Image Vectorization

Suraj Khetarpal, Dinesh Malav, Jeffry Pincus, and Jeremy Watson

Our demo video can be viewed at https://www.youtube.com/watch?v=Eap16KMvu04

Overview

Our project attempts to solve a common engineering and design problem: taking scanned technical drawings or plans and converting them into a more usable vector format. The vector format allows for better scaling and manipulating of images, as well as better compatibility with common design tools. Our program uses multiple algorithms to accomplish this task. The first is the Canny edge finding algorithm, which combines the Sobel edge detection method with noise reducing and edge thinning algorithms to produce a cleaner and more useful edge map. This is then converted into a series of contours, or related points, using the Moore Neighborhood algorithm. The contour points are then smoothed using Douglas-Peucker algorithm to reduce file size and complexity, and then the program outputs the results to an SVG (scalable vector graphics) file. We also implemented a system for previewing the image using OCaml's graphics library.

Planning

We were largely able to stick with our plan, with some minor adjustments during the course of development. For example, we had initially planned on using the Imagemagick program to help in handling image files, but found there was a library (camlimages) that not only could easily read multiple image file formats, but made working with images in OCaml much easier. Please find our draft spec here, and our final spec here.

Our project broke up somewhat naturally, which meant we could work in parallel while only needing to know the interfaces that each of our parts implemented. We hit our milestones, although we did experience some of the 99 rule (first 90% of code takes 10% of the development time, last 10% takes the other 90%). We found no matter how nice it is to know the interface of your collaborators, there will still likely be problems when putting everything together at the end. Issues with the camlimages library meant some of our group had trouble compiling the whole project, but as the library was only needed for the first portion we were able to segment it away and combine it in the end after some of the process had been ironed out. Because our agreed upon interfaces changed very little over time, our individual sections linked quite smoothly. In only one linkage case, the linkage between the edge detection and contour extraction, did we discover the need to redesign slightly (to avoid stack overflows when working with the large data sets that are output by edge detection).

Design and Implementation

We chose to use OCaml for the implementation, that way everyone in the group would be on the same page regarding the language we were going to use. It also meant we could continue to utilize the development environment we had been working in all semester. Looking back, we feel that this was a good choice as it meant that we knew exactly what we could expect of our group, and having a common starting point almost surely sped up the process of setting up the environment and getting started. This did not mean everything went smoothly; as mentioned previously, all group members had trouble installing and using the camlimages library in the appliance. We quickly learned that there is no aspect of development that you can take for granted. Just arriving at the point where we could being coding was a large chunk of the overall work that went in to the project.

The initial algorithm we chose for edge finding was the Sobel method, but we eventually found this alone was not giving us exactly what we needed. Fortunately the convolution code that implemented the Sobel algorithm could be generalized, and eventually we settled on the Canny algorithm, which utilizes Sobel but includes some extra steps that helped eliminate image noise (a concern for technical documents that might be scanned poorly) and make the output more compatible with steps down the line (edge thinning). This complicated the initial implementation as Canny requires extra information that Sobel leaves out, such as the direction of the gradient forming the edge.

The first algorithm we considered for contour extraction was the Moore Neighborhood Algorithm, and in the end, we decided it was our best option. Unfortunately, the majority of the information we found on its implementation only dealt with excessively simple matrix configurations. We had to improvise and expand the algorithm to handle complexities such as nested contours and intersections. In order to avoid stack overflows when dealing with large matrices, we had revise certain pieces of our code to go from functional to imperative programming. We also had to account for pixel neighborhood edge cases that were sending us into infinite loops.

The output from contour extraction generates very dense polyline and needed to be simplified. For this purpose we chose Douglas-Peucker algorithm combined with pre-processing polylines using radial distance based reduction. Polyline simplification dramatically reduces the number of points in a polyline while retaining its shape, giving a huge performance boost when processing it and also reducing visual noise.

The final step was parsing the simplified polylines, which are lists of (x,y) coordinates, into an XML formatted SVG file. There are many ways to do this; we decided to keep it simple and to connect all the coordinates with straight-line segments.

Contributions

Every member of the group was heavily involved in the development of our high-level program design, our interfaces, and our documentation. Suraj Khetarpal made exceptional contributions in the area of contour extraction. Jeffry Pincus made exceptional contributions in the areas of SVG file generation and development environment creation. He also did the animations for our demo video. Jeremy Watson made exceptional contributions in the areas of edge detection and development environment creation. Dinesh Malav made exceptional contributions in the areas of contour point smoothing and vector data structure creation.

Reflection

We found that we had many options when it came to choosing our methods for each step of the vectorization process, which was a pleasant surprise in that it allowed us to shop around to find the best fit algorithm for our chosen.

The difficulty we had in compiling and running the camlimages library was a very unpleasant surprise. Many group members worked for hours trying to get the library properly installed, and in the end only a couple of us had it fully working. We had hoped using the same environment would mitigate or eliminate problems like this, but clearly that was not the case.

If we had more time, we would incorporate additional vector shape types into the implementation. We currently rely heavily on the polyline object type. This works quite well, but we would like to see how the use of a Bezier curve object type might further reduce complexity, file size, and improve output quality. We have structured our display element code with abstract data types so that the addition of new vector shape types would be a seamless enhancement.

Advice for Future Students

During the first half of the project timespan, we did a great deal of group collaboration to create a high level roadmap of how we would implement the program. We also put a great deal of thought into what our interfaces would look like prior to splitting up work. We believe that these two strategies were crucial to the success we had later in the project. If we were to do anything differently next time, it would be to consider more seriously the time that can be consumed in trying to set up an unfamiliar development environment.