Seam Carving

第二组：李思哲 徐蕊琦 祁煜

北京大学 信息科学技术学院

2021.6.18

目录

CONTENT

1 Seam Carving基本算法实现

2 Seam Carving应用实例

3 关于能量函数的调研

4 成果展示（UI界面及视频）

Seam Carving基本算法实现

By Sizhe Li

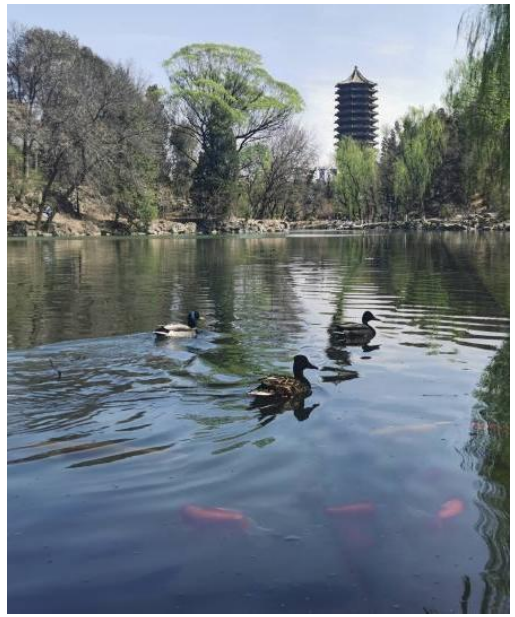
1

Seam Carving基本算法实现——缩小

step1
读取原图，计算能量函数

step2
利用动态规划找能量最低的seam

step3
依次删除能量最低的seam



具体实现——动态规划找能量最低路

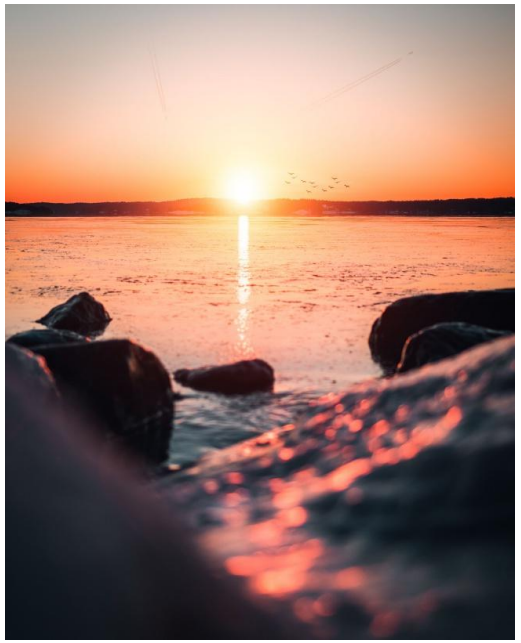
$$\mathbf{s^x} = \{s_i^x\}_{i=1}^n = \{(x(i), i)\}_{i=1}^n, \text{ s.t. } \forall i, |x(i) - x(i-1)| \leq 1,$$

$$\mathbf{s^y} = \{s_j^y\}_{j=1}^m = \{(j, y(j))\}_{j=1}^m, \text{ s.t. } \forall j |y(j) - y(j-1)| \leq 1.$$

```
for i in range(1, r):
    for j in range(1, c+1):
        ls = [tmp[i-1][j-1], tmp[i-1][j], tmp[i-1][j+1]]
        note[i][j-1] = ls.index(min(ls))-1  # 记录上一个点的方位
        tmp[i][j] += min(ls)    # 记录到该点的最低能量

index = int(np.argmin(tmp[r-1]))    # 从最后一行找到最低能量
road = [index]
for i in range(r-2, -1, -1):
    index += note[i][index]  # 上一个点的横坐标
    index = int(index)
    road.append(index)
road.reverse()
```

