Design and Implementation of Style-Transfer Operations in a Game Engine

Haechan Park[1], Nakhoon Baek[2]
School of Computer Science and Engineering, Kyungpook National University, Daehak-ro 80, Daegu 41566, Korea[1,2]
Graduate School of Data Science, Kyungpook National University, Daehak-ro 80, Daegu 41566, Korea[2]
Data-Driven Intelligent Mobility ICT Research Center, Kyungpook National University, Daehak-ro 80, Daegu 41566, Korea[2]

***Abstract*—The image style transfer operations are a kind of high-level image processing techniques, in which a target image is transformed to show a given style. These kind of operations are typically acquired with modern neural network models. In this paper, we aim to achieve the image style-transfer operations in real time, with the underlying computer games. We can apply the style-transfer operations to the all or part of rendering textures in the existing games, to change the overall feeling and appearance of those games. For a computer game or its underlying game engine, the style-transfer neural network models should be executed so fast to maintain the real-time execution of the original game. Efficient data management is also required to achieve deep learning operations while maintaining overall performance of the game as much as possible. This paper compares several aspects of style-transfer neural network models, and its executions in the game engines. We propose a design and implementation way for the real-time style-transfer operations. The experimental result shows a set of technical points to be considered, while applying neural network models to a game engine. We finally shows that we achieved real-time style-transfer operations, with the Barracuda module in the Unity game engine.**

***Keywords*—Style transfer; neural network models; game engine; rendering textures; real-time operations**

I. Introduction

Game engines are now general development tools, which help users develop games conveniently through providing the functionalities needed to develop any kind of computer games. A game engine typically includes a *rendering module* that draws objects on the screen, a *physics simulation module* for adding physically-simulating effects, such as collisions and/or gravity effects, a *sound module* for background music and/or sound effects, and an *event processing system* for user input and system events [1].

It can also support network communication features, external database connections, and on-line storage connections for storing or retrieving information about users or data for the game. A game engine is now typically a complex and heavy program, since it provides various sets of functions, as show here.

Recently, game engines are emerged to support machine learning features, typically as *add-on* modules. Actually, in artificial intelligence and machine learning fields, various researches and developments have drawn much attentions from general public persons. For example, *AlphaGo*[2] which is artificial intelligence in the game of *Go*, *AlphaStar*[3] which is artificial intelligence in the *StarCraft 2*, and *Vizoom* [4] which is a study that trains neural networks to play the first-person shooter game *Doom* through visual information. These artificial intelligence works are implemented based on the *deep reinforcement learning* [5]. Due to the remarkable achievements of deep reinforcement learning, many studies of artificial intelligence used in the computer games are mainly focused on reinforcement learning.

In fact, the artificial intelligence methods can be used in a variety of ways, even in game-specific applications. Recently, artificial intelligence has also been used in the areas of creation, such as painting pictures as sophisticated as photographs [6], writing [7], or music composing [8]. Several major gaming companies carry out research on the generation of images, animations, music, etc. to be used in games across various neural networks. These game resources are usually created outside of the game engine, and the generated resources are provided in the general file formats. The game engines read those resource files, and then use them in the games.

We expect that it would be innovative to generate resources such as images, videos, and sounds through neural networks inside the game engine and apply them to the game in real-time. However, most generative neural networks are too slow for real-time applications. Thus, applications of neural network models to real-time games are limited to relatively simple neural network models to quickly generate game resources.

In this work, we focused on the convergence development of real-time games and artificial intelligence models, through applying generative neural networks in real-time. More precisely, we focused on the *image style transfers* [9] to generate the texture images in real-time, as game resources, as shown in Fig. 1.

The texture images are used for various purposes in the game, and thus, our work can also be extended to the various applications. In contrast, the style transfer is not possible to be implemented in other ways except machine learning methods. Thus, it is a good case study for game engines, which support machine learning features. The image style transfer is even more practical, since it can be easily adopted for the special effects in games.

Applying style transfer methods to the real-time games shows some technical issues. Firstly, the resulting images of the neural network models should be generated quickly so that it can be applied in real-time. Thus, efficient architectural models for running the game itself and also the neural network models simultaneously are needed.

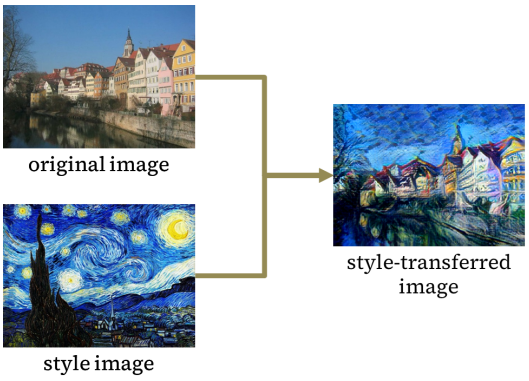

Fig. 1. An example of the style-transfer operation.

