

Seam Carving

Seam Carving

问题描述

算法内容

Seam Carving for Context-Aware Image Resizing

Context-Aware Saliency Detection

实现细节

针对*SeamCarving*

针对*Context – Aware Saliency*

实验结果

采用提供的能量图和局部Context_Aware能量图的结果对比

原图500*324

扩大800*324

扩大500*600

双向扩大600*400

双向扩大800*600

裁剪300*324

裁剪500*200

裁剪300*200

能量图

原图512*512

缩小430*512

缩小430*430

放大630*512

放大630*630

缩小图片分辨率, 实验Context_Aware Saliency算法

原图128*128

能量图

放大180*128

放大180*180

缩小100*128

缩小100*100

注意事项

另外, vs自动给我的工程加了一个断点, 请删除所有断点后再编译运行程序。

问题描述

图片的缩放是日常生活中计算机所能提供的常用功能, 而简单的缩放有时并不能很好的满足人们的需求, 因为他们会认为图片的一些部分是特殊且重要的, 如果缩放时改变这些部分, 那么整张图片就会十分不自然。常见的例如人体, 动物, 火, 树木等都可能是图片中较为重要的信息。

因此, 如何确定删除或者增加的像素, 以及如何在确定后进行正确的删除或增添的操作, 是本次作业的重点问题

算法内容

Seam Carving for Context-Aware Image Resizing

$$\text{原论文定义了视觉显著性 } e(I) = \left| \frac{\partial I}{\partial x} \right| + \left| \frac{\partial I}{\partial y} \right|$$

而 *Seam* 就是图像中穿过较低视觉显著性区域的连线，比如竖直缝隙为 $s^x = \{x(i)\}_{i=1}^n$

其中 $\forall i, |x(i) - x(i-1)| \leq 1$

而最优缝隙就需要保证其能量和最小，即 $s^* = \min_s E(s) = \min_s \sum_{i=1}^n e(I(s_i))$

算法要求每次找到一条这样的缝隙，并删除，接着再找，直到图片大小与我们需求大小相同。

实际上，图片放大也可同理，找到能量和最小的一条缝隙并复制它，但是其问题在于，若放大后不修改能量图，则算法将一直选定那条“最优”缝隙进行复制，最终导致非常难看的效果。



一种可能的解决方法是，每次复制后，将该条缝隙和复制后缝隙的能量图位置能量取最大值，这样算法下次一定不会取该条线，但这个方法的问题是，当扩大一定多的像素后，能量图将不再有效，因为大部分被设置为最大值，导致本来应该保护的重要部分被判定到新的最优缝隙中进而被复制：



因此本次实验我选取了一种折衷的方法，给复制过的能量图部分能量值增加一个常数(15-30)，这样在多倍扩张图片时，原来图片的重要部分不会发现明显的变形。

Context-Aware Saliency Detection

该论文针对图片中各个部分的视觉显著性具体的计算，给出了一种较为可靠的方式。

首先定义两类距离：

$d_{color}(p_i, p_j)$ ，指的是两个像素在颜色空间的欧式距离规范化，即每一个通道取差，除以255，再计算最后得到的向量模，该模长取值为 $[0, 1]$

$d_{position}(p_i, p_j)$, 同理, 不过是二维的位置空间, 同时横坐标除以图像宽, 纵坐标除以图像高, 计算处理后的向量模, 取值也是 $[0, 1]$

最终定义像素差异度:

$$d(p_i, p_j) = \frac{d_{color}(p_i, p_j)}{1 + c * d_{position}(p_i, p_j)}$$

作者在实现时取 $c = 3$, 这也是我们实现算法时的取值。

接下来, 针对每一个像素, 计算其与图片中其他像素的差异度, 并对结果排序, 取前 K 个最小差异值的像素 $\{q_k\}_{k=1}^K$, 得到像素的显著值:

$$S_i^r = 1 - \exp\left\{-\frac{1}{K} \sum_{k=1}^K d(p_i^r, q_k^r)\right\}$$

接下来可以取不同大小的度量, 再进行上面的排序, 计算操作, 得到新的显著值指标, 最终将指标取平均, 得到图像各像素显著值的最终结果。

实现细节

针对SeamCarving

我们采取动态规划的方法获取最优缝隙。

取针对行方向的动态规划计算函数:

```
vector<int> cost(w * h);
int down, left, up;
int index;
for (int i = 0; i < h; i++)
    cost[i * w] = input.pixels[i * w]; // same value as Image read
for (int j = 1; j < w; j++)
{
    index = min(cost[j - 1], cost[w + j - 1]); // if j-1 < w+j-1 return
    // this is the smallest
    cost[j] = cost[index * w + j - 1] + input.pixels[j]; // fill first
row
    for (int i = 1; i < h - 1; i++)
    {
        up = cost[(i - 1) * w + j - 1]; // rightup
        left = cost[i * w + j - 1]; // right
        down = cost[(i + 1) * w + j - 1]; // rightdown
        index = min(up, left, down);

        cost[i * w + j] = cost[(i + index) * w + j - 1] + input.pixels[i
        * w + j]; // fill j th col
    }
    index = min(cost[(h - 2) * w + j - 1], cost[(h - 1) * w + j - 1]);
    cost[(h - 1) * w + j] = cost[(h - 2 + index) * w + j - 1] +
    input.pixels[(h - 1) * w + j];
}
return cost;
```

用一个vector容器, 将图像每个像素点的能量值按行优先存储进去, 成为上面返回的cost。

首先, 将最左侧的一列赋上能量值, 接着, 从第二列开始, 对每个新的位置, 由于裂缝路径的约束, 取其左上方, 左方, 左下方的cost值比较, 取最小的, 加上自己的能量值后, 填入自己的cost中。

那么如何找到最优缝隙呢:

```

vector<int> cost = this->Energy_Cost_Row(input);
vector<int> index(w);

int currRow = 0;
for (int i = 1; i < h; i++)
{
    if (cost[(i + 1) * w - 1] < cost[(currRow + 1) * w - 1]) // from
the rightest to calculate
        currRow = i;
}
index[w - 1] = currRow;
int pathFinder = cost[(currRow + 1) * w - 1] - input.pixels[(currRow +
1) * w - 1];
for (int j = w - 2; j >= 0; pathFinder = cost[currRow * w + j] -
input.pixels[currRow * w + j], j--)
{
    if (currRow == 0) // if it is the first row
    {
        if (pathFinder == cost[j])
        {
            index[j] = currRow; //left
        }
        else
        {
            currRow += 1; //down left
            index[j] = currRow;
        }
        continue;
    }
    if (pathFinder == cost[currRow * w + j])
    {
        index[j] = currRow; //left
        continue;
    }
    else if (pathFinder == cost[(currRow - 1) * w + j])
    {
        currRow -= 1; //up left
        index[j] = currRow;
    }
    else
    {
        currRow += 1; //down left
        index[j] = currRow;
    }
}
return index;

```

当要选取最优缝隙时，只需要从最右一列的cost中选取最小的值，即可找到起点，因为路径信息已经被隐含在其中：

选取该点cost值，减去该点能量值，将得到的值与左上方，左方，左下方的cost值比较，哪个符合，就代表路径走哪里，一直反复直到复原整条最优缝隙。

最后返回的index正是对应缝隙的每一个位置的纵坐标，交由删除或扩张函数来对图片本身做实质性的改变。

针对Context – Aware Saliency

其实现起来并不困难，主要问题在于计算量太过庞大，可能需要一些库的支持以提升计算速度。

```
double color_distance(MyImage input, int row1, int col1, int row2, int col2)
double distance(MyImage input, int row1, int col1, int row2, int col2)
```

以上两个函数的实现，和我们对算法讲解时的描述完全相同

而算法的主要耗时来自下面的函数：

```
double salient(MyImage input, int r, int c)
{
    int rows = input.height();
    int cols = input.width();
    vector<double> diffs;
    int count = 0;
    for (int row = 0; row <= rows; ++row)
    {
        for (int col = 0; col <= cols; ++col)
        {
            diffs.push_back(distance(input, r, c, row, col));
            count++;
        }
    }
    sort(diffs.begin(), diffs.end());
    double sum = 0;
    int n = 0;
    for (n = 0; n <= 64 && n < diffs.size(); ++n) // k = 64
        sum += diffs[n];
    return 1 - exp(-sum / n);
}
```

论文中提到了，要对所有点差异度排序，再取最小的K个，带入函数进行计算，得到一个像素点的显著值。以一个普通图像256*256的像素来计算，最少需要 $255!/2$ 次计算才能得到所有像素点的差异度，更别提还需要排序等操作。

