HW3: report

> 姓名：杜朋澈
>
> ID：68
>
> 学号：PB21050988

算法

问题描述

如上两幅图，现我们需要将Figure 3.1中的女孩搬到Figure 3.2的海水中，为使得复制粘贴更加逼真自然，我们需要设计算法来满足我们两幅图像融合的需要。

Poisson Image Editing

Poisson Image Editing算法的基本思想是在尽可能保持原图像内部梯度的前提下，让粘贴后图像的边界值与新的背景图相同，以实现无缝粘贴的效果。从数学上讲，对于原图像$f(x,y)$，新背景$f^*(x,y)$和嵌入新背景后的新图像$v(x,y)$，等价于解最优化问题：

$$\min_f \iint_\Omega |\nabla f - \nabla \boldsymbol{v}|^2 \ \text{ with } f|_{\partial\Omega} = f^*|_{\partial\Omega}$$

利用变分法可转化为具有Dirichlet边界条件的Poisson方程：

$$\Delta f = \Delta \boldsymbol{v} \text{ over } \Omega \ \text{ with } f|_{\partial\Omega} = f^*|_{\partial\Omega}$$

以Figure 3.1和Figure 3.2为例，将Figure 3.1中需要复制的区域设为S，定义N_p为S中的每一个像素p四个方向连接邻域，令$< p, q >$为满足$q \in N_p$的像素对。边界Ω定义为

$$\partial\Omega = p \in S \setminus \Omega : N_p \cap \Omega \neq \emptyset$$

设f_p为p处的像素值f，目标即求解像素值集$f|_\Omega = f_p, p \in \Omega$。

利用Poisson Image Editing算法的基本原理，上述问题转化为求解最优化问题：

$$\min_{f|_\Omega} \sum_{<p,q>\cap\Omega\neq\emptyset} (f_p - f_q - v_{pq})^2, \text{with } f_p = f_p^*, \text{for all} p \in \partial\Omega$$

化为求解线性方程组：

$$\text{for all } p \in \Omega,\ |N_p| f_p - \sum_{q\in N_p\cap\Omega} f_q = \sum_{q\in N_p\cap\partial\Omega} f_p^* + \sum_{q\in N_p} v_{pq}$$

对于梯度场$\boldsymbol{v}(\boldsymbol{x})$的选择，文献给出两种方法，一种是完全使用前景图像的内部梯度，即：

$$\text{for all } < p, q >, v_{pq} = g_p - g_q$$

另一种是使用混合梯度：

$$\text{for all } \boldsymbol{x} \in \Omega,\ \boldsymbol{v}(\boldsymbol{x}) = \begin{cases} \nabla f^*(\boldsymbol{x}) & \text{if } |\nabla f^*(\boldsymbol{x})| > |\nabla g(\boldsymbol{x})|, \\ \nabla g(\boldsymbol{x}) & \text{otherwise} \end{cases}$$

实现

predecomposer

矩阵预分解工具。在选区完成时既完成A的填充和分解。

- 接口

```
class Predecomposer
{
   public:
    Predecomposer(std::shared_ptr<USTC_CG::Image> mask) : mask_(mask){}
    ~Predecomposer() = default;

    void solve();

   public:
    std::shared_ptr<USTC_CG::Image> mask_;

    Eigen::SparseMatrix<double> A_;

    // pos to index map
    std::unordered_map<std::pair<int, int>, int, pair_hash> index_map;
    // index to neighbors pos map
    std::unordered_map<int, std::vector<std::pair<int, int>>> neighbor_map;
    // index to borders pos (exclude, if exists) map
    std::unordered_map<int, std::vector<std::pair<int, int>>> border_map;

    Eigen::SimplicialLLT<Eigen::SparseMatrix<double>> solver;
};
```