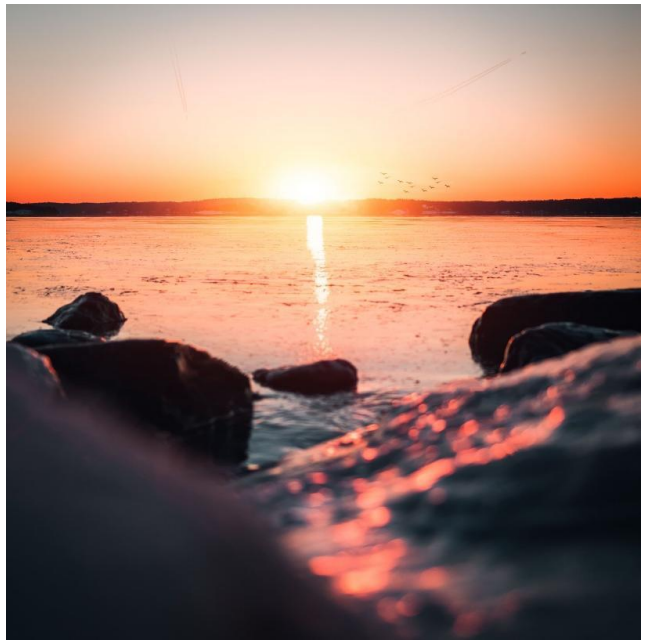
Seam Carving基本算法实现——放大

step1
读取原图，计算能量函数



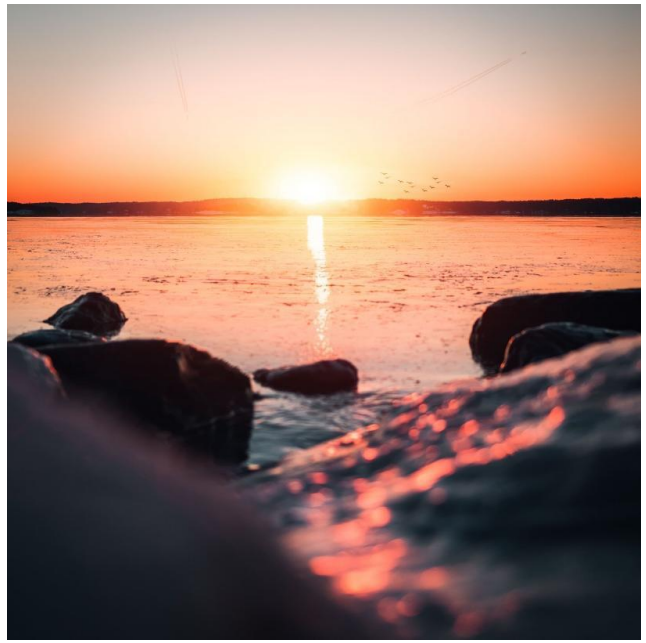
step2
利用动态规划找能量最低的N条seams

step3
依次复制能量最低的N条seams

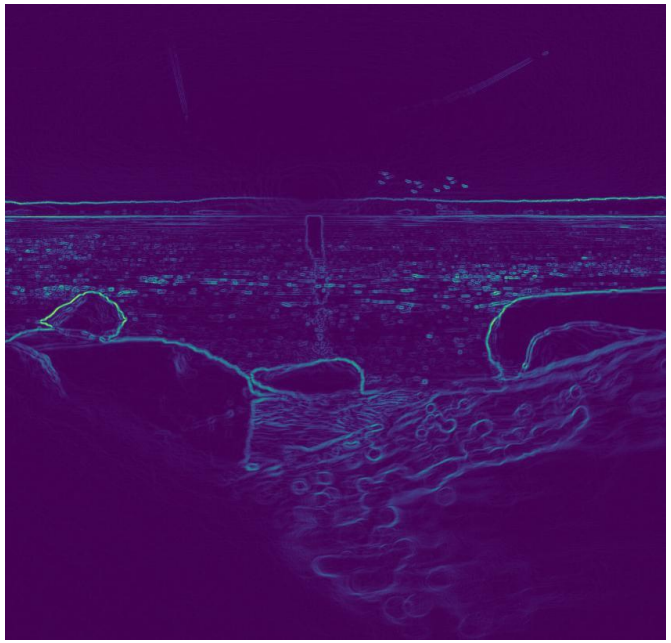


Seam Carving基本算法实现——放大

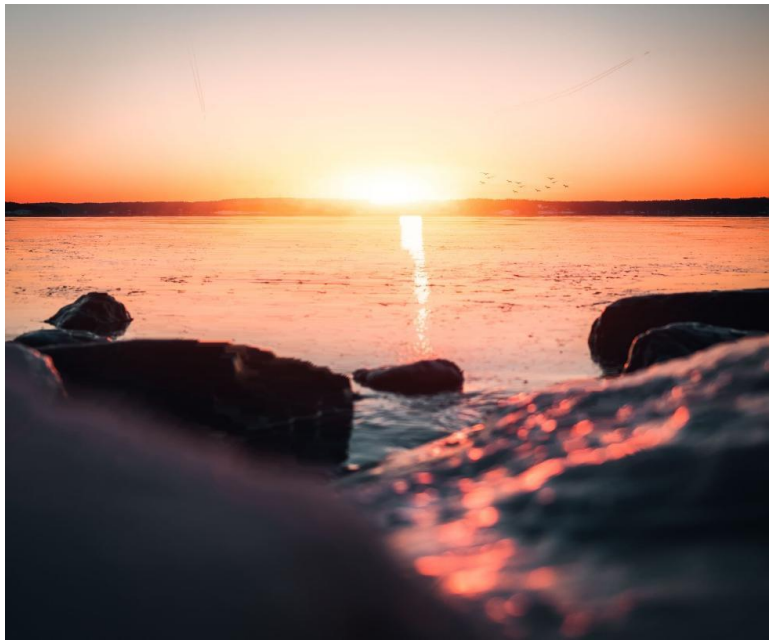
step4
将初次放大得到的图像视为下一次的输入



step5
利用动态规划找能量最低的N条seams



step6
依次复制能量最低的N条seams，如此反复



Seam Carving基本算法实现——技术细节


Detail 1

将竖直方向的变化转化为水平

```
def func(img, scale):
    if direction == 1:
        img = np.rot90(img, 1, (0, 1))

    r, c, _ = img.shape
    if scale < 1:...

    else:...

    if direction == 1:
        img = np.rot90(img, 3, (0, 1))

    return img
```

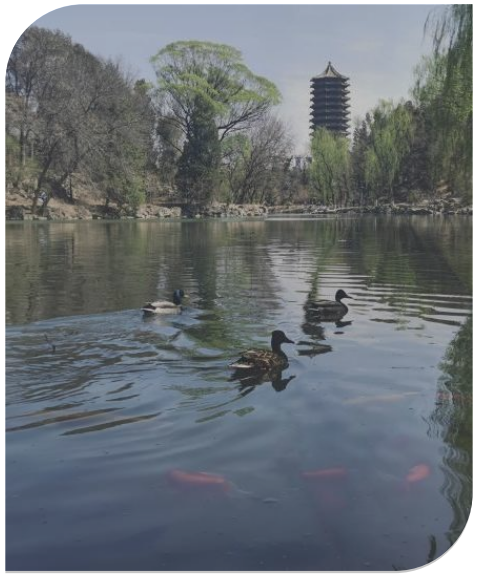
Detail 2

如何寻找、添加能量最低的n条seams

```
for i in trange(n):
    r, c, _ = img.shape
    road = dp(img_e)
    for j in range(len(road)):
        img_e[j, road[j]] = inf # 已经访问过的能量记为inf
        note[j, road[j]] += 1   # 记录该点需被复制多少次

note = note.flatten()
img = np.reshape(img, (r*c, 3)) # reshape成2维进行操作

for i in range(r*c-1, -1, -1):
    if note[i] == 0:
        continue
    for j in range(int(note[i])):
        img = np.insert(img, i, img[i], axis=0)
```

