# Neural Style Transfer in Gameplay

## AIST4010 Final Report

**Name: CHOI, Jun Seok / SID: 1155100532**

## Abstract

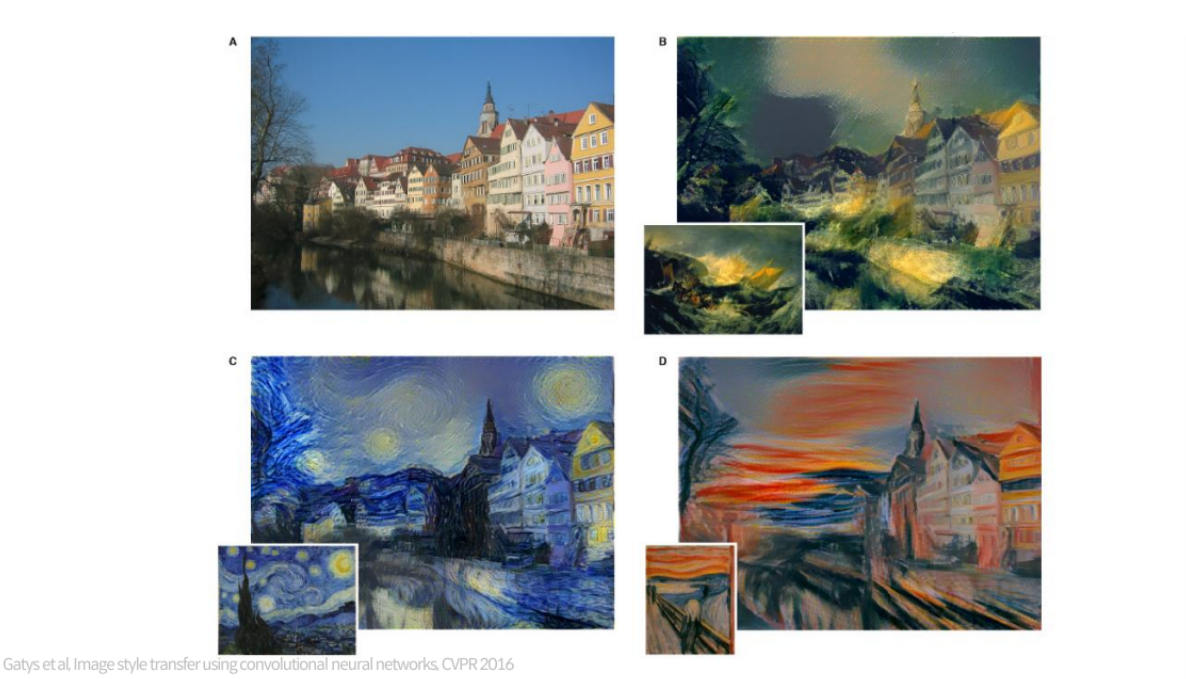
Neural Style Transfer is one of the interesting applications that can be implemented with neural network. Through this network, you can create your own art-like images which will diversify the user experience. Hence, it is interesting that utilizing style transfer to the gameplay, as it will give more interesting and immersive experience to the players. Utilizing style transfer in gameplay implies you need to set the style transfer algorithm, the neural network model, on your game development platform. Thankfully, Unity, one of the most popular game engine, provides the package called ‘Barracuda’ which enables utilizing the neural network model under Unity(C#) environment. I experimented how well style transfer can be adapted to the game scene on Unity, and experimented with various style transfer algorithms for higher quality and faster rendering.

Overall, I achieved to build the game application with following performances:

- High quality rendering with 14FPS
- General quality rendering with 30FPS
- Low quality rendering with 48FPS

## Intoduction

Neural Style Transfer is one of the interesting applications that can be implemented with neural network. Through this network, you can create your own works of art. To transform an image into a new style, it is to produce a synthesized image of a new style (Generated Image) with the original image (Content) and the style to be transformed.




Real-time style transfer enables us to plate several styles of streaming based contents, such as streaming video or gameplay. Let's imagine about gameplay as an example. In the case, you need to change the style of the game scene differently according to the player's choice. It can be achieved by stylizing on every rendering object on the scene, but it will consume the large amount of computational cost as it needs to calculate specific style patterns on the fragment shader, which is not preferable. If we can adapt real-time style transfer to the gameplay with a little cost, it will be a benefit to diversify the user experience.

