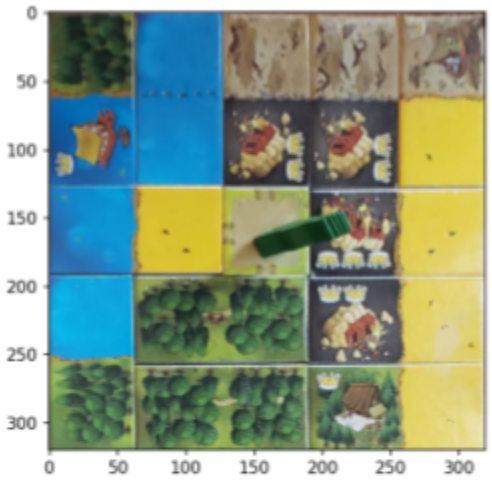
Automatic Scoring of Kingdomino

CS445 - Rick Bischoff - rdb4@illinois.edu rdbisch@gmail.com

This project takes candid pictures of the board game *Kingdomino* and computes the score according to that game's rules. My original motivation for doing this was to learn how to do homography and image classification in a practical (yet silly and fun) application.

The board game itself for the uninitiated is quite simple. Players take turns choosing from a set of dominos to add to their "kingdom", which will ultimately be a 5x5 grid. Each domino is a 1x2 rectangle, and each side of the rectangle has a distinct "terrain" -- one of Forest, Desert, Ocean, Grass, Swamp, and Mine. Further, each tile can have 0, 1, 2, or 3 "crowns", which are scoring indicators. The score of your final kingdom is the sum over each contiguous region of like-terrain, the number of crowns multiplied by the area.



Score for Board

Region	Tiles	Crowns	Subtotal	Tiles
forest	1	0	0	A1
ocean	5	1	5	B1,B2,A2!,A3,A4
swamp	3	1	3	C1,D1!,E1
mine	4	7	28	C2!!,D2!!,D3!,D4!!
desert	4	0	0	E2,E3,E4,E5
desert	1	0	0	B3
forest	6	1	6	B4,B5,C4,A5,C5,D5!
Total			42	

Figure 1 - Demonstration of Kingdomino scoring.

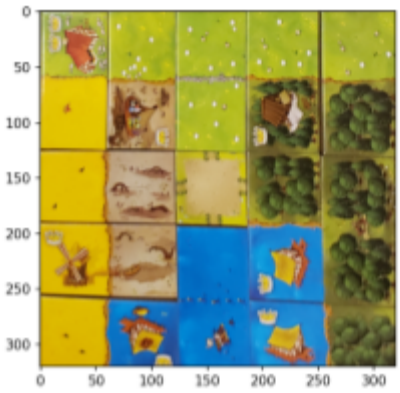
General Approach

Given a candid photograph of the playing area, the user is instructed to click the four corners of their game board. Then, a perspective transformation is calculated that maps this game board onto a square area. Using the fact that the dominos are regular sized, the algorithm naively breaks this image up into a 5x5 grid. Each cell of the grid is run through an image classification algorithm to predict both the terrain type and the number of crowns. Then a simple algorithm is used to score the board and output the table.

An additional bell & whistle allows the user to manipulate their photograph to “cheat”. The user can select from an arbitrary choice of dominos from the game, and using the inverse of the perspective matrix above, as well as some rudimentary color correction, insert the domino back into the original photo. This can be used to fool your friends that you are a master of *Kingdomino.*

Figure 2 - Artificial dominoes were inserted into this image

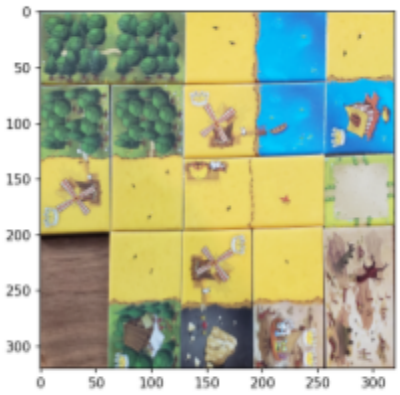
Further Examples



Score for Board

Region	Tiles	Crowns	Subtotal	Tiles
grass	6	2	12	A1!!,B1,C1,C2,D1,E1
desert	4	1	4	A2,A3,A4!,A5
swamp	3	1	3	B2!,B3,B4
forest	6	1	6	D2!,D3,E2,E3,E4,E5
ocean	5	3	15	C4,C5,D4!,B5!,D5!
Total			40	

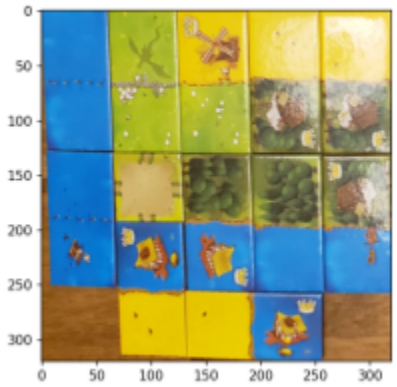
Figure 3 - Yet another example -- Clockwise from top-left
Candid photo, top-down reprojection, "Cheat" photo, and scorecard for original



Score for Board

Region	Tiles	Crowns	Subtotal	Tiles
forest	4	0	0	A1,A2,B1,B2
desert	9	3	27	C1,C2!,C3,C4!,B3,D3,B4,D4,A3!
ocean	3	1	3	D1,D2,E2!
desert	1	0	0	E1
swamp	3	2	6	E4,E5,D5!!
forest	1	1	1	B5!
mine	1	0	0	C5
Total			37	

Figure 4 - Example of an incomplete board. Bad result from “cheat” algorithm due to bad perspective transformation - it is hard for the user to click on a corner that doesn’t exist.



Score for Board

Region	Tiles	Crowns	Subtotal	Tiles
ocean	9	3	27	A1,A2,A3,A4,B4!,C4!,D4,D5!,E4
grass	3	0	0	B1,B2,C2
desert	3	1	3	C1!,D1,E1
forest	5	3	15	D2!,D3,E2!,C3,E3!
desert	2	0	0	B5,C5
Total			45	

Figure 5 - Another exampe of an incomplete square

Details

All details are in the appendix but omitted here for brevity.