2

Seam Carving 应用实例



Multi-Size缩放

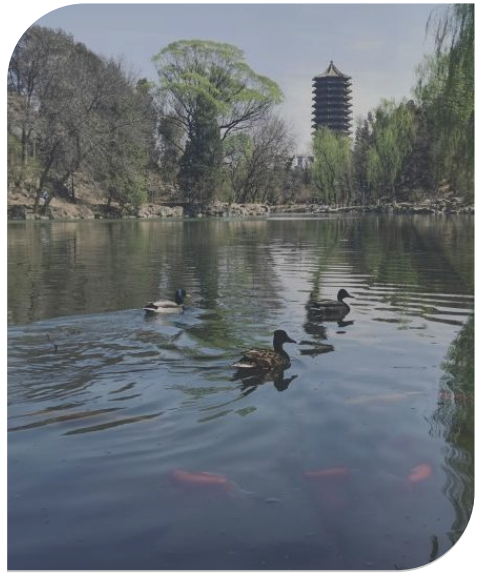
By Sizhe Li

视频中的应用

By Sizhe Li

2 Directions

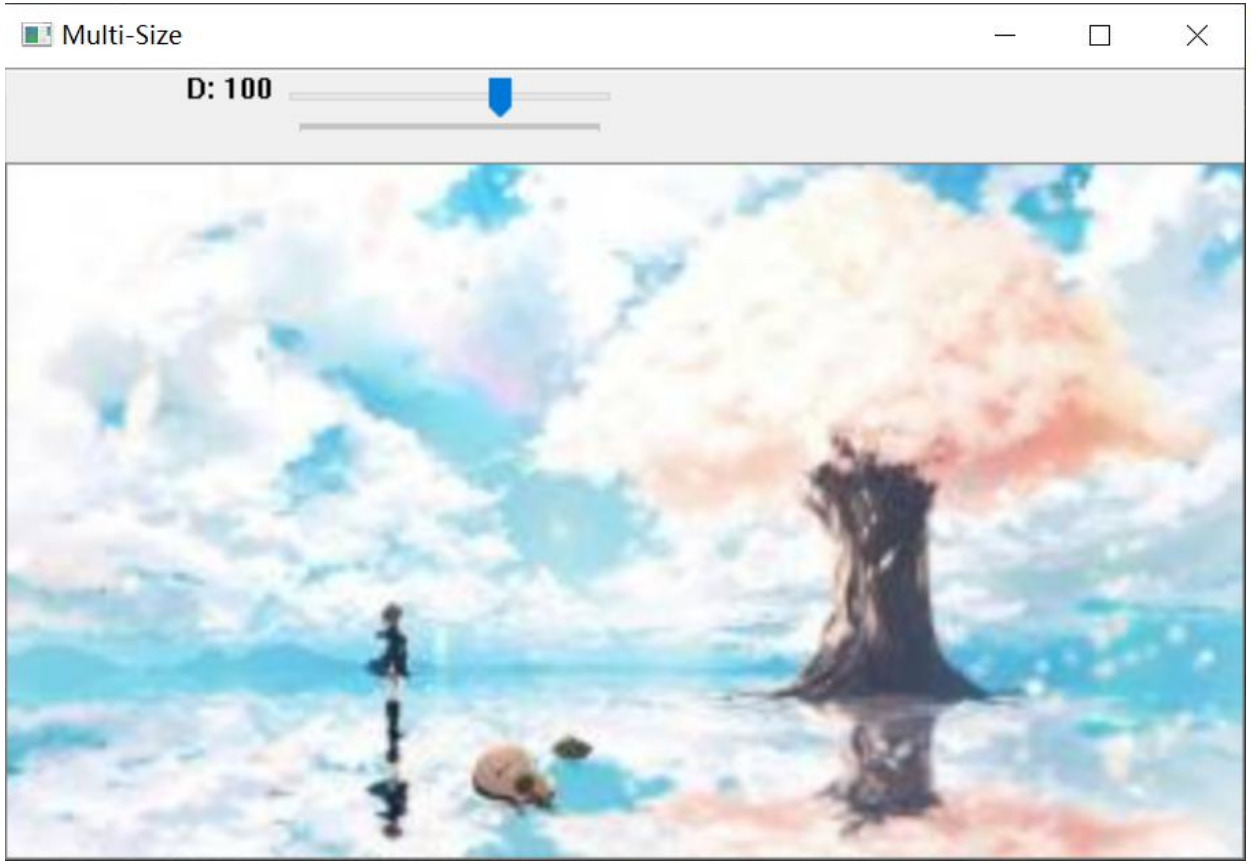
By Ruiqi Xu



与Grab Cut的结合

By Ruiqi Xu

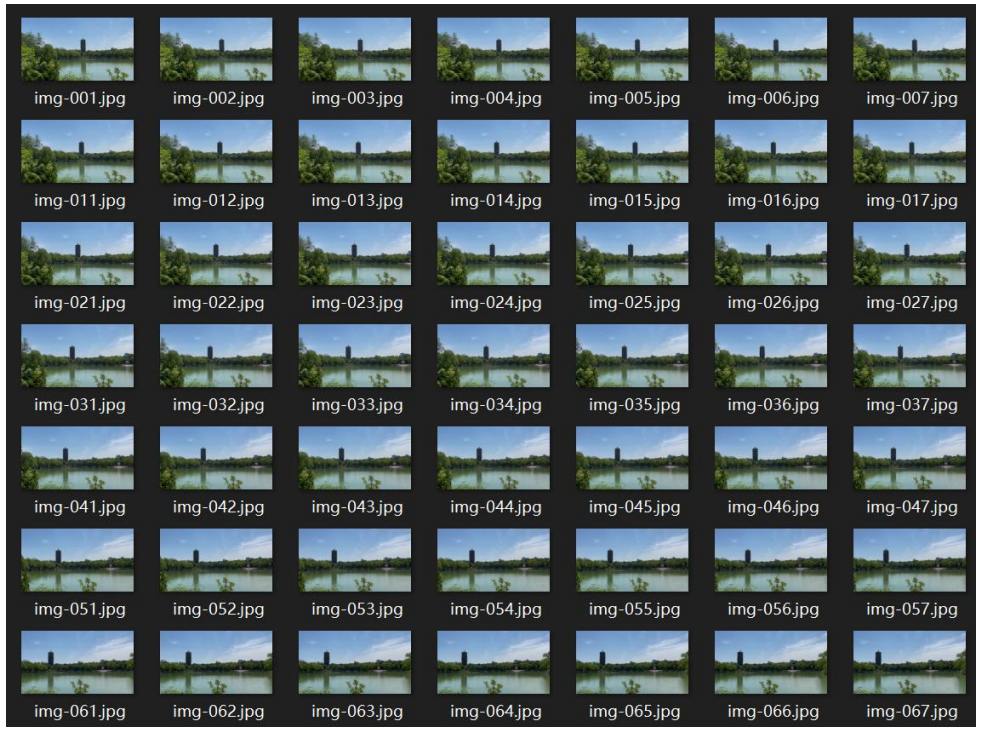
Multi-Size 缩放



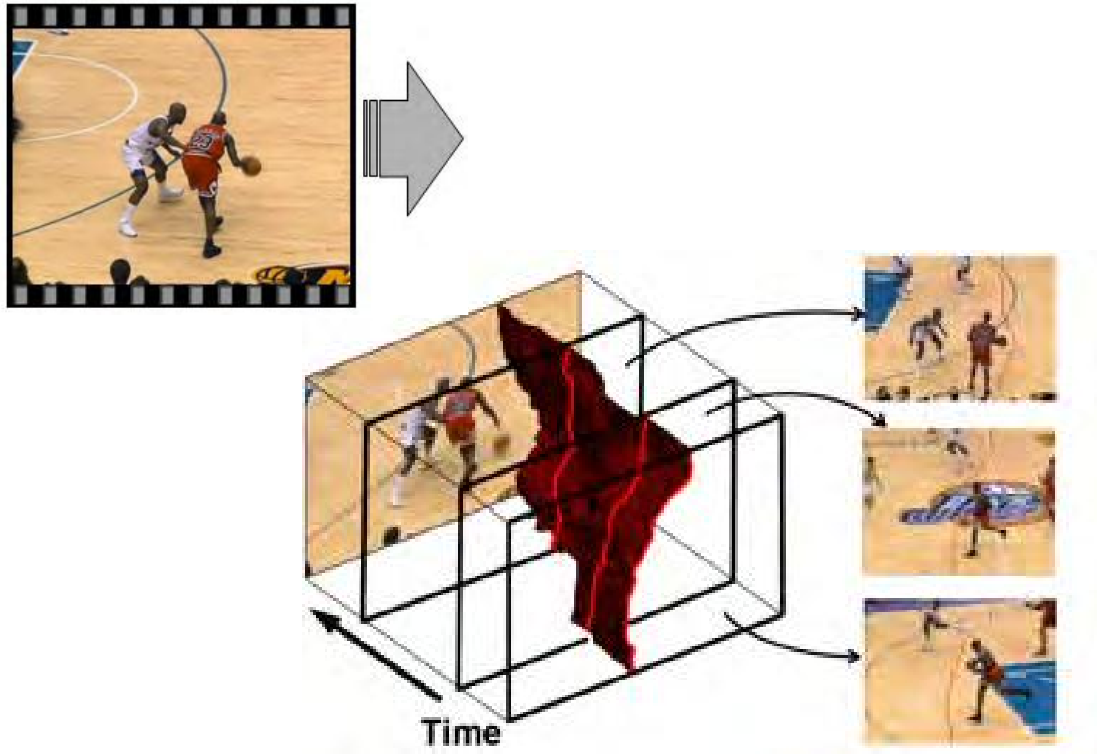

```
for i in trange(c):
    road = dp(energy_function(img))
    img = delete_one_road(road, img)
    ls.append(road)

ls.reverse()
for road_now in ls:
    for i in range(r):
        num_map[i].insert(road_now[i], cnt)
    cnt -= 1

for i in range(r):
    for j in range(c):
        if order_map[i][j] <= total:
            mask[i][j] = False

mask = np.stack([mask] * 3, axis=2)
img = img[mask].reshape((r, c - total, 3))
```

视频中的实现方法

step1
将Video按帧提取图片
工具：ffmpeg



step2
将其视为三维空间中寻找能量
最低平面的问题



Reference:《Improved Seam Carving for Video Retargeting》

视频中的实现方法

step3
对于第一帧，利用常规方式寻找能量最低的seam

step4
将后一帧与前一帧的时序关联，用seam之间的距离进行限制

```
for i in range(r):
    # 计算新的路径可能的范围，缩小动规的区域
    left = max(1, old_road[i] - n)
    right = min(c, old_road[i] + n)
    for j in range(1, c+1):
        if j < left or j > right:
            tmp[i][j] = inf
        elif i != 0:
            ls = [tmp[i-1][j-1], tmp[i-1][j], tmp[i-1][j+1]]
            note[i][j-1] = ls.index(min(ls)) - 1
            tmp[i][j] += min(ls)
```

视频中的实现方法

step5

结合seam carving和等比缩放

1920*1080

1920*960

Seam Carving

等比缩放

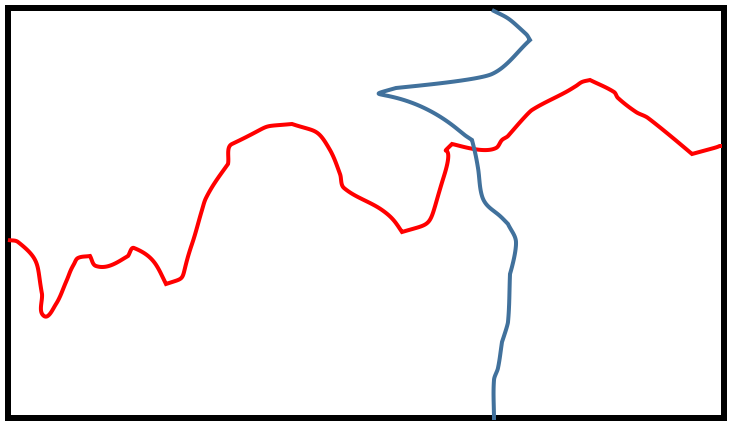
1920*840

二维缩放

实现方式：①直接比较横竖seam的平均能量大小
②论文中的动规实现方式

用grabcut保护重要区域

①grabcut实现流程
②grabcut底层的maxflow实现机制



直接比较road1和road2的平均能量，谁小就删谁。

当缩放尺寸与原尺寸相差不大时，其实效果很好。但当尺寸相差较大时，效果比较差。

在时序上进行动规：

T(r, c) = min{T(r – 1, c) + 在得到T(r – 1, c)的基础上删除能量最小的水平线，
T(r, c - 1) + 在得到T(r, c - 1)的基础上删除能量最小的竖直线}

