Detecting and Tracking Games of Carcassonne

Adrien Truong
Stanford University
Stanford, CA 94305
aqtruong@stanford.edu

Abstract

In this paper, we detail a system to track a board game called Carcassonne. Carcassonne is a turn based board game in which players take turns laying down tiles in a grid pattern. Our system is split into two modules: (1) recognition module and (2) tracking module. The recognition module uses a combination of Canny Edge Detector, Hough Transform, and RANSAC to fit a projective grid onto the scene and uses a template matching based approach to classify tiles. The tracking module uses the recognition module to translate scene changes into actual moves in the game. It also leverages its understanding of the game to make recognition easier by providing known constraints to the recognition module. We find that combined, these modules perform well and track the game adequately.

Code: https://github.com/adrientruong/carcassonne
Short demo: http://bit.ly/cs231aCarc

1. Introduction

Tracking board game state is a fun application of scene understanding. The camera observes pieces moving around in its field of view and the goal is to have a system that can understand the rules of the game and interpret what is happening. This is useful if we want to create a written history of the game or if we want to build an AI for the game that doesn't require manual input of game moves.

In this paper, we look at the game of Carcassonne specifically. Carcassonne is a turned based board game in which players take turns laying down tiles in a grid pattern. Players can only place tiles adjacent to an existing tile and they must ensure their edges match. See Figure 1 for a visualization of this.

We detail a system that can observe the board game from start to finish and recognize the state of the game at each time point. This system was designed for ease of use and therefore does not require an overhead camera. The system is designed for a camera placed off to the side.

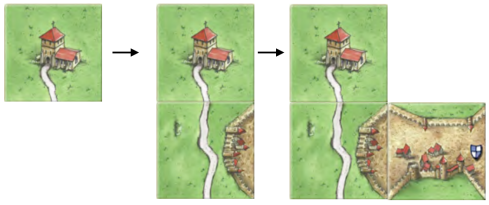

Figure 1. 3 turns of Carcassonne

1.1. Challenges

There are various challenges to building a system to track games of Carcassonne. First, it does not have a fixed size grid. Players place tiles one by one to form a grid structure that expands in arbitrary ways. Second, the tiles are not high contrast as in a game like Chess which makes segmenting them for classification tough. Third, since the grid is formed by players freely placing tiles, the grid is not a perfect grid. Tiles are imperfectly placed which means grid lines are not perfectly straight. Lastly, since we do not use an overhead camera, we must deal with a perspective view of the game.

1.2. Problem Statement

As input, we take in a video from a static camera observing the board game from a side view. The frames contain a projective transformation of the board. As output, our system outputs a structured state of the game at each time point. Meaning, it knows where all the tiles are and what tiles contain player pieces. From this explicit information, it then needs to be able to determine the scores for each player, including both their current points and their expected points that they've earned but haven't officially tallied due to game mechanics.

2. Background / Related Work

There has been little work in the area of board game recognition and understanding. Of the work in this area,

most of it has been on the game Chess. Fortunately, the challenge of tracking Carcassonne shares some similarities with the challenge of tracking Chess, specifically board recognition and move tracking.

In CVChess [1], the authors first find corners using Harris Corners and then fit lines to the corners using a Hough Transform. To assign indices to the lines to recover board coordinates, they simply look at the x and y intercepts respectively of their horizontal and vertical lines. To track moves, they generate a heat map for each frame where areas of large differences have higher values.

In Automatic Grid Detection of Populated Chessboard Taken at a Lower Angle View [2], Tam et al. start with a Canny Edge Detector and then move to a Hough Transform to detect grid lines. They exploit the geometric properties of the grid and note that only the edges of 1 square are required to extrapolate to the rest of the grid.

3. Technical Approach

Our system is divided into 2 main components. First, there is the recognition module which given an image of the board can extract structured data about the state of the game such as where tiles are placed and what positions on the board players occupy. Then, there is the tracking module which will take in a video and track the state of the board game across frames.

3.1. Recognition Module

3.1.1 Recognizing the grid/board

Our goal is to get a structured model of the grid so that we can extract individual tiles. At a high level, we try to detect the corners of the tiles and label each corner with a coordinate that represents the row and column. In this section, we explore our method for how we accomplish this. Initially, our system has no information about the scene and we need to bootstrap the system with an initial guess of the layout of the grid. We start with an initial image taken by the camera.

Canny Edge Detector We start processing by applying a Canny Edge Detector to detect the lines of the grid. The raw results from the detector often produce 1 pixel thin lines. This tends to not work well with the Hough Transform in the next step so we dilate the edges with a 2×2 kernel to increase the thickness of the lines.

