Master Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Real-Time Rendering Super Resolution with Unreal Engine 5

Tobias Brandner
Department of Computer Science
Chair of Computer Science IV (Computer Vision)

Prof. Radu Timofte
First Reviewer

Dr. Nancy Metha
First Advisor

M.Sc. Eduard Zamfir
Second Advisor

Submission
26. August 2024

Abstract

Playing video games is a widespread hobby that has long since become mainstream. To create a pleasant visual experience for the players, game engines, like Unreal Engine, must generate the images of a certain quality in real-time. But as players demand higher frame rates and playing in 4k gets more widespread, modern hardware struggles to keep up. To solve this problem, several methods have been developed that offer super-resolution in real time. Real-time rendering super resolution describes the process of increasing the resolution and improving the image quality of real-time rendered content before displaying it to a user. In contrast to super resolution tasks targeted at images or videos, real-time rendered content suffers from additional artifacts in form of aliasing, dithering and detail loss. On top of the rendered frame, real-time rendering also provides access to auxiliary buffer data called G-buffers. A series of research studies has been published, that tackle this issue using neural methods. Despite achieving good results, they do not operate within the limit of 4k gaming. In this work, we investigate the restrictive use case of increasing the resolution of rendered content from 1080p to 4k while improving its quality, 60 times per second. For this, we created our own dataset within Unreal Engine 5, containing stylized environments with fully animated characters. We call it Unreal Stylized Motion Matching, or USMM. In addition, we provide a custom plugin to get G-buffer information, as well as different resolutions from any Unreal Engine project. We trained a neural network called Unreal Real-Time Super Resolution, or URTSR, with our dataset to learn to mitigate the problems of real-time rendered content. Our URTSR network is capable of increasing the resolution to 4k in a real gaming context, while also achieving respectable image quality. Yet, our method is not capable to utilize all the provided G-buffer information effectively. We believe further investigations in regard to the additional information could improve the effectiveness of neural methods for our restrictive use case.

Zusammenfassung

Das Spielen von Videospielen ist ein weit verbreitetes Hobby, das längst zum Mainstream geworden ist. Um den Spielern ein angenehmes visuelles Erlebnis zu bieten, müssen Game Engines wie die Unreal Engine die Bilder in einer bestimmten Qualität in Echtzeit erzeugen. Da Gamer jedoch immer höhere Bildwiederholraten verlangen und das Spielen in 4k sich immer weiter verbreitet, hat moderne Hardware Schwierigkeiten, damit Schritt zu halten. Um dieses Problem zu lösen, wurden mehrere Methoden entwickelt, die eine höhere Auflösung in Echtzeit ermöglichen. Real-time rendering super resolution erhöht die Auflösung von gerenderten Inhalten und deren Qualität bevor sie Nutzern angezeigt werden. Im Gegensatz zu Bildern oder Videos die für super resolution Aufgaben genutzt werden, haben Echtzeit gerenderte Inhalte zusätzlich herausfordernde Artefakte. Diese Artefakte kommen vom aliasing Effekt, dem dithering Effekt und grundsätzlichen Detail Verlust. Dafür hat real-time rendering den Vorteil zusätzliche Buffer Informationen, die G-buffer, auf der Grafikkarte nutzen zu können. In dieser Arbeit fokussieren wir uns auf die Herausforderung die Auflösung von 1080p auf 4k zu erhöhen, die Bild Qualität dabei zu verbessern und das 60 Mal die Sekunde. Dafür haben wir einen eigenen Datensatz mit gerenderten Inhalten aus Unreal Engine 5 erstellt, das wir Unreal Stylized Motion Matching nennen, oder in kurz Form USMM. Um den Datensatz zwischen unterschiedlichen Projekten in Unreal Engine einfacher zu generieren, haben wir ein Plugin geschrieben, das es erlaubt die Daten, wie die G-buffer und die gerenderten Inhalte in verschiedenen Auflösungen aus der Engine zu ziehen. Wir haben eine eigene neuronale Methode entwickelt namens Unreal Real-Time Super Resolution, oder URTSR, um sie auf unseren gesammelten Daten zu trainieren. Die Methode soll dabei die Problemstellung erlernen um die beschriebenen Artefakte zu minimieren. Unsere URTSR Methode kann die Auflösung in einem echten Gaming bezogenen Anwendungsfall auf 4k erhöhen und dabei die Bild Qualität merklich verbessern. Leider kann unsere entwickelte Methode nicht alle Informationen aus den verfügbaren G-buffern effektiv nutzen. Weitere Erkenntnisse sollten in dieser Richtung gesammelt werden, da wir glauben die zusätzliche Information könnte neuronale Methoden für unsere Problemstellung noch weiter verbessern.

Contents

1 Introduction

As games become more computationally intensive and users demand higher resolutions and frame rates, the usage of super resolution (SR) increases. SR aims to upscale degraded low-resolution (LR) images to detail-rich high-resolution (HR) images [1]. In the context of gaming, this means the image, or, to be more precise, the frame, is internally rendered at a lower resolution, such as 1920x1080 (1080p). An upsampling algorithm is applied to this frame, to achieve a higher display resolution, of e.g. 3840x2160 (4k), which is then displayed to the user. To lower the computational load of the graphics processing unit (GPU) the upscaling needs to happen in real-time, since games typically run from 30 frames per second (fps) or 60 fps up to 240 fps. This means the time available for rendering and upsampling the frame is around 33.3 ms or 16.6 ms down to 4.1 ms, decreasing the time window for the super resolution operation even further. We call this sub-category real-time rendering super resolution (RRSR).