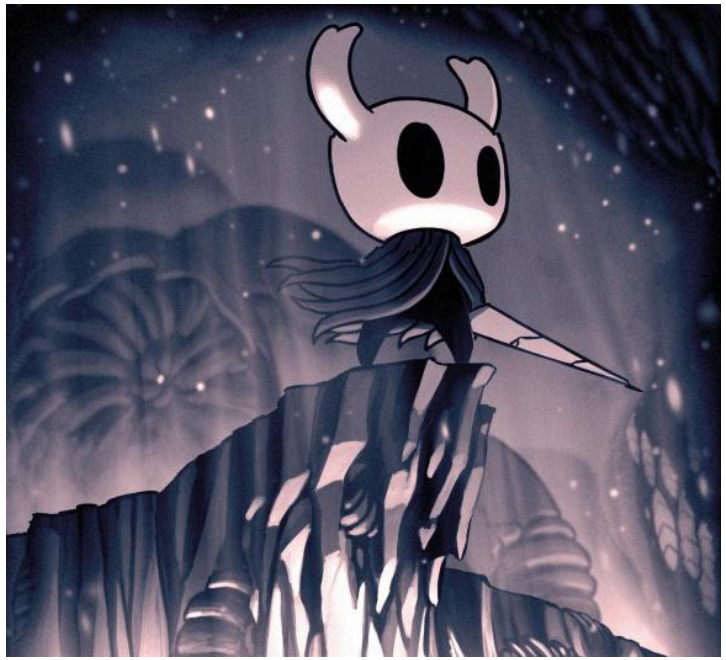
问题：

①太慢

②消耗空间极大

③当缩放尺寸较小时效果与方法一相同，当缩放尺寸较大时效果也并不好

方法一



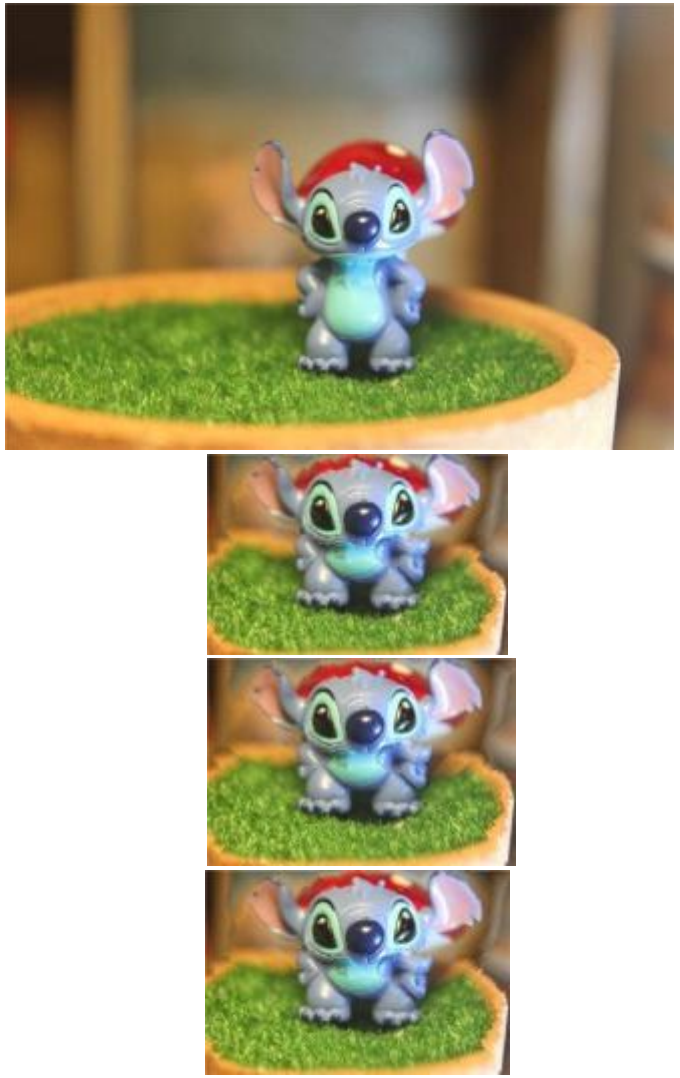
方法二

说明：

①真正实现时并不是完全复现论文的整个动规，为了节省空间做成了可调范围内的局部动规

②和方法一效果相差不大

```
if r < c:
    for i in range(int(E / c * r) + 1):
        T.append([])
        for j in range(E):
            T[i].append(np.zeros((m, n, 3), dtype=float))
    k = int(c / E)
    t = np.zeros((int(E / c * r) + 1, E), dtype=float)
else:
```



```
cv2.grabCut(src, mask, rect, bgdmodel, fgdmodel, 20, mode=cv2.GC_INIT_WITH_RECT)
```



Initialisation

- User initialises trimap T by supplying only T_B. The foreground is set to $T_F = \emptyset$; $T_U = \overline{T}_B$, complement of the background.
- Initialise $\alpha_n = 0$ for $n \in T_B$ and $\alpha_n = 1$ for $n \in T_U$.
- Background and foreground GMMs initialised from sets $\alpha_n = 0$ and $\alpha_n = 1$ respectively.

Iterative minimisation

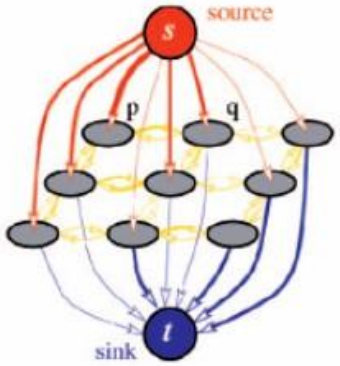
1. *Assign GMM components to pixels:* for each n in T_U,
$$k_n := \arg\min_{k_n} D_n(\alpha_n, k_n, \theta, z_n).$$
2. *Learn GMM parameters* from data $\mathbf{z}$:
$$\underline{\theta} := \arg\min_{\underline{\theta}} U(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z})$$
3. *Estimate segmentation:* use min cut to solve:
$$\min_{\{\alpha_n:\, n \in T_U\}} \min_{\mathbf{k}} \mathbf{E}(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z}).$$
4. Repeat from step 1, until convergence.
5. Apply border matting (section 4).

User editing

- *Edit:* fix some pixels either to $\alpha_n = 0$ (background brush) or $\alpha_n = 1$ (foreground brush); update trimap T accordingly. Perform step 3 above, just once.
- *Refine operation:* [optional] perform entire iterative minimisation algorithm.

Figure 3: **Iterative image segmentation in GrabCut**

Grabcut



$$\mathbf{E}(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z}) = U(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z}) + V(\underline{\alpha}, \mathbf{z})$$

$$\sum_n D(\alpha_n, k_n, \underline{\theta}, z_n) \qquad \gamma \sum_{(m,n)\in \mathbf{C}} [\alpha_n \neq \alpha_m] \exp -\beta \|z_m - z_n\|^2$$

$$-\log \pi(\alpha_n, k_n) + \frac{1}{2} \log \det \Sigma(\alpha_n, k_n)$$

$$+\frac{1}{2}[z_n - \mu(\alpha_n, k_n)]^\top \Sigma(\alpha_n, k_n)^{-1} [z_n - \mu(\alpha_n, k_n)]$$