因此我采用了一取局部的方法，减少时间复杂度，但相应的，整个显著值能量图结果并不能达到最优

```
for (int row = r - 2; row <= r + 2; ++row)
{
    if (row < 0 || row >= rows)
        continue;
    for (int col = c - 2; col <= c + 2; ++col)
    {
        if (col < 0 || col >= cols)
            continue;
        diffs += (distance(input, r, c, row, col));
        count++;
    }
}
```

此外，还参照论文的取不同范围像素，获得不同的显著度值取平均的方法。尝试对结果进行优化。

遗憾的是，程序执行时间实在太长，针对boy.png，程序运行了近40分钟才得到了能量图。



针对像素更多的图，运行了近一个小时。

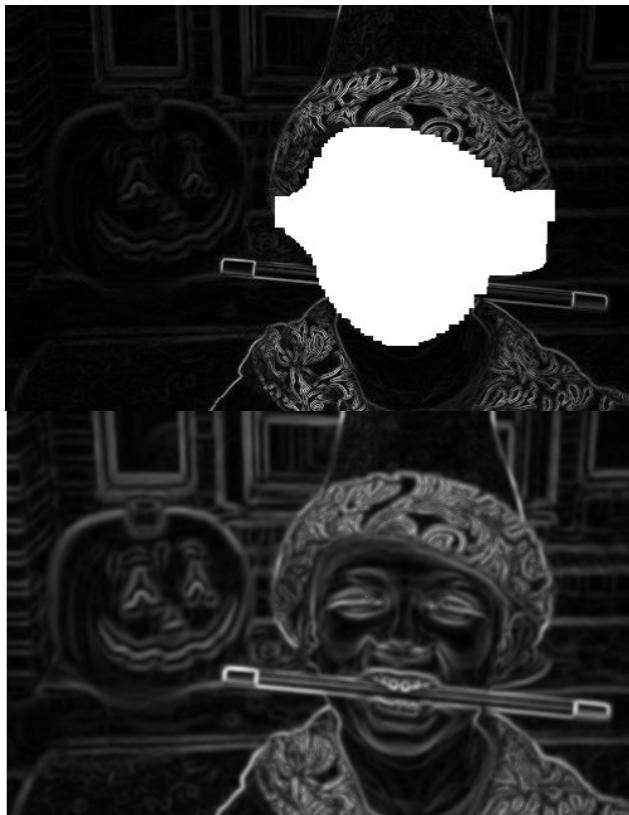


可以看到，取局部导致针对人脸的显著度计算有相当的误差。

实验结果

采用提供的能量图和局部Context_Aware能量图的结果对比

左为提供能量图结果，右为局部CA能量图结果



原图500*324



扩大800*324





扩大500*600



双向扩大600*400



双向扩大800*600



裁剪300*324



裁剪500*200



裁剪300*200



在这样的对比中，局部Context_Aware的表现并不尽如人意，因为实现以及未优化的问题，其效果不及论文的表现。

接下来针对其他图片尝试局部Context_Aware算法

能量图



