Automated Conversion of Sketches into Source Game Engine Maps

Cameron Stevenson

Supervisor: Richard Green

Abstract—**This paper proposes a method to automatically generate Source game engine maps from hand sketched layouts. Segmentation is achieved through adaptive thresholding and flood fill, with an 8% difference from a reference segmentation. Text recognition is achieved through squared difference template matching, with 100% detection and recognition and no false detections. A compilable and playable CS:GO map is generated.**

I. Introduction

Video games made by Valve run on the Source game engine, and share a common map editor and map file specification. Counter Strike: Global Offensive (CS:GO) is one such game, which has many past and present community-made maps for the competitive game mode. The map editor operates in 3D and is more suited to blocking in finer details. Blocking in the basic layout of the map can be time consuming, especially with complex geometry such as curves. If the process of taking a 2D layout idea and making a 3D map from it were automated, it would free up creative potential for map authors. This paper aims to automatically interpret sketched layouts to create the geometry and entities necessary to have a compilable and playable map.

II. Background

A. Text Detection and Recognition

Rosebrock [1] and others detail a method of detecting text bounding boxes with EAST, and recognising words with Tesseract. EAST is capable of detecting word and line text in natural backgrounds [2]. Tesseract [3] is Google's text recognition engine which maintains uncertainty throughout the letter recognition process until words are recognised. Feeding the bounding boxes found by EAST into Tesseract is usually a robust method of recognising font text in real images.

Methods exist for recognising handwritten characters from different people. Pal et al. [4] skeletonize characters and use neural networks to recognise characters. This method operates on images of single characters.

General shape similarity methods may be applied, such as those found in Skiena's book [5] Hamming distance of two shapes is how much area they don't overlap in when placed on top of each other. Hausdorff distance focusses on the place where 2 shapes differ the most, measuring the furthest distance you can be from one shape if you lie on any point in the other shape. Skeleton comparison reduces shapes to their core structure, then compares topology and length/angle features.

B. Sketch Interpretation

Naya et al. [6] recognise the need for allowing users to sketch designs before converting them into computer models. Their proposal shows how sketched lines can be interpreted within certain tolerances to give digital representations.

Similarly Zargar et al. [7] interpret perspective sketches to generate 3D models.

Yetiş et al. [8] observe that freeform sketches generally rely on human experience to resolve ambiguities when interpreting their meaning. Also, discontinuities and irregularity in lines can cause imprecise models. They attempt to use deep learning in the 2D to 3D conversion, and it gets the basic outline of architectural sketches correct.

C. Polygon Simplifcation

Polygons sourced from real life data (such as a photo scan) can have more points representing them than realistically necessary. Polygon simplification adds, removes or modifies vertices to produce a similar enough polygon with much fewer vertices.

The Douglas-Peucker algorithm [9] for polygon simplification is additive. It begins with 2 points of the original polygon, and adds other points from the original polygon which are sufficiently far from the current approximation. Once every non-added point is sufficiently close to the approximation, it terminates.

D. Polygon Triangulation

Many applications use triangles as primitives, so breaking down a polygon into triangles is useful.

A typical triangulation has $n-2$ triangles for n vertices. Asano et al. [10] demonstrate how collinearity of vertices can be used to minimise triangle count beyond $n-2$, at the cost of time complexity. Triangulation cannot be faster than sorting.

Shewchuk's Triangle [15] creates Delaunay triangulations, optionally with no large or small angles. Delaunay triangulations spread out triangles fairly evenly throughout the polygon.

E. Valve Map Format

Valve Map Format [11] describes the structure of an editable map file. It has a key-value and nested structure similar to JSON. Of particular interest is the brush, referred to as "solid" in this wiki. A brush is a convex 3D shape, defined by a number of sides making up its shape within them. Each side is a plane, and where these planes intersect create the vertices and edges of the brush.

III. Proposed Method

A. Segmentation

We aim to find the segments divided by pencil lines.