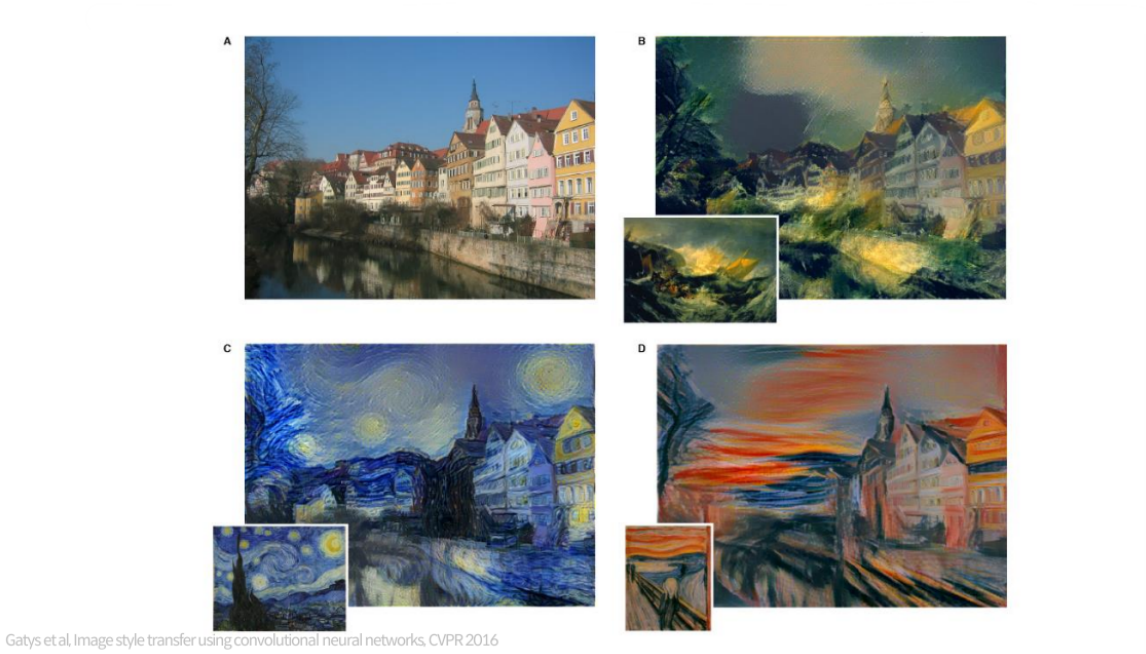
Utilizing style transfer in gameplay implies you need to set the style transfer algorithm, the neural network model, on your game development platform. Thankfully, Unity, one of the most popular game engine, provides the package called 'Barracuda' which enables utilizing the neural network model under Unity(C#) environment. I experimented how well style transfer is adapted to the game scene on Unity, and experimented with various style transfer algorithms for higher quality and faster rendering.

The experiment is conducted with 960x540 size (half of Full HD) images. The model from Perceptual Losses for Real-Time Style Transfer and Super-Resolution shows great quality of stylization, but without desirable fast (12~14FPS). The model with fewer parameters is relatively faster (47~48FPS) but the quality is much less than previous model. With further experiments, I could achieve increase the quality with reasonable frame rate(30FPS). As a result, therefore, the interactable real-time game application is implemented with 6 different styles.

# Related Work

## Research Papers

- ***Image Style Transfer Using Convolutional Neural Networks****:* The first paper that applied neural network for style transfer.



Gatys et al, Image style transfer using convolutional neural networks, CVPR 2016

  - **Features:** Using pretrained CNN model, extract each features from content image and style image respectively. Training the target image by calculating pixel-wise loss with content features and gram matrix loss with style features.
  - **Evaluation:** Slow and not reproducible. As it trains the image(not the network), the output image should be trained each time you want to adapt style, which means the model is not reproducible. Hence, not preferable on real-time style transfer.
  - **Link:** https://arxiv.org/abs/1508.06576

- ***Perceptual Losses for Real-Time Style Transfer and Super-Resolution:*** The first real time style transfer model with high quality
  - **Features:** To solve the retraining problem in the first paper, this paper replaces the per-pixel loss with perceptual loss to train a style image on the network and use the network as it is for faster generation.
  - **Evaluation:** Some limitation as the model needs to be retrained for different stylization, but still available for real-time style transfer.
  - **Link:** https://arxiv.org/abs/1603.08155

- ***Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization:*** The first real-time arbitrary-style transfer model.
  - **Features:** This model enables real-time style transfer by inferencing trained network. Also, it can extract style information from style image using adaptive instance normalization. Hence, it can produce arbitrarily stylized image without further training in real-time.

- **Evalutaion:** This is suited model for real-time style transfer application.
- **Link:** https://arxiv.org/abs/1703.06868

- ***Multi-style Generative Network for Real-time Transfer:*** Real-time and arbitrary-style transfer model with higher quality
  - **Features:** This paper proposed the model which learns to match the second order feature statistics with the target styles. It enables comprehensive style modeling, which is difficult to achieve original 1-dimensional style embedding. Also, by employing unsampled convolution, it enhanced the quality of the output image.
  - **Evalutaion:** This is suited model for real-time style transfer application.
  - **Link:** https://arxiv.org/abs/1703.06953

