Art & Machine Learning, 2018 Spring CMU

Final Project Proposal

Group: Oscar Dadfar & Hizal Celik

1 Abstract

In this research, we analyze the possibility of procedurally generating a canvas reflective of the Grand Theft Auto style. Using a semantic segmented GTA V dataset, we train the pytorch implementation of Pix2Pix, as well as Pix2PixHD [3]. We merge a procedurally-generated Unity environment with the Pix2Pix style transfer algorithm to get the generative GTA environment, and feed the data back into Unity to produce a real-time style transfer of the procedurally generated environment.

This research builds upon the Assignment 3 submission performed by Lingdong Huang, Hizal Celik & Shouvik Mani in which they performed a style transfer on a procedurally-generated Unity environment and compile the frames into a video. We build upon this research by changing the training data to a GTA V dataset, and attempt to make the style-transfer realtime by exporting frames from Unity and importing them back in to be displayed on the main camera after the style transfer has been performed.

The research is broken into 3 components: procedurally generated environments in Unity, training the Pix2Pix & Pix2PixHD algorithms on the GTA V dataset, and bridging together Unity and Pix2Pix to generate style-transferred frames on the screen in realtime.

2 Procedurally-Generated Cities

2.1 Introduction

Having a procedurally generated GTA environment requires a set of images to style transfer on. By using a generated Unity environment and exporting an image sequence, we can achieve the desired style transfer effect while still maintaining the interactivity of a Unity game engine.

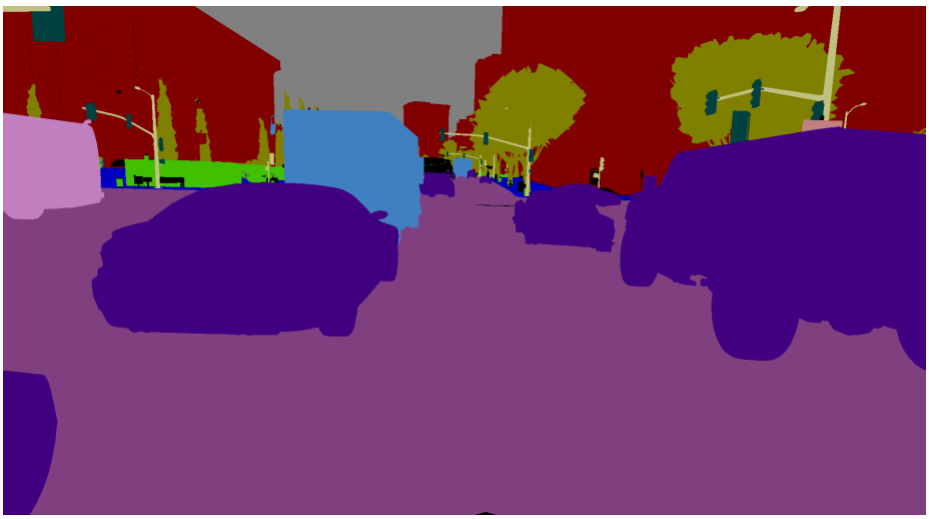
2.2 Semantic City Baseline

We use Lingdong's Semantic City Unity scripts as a base for the procedurally-generated cities [1]. Lingdon's Semantic City starts by treating the entire surface as one entire block, and proceeds to divide the block into smaller blocks, giving way to roads in between adjacent blocks. Within each block, rectangles are placed with random transformation properties (i.e. rotation, scale, position) to represent buildings. Alongside the roads, elements such as street signs, traffic lights, and trees populate the areas adjacent to the street. Vehicles are generated onto the road with a simple AI that attempts to get each vehicle to a specific destination by only moving forward or backwards and turning on roads.

For a more comprehensive review of how this procedurally-generated baseline algorithm works, please refer to the appendix.

2.2 Color Mapping

In order to undergo successful style transferring, the color mapping of the Unity environment must match the color keys of the GTA V dataset that Pix2Pix was trained on. This requires that all objects of a certain class (i.e vegetation, vehicles) must have the same color schemes. In Unity, this requires them to have the same texture of a specific color. The baseline Semantic City shares the same color mapping as the GTA V semantics (with a few alterations here and there) [3].



