Problem 0 – Program Functionality

採用 Stack 的資料結構來實作 Program Functionality，每 Load 或處理完一張圖後即 Push 到此 stack 中，而每按一次 Undo 即 Pop 一張圖片，讓 Top 指標(我命名為 now)指向最後一張圖作為當下要處理的影像；基本的功能，如判斷 stack 是否為空/滿、stack 是否只有一張圖、現在指向的圖是否為灰階值等等。

```
/* Design the Data Structure for image file */
int pre = -1, now = -1, capacity = 4;

Bitmap^ temp;
cli::array<Bitmap^> ^stack_imgs = gcnew cli::array<Bitmap^>(capacity);

bool IsEmpty() { return now == -1; }
bool IsFull() { return now == capacity - 1; }
bool IsOne() { return pre == 0 && now == 1; }
bool IsGray() {
    bool isG = true;
    for (int i = 0; i < stack_imgs[now]->Height; i++)
        for (int j = 0; j < stack_imgs[now]->Width; j++) {
            Color RGB = stack_imgs[now]->GetPixel(j, i);
            if (RGB.R != RGB.G || RGB.G != RGB.B || RGB.R != RGB.B) {
                return isG = false;
            }
        }
    return isG;
}
```

(圖 1, Stack 資料結構)

其他的功能實作如 clone Bitmap, padding Image, applyFilter, Mat 類別的實作等等皆是為了方便每個作業的進行與要求，以中值濾波器為例，因為除了第一題之外，要求其他題目只能對灰階影像進行處理，所以先進行判斷是否為灰階圖，而進行卷積之前需先對原圖 padding 一圈"0"，再做卷積運算，運算後的圖再 push 到 stack 中成為即將要被處理的圖，如此一來即可達到 Program Functionality 的效果。

```
private: System::Void medianFilterButton_Click(System::Object^  sender, System::EventArgs^  e) {
    if (IsEmpty()) {
        cout << "Cannot work on empty image..." << endl;
        return;
    }

    if (!IsGray()) {
        cout << "The image is not a gray-level image..." << endl;
        return;
    }

    paddingImage(stack_imgs[now], 1);
    updateParameters();
    temp = cloneImage(stack_imgs[now]);
    Mat kernel(filterHeight, filterWidth, "median"); // Initial the kernel
    kernel.show();
    applyFilter(temp, kernel, kernel.name);
    pushImage(temp);
    showImage();
}
```

(圖 2, Median Filter 功能示例)

Problem 1 – RGB Extraction & transformation

- **Color extraction**

取得原 pixel 的 Color 物件後，將要轉換的 pixel 的 Color 用原 Color 的某 channel 替代，如要進行 Red 轉換，則將要轉換的 pixel 全用原 Color 的 R-channel 替代。

```
temp = cloneImage(stack_imgs[now]);
for (int i = 0; i < temp->Height; i++)
    for (int j = 0; j < temp->Width; j++) {
        Color RGB = temp->GetPixel(j, i);
        temp->SetPixel(j, i, Color::FromArgb(RGB.R, RGB.R, RGB.R));
    }
```

- Result

(原圖)

(結果 1, Red)

(結果 2, Green)

(結果 3, Blue)

- Discussion

若原圖的某 channel 的比例多，則結果會越亮，如此示例的綠色含量大，結果在 G-channel 的值會越大。

- **Color transformation**

將 RGB 3 個 Channel 的值取平均(i.e., (R+G+B)/3)作為灰階值。

  - Result

(結果 4, 灰階值)

  - Discussion

計算灰階值的算法有很多，根據人眼感官系統與心理學上，常見的為"Gray = R*0.299 + G*0.587 + B*0.114"，原因為人眼對綠色的亮度感最大，藍色最小，因此成為彩色轉灰階的標準，而我為了方便實作所以直接採用平均算法，讓 R、G、B 共享相同值作為灰階，如此一來直接取其中一個 channel 的值來運算即可，減少記憶體使用與計算量，但缺點是灰階影像較不適合人眼感受。