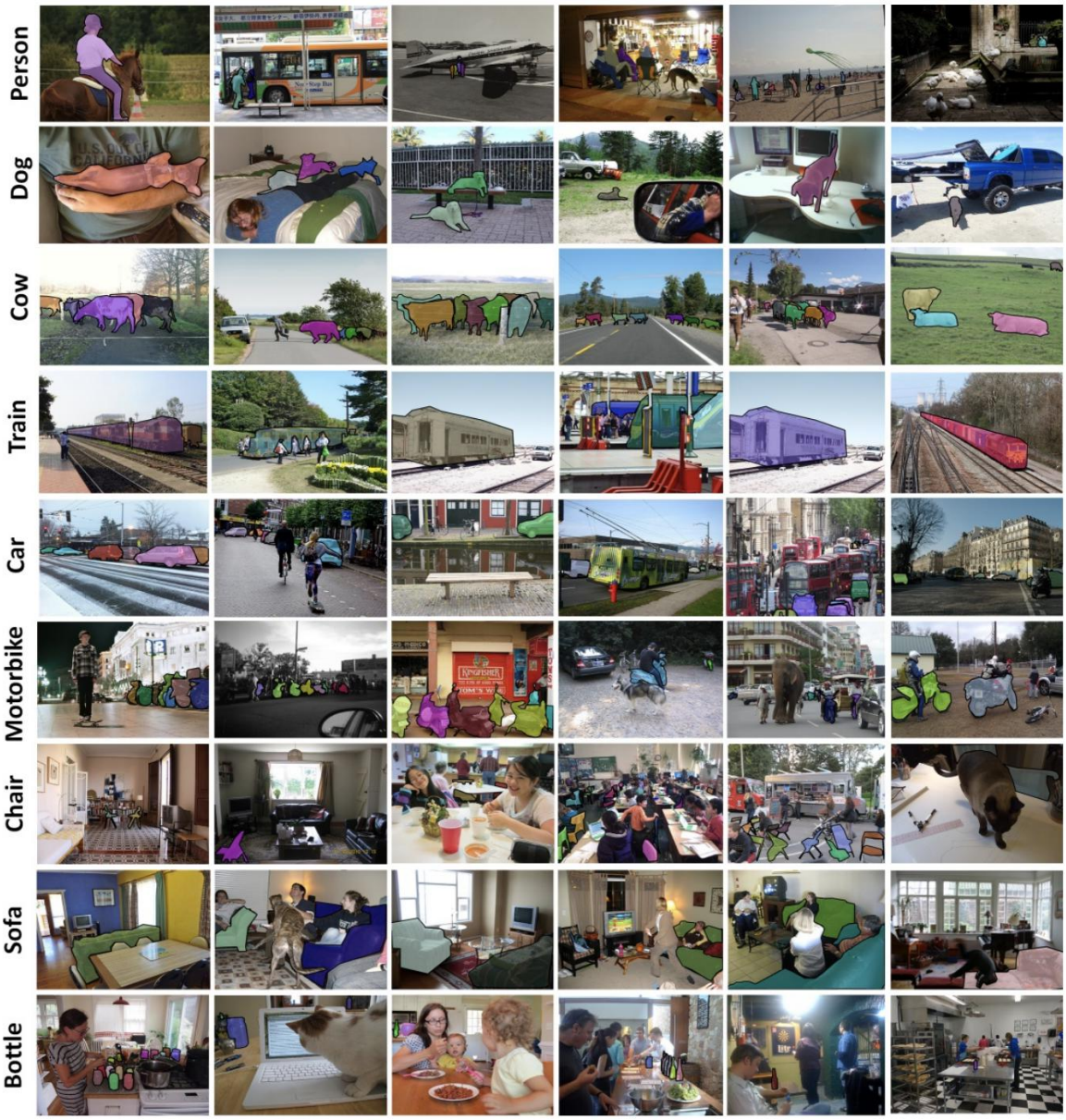
**Others**

- *Instance Normalization: The Missing Ingredient for Fast Stylization*
- *Photorealistic Style Transfer via Wavelet Transforms*
- *Real-time Localized Photorealistic Video Style Transfer*
- *A Style-Based Generator Architecture for Generative Adversarial Networks*
- *Adjustable Real-time Style Transfer*
- *Image-to-Image Translation with Conditional Adversarial Networks*
- *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*
- *(Automated) Deep Photo Style Transfer*

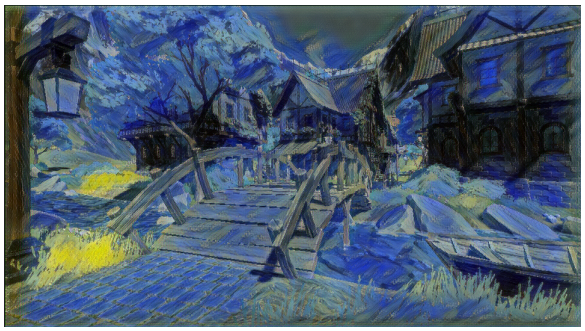
## Data

- **WikiArt:** Images of art
  - **Source:** https://github.com/cs-chan/ArtGAN/tree/master/WikiArt%20Dataset
  - **Size:** 737MB
  - **Composition: 42129 images** for training and **10628 images** for testing, painting from **195 different artists**.
  - **Purpose:** To be selectively used as style image and used to train AdaIN model.