Guide to Code Base and Instructions for Reproduction

All code and data are contained in the same git repository, available to the public at https://github.com/rdbisch/cs445_final_project

Step by Step to Reproduce Examples

Unix and python 3, with tensorflow/keras, openCV and numpy were used to generate this project. With the exception of “cvtColor” and “resize” no other openCV functions were used if possible.

There is nothing specific to Unix here, but I do not know the same commands in Windows to replicate. Please contact the author for assistance if problems arise.

Reproduction from ZIP File Attached to Cloudera

Please note this archive is severely limited and it is recommended the github repository is cloned instead. If that is not possible:

1. Unzip zip archive into a local directory, say, `cs445_final_project_short`
2. Change directory

```
cd cs445_final_project_short
```

3. Create a new virtual environment and activate it.

```
python3 -m venv rdb4_venv

source rdb4_venv/bin/activate
```

4. Install required PIP packages

```
python -m pip install -r requirements.txt
```

5. Start Jupyter Notebook and open up FinalScoringExample.ipynb

```
jupyter notebook
```

Note due to space limitations the zip-file only contains one example.

Reproduction from Github

1. Clone git repository into a local directory, say, `cs445_final_project`.

```
git clone https://github.com/rdbisch/cs445_final_project.git
```

2. Change Directory to the project directory.

```
cd cs445_final_project
```

3. Create a new virtual environment and activate it.

```
python3 -m venv rdb4_venv

source rdb4_venv/bin/activate
```

4. Install the required pip packages.

```
python -m pip install -r requirements.txt
```

5. Start Jupyter Notebook and open up FinalScoringExample.ipynb

```
jupyter notebook
```

After Installation

Note that the notebook is meant to be used interactively. Running all Cells at once will not work. After you have reviewed the notebook and are ready to run your own example, follow these steps:

1. Kernel menu / Restart & Clear Output
2. Evaluate the first six cells -- all the way down to “Start Here!”
3. You can either uncomment one of the provided training images, or reference your own. Simply change the filename to point to the right location. Evaluate this cell. You are then prompted to choose 4 corners of the game board.
4. The next four cells are optional--they let you fine tune your selection. More accurate points can lead to better results.
5. Evaluate the remaining cells down to “Cheat!” and it will project your game board and then run an automatic scoring routine on it.

If you are interested in running “Cheat!”, do the above first. The first few cells do color matching. You will be required to pick the domino you want to replace--it is asking you to pick the top-left domino. The next cell is a 10x10 grid of all possible dominos, pick a left-side and a right-side by clicking twice. If you want the domino to be rotated, comment out the right option in the next cell.

The rest of the cells can be evaluated from there. The cheat board is reprojected onto your original photograph and a new score is calculated.

Core Files to Reproduce Examples

`requirements.txt` **-** Package requirements to recreate the virtualenv used to run this project.

`FinalScoringExample.ipynb` - A Jupyter notebook used to demonstrate the application and used to generate figures for this report.

`TileUtility.py` – An abstraction to a database of the pristine tiles as well as housing the inference of predictive models.

`processed_tiles.json` - A database of information about tiles. Used by TileUtility.

`processed_tiles/` - This directory contains the pristine overhead view of tiles.

Other Files of Interest

`raw_tiles/` - The original photographs of individual tiles.

`dev_images/` - Training examples for model build and for exhibition.

`training_images/` - Training data extracted from other programs below

`Process_tiles_dev_notebook.ipynb` - Jupyter Notebook used to develop the subsequent file.

`Process_tiles.py` - Final script to project raw_tiles/ into the files seen at processed_tiles/

`ExtractTrainingTiles.ipynb` - Notebook used to house and extract labels from dev_images/

`AcceleratedLabelMaker.ipynb` - Productivity notebook used to quickly correctly label model mistakes

`BoardDetector.ipynb` - Attempt to make a board detector

`BoardScratch` and BoardScratch2.ipynb - Working Jupyter notebook used to try out new techniques.

`exportTilesJson.py` - Script used to create Processed_Tiles.json

`cs445 final project proposal.pdf` - Original proposal for this project

`cs445_final_project.pdf` - This document

Conclusion

I achieved what I set out to achieve, in that I created some code to take candid photos and score the game. The result is far from perfect, but it is passable. The image classification isn't perfect, and frequently runs into trouble on "Swamps". I would also like to replatform this onto a web-app so that the larger public could use it, as opposed to a clunky Python notebook. And finally the color correction in the "cheat" mode is very ad-hoc, and while passable, is far from ideal.

Acknowledgements

Besides the CS445 course material itself--lectures, MP utility code, etc., I also used:

- OpenCV2 documentation
- Wikipedia for Hough Transformation
- http://www.cse.psu.edu/~rtc12/CSE486/lecture16.pdf
- Keras tutorial for MNIST

Appendix

Ground Truth Data Collection

Using a tripod with the camera pointing nearly straight down, I attempted to capture a non-distorted overhead view of all 48 dominos. This proved futile, however, as slight bumps threw off any hope of redemption.

Instead of spending 20 minutes retaking photographs, or 40 minutes editing by hand, I instead wrote a script over 10 hours to do it.