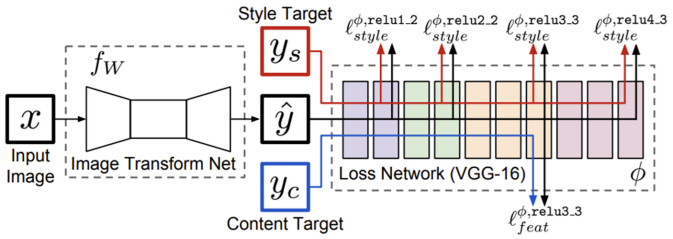

Fig. 2. The overall structure of the feed-forward style-transfer neural network model [11].

In this paper, we will solve those technical issues. From the viewpoint of execution speeds, we aim to achieve real-time performance even with applying neural networks that perform style transfer. We will use a commercial game engine, as the case study, which provides machine learning model drivers to analyze and optimize the inter-working method to check performance and limitations.

II. Related Works

The *style transfer method* can be achieved by neural network models which create the resulting images by transferring the image styles of the given style images to the content images. The work performed by the neural network model can be checked through the resulting image created with the content image and style image, as shown in Fig. 1.

The key concept in the style transfer method is that it is possible to separate content and style from an image, by extracting features of the image through a set of trained convolution layers [10]. The initial style transfer method [9] operated by gradually transforming the input image through gradient descent, and it took a long time to obtain the resulting image, making it difficult to process video streams or real-time applications.

Later, the *Feed-Forward style transfer method* [11] was proposed, which solved the problem of slow conversion speed. This method generates the transformed image at once through the *encoder-decoder* networks. As shown in Fig. 2, this method creates a transformed image by receiving the content image from the *Image Transform Net*, which is actually an encoder-decoder network. The input image is separated into the content image and the style image. The trained *VGG* (Visual Geometry Group) neural network [12] calculates content loss (L_c) and style loss (L_s), respectively, and then trains the *Image Transform Net* through the back-propagation. Since the entire neural network is trained on a single style image, it can be suitable for a single style transfer.

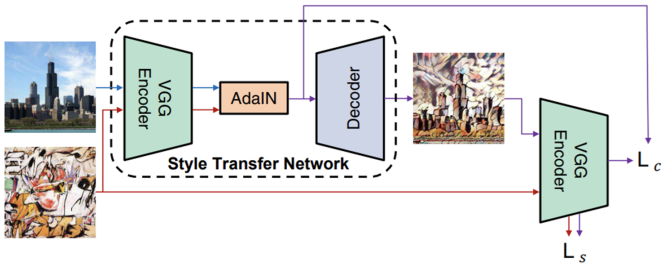

Fig. 3. The overall structure of the AdaIN style-transfer neural network model [13].

Style transfer methods with *Adaptive Instance Normalization* (AdaIN) [13] can be used for arbitrary number of style images, even in real-time or at least in pseudo real-time. The AdaIN layer calculates the *mean* and *variance* for each channel of the features extracted from the content images and the style images. Then, it normalizes the means and variances of the content images, with respect to the means and variances of the style images.

The *AdaIN* operations can be summarized as the follow equation:

$$\text{AdaIN}(x, y) = \sigma(y)\frac{x - \mu(x)}{\sigma x} + \mu(y), \qquad (1)$$

where μ is the mean and σ is the variance function, from statistics. The parameter x is the feature values extracted from the content image through the VGG encoder, while the parameter y is the feature values from the style image through the VGG encoder.

As shown in Fig. 3, VGG encoder is trained by classification tasks and network weights are fixed. Since AdaIN has no parameters used for learning, training is performed only on the VGG decoder part. There are two loss functions used for training: *content loss* (L_c) and *style loss* (L_s) functions. The differences between the results of applying VGG encoder to the images that has been constructed through VGG decoder, and the results obtained when the content images go through VGG encoder and the AdaIN layer, are defined as the content losses. Similarly, the differences between the results of applying VGG encoder to the images that has been constructed through VGG decoder and the style images are defined as the style losses. The final objective function is the sum of content losses and style losses, and the VGG encoder is trained to minimize it.

For a computer game or its underlying game engine, the style-transfer neural network models should be executed so fast to maintain the real-time execution of the original game. Unfortunately, the previous works are insufficient to get real-time results, and we aims to get the style-transfer operations in real-time or even in pseudo real-time. In the next sections, we will show our experimental details and the results.