- **COCO Train2014:** Images of various categories
  - **Source:** https://cocodataset.org/
  - **Size:** 13GB
  - **Composition: 83K** of images with various size
  - **Purpose:** To train Transformer part of style transfer model.



- **Others:** Style images used for the stylization



## Approach

As mentioned earlier, interactable stylized real-time rendering in a game with several style models under the Unity environment is the goal of this project. So, I will use the package ‘Barracuda’ and Unity for game rendering.

Followings are steps of utilizing style model on Unity

- **Step 1:** Implement and train the NN model
  - **Read papers:** Understand selected papers to earn basic knowledge required
    - *Image Style Transfer Using Convolutional Neural Networks*
    - *Instance Normalization: The Missing Ingredient for Fast Stylization*
    - *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*
    - *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization*
    - *Multi-style Generative Network for Real-time Transfer*
  - **Implementation:** Implement the important model for solid knowledge and that will be used for stylization in this project. Please see enclosed files for details.
    - *Image Style Transfer Using Convolutional Neural Networks*
    - *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*
    - *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization*
    - *Multi-style Generative Network for Real-time Transfer*

- **Step 2:** Import the trained model to Unity
  - **Pipeline:**
    - **ONNX:** Save model to ‘onnx’ format to utilize them under the Unity environment.

      **Exporting code**

      *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*

```
input = config.style_aug(load_image(config.content_img))
input = input.unsqueeze(0)

torch.onnx.export(stylizer.cpu(),
                  input,
                  f'{config.content_img}.onnx',
                  export_params=True,
                  opset_version=9,
                  do_constant_folding=True,
                  verbose=True)
```

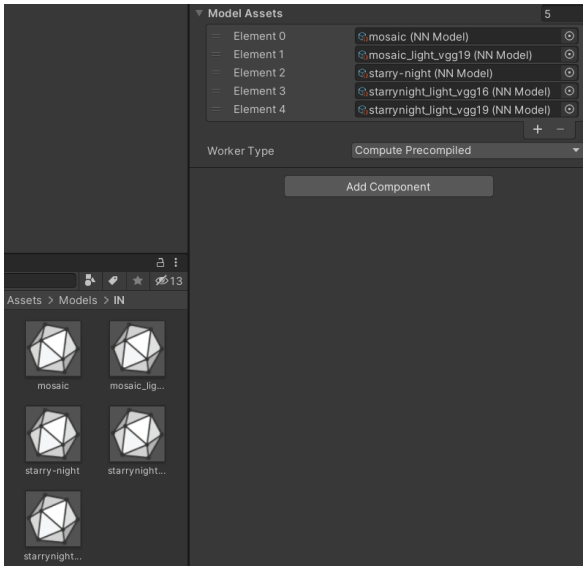
      Export Code: *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization*

```
content = config.augmentation(Image.open(config.content_img))
style = config.augmentation(Image.open(config.style_img))

content = content.unsqueeze(0)
style = style.unsqueeze(0)

vgg.cpu().eval()
torch.onnx.export(vgg,
                  content,
                  'encoder.onnx',
                  export_params=True,
                  opset_version=9,
                  do_constant_folding=True)
```

```
content = vgg(content)
style = vgg(style)
feature = AdaIN(content_f, style_f)

decoder.cpu().eval()
torch.onnx.export(decoder,
                  feature,
                  'decoder.onnx',
                  export_params=True,
                  opset_version=9,
                  do_constant_folding=True)
```

- **Barracuda(3.0.1):** Build pipeline to connect NN model and Unity.
  - **Model import:** Declare the list of NN models and simply input trained models to elements on the Unity editor.

```
public List<NNModel> modelAssets;
```




  - **Model initialization:** Load the imported model and create the worker which is used for inference.

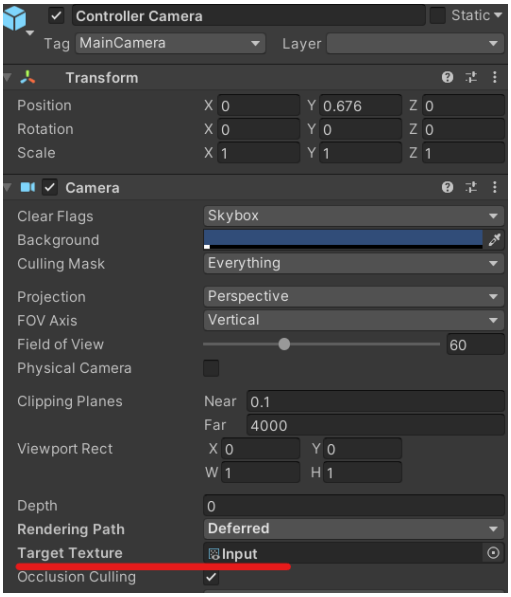
```
m_RuntimeModel = ModelLoader.Load(modelAssets[_index]);

_engine = WorkerFactory.CreateWorker(workerType, m_RuntimeModel);
```

