CREATING A PLAYER AID FOR GEOMETRY DASH APS360 PROGRESS REPORT

Jaden Dai
Student# 1009972228
jaden.dai@mail.utoronto.ca

Joel Vadakken
Student# 10010089798
j.vadakken@mail.utoronto.ca

Skyler Han
Student# 1009794830
hs.han@mail.utoronto.ca

Ian Lu
Student# 1009972139
i.lu@mail.utoronto.ca

ABSTRACT

Our team intends to create a program that can create semantic segmentation maps of a Geometry Dash Level using machine learning. This program can be used to assist players in recognizing obstacles without the usage of mods. We are using transfer learning to achieve this task. —-Total Pages: 4

1 BRIEF PROJECT DESCRIPTION

The goal of our project is to develop a model that takes a screenshot from the game "Geometry Dash" and detects the location of collision boxes automatically. For example, given a live screenshot of the game, it should be able to recognize objects and determine their appropriate collision box:

Above is an example of our desired output for a given screenshot input. Note that Geometry Dash collision boxes are somewhat strangely-shaped, especially for spike objects.

We believe this project is appropriate for deep learning due to the many complexities and edge cases that may appear. For instance, there are many decorative objects that a player can recognize as irrelevant, but may trick a traditional hard-coded or image matching algorithm.

Here, a player can tell the squares are part of the background, while a hardcoded model may mistake it for a foreground object.

We believe this model could be a useful tool for players, who may not have access to other paid tools to view collision boxes. Furthermore, this may be a useful model for future work in training a RL model to play Geometry Dash by simplifying inputted information.

2 INDIVIDUAL CONTRIBUTIONS AND RESPONSIBILITIES

Our team is working well together. We are using github to share code and the latex document, and google docs to share documents for rough work, brainstorming, rough drafts, and data collection. Please see Table 1 for the tasks we have completed so far. The remaining work we have includes: collecting more data, creating a test set, trying out different architectures, and hyperparameter tuning. Please see Table **??** for how we have decided to divide up these tasks and their deadlines. Since hyperparameter tuning, trying out different architectures, and creating a testset.

Table 1: Individual Contributions and Tasks

NAME	CONTRIBUTIONS	TASKS
Jaden Dai	Did most of the data collection, including creating macros for Geometry Dash levels, creating texture packs, and creating macros for taking screenshots.	Testset creation. Hyperparameter tuning.
Joel Vadakken	Helped with the data collection. Wrote the code to preprocess the data before training the model.	More Data collection. Experiment with different models.
Skyler Han	Created the Baseline model for comparison. Wrote the training code for the model.	Improve baseline model. Experiment with different model architectures
Ian Lu	Created the architecture for the model, and did some hyperparameter tuning.	Hyperparameter tuning. Testset creation.

Table 2: Task Deadlines

TASK	DEADLINE	JUSTIFICATION
Collecting more data	July 13, 2024	We need most of our data ready to train our models, so it important we have all of our data ready by then.
Creating a test set	July 20, 2024	We won't need to evaluate our model until closer to the submission deadline, so this is not as high priority a task
Trying out different architectures	July 20, 2024	We should be done most of our model training so that we can work on our final submission
Hyperparameter tuning	July 20, 2024	Hyperparameter tuning and trying out different architectures should occur simultaneously, as they are linked together.

3 Data Processing

We have collected a set of preliminary data directly from the game. The data consists of a live game screenshot with a paired screenshot with labels. See Fig. 1

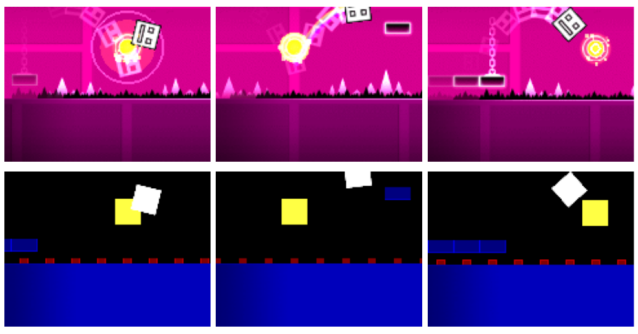

Figure 1: 3 sample paired screenshots. Top: live screenshot from game. Bottom: labeled screenshots.

