Sketch 2 Graybox: A Tool for Rapid 3D Video Game Level Prototyping

Cole Sohn
Stanford University
csohn@stanford.edu

Abstract

Sketch 2 Graybox is a tool for 3D game designers to automatically generate a graybox prototype level from an image of a top-down level layout drawing [Fig 1]. This approach for level prototyping aims to speed up game development by removing the bottleneck of modeling 3D graybox levels. It allows the designer to control wall and floor geometry, as well as the placement of gameplay actors based on the contents of their sketch. This is acheived through computer vision techniques such as edge detection and contour finding, as well as drawing classification through a multi-class non-linear SVM classifier trained on the Google Quickdraw Dataset.

1. Introduction

To build a level for a 3D video game, the game designer first draws a sketch: a top-down level layout on paper to fulfill a set of gameplay objectives. Next, the designer models a 3D prototype level, often called a blockout or graybox level, which matches the 2D layout. This prototyping process allows the designer to playtest the level and get feedback to assess if the level fulfills its objectives. Oftentimes, the level fails and the designer returns to the drawing board to iterate on their initial sketch or start over. The designer repeats this process until the level design fulfills its objectives. This process is illustrated in [Fig 2].

The modeling of graybox levels can be a bottleneck in the level design process. It slows the designer's ability to iterate because with every unique layout the designer has to reconstruct the level in 3D. This discourages significant changes due to the extra work generated. The designer may prefer to tweak a level they have already modeled over starting from scratch. Additionally, this can be a costly process for a game studio as an art department may be waiting for a completed level design to begin their work shaping the final level.

This project addresses the bottleneck problem by automatically generating level grayboxes from sketches, with results as seen in [Fig 1]. This approach aims to boost creativity by allowing the designer to instantly playtest their designs and iterate without 3D modeling slowing down the process. It can also save money by speeding up the level design process and allowing artists to begin working on levels earlier in production.

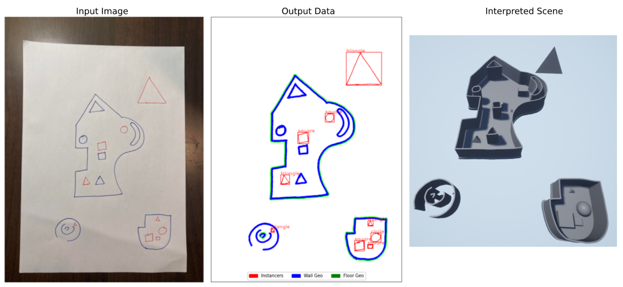

Figure 1. Example of Sketch 2 Graybox Results

An image of a top-down level layout drawing is converted to a graybox level as such. First, relevant data is extracted from the image such as level contours and instancer object bounding boxes. Secondly, 3D static level geometry and instance points are generated from that data. Thirdly, static level geometry is processed and actors are transformed to the instance points and scaled appropriately [Fig 6]. This whole process is packaged into the game engine such that the level designer only needs to specify a filepath to their input image and tune relevant parameters such as wall and floor thickness to generate a graybox level.

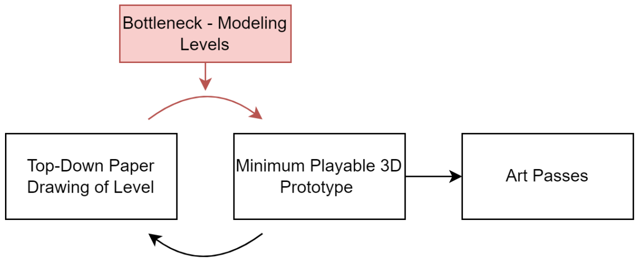
Bottleneck - Modeling Levels
Top-Down Paper Drawing of Level
Minimum Playable 3D Prototype
Art Passes

Figure 2. Iterative approach to designing successful 3D levels.