参考论文：

An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision

*Yuri Boykov and Vladimir Kolmogorov**

Dinic算法、FF算法……Yuri提出的算法在实现中效率更高，其主要循环三个阶段：

①增长：

从S和T分别建树——每个叶子搜索相邻并且邻边仍有可用流量的节点囊括进树里，直到两棵树碰到了一起（找到了一条增广路径）

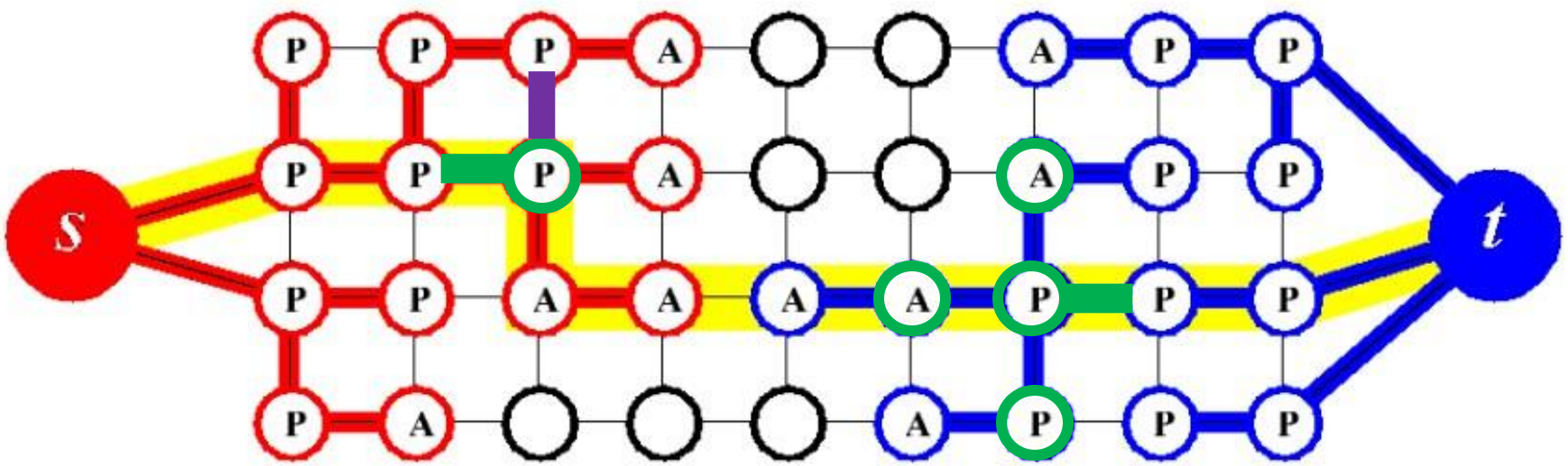
②增广：

求出这条路上的最大流量并加上，此时至少有一条饱和边，若在同一棵树上则将饱和边指向的点加入孤儿顶点集（也就是断开这条饱和边）

③领养：

从孤儿顶点集中拿出一个点，若相邻的点里有和它原来一棵树的并且邻边仍有可用流量，那就让这个点收留这个孤儿点；如果找不到就把这个孤儿点重新放为自由状态，并把它的所有子节点设为孤儿。

设为孤儿顶点：把两棵树变成了更多树的森林



算法实现中几个有意思的地方：
①粗略看一下GCGraph类中我们需要实现的函数
②S和T是两个“方向相反”的树，除了每次都判断方向（if else）怎么用一个式子来定呢?
③饱和边刚刚好在中间怎么办？还设孤儿顶点吗?
④用时间戳来优化：尽可能找得离根节点更近

```
public:
    GCGraph();
    GCGraph( unsigned int vtxCount, unsigned int edgeCount );
    ~GCGraph();
    void create( unsigned int vtxCount, unsigned int edgeCount );
    int addVtx();
    void addEdges( int i, int j, TWeight w, TWeight revw );
    void addTermWeights( int i, TWeight sourceW, TWeight sinkW );
    TWeight maxFlow();
    bool inSourceSegment( int i );
```