- **Conclusion**

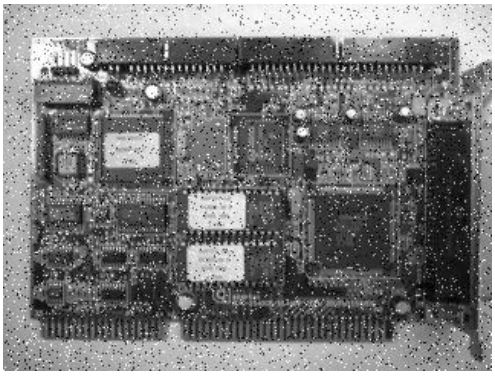
彩色與灰階值的提取雖然感覺很簡單，但要找出適合人眼感受與心理上的模型是需要嘗試的，尤其是應用於特殊的處理上(如醫學影像、衛星影像等等)，因此逐漸發展出各式各樣的算法。

Problem 2 – Smooth filter (mean and median)

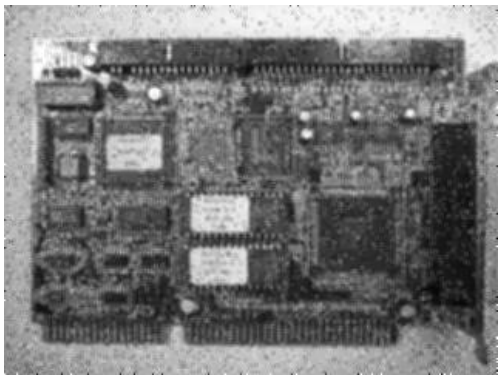
- **Mean filter**

  以一個 3x3 大小的 filter 其值皆為 0.333…，對原圖進行卷積。

  - Result



(原圖)



(結果 1)

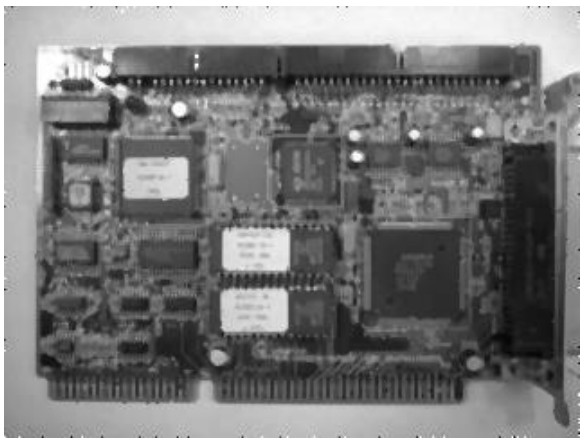
  - Discussion

white/black 的雜訊雖然會被周圍 pixel 的灰階平均值取代，但那些周圍若有 white/black 雜訊的 pixel，則會被用過高或過低的灰階平均值替代，造成影像模糊。

- **Median filter**

取周圍 8 個 pixel 且包含自己共 9 個 pixel 的中間值來設值，我使用 Merge Sort 對 9 個 pixel 進行排序後取中值(i.e., 排序後第 5 個灰階值)。

  - Result



(結果 2)

  - Discussion

採用 Merge Sort 的原因是其複雜度為 O(NlogN)相對其他的排序法算快的，且用中值來進行過濾雜訊的效果十分良好，主要是雜訊通常出現在一物體或一小塊面積中，所以用周圍的灰階值直接代替會最自然，如出現在(原圖)晶片上的那些雜訊，所以直接用晶片的像素去替代的話感覺最不突兀。

- **Conclusion**

White/black 雜訊的影像用中值濾波器效果最自然，而平均值濾波器則適用於需要做柔化、去銳利化或模糊化處理的影像。而實作上若有使用到卷積的話，我皆會先對原圖進行 Padding 一圈 ”0” 來使卷積後的圖仍保持原圖大小。