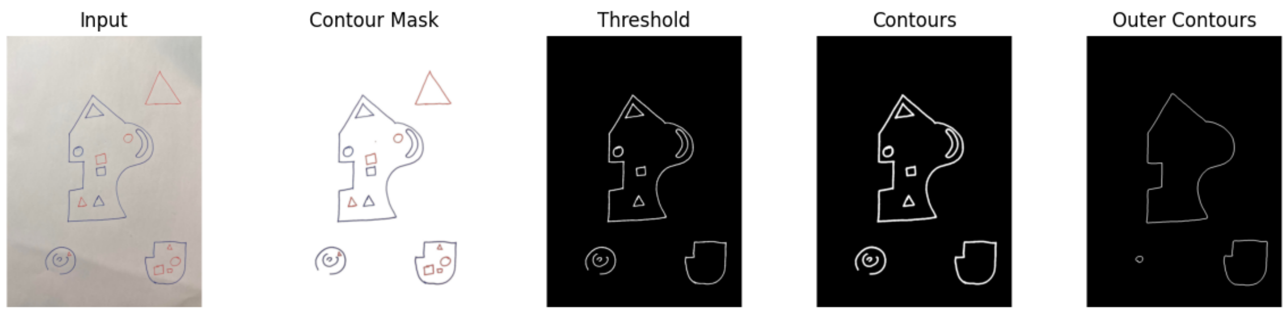

Figure 3. Example of HSV thresholding and level geometry contour extraction.

2. Related Work:

Procedural Content Generation (PCG) is the automated creation of game content. Existing methods have been applied to many types of game content such as levels, audio, character models, and 2D textures [14]. Traditionally, this has been achieved through algorithms based on grammars, search, constraints, and solvers.

Summerville et al.[15] have defined the term PCGML to categorize PCG that utilizes machine learning techniques to further extend what content creation is possible to automate. They define several use cases for PCGML one of which is Mixed-Initiative Design where traditional Autonomous Generation PCGML is focused on fully automating the creation of a piece of content. Mixed-Initiative Design utilizes algorithms to assist the designer in their work by speeding up bottlenecks in the design process, by suggesting new directions during iterative processes, or by evaluating a designer's work.

Some of these approaches interact with a designer by utilizing hand-drawn input. Wang et al.[18] propose a GAN approach for generating terrain heightmaps from hand-drawn contours representing the bounds of bodies of water such as lakes and rivers. This approach can save time and allow for more artist control in the design of games featuring open world environments. Another exciting approach for generating levels from a hand drawn input by Dewantoro et. al.[6] utilizes a CycleGan with the Google Quickdraw Dataset to generate levels for the mobile game Angry Birds. While this is a specific use case, a similar approach can be generalized to many types of games.

There is also much work being done focused on the application of deep learning to the interpretation of sketches [19]. Many of the techniques discussed can be applied to game level design if combined with existing PCG techniques. Level prototyping processes vary greatly depending on the studio, designer, and use-cases. There has been work to understand the current prototyping process [13]. Level designers often start with a top-down sketch [22] which is used as reference to construct a playable prototype called a graybox [21].

Because traditional 3D modeling processes are often too slow and cumbersome for level prototyping, the current remedy is to use simplified "geometry brushes", which are variations on traditional Constructive Solid Geometry (CSG) approaches. Unreal Engine 4[8] uses an approach called BSP Brushes and Unity[16] uses a toolset ProBuilder which are both built using traditional CSG approaches. Realtime CSG allows for basic 3D modeling for level designers through a series of extrudes and boolean operations. More detailed modeling operations require manual work in a standalone modeling package such as Blender[4], or Autodesk 3DSMax[10] and Maya[1], which introduces more complexity for exporting models and importing them into the level designer's game engine. While CSG modeling is simpler and faster than traditional modeling, it can still create a bottleneck when attempting fast iteration of level designs.

Inspiration for the approach detailed in this paper came from previous projects for CS231A that have interpreted 2D information using contour finding approaches [12, 7], which serves as an essential step of this project. [17] developed a simple linear SVM classifier trained on the Google Quickdraw dataset which also served as inspiration for the sketch classification portion of this project. There are many applications and methods for page scanning. [20] provided inspiration for my approach.

3. Approach:

Automatically generating a graybox level from an image requires 1. Extracting relevant data from an input image of a sketch, 2. Generating level geometry and instance points from the data, and 3. Processing level geometry and matching game actors to instance points in a game engine.