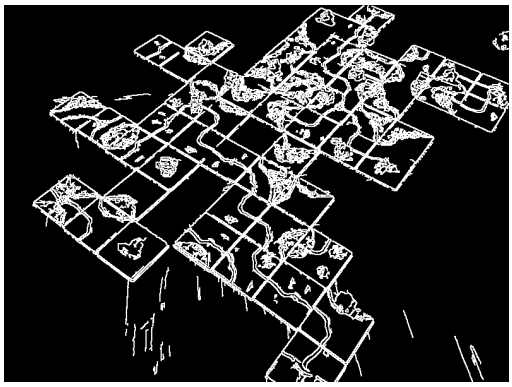

Figure 2. Example of canny edge detector output

Hough Transform Next, we use a Probabilistic Hough Transform to detect line segments that correspond with grid lines. From this, we can partially recover some of the grid lines in addition to some extraneous lines due to the patterns on the tile or general noise. We look for line segments of a minimum length to avoid detecting noise.

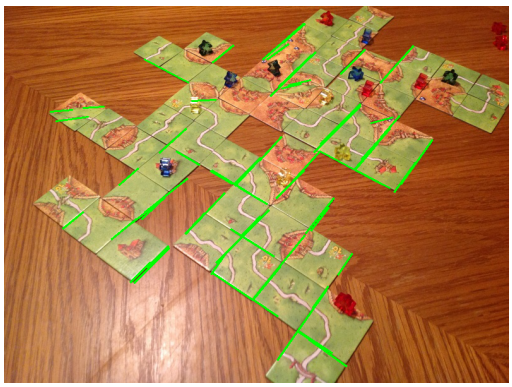

Figure 3. Example of hough transform output

KMeans To process these lines, we start by segmenting them into vertical and horizontal lines using KMeans clustering (N=2) based on their angles. After this, to eliminate obvious outliers, for each group of lines, we use RANSAC to compute the actual angle of the lines and throw out any outliers which deviate more than 10 degrees from the best angle found.

Meanshift After this, we use Mean Shift clustering to cluster line segments which are part of the same grid line. We do this by comparing their intersection points on a reference perpendicular line passing through the center of the image. We also assign ordered indices based on the location of these intersection points on the reference perpendicular line. Then, to find corners, we use a voting algorithm. For each pair of lines and their intersection point, we increment the score for the intersection point by

the length of the line segments. From this, we get lots of potential intersection points, most of which are clustered around the real intersection points.

Figure 4. Example of all intersections

Non Maximum Suppression To remove these clusters, we perform non maximum suppression for a fixed window size. Finally, we have a list of mostly real intersection points with a couple of outliers. Each intersection point is labeled with coordinates which will allow us to extract tiles and their positions on the board.

Figure 5. Example of intersections after non-maximum suppression

Homography Fitting Finally, we need to extrapolate to missing corners which were not detected in the above process. We do this by finding a homography using the set of corners we found. More formally, we set up the problem as follows:

$$P' = HP \tag{1}$$

where P' are corners in camera reference system and P are the corners in the board coordinate system.

To filter outliers, we use RANSAC. We solve the above equation with a different subset of points each time and keep the homography that has the most inliers determined by the reprojection error. With this, we should now have identified every possible corner in the image and can form a list of points that bound a possible tile.

3.1.2 Classifying Tiles

Once we have a bounding box for each tile, the first step is to unwarp the tile back into a square. We can do this by finding a homography between the unwarped coordinates and the desired warped image coordinates. Similarly to fitting the grid, we setup the problem as follows:

$$HP = P' \tag{2}$$

where $P = \begin{bmatrix} 0 & 0 & 64 & 64 \\ 0 & 64 & 0 & 64 \end{bmatrix}$ and P' is a matrix of coordinates of the corners identified in 3.1.1.

Using this homography, we get a 64×64 square image of a tile. First we need to determine if this image even contains a tile or is just an image of the table.

To distinguish between a tile and the table, we observe that the table has relatively little texture compared to the tile. We apply a Canny Edge Detector to the image and take the mean of the image. If the mean is below a certain threshold, we choose to classify as not a tile.

In the case that we think it is a tile, we proceed with template matching. We compiled a set of templates for each tile for use in matching. We use normalized cross correlation to account for various lighting conditions. To normalize images, we subtract the mean and divide by the standard deviation to have a mean of 0 and standard deviation of 1:

$$image[i,j] = \frac{image[i,j] - \frac{\sum_x^X \sum_y^Y image[y,x]}{X * Y}}{\sigma_{image}} \tag{3}$$

$$Score(tile, template) = \sum_y^Y \sum_x^X tile[y,x]\ template[y,x] \tag{4}$$

In addition, our matching must be rotation invariant. To accomplish this, since we have a relatively small amount of tiles, we use a bruteforce approach and perform template matching against the 4 possible rotations.