Problem 3 – Histogram Equalization

- **Histogram equalization**

計算原圖灰階值(0~255)出現的個數，再將這些個數同除原圖的面積得到灰階值出現的機率，計算均等化後的灰階值分布，均等後的灰階值 k'的為前 k 個灰階值的出現機率總和再乘於 255(i.e., L - 1)，再將原圖的每個 pixel 的灰階值用均等後的灰階值 k'替代即可。

- Result

(原圖)

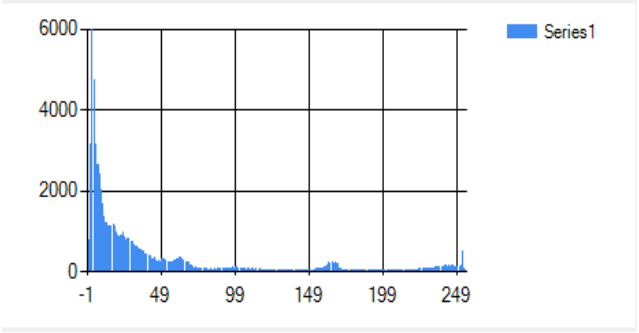
(結果 1)

- Discussion

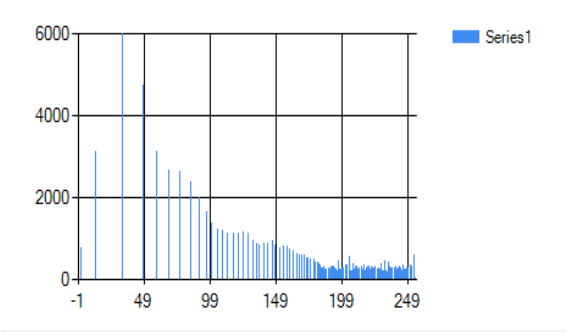
直方圖均等化的方法可以分散過度集中出現的灰階值機率，因此較灰暗的圖經過均等化後會變得明亮，反之亦然，實作上沒有太大的問題，我這裡多做了幾個步驟，先將 Bitmap 存成矩陣的形式並進行 Merge Sort 的排序，因此計算個數的時候複雜度會從 $O(N^2)$變為 O(NlogN)。

- **Show histogram**

使用 Chart 工具進行畫直方圖，(結果 2)為經直方圖後的灰階圖分布。



(原圖)



(結果 2)

- **Conclusion**

直方圖均等化非常的實用，尤其是在影像過度曝光或是光線不足的時候效果非常佳，而且演算法的複雜度也不會太大，實作也相當容易。

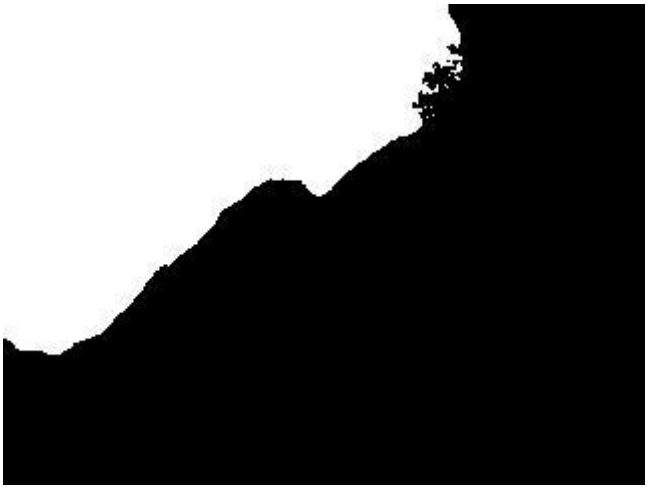
Problem 4 – A user-defined thresholding

- **Thresholding**

二元的閥值判斷，取出原圖的 pixel 判斷是否大於/小於閥值，如果成立則用 255/0 取代其原本的灰階值。

  - Result

(原圖)



(結果 1, 閥值 127)