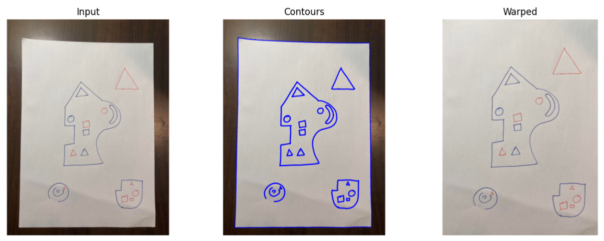

Figure 4. Example of contour-based page scanning algorithm.

3.1. Data from Image

The designer supplies an image of a sketch, a photo of a drawing of a top-down level. The sketch should contain blue contours (drawn using a regular blue pen) representing the walls and bounds on the floors, and red drawings (red pen) representing instancer objects [Fig 3]. These instancer drawings represent some actor or primitive object the designer would like to be placed in the scene. In the figures shown, an instancer drawing can be a circle, square, and triangle which represent a sphere, cube, or pyramid primitive object, respectively. This system can be extended to include more categories and the user can define new correspondences to allow for any placeable gameplay actor in the game engine to be represented by a drawing. For example, a gameplay actor can be a point light, a player character, or an enemy.

Constraints on this input are that the full page must be visible on a background which is a different color from the page and which is preferably a lambertian material so light reflections do not mask the page contour.

3.1.1 Page Scanning

First, an image containing only the contents of the page projected onto the image plane is extracted. This is achieved using the approach described by [20].
First, a gaussian blur kernel is convolved with the image then Canny Edge Detection is run on the output. Canny Edge detection is an algorithm that uses gradient extraction via Sobel Filter convolutions, non-max supression, and hysteresis thresholding to return a binary image from an input image with white pixels representing contour lines. The OpenCV[2] function `cv2.findContours()` is called on the output of Canny edge detection to approximate these contours with a number of vertices connected by edges. The Douglas-Peuckar algorithm is run on the output with $\epsilon = 0.05p$ where p is the perimeter of the contour. This approximates fewer essential vertices from the contours. The closed contour bordering the largest area with a four-point Douglas-Peucker approximation is chosen as the image contour. A four-point transformation is done using the corner points of this contour to "scan" the image by warping the input to the image plane. The image is cropped around the edges to remove any artifacts on the edges due to the page bounds not being straight. Results can be seen in [Fig 4]

3.1.2 HSV Thresholding

To extract level contours, first all contours are extracted from the image using a similar approach to the previous section. These contours are used as a mask to the input scanned image to set all pixel values not on or near a contour to white. Otherwise, thresholding may create artifical contours on undesired parts of the image, such as a shadow.

Next, the blue and red contours are separated. To do so, the image is converted to a Hue-Saturation-Value (HSV) colorspace and use the bounds $[70, 0, 0]$ and $[150, 255, 255]$ to segment blue contours and $[0, 80, 20]$, $[10, 255, 255]$ unioned with $[160, 80, 20]$ and $[179, 255, 255]$ to segment red contours. Thresholding returns two unique bit masks representing blue and red contours. Results can be seen in [Fig 3, 5].

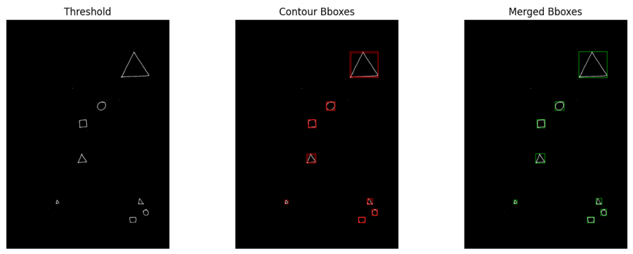

Figure 5. Example of bounding box detection

3.1.3 Level Contours

The output blue contour bitmasks represent the level contours. `cv2.findContours()` is called to find vertices and edges representing the bounds of the blue contour bitmask. `cv2.findContours()` returns hierarchical information from which the outermost contour can be deduced. The outermost contour represents the floor bounds for closed surfaces and all contours are used to represent walls. Results can be seen in [Fig 3].