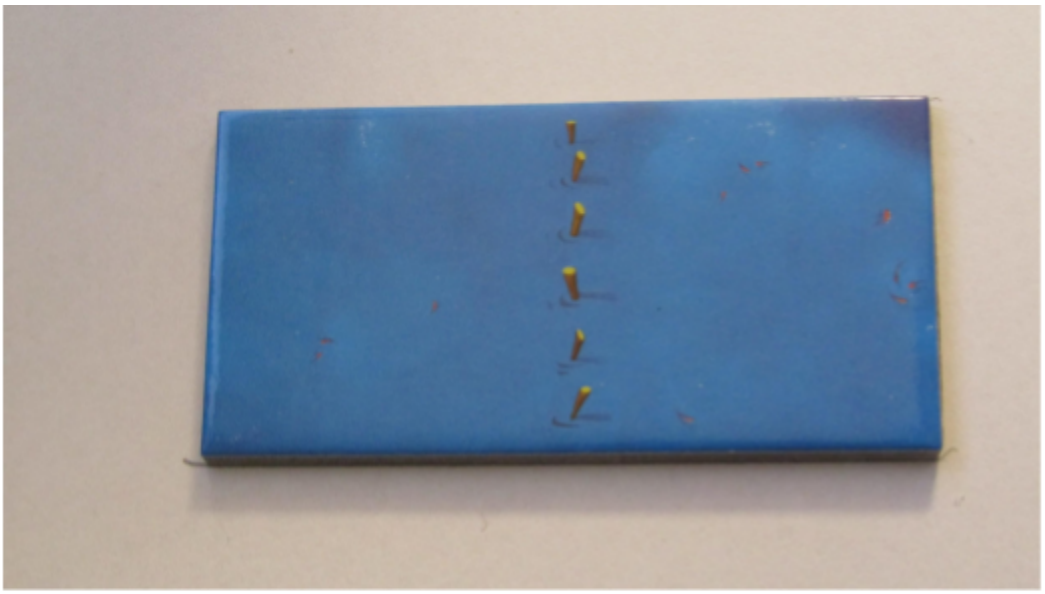


Figure 6 - Raw unprocessed image of a domino.

Because the background was so bright, I was able to use a simple thresholding and manual adjustment to detect the domino. Anything above the mean intensity was masked out, as well as anything darker than 2 sigma below the mean.

The mask was then blurred with a simple average filter, and then thresholded again. This result was subtracted out to make a crude edge detector. The motivation here was to avoid stray pixels giving false positives on the extents.

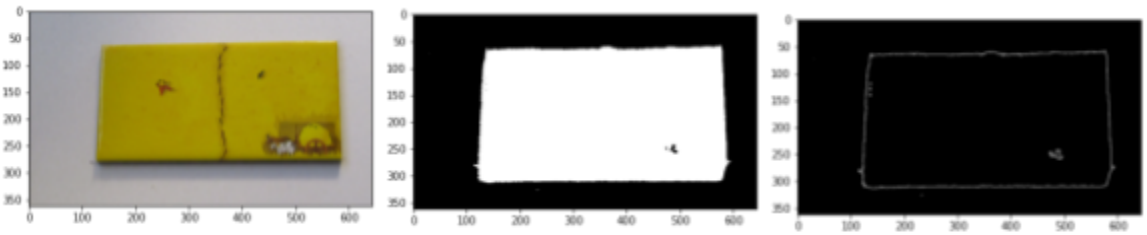


Figure 7 - Example of thresholding and mask calculation. Left ro right -- original tile, threshold mask, edge detection

Next, the mask is divided into quadrants and the extreme point of the mask is found. These extents form the quadrilateral ABCD. The goal is to extract this quad and map onto a rectangle of fixed dimension. Figure?? illustrates.

Linear maps are then constructed using straightforward linear algebra. In particular, we create two affine matrices P and Q that map points ABCD onto a fixed rectangular grid. Both matrices have 6 unknown parameters, and solving them required coordinates from 3 points to achieve. Fortunately triangles have three points.

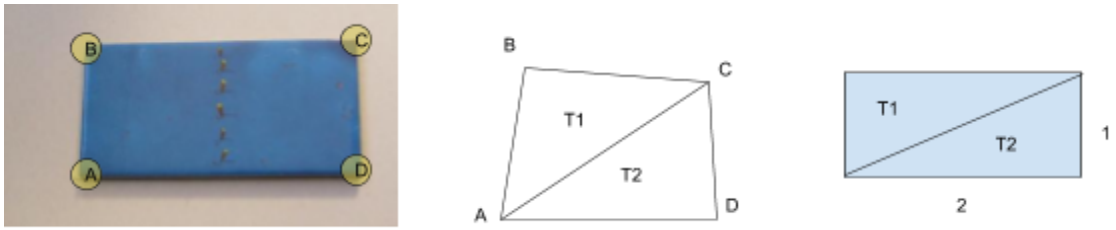


Figure 8 - Illustration of goal of ground truth tile importing

For example, let P be the desired affine matrix translating triangle ABC from the image onto the ideal rectangle.

$$P = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}, \text{ and}$$

$$P \begin{bmatrix} A_x & A_y & 1 \end{bmatrix}^\intercal = \begin{bmatrix} 0 \\ R \\ 1 \end{bmatrix}$$

Similarly,

$$P \begin{bmatrix} C_x & C_y & 1 \end{bmatrix}^\intercal = \begin{bmatrix} 2R \\ 0 \\ 1 \end{bmatrix}$$

These can be done for the other points similarly. These equations form another system of equations in the unknowns P, which are then solved for in the usual way

$$A \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} p_{00} \\ p_{01} \\ p_{10} \\ p_{11} \\ p_{20} \\ p_{21} \end{bmatrix}$$

$$A = \begin{bmatrix} A_x & A_y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & A_x & A_y & 1 \\ B_x & B_y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & B_x & B_y & 1 \\ C_x & C_y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & C_x & C_y & 1 \end{bmatrix}$$

Where A_x, B_x, etc are the coordinates of the quad found earlier, and p_ij is the ith corner of the ideal rectangle and j indexes the x- or y- directions.

After P is recovered by solving for a-f, and identically for Q, we then invert P and Q to find the inverse mapping of the ideal rectangle onto the image. Then using interpolation, the new image is formed.

This worked well enough in practice. Some manual clean up was done to handle special cases. Figure ?? shows a few examples.

Figure 9 - Examples from process

These examples were generated during early days of working on the project. I expect that the extent detector will get better with more work, but I will not update here as improvements are tangential to project results.

