(Nearly almost) Real-Time (but not quite) Gradient-Domain Painting

Viacheslav Kroilov, Yevhen Kuzmovych

ČVUT - FIT

kroilvia@fit.cvut.cz, kuzmoyev@fit.cvut.cz

May 16, 2018

1 Introduction

This project explores methods for painting in the gradient domain described in the paper by James McCann and Nancy S. Pollard[1]. When user paints a stroke, gradient is emitted perpendicular to his stroke as shown on fig. 1.

In the frameworks of this project, simple GUI application that allows user to draw with gradient-painting brash will be implemented.

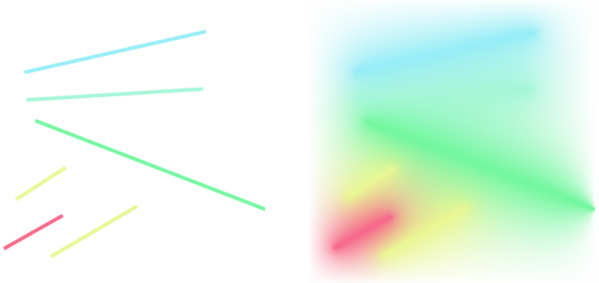

Figure 1: Gradient-domain painting example.

2 Methods

Gradient painting is effectively the same problem as guided interpolation in image editing. Taking blank image as a source (S), image with users stroke as a target (T), and mask Ω with the boundary δ on the whole target image except the place of the stroke, finding the resulting image with gradient is equivalent to solving a Poisson equation:

$$\bigtriangledown^2 u = f \tag{1}$$

Which in our case means for each pixel x, y in the mask solving:

$$\begin{aligned} I_{x+1,y} + I_{x-1,y} + I_{x,y+1} + I_{x,y-1} - 4I_{x,y} = \\ S_{x+1,y} + S_{x-1,y} + S_{x,y+1} + S_{x,y-1} - 4S_{x,y} \end{aligned} \tag{2}$$

where I is the resulting image with boundary conditions on the edges of the image and on the users strokes. Boundary conditions means that pixels on δ are known and $I_{x,y}$ for $(x, y) \in \delta$ are replaced with known $T_{x,y}$.

Using equation(2) we build sparse matrix A and *right-hand side* vector B. Then the solution for $AX = B$ is a vector of the pixel values of the resulting image.

In their paper[1], James McCann and Nancy S. Pollard suggest approximating solution with **multi-grid** method, iterative algorithm for solving large sparse linear systems. In the framework of this project, we use **Gauss–Seidel** algorithm.

3 Implementation

3.1 Technologies

Application is implemented in C++ programing language with the usage of the following libraries:

- **Qt framework**. Framework used for GUI and image processing.
- **amgcl**. Library for solving large sparse linear systems with algebraic multigrid method. It was used as a solver on the initial stages of the development.
- **CUDA**. Framework for parallelization of the computations on the Nvidia video cards.

3.2 Sparse matrix

Matrix A has at most 5 non-zero values (1s and/or -4) on each row and dimensions of $p \times p$ where p is a number of "white" pixels in a mask, which, in our case, are all pixels excluding ones with the stroke. So to minimize memory usage and to simplify CUDA parallelization it has structure as shown on figure 1.

For $|\Omega| = p$ and $|\delta| = p - n$, *pixels* vector stores coordinates of pixels in Ω. $M_{i,j}$ is an index of js neighbour of pixel i. Vector *rhs* contains right-hand side values from equation (2), but if pixel has one or more neighbours from δ, values of those neighbour-pixels are subtracted from the right-hand

	pixels	mat				sol	rhs
unknown	x_0, y_0	$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	I_0	R_0
	x_1, y_1	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	I_1	R_1
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	x_n, y_n	$M_{n,0}$	$M_{n,1}$	$M_{n,2}$	$M_{n,3}$	I_n	R_n
	x_{n+1}, y_{n+1}					T_{n+1}	
known(δ)	$\vdots$					$\vdots$	
	x_p, y_p					T_p	
						0	

Table 1: Sparse matrix structure

side. Solutions vector *sol* stores resulting values of pixels from Ω. If js neighbour of is pixel is on δ, $M_{i,j}$ is an index of *0* in the end of *sol* (because its value is already subtracted from *rhs*).

So instead of solving matrix $p \times p$, we need to solve matrix $n \times n$ but using all p pixels.

3.3 Gauss–Seidel

Gauss–Seidel is a simple iterative method for solving linear systems. In our implementation it is defined as follows:

1. For $it = 0$ to number of iterations
   2. For $i = 0$ to n

$$\text{Set } I_i = \frac{R_i - \sum_{j=0}^{3} I_{M_{i,j}}}{-4}$$

The primary problem of this algorithm is that it might not converge to the solution fast enough, which would require more iterations and consequently more computation time. The solution is the **multigrid** method. The idea behind multigrid is in precomputing solution on the smaller image and initializing solution vector with precomputed values instead of random values (or all 125s as in our implementation). This method allows to perform much more iterations on a smaller image in a short period of time and much less iterations on the original image. We scale image recursively down to image 5×5 and perform $\approx$ 381000 iterations while on the original image (300×300) only $\approx$ 6000 iterations.

4 Conclusion

In frameworks of this project, we've implemented application with graphical interface that allows user to draw in the gradient-domain using gradient brash in real-time (not really $\approx$ 0.2 fps).

Gradient painting is performed using gauss-seidel algorithm with multigrid optimization. Implementation is also parallelized with CUDA framework on the GPU .

References

[1] James McCann and Nancy S. Pollard. Real-time gradient-domain painting. *ACM Transactions on Graphics (SIGGRAPH 2008)*, 27(3), August 2008.