapOriginal = new Bitmap (bm);
        }
        catch (Exception ex)
        {
            error?.Invoke(ex.Message);
        }
    }

    // a COPY of the original bitmap (assigned from the constructor)
    public Bitmap BitmapOriginal { get; private set; }
```

```
    // build of a dictionary of colors and their frequencies from
    //  the source image
    // this will tell us the total number of colours in the image
    //  and the results may be ordered to find the most popular
    // colour (something of value for the manipulation)
    public Dictionary<Color, int> BuildColourTable()
    {
        // details omitted – you need to implement this

        return result;
    }

    // generate a 'difference' value from two colours
    // will be compared against a threshold value
    public static int GetColourDifference (Color A, Color B)
    {
        int iR = Math.Abs(A.R - B.R);
        int iG = Math.Abs(A.G - B.G);
        int iB = Math.Abs(A.B - B.B);

        return iR + iG + iB;
    }

    // abstract method to perform the image manipulation (reduction)
    //  returns a new image that is the reduced version (original unmodified)
    // your derived class will implement this behaviour
    public abstract Bitmap ReduceImage(int Threshold);
}
```

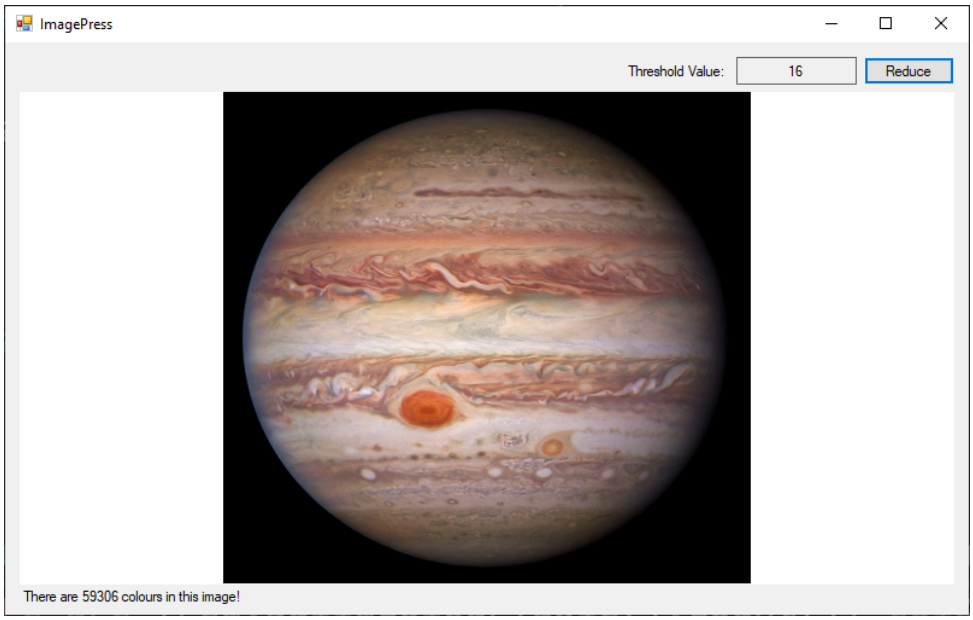
Create a standard Forms application with a similar appearance to the following:



The main control is a **PictureBox** control, with **Zoom** as the **SizeMode**, a white background, and anchored to grow with the application.

Complete the base class and also create a derived class. You may just put a placeholder in the derived class for the **`ReduceImage`** method. You can test these classes at this point.

When the user drops an image into the form, the image will be displayed, and the number of colours in the image will be displayed in the status label at the bottom of the form:



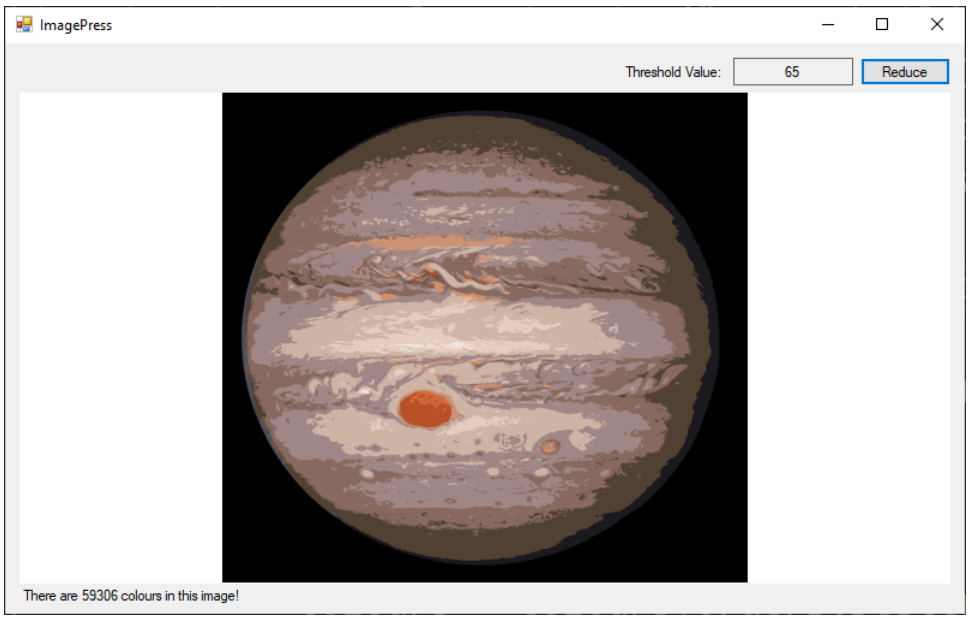
What does, and what *should* happen if the user drops a text document on the form? Make sure this is appropriately handed *now*.

The threshold value shown is just a label, but has the **`MouseWheel`** event wired such that the value will clamp in the range of **`1`** to **`256`** when the mouse wheels over this control. In the case above, the control has a border, and is not set to auto size. Also, the text alignment is full center.

When the user clicks the **`Reduce`** button, the image should have the colours reduced as informed by the threshold value. The general steps for this are:

- Figure out what colour is the most popular colour. This *could* use the **`BuildColourTable`** method to categorize the colours, then order the result (descending) by occurrence count. You are free to use other methods, or a different approach to pixel/colour management.
- Using the most popular colour, determine all the other colours that are less than or equal to the parameterized threshold value, using the **`GetColourDifference`** method. You should build this as a collection of **`Color`**.
- All the matching colours that you have just identified need to be replaced with the value of the most popular color, making them all the same (a reduction). Iterate over the image and replace any pixels that match any of the matching colours with the popular colour.
- Remove all of the matching colours from the table of colours in the image, then repeat the entire process until the colour table is empty.

The reduction in colour complexity will cause the image to lose detail, with the effect being more exaggerated as the threshold value increases:



The above steps may be implemented with brute force, or with substantial help from the collection classes and their extension methods. You are permitted to use whatever approach you deem appropriate to perform the reduction, as long as the result is the same as described above. Test images, suitable threshold values, and expected results will be made available.

However you implement your reduction, you will likely find that manipulating a large image will take significant time. Because of this, you will need to commit the manipulation code to a background thread. All of the best practices that go along with running this type of activity in a background thread are required. You get to decide where and how to implement this, but it needs to conform to strategies covered in previous courses, and needs to be obvious in the user experience.

Rubric

Documentation and application design guideline compliance are required to qualify for check-off.

Element	Description	Grade
Base Version	UI Correct, drop of image correctly displays and correctly identifies unique colour count in source image.	30
Manipulation	Manipulation functional, but some elements not correct (wrong colour used for reductions, incorrect effect). Completely wrong or arbitrary manipulations will not be accepted.	10
Manipulation	Manipulation functions as described.	30
Threading	Background thread correctly used for any and all blocking or potentially blocking activities. User interface shows that a transformation is in progress, and no other operations are permitted.	30