Update: Of course after I develop this, I watch the Week 11 lectures and Prof. Hoiem shows how to do homography the right way. The final version of board mapping uses the homography discussed in the lecture,with further detail provided by http://www.cse.psu.edu/~rtc12/CSE486/lecture16.pdf

Of particular pride for me… While I could have just used opencv built-ins to do this, I wanted to see if I could write Python code that provided similar functionality. My first attempts worked fine, but they were about 1000x slower than OpenCV, and heavily depended on numpy "griddata" function. On further reflection I realized that because the target was a perfect rectangle, I was thinking about the problem backwards, and it was a simple average that could be facilitated by a simple aggregation. See "FinalScoreExample.ipnyb", in particular the function "fourPointExplodingHeartHomography"

Terrain and Crown Identification

With a set of ground-truth tiles, I sought out to create a simple classification task to identify the terrain type of a tile, and the number of crowns. Breaking the problem up into two sub-components, rather than “identify this particular tile”, was thought to be more tractable and straightforward.

Because the processed_tiles are in a consistent format, I broke up each tile into halves by separating it right down the middle.

Terrain Identification

I hit success straight away with terrain type using the average RGB value of each terrain type. First, I calculated the average RGB using the known terrain type. Then I plotted these in a 3D scatter plot to see if visually the clusters were distinct. See Figure ??

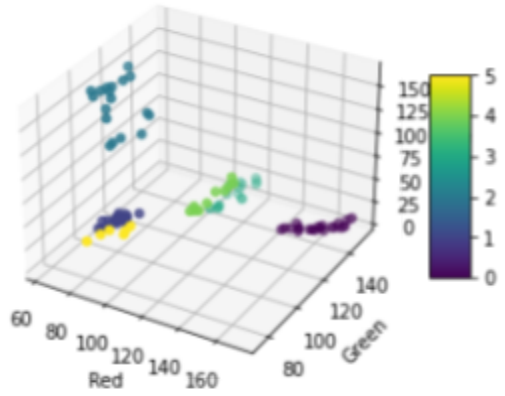


Figure 10 - RGB centers of different terrain types (0,1,2,3,4,5)

Other than the yellow (mines) and dark blue (forests) clusters, the other 3 are very distinct. As figure 10 Shows, visually there is a very distinct difference between these two, and it was somewhat surprising that these were right on top of each other in the scatter plot.

Figure 11 -- The left tile shows the “Forest” terrain, and the right-hand side of the right tile shows the “Mine” terrain.

Zooming in on this plot, and by looking at only two channels at a time, it appears the overlap is perception only. There is a clear boundary between the two as figure ?? Shows.

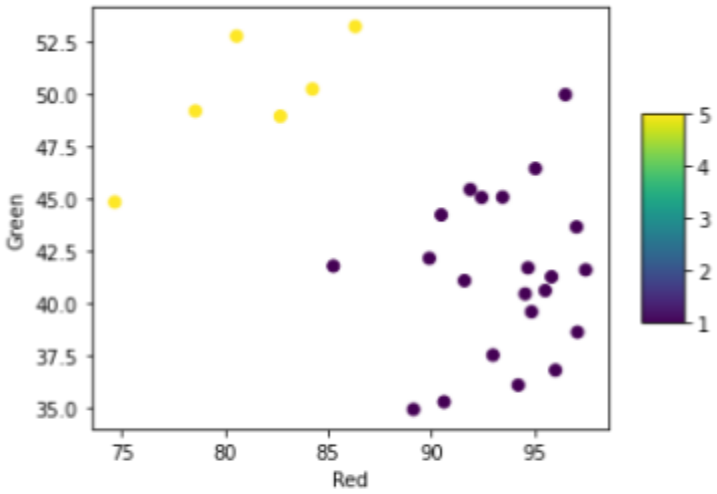


Figure 12 - Average Red and Green channels shown for Forest (dark blue) and Mine (yellow) terrain tiles.

Using the observed centers of these clusters as the true centers, I then tried to “predict” which terrain a given tile had based on how close the average R, G, B was to these centers. The result was 100% accuracy.

At this point I have not tested on real images of a Kingdomino playing board and do not know if the result will hold up under different lighting or angles.

Crown Identification

An attempt to use the same RGB clusters trick yielded no discrimination between tiles with crowns vs not. Figure 13 shows visually no discrimination apparent from RGB channels, or RG, or even (R+G) vs B.

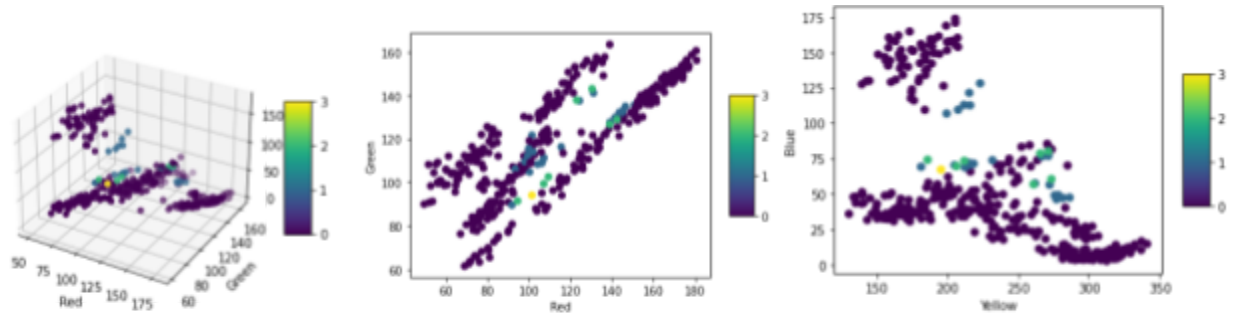


Figure 13 - Failed attempts to visually find clusters correlated with the number of crowns on an image.

The next approach was to first subtract out the average values for the predicted terrain. This produced better results, but didn't result in a perfect discriminator. Figure ?? shows the details of that approach. Ultimately this was doomed to failure as the crown has the same color regardless of underlying terrain, and subtracting out terrain will change the color.

