Automated Replication of Art Styles in Shaders through Generative Adversarial Networks

Student name: Curtis Wiseman

Student ID number: 15012838

Supervisor name: Tom Bashford-Rogers

Date submitted: 26th of April 2017

Module: UFCFS4-30-3 Creative Technologies Project



Summary

A combination of the deep convolutional network technology of a generative adversarial network with the linear optimisation capabilities of a genetic algorithm to manipulate pre-set shaders into approximating a given cartoon-like art style. The logical systems are written in Python 3.6 and the rendering is performed in Unity 2017.1.

Brief biography

Since a young age I have always appreciated the visual aspect of games and films, often being just as inquisitive about the technical achievements in this regard as the artistic ones. I intend to pursue technical art for 3D animated media in my own career and being able to offer a tool that provides a layer of mutual communication between the creative artist and the technical programmer would represent my strengths as a bridge between specialists.

How to access the project

Project and code URL: https://github.com/CurtisWiseman/CTP---Art-Style-Replication/

Please pull from the commit labelled “SUBMISSION”.

The project is written in Python 3.6 and can be accessed through ArtStyleReplicator.sln. There is a list of dependencies in README.md. Make sure Python 3.6 and all the dependencies are installed if running the program from Visual Studio. A standalone executable was planned but not completed, apologies for any inconvenience this may cause.

Be sure that the UnityScreenshots folder is empty before running the program.

When running the Python program, it will ask you to open and run Unity. The executable ToonRendering.exe, found in the ToonRendering folder, will need to be opened. After opening, switch back to the Python window and press enter. The program should now automatically run without further input. If you accidentally stop or close the Unity executable, you can open and run it again without resetting the Python program, and it will continue where it left off.

IMPORTANT NOTE: Due to an unknown error, the automatic window-switching may fail when moving the mouse. Such failure will result in a Python crash. To avoid this, please refrain from moving the mouse while running the program. The same crash can also be caused by not clearing out the UnityScreenshots folder before starting the program.

The program would take several hours to run to completion, but a few minutes is enough to build up a selection of generated images.

Video URL: https://youtu.be/GeQhDN4QNCc

Introduction

Many 3D video games and animated films try to replicate the aesthetics of 2D animation. Films may simply be doing so as an artistic choice, like with the 2012 short film *Paperman*. This can also happen in video games, as in Clover Studios’ *Ōkami* (Clover Studio, 2006), but oftentimes games are trying to mimic a cartoon or comic book that the game is based on. Examples of this include *The Simpsons Game* (EA Redwood Shores, 2007) and *Teenage Mutant Ninja Turtles: Mutants in Manhattan*. (PlatinumGames, 2016)

The issue therein is that 3D animation is fundamentally different to 2D animation in many ways, and so replicating the style of one in the other can be difficult. 3D animation may have been popularised in the 1990s by the likes of Pixar, but even to this day studios struggle to bridge the gap between the two mediums. To do so, shaders must be created to mimic 2D hallmarks such as limited shading, linework, and hatched shadows. 3D modeller Tom Oliver details some of the ways in which 3D sequels to 2D animated series often fail in the transition in his video essay series, *Why 3D Anime Fails*. (Oliver, 2017) The most impressive attempts, such as those shown by Arc Systems Works in their fighting games *Guilty Gear Xrd* and *Dragon Ball FighterZ*, take so much time and effort to craft that it may even be more economic to authentically 2D animate them. (Arc System Works, 2018)

The purpose of this project then was to create a tool that could be used by these studios to automate much of the shader creation process, freeing up the studio’s workforce to focus on other tasks. It should be able to take in a set of images displaying the 2D art style wanting to be replicated, and, within a reasonable timeframe, produce a set of shaders. These shaders, when used together in a 3D rendering program, will give the impression that the 3D objects being rendered are in fact 2D drawings.

It was also an exercise in experimentation, to see if it is truly possible for a machine to learn to replicate art styles more efficiently than a human. The recent upswing in machine learning research shows that the focus has shifted, from building smarter artificial intelligence (AI) machines to building ones that can make themselves smarter. It is possible that in the coming years, AI will have a much larger hand in translating artistic intent into a digitally replicable form than ever before.

All in all, the objective of this project was to produce a relatively easy-to-use program that could be set running and left, with the user returning later to find a convergence on an appropriate shader configuration. A Python program and Unity application are to be used in conjunction for the full pipeline to be functional.