3.2. Tracking Module

The tracking module is generally responsible for taking data from the recognition module and maintaining a structured model of the game. It is also responsible for feeding key frames into the recognition module.

3.2.1 Monitoring for Changes

To identify key frames, we maintain the last frame that we processed and successfully turned into a move in the game. For every new frame that comes in, we get the absolute difference between the gray scale version of the last identified frame and the gray scale version of the new incoming frame. Using the homography calculated by the recognition module, we overlay a grid onto the difference. Since players are restricted to placing tiles only adjacent to other tiles, we only look at the differences at the positions adjacent to tiles we have previously detected. If we find that there is a large change localized to a single position where we predict a new tile could be placed, we identify the frame as a key frame.

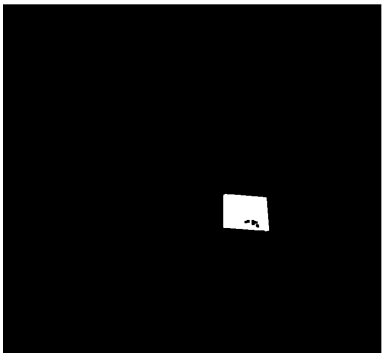

Figure 6. Example of difference heatmap. Everything but the newly placed tile will be black.

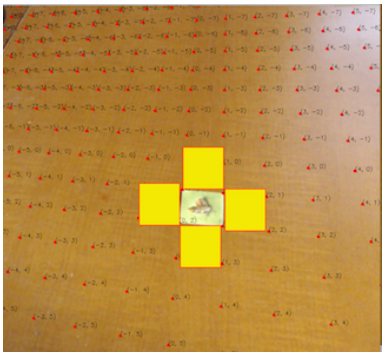

Figure 7. The system only pays attention to locations adjacent to existing tiles.

3.2.2 Integration with Recognition Module

Once a key frame has been identified, we look at the heat map to determine where a new tile has been placed. We iterate through the possible locations of new tiles given constraints due to game rules and choose the position which has the largest difference. To segment the tile that was just placed, we also reference the heat map. We threshold the heat map to remove noise and then find the largest blob on the image and fit a rectangle to it. We then take the four corners of the rectangle and hand it off to the recognition module. The recognition module adds these points to its list of known corners and generates an updated homography. It then classifies the tile using the method outlined in 3.1.2. To aid in classification, the tracking module leverages its knowledge of game rules to generate known constraints about the new tile. Since players are required to place tiles adjacent to other tiles and tile edges are required to match, at minimum we know the feature (e.g. grass, city, road) of one edge (see figure 8). The recognition module then can limit its range of possible classifications to tiles that have this feature on the specified edge.

Figure 8. The tile on the left is an existing tile. Based on this, we know any tile placed to the right of it must have a city feature on its left edge.

3.2.3 Detecting Player Pieces

Once a player has placed a tile down, they may optionally choose to place a player piece down. They may only place a piece down on the tile they just played. Based on this constraint, a simple solution to detecting this is to use the heatmap we generate. If we notice a spike in a location where a player just placed down a tile, we know this must be due to the player placing their piece down.

3.2.4 Scoring

At this point, our system is aware of all the tiles and player pieces on the board. The system translates scene changes into moves using the data recovered in 3.2.1-3.2.3. We feed these moves into an internal model of the game which performs a breadth first search across the grid to find any completed features. This part of the system is unrelated to the main computer vision task at hand, so we leave the process of how to do this exactly as an exercise to the reader if they are interested.

An interesting interaction in the game is when players implicitly place a player piece when they lay down a tile and then immediately complete a feature, score points, and then remove their piece since the feature has been completed. We emphasize that this is implicit because players often do not physically move their piece onto the board and remove it. This often is communicated verbally. Thus, if we were to rely solely on the video feed, we would miss this interaction. However, we can leverage our understanding of the game and its norms to recover this information. If we detect that a player places a tile that completes a feature and does not place a piece down anywhere else, we automatically assume the player is playing rationally and intended to place their piece on the completed feature and score points. Thus, this is a way that our system demonstrates its understanding of the game by recovering a move that it didn't even see.

4. Experiment

Below is a picture of our experimental setup.

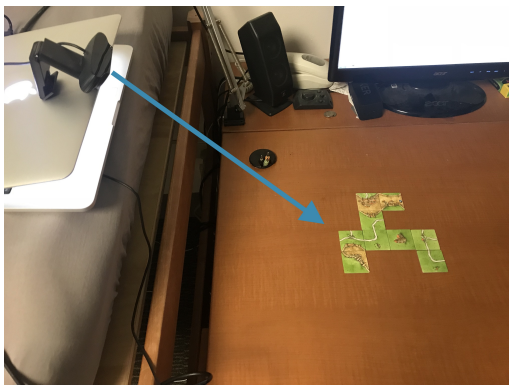

Figure 9. Picture of experimental setup

We took 30 tiles and 5 pieces and placed them one by one randomly (given the rules of the game) and recorded various accuracy metrics. We performed this test 5 times.