Example of semantic segmentation from the GTA dataset.

2.3 Optimizations on Runtime

The Semantic City Baseline initially ran at a framerate range of 9-10. All of the objects are generated independently, making for excess use of memory considering that most objects share similar geometries.

To counter this, we introduce the idea of calling the generative functions for each object a set number of times (say 3) and save the models to a global list that we can then call to create instances of the object on the map. By making copies of the object, we would be reducing memory requirements needed when having to generate different meshes for every single tree when we could choose from an assortment of 3 different trees each time. We repeat this process for all the generated objects.

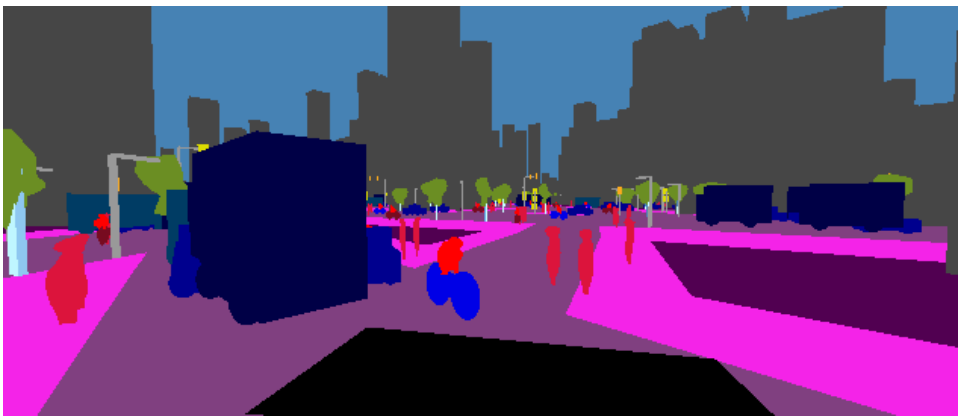
Following this process, we see a staggering increase in framerate by an additional 0.5 frames each second.

2.4 Character Overlays

In order to increase interactivity with the city, we added the feature of being able to control objects (i.e trees, cars, buses). We saved a few of the generated models as prefabs for each class and attach them as global prefabs for our Unity script. Upon pressing the keys:

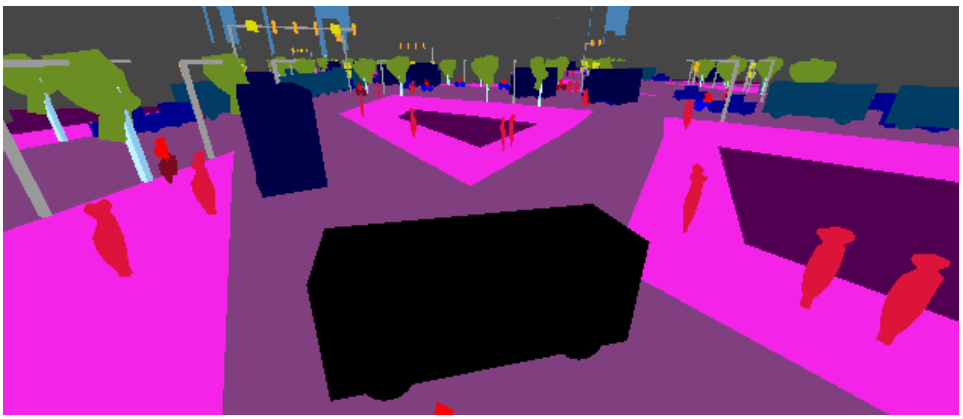
t - tree | c- car | b - bus

The prefab for the corresponding object is generated on the canvas and is attached to the first person camera so that it remains on screen. Toggling between keys will destroy the current prefab and create a new instance of the prefab selected.



Instantiating a car controller.

For buses, since the bus is high enough to block off the camera, we transform the camera's position a few units in the y direction in order to still get a clear view of the city and still feel in control of the bus.