Practice

A program has been produced that adequately performs the tasks set for it. Tested on both frame data extracted from episodes of *Skeleton Warriors* (Goddard, 1993) and a 6000-page archive of the comic *Homestuck* (Hussie, 2009), both known for their distinct art styles, it configures shaders that mimic them as closely as possible within the parameters of the shaders provided. Given robust enough shader parameters and plentiful training time, the configurations produced would theoretically converge towards accurate replication. The data shown by the limited training performed does show a convergent trend, though whether it is towards better mimicry of the styles in unclear. The biggest obstacles preventing further progress are the base shaders being used. They are simple and do not incorporate advanced effects such as normal manipulation or post-processing effects like bloom lighting or screen-space colour grading. Expanding the repertoire of available effects would greatly enhance the program’s ability to produce more useful configurations. This would require a large amount of human input. However, the base shaders would only need to be written once, thus meaning that less human intervention would be required once a significant shader library is built up.

Initially, it was intended for the program to configure two “ubershaders”, large shaders that incorporate many effects each; one would be applied to dynamic or “foreground” objects and the other to static or “background” objects. This idea was dropped for the sake of time, and because of a general move away from ubershaders. The reason for the move to instead use a collection of regular shaders was to increase compartmentalisation and customisability by allowing for the end user to outright disable or enable certain effects as they please. If an effect in a ubershader is not being used, then either it is still being calculated and multiplied by zero or there is a conditional branch that averts the code section entirely. Both are relatively costly procedures when compared with simply removing a specialised effect shader entirely. However, the program can still be used with an ubershader if the user has one prepared. Aside from the shaders themselves, the only other input needed is an XML file containing the names of all the shader properties to be trained on by the program. The properties must be represented as floating-point numbers normalised to the range 0-1, but within the shader itself this can be multiplied to a more fitting range if needed.

The main program was written in Python 3.6, which was chosen because of the useful machine learning libraries that have been made for it. One used for this project is TensorFlow, the tool of choice for producing generative adversarial networks, also known as GANs. (Goodfellow et al., 2014) It was decided to use a GAN because they are an emerging technology and a hot topic for machine learning computer scientists at the moment. GANs are networks that generate new data mimicking the controlled input data by pitting two competing networks against each other, one learning to produce more convincing data while the other learns to discern the difference between real and generated data. The capabilities on GAN learning are being pushed on many fronts, but during the research phase, no evidence of them being used to generate shaders was found. This opportunity to explore a relatively uncharted field of research was a major motivational factor in pursuing it.

However, GANs are defined in such a way that they specifically generate data in the same format as the input data; if the input data is an image set, the generated data will all likewise be images. This is a problem when trying to produce shaders, as their effects are visual and therefore best represented by images, while the data that controls them is parameterised and made of floating-point number variables. These two conflicting formats would not work together in a normal GAN, as became clear during my research. Instead, it was decided to modify the underlying structure of the GAN to suit the two formats. The discriminator network, that classifies data as real or generated, is already suitable as shaders are judged on the appearance of objects rendered with them. The generator network required a complete overhaul though. Most of its functionality was replaced with a genetic algorithm. Genetic algorithms (GAs) are loosely based on the theory of evolution, wherein solutions, represented by individuals in a larger “population” of solutions, are at first randomly generated. They then are tested by some metric to decide their “fitness” and the most fit individuals are cross-referenced to produce “offspring” individuals. Some randomly-picked individuals are selected for this “breeding” process as well for the sake of diversity and to avoid reaching a sub-optimal solution, and parts of individuals are sometimes randomly changed for this purpose as well. This implementation is a good fit for shader optimisation. The solutions are dictionary objects containing the names of different effects and a float value for each of them. The discriminator network’s judgement on how likely the solution’s rendered output is to be real makes for a well-defined fitness function. There is also a theoretical optimal solution, and genetic algorithms are guaranteed to find optimal solutions if their fitness functions are smart enough and if they have enough time to iterate through a sufficient number of generations.

There were, naturally, difficulties in combining the two approaches. A GAN implemented in TensorFlow deals almost entirely in tensors, an abstract data type that can be cumbersome to work with outside of functions specifically designed to accept them. The biggest hurdle in this regard was translating the output from the discriminator network into something the GA could use for its fitness value. For all its calculations, the GA requires that the fitness of its population’s individuals be a float or double, or similar high-precision basic type. Tensors are somewhat like multi-dimensional arrays. However, they do not make their values easily obtainable as they are not stored in the same way. Tensors act more like instruction sets, being evaluated on-demand within a TensorFlow session. This challenge was overcome by reducing the tensor down to zero dimensions, evaluating the value in a new TensorFlow session into a more easily-readable array and then copying the useful data into a new variable.