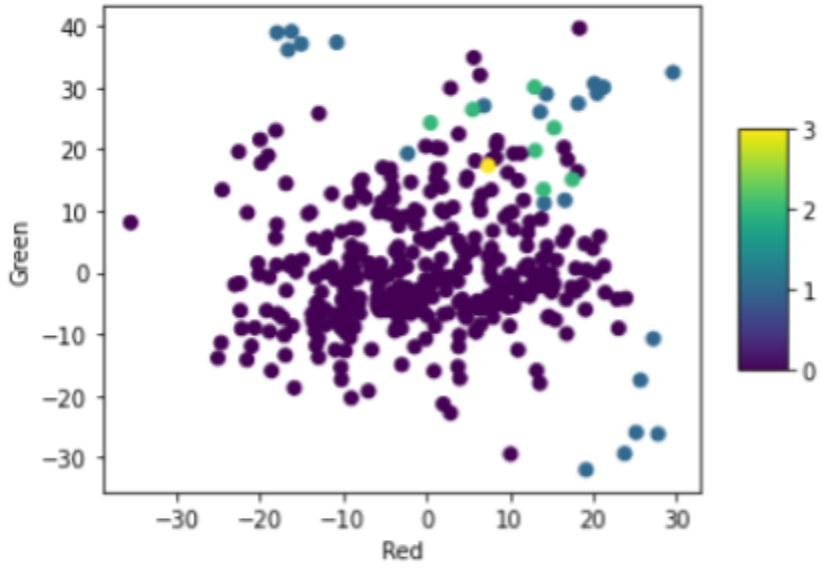


Figure 14 - Crown identification. First the underlying predicted terrain's cluster center R', G', B' was subtracted out from the image. Then the average values of R-R', G-G', and B-B' were plotted against the number of crowns. You can see most non-zero crowns on are on the periphery of the scatter plot.

Remembering the loss cost function from MP2, I decided to give that a shot. Using a hand-cropped and cleaned up crown from one of the processed tiles, I created a mask and template and used the loss function from texture synthesis. Then I plotted the extreme values of those loss functions for all tiles by crown present or not. Figure?? Confirms there is a clear threshold in these processed images to detect crowns!

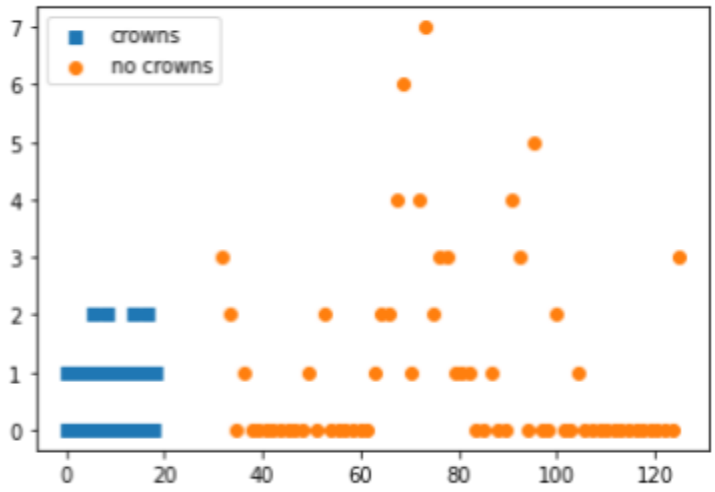


Figure 15 - Count of images vs. extreme value of loss function. Clear deliniation between tiles with crowns and tiles without.

Next I built a wrapper function to use this knowledge to identify and count the crowns on an individual tile. The algorithm uses the threshold from the scatterplot (here 19, but later tuned up to 30), and finds all likely pixels where the crown might be. Next, an algorithm runs through and eliminates any duplicate pixels within a radius of 5-pixels from each other--the pixel with the lowest loss is kept. This procedure has 100% accuracy and recall on the ground-truth images. Figure ?? shows some examples of how it works.

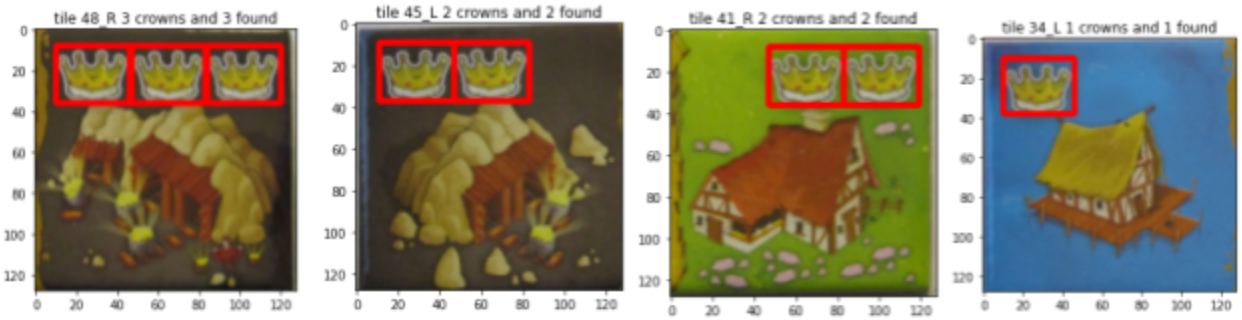


Figure 16 - Examples of crown identification using filters on select tiles.

Deep Learning

On playtesting both of the methods (RGB clustering, and crown identification via template) proved to be too sensitive to lighting conditions to work well in practice. Training samples provided roughly 50% accuracy, not nearly enough to be a useful tool. Some additional work went into trying to get these techniques to work--generic clusters, different color spaces, automatic color balancing, etc. But ultimately they didn't work.

The core problem was that the tiles for this game are extremely glossy and reflect lots of light that make naive approaches work poorly.

As I never intended deep learning to be the key component of the project, I settled on using the Keras MNIST tutorial model architecture (https://keras.io/examples/vision/mnist_convnet/) for both predicting crowns and terrain. Later on as performance issues with that arose, I modified to a slightly different architecture that works well, albeit not-perfectly.