  - Discussion

用基本的邏輯判斷即可完成實作，可以將圖轉換成二元圖，即只有黑白兩色，因此相當適用於邊緣提取。

- **Conclusion**

閥值的設定有相當多種變化，如動態閥值、取灰階平均值作為閥值等等，而使用者自定義的話則相當容易實作，但效果優異取決於使用者的主觀。

Problem 5 – Sobel edge detection

- **Vertical, Horizontal**

```
if (name == "sobel_x")
    data = gcnew cli::array<float, 2>(height, width)
    { {1, 0, -1 },
    { 2, 0, -2 },
    { 1, 0, -1 }
};
```

(垂直邊緣偵測 filter)

```
else if (name == "sobel_y")
    data = gcnew cli::array<float, 2>(height, width)
    { { 1, 2, 1 },
    { 0, 0, 0 },
    { -1,-2,-1 }
};
```

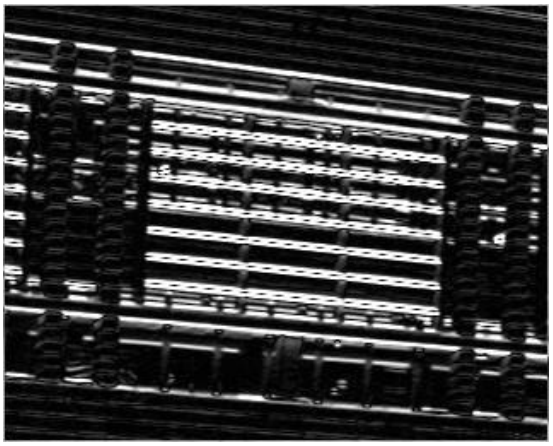
(水平邊緣偵測 filter)

分別利用上面兩個 filter 對原圖進行卷積即可得到兩不同方向的邊緣圖。

- Result

(原圖)

(結果 1, 垂直偵測)



(結果 2, 水平偵測)

- Discussion

filter 公式是固定的且只是進行卷積運算，實作上不算困難，但要注意卷積後的值可能會出現負值(filter 中有相減的運算)，因此可以將運算後的值平方開根號，或者直接取絕對值來減少計算量。

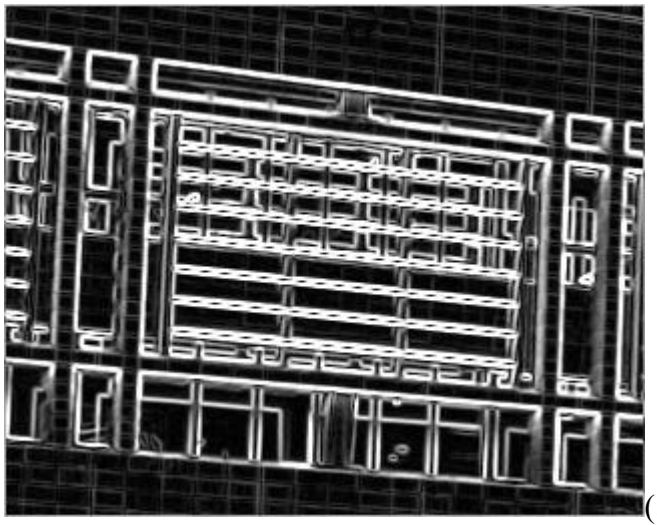
- **Combined**

對原圖同時進行垂直與水平偵測，並將各別求得之值(g_x, g_y)去計算其 Norm 值 ($\sqrt{g\_x^2 + g\_y^2}$, i.e., 梯度大小)即可。

```
Mat sobel_x("sobel_x");
Mat sobel_y("sobel_y");

for (int i = 1; i < Height - 1; i++) {
    for (int j = 1; j < Width - 1; j++) {
        double gray = 0, sum = 0, g_x = 0, g_y = 0;
        for (int h = i - 1; h < i + filterHeight - 1; h++) {
            for (int w = j - 1; w < j + filterWidth - 1; w++) {
                Color RGB = stack_imgs[now]->GetPixel(w, h); // Getpixel (y,x)
                gray = RGB.R;
                g_x += sobel_x.data[h - i + 1, w - j + 1] * gray;
                g_y += sobel_y.data[h - i + 1, w - j + 1] * gray;
                sum = sqrt(g_x * g_x + g_y * g_y);
                sum = sum > 255 ? 255 : sum;
                temp->SetPixel(j, i, Color::FromArgb(RGB.A, sum, sum, sum));
            }
        }
    }
}
```