Another issue was rendering test images with the shaders being configured. It was decided early on that the rendering would be done in Unity due to its high compatibility with various shaders and ease of use. Unfortunately, Python code does not run natively in Unity’s C#-based pipeline. Because of this, making the Python code and Unity renderer work together presented a challenge. Both programs had to bridge the gap in different ways. The Python code would serialise the shader data into an XML file with a unique identification number as its first node. Unity, then, would be constantly monitoring for changes in that ID number, and reading the full file whenever a change was made. The file data would then be unpacked into the corresponding shaders’ run-time attributes. Because shaders can be updated in real time, Unity would then take a screenshot using the built-in screen capture method and save it to a new file with the aforementioned ID number as its file name. The Python program, at this point, would be anticipating a new file with that ID number to be created, and would therefore be ready to read and decode to be passed back to the discriminator network for fitness testing. There were issues wherein the Unity side would only call its update function once, and the only workaround outside of restarting the program would be to click on a different window and back onto the Unity one every time an update was needed. This was eventually resolved by having the python program force window focus back and forth whenever Unity was needed. An inelegant solution, perhaps, but a necessary one.

The three shaders used for testing were an outline shader, cel shading, and cross-hatching. These were downloaded from the internet but modified to be easily configurable via trainable float values. The reason these three were chosen is because they are visually distinct but quite common in drawings and animation. Outlines and cell shading are more common in dynamic objects as they are easier to draw many times, whereas cross-hatching is rarely used on characters and is more often found in background art. However, it is possible to use all three together, each with different weights and colours. While a broader range of effects would have made the system more versatile, that is something more easily implemented by the end user. The system is left open so that new shaders can be added by adding them to the Unity project and providing entries for them and their trainable values in the XML file. After all, new shader techniques are being developed all the time, so leaving the door open to future possibilities is a must.

Discussion of outcomes

The research done here proves that GANs are not limited to image generation, nor do they need be constrained by having inputs and outputs of the same format. While it may not mean much in the wider scope of machine learning, small steps like this help ensure that progress is being chipped away at. This kind of research may open the door to future endeavours for machines to learn processes rather than outcomes. A machine could learn to replicate a style of music from the instruments up, or to design house floor plans in an aesthetically-pleasing manner by placing furniture and appliances individually. Having building blocks, human-designed but modifiable elements at its disposal, the machine would have a much more accurate set of tools to work with when compared to traditional neural network generation. In a regular neural network, the machine tries to guess the construction of an object from the pixels, or other primitives. This leads to a lot of networks failing to recognise spatial cohesion, whereas a network that builds solutions from more easily-labelled parts would be able to more easily infer how elements interact to form a whole. The total number of possible outcomes would be reduced, but the ones remaining would be more likely to pass a human-run quality test.

With that said, the implementation here is not quite so impressive. Its outputs are extremely limited due to the small number and simplicity of shaders used. It has the potential to be a broader and more robust program given further work, but it is more substantial in theory than in practice. Combining this tool with advanced shaders such as the ones found in the SIGGRAPH ’03 paper Coherent Stylized Silhouettes could lead to extremely style-accurate shaders with minimal human interaction. (Kalnins et al., 2003)

The project has changed course since the original proposal. Initially, it was intended to simulate the motion of 2D animation in 3D by using reverse kinematics to alter animations in real-time. This proved to be too ambitious for the scope of this project. As explained by Arc Systems Works technical artist Junya Christopher Motomura in a GDC talk, simulating the motions of imprecise brush strokes is best done manually, and even then is a highly time-intensive task. (Motomura, 2015) The decision to take the project in a different, more machine-learning-orientated direction greatly increased its feasibility.

Conclusion and recommendations

Overall, the project is a theoretical and research success, but the practical implementation is barebones and lacking in substance. More work needed to be put into polishing the user experience and better showing off the program’s capabilities.

The possibilities of combining machine learning types open up many more opportunities for developers to find hybrid solutions – much like the breeding function in a genetic algorithm, the best qualities of different approaches could combine to make one greater than the sum of its parts. Perhaps, in the future, it may even reach the point where a machine learns how to produce better machine learning algorithms itself.

This project, in particular, has the potential to be a powerful tool if given proper care and attention past its initial release. With a large library of shaders representing a breadth of effects at its disposal, it could be used to automate much of the process of turning an artist’s vision into dynamically-rendered visuals on the screen. Artists and programmers alike could use it to bridge the gap between their specialities.

References

Arc System Works (2018) Dragon Ball FighterZ. Yokohama, Japan: Bandai Namco Entertainment.

Clover Studio (2006) Ōkami. Japan: Capcom.

EA Redwood Shores (2007) The Simpsons Game. Redwood Shores, CA, USA: Electronic Arts.

Goddard, G. (1993) Skeleton Warriors.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. & Bengio, Y. (2014) Generative Adversarial Networks.

Hussie, A. (2009) Homestuck. [online]. MS Paint Adventures. [Accessed 7 April 2018].

Kalnins, R., Davidson, P., Markosian, L. & Finkelstein, A. (2003) Coherent Stylized Silhouettes. In: SIGGRAPH. [online]. 2003 New York, NY, USA: ACM. pp. pp. 856-861. Available from: https://dl.acm.org/citation.cfm?id=882262.882355 [Accessed 16 April 2018].