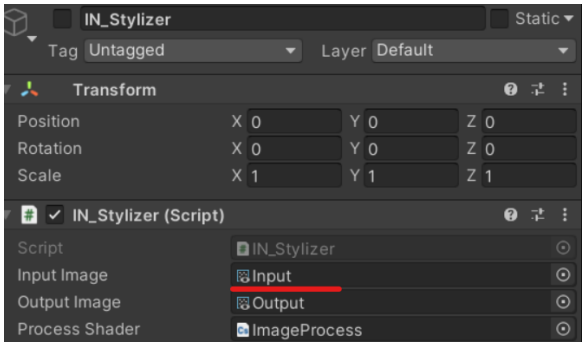
  - **Inference:** Inference the preprocessed image and returns stylized output.**Step 3:** Build the real-time stylized renderer on Unity engine.

```
Tensor input = new Tensor(tmpTex, channels: 3);

_engine.Execute(input);
Tensor output = _engine.PeekOutput();
input.Dispose();
```

- **Unity(2020.3.32f1):** Produce the stylized game scene in real-time.
  - **Game map:** Import ‘Suntail Village’ asset from the Asset Store

  - **Input image:** Inject current frame as an input image and preprocess (0~1 → 0~255) the image.



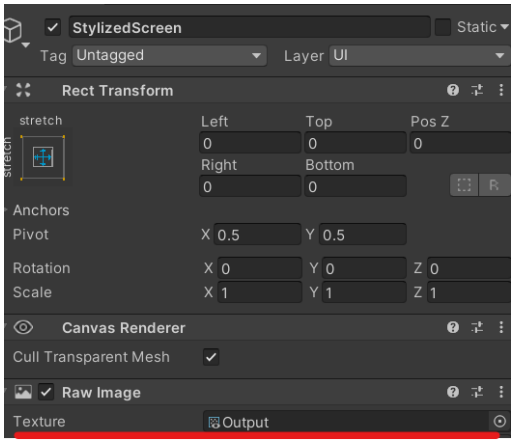

```
RenderTexture tmpTex = RenderTexture.GetTemporary(InputImage.width, InputImage.height
Graphics.Blit(InputImage, tmpTex); // Copy InputImage -> tmpTex
ProcessImage(ref tmpTex, "PreProcess");
```

  - **Inference:** Inference via Barracuda

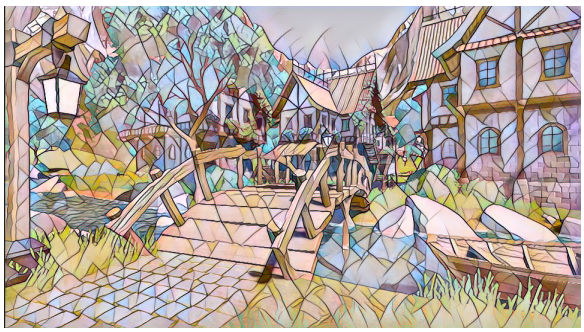
```
Tensor input = new Tensor(tmpTex, channels: 3);

_engine.Execute(input); // Inference
Tensor output = _engine.PeekOutput(); // Get result
input.Dispose();

RenderTexture.active = null;
output.ToRenderTexture(tmpTex); // Copy output -> tmpTex with casting
```

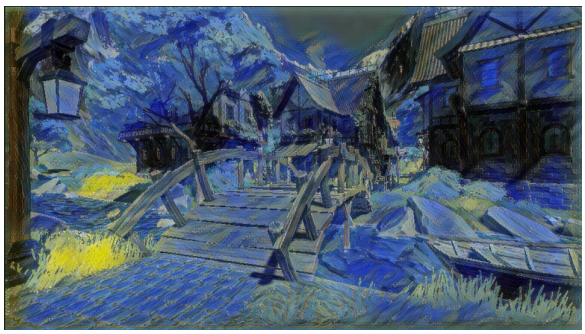
- **Output image:** Post process and set the image as the game scene

```
ProcessImage(ref tmpTex, "PostProcess");
Graphics.Blit(tmpTex, OutputImage); // Copy tmpTex -> OutputImage
```




- **Results:** 12~14FPS





## Experiments

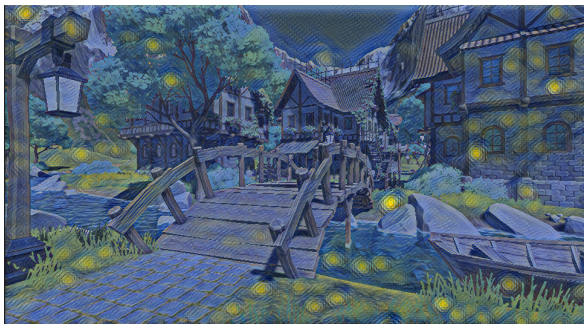
I conducted experiments to increase the frame rate. Since the low frame rate is not desirable for the game application, I set the lower limit of FPS as 24. Followings are the base setting of this experiment.

