112 TP1 Design Proposal

Ethan Lu, AndrewID: ethanlu

December 5, 2018

1 TP3 Update:

Since TP3, I've added the following:

1. Made the style calculations non-blocking through the use of threading. In particular, the progress bar will now update while doing calculations, and the UI won't show a "not responding" prompt during the calculations.

2. Added paramter tuning to the style importation wizard. In particular, the image process resolution and iteration count can be adjusted at runtime.

3. Implemented the benchmarking interface (which is contained in bench.py). By dynamically adjusting step size/number of steps for comparison in the benchmarking, users can generate a dynamic graph that relates runtime of the calculations on a single image to the number of iterations it's processed for. This was done through the library pyqtplot.

2 TP2 Update

So far, I've accomplished the following:

1. Added all functionality to the QT interface.

2. Added a fully working implementation of the Style Importation wizard.

3. Added working implementations of the Conv_NN algorithm, which can also be computed through the QT interface

4. Added fully functional game launching with modified assets through the interface.

5. Added options in the interface to specify the location of the game file and the dolphin installation path.

6. Added status updating as necessary for the above two options.

The primary difficulty that I've run into has been implementation of the Cycle GAN algorithm. Separately from implementing it in this project, running and training the models in the various demo projects I've seen has required absurd amounts of computing time, requiring 12+ hours to train the models and 30+ minutes to run it on a single image. Furthermore, the results have been particularly bad, so I've decided to drop this algorithm. Instead of implementing it, I'll be adding in options to change the image processing resolution and the iteration count for the Conv_NN algorithm.

Additionally, when running the algorithm, the progress bar doesn't seem to update, even though it does internally. I'm fairly sure this is a limitation of QT, but I'll see if I can get aroung it.

3 Project Overview

3.1 Project Description

Overall, the goal of this project is to provide a useful and fluid framework for performing style transfer in real time through modification of game assets and textures. Most notably, this project is designed to have novel applications for both game development and fan mods, which motivate much of the project's design.
That's quite a wordy introduction though, so let's break down the various terms and components of the project:

3.1.1 Style Transfer:

A cool subset of Machine Learning/Deep Learning that seeks to quantify the "style" or an image or painting, then apply it to other images. For this project, I'm interested in realizing these algorithms in real time.
Examples:

3.1.2 Issues with generalized style transfer algorithms for real time video

Very computationally expensive. For arbitrary video streams, based on the algorithm used, single frames can take anywhere from seconds to minutes to process, even on (very) good hardware. Also, these algorithms don't preserve temporal coherence, frequently resulting in flickering between frames.

My project aims to implement real-time style transfer on a very narrow subset of "arbitrary video streams:" video games.

Eseentially, I'm taking advantage of the fact that the output of video games is algorithmically generated, computationally using a combination of in-game models and textures to specify the content of each image. Thus, we can "style transfer" onto the assets/textures used by the game, effectively getting similar results. Analogy: doing style transfer at "compile-time" instead of "run-time."

3.2 Competitive Analysis

As mentioned above, most of the existing solutions out there are extremely computationally demanding. The only working demo that does style trans-

fer in remotely real time is NVIDIA's demo, which requires 4x Titan Vs, something far beyond a typical workstation.

Furthermore, it doesn't seem like anyone has attempted to apply style transfer to video games at all, which means this project represents something entirely new.

3.3 Structural Plan

In terms of code, my source files will be divided into three primary files:

1. application.py: Implements the main user interface in QT. Handles function calls and other changes as necessary.

2. alg.py: The actual implementations of the two algorithms described above.

3. fileManager.py: Various utility functions that are used throughout the previous two files, with a particular emphasis on file system management.

3.4 Algorithmic Plan

3.4.1 Terms:

Style Image: the image from which the "Style" of the desired output image is extracted
Input/Content Image: the image that dictates the "content" and overall geometry of the desired output image

3.4.2 Convolutional Neural Network + Gradient Descent:

Using a pretrained neural network (traditionally, vgg16/vgg19), we can quantify "style" and "content" differences between two image by looking at various layers of the convolution. To do this, we simply run each image through the neural network, examine the desired layer, and take the RMS difference between each component in the tensor. This gives us two different notions of "distance": style distance and content distance. Our goal is to minimize these.

To do this, we simply compute the gradients of each entry of the "image tensor," and perform gradient descent on the entire tensor.

After the desired number of iterations, we return the output image and save it to a file.

3.4.3 Cycle GANs:

Two primary components: Generator Discriminator First, we write a generator that naively does some basic operations on the input/style images. Most importantly, this generator is very fast, and can produce multiple images from a given pair of inputs very quickly.

Then, we write a discriminator that can pick the "best" image out a set of candidate images, and run the generator again on this "best" image. As part of the "cycle" aspect, we also frequently regularize the outputs of this as to not get stuck in local maxima.

