# Poisson Image Editing Report

## Features and Implementation

**The idea is straightforward: implement a bunch of helper methods, break the task down into several consecutive layers, and use these methods in clone().**

### Structure

- layer 1: *boundary_condition()* and *gradient_condition()* return the boundary and the selected region with color and coordinates info packed in the structure *Pixel*. We also maintain two bool 2d vectors *region* and *boundary* for later use.
- layer 2: now that we have boundary and region, we feed them into *get_target_vector()* and *get_sparse_matrix()* to obtain *v*'s on the RHS and the matrix $A$. The sparse matrix is also tracked as a private field for better real-time performance.
- layer 3: now we have the matrix and target (a vector of *v*'s, one for each channel), we feed them into the main method *possion_solver()* which returns a vector of Pixels, they are then iterated through in *clone()* to update the image.

### features

- *poisson_solver()* does solve the function🤣🤣🤣
- mixed Poisson: I also implemented a counterpart of *get_target_vector()* called *mixed_get_target_vector()* to take into account the gradient field of the target picture, as detailed in the paper. It's fairly easy under our project structure since the rest of the code is completely reusable.
- my implementations of *boundary_condition()* and other related methods don't rely on the fact that the region is a rectangle, paving the path for easy extension to polygonal regions.
- the structure is relatively compartmentalized, meaning that we can change the specific implementation of one part without disenabling others, so it's easy to adopt different shapes, different discretization schemes, or even different PDEs.
- the program is fairly robust, whether you have the real-time option on before pasting / seamlessly editing doesn't matter, you can also alternate restore and seamless.

### miscellaneous:

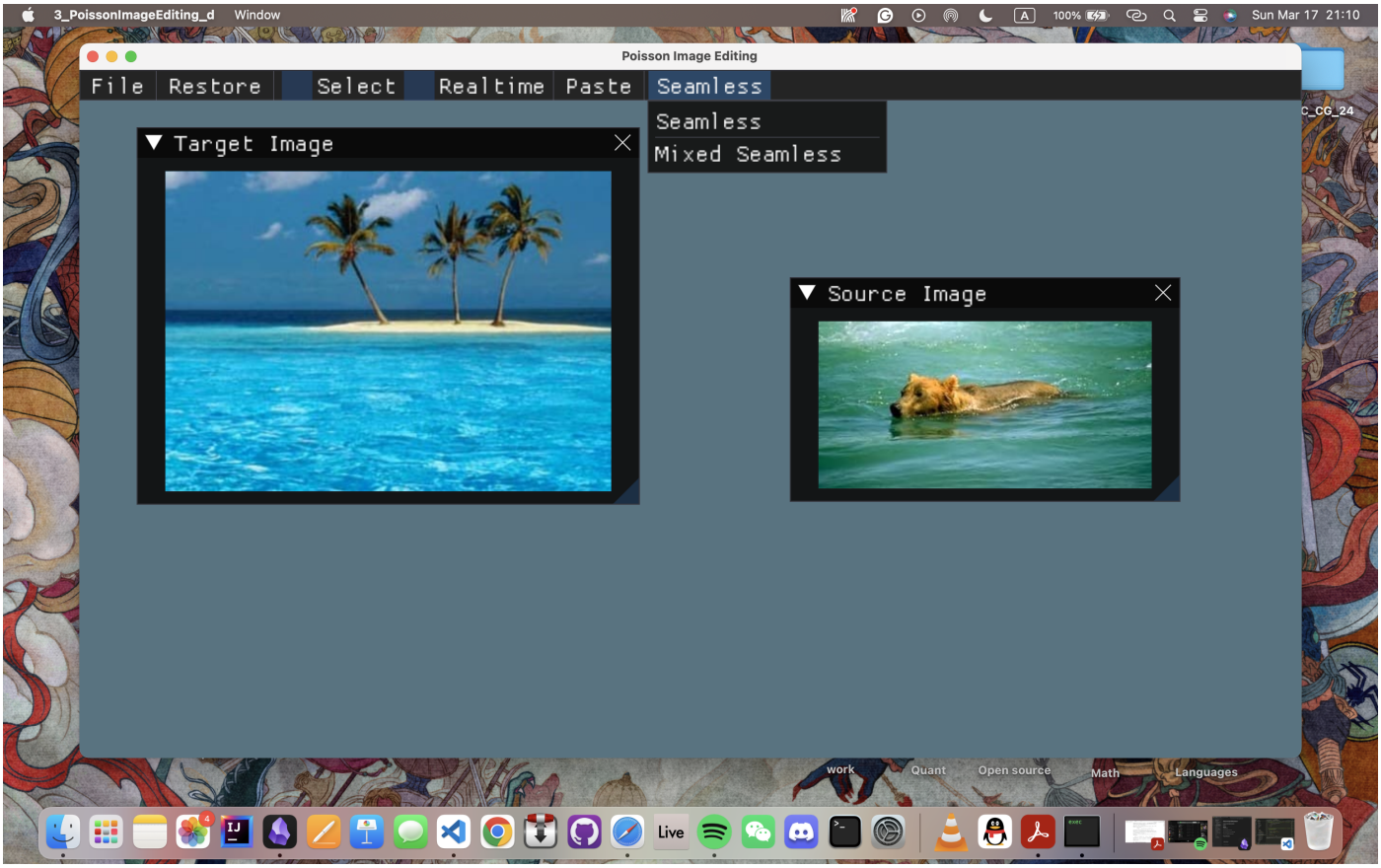
```
we keep track of a SimplicialLDLT<SparseMatrix< float >> solver so that we only have to decompose once for real-time dragging, I use LDLT because the matrix is very sparse and symmetric. Of course, there are a bunch of other helpers and private variables such as *inverse()*, and *region_map* to accommodate implementations or to optimize runtime.
```

