实验名称：image morphing

实验目的：利用 imagemorphing 计算图像像素点之间的插值，得到“中间图像”

实验要求：

根据 Image Morphing 的方法完成给定的两个输入图像中间 11 帧的差值，得到一个 Image Morphing 的动画视频。

实验内容：

1. 将原图像和目标图像调整为同样的 size
2. 手动采样两张图片的特征点
3. 三角形剖分
4. 找出 11 个不同加权的“中间图像”
5. 保存，制作 gif

1. 只需调用 cimg 的 resize 函数即可实现对图像的大小调整
2. 这个比较费时费力，但是不费脑力。需要小心一点，不要搞错。
3. 三角形剖分的部分代码出于 github，将编译错误改了几下，能够直接使用，使用后对照伪代码也弄懂了怎么实现的。
4. 设 α 从 0-1.0 以 1/（frameNumber+1）的步长递增，使得中间图像的权重从原图到目标图渐变。FrameNumber 是中间图像的个数。
5. 将得到的一系列中间图像加上原图像和目标图像一共 13 张图像保存只作为 gif 图。

重要代码：

三角剖分：

```
const std::vector<TriangleType>& triangulate(std::vector<VertexType> &vertices)
{
    // Store the vertices locally
    _vertices = vertices;

    // Determinate the super triangle
    double minX = vertices[0].x;
    double minY = vertices[0].y;
    double maxX = minX;
    double maxY = minY;

    for(std::size_t i = 0; i < vertices.size(); ++i)
    {
        if (vertices[i].x < minX) minX = vertices[i].x;
        if (vertices[i].y < minY) minY = vertices[i].y;
        if (vertices[i].x > maxX) maxX = vertices[i].x;
        if (vertices[i].y > maxY) maxY = vertices[i].y;
    }

    const double dx = maxX - minX;
    const double dy = maxY - minY;
    const double deltaMax = std::max(dx, dy);
    const double midx = half(minX + maxX);
    const double midy = half(minY + maxY);

    const VertexType p1(midx - 20 * deltaMax, midy - deltaMax);
    const VertexType p2(midx, midy + 20 * deltaMax);
    const VertexType p3(midx + 20 * deltaMax, midy - deltaMax);

    //std::cout << "Super triangle " << std::endl << Triangle(p1, p2, p3) << std::endl;

    // Create a list of triangles, and add the supertriangle in it
    _triangles.push_back(TriangleType(p1, p2, p3));
```

设置超级三角形，并将超级三角形加入队列中。

```
for(auto p = begin(vertices); p != end(vertices); p++)
{
    //std::cout << "Traitement du point " << *p << std::endl;
    //std::cout << "_triangles contains " << _triangles.size() << " elements" << std::endl;

    std::vector<EdgeType> polygon;

    for(auto & t : _triangles)
    {
        //std::cout << "Processing " << std::endl << *t << std::endl;

        if(t.circumCircleContains(*p))
        {
            //std::cout << "Pushing bad triangle " << *t << std::endl;
            t.isBad = true;
            polygon.push_back(t.e1);
            polygon.push_back(t.e2);
            polygon.push_back(t.e3);
        }
        else
        {
            //std::cout << " does not contains " << *p << " in his circum center" << std::endl;
        }
    }

    _triangles.erase(std::remove_if(begin(_triangles), end(_triangles), [](TriangleType &t){
        return t.isBad;
    }), end(_triangles));
```

将点逐个插入

检测外接圆内部有插入点的圆，将该圆标记为非 delaunay 三角形并从队列中删除。

将非 delaunay 三角形的边标记为影响边。

```
    for(auto e1 = begin(polygon); e1 != end(polygon); ++e1)
    {
        for(auto e2 = e1 + 1; e2 != end(polygon); ++e2)
        {
            if(almost_equal(*e1, *e2))
            {
                e1->isBad = true;
                e2->isBad = true;
            }
        }
    }

    polygon.erase(std::remove_if(begin(polygon), end(polygon), [](EdgeType &e){
        return e.isBad;
    }), end(polygon));
```

将同时属于两个或以上非 delaunay 三角形的边标记为即将删除的边并将其删除。

```
for(const auto e : polygon)
    _triangles.push_back(TriangleType(e.p1, e.p2, *p));
```

将未被删除的边和插入点构成三角形加入三角形队列中。

直到所有的点插入完毕。删除和超级三角形有公共点的三角形即得到完备的 delaunay 三角形网络。

```
    _triangles.erase(std::remove_if(begin(_triangles), end(_triangles), [p1, p2, p3](TriangleType &t){
        return t.containsVertex(p1) || t.containsVertex(p2) || t.containsVertex(p3);
    }), end(_triangles));

    for(const auto t : _triangles)
    {
        _edges.push_back(t.e1);
        _edges.push_back(t.e2);
        _edges.push_back(t.e3);
    }

    return _triangles;
```