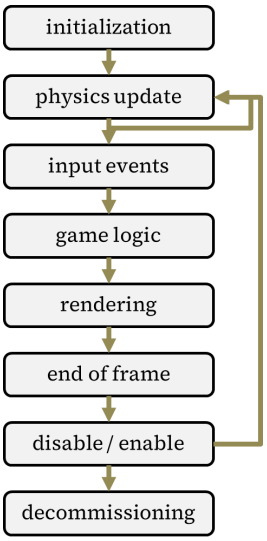

Fig. 4. A typical execution order of Unity event functions.

III. Experimental Environment

A. System Configurations

Our system configuration for the experiments are as follows:

- CPU : Intel Core i7, 2.30 GHz
- GPU : NVIDIA GeForce RTX 2060
- Main Memory : 16GB
- Operating System : Windows 10 Pro 64bit Edition

As shown here, we used typical commercial PCs with mid-tire computing powers, rather than high-tire ones, for more practical uses and more real-world experimental results.

B. Game Engine Integration

Our work is carried out on the *Unity* game engine. It manages various functions necessary for game development in module units. Each module must be operated efficiently at an appropriate time, according to the characteristics of the game to be made, to obtain a positive response from game service users. Each module is controlled through *event functions*. Event functions perform actions that need to be handled at a specific point in time or situation in the game.

The Unity game engine provides many event functions to control modules in various situations, so it is inadequate to mention all of them in this paper. Taking it into consideration, the execution orders of the event functions are iterated in groups, according to their purpose, as shown in Fig. 4.

At the start of the event processing loop, event functions for *initializing* the game are called, and then event functions related to *physical effects* are called. Physical effects are updated separately from the main thread at a specified time through a reliable timer system. By applying this method, if the physical effects are not updated within a fixed time interval, it can be applied inaccurately unlike in reality, but it is possible to prevent unexpected situations, in which screen rendering is delayed until all the physical effects are applied.

TABLE I. Comparison of *Unity* and *Self-Engine* Game Engines, According to the Machine Learning Features

game engine	Unity	Self-Engine
machine learning module	ML-Agents	Neuro
programming language	Python, C#	C++
training	Pytorch, Tensorflow	cuDNN
inferencing	Barracuda	cuDNN
additional features	support reinforcement learning	lightweight

Then, the *input* event functions are called to handle user inputs to interact with the game. Next, *game logic* event functions that perform calculations for various decisions to be performed in the game are called, and event functions that draw the screen with the data updated so far are called. Next, the event function that defines the action to be applied to the *end of the frame* is called, and the event function specifies the action to be performed when it is decided to make the object be *disabled or enabled* in the game scenario is called. Finally, when the game ends, event functions for resource *decommissioning* are called.

C. Machine Learning Modules for Game Engines

Before starting our full-scale experiments, we investigated several game engines, for their features which support machine learning models, and also the overall environment for the inference executions. At that time, the Unity engine is one of the best-suitable commercial game engines, and it officially supports machine learning features. We also compared those features with the *Self-Engine* [14], which is an open-source, lightweight game engine which supports machine learning models, as shown in Table I.

Unity supports the *Unity Machine Learning-Agents Toolkit* (or shortly, *ML-Agents*) [15], which is an add-on module to apply machine learning features to the games. In ML-Agents, the *C# scripting language* [16] is used to train neural networks in the game engine environment. *Python terminals* [17] work in conjunction with it, for neural network training.

For the general cases, the machine learning modules for neural network training in the Python environment can be selected from *PyTorch* [18] and *TensorFlow* [19], which are already familiar to AI researchers. In contrast, since we use ML-Agents for functional inferences, we had better to use *Barracuda* [20], which is a specialized module that executes pre-trained neural network models with C# scripts. Additionally, ML-Agents support other various functions to efficiently perform reinforcement learning.

Self-Engine uses *Neuro* [21] for its machine learning module. Both the game engine itself and the machine learning module are implemented in the same C++ programming language and thus, the machine learning module is naturally integrated into the game engine. With the integrated machine learning module, neural network training and inferences are performed in the similar manner. In the Self-Engine, the machine learning

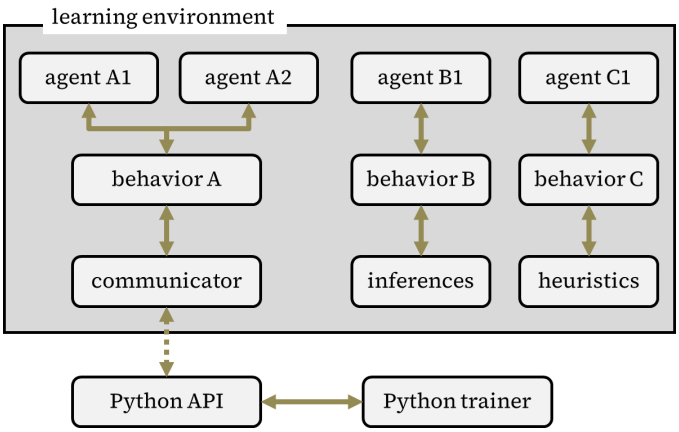

Fig. 5. The internal structure of the ML-Agent module.