二进制末尾是0

二进制末尾是1

每个点自带属性t：前景0后景1
运算的时候直接ei^t

能量函数-调研

By Yu Qi

1、灰度L1梯度

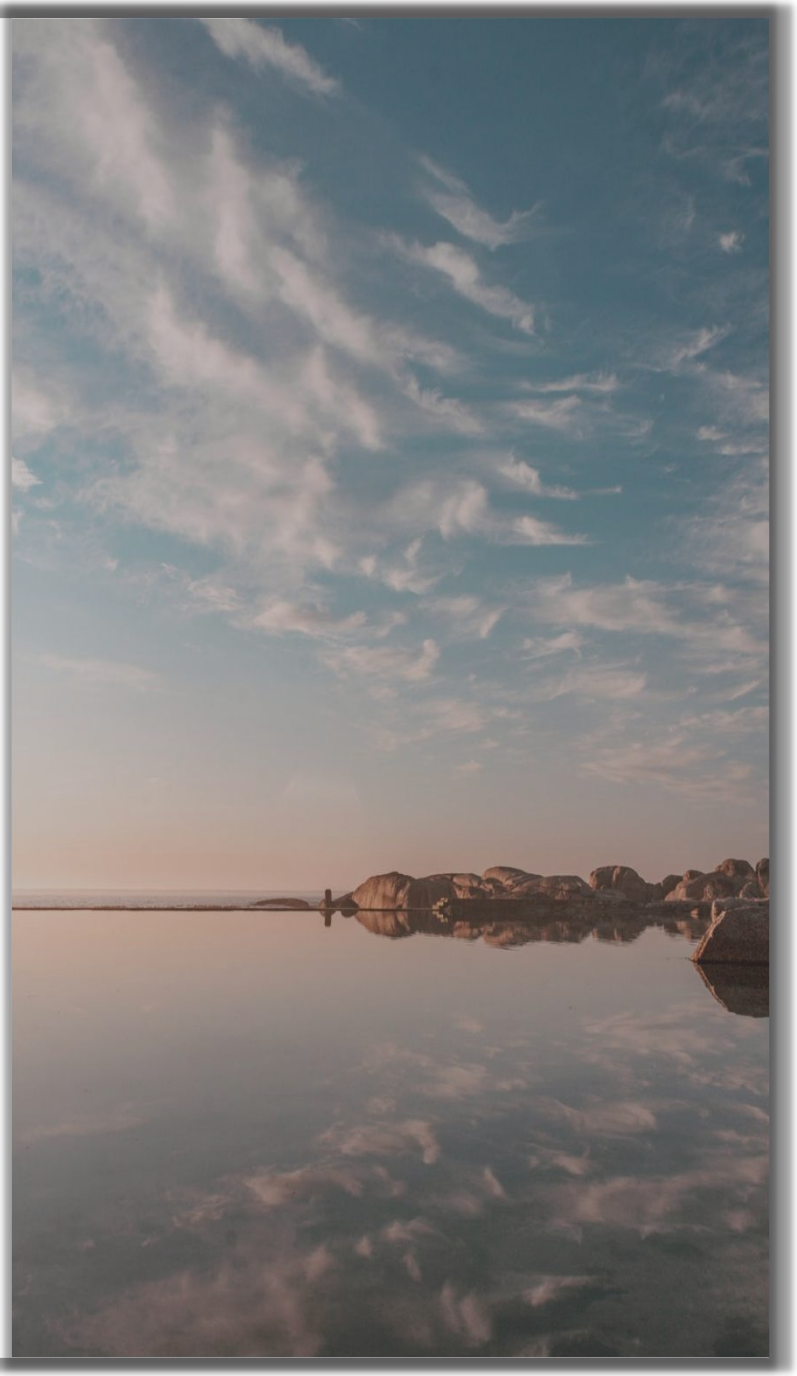
2、灰度L2T梯度

3、RGBL1梯度

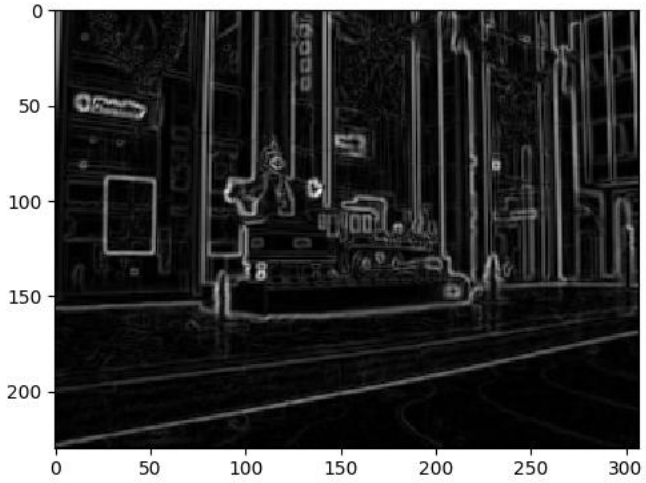
4、RGBL2梯度

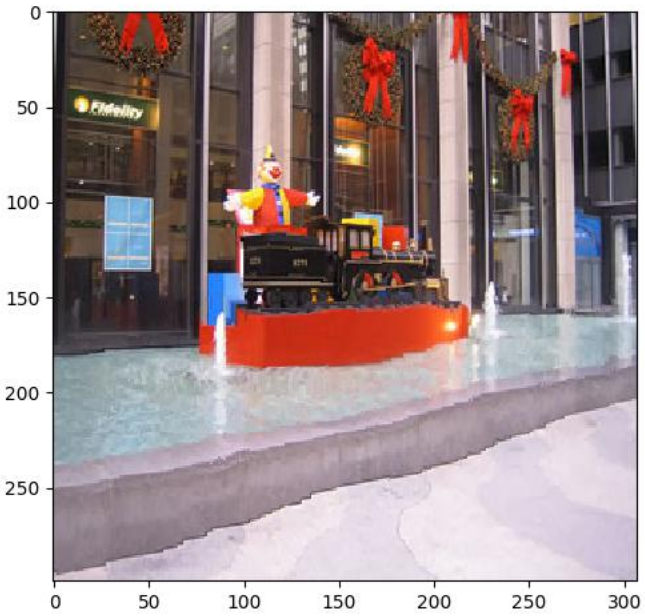
5、拉普拉斯算子

6、显著性区域检测



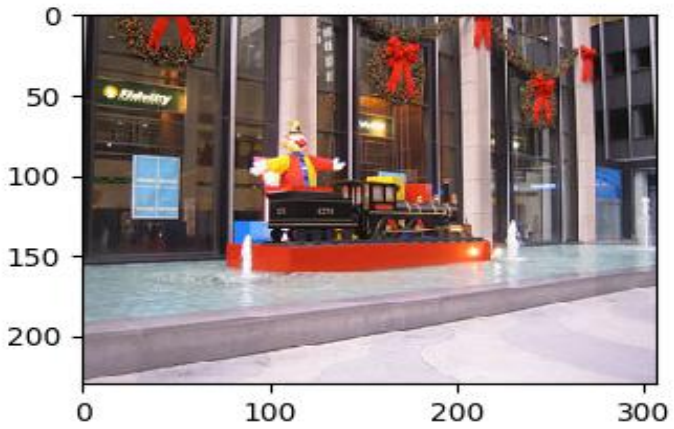
灰度L1梯度

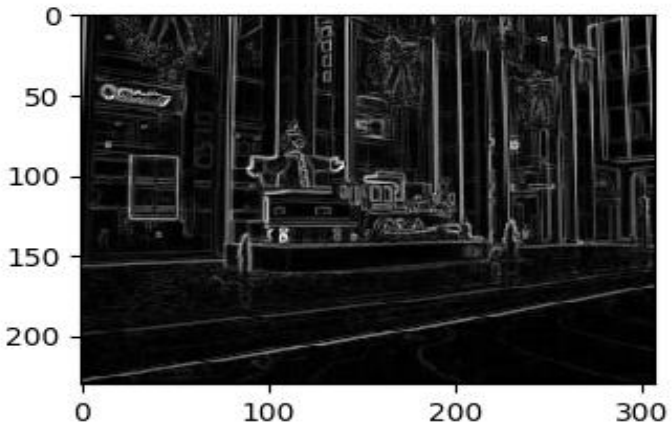


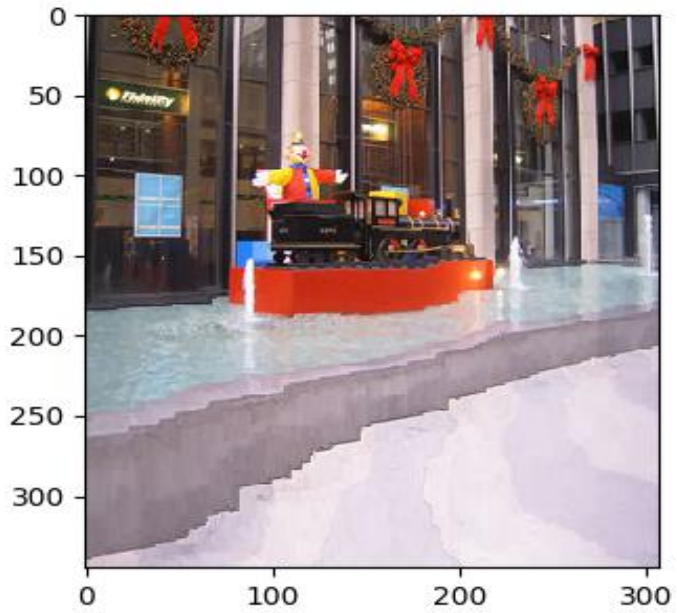



```
return cv2.add(np.absolute(dx), np.absolute(dy))    #对于数组中每个元素，求绝对值相加
```

RGB-L1梯度

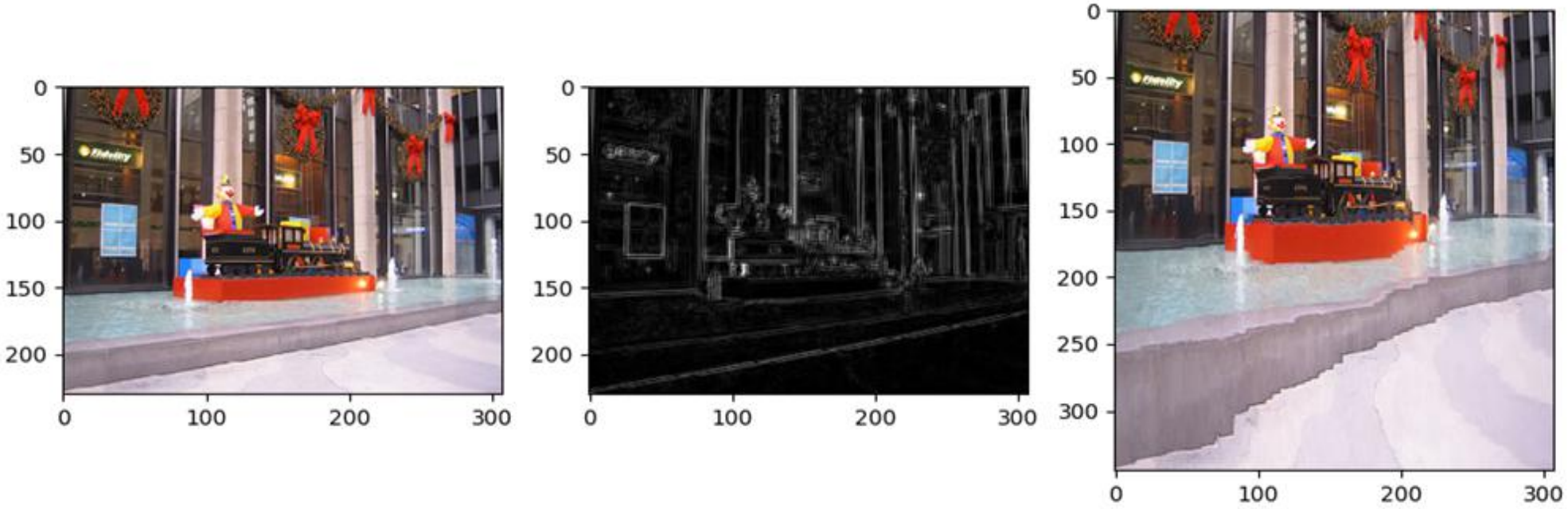






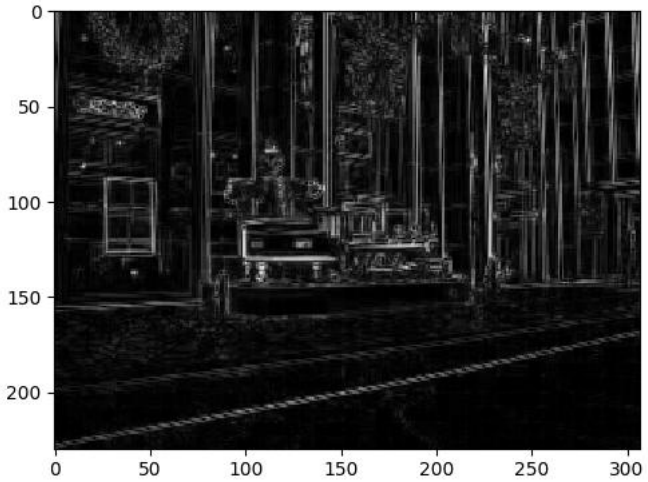

```
    return cv2.add(cv2.add(b_energy, g_energy), r_energy)     #之后分别对其L1算子求和
```

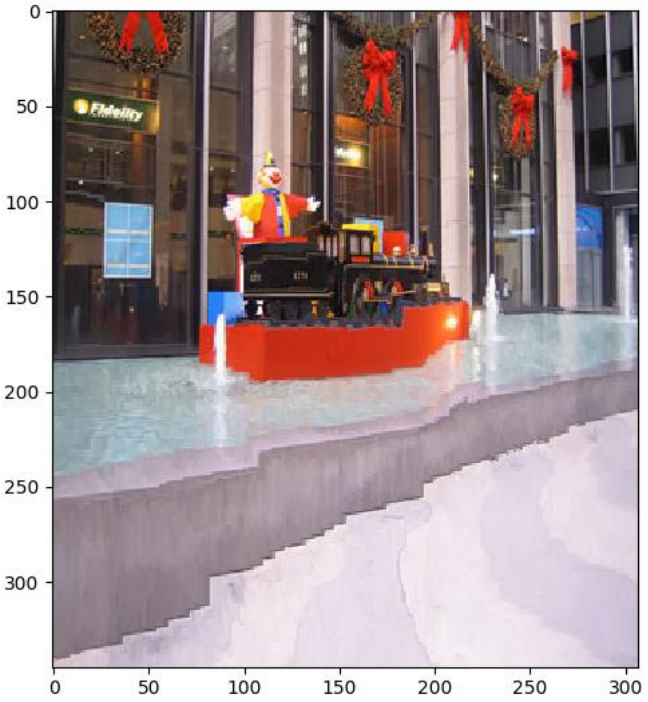
灰度L2梯度




```
    return cv2.add(np.absolute(dx), np.absolute(dy))  # 对于数组中每个元素，求绝对值相加
```