3.1.4 Instancer Bounding Boxes

Bounding boxes are drawn around each red instancer sketch. To do so, contours representing red pen strokes are extracted using the HSV thresholding method described above in Section 3.1.2. A bounding box is drawn around each separate red contour. Next, bounding boxes are merged together iteratively based on the intersection over union method. This outputs a unique bounding box per separate drawing, which can be used to extract and classify individual sketches. Results can be seen in [Fig 5].

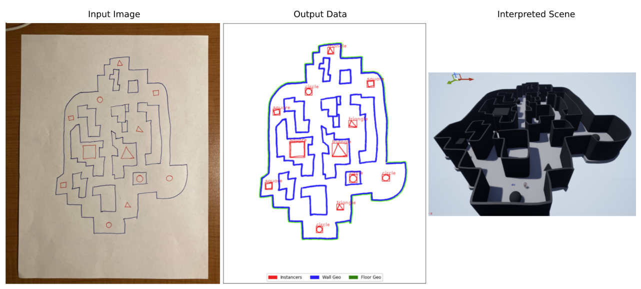

Figure 6. Example of results using level design as an input

3.1.5 SVM Model Training:

In order to assign labels to sketches, an SVM classifier is trained using data from the Google Quickdraw Dataset [9]. The classifier is trained to assign one of three labels to an input bitmask based on learned weights. This model is trained with 2000 images from each category and achieves a 95% accuracy on the test dataset which consists of another 2000 images per category. The resulting confusion matrix can be seen in [Table 4.1].

The SVM classifier model used is non-linear and uses the Radial Basis Function as a kernel, which determines how the datapoints are categorized when training.

$$\phi_{\text{rbf}}(x) = exp(-\gamma \|x - x'\|^2) \tag{1}$$

The rbf kernel equation (1) takes one hyperparameter γ which defines how much influence a datapoint has on the training[5]. The SVM classifier solves the following optimization problem:

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i \tag{2}$$

$$\text{subject to } y_i(w^T \phi_{\text{rbf}}(x_i) + b) \geq 1 - \zeta_i,$$

$$\zeta_i \geq 0, i = 1, ..., n$$

The optimization formulation in (2) maximizes the margin that separates different classes in the data. The formulation takes one hyperparameter C. Since some datasets cannot be perfectly separated, the hyperparameter C allows this formulation to be a soft margin SVM classifier and controls the strength by which samples in the wrong region are penalized[5].

When the training is complete and the optimization problem solved, the trained model classifies test data by using the following formula:

$$\sum_{i \in SV} y_i \alpha_i K(x_i, x) + b \tag{3}$$

The predicted class is the sign of the output from Equation(3). When training the multi-class SVM, the `sklearn`[3] package implements a “one-versus-one” approach[5] under the hood where multiple SVM classifiers are trained, one for each set of two labels. Since the dataset used contains 3 classes, the multi-class SVM classifier internally constructs and trains the following number of classifiers:

$$\text{n\_classes} * (\text{n\_classes} - 1)/2 = 3 * (3 - 1)/2 = 3 \tag{4}$$

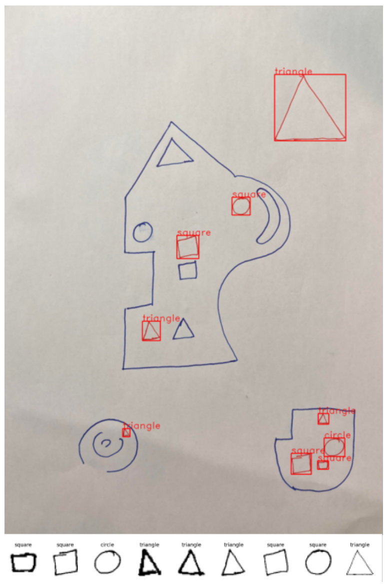

Figure 7. Example of real predictions using the classifier.

3.1.6 Instancer Classification

Once an input image is processed, the bounding boxes are extracted and resized to a 28x28 bitmap using padding to preserve its aspect ratio. Next, pixel intensity thresholding is applied to extract a bitmap representing the sketch. With the bitmap as an input, the trained SVM model 3.1.5 is used to predict and assign an output label to the bitmap. This process is repeated for each segmented sketch. Results can be seen in [Fig 7].