## Issues

- the seamlessly edited image has some green spots.
- the performance is still too slow, dragging in real-time is slow and freezes often, I suspect that my implementations of *boundary_condition()* and *gradient_condition()* are slow. The program crashes when you start off dragging too fast, it somehow requires the user to remain still for the first few seconds then you can drag slowly.

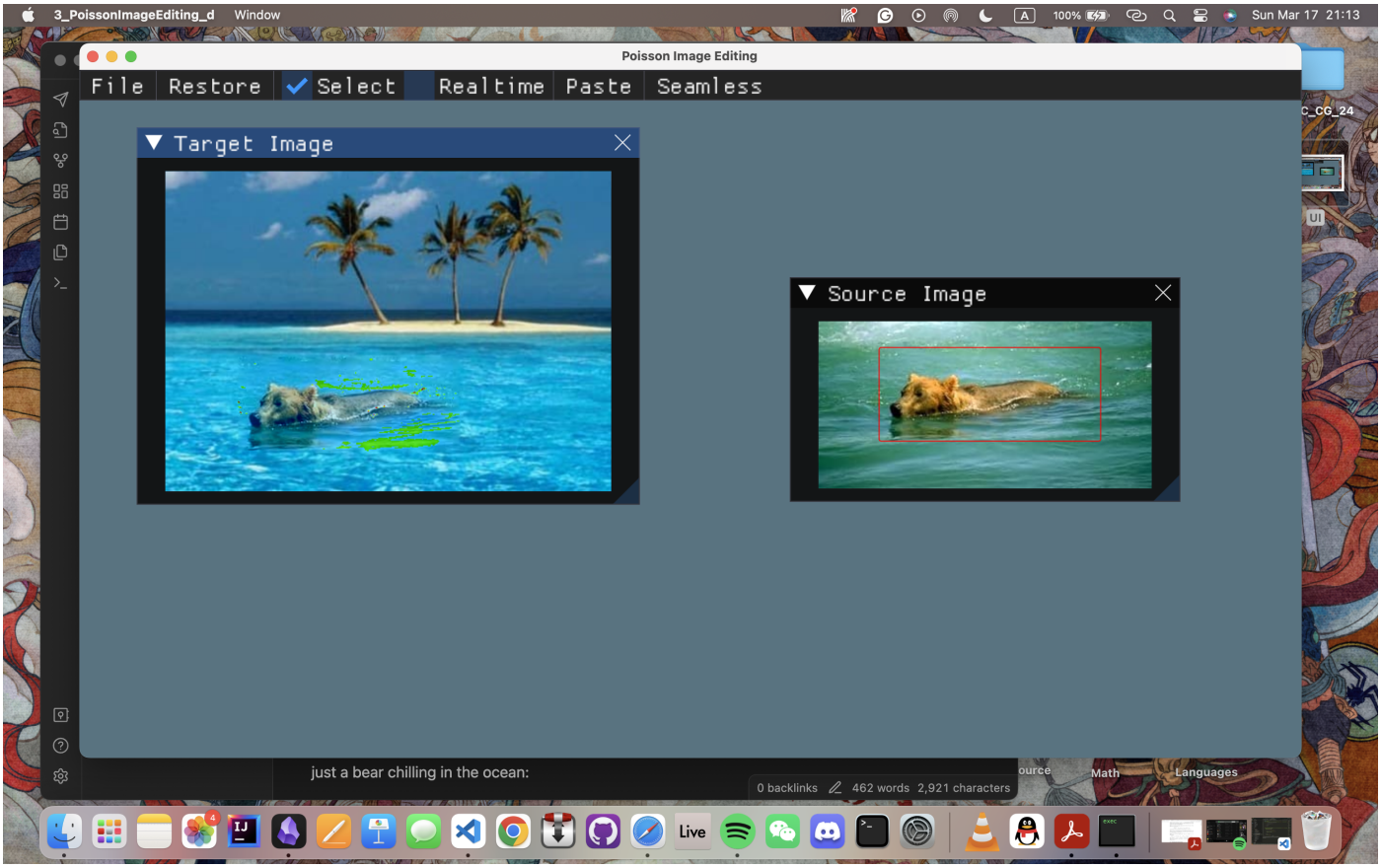
## Results

a pretty basic UI:

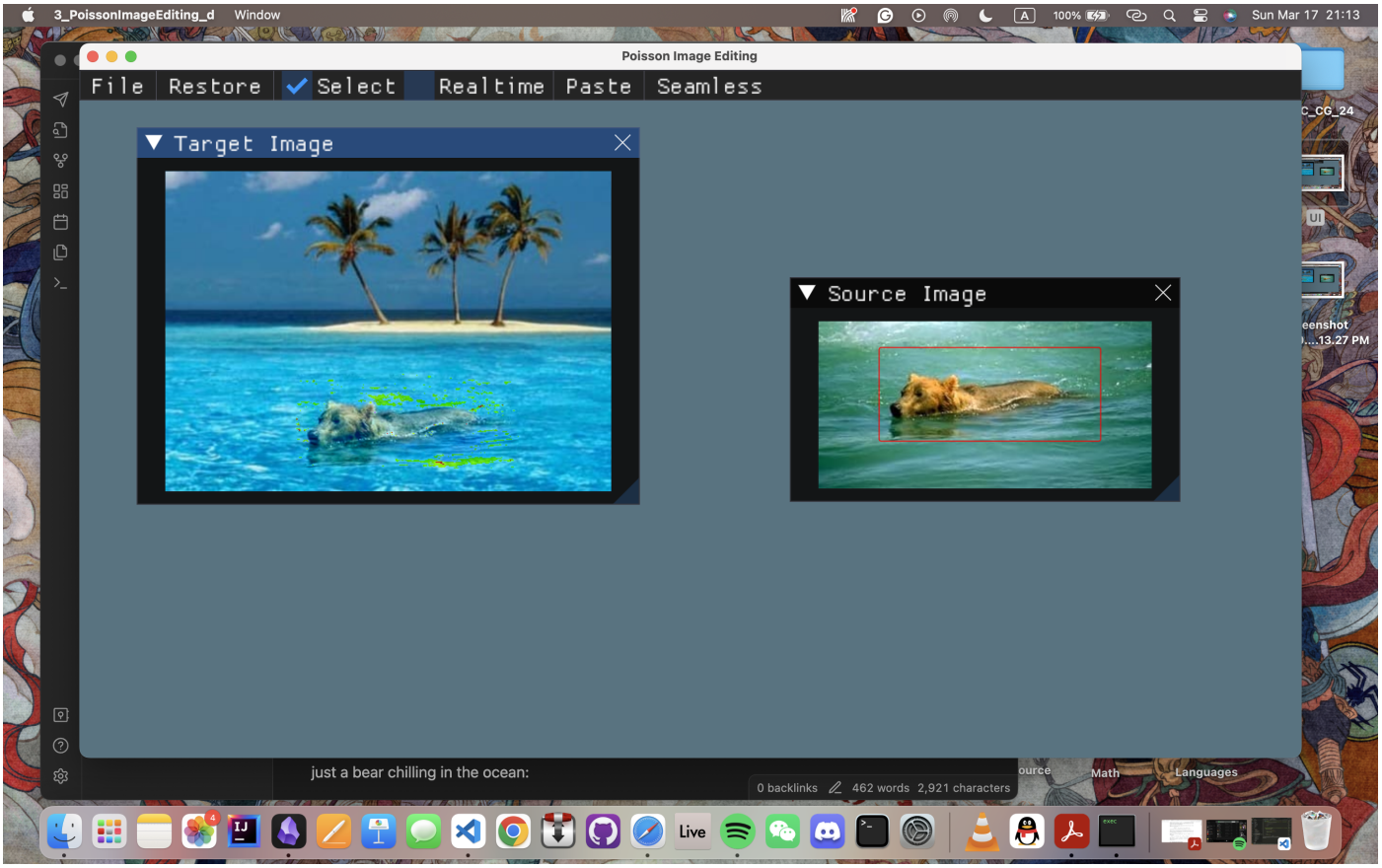



just a bear chilling in the ocean:

unmixed:

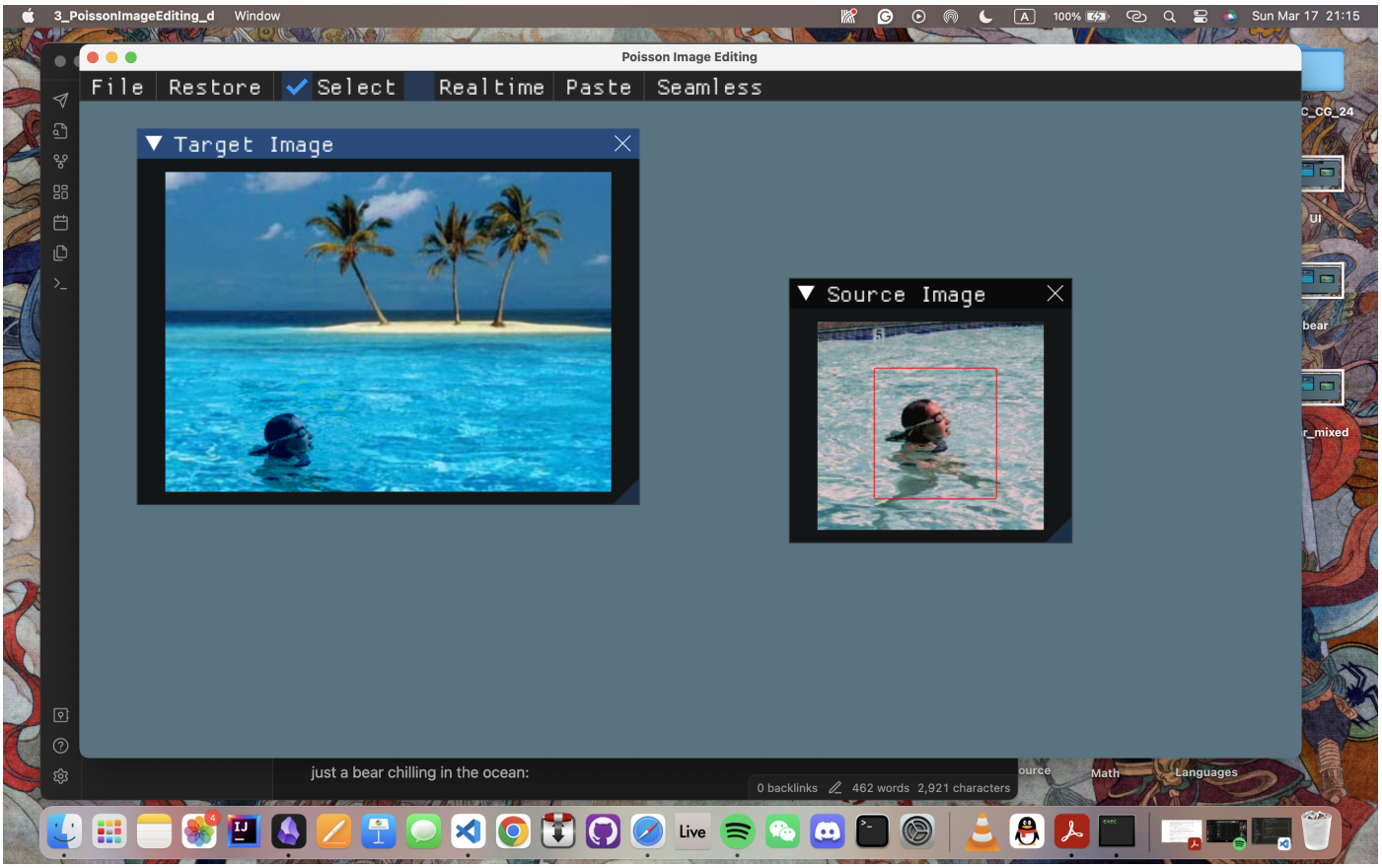


mixed gradient:

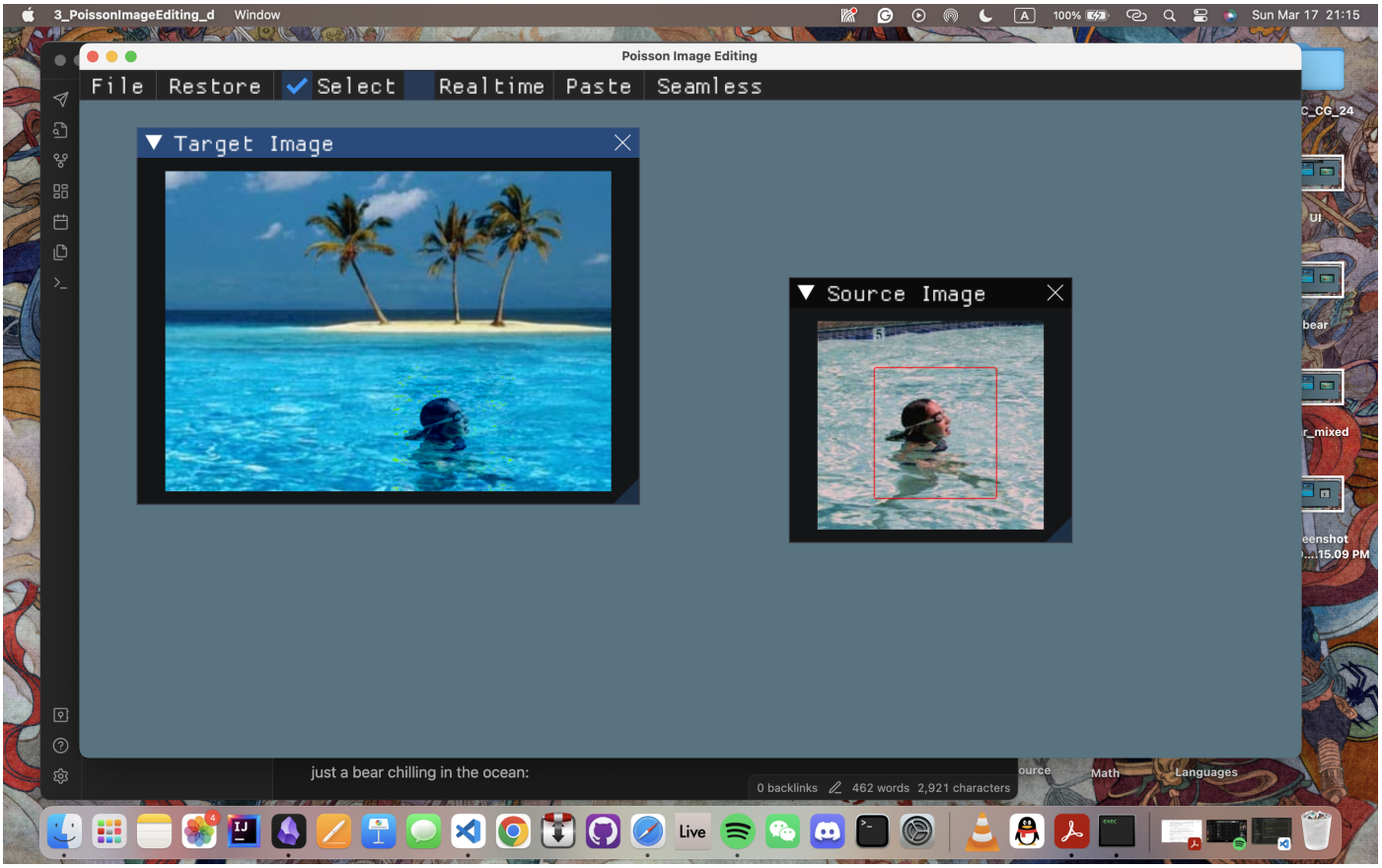



WTF she's drowning in the sea, SAVE HER!