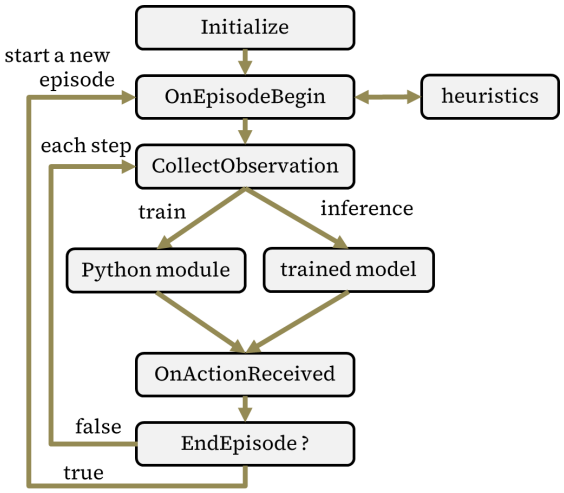

Fig. 6. The internal structure and function calling orders in the agent class.

module of Neuro internally uses *cuDNN* [22] as its underlying deep neural network operation tools.

Actually, Self-Engine is a lightweight game engine implemented as an open-source, and its internal structure is less complex than that of commercial game engines. However, at least at this time, most of artificial intelligence researchers prefer PyTorch and TensorFlow, and Self-Engine has fewer convenient features compared to Unity, which officially supports machine learning modules. Conclusively, in our experiments, we use the Unity game engine, as our major target system.

IV. Implementation Details

A. ML-Agents Features

The ML-Agents performs training between the *Unity editor* and the *Python terminal* via *Inter-Process Communication* (IPC), as shown in Fig. 5. The two processes conduct training by exchanging data with each other in socket communication. For this purpose, ML-Agents must be installed in both Unity editor and Python terminal respectively. ML-Agents installed on the Unity editor side is implemented in the C# programming language, which is part of the Learning Environment. ML-Agents installed on the Python side is the Python API part, as shown in Fig. 5.

After each installation, the Unity editor needs to set up a project to use ML-Agents. An object to be used as an agent must be placed in the scene of the Unity editor, and Behavior Parameters and Decision Requester script must be added to the object as components. In the Behavior Parameters script, the agent specifies the settings for the information the agent wants to observe in the environment, and when not used in the learning mode, the neural network model saved as an *ONNX file* [23] to be used for inference.

After that, we implemented the actions to be performed by the agent by inheriting the Agent Class, and also the observation data to be sent to the Python module during training and the processing to be performed when the data is received from the Python module. After implementing the internal functions by inheriting the Agent Class, write a Python script using the ML-Agent Python Low-Level API to access the Unity process from the Python terminal. Fig. 6 shows the overall structure of the agent class and its internal functions with their calling orders.

The *Initialize* function inside the *Agent Class* performs necessary works when the game starts. The *OnEpisodeBegin* function implements the initialization work to be performed when an episode starts at each time. After implementing the inside of the *CollectObservation* function that obtains observation information about the environment to be used for the training or the inference, it is divided into the training mode and the inference mode, and the next task to be performed is determined. Mode selection can be set in *Behavior Type* of the *Behavior Parameters* script.

The result from the neural network models being trained in Python terminals or the pre-trained neural network models used for the inference can be applied to the agent through the *OnActionReceived* function. When the episode ends by applying the action output from the neural network model, a new episode is started and training proceeds again. In other cases, the process of receiving observation data in the next step is repeated. The *heuristic* function is used when a user controls an agent directly, without using a neural network model or with the user-provided artificial intelligence.

After implementing the internal functions for inheriting the Agent Class, write a Python script using the ML-Agent Python low-level API to access the Unity process from the Python terminals. When a script is written using the Python low-level API of ML-Agents, the approximate structure of the script code is shown in Fig. 7.

The neural network training method in ML-Agents starts with selecting Unity environments to control with Python APIs. If the build completed executable is selected as an environment to use, the path to the executable must be specified, and if the path is left blank, it will automatically connect to the Unity editor. The successful connection to the Python terminals will result in the reset operation of the Unity environment at the first start. Then Unity conducts simulation by its agents and the Python terminal starts training either PyTorch or TensorFlow neural network models with the received observation data, as the *get observations* step in Fig. 7.