For testing purposes, only three categories were used. However, the Google Quickdraw dataset contains 345 categories so the system can be extended to classify more categories. For example, a sketch of a person could specify the starting location of a player character, a light-bulb could specify the location of a point light, and a cross might specify some objective location or special item. Implementing an extended number of categories and allowing categories to be chosen by the user from within the game engine remain as possible future work.

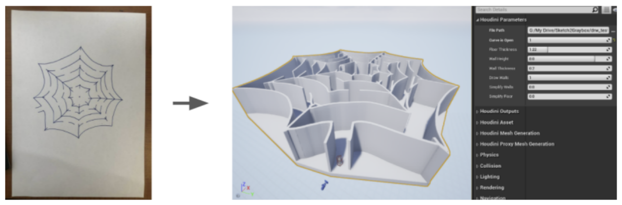

Figure 8. Example of results using a maze level as an input. Example of convex-hull approximation for open shape floor geometery.

3.2. Geometry from Data

Geometry is generated from the extracted data using processes built with SideFX Houdini[11]. The Houdini subnetwork used to generate geometry is then packaged into an importable asset called an HDA, which is loaded into the game engine, Unreal Engine 4[8]. The same approach can be used for any game engine that supports Houdini Engine, such as Unity[16].

Wall and floor polygons are generated from the input level contours. The outermost level contours are used as input to generate floor polygons. If desired floor polygon contours are not closed, the user can instead specify to generate a floor using a Convex Hull approach, which is shown in [Fig 8]. Next, the polygon is extruded based on the users input for thickness to generate floor geometry.

Wall contours are approximated from all the contours. Because contours represent the edges of pen strokes, center-lines are desired to be the spline driving wall locations. The centerlines are extracted in Houdini[11] through a series of fuse, re-sample, and feature sharpen nodes. These centerlines are given thickness by extruding polygons in the direction obtained from $\langle [0\ 1\ 0], t \rangle$ and $-\langle [0\ 1\ 0], t \rangle$ where t is the tangent direction for each contour vertex. The amount of this extrusion is based on a user-defined thickness parameter. Next, these polygons are extruded upwards to form the level walls based on user input for wall height.

Instancer bounding boxes and labels are extracted from the input and used during geometry generation to place instance points and set instance attributes to be interpreted by the game engine. Instance points are placed in the center of each bounding box. Bouding box max sizes are used to set a pscale attribute, which is interpreted in Unreal Engine[8] to set the scale of the actors. Instancer labels are combined with an unreal directory specified by the user to copy objects into the scene that have the same name as the category label. In the results shown, a cube primitive is named "square" to be placed on instance points represented by the square label, spheres with "circle" and pyramids with "triangle".

3.3. Final Level

The Houdini Digital Asset, which calls the data extraction code and generates the geometry, is imported to Unreal Engine 4[8] using the Houdini Engine Plugin[11]. From Unreal Engine, the user can specify the filepath to their input drawing and tune relevant geometry parameters. Unreal Engine's integration with Houdini allows collision geometry to be generated from Houdini's output geometry and actors to be copied to Houdini's output instance points. This allows a final playable level to be generated from within the game engine.

The user can specify input parameters such as wall height, wall thickness, floor thickness, and a toggle for convex hull approximation of the floor in the case of open contours. The user can also specify different actors other than the default cube, sphere, and pyramid provided to spawn with the relevant transformations. They can do so by specifying a directory parameter to an Unreal Engine folder which contains actors with names corresponding to the desired class label name. This provides the flexibility to use new primitive assets. For example, a cone instead of a pyramid can be specified to spawn where triangles are drawn.

This approach could be used with any other gameplay actor such as lighting, playable and nonplayable characters, objectives, etc, so long as the actor can be made a spawnable blueprint in the content browser of the engine.