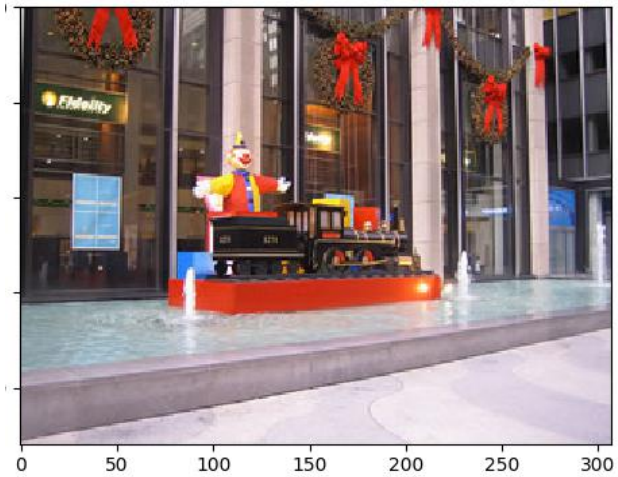
RGB-L2梯度

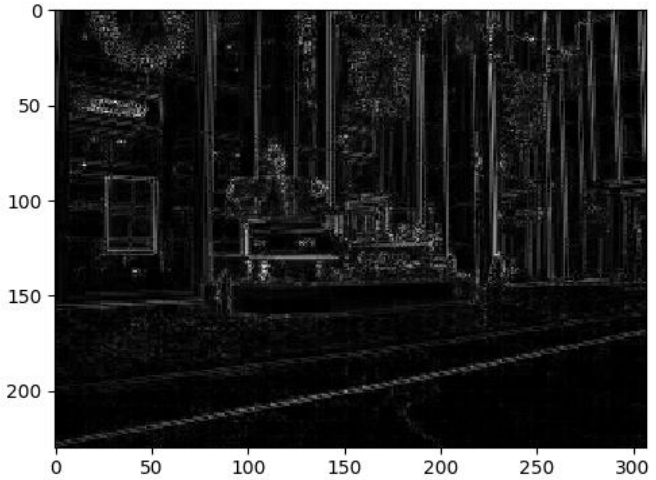


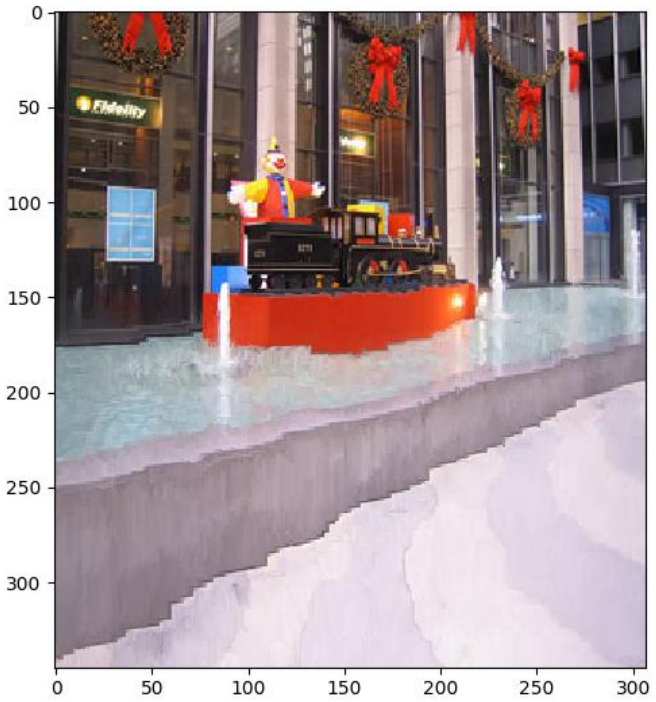



```
r_energy = np.absolute(cv2.Sobel(r, cv2.CV_64F, 2, 0)) + np.absolute(cv2.Sobel(b, cv2.CV_64F, 0, 2))
return cv2.add(cv2.add(b_energy, g_energy), r_energy)    #之后分别对其L2算子求和
```

拉普拉斯算子








```
    return cv2.add(cv2.add(b_energy, g_energy), r_energy)
```

显著性检测：LC检测

参考论文：Visual Attention Detection in Video Sequences Using Spatiotemporal Cues Yun Zhai and Mubarak Shah. Page 4-5

算法原理部分见论文的第四第五页。简单说就是计算某个像素在整个图像上的全局对比度，即该像素与图像中其他所有像素在颜色上的距离之和作为该像素的显著值。

当观众观看视频序列时，他们不仅会被有趣的事件所吸引，有时还会被静止图像中的有趣物体所吸引。这被称为空间注意力。根据心理学研究，人类感知系统对视觉信号的对比度很敏感，如颜色、强度和纹理。以此为基础假设，我们提出了一种使用图像颜色统计计算空间显着性图的有效方法。该算法的设计具有相对于图像像素数量的线性计算复杂度。图像的显着性图建立在图像像素之间的颜色对比度上。

The saliency value of a pixel I_k in an image I is defined as,

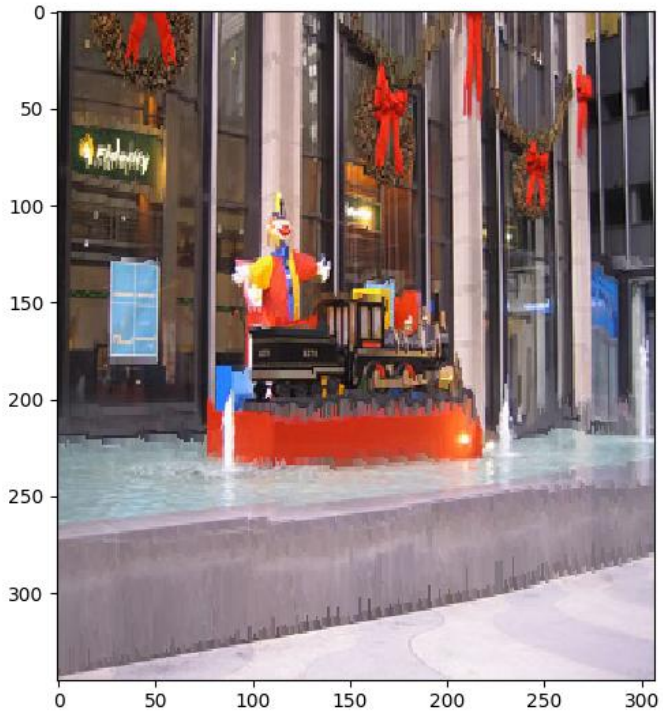
$$Sal(I_k) = \sum_{\forall Ii \in I} ||I_k - I_i||$$

Where the value of I_i is in the range of [0,255] and || * || represent the color distance metric

显著性检测：LC检测

参考论文：[Visual Attention Detection in Video Sequences Using Spatiotemporal Cues](#) Yun Zhai and Mubarak Shah. Page 4-5

```
def calc_dist(hist):        #计算hist与其他元素的RGB上的距离
```