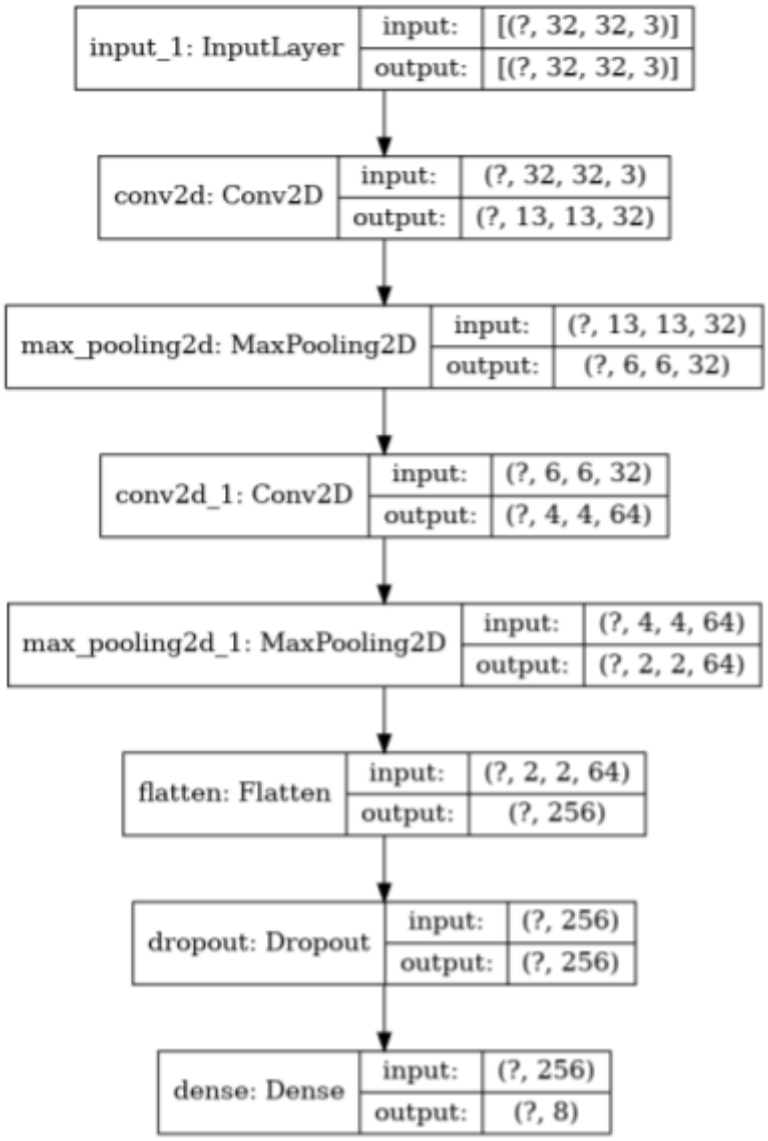


Figure 17 - Keras model architecture for terrain prediction

The crown model had an identical architecture with the exception of the final layer.

The bigger challenge of this step was labelling data and keeping track of encodings (1=Forest or Grass? Depends on which file you have open…) I spent many hours collecting photographs, recording corners of game boards, and painstakingly labelling them to use as training samples. All of these labels are collected in “ExtractTrainingTiles”. Later, I wrote

some helper applications to take new photographs, label them, and have the program output only the “mistakes.” This technique is commonly referred to as “weak-supervision” and ultimately proved to be very impactful to model training.

All of the training examples were replicated in multiple orientations to ensure that the resulting model didn’t retain a concept of “right-sidedness.” This was important because players can rotate their dominos any way they desire.

Of further note, the terrain types in the training set, as well as the game itself, are imbalanced. There are a lot of deserts and comparatively few “mines”. Similarly, crowns are “rare”, and even so for tiles with “2” or “3” crowns. This was fixed by oversampling the rare cases. This was a bit sloppy as I overbalanced terrain, and then overbalanced crowns. With some further thought, it should be possible to do both simultaneously.

The Model Training itself happens in “ModelTraining” and the output is saved into a subdirectory for later use by “TileUtility” which provides some convenient abstractions.

Board Detection

Having the user click the four corners of the game board is not a good user experience. I set out to automatically detect the game board from the photograph. Ultimately I failed at this task, but the below documents the journey.

The idea I had for this was that the lines from the dominos should be strong and dominate in the picture. I could use these lines as strong hints for the board, and use the intersection points of the outside lines as the corners.

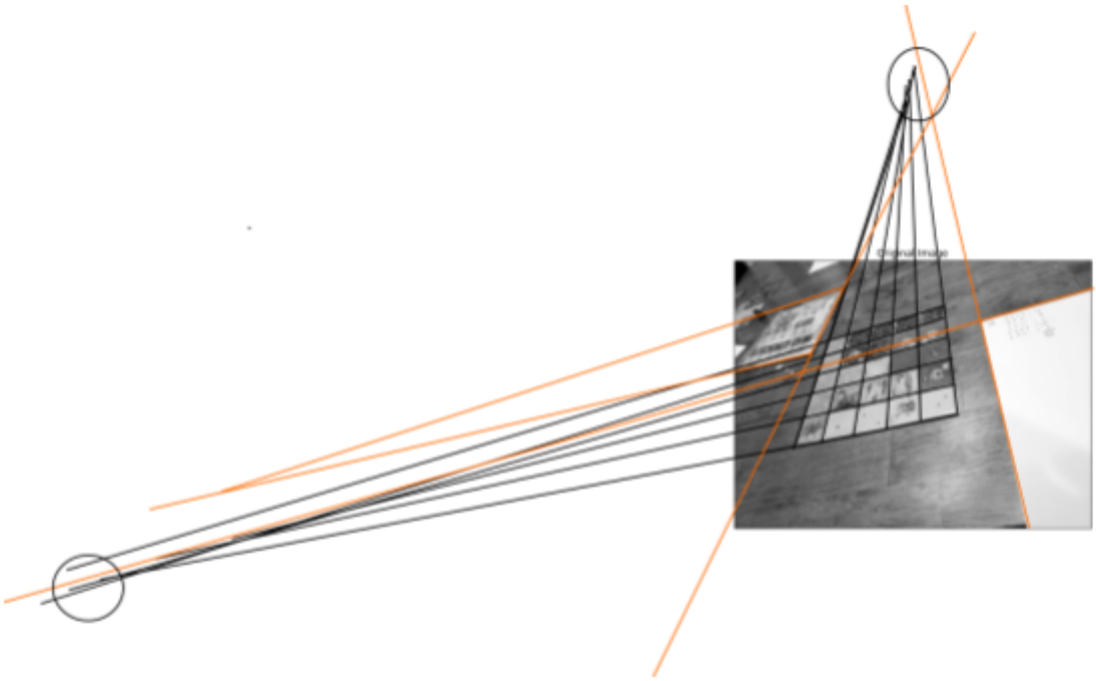


Figure 18 - Conceptual Drawing of Board Detection

In practice, I could not find the magic combination of parameters for edge detection or line detection to make this work. The lines of the background tables, walls and windows proved to be too dominant, and relaxing parameters allowed too much noise through. A working Hough transformation tool was built, as well as code to find common vanishing points. Figure 19 shows some of the resulting artifacts.