重要代码分析 2：image morphing

```
vector<CImg<unsigned char>> morphing::morph(int frames)
{

    vector<CImg<unsigned char>> resImgVec;
    //detect feature;
    //triangle depart
    initSrcTriangle();
```

返回结果 resImgVec，进行三角剖分。特征提取人工操作的。

```
double steps = 1.0 / (frames+1);
for (double i = 0; i <= 1.0; i+= steps) {
    //CImg<unsigned char> mid(source.width(),source.height(),1,3);
    CImg<unsigned char> mid(source);
    myPoint midPoint[39];
    for (int p = 0; p < 39; p++) {
        midPoint[p].x = (1 - i)*srcPoint[p].x + i * tarPoint[p].x;
        midPoint[p].y = (1 - i)*srcPoint[p].y + i * tarPoint[p].y;
        /*unsigned int color[3] = { 255,0,0 };
        mid.draw_circle(midPoint[p].x, midPoint[p].y, 3, color);*/
    }
```

将循环分为 frameNumber+2 个，分别为 0*steps,steps,2*steps,...,frameNumber*steps,1.0.初始化一个中间图像并做中间特征点的插值。

```
for (int j = 0; j < srcTriangle.size(); j++) {
    //break;
    auto srcTran = srcTriangle[j];
    int index1 = getPointIndex(myPoint{(int)srcTran.p1.x,(int) srcTran.p1.y});
    int index2 = getPointIndex(myPoint{ (int)srcTran.p2.x,(int)srcTran.p2.y });
    int index3 = getPointIndex(myPoint{ (int)srcTran.p3.x,(int)srcTran.p3.y });
    auto srcpoint1 = srcTran.p1;
    auto srcpoint2 = srcTran.p2;
    auto srcpoint3 = srcTran.p3;
    auto midpoint1 = midPoint[index1];
    auto midpoint2 = midPoint[index2];
    auto midpoint3 = midPoint[index3];
    auto tarpoint1 = tarPoint[index1];
    auto tarpoint2 = tarPoint[index2];
    auto tarpoint3 = tarPoint[index3];

    double a = 1.0 / maxEdge(midpoint1, midpoint2, midpoint3);
    for (double p = 0; p < 1; p += a) {
        for (double q = 0; q < 1 - p; q += a) {
            int mid_x = p * midpoint1.x + q * midpoint2.x + (1 - p - q)*midpoint3.x;
            int mid_y = p * midpoint1.y + q * midpoint2.y + (1 - p - q)*midpoint3.y;
            double mu, v;
            myPoint midpoint{ mid_x,mid_y };
            calculate_mu_v(midpoint1, midpoint2, midpoint3, midpoint, mu, v);
            int src_x = (1 - mu - v)*srcpoint1.x + v * srcpoint2.x + mu * srcpoint3.x;
            int src_y = (1 - mu - v)*srcpoint1.y + v * srcpoint2.y + mu * srcpoint3.y;

            int tar_x = (1 - mu - v)*tarpoint1.x + v * tarpoint2.x + mu * tarpoint3.x;
            int tar_y = (1 - mu - v)*tarpoint1.y + v * tarpoint2.y + mu * tarpoint3.y;

            mid(mid_x, mid_y, 0) = (1 - i)*source(src_x, src_y, 0) + i * target(tar_x, tar_y, 0);
            mid(mid_x, mid_y, 1) = (1 - i)*source(src_x, src_y, 1) + i * target(tar_x, tar_y, 1);
            mid(mid_x, mid_y, 2) = (1 - i)*source(src_x, src_y, 2) + i * target(tar_x, tar_y, 2);
        }
    }
    //auto midTran = Triangle(midPoint[getPointIndex(srcTran.p1)])
}
```

对中间图像的每一个三角形里的每一个像素点，求出其对应的原图像的像素点的位置和目标像素点的位置，用 i 做插值求出中间图像相应位置的像素值。

```
                    mid(mid_x, mid_y, 2) = (1 - l)*source(src_x, src_y, 2) + l * target(tar_x, tar_y, 2);
                }
            }
            //auto midTran = Triangle(midPoint[getPointIndex(srcTran.p1)])

        }
        resImgVec.push_back(mid);
    }
    //resImgVec.push_back(CImg<unsigned char>(target));
    return resImgVec;
}
```

将中间图像加入返回容器。最终将结果返回。

实验结果展示：

详情见 output 文件夹.