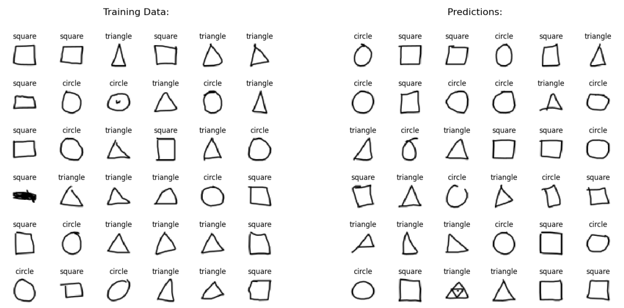
Training Data:
square square triangle square triangle triangle
square circle circle triangle circle triangle
square circle triangle square triangle circle
square triangle triangle triangle circle square
square circle triangle triangle triangle square
circle square circle triangle triangle square
Predictions:
circle square square circle square triangle
circle square circle circle triangle circle
triangle circle triangle square square circle
square triangle circle triangle circle square
triangle triangle triangle circle square circle
circle square triangle triangle square square

Figure 9. Example of training data and predictions from the Google Quickdraw Dataset [9].

4. Experiments:

4.1. Quantitative

For classifying instancer doodles, an multi-class non-linear SVM classifier was trained using data from the

Training	Testing	Accuracy
6000	6000	0.95

$\downarrow$

Label	Precision	Recall	f1-score
Circle	0.96	0.93	0.95
Square	0.94	0.98	0.96
Triangle	0.95	0.93	0.94

Training	Testing	Accuracy
15000	15000	0.96

$\downarrow$

Label	Precision	Recall	f1-score
Circle	0.96	0.95	0.95
Square	0.96	0.98	0.97
Triangle	0.95	0.94	0.95

Table 1. Confusion Matrices for models with different numbers of datapoints.

Google Quickdraw Dataset[9] and the SciKit Learn[3] Library. This dataset contains 50 Million doodles over 345 categories extracted from the web-based drawing game Quickdraw [9]. For this paper, only three categories were used, but the large number of possible categories provides room for expansion on this idea. Examples of training data can be seen in [Fig 9]. The SVM was trained with only 6,000 samples spread over the three categories. When tested on another 6,000 samples, it acheived 95% accuracy. Example predictions are shown in [Fig 7]. Increasing the training and testing samples to a total of 15,000 each increased accuracy to only 96%. The confusion matrices are shown in [Table 4.1]. A softmax layer was added so that the probability for each category could be read.

4.2. Qualitative

4.2.1 Quality Evaluation: Classification

There are classifying inaccuracies higher than 95 percent when used on real-world hand drawn data. These inaccuracies mostly occur on larger drawings. This is due to a difference between the training data and the data extracted from the drawings. Bounding boxes are extracted from the image and the image data is resized to have a width of 28 pixels to match the training data. The result of this is that larger objects have a thinner contour width than the training data with constant contour widths.

There are two methods that can be employed to resolve this. One method is to use contour detection from the extracted and resized image inputs. These contours can be redrawn on a blank image at constant widths matching the training data.

Another possible method is to extend the training data using morphology functions such as erode and dilate to account for varying line widths. This method has the cost of addition data pre-processing with the benefit of additional robustness against drawing tools of varying line widths.

Other challenges arose when working with the Google Quickdraw Dataset. The dataset appears unfiltered from the data collected using the Google Quickdraw web game. As a result, there are a number of inaccurate and obscene results contained in the dataset. This may pose a problem when increasing category counts and may be a problem for projects that attempt to synthesize new sketches.

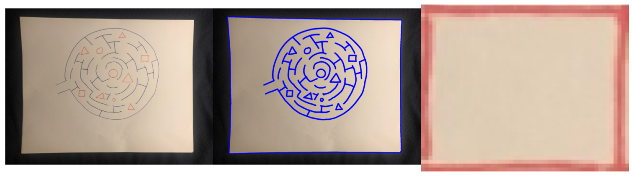

Figure 10. Qualitative Error: Page Scanning on warped page edges.

4.2.2 Quality Evaluation: Page Scanning

As seen in [Fig 10], page scanning is prone to failure in cases where the edges of the page are overly concave or convex in the input image. This is because the Douglas-Peuckar algorithm will approximate the page bounding contours with more than four points. As a result, another contour or no contour will be chosen instead. In [Fig 10], a drawing of a square is the largest closed four point approximated contour in the image, so it is chosen as the page bounds.

Other page scanning problems occur when the whole page is not contained within the image, or the background contains bright white reflections that obscure a page boundary.