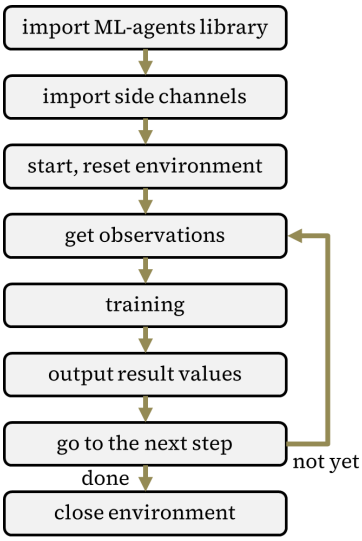

Fig. 7. The ML-Agent python API script structure.

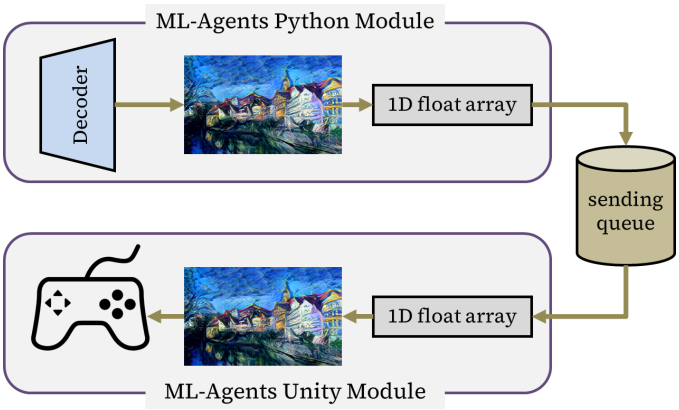

Fig. 8. Our data transmission model, with the side channel.

After that, the training proceeds with repeating the number of steps set in the Python script. At each step during training, information about the agent that needs a decision on the next action to be performed and the agent whose episode has ended collects the output of the neural network results, as the *output result values* stage, and applies it to the next step for agents that need a decision, as the *go to next step* stage. During training, user input does not work in the Unity editors or the built environment, since the Python terminal controls input events in Unity.

Additional work is required to perform the *style-transfer* operations with the ML-Agents. ML-Agents can transmit the information observed through the camera sensors of agents in the Unity environment to the Python terminal. This feature is provided for training deep reinforcement learning networks with environmental observation data obtained from the camera sensors. Performing style transfer on this image is applying style transfer to the screen that the agent sees. However, the Python module of ML-Agents does not support sending images to the Unity process. Currently, ML-Agents only considered transmitting the behavior of the agent to be performed in the next step. To solve this problem, we inherited the Side channel class and used it for sending user-defined data.

The *side channel* is a supported function to transmit user-defined data or inform the user about a specific state during the neural network training process. To use this function, we need to implement the internal functions directly from inheriting the Side Channel class. The side channel implemented in this way can be used in the *import side channels* step, as shown in Fig. 7.

Similar to the structure of ML-Agents, *Side Channel Class* must be implemented in Unity and Python, respectively. On the Unity side, the *Side Channel C# class* is inherited, and on the Python side, the *Side Channel python class* is inherited. The process of data transmission through the side channel is shown in Fig. 8. In Python terminals, after converting the style-transferred image into a one-dimensional array of floating-point values, it is transmitted to the Unity side through the side channel. In the Unity game engine, the received floating-point one-dimensional array is converted into a texture image, which will be rendered on the screen. However, this method transfers the image pixel-by-pixel through the send/receive queue and copies the received float array pixels one-by-one to the texture, which requires a large amount of CPU computing power.

Our style-transfer neural network model is based on the *AdaIN* layer. The model was implemented in PyTorch and used only for inference, when the training was completed. In the case of VGG encoder, the number of filters in the convolutional layer increased from 256 to 512 except for the end of the VGG19 neural network model.

In the VGG encoder, the feature map is extracted from the image by repeatedly stacking the Convolution (Conv2D) layers and the Rectified Linear Unit (ReLU) [24] activation function layers. The subsequent MaxPool2D operation reduces the horizontal and vertical resolution of the image by half. The ReflectionPad2D operation [25] adds spaces to the edges of the output tensors of the current layers, as if reflected in a mirror so that the inverted image appears repeatedly [26].

The VGG decoder also has a convolution layer and a ReLU activation function layer like the VGG encoder [27]. The difference is that the number of filters in the convolution layer is reduced by half, and the resolution of the output tensor is doubled through the added Upsample layer. If the mode of the Upsample layer is set to nearest, the extended part is filled with the same value as the element value of the nearest feature map.

B. Using the Barracuda Module

Barracuda is an inference-specific module built into the Unity game engine. Barracuda executes neural network models stored in ONNX file formats, through C# scripts. Barracuda internally uses the compute shaders [28] to handle the operations required for the neural network inference. These operations are quickly processed in parallel, using the multiple cores of the *Graphics Processing Unit* (GPU). Since Barracuda can run the neural network itself, there is no need to use PyTorch or TensorFlow, there is no cost required for inter-working between the two processes.