- 核心算法`solve()`实现及解释

```cpp
void Predecomposer::solve()
{
    //根据编号索引构造index map
    //初始化border map
    int counter = 0;
    for (int i = 0; i < mask_->width(); ++i)
    {
        for (int j = 0; j < mask_->height(); ++j)
        {
            if (mask_->get_pixel(i, j)[0] > 0)
            {
                index_map.emplace(std::pair<int, int>(i, j), counter);
                border_map.emplace(
                    counter, std::vector<std::pair<int, int>>(0));
                counter++;
            }
        }
    }
    A_.resize(counter, counter);

    //构造A矩阵
    for (int i = 0; i < mask_->width(); ++i)
    {
        for (int j = 0; j < mask_->height(); ++j)
        {
            //对于掩码中的每个像素
            if (mask_->get_pixel(i, j)[0] > 0)
            {
                int idx = index_map[std::pair<int, int>(i, j)];
                int neighbor_count = 0;
                std::vector<std::pair<int, int>> near = {
                    { i - 1, j }, { i + 1, j }, { i, j - 1 }, { i, j + 1 }
                };
                //遍历(i, j)4个邻居节点
                for (const auto& n : near)
                {
                    int near_x = n.first;
                    int near_y = n.second;
                    // 如果没有超出图像范围
                    if (0 <= near_x && near_x < mask_->width() && 0 <= near_y &&
                        near_y < mask_->height())
                    {
                        //将邻居节点加入该节点(i, j)的邻居索引中
                        neighbor_map[idx].push_back(
                            std::pair<int, int>(near_x, near_y));
                        //如果没有超出掩码范围, 调整矩阵A对应条目
                        if (mask_->get_pixel(near_x, near_y)[0] > 0)
                            A_.coeffRef(idx, index_map[{ near_x, near_y }]) =
                                -1;
                        //如果超出范围，将邻居节点加入该节点(i, j)的边界索引中
                        else
                            border_map[idx].push_back(
                                std::pair<int, int>(near_x, near_y));
                        ++neighbor_count;
                    }
                }
                A_.coeffRef(idx, idx) = neighbor_count;
            }
        }
    }
    A_.makeCompressed();
    solver.compute(A_);
}
```

SeamlessCloner

利用预分解信息进行实时求解。

- 接口

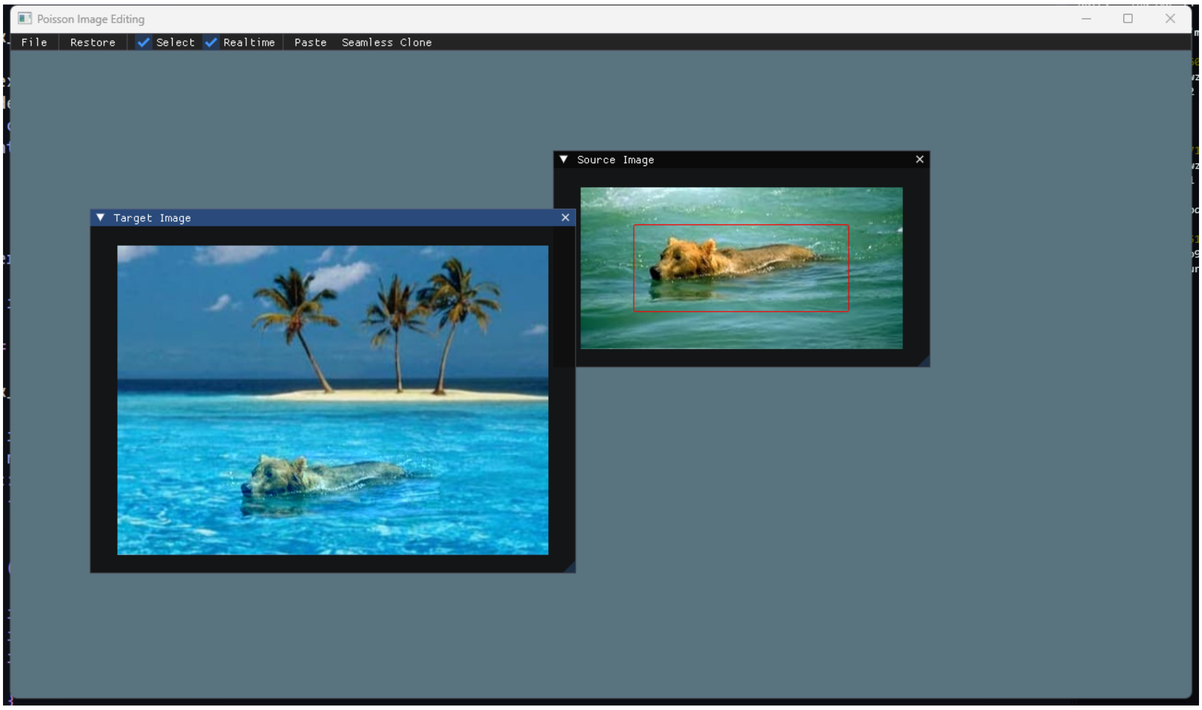
```
#pragma once

#include "comp_source_image.h"
#include "predecomposer.h"

class SeamlessCloner
{
   public:
    SeamlessCloner() = default;
    ~SeamlessCloner() = default;

    void set_decomposer(std::shared_ptr<Predecomposer> decomposer);
    void set_target_image(std::shared_ptr<USTC_CG::Image> dst_image);
    void set_source_image(std::shared_ptr<USTC_CG::Image> src_image);
    void set_offset(int offset_x, int offset_y);

    void solve();

    std::vector<unsigned char> get_pixel(int i, int j, int channels);
```