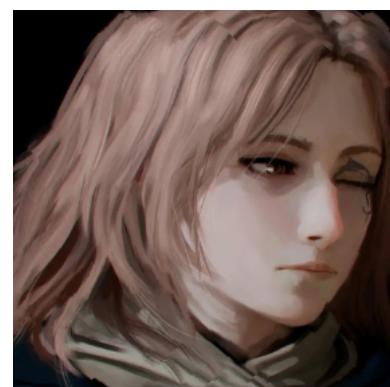
原图512*512



缩小430*512



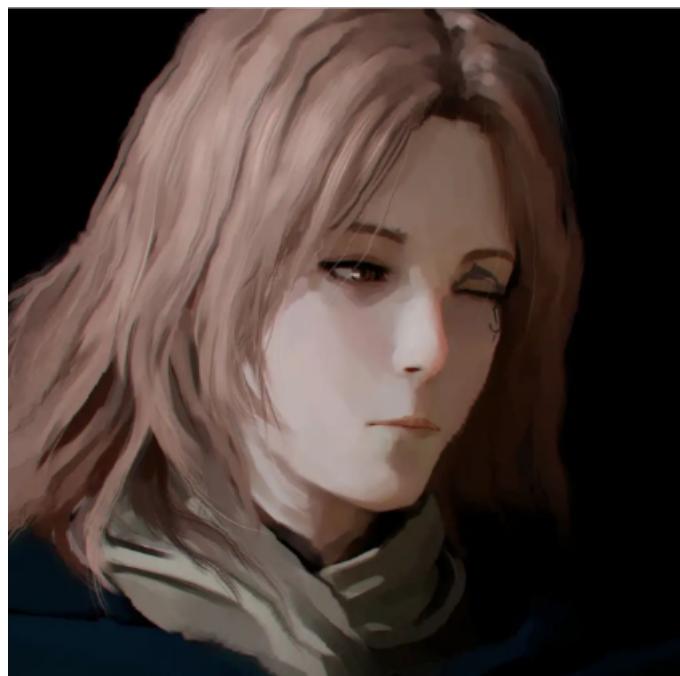
缩小430*430



放大630*512



放大630*630



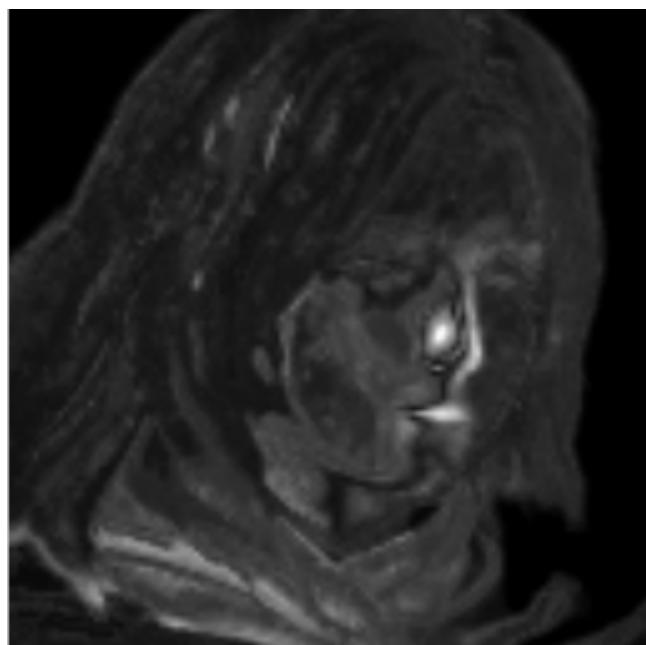
缩小图片分辨率，实验Context_Aware Saliency算法

因为计算量大，缩小图片后，在一定时间内还是可以得到结果，可以看到这次求出的能量图比较类似论文的效果：

原图128*128



能量图



放大180*128



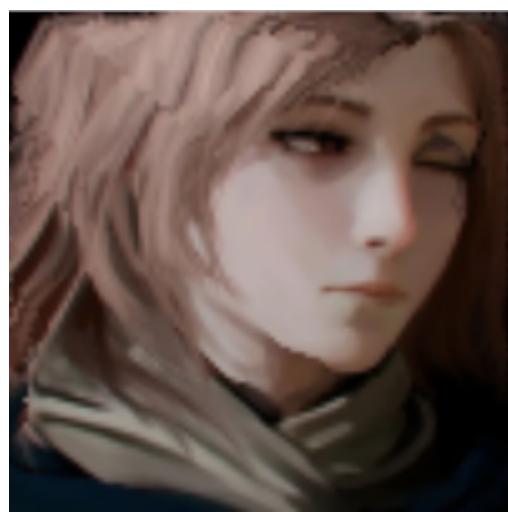
放大180*180



缩小100*128



缩小100*100



注意事项

程序用CA计算出的能量图以*Context_Aware.png*保存

在运行时应先点击 **Read Image** 

如果要用默认能量图，点击 **Read Energy Image** 

计算时输入好分辨率，点击 **Run Seam Carving**  即可执行

如果要用CA能量图，点击 **See Context-Aware Saliency** 

输入分辨率后直接点击 **Run Seam Carving**  或

Use Context-Aware to Calculate  均可

但是最好不要勾选 **Use Context-Aware Saliency..** ，这意味着针对输入图片计算能量图，会花费特别长的时间

实验根目录提供了三张用CA算好的能量图，已在报告中展示。

另外，vs自动给我的工程加了一个断点，请删除所有断点后再编译运行程序。