Motomura, J. (2015) GuiltyGearXrd's Art Style: The X Factor Between 2D and 3D.

Oliver, T. (2017) Why 3D Anime Fails. [online]. Available from: https://www.youtube.com/watch?v=0fEtGrHfduA [Accessed 21 April 2018].

PlatinumGames (2016) Teenage Mutant Ninja Turtles: Mutants in Manhattan. Osaka, Japan: Activision.

Bibliography

AnomalousUnderdog (2009) Silhouette-Outlined Diffuse. Available from: http://wiki.unity3d.com/index.php/Silhouette-Outlined\_Diffuse [Accessed 18 December 2017]

Leon, D. (2015) Next-Gen Cel Shading in Unity 5. Available from: https://www.gamasutra.com/blogs/DavidLeon/20150702/247602/NextGen\_Cel\_Shading\_in\_Unity\_5.php [Accessed 18 December 2017]

Halladay, K. (2017) A Pencil Sketch Effect. Available from: http://kylehalladay.com/blog/tutorial/2017/02/21/Pencil-Sketch-Effect.html [Accessed 18 December 2017]

Lespol (2014) Horse(1). [online]. Available from: https://www.turbosquid.com/FullPreview/Index.cfm/ID/810753 [Accessed 20 April 2018].

Buonarotti (2013) Pineapple. [online]. Available from: https://www.turbosquid.com/FullPreview/Index.cfm/ID/741563 [Accessed 20 April 2018].

Tosca, P. (2014) Old Man. [online]. Available from: https://www.turbosquid.com/FullPreview/Index.cfm/ID/864833 [Accessed 20 April 2018].

Appendixes: (not included in word count)

Log sheets

September 6th: I have made a good second draft of my project proposal. My project will be to produce a set of shaders and an animation handler that will be customisable by the end user to make their 3D animations look like various styles of cartoon. From my research, I have found that making these “toon shaders” is time-consuming and the motion of 3D animation makes it obviously distinct from 2D animation unless the animators put in an awful lot of work into each and every frame of animation. I want my project to offload much of that process to a set of pre-defined effects that can provide a good basis for artists to prototype different art styles and see what they will look like in motion.

September 26th: Preliminary meeting with supervisor. He has suggested that creating an animation handler is very difficult and I should look into reverse kinematics if I want to continue with the project in its current form. He does like the idea of creating a tool to aid 3D artists but worries that the project’s current focus may be out of scope.

October 10th: Due to my supervisor’s advice, I have decided to rethink the framework of my project. Instead of directly making shaders and other run-time assets, I will instead be making a program that itself creates shaders. I do not yet know how this will work but it is a relatively unexplored area of technology and there is a lot of potential to develop something challenging and unique.

October 26th: The final version of my project proposal is submitted. My program will be a generative adversarial network (GAN) that is trained on images of 2D cartoons and outputs a large, all-encompassing shader (ubershader) that mimic the training data’s art style. I will write it in Python 3 because it is well-suited to repetitive machine training thanks to tools like TensorFlow.

November 21st: I have been researching GANs and how to implement them in Python 3. I have followed a tutorial and written a GAN that trains on MNIST data to recreate handwritten numbers. I will use this as a base when producing the final GAN for the project.

December 12th: My supervisor has suggested that a genetic algorithm may be an efficient method of generating shaders. I will try to link this into the GAN, perhaps by replacing the generator network with a variation of a genetic algorithm. Something like this will be necessary since a traditional GAN cannot output data in a different form than its input data; it is designed to replicate its input data as convincingly as possible.

January 10th: I have finalised the structure of my program. It will primarily be based on a GAN, but the generator network will be replaced by a genetic algorithm that defines variables for two ubershaders - one for static objects and one for dynamic ones. A test scene will be rendered with these shaders and the image renders from that will be passed back to the discriminator network for testing.

February 24th: I have decided against using ubershaders. They are too unwieldy and would discourage developers from using the tool, as they would have to write their own ubershader rather than be able to use existing shaders. Having separate shaders may also be more efficient if some effects are not in use.

March 29th: Uploaded project to GitHub. Moved genetic algorithm code into GAN, though they do not interact yet. I am not sure how exactly I will translate the inputs and outputs of the genetic algorithm to and from TensorFlow Tensors. They appear to be fundamentally different.

April 19th: Have been working on other modules for the past three weeks due to deadlines. Ready to focus solely on this until submission. Integrated Unity into the pipeline for rendering purposes.

April 22nd: Properly got the genetic algorithm integrated. If not for bugs, the whole pipeline should be working now.

April 25th: Got everything working, made video and wrote the report. All ready for submission.

Project timeline (Initial plan)