The sketch is grayed, inverted, and blurred. An adaptive threshold highlights both sides of each pencil mark as black, with the background in white. This is inverted so pencil edges are white against a black background. A closing operation with 2 dilations and 1 erosion of a 5x5 kernel joins the two sides of each pencil mark, giving thick white pencil against a black background. These white edges may contain black pixels within them. This is acceptable as long as the edge has no black path between segments. See Figure 1.

Now that black segments are divided by solid white boundaries, flood fill is used to identify internal segments and the external background. Each pixel in the image which is still black is used as a seed point for flood fill with intensity value 128. A segment is accepted as internal if it is large enough (more than a few pixels) and does not touch the image border. These segments are given intensity value 192 to indicate they are discovered. Rejected segments are given intensity value 255, so that they are treated as equivalent to pencil. The resulting image has background and pencil in white, and internal segments in gray. See Figure 1.
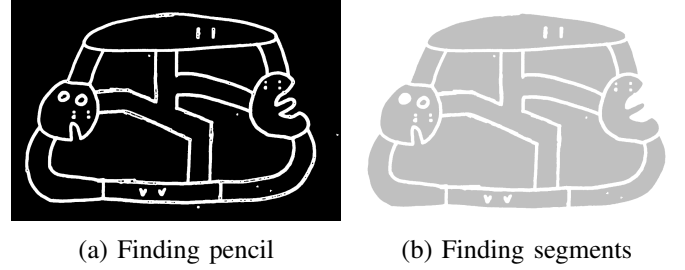

(a) Finding pencil (b) Finding segments

Fig. 1: Segmentation

B. Text Recognition

Short handwritten text is used to label segments based on what they will be in a map (floors, walls etc.). We aim to locate text precisely enough for labelling, and recognise the handwritten letters.

The common method of detecting text with EAST and recognising text with Tesseract was tried. It was found to work best with longer words and a background without other pencil. Our application requires short labels (up to 2 letters at most) and text near other pencil, so this approach was unsuitable.

Instead convolution methods were tried. Inverted samples of handwritten letters were turned into kernels of various scales and convolved over the inverted image, with the idea being that highlights in the output showed presence of the letter. This detected well, but structural features in the sketches were also detected as text. Convolution by its definition rewards white overlap but does not reward black overlap, and does not explicitly punish opposites. For example a solid block of pencil would be detected as any sampled letter. A more specific sort of convolution is needed.

Template matching gives us the specificity needed by rewarding pixel similarity and punishing dissimilarity. Squared difference template matching was used with the sampled letters at various scales over the image, to produce a mostly whitish image with black spots where letters are found. This is inverted and thresholded so that only confident highlights are retained. These highlights from all scales are merged into one image (as a weighted average where every scale has the same weight). It was found that each true highlight tends to appear on multiple scales, so thresholding this merged image retains these and discards some random highlights. Suzuki et al. [12] contour finding algorithm is used to find the location of remaining highlights. As a further step of narrowing down true highlights, on the sketch we use double letters as labels (e.g. "XX"). So we look for pairs of highlights within a certain distance and angle range. The midpoint of each pair and the letter is recorded. This gives us the

text label letters and locations. Figure 2 shows this whole process.
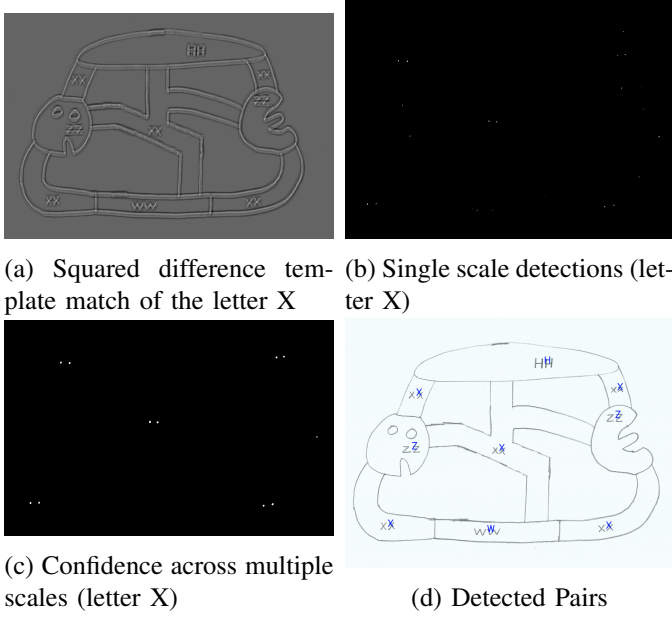