- Result



(結果 3, 混和)

- Discussion

同時偵測兩個方向的邊緣，Sobel Filter 旨在計算每個 pixel 間的與周圍的亮度梯度近似值，因此與周圍點的灰階值變化越大，得到的值越大(越接近白色 255)並視為邊緣。

- **Conclusion**

Sobel 邊緣偵測的效果與 Canny 算子等比起來雖然有限，但實作上卻相當容易，如果不是要應用在邊緣複雜、景象變化大的影像，採用 Sobel 是相當實用的，如示例，對簡單的線條邊緣提取效果還不錯。

Problem 6 – Edge overlapping

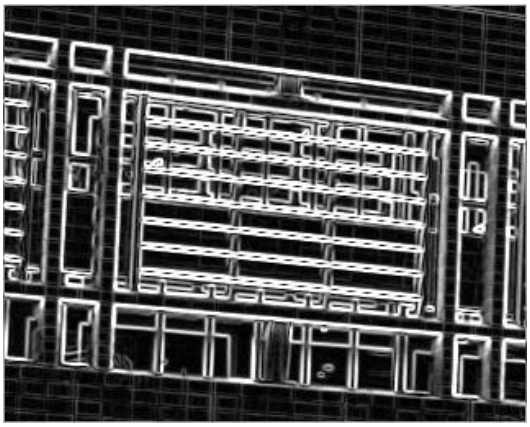
- **Edge extraction and overlapping**

對提取邊緣後的圖進行閥值判斷並轉換成二元圖(只有黑白二值)，將灰階值 255 的 pixel 視為邊緣，在原圖中的該位置用綠色去取代(i.e., Setpixel(Color::From(0,255,0))，否則用原灰階值設定即可。

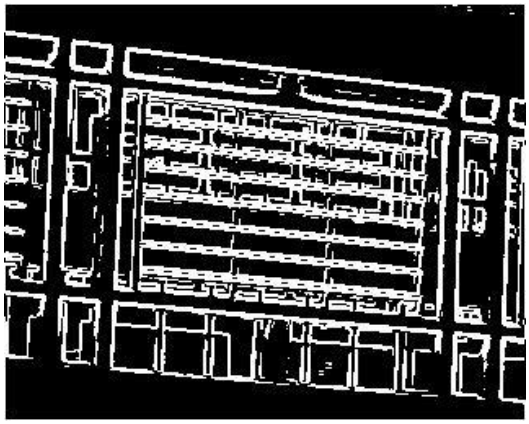
```
for (int i = 0; i < Height; i++) {
    for (int j = 0; j < Width; j++) {
        /* Extract the edge */
        Color RGB = stack_imgs[now]->GetPixel(j, i);
        edge = RGB.R;
        /* Extract the original pixel */
        Color RGB_2 = temp->GetPixel(j, i);
        gray = RGB_2.R;
        if (edge == 255) temp->SetPixel(j, i, Color::FromArgb(0, edge, 0));
        else if (edge == 0) temp->SetPixel(j, i, Color::FromArgb(gray, gray, gray));
    }
}
```

- Result

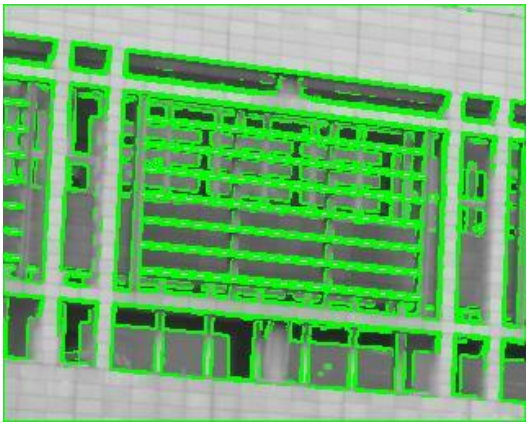
(原圖)