3.4.4 Complete toolchain:

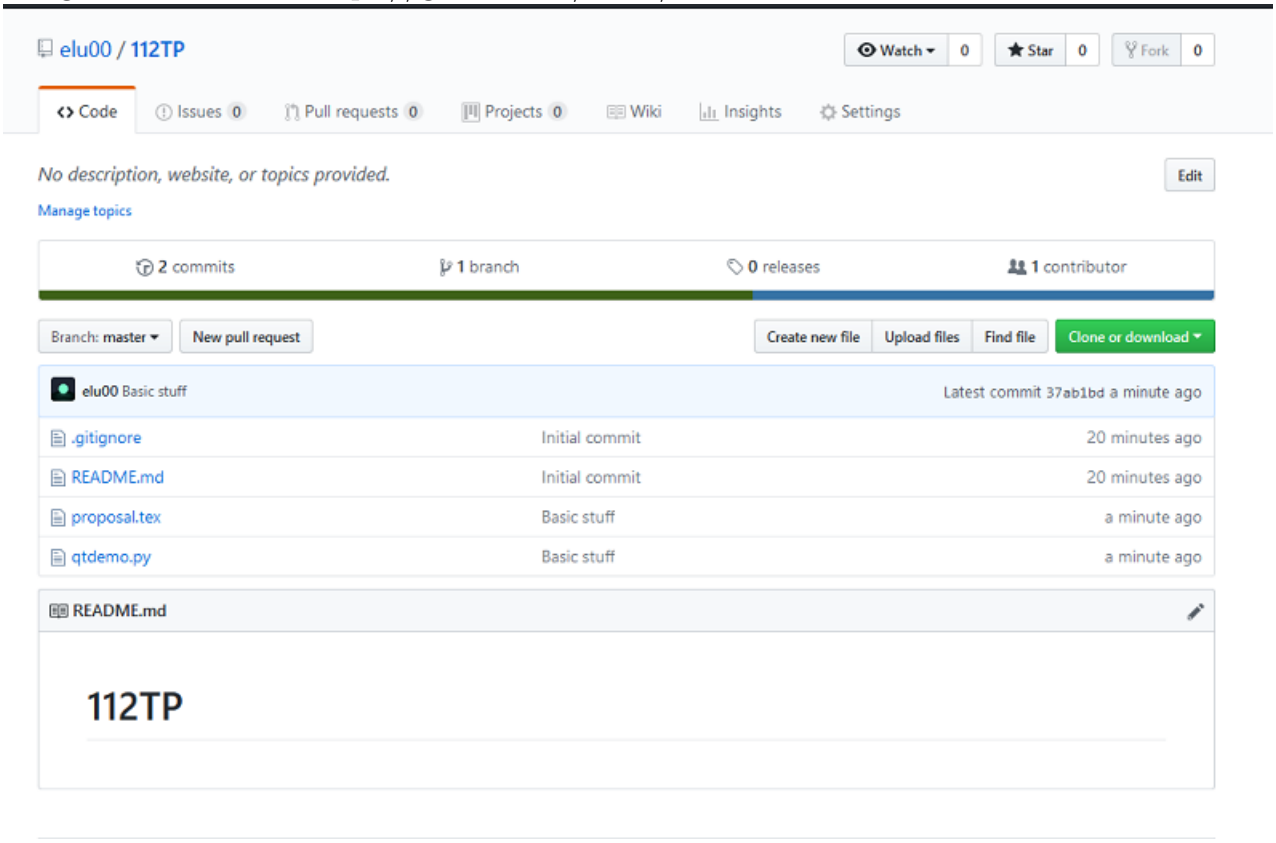
Main technologies: Pytorch, QT, Dolphin Emulator (for actually running the games).
Workflow:
(Optional/computationally demanding) Extract textures from the game using Dolphin Emulator, facilitated by a QT GUI.
Parse the extracted textures from dolphin with Python builtin file operations
Run the style transfer algorithms on above images with custom implementations in Pytorch
Replace the original textures in dolphin
Play the game again, using a button somewhere in the GUI.

3.5 Timeline Plan

TP1: Have user interface/GUI designed and implemented. Seperately, have a working version of the convolutional NN algorithm.
TP2: Bind the two together, add necessary filesystem manipulations to the GUI.
TP3: Add synthetic benchmarking and related graph/statistic visualization.

3.6 Version Control Plan

I'll be using GitHub to backup/version control my code. The repo I'll be using can be found at https://github.com/elu00/112TP.



3.7 Module list

1. Pytorch/torchvision + PIL.
2. PyQT for QT bindings/User Interface.
3. Built-in Python packages (os, subprocess) for file manipulations.

4 Storyboard

See storyboard.png

(1) The default user interface

(2) Styles can be dynamically changed using the dropdown menu

(3) After clicking "play game" ....

(4) Dolphin will launch with the modified textures

(5) If no preview image is available, fallback to the style image.

(6) New styles can also be added with an input image.