4.1. Positions on grid

Test	Accuracy
1	100%
2	100%
3	100%
4	100%
5	100%

Table 1: Accuracy for positions on grid

Our accuracy for detecting where tiles are placed is pretty much perfect. This is due to the fact that tiles are so big which means we have a large margin of error before our system decides that a tile is a whole position away from where it is supposed to be. Qualitatively however, we notice that our accuracy in terms of predicted bounding box vs actual bounding box tends to drift as we place more and more tiles. This is likely due to the fact that we assume our grid is perfect. For the most part, players do not egregiously misplace tiles so this assumption is fine. But as more and more tiles are placed, the grid placement errors start accumulating. Our fitted grid using a single homography cannot account for the random errors.

4.2. Tile Classification

Test	Num Accurate	Accuracy
1	25	83%
2	27	90%
3	24	80%
4	25	83%
5	28	93%

Table 2: Accuracy for classification of tiles

Our tile classification accuracy is good for such a basic template matching algorithm. Most of the errors come from the same 4-5 tiles which are often confused with each other.

Figure 10. Example of often confused tiles

See Figure 10. The common trend is that the system fails to detect roads. This is a limitation of our template matching approach. For one, the brightness of the roads and grass are very similar when converted to gray scale, so normalized cross correlation does not penalize mismatches heavily. This is exacerbated by an overexposed image which washes out colors and makes the road and grass look very similar. Second, since the roads are so small, a small translation due to an imperfect bounding box will cause the roads not to overlap when the raw tile and tile template are overlaid. City features do not suffer from this problem due to their size.

With a better camera and better lighting, these problems could be alleviated and could be not as pronounced.

4.3. Piece Detection

Test	Accuracy
1	100%
2	100%
3	100%
4	100%
5	100%

Table 3: Accuracy for piece detection

Piece detection, like position detection, is also perfect. We don't need to do any sort of classification for the piece since we know whose turn it is. All we require is finding the location of a difference spike which means if grid position accuracy is good, this metric will also be good.

4.4. Number of Moves Until First Incorrect

Test	# until first incorrect
1	24
2	25
3	22
4	23
5	27

Table 4: Number of Moves Until First Incorrect

If we compare the number of correct moves until the first incorrect move to the number of tiles classified correctly, we see a correlation. Due to the fact that we generate constraints based on previously classified tiles for classification of new tiles, errors snowball. Once we have misclassified a tile, any new tile adjacent to it has a high probability of being misclassified due to incorrect constraints. Thus, once we have misclassified one tile, as time goes on, the chance that we make more errors increases dramatically.

4.5. Scoring

We intentionally do not discuss errors in score here because they are calculated from the above data. The errors are 100% correlated with any errors from the above.

5. Conclusion

In this paper, we demonstrated a proof of concept. There are some interactions in the game that we did not have time to model or there are some simplifications that we made. For example, for piece detection, we only identified which tile a player placed their piece on. However, we actually need to know which feature on the tile the player has placed their piece on. For some tiles with only 1 feature, this is trivial. For those with 2, it's slightly more complicated.

In the future, more work could be done in the area of tile classification. We currently use a template matching based approach. However, this requires the system to have a ground truth image of each possible type beforehand. It cannot generalize to tiles with arbitrary feature arrangements. A tile classification module that could classify tiles based on their features would be a good extension. For example, instead of classifying a tile as "Tile A", we could classify a tile as "Has road on left. Has grass on top. Has city on right. Has grass on bottom." This would eliminate the need for template images and reduce the amount of priors required for the system.

In addition, with a complete system that tracks all aspects of the game, a logical progression would be to build an AI that is capable of playing the game in real time with the aid of a video feed. Or even simpler, it could just be

a player helper that analyzes odds of completing a feature given the remaining tiles in the game.

In conclusion, this project starts a solid foundation to solve the problem of tracking Carcassone completely. The hardest problem of tracking tile placements has been solved. The author hopes to continue this work for fun in the future and eventually get to a more accurate system that can track all aspects of the game with no simplifications.

Acknowledgments

The author would like to say thank you to the course staff, Prof. Savarese, Amir Sadeghian, Kuan Fang, Danfei Xu, Fei Xia, and Julian Gao, for teaching this course and helping the author with his project. It was a great quarter and the author learned a lot.

References

[1] P. R. Jay Hack. Cvchess: Computer vision chess analytics, 2015. 2

[2] J. A. L. K.Y. Tam and D. Levy. Automatic grid segmentation of populated chessboard taken at a lower angle view. *IEEE*, pages 294–299, 2008. 2