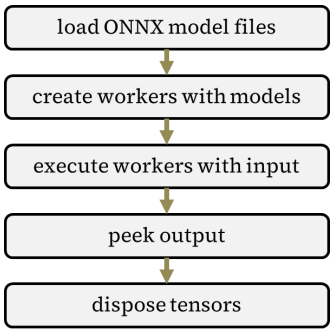

Fig. 9. Overall operations in the barracuda module.

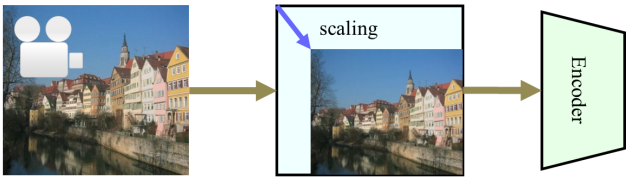

Fig. 10. Input data preprocessing process.

A neural network model executed with Barracuda can use Texture2D data types, which works as an input tensor in the Unity game engine, and vice versa. In the case of ML-Agents, the CPU reads pixel values from the resulting tensor, and injects them into the Texture2D object, actually in a step-by-step manner. In contrast, Barracuda runs faster by simultaneously processing multiple pixels at once, with the GPU. Since Barracuda was developed specifically only for the inference, it does not support any training of the neural network models.

The operating sequences of the Barracuda module are as shown in Fig. 9. A target neural network model will be stored in the ONNX file format, and it will be loaded through the C# script, at the *load ONNX model files* stage. The module creates a worker object corresponding to the loaded neural network model, at the *create worker with models* stage. A Texture2D object is passed as the input tensor to the neural network model, and also as an argument to the worker object, at the *execute workers with input* stage. The resulting tensor will be obtained with the *PeekOutput* function. The final result will be returned as a Tensor type, and can be converted to other types including Texture2D and floating-point number arrays.

To achieve the style-transfer operations with the Barracuda module, we provide the trained style-transfer neural network models as the ONNX files. Those files are imported into the Unity game engine. In our experiments, we use the ONNX files for the neural network models trained with a single style image, in a feed-forward manner [11].

To clearly check the performance of the style-transfer operations, we applied the style-transfer operation to the real-time texture images, from a webcam-based video stream. The video stream provides 60 frames per second, in the 1920×1080 resolution. Since the video resolution is too large

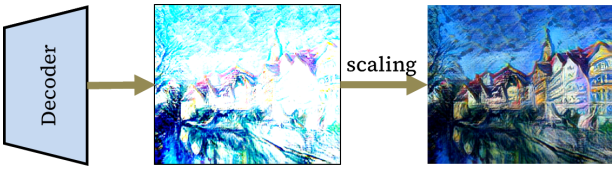

Fig. 11. Post-processing at the decoder.

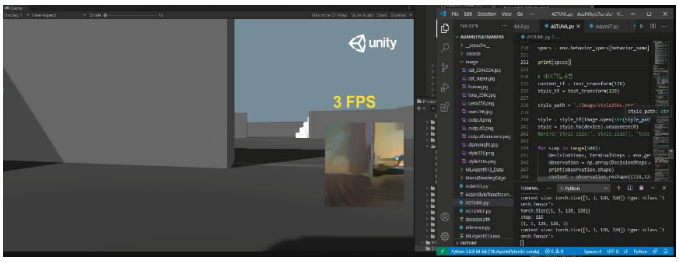

Fig. 12. A brute-force implementation of the style-transfer operation, with the python terminal in the unity engine.

to achieve real-time style-transfer, the video streams are scaled down to the internal render-texture area, with lower resolution. We achieved the highly efficient texture transfers, by using Shader functions of Unity to modify texture images at high speed through GPU's parallel processing features, as shown in Fig. 10.

In addition, appropriate post-processing is needed to the resulting tensor, to be properly used in the games. For example, the final pixel values from the Barracuda module may be normalized floating-point numbers between 0.0 and 1.0, while the rendering module needs 8bit unsigned integer values, as the corresponding color values. We also integrated these kinds of post-processing operations into the Shader programs, for more efficient and faster operations, as shown in Fig. 11.

V. Experiment Result

A. Python-based Implementation

For comparison purposes, we implemented the style transfer operations in a brute-force way, to directly link the Unity editor to the Python terminals, as shown in Fig. 12. As the starting point, the underlying game shows 170 frames per second, without connecting to the Python terminals. Since it was expected that this direct connection would be inefficient, we use low-resolutions of 128×128, for the input video streams. The style-transfer neural network model is executed on the Python side, as an AdaIN network. The resulting style-transferred image is in 256×256 resolution, and read directly from the Python terminal.