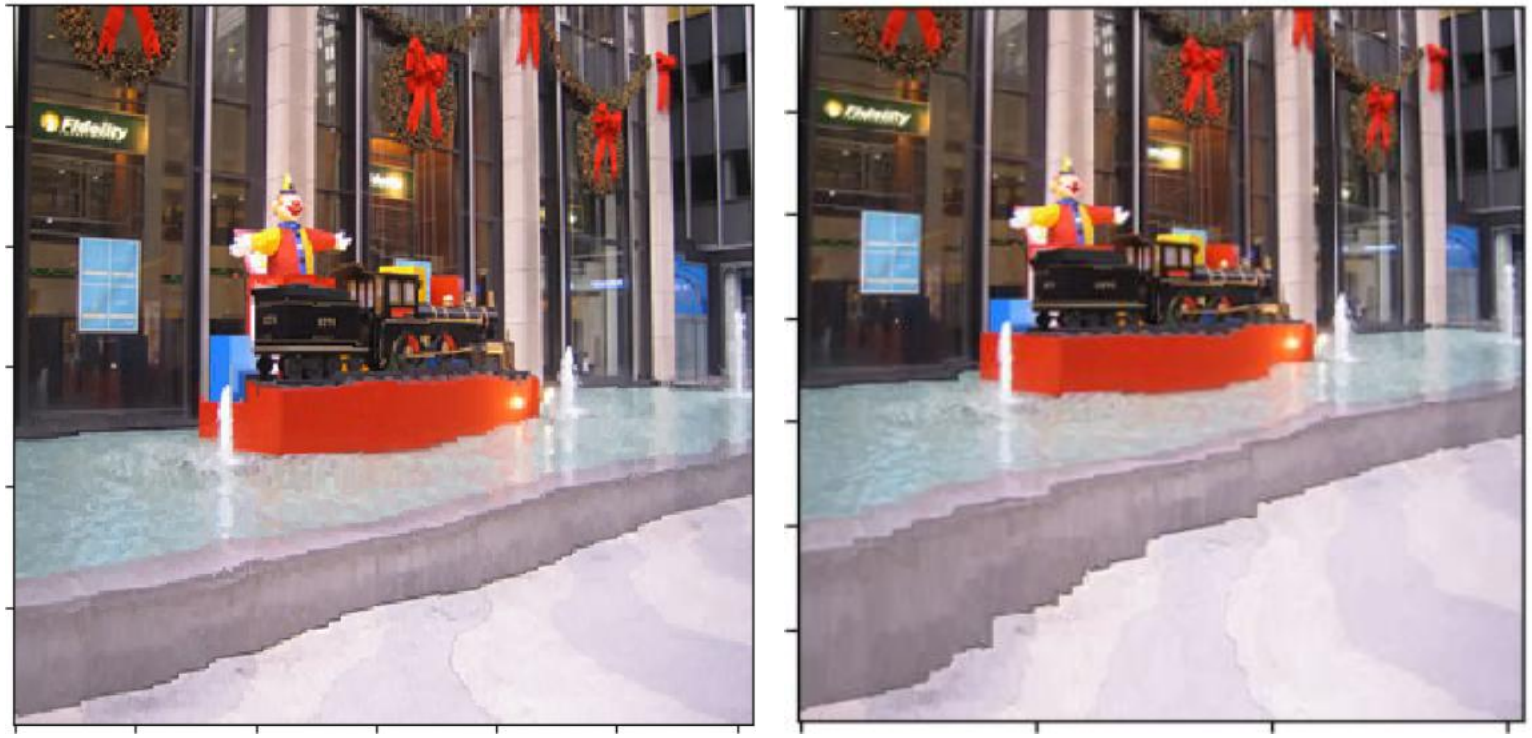
```
            image_gray_copy[j][i] = gray_dist[temp]     #image_gray_copy为对应灰度的gray_dist的值
    result = (image_gray_copy - np.min(image_gray_copy)) / (np.max(image_gray_copy) - np.min(image_gray_copy)
    return result   #返回值
```

对比分析

•L1算子，无论是灰度层面上还是rgb三个通道加和层面上，都能够比较好的保持原来的线条的形状，不至于弯曲变形

•拉普拉斯算子和L2，能够对横纵梯度比较大大和比较小的地方做到比较好的形状保存（如斜线或图片中红色底座），但是对线条的形状保存不好，容易弯曲变形

•LC显著性区域算子，对相应的显著性区域有比较好的保存效果，但是图像模糊化



对比分析

•L1算子，无论是灰度层面上还是rgb三个通道加和层面上，都能够比较好的保持原来的线条的形状，不至于弯曲变形

•拉普拉斯算子和L2，能够对横纵梯度比较大大和比较小的地方做到比较好的形状保存（如斜线或图片中红色底座），但是对线条的形状保存不好，容易弯曲变形

•LC显著性区域算子，对相应的显著性区域有比较好的保存效果，但是图像模糊化

对比分析

•L1算子，无论是灰度层面上还是rgb三个通道加和层面上，都能够比较好的保持原来的线条的形状，不至于弯曲变形

•拉普拉斯算子和L2，能够对横纵梯度比较大大和比较小的地方做到比较好的形状保存（如斜线或图片中红色底座），但是对线条的形状保存不好，容易弯曲变形

•LC显著性区域算子，对相应的显著性区域有比较好的保存效果，但是图像模糊化

成果展示 4

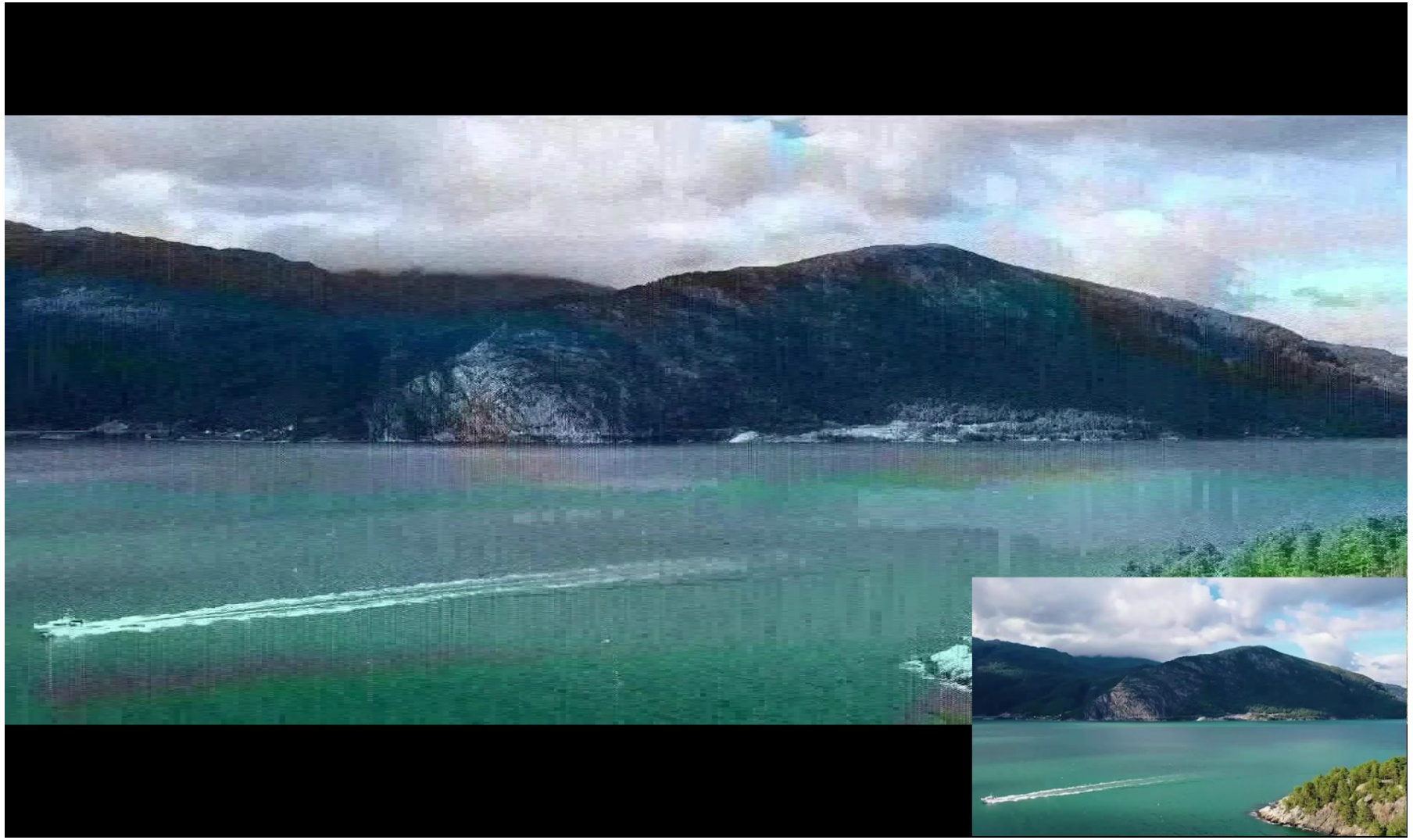
UI界面

By Yu Qi

视频中的Seam Carving

By Sizhe Li

Seam Carving在视频中的应用




THANKS
By 第二组