**Base**

- **Model:** *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*
  - **Feature Extractor:** VGG16(pre-trained)
  - **Transformer:** 3→32→64→128→5xResBlock(128)→128→64→32→3 (*the first 4 numbers denote the channel output of each up-convolution layer and the middle one denotes 5 residual blocks with 128 channels and the last 4 denote down-convolution channels*)
  - **Dataset:** COCO train 2014
- **Parameters:**
  - **Content weight:** 1e5 / **Style weight:** 1e10
  - **Epochs:** 2
  - **Barch size:** 4
- **Hardware:**
  - **CPU:** AMD Ryzen 9 5900HX with Radeon Graphics
  - **RAM:** 16GB
  - **GPU:** NVIDIA GeForce RTX 3060 / **VRAM:** 6GB

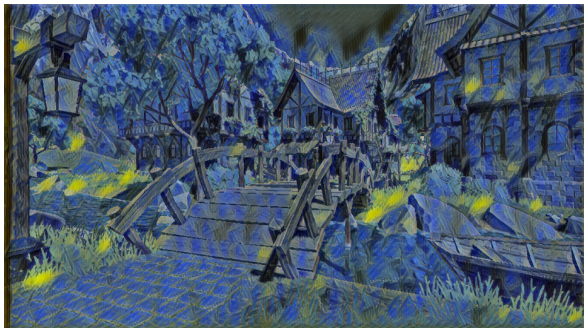
Firstly, I experimented with different structures of Transformer. Intuitively, the easy and simple way to reduce inference time is just decreasing the number of parameters. So, I started experiments with Transformer of fewer parameters.

- **3→8→16→32→5xResBlock(32)→32→16→8→3:** 47~48FPS



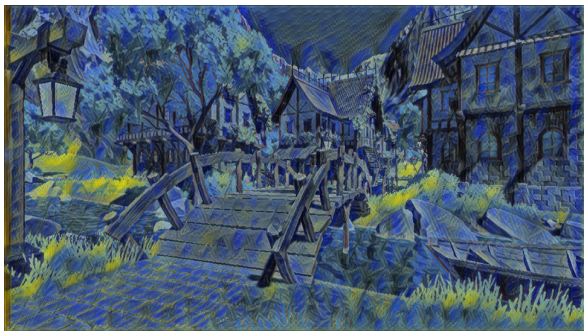
FPS is enough, but the image quality is not satisfying. I tried higher parameters structure.

- **3→16→32→64→5xResBlock(64)→64→32→16→3:** 23~24FPS

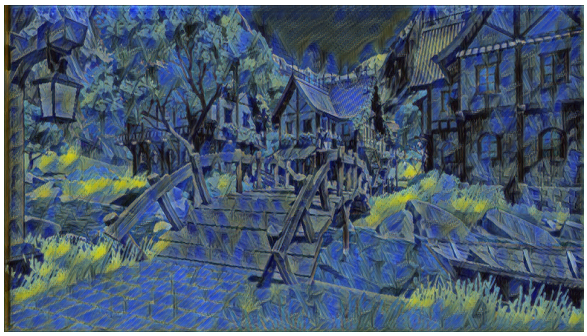


The image quality is much better, but the frame rate is not desirable. I experimented with several other structures. Especially, I concentrated on the number of residual blocks.

- **3→8→16→32→15xResBlock(32)→32→16→8→3:** 32~33FPS



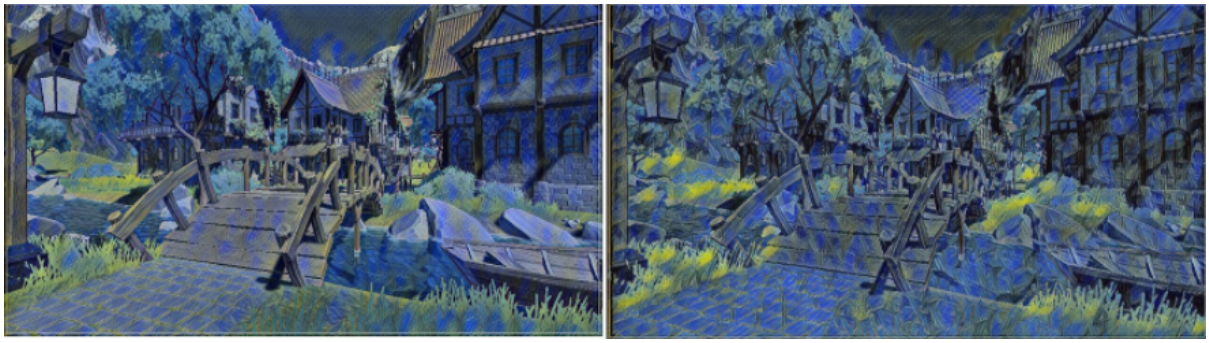
- **3→8→16→32→20xResBlock(32)→32→16→8→3:** 30~31FPS



Besides the structure of Transformer, I also experimented with different extractor and parameters(style weight).