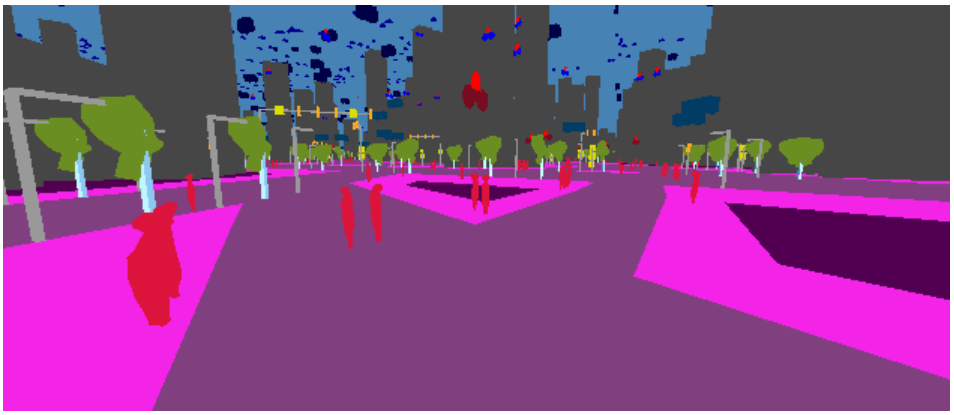
Bus controller with higher altitude camera.

The GTA training dataset maps the color black to any vehicle that is being controlled by the player. We change the skin colors of the car and bus prefab to black to account for this.

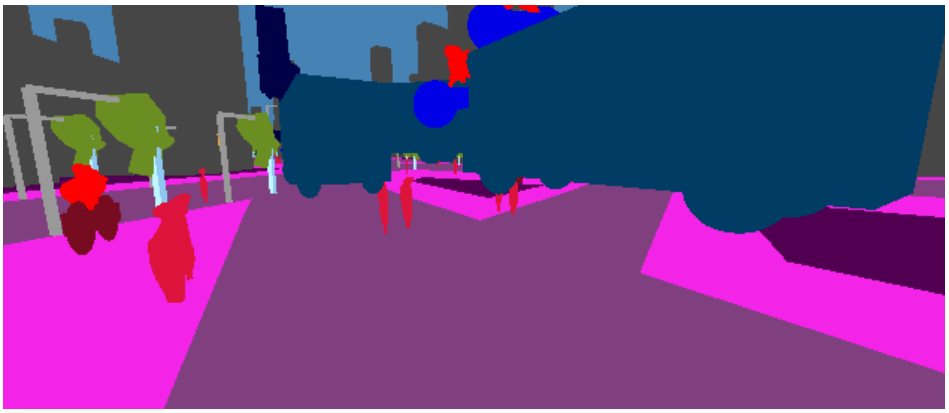
2.5 Flying Cities

2.5.1 Flying Traffic

To move past the standard GTA environment, we also added the ability to have cars and traffic fly. Originally demoed in Lingdong, Hizal, and Shovic's city, we experimented with the flying traffic in different ways. Our initial trial was to modify the generate function for each object to transform the position of each object 200 points off the ground. What we were left with was a comic scenario of each car falling from the sky but then easing towards the ground as if a plane was landing.



2 seconds after game instantiation.



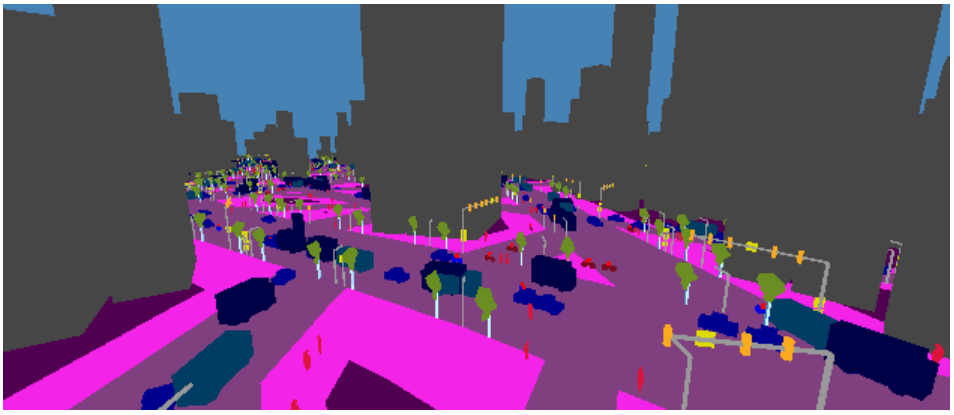
4 seconds after game instantiation.