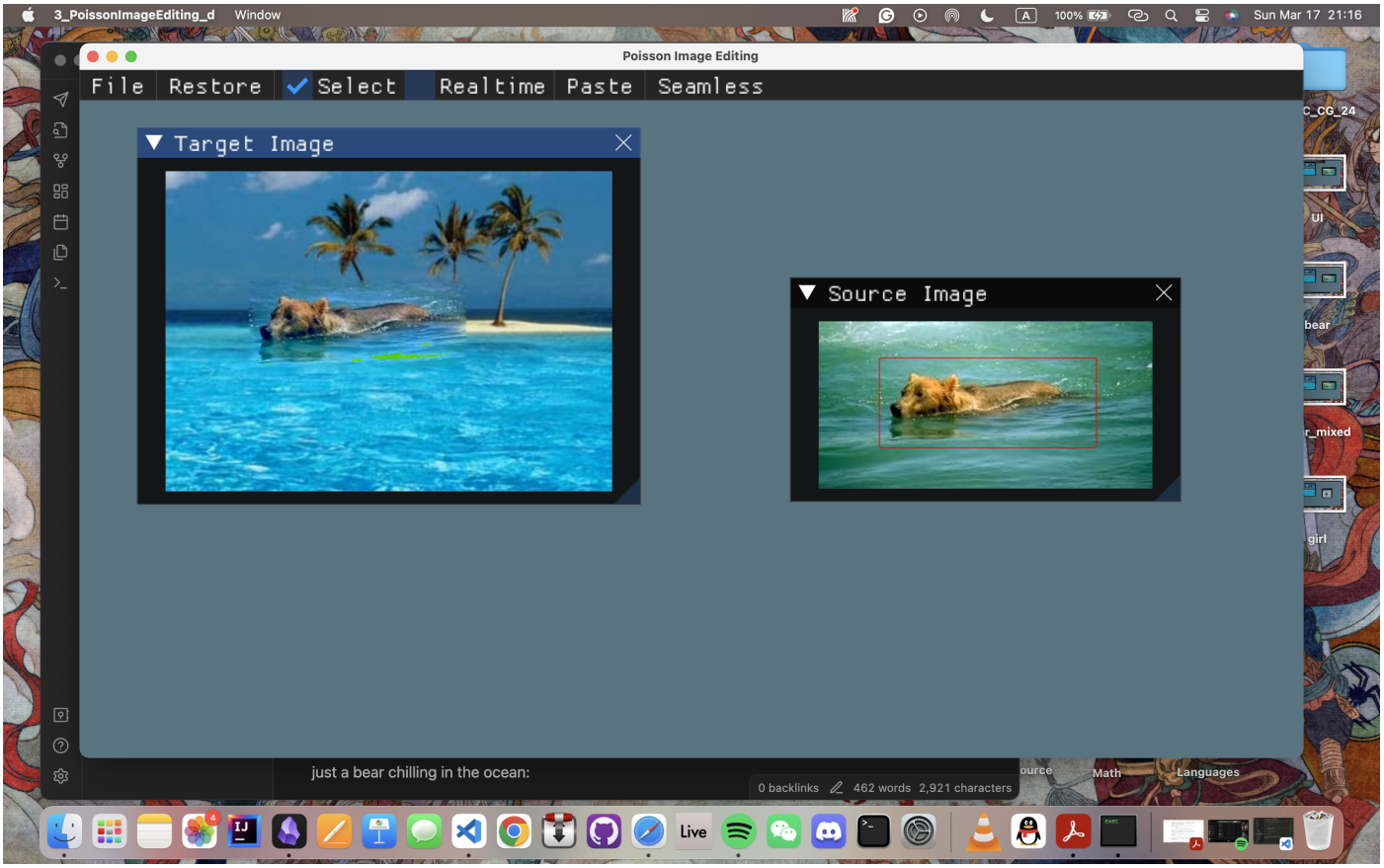
unmixed:

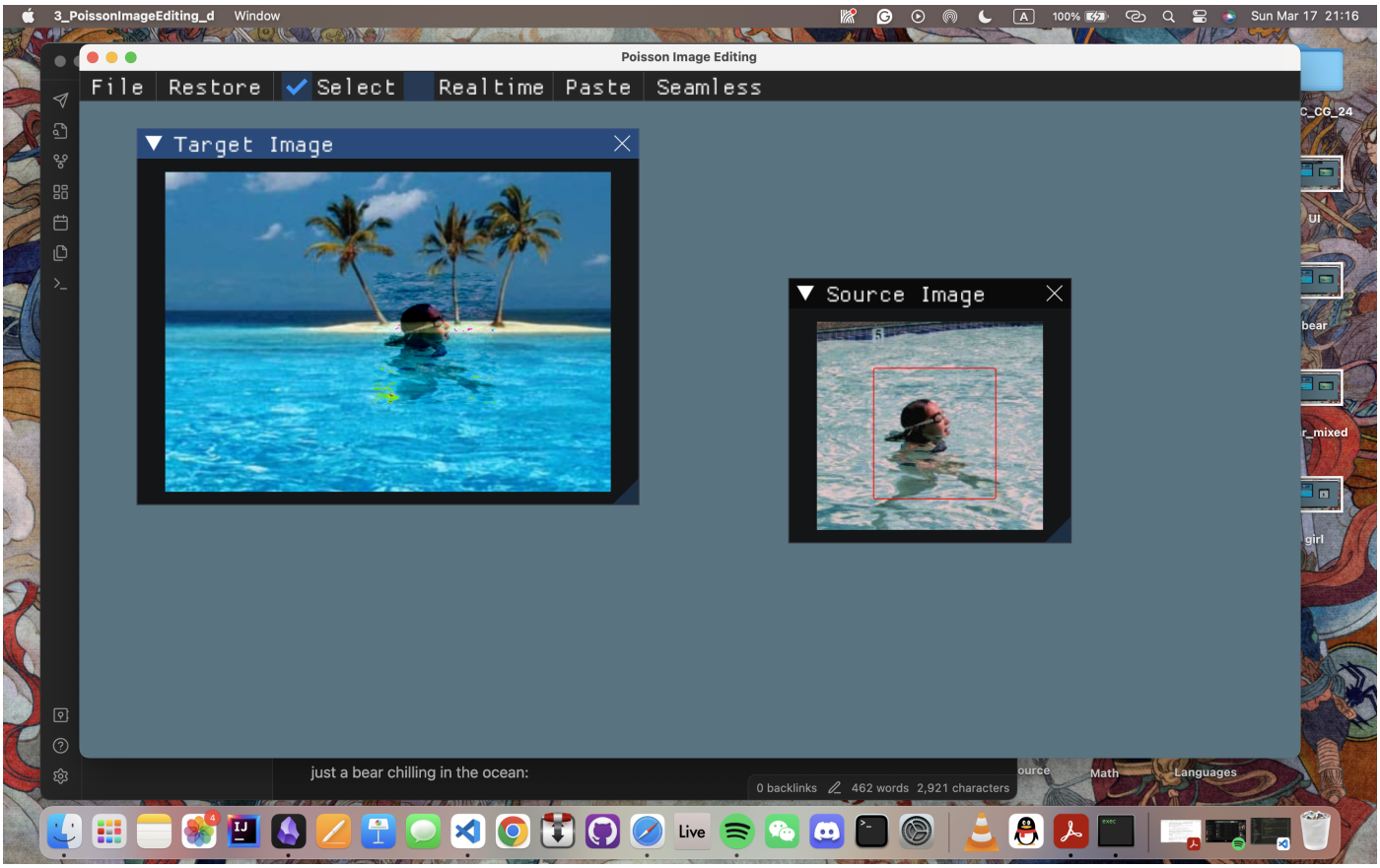


mixed:




They are not supposed to be there???






## Future Directions

- speed up *boundary_condition()* and *gradient_condition()*. Or just somehow speed up the whole thing.
- better mixed schemes.
- add in the polygon region, change *gradient_condition()*, i.e. rasterization, using scanning.