(a) Squared difference template match of the letter X

(b) Single scale detections (letter X)

(c) Confidence across multiple scales (letter X)

(d) Detected Pairs

Fig. 2: Text detection and recognition through template matching

C. Labelling and Gap Filling

From the segments and text labels we want to produce an image where each label has a certain colour, and each segment with that label is coloured that way. See Figure 3.

Prior to segmentation, for each text label we fill a white rectangle over it so it isn't interpreted as a structural segment. This does not need to perfectly cover the text, but does need to reach the top and bottom of the text, so that flood filling and pencil gap filling work later.

After segmentation we colourise segments. For each text label, we use a user-defined lookup table to find its colour, then flood fill that colour from the label's location. This gives an image where labelled segments are coloured, background and pencil is white, and unlabelled segments are gray. From this point on we consider background and unlabelled segments to be "wall". We replace white and gray with black as they can be treated as the same from now on. To improve efficiency of the gap filling step we crop down the image to remove extraneous background, and scale the image down (the loss in precision is irrelevant when compared to the inaccuracy in segmentation).

We want to fill in the black gaps left by pencil marks, both between segments (intended structural marks) and within segments (typically produced by uncovered text).

For this we make PENCIL_THICKNESS passes over the image, where PENCIL_THICKNESS is half the width of the widest pencil mark we need to cover. In each pass if a black pixel has a coloured neighbour, it becomes that colour. This slowly grows the coloured segments until they meet each other, removing inter-segment gaps. It also fills in internal holes. After this step any remaining black segments are considered to be wall, and filled in with the "wall" colour. This gives a simplified image with colourised segments for the rest of the program to operate on. See Figure 3.
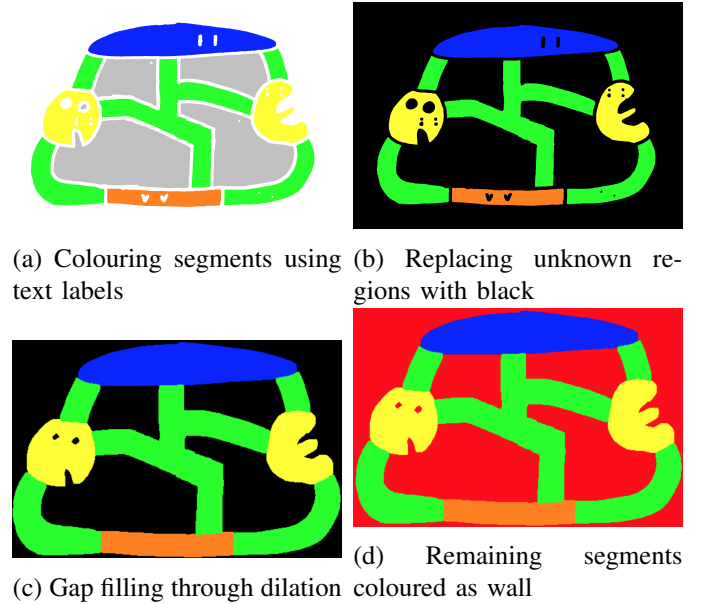

(a) Colouring segments using text labels

(b) Replacing unknown regions with black

(c) Gap filling through dilation

(d) Remaining segments coloured as wall

Fig. 3: Segment labelling and gap filling

D. Segment Bordering and Triangulation

We aim to find the border of each coloured segment, simplify it, and triangulate it to make brushes for the map file. This section operates independently of the prior sketch processing so that computer drawn images can be used.