This was an unusual effect, especially considering the vehicles would ease into the ground rather than accelerate further. Our initial instinct was to disable gravity on all objects, but that had not effect. After consulting Lingdong, we found that there was a global variable in the scripting files that controlled the absolute altitude that traffic would be generated at, and that it was using linear interpolation to bring the vehicles from their original spawning point to this

altitude, thus explaining why the vehicles would slow their descent when reaching the ground. Changing the altitude to a non-zero value allowed us to have floating traffic.

2.5.2 Flying Camera

For further exploration, we also wanted the main camera to fly around the city to capture more exotic shots. We added an additional axis of movement and mapped the ‘k’ and ‘i’ keys to it to allow the camera to ascend and descend. This allowed for more cinematic shots of the entire city that could function as cutscenes.



Flying camera overview of city.

We remapped the camera altitude to the scroll wheel for easier motion.

3 Pix2Pix Training

3.1 Overview

In total we had 4 approaches to training our models. In one pix2pixHD model, we trained on all 25,000 images available in the GTA dataset. In another pix2pixHD model, we trained on only 2,500 images, taking every 10th image in the dataset for a total of 10% the original dataset. In the third pix2pixHD model, we trained on just sunset/nighttime images, which we gathered based on an algorithm that detects perceived brightness in an image, and only kept those below 60/255. Unfortunately only 600 images matched this dataset. And finally, we trained a pix2pix-pytorch model on all 25,000 images. This model had our converted dataset, creating a AtoB dataset with max 1024px wide images.

3.2 Time

The pix2pix model, and the two pix2pixHD models with smaller datasets, all took maximum 5 days to train using a varying number of *6GB*

NVIDIA GeForce GTX 980 Ti's (max 4 for each model). These all were trained to the target of epoch 200. The larger pix2pixHD model, at 25,000 images, took over 4 weeks to train up to epoch 140 before we stopped it (after the exhibition).

Part of the reason it took so long was the occasional crashing of the GPU clusters due to other training sessions, which kicked us off and interrupted our training. Another time consuming issue was when we tried to change the batch size, which we'll get in to in section 3.3.

3.3 Issues

A big issue we had was finding the right combination of parameters to create the most optimal training environment. Parameters included the choice of GPUs (4x *6GB NVIDIA GeForce GTX 980 Ti* and 4x *12GB NVIDIA K40c*), batch size, and number of GPUs to use simultaneously.

In the end, the best combination was using 3x GTX 980s at a batch size of 3. We found that using 4 GPUs, of any kind, resulted in the hanging of the training and ultimately crashing the GPUs causing a cluster restart. We also found that increasing batch size to anything higher than the number of GPUs we were using caused the training to be stuck on the current epoch, gradually increasing the iterations to absurd levels. We lost many days while the training was stuck on epoch 20, starting at iter = 2,300 and ending when we stopped it at iter count over 155,000 (the average iter count was 25,000, equal to the dataset length). I had to manually reset the log files to start it again with better training.

4 Real-Time Style Transfer

4.1 Overview

In order to provide a real-time style transfer capability in Unity, we needed to send the image sequences from Unity to another CPU to handle the image processing, and send the images back to Unity to overlay on the screen. Doing this required two separate machines: one to handle the Unity environment, and one to upload the images to Pix2Pix. The computers communicate to each other using zmq to send image data over to each other.

4.2 Unity Side

The Unity city was traditionally rigged to export an image sequence to a local directory that then would manually have to be loaded in batch into Pix2Pix for style transfer, then download the frames and convert them to a video. Rather than locally saving them, we included a script to the Character Camera that would send the frames over zmq to a local server on another machine given their

host address [2]. We created a new camera and render texture that would receive the style-transferred images from the machine and overlay them on top of the semantic segmented city.

Because the training data were of size 512x256, we had to lock the Unity canvas resolution to the same size.