These screenshots were taken from version 2.1 using the MegaHack V7.1 mod, which features several useful tools (ie. a bot/"macro" to play through levels, a filter for non-gameplay elements). Our exact method is as follows. The exact method is as follows:

- Set the game to 1920x1440 resolution with low textures.
- Record a "macro" using the built-in MegaHack tool.
- Turn on "frame stepper" in MegaHack.
- Use a Python script to step through the macro frame-by-frame and take screenshots.
  - The bot takes a screenshot every 50 frames, but this can be adjusted.
- Load our self-created texture pack by importing files into the game's directory.
- Turn off all visual effects using MegaHack.
  - Orb rings, particles, gravity effects, etc.
- Enable "Show Layout," set background to black and ground color to white.
- Enable "Show Hitboxes", set opacity to 100, enable fill, disable player and special hitboxes.
- Use the Python script again to take screenshots, which produces the labeled screenshots.

Then, from these preliminary labeled screenshots, we further process the data into what can be seen in Fig 2

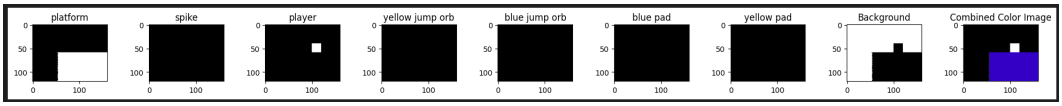

Figure 2: Post processed data featuring label masks for each class of object.

Since each class already has its own color, we can then transform this into our dataset by reading in the pixel RGB values and mapping that to a multi-channel mask. Each respective channel represents a specific type of object, which can be seen above. This output data is then used to train our semantic segmentation model.

There were some challenges that we encountered during this process: The newest version of Geometry Dash (2.2) does not have some tools that are necessary to generate labeled screenshots. For instance, it lacks the ability to "step through frames" which is essential for creating paired screenshots. We attempted some other methods of pausing the game and taking screenshots with some post processing methods, but these methods generated messier and worse data. Since we used a texture pack, the collision box labels for "orbs" were inaccurate, as the collision box stretches beyond the size of the textures.

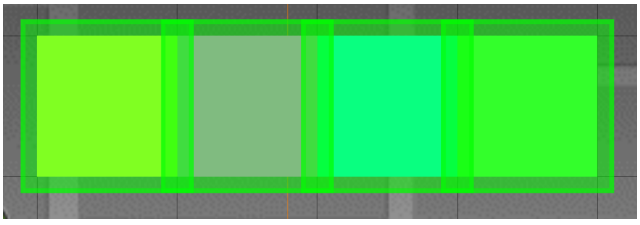

Figure 3: Green: Collision box: Inner square: Largest possible label for orb using texture pack

This means our final result will be slightly inaccurate.

Most of these challenges could be remedied through the use of a custom-built mod for the game. However, given that this would require learning to inject C++ code, we decided that this was out of the course's scope, and we believe that these limitations do not compromise our overall goal.

4 Baseline Model

Our baseline model is a template matching model. It uses the OpenCV library to match a template image (an object) to a larger image (the game screen). The model uses the "matchTemplate" function

to find the location of the template image in the larger image. If the match is above a certain confidence threshold, the model draws a rectangle around the location.

Our model does this for template images, including such as spikes, portals, players, and small spikes. The model saves the resulting image with the rectangles drawn around the objects, then takes the difference between the original image and the image with rectangles, then saves the difference image. The model provides a qualitative result by showing the original map image, the resulting image with the rectangles drawn around the objects, and the difference image.

The model also provides a quantitative accuracy, which is calculated by the correct number of classifications divided by the total number of objects in the scene. After tuning confidence thresholds and using different feature matching functions provided by OpenCV, the model achieves an accuracy of 0.83 on the selected samples. However, the accuracy is significantly worse once the image has a different resolution or the objects are placed in a different orientation. Moreover, compared with the ground truth data, the baseline model sometimes misclassifies small objects such as small spikes, which leads to a lower accuracy compared to the actual model.