A solution to these problems is to use a more robust page-scanning algorithm that detects corner points and parallel lines. In such cases only two points would need to be visible to do a perspective projection of the page contents to the image plane.

4.2.3 Quality Evaluation: HSV Thresholding

HSV Thresholding can produce different results under different lighting conditions, as shown in [Fig 11]. This could be resolved by using automatic white balancing or color balancing algorithms to fix the color space of the input image. We can find a transform that takes the page background off-white color to a pure white value. We can apply the transformation to all pixels to get a standardized color set.

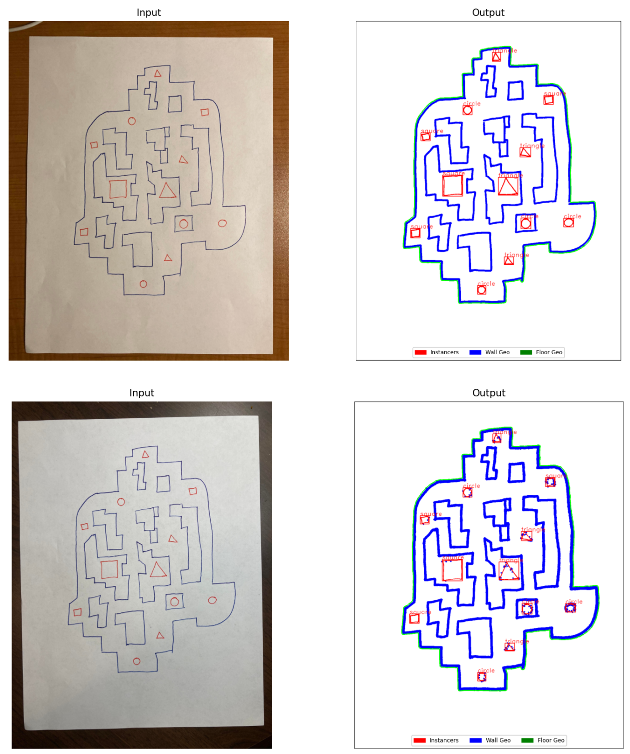

Figure 11. Qualitative Error: Thresholding under varying lighting conditions

4.2.4 Quality Evaluation: Level Contours

Another significant challenge was interpreting level contours and creating geometry. Contours are extracted from both sides of a pen stroke when using the described Canny edge dection and `cv2.findContours()` approach. As a result, wall contours had to be merged using Houdini Engine[11] through a series of point merges. This caused features to become smooth, which was counteracted with geometry sharpening. This approach is not ideal as it does not always provide results that match hand drawn input. A better solution would be an approach that either estimates centroid contours from the pen stroke outline contours, or an approach to extract a contour that estimates the stroke by using a thresholding technique as apposed to canny edge detection.

5. Conclusion:

This project highlights the challenges and possibilities for generating procedural level prototype geometry from 2D input images.

Future work involves investigating how to approximate depth changes in the ground plane that actors are placed on. Current results do not produce variations in the ground plane, which a level designer may desire for their prototype. This is a difficult problem to solve due to the inherent difficulties of interpreting 3-dimensionality from a 2D sketch. Ad-hoc solutions could be to require different colored pens for different floors as input. Another ad-hoc solution could be using stair symbols to specify changes in elevation.

There are many more difficulties that arise from attempts to interpret 3-dimensionality from more complex images - such as underpasses and multi-story overlapping levels. One potential direction is to utilize more automation in the generation of 3D levels from 2D contours. For example, Wave Function Collapse is a popular algorithm that can be used to generate 3D levels from a grid. A grid could be fit to user-defined contours, and a constraint-based algorithm such as WFC could generate multiple tile-based 3D levels with a combination of designer input and selection.

While the current approach requires little input from the designer except for a file path to generate levels, it requires some set up such as installing the Houdini Engine plugin[11]. Future work could include implementing this approach directly in the game engine as a plugin or stand-alone build to allow use with minimal setup. Additionally, a stand-alone build could allow levels to be generating on mobile devices such an a smart phone or tablet, which could allow for faster seamless iteration in the design process as photos can be taken directly from the device from within the build environment, and time would not be lost saving, sending, and selecting files.