(結果 1, Combined)



(結果 2, 閥值 127)



(結果 3, 疊合綠色邊緣)

- Discussion

實作上只是簡單的邏輯判斷而已，並無太大的問題。

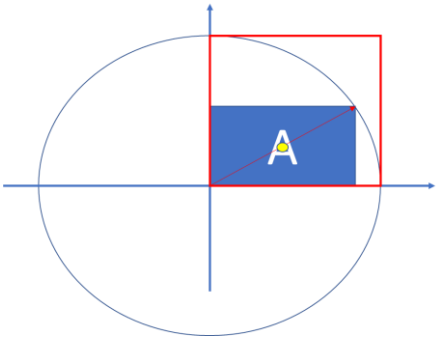
- **Conclusion**

閥值的設定非常主觀，閥值設定太大會讓許多非邊緣的 pixel 被判定為邊緣，太小則太過嚴謹。

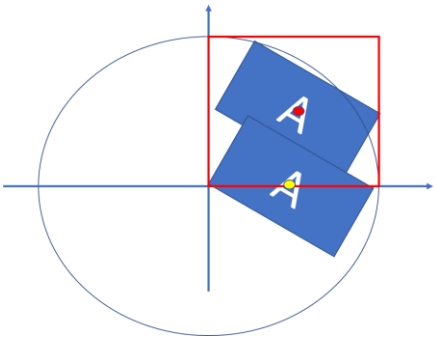
Problem 7 – Image rotation, stretching in both horizontal and vertical Direction

- **Rotation**

原圖經旋轉矩陣 R 轉換後會以原圖的對角線為半徑，原點(0, 0)為圓心做旋轉(如圖 1)，落在四個象限中，故初始化一個長寬皆為原圖對角線長(diagonal)的 Bitmap temp 來存旋轉後的圖，而 Bitmap 的 pixel 位置無法存放負值(旋轉後可能會落在 2, 3, 4 象限)，因此需要對每個旋轉後的 pixel 做平移，也就是將整張圖移到第一象限中(如圖 2)，首先計算旋轉前的中心點 p(x = Height/2, y = Width/2)，並將此中心點經旋轉矩陣 R 轉換得到新中心點 p'(x' = R(x), y' = R(y))，再計算新中心點 p' 到 temp 中心的位置(nx = diagonal/2, ny = diagonal/2)，其水平與垂直距離分別為 dx = nx – x'、dy = ny – y'，如此一來，將原圖每個 pixel 旋轉後同時再加上 dx、dy，即可平移到第一象限中。



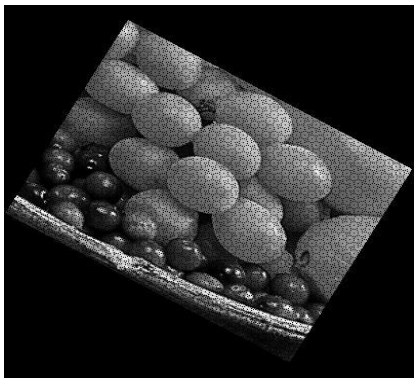
(圖 1, 黃點為中心點 p)



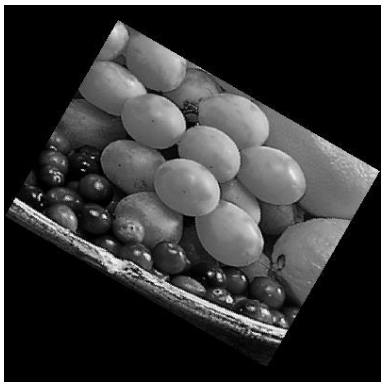
(圖 2, 紅點為 temp 中心點)

- Result

(原圖)



(結果 1, 空洞現象)



(結果 2, 反轉換找值)