Even the style-transfer operations can be achieved efficiently in Python environment, we found that the bottleneck is the data transmission between the Python terminal and the Unity game engine. To check the transmission speed, we once used a dummy Python kernel, which just re-transmit the input data as the result. Since they use network sockets for the data transmission, it shows very slow result of even 3 frames per second, as shown in Fig. 12.

Fig. 13. An example of feed-forward style transfer.

Fig. 14. Applied style image.

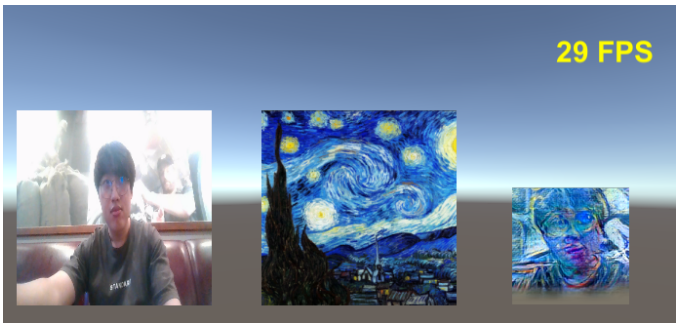

Fig. 15. Experimental results from a sample AdaIN style-transfer implementation.

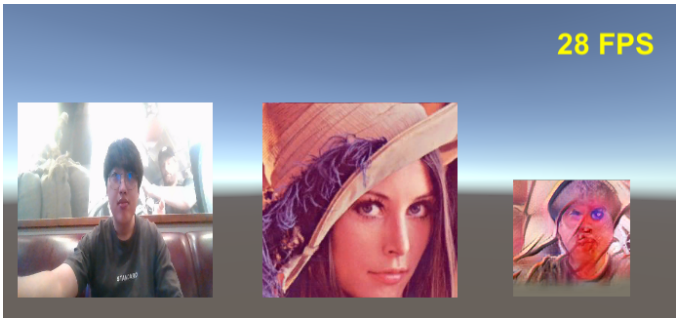

Fig. 16. Experimental results when applying the different style image.

Even worse, the received data need additional post-processing on the CPU side. For example, the color values should be converted from floating-point values to 8bit unsigned integer values for the graphics rendering. Although using shared memory seems more appropriate for the images, this simple use of Python terminals for the style-transfer purpose should consider the optimal connect of two different programming languages: Python and the C# programming language used in the Unity engine. Conclusively, we need another fully efficient way of providing style-transfer operations to the Unity engine.

B. Our Barracuda-based Implementation

In this work, we used a special device driver module, named Barracuda, in the Unity game engine. The Barracuda module is built into the Unity engine, and drives a trained neural network for inference. We first extracted input data of the real-time video streams, from our small size Web Cameras (or webcams). The input data stream is actually texture images extracted from the webcam video streams. It is then resized to be processed by the neural networks.

The style transfer neural networks are used in two aspects: a neural network trained on a single style image in a feed-forward manner, and another neural network with AdaIN, which can apply style images to the target texture, in real-time. The input images extracted from the video camera are converted into the 256×256 resolution render textures, which are used as inputs to the neural networks. The final resulting images are then used as surface textures for cube objects, which are actually a physically-simulated moving object, in our scenario, as shown in Fig. 13.

Fig. 14 shows our sample style image used for this experiment. Since there is little data transmission/reception cost, the final frame rate of the game engine is affected by the resulting texture image generation speed of the neural network models. In our experiment, we confirmed that the game engine works fast enough to be used in real-time.

VI. Analysis of Experimental Results

In our experiments, we found that the *Barracuda* module is specialized in inference, and suitable for the game engines to use the neural network features. This module also shows some limitations: it currently works well with limited sets of neural network models. Actually, we used two style-transfer models: the *feed-forward model* and the *AdaIN model*. The feed-forward model shows fast conversion speeds, and also limitations of being best suitable for a single style image training. In contrast, the AdaIN model can be used for several style images even in real-time, as shown in Fig. 15 and 16.

However, the AdaIN model was suitable for small-size images, due to the real-time requirement. Fig. 17 shows the speed of the style transfer neural networks for input data of various sizes. In this case of the AdaIN models, we found that the size of the image used as the style image was best suitable with 256×256 resolutions. The style transfer performance was greatly decreased with the image sizes larger than this.