Though I didn't finish this part, this feature would be an absolute requirement in a non-academic environment.

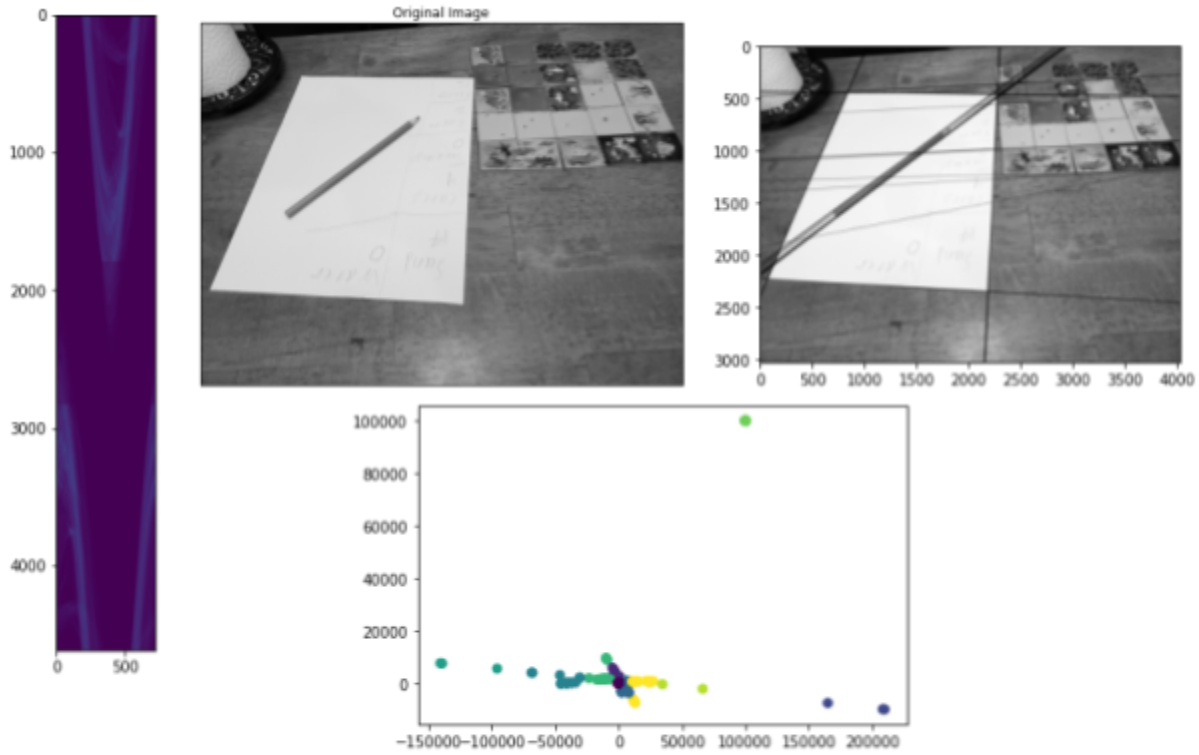


Figure 19 - Attempt at Board detection. From Left to Right: Hough transformation, source image, detected lines drawn to infinity, points of intersection (colors show k-means cluster assignment)

Cheating

To develop the cheating feature, I had to solve a few problems.

The first is, the dominos itself are incredibly glossy, so much that the ambient lighting of the scene has a big impact on the coloring of the domino. In order to measure this light, I used the “pristine” labels I collected in the project under a consistent lighting. To do this, I use the predicted terrain and crowns of a specific domino and find some suitable candidates that match. Then I generate an artificial board that matches the kind and quality of the original board. See Figure ??