The model faces challenges in accurately detecting objects in the larger image due to variations in lighting, color, scale, rotation, and occlusion in the game. For example, the model would not be able to detect a spike if it's placed upside down. The model also faces challenges in accurately matching the template image to the larger image due to noise and artifacts in the images. It can be improved by using more sophisticated computer vision techniques, such as feature matching, object detection, and image segmentation with neural networks, which is what we did in the actual model.

5 Primary Model

The function of the primary model is to perform semantic segmentation on Geometry Dash maps, distinguishing and mapping out different terrain objects and returning their collision boxes. Semantic segmentation involves extensive pre-processing of the data to create labels for each unique class as the target masks to then output an image with the same dimensions as the original image where each pixel of the output is assigned a class from an input image. Specifically, the dimension of the output tensor will be in the form of [batch_size, n_classes, height, width] such that the final output will be the composition of each mask, determined by the number of classes.

In our case, our neural network will be fed an input image, a processed target image segmented into layers of masks corresponding to the number of classes associated with an object terrain on a geometry dash, then the output of the network will display the input image with each pixel corresponding to the class of a terrain object in geometry dash.

To perform semantic segmentation the primary model will be based upon a large convolutional neural network with an adapted UNet architecture for our input images Ronneberger et al. (2015). The UNet architecture is a fully convolutional neural network model with an encoder-decoder structure and skip connections between the coders. It is the standard architecture for image or semantic segmentation The encoder part of the network extracts features of the image increasing in level as the number of feature channels increases down the encoder. The decoder part of the network then performs upsampling and convolutions rather than transposed convolutions along with skip connections from the encoder to pass on spatial information during the restoration of the original image to ultimately arrive at the input dimensions.

References

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015. URL https://arxiv.org/abs/1505.04597.

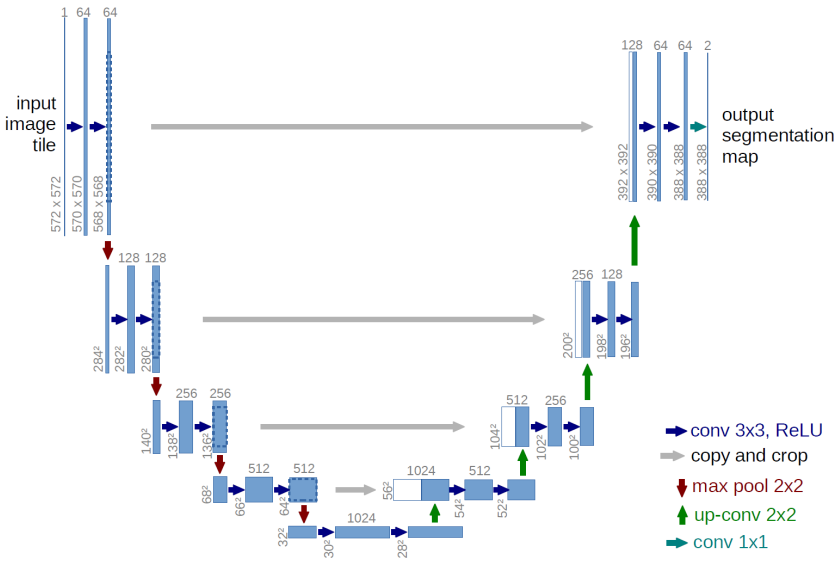

Figure 4: UNet architecture diagram

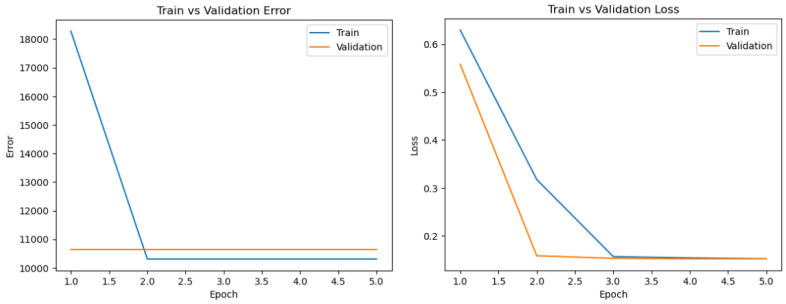

Figure 5: Training and Validation error and losses