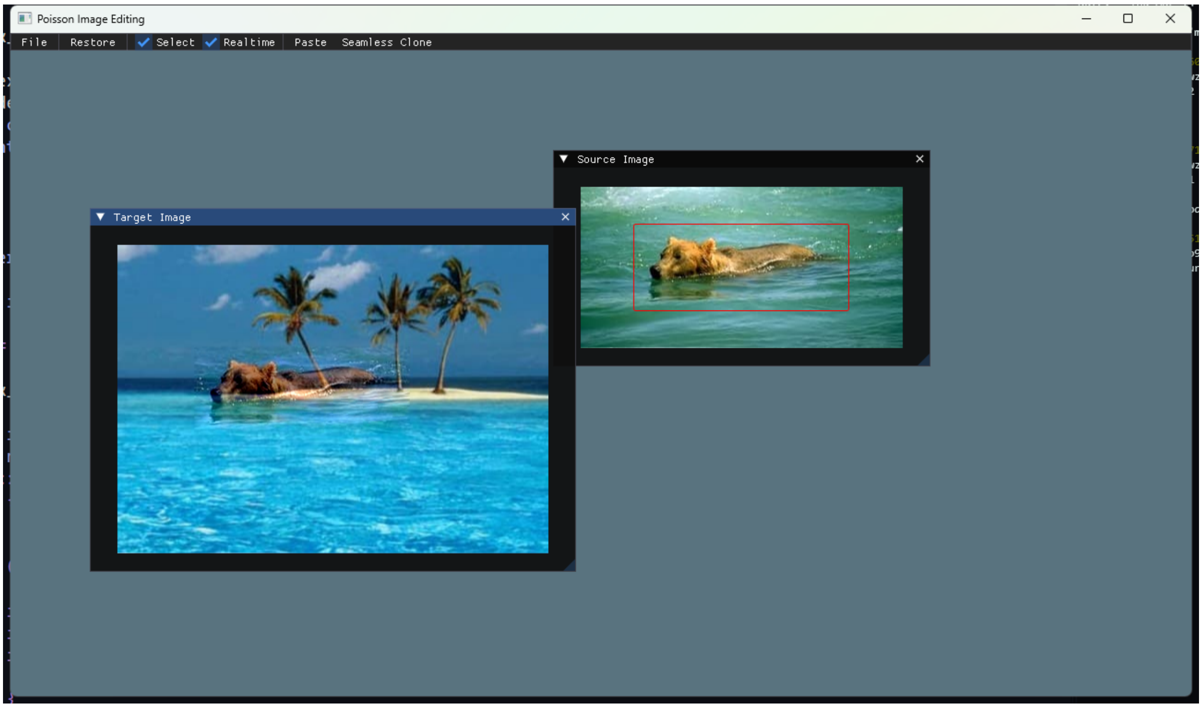
```cpp

   private:
    inline double get_gradient_mix(double f_p, double f_q, double g_p, double g_q)
    {
        double v_pq_f = f_p - f_q;
        double v_pq_g = g_p - g_q;
        return (v_pq_f * v_pq_f) > (v_pq_g * v_pq_g) ? v_pq_f : v_pq_g;
    }

   private:
    inline bool check_valid_range(int i, int j)
    {
        return i >= dst_image_->width() ||
            j >= dst_image_->height();
    }

   private:
    int offset_x_;
    int offset_y_;
    std::shared_ptr<USTC_CG::Image> src_image_;
    std::shared_ptr<USTC_CG::Image> dst_image_;
    std::shared_ptr<Predecomposer> decomposer_;
    std::vector<Eigen::VectorXd> color_vec;
};
```