Flood fill is used to find the pixels in each segment. A simple border walk is used to find the exact pixel by pixel border. Borders are simplified with a tolerance as a proportion of the area of the segment. Some preprocessing passes are made which operate locally, to reduce complexity. Firstly for any 3 collinear points the middle point is removed. Secondly when a corner of 1 pixel in width or height is encountered, the corner point is removed. After this tolerance based removals are applied globally, each time looking for the point which when removed changes the segment area the least. Only concavities are removed, which is suitable in this context as it ensures no gaps between the final map objects.

The colour of segments are used as labels to generate map objects. Map objects have borders and a type (wall, floor etc.). For each object its border is triangulated. This uses a fairly simple approach of cycling around the border looking for valid triangles (counter-clockwise, not intersecting the border's edges, and not containing another border point). A valid triangle is added to the result. The polygons left over from removing this triangle are themselves triangulated recursively to find the remaining triangles.

IV. RESULTS

A. Setup

OS: macOS 10.13.4
Processor: 2.4 GHz dual core I5-4258U
IDE: Visual Studio Code
Language: Python 3.7.2
Device: PC
Camera: 300dpi printer scanner (Deskjet 3070 B611)
OpenCV Version: 4.5.1

B. Segmentation

Segmentation accuracy was tested with Hamming distance, comparing against a manual segmentation. In proportion to the size of the image, there was an 8.0% difference between the algorithmic segmentation and the manual segmentation. For reference two different manual segmentations had a 2.6% difference. We can visually inspect Figure 4 to see that the algorithmic segmentation thickens non-walls and thins walls, which accounts for the larger difference.
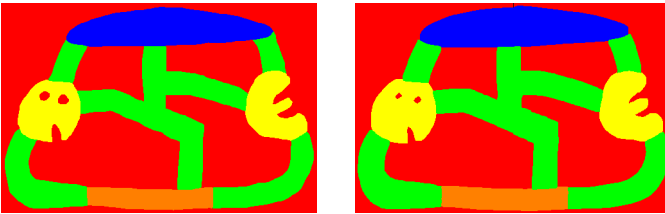

(a) Manual Segmentation   (b) Our Segmentation

Fig. 4: Segmentation comparison

A better solution would treat the pencil filling stage (which acts on non-wall) as a closing operation rather than just a dilation.
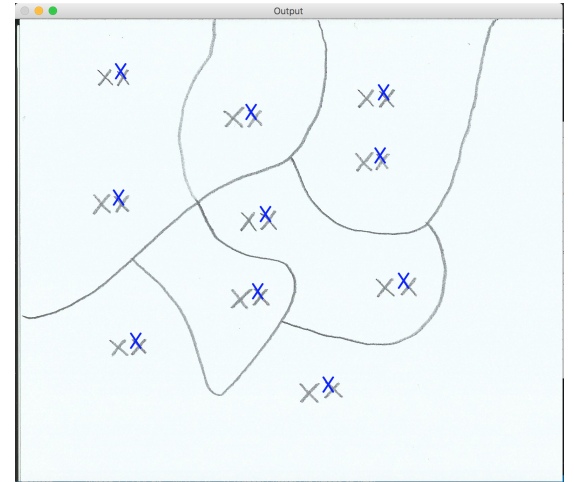
C. Text Recognition

The commonly described EAST + Tesseract method detects 90% of double characters but only recognises 30% of them due to its dependency on recognising complete words. Our method using sample template matching detects and recognises 100% of double characters, with no false detections on other lines. For a comparison see Figure 5.
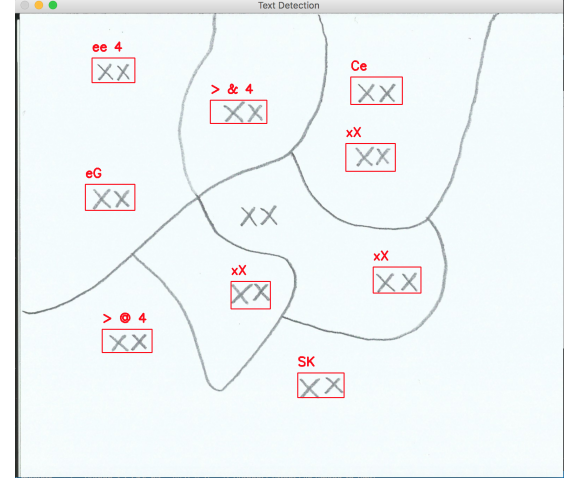
(a) Common OCR Method