- **Extractor:** VGG16 → VGG19

- **Parameter: Style weight** 1e10→5e10

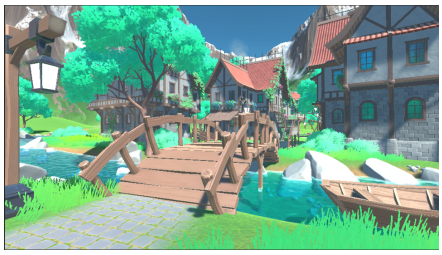


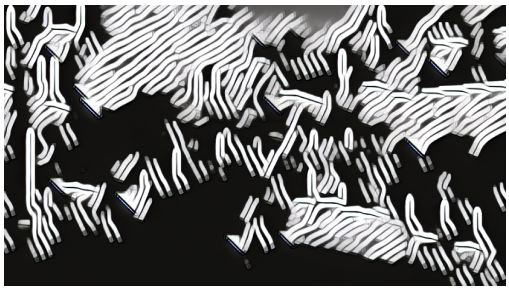
## Results

### Wrap up

- **Extractor:** Almost no influence on the quality
- **Transformer:** Significant influence on the quality. The comparison below is based on quality.
  - **128**→5xResBlock(**128**)→**128 > 64**→5xResBlock(**64**)→**64 > 32**→5xResBlock(**32**)→**32**
  - 32→**5xResBlock**(32)→32 **<** 32→**15xResBlock**(32)→32 **<** 32→**20xResBlock**(32)→32
  - 32→20xResBlock(32)→32 **≈** 64→20xResBlock(64)→64 **≈** 32→(>20)xResBlock(32)→32

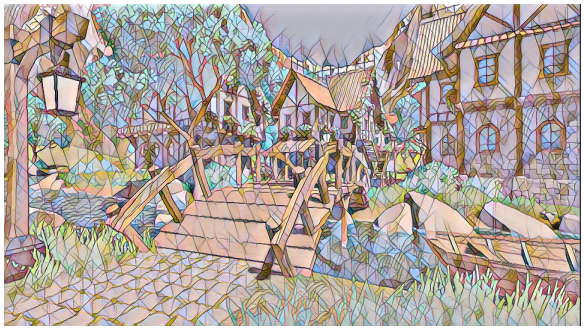
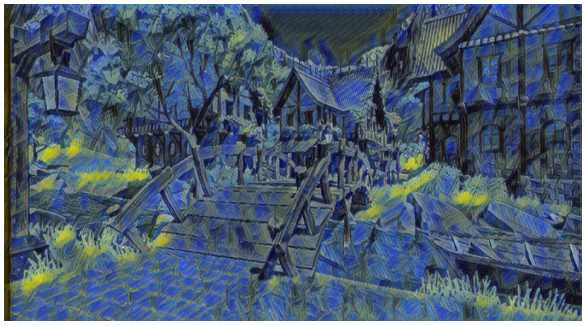
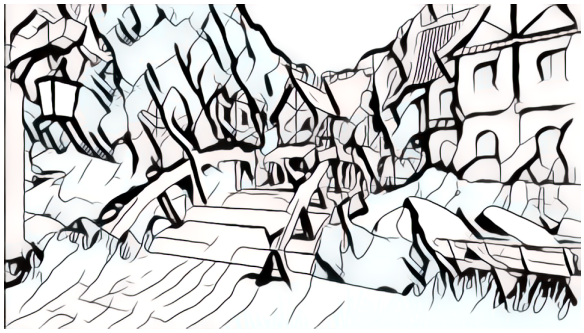
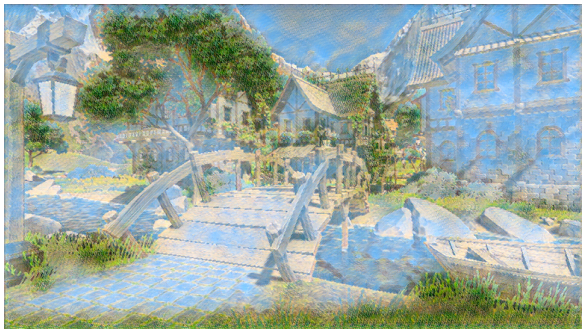
  As you can notice, more channels and more residual blocks give higher quality. However, if there are 20xResBlocks, there is not much difference between 32 and 64 channels. Also, more than 20xResBlocks does not show much different quality compared to 20xResBlocks. Therefore, I decided the Transformer structure as **3**→**8**→**16**→**32**→**20xResBlock(32)**→**32**→**16**→**8**→**3** for the final product.
- **Style weight:** Style weight can not be fixed since it depends on the style image. For example, following images show the failure case of stylization. The original content is not distictable in stylized image although the style weight is same as the base setting(1e10).