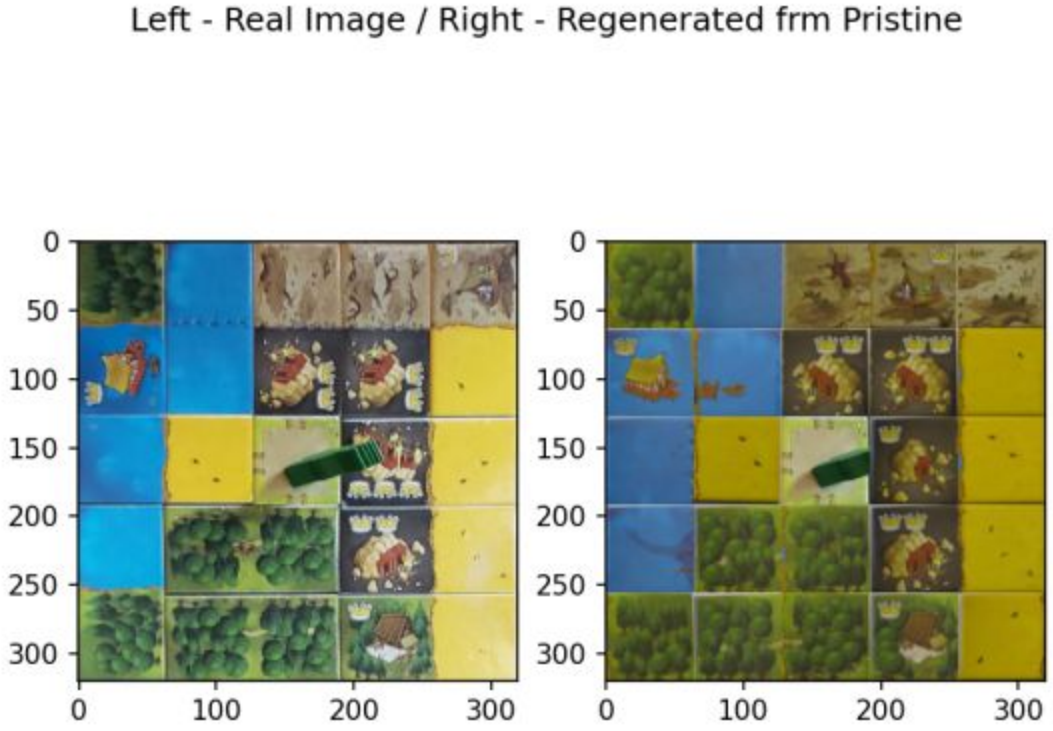


Figure 20 - Reconstruction of a scene from candid photograph using pristine photos of dominos

Because the dominos itself don't match up pixel-to-pixel (e.g. compare tile A2 on both images), I had to resort to using an average difference of colors. To do so, I converted both images to the Lab-color space and matched the luminance channel multiplicatively. I also tried using additive corrections, and only using the L- channel, but using all three multiplicatively produced the best results.

After the user selects the domino to replace, the color correction is done again and then inserted back into the square grid. Then, using the inverse perspective transformation, that domino only is placed back into the source image.

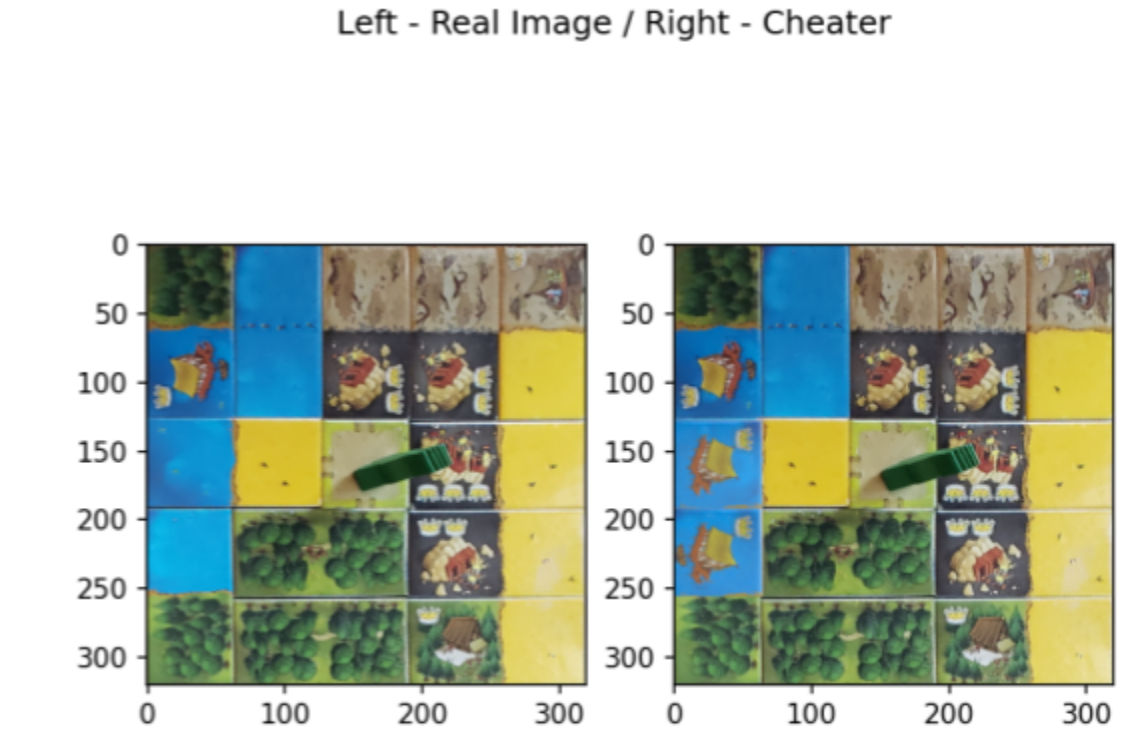


Figure 21 - Inserting Cheat Tile into overhead view.

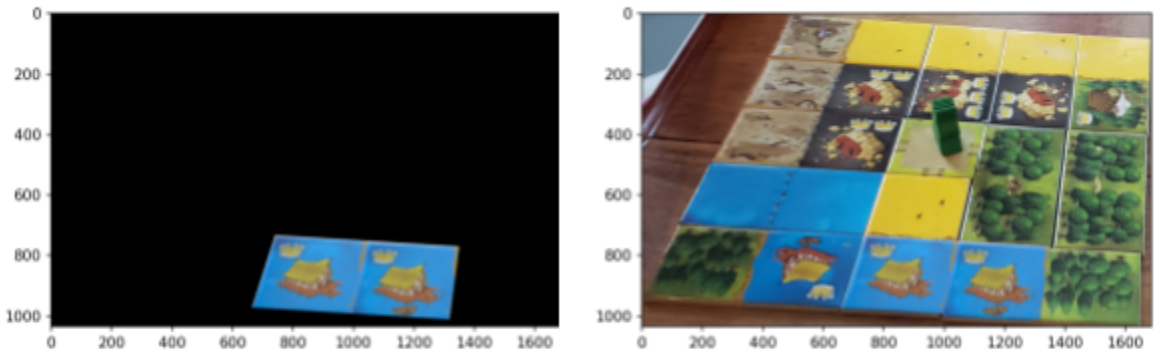


Figure 22 - Mapping back to the perspective view

This is far from perfect and would be an area of further research. In particular, exact matching of the domino should be straight forward, as well as the orientation. Once this

happens, a light-map could be reconstructed on a pixel-by-pixel level and applied to any overlay. This seems to be the only feasible way of capturing the specular reflection from the dominos due to their glossiness.