For the game engine, it can achieve about 170 frames per second, without any style-transfer operations. Applying the style-transfer operations, the maximum texture image size was 300×300, with our experiment environment settings. Since we used mid-powered PCs, rather than the best-performance

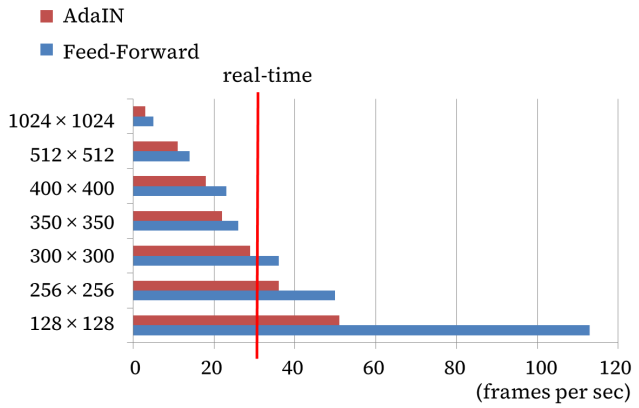

Fig. 17. Real-time style transfer performance based on input image sizes.

computing environment, we expect that the performance and also the size of the input texture would be improved with enhanced computing power machines.

VII. Conclusions and Future Works

Through this simulation, we examined the real-time style transfer performance in the game engine and efficient data processing methods. When interlocking a neural network that generates game resources in real-time with a game engine, the data transmission and reception method between the neural network and the game engine must be efficient. In addition, input and output data from neural networks must be fast and efficient when converted to game resources. In the paper, the Shader function of the game engine was used to efficiently convert and copy data through GPU parallel processing.

This paper shows a simulation of applying a deep neural network to textures on the game in real-time. Since the texture is used in various ways in the game, it will be possible to conduct various studies through future applications. It will also be necessary to study the performance optimization problems that may arise during game development using deep neural networks.

Additionally, research will be needed on performance optimization issues that may arise while developing games using neural network models. Just as the optimization techniques applied are different depending on the genre or characteristics of the game itself, optimizations based on the characteristics of the used neural network models should be investigated. Considerations for the mobile environment are also needed.

Acknowledgment

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-RS-2024-00437756) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

This study was supported by the BK21 FOUR project (AI-driven Convergence Software Education Research Program) funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Korea (4199990214394).

References

[1] J. Gregory, *Game Engine Architecture, Third Edition*. CRC Press, 2018.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[3] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[4] M. Wydmuch, M. Kempka, and W. Jaśkowski, "Vizdoom competitions: Playing doom from pixels," *IEEE Transactions on Games*, vol. 11, no. 3, pp. 248–259, 2019.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Proc. of the 27th International Conference on Neural Information Processing Systems - Volume 2*. Cambridge, MA, USA: MIT Press, 2014, pp. 2672–2680.

[7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[8] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, I. Simon, C. Hawthorne, A. M. Dai, M. D. Hoffman, M. Dinculescu, and D. Eck, "Music transformer," 2018. [Online]. Available: http://arxiv.org/abs/1809.04281

[9] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2414–2423.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[11] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 694–711.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015.

[13] X. Huang and S. Belongie, "Arbitrary style transfer in real-time with adaptive instance normalization," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1510–1519.

[14] H. Park and N. Baek, "Developing an open-source lightweight game engine with dnn support," *Electronics*, vol. 9, no. 9, 2020.

[15] A. Juliani, V. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *CoRR*, vol. abs/1809.02627, 2018.

[16] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde, *The C# Programming Language*, 3rd ed. Addison-Wesley Professional, 2008.

[17] G. V. Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: an imperative style, high-performance deep learning library*. Red Hook, NY, USA: Curran Associates Inc., 2019, no. 721.

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: a system for large-scale machine learning," in *Proc. of the 12th USENIX Conference on Operating Systems Design and Implementation*. USA: USENIX Association, 2016, pp. 265–283.

[20] Unity, *Unity Barracuda: a lightweight and cross-platform Neural Net inference library for Unity*, retrieved in July 2024. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.barracuda0.3/-manual/index.html

[21] Cr33zz, "Neuro_: C++ implementation of neural networks library with keras-like API," retrieved in July 2024. [Online]. Available: https://github.com/Cr33zz/Neuro_

[22] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014.

[23] ONNX, "Open Neural Network Exchange," retrieved in July 2024. [Online]. Available: https://onnx.ai/

[24] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. of the 27th International Conference on International Conference on Machine Learning*. Madison, WI, USA: Omnipress, 2010, p. 807–814.

[25] G. Liu, A. Dundar, K. J. Shih, T.-C. Wang, F. A. Reda, K. Sapra, Z. Yu, X. Yang, A. Tao, and B. Catanzaro, "Partial convolution for padding, inpainting, and image synthesis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 5, pp. 6096–6110, 2023.

[26] P. Baldi, "Autoencoders, unsupervised learning and deep architectures," in *Proc. of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27*. JMLR.org, 2011, p. 37–50.

[27] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *J. Mach. Learn. Res.*, vol. 11, pp. 3371–3408, Dec. 2010.

[28] V. S. Gordon and J. Clevenger, *Computer Graphics Programming in OpenGL with C++, Third Edition*. De Gruyter, 2024.