More work could be done interacting with game designers using a human-computer-interaction design framework to make the tools more seamless and useful to level designers in ways not previously thought of.

6. Code:

The code implementation for Sketch 2 Graybox can be found at this address: `https://github.com/ColeSohn/Sketch-2-Graybox`

References

[1] Autodesk, INC. *Maya*. Version 2019. Jan. 15, 2019. URL: `https:/%20autodesk.com/maya`.

[2] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000).

[3] Lars Buitinck et al. “API design for machine learning software: experiences from the scikit-learn project”. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[4] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018. URL: http://www.blender.org.

[5] scikit-learn developers. *Support Vector Machines*. Scikit Learn. URL: https://scikit-learn.org/stable/modules/svm.html.

[6] Mury F. Dewantoro et al. "Enhancement of Angry Birds Level Generation from Sketches Using Cycle-Consistent Adversarial Networks". In: *2020 IEEE 9th Global Conference on Consumer Electronics (GCCE)*. 2020, pp. 564–565. DOI: 10.1109/GCCE50665.2020.9291893.

[7] Bradley Emi. "CS231A Final Project: Optical Recognition of Hand-Drawn Chemical Structures". In: 2020.

[8] Epic Games. *Unreal Engine*. Version 4.22.1. Apr. 25, 2019. URL: https://www.unrealengine.com.

[9] Googlecreativelab. *Googlecreativelab/quickdraw-dataset: Documentation on how to access and use the quick, draw! dataset.* URL: https://github.com/googlecreativelab/quickdraw-dataset.

[10] Jeffrey Harper. *Mastering Autodesk 3ds Max 2013*. John Wiley & Sons, 2012.

[11] Houdini. *SideFX*. URL: https://www.sidefx.com/.

[12] Aditi Jain. "CS231A Final Project Report: Set AI". In: 2021.

[13] Annakaisa Kultima, Juha Köönikkä, and Juho Karvinen. "The Four Different Innovation Philosophies Guiding the Game Development Processes : An Experimental Study on Finnish Game Professionals Development Processes". English. In: *Games and Innovation Research Seminar 2011 Working Papers*. Ed. by Annakaisa Kultima and Mirva Peltoniemi. TRIM Research Reports 7. 2012, pp. 34–41. ISBN: 978-951-44-8705-7.

[14] Jialin Liu et al. "Deep learning for procedural content generation". In: *Neural Computing and Applications* 33.1 (Oct. 2020), pp. 19–37. ISSN: 1433-3058. DOI: 10.1007/s00521-020-05383-8. URL: http://dx.doi.org/10.1007/s00521-020-05383-8.

[15] Adam Summerville et al. "Procedural Content Generation via Machine Learning (PCGML)". In: *IEEE Transactions on Games* 10.3 (2018), pp. 257–270. DOI: 10.1109/TG.2018.2846639.

[16] Unity Technologies. *Unity*. Version 2019.4 LTS. URL: https://unity.com/.

[17] *Using google's quickdraw to create an mnist style dataset!* URL: http://projects.rajivshah.com/blog/2017/07/14/QuickDraw/.

[18] Tong Wang and Shuichi Kurabayashi. "Sketch2Map: A Game Map Design Support System Allowing Quick Hand Sketch Prototyping". In: *2020 IEEE Conference on Games (CoG)*. 2020, pp. 596–599. DOI: 10.1109/CoG47356.2020.9231754.

[19] Peng Xu. "Deep Learning for Free-Hand Sketch: A Survey". In: *CoRR* abs/2001.02600 (2020). arXiv: 2001.02600. URL: http://arxiv.org/abs/2001.02600.

[20] Rouizi Yacine. *Learn opencv by building a document scanner*. URL: https://dontrepeatyourself.org/post/learn-opencv-by-building-a-document-scanner/.

[21] Robert Yang. *Blockout - How to build a basic 3D version of the level with massing, metrics, composition, and iteration*. URL: https://book.leveldesignbook.com/process/blockout.

[22] Robert Yang. *Layout - How to draw a top-down floor plan for a level, with flow, balance, encounters, and typology*. URL: https://book.leveldesignbook.com/process/layout.