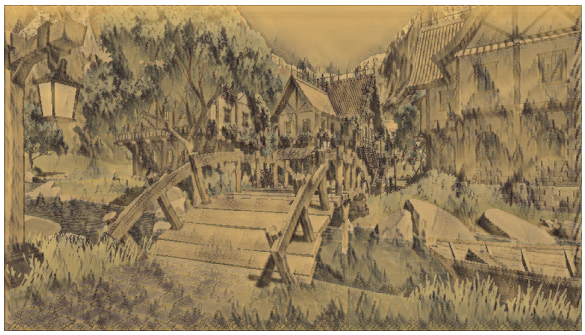




## Final product

Six different stylized map with 30~32



## Limitations

- **MSG-Net:** The model from *Multi-style Generative Network for Real-time Transfer* cannot be exported to '.onnx' format since it includes the aTen operation which is not yet supported by 'ONNX'. This shows that there are some limits in the variation of the model you can utilize via Barracuda.

  For more details about supporting aTen operation by ONNX**:**
  *https://github.com/onnx/onnx/blob/main/docs/Operators.md*

- **AdaIN:** To utilize the model from *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization*, you need to extract features from content and style image, respectively. Then, standardize and normalize the content feature with parameters of style feature. Lastly, decode the normalized feature to produce output. This process of work has some limits.

  The element-wise operation is not supported by Tensor type in Barracuda. Also, other Unity or NuGet packages which enable the element-wise operation such as NumSharp and Tensorflow.Net are not available since they conflict to Barracuda package as they contain same namespace. I tried to implement AdaIN operation in brute-force way, but it takes too long time to calculate which speed cannot be adapted in real-time.

```
for (int h = 0; h < feature.height; h++)
{
    for (int w = 0; w < feature.width; w++)
    {
        for (int c = 0; c < feature.channels; c++)
        {
            sum[0, 0, 0, c] += feature[0, h, w, c];
            sum2[0, 0, 0, c] += feature[0, h, w, c] * feature[0, h, w, c];
        }
    }
}
```

```
Tensor result = new Tensor(content.shape);
for (int h = 0; h < result.height; h++)
{
    for (int w = 0; w < result.width; w++)
    {
        for (int c = 0; c < result.channels; c++)
        {
            float normalize = (content[0, h, w, c] - c_params.Item1[0, 0, 0, c]) / c_params.Item2[0, 0, 0, c];
            result[0, h, w, c] = normalize * s_params.Item2[0, 0, 0, c] + s_params.Item1[0, 0, 0, c];
        }
    }
}
```

  Although there should be some methods to resolve this problem such as GPU programming, it should be not simple work and it is not within the range of the course. Therefore, I discarded this approach.

## Conclusion

I presented a style transfer application which is real-time and interactive. The application shows that the neural style transfer algorithm can be applied to the gameplay. It can render 12~14 frames per second. If you compromise some extents of quality or resolution, you can render about 50 frames per second.

Through this project, I have analyzed and implemented several neural style transfer models myself. I could gain the solid knowledge about the following style transfer algorithm and how those models work.

*Image Style Transfer Using Convolutional Neural Networks*

*Perceptual Losses for Real-Time Style Transfer and Super-Resolution*

*Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization*

As a result of this project, I could build an game application with stylization. Also, to increase the FPS, I conducted experiments.

Overall, I achieved to build the game application with following performances:

- High quality rendering with 14FPS
- General quality rendering with 30FPS
- Low quality rendering with 48FPS

While I am trying to adapt the model on Unity, I have learned about Barracuda package and some Unity functions such as Texture, Shader etc. Although I focused on style transfer models in this project, this process of work can be applied to the other neural network models such as object detection, super resolution etc. Hence, several kinds of models can be utilized on the Unity environment, which would be a benefit to build various real-world applications. It would be great if this project can be helpful for those who are interested in developing game application with forefront neural networks research.

Despite the successful game application with neural network models, it is not suited for current games, especially for AAA titled games. As graphics rendering is the most important key in the game industry and modern games are mostly supports higher than 120FPS. Hence, further optimization process should be investigated such as GPU programming, using depth buffer etc. Of course, optimizing inference time is a must. In addition, researching to reduce flickering effect should be conducted for higher quality.

Overall, in fact, there are still many areas to be developed in order to apply these neural network models to actual games. However, as you can see from the process of this project, many game companies are preparing games based on the neural network model(Barracuda is officially supported by Unity!). Based on these things, I look forward to seeing more novel and innovative games in the future.

## References


- *Image Style Transfer Using Convolutional Neural Networks*
- *Instance Normalization: The Missing Ingredient for Fast Stylization*
- *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*
- *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*
- *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization*
- https://github.com/pytorch/examples/tree/36441a83b6595524a538e342594ee6482754f374/fast_neural_style
- https://github.com/naoto0804/pytorch-AdaIN
- https://github.com/Unity-Technologies/barracuda-release
- https://onnx.ai/
- https://github.com/onnx/onnx/blob/main/docs/Operators.md
- https://assetstore.unity.com/packages/3d/environments/fantasy/suntail-stylized-fantasy-village-203303
- https://medium.com/element-ai-research-lab/stabilizing-neural-style-transfer-for-video-62675e203e42
- https://blog.unity.com/technology/real-time-style-transfer-in-unity-using-deep-neural-networks