(b) Our Sample Recognition

Fig. 5: Text detection and recognition

On the sample image in Figure 5 the EAST + Tesseract method takes just 5 seconds to complete, whereas our method takes 43 seconds. For an image $W * H$, template symbols $M * N$, and number of template rescalings $X * Y$, our method has complexity $O(W * H * M * N * X * Y)$. On A4 pages at 300 ppi, our method takes about 3-4 minutes for text recognition. Template size isn't feasible to adjust, however image size and template rescalings are. Image size can be adjusted by manual cropping after scanning. Number of template rescalings can be reduced by consistently drawing letters at similar sizes, and setting tighter bounds on their possible size.

V. Conclusion

Pencil sketches of map layouts were segmented and coloured based on text labels. Segmentation was achieved through a robust pencil finding method, and flood filling between pencil. Text recognition was achieved through template matching of hand drawn letter samples.

The segmentation game within 8.0% of a reference segmentation, with no topology differences. The text recognition had a 100% detection and recognition rate, with no false detections. The map produced was compilable and playable, with no errors related to brush size or shape.

VI. Further Research

The approach described is a solid start to automating map creation, however there are many possible improvements.

A. Text Recognition

At many points throughout template matching, seemingly arbitrary parameters have been chosen as they were found to correctly distinguish between letters and other structures. The margins for these parameters to work were small, and I suspect they are tuned to my exact scanning setup. To make text recognition more robust, these parameters could be investigated or made adaptive. Here are some ideas:

- Squared difference produces a large range of values. Currently the largest value in the output is found, and the output normalised from this value. There should be a way to calculate a theoretical "sample on plain background" or "sample on exact opposite" expected value, based on template size and colour dimensions, to consistently normalise by the same amount across all scales.
- We expect 1 of each team spawn and 1-3 bombsites. We could adaptively alter detection sensitivity until the correct number are detected, which will also improve detection accuracy of floors.
- Each letter is different in its density and similarity to other drawn structures, so each letter could be tuned with its own parameters.
- Run a template match on the template itself to see what we'd expect to find from an exact match, and try to find similar outputs on the template matched sketch.

Performance is a key issue with template matching. GPU implementations with kernels we define based on templates, such as with CuPy [13], could be investigated.

More advanced methods such as feature descriptors could be used. Neural networks could be trained on handwritten 2 letter words for this particular application.

B. Segmentation

There is a small margin between the intensity of background paper and intensity of pencil. Adaptive thresholding is used currently, but we may want to know the background and pencil intensities for other uses (the white rectangle that covers up text for example). An intensity histogram could be used, looking for two peaks representing the background and pencil intensities.

As previously mentioned, the pencil gap filling stage only acts as a dilation, resulting in enlarged non-walls and shrunken walls. If precise scales are needed, an erosion pass of the same size could be made after gaps are closed to fix the scales. Also the pencil filling stage has a fairly naive implementation, with one pass at a time looking in a $3*3$ region around each pixel. Typical morphology implementations could be extended to this colour and black situation, to improve performance. A kernel convolution solution may exist.

The white rectangle used to cover up text can be made more precise if needed in future. We could record the detection size which produced the matched pair to get the exact height needed. A marching algorithm could be used to work outwards from the centre until the outside of the text is found.

C. Border Simplication and Triangulation

Currently we border march every pixel to find an exact border, then simplify it separately. Approaches exist to border march multiple pixels at a time, effectively combining these two steps into one. Pu et al. [14] propose adaptive border marching to dynamically choose step sizes whilst marching around a border. Step sizes are made smaller if a larger step excludes a sufficiently large concavity or protrusion. This method retains necessary details whilst reducing vertex count. Alternatively we could keep the current 2 step algorithm and implement Douglas-Peucker [9] for simplication.