- Discussion

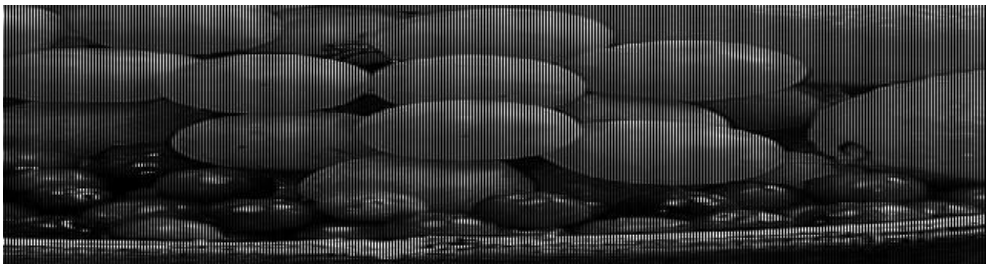
temp 的長寬為(原圖)對角線長，因此並非每個 pixel 都會被設定到值而造成空洞現象(如結果 1)，可以採用內插法(如雙線性、最近相鄰點等)解決，而我將有空洞的圖經反轉換(順轉變逆轉)，讓 temp 的每個 pixel 去取反轉換後在原圖之值，如此一來可將所有空洞補齊(如結果 2)，且感官效果也較佳，而在實作上要注意反轉換會連同 temp 未設值的部分(全黑的地方)一起轉，因此反轉換後若超出原圖長寬的那些座標值要忽略之。

- **Stretching**

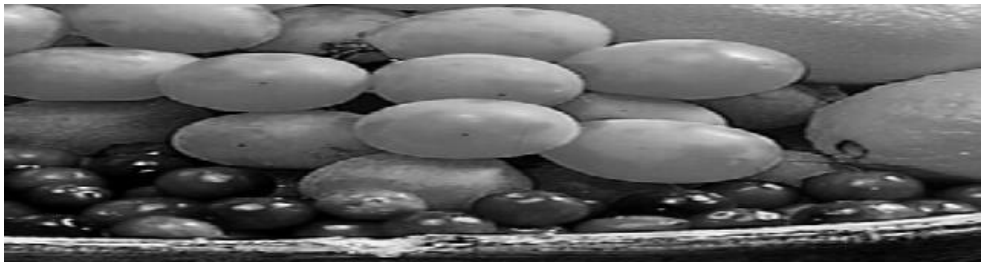
原圖經拉伸矩陣 S 轉換後，每個 pixel 會沿 x、y 方向分別放大(或縮小)cx、cy 倍，因此可直接初始化一個 Bitmap temp，長寬分別為原圖的 cx、cy 倍，但實作上還要再分別減去 cx – 1、cy – 1，以免 temp 邊緣出現為賦值的空洞。

- Result (cx=0.7, cy=2)

(原圖)



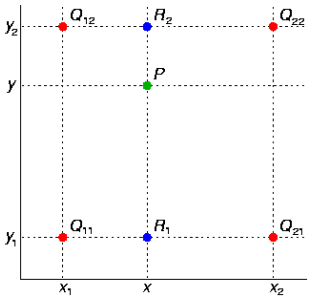
(結果 1, 空洞現象)



(結果 2, 雙線性內插)

- Discussion

與旋轉一樣在 temp 中會有未被賦予值的 pixel 而造成空洞現象，這裡我採用雙線性內插法，將原圖中相鄰四個 pixel(Q11, Q12, Q21, Q22)經過轉換後所圍起來之範圍的每個 pixel P 作用，根據 P 與此四個值的距離並按比例賦值，公式如下，概念為做兩個方向的單線性內插。



$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}),$$
$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}).$$

$$f(x, y) \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2)$$

- **Conclusion**

進行 Affine Transform 會造成資訊失真的情況，要如何挑選適當的內插法皆是根據問題需求，如最近鄰居內插實作易與計算速度較快，但視覺上會有 blocking artifact，雖然雙線性內插相比起來較難實作且計算速度慢，但優點是視覺效果還不錯；旋轉的內插則是用反轉換回去找值來補的效果會最自然，但缺點是要進行兩次轉換，必須犧牲計算速度。