4.3 Pix2Pix Side

Thanks to help from Aman, we were able to use ZMQ to create a server on one computer for the other computer, running Unity, to connect to. We created a modified test file from the pix2pix-pytorch repo that ran continuously, looking to receive byte data from Unity (the screenshot information), convert it using the loaded model and send it back. We had issues regarding converting the RGB values, which we believe resulted in lost data and muddy images when rendered in Unity. See 4.4. The framerate was also very low, at around 1-2 fps, due to the fact that we were using WiFi to send images back and forth, and the Mac could only use CPU to render images with pix2pix-pytorch.

We also never were able to begin the process of real-time style transfer using our pix2pixHD models. Although it shares a similar testing code in Python to pix2pix-pytorch, we had the disability of needing to connect to a remote cluster server that requires a password. We didn't have time to create a porthole for streaming images.

4.4 Review

4.4.1 Real-Time Style Transfer

We were surprised to find that the real-time style transfer did not look as good as anticipated. The frame rate reached a max of 3 frames each second, and the colors looked very muddy. This could be a result of the CPU we are using to style transfer the images being slow in transferring the images over. The delay in frame retention hurts the interactivity of the piece.

The muddy colors could be a result of incorrect or skewed color mapping when sending the images over between computers. Despite this drawback, the colors make the frames seem more surreal and paint-like. It gives a greater artistic quality to the images that the previous style transfers could not.

4.4.2 Non-Real-Time Style Transfer

The style transfer worked well on the pre-rendered frames, especially on the 4K monitor. Because the Unity frames were mostly cube-like geometries (for rendering optimization), the output was a mix of "Picaso's Cubism" merged with (what we commonly referred to as) GTA-ism. The style transfer maintained the

blocky geometries of the unity environment. Combining this with the effects of flying cars added an additional surrealist style.

Rendering the videos in post-process style gave us the ability to render the same scenes using different models or resolutions across both pix2pix and pix2pixHD, allowing us to compare the visual differences between them. We rendered pix2pixHD footage at 1280, 1500, and 1920 pixel width resolutions (a 6GB GPU was required for higher than 1500 pixels), and we also rendered footage with pix2pix-pytorch for comparison. The results showed that pix2pix-pytorch looked the worst, visually, and that the textures improved when pix2pixHD rendered at higher resolutions, but that patterns also repeated more often when rendering at higher resolutions. We briefly considered rendering at 4k but realized it may not be worth the effort when the dataset itself was only 2k resolution.

The pix2pixHD model trained on only 2,500 images looked worse than the full dataset model (this disproved our suspicion that 25,000 images was "overkill"). The model trained on sunset data looked beautiful but was not trained on a high enough dataset. In the future we may train again, by inverting and cropping the dataset to increase its size.

A curious effect had also occured when we tried style-transfer using only epoch 1 of a pix2pixHD model, where simply turning the camera caused it to change between day and night. We hoped this effect would last into later training, but unfortunately it did not.

4.5 Potential Impact

We discussed the possibility of this Pix2Pix style transfer in the video game industry. Rockstar (creators of GTA) employed thousands of artists to texture every alley, every tree, every grass field. Yet what we did was create an AI that could automatically texture a generative environment so long as it has a correct semantic segmentation. When we are able to achieve faster GPUs in computers and gaming consoles that can handle real-time style transfer capabilities, then companies could start training networks to do the automatic texturing rather than having artists do it themselves. Doing this also provides ease in swapping AIs for other video games in order to mod them, and having more dynamically created environments in video games that can be automatically textured to look as if artists had textured them manually.

5 Results

5.1 Exhibition Video

https://youtu.be/eP5hHKne_gE

Side-by-side comparison of Pix2Pix results on different trainings

References

[1] L. Huang, H. Celik, S. Mani, "Photographic Video Rendering of Procedurally Generated Cities with Pix2pix," March 2018.

[2] A. Tiwari, T. Bai, G. Sun, "Realtime Interactive Scene Style Transfer," March 2018.

[3] S. Richter, V. Vineet, et. al., "Playing for Data: Ground Truth from Computer Games," Springer International Publishing, 2016.