Segments are currently allowed to become larger during border simplification, which overlaps brushes from different objects in the map. This is good for gameplay as there are no gaps, but it may cause unforeseen problems with the visibility algorithm or other parts of map compilation. The border simplification could be adjusted into a global algorithm, where adjacent borders share the same vertices, and removing or moving vertices applies to any segments using them in their border. So

borders end up perfectly joined between segments, and brushes align perfectly in the map.

Donut topology (holes in the middle of segments) are not accounted for in the current program. As a cheap fix the outer wall has a pixel of black drawn through it to break up the donut, but internal segments are still vulnerable to the issue if a user draws an object with donut topology. An accurate fix would be to find holes in segments and record them as internal borders, simplify these borders the same as the external border, then use a triangulation algorithm which can cope with internal borders. For example Shewchuk's Triangle [15]. A cheaper fix would be to detect holes in segments, then split the segment in two, and recursively fix holes in the two new segments produced, until there are no holes remaining.

D. Other Features

Config options such as scaling and skybox texture can be added. Some way of sketching crates should be added.

CS:GO maps typically have some height variation, and even overlapping floors. A way of defining a height map through drawn elevation numbers could be added, and some way of sketching overlapping floors. Perhaps a symbol indicating where two nearby floors should be joined together could be added.

REFERENCES

[1] A. Rosebrock, "Opencv ocr and text recognition with tesseract," 2018. [Online]. Available: https://www.pyimagesearch.com/2018/09/17/opencv-ocr-and-text-recognition-with-tesseract/

[2] X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He, and J. Liang, "East: an efficient and accurate scene text detector," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2017, pp. 5551–5560.

[3] R. Smith, "An overview of the tesseract ocr engine," in *Ninth international conference on document analysis and recognition (ICDAR 2007)*, vol. 2. IEEE, 2007, pp. 629–633.

[4] A. Pal and D. Singh, "Handwritten english character recognition using neural network," *International Journal of Computer Science & Communication*, vol. 1, no. 2, pp. 141–144, 2010.

[5] S. S. Skiena, *The algorithm design manual*. Springer International Publishing, 2020.

[6] F. Naya, J. Jorge, J. Conesa, M. Contero, and J. M. Gomis, "Direct modeling: from sketches to 3d models," in *Proceedings of the 1st Ibero-American Symposium in Computer Graphics SIACG*, 2002, pp. 109–117.

[7] S. H. Zargar, J. Sadeghi, M. Hashemi, N. Motallebi, and A. A. Garmaroodi, "Introducing a new method to convert a 2d hand-drawn perspective to a 3d digital model," *Nexus Network Journal*, pp. 1–17, 2019.

[8] G. Yetiş, "Auto-conversion from 2d drawing to 3d model with deep learning," Master's thesis, 2019.

[9] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica: the international journal for geographic information and geovisualization*, vol. 10, no. 2, pp. 112–122, 1973.

[10] T. Asano, T. Asano, and R. Y. Pinter, "Polygon triangulation: Efficiency and minimality," *Journal of Algorithms*, vol. 7, no. 2, pp. 221–231, 1986.

[11] V. Authors, "Valve map format," 2006. [Online]. Available: https://developer.valvesoftware.com/wiki/Valve_Map_Format

[12] S. Suzuki *et al.*, "Topological structural analysis of digitized binary images by border following," *Computer vision, graphics, and image processing*, vol. 30, no. 1, pp. 32–46, 1985.

[13] "User-defined kernels," 2015. [Online]. Available: https://docs.cupy.dev/en/stable/user_guide/kernel.html

[14] J. Pu, J. Roos, A. Y. Chin, S. Napel, G. D. Rubin, and D. S. Paik, "Adaptive border marching algorithm: automatic lung segmentation on chest ct images," *Computerized Medical Imaging and Graphics*, vol. 32, no. 6, pp. 452–462, 2008.

[15] J. Shewchuk, "Triangle: A two-dimensional quality mesh generator and delaunay triangulator," 2005. [Online]. Available: http://www.cs.cmu.edu/q̃uake/triangle.html