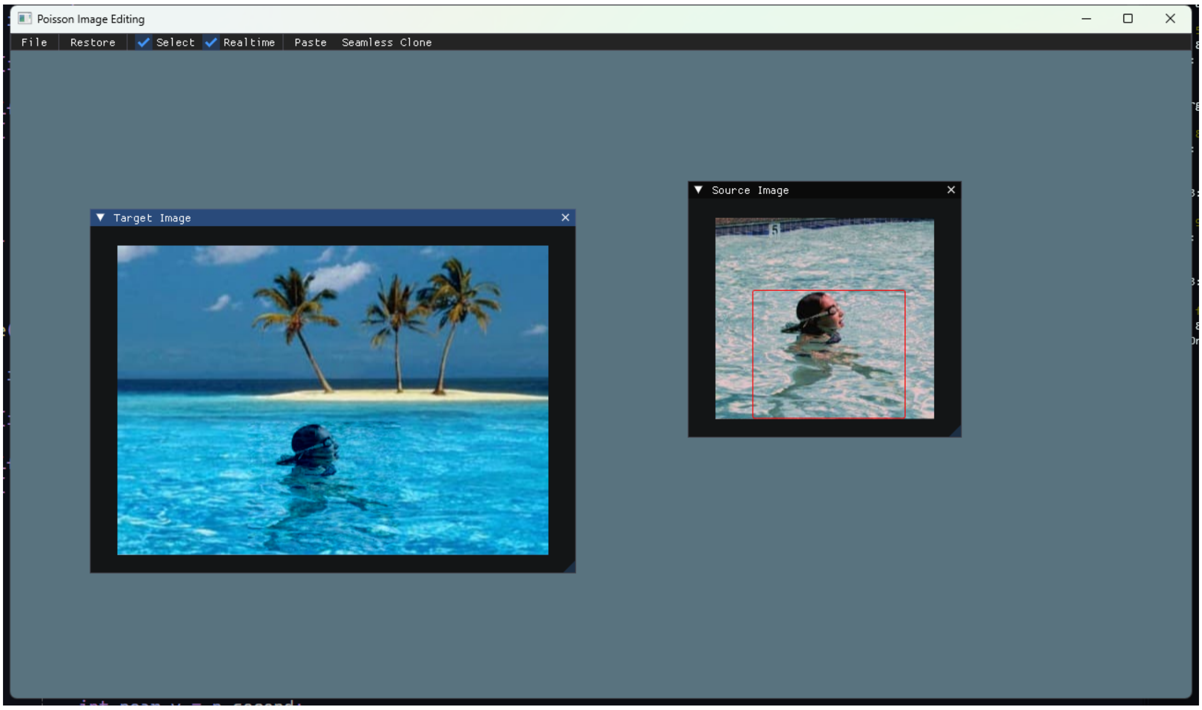
- 核心算法solve()实现及解释

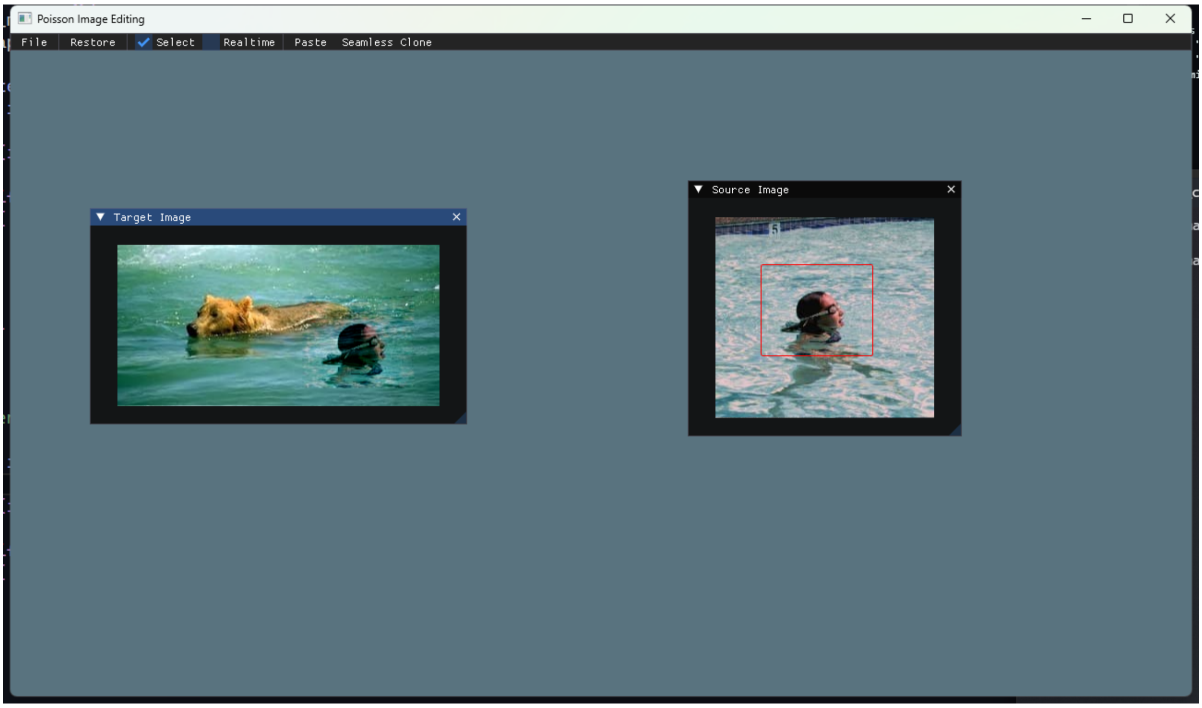
```cpp
void SeamlessCloner::solve()
{
    color_vec.clear();
    int channels = src_image_->channels();

    // 分channel单列求解
    for (auto channel = 0; channel < channels; ++channel)
    {
        Eigen::VectorXd b = Eigen::VectorXd::Zero(decomposer_->A_.rows());
        //index map存储了所有掩码内的像素及对应索引
        for (const auto& pair : decomposer_->index_map)
        {
            int i = pair.first.first, j = pair.first.second;
            int idx = pair.second;
            if (check_valid_range(i + offset_x_, j + offset_y_)) continue;

            // calculate gradient mix color channel
            double total = 0;
            double d_pivot = dst_image_->get_pixel(i + offset_x_, j + offset_y_)[channel];
            double s_pivot = src_image_->get_pixel(i, j)[channel];
            for (auto &neighbor : decomposer_->neighbor_map[idx])
            {
                if (check_valid_range(neighbor.first + offset_x_, neighbor.second + offset_y_)) continue;

                double d_neighbor = dst_image_->get_pixel(
                    neighbor.first + offset_x_, neighbor.second + offset_y_)[channel];
                double s_neighbor = src_image_->get_pixel(neighbor.first, neighbor.second)[channel];
                total += get_gradient_mix(d_pivot, d_neighbor, s_pivot, s_neighbor);
            }

            b(idx) = total;
            // smooth border
            for (auto &border : decomposer_->border_map[idx])
            {
                if (check_valid_range(border.first + offset_x_, border.second + offset_y_)) continue;

                b(idx) += dst_image_->get_pixel(
                    border.first + offset_x_, border.second + offset_y_)[channel];
            }
        }

        auto x = decomposer_->solver.solve(b);
        //按列添加到最终颜色向量集合中
        color_vec.push_back(x);
    }
}
```

演示


Poisson Image Editing
File
Restore
Select
Realtime
Paste
Seamless Clone
Source Image
Target Image


Poisson Image Editing
File
Restore
Select
Realtime
Paste
Seamless Clone
Source Image
Target Image


Poisson Image Editing
File
Restore
☑ Select
☑ Realtime
Paste
Seamless Clone
Target Image
Source Image


Poisson Image Editing
File
Restore
☑ Select
☐ Realtime
Paste
Seamless Clone
Target Image
Source Image