Deep Learning Super Sampling (DLSS) [2], Fidelity Super Resolution (FSR) [3] and Xe Super Sampling (XeSS) [4] are widespread RRSR methods from well-known companies like Nvidia[1], AMD[2] and Intel[3]. They are used in many game titles, such as Alan Wake 2 [5] or The Legend of Zelda Tears of the Kingdom [6], which work on multiple platforms, such as Xbox or Nintendo Switch, and are implemented in game engines, e.g. Unreal Engine[4], Unity[5] and Godot[6]. Allegedly, Sony Interactive Entertainment is working on PlayStation Spectral Super Resolution, a RRSR method explicitly designed for the Playstation 5 [7]. RRSR is even used for games, which do not support any upscaling method themselves, with driver based solutions like the AMD Radeon Super Resolution [8] or NVIDIA Image Scaling [9]. These read the frame directly from the buffer on the GPU. On PC, Lossless Scaling [10] gained popularity as a third party software used for upscaling any game. Recently, Microsoft released a solution based on a similar idea with Auto SR [11], an automatic super resolution method. It is integrated into the operating system and automatically activates upscaling when the game is started. Therefore, we think RRSR is and continues to be an integral part of game development and gaming as a whole.

While there are different degradation models for different aspects in super-resolution tasks,

[1] `https://www.nvidia.com`
[2] `https://www.amd.com`
[3] `https://www.intel.com`
[4] `https://www.unrealengine.com`
[5] `https://unity.com`
[6] `https://godotengine.org`

e.g. different blur kernels like bicubic downsampling, rendered content suffers from unique degradation artifacts in form of (a) aliasing, (b) dithering, especially in regard to shadows, and (c) detail loss. In contrast to single image super resolution (SISR) methods, which only have the spatial information from the input image to perform super resolution and video super resolution (VSR), which in addition has temporal information from previous frames to perform super resolution, RRSR methods have extra information in form of auxiliary buffers, which are taken from the rendering pipeline to output the final frame. RRSR methods need to mitigate these artifacts, while using the available information smartly.

Despite similar projects doing an excellent job in preserving detail and mitigating artifacts during RRSR performance, they focus on upsampling with a scale of 2, leading to 960x540 (540p), and 4, resulting in 480x270 (270p) to 1080p [12, 13, 14, 15]. Other work upsamples to 4k, but with scaling factors of 4 and 8 [16]. To the best of our knowledge, the use case of upsampling to 4k in a highly time-sensitive manner, like 60 times per second, seems underrepresented in literature. Addressing this use case, we created a 4k RRSR dataset containing 16800 (LR, HR) frame pairs called **USMM**. USMM includes anti-aliased 4k ground truth HR frames and 540p and 1080p aliased LR frames. The LR frames include auxiliary buffers, like base color, depth and motion vectors. In addition, we provide a small plugin containing the configurations to create (LR, HR) frame pairs and shaders necessary to retrieve the auxiliary buffers in any Unreal Engine 5 project. Focusing on 4k upsampling from 1080p, limits us to design an even more efficient network called **URTSR**. URTSR utilizes a shallow U-Net architecture [17] with an attention block [18] in the bottom layer to achieve an inference time of $<$16 ms.

To summarize the main contributions of this thesis are as follows:

- Generating a 4k RRSR dataset, called **USMM**, containing 16800 (LR, HR) frame pairs, including 4k anti-aliased HR frames and 540p and 1080p aliased LR frames.
- Providing a custom plugin for generating (LR, HR) frame pairs with auxiliary buffers in any Unreal Engine 5 project.
- Designing our **URTSR** network for RRSR capable of upsampling from 1080p to 4k in a 60 fps setting.

In the following we explain the background of RRSR and the problems associated with it in chapter 2, how related work tried to mitigate these problems in chapter 3, how we generated our own dataset for this restrictive use case and how we designed our method to handle that in a highly efficient manner in chapter 4. Further, we evaluate our proposed methods against other traditional and related work in chapter 5 and discuss the results, limitations we faced and ideas we would like to pursue in the future in chapter 6. Finally, we conclude our contributions in chapter 7.

2 Background

In this section we want to go over some basic principles of real-time rendering and the unique artifacts associated with it, as well as methods to mitigate them. Further, we explain traditional upsampling approaches and the objective metrics we used for comparing image quality between all of our tested methods.

2.1 Real-Time Rendering

Real-time rendering describes the process of generating images based on virtual scenes in a short amount of time. In the case of three-dimensional (3D) virtual scenes, real-time rendering is one of the most complex and computationally intensive areas of interactive computer graphics [19]. A render program, such as Unreal Engine, needs to display a frame at a rate of at least 30 or better 60 times per second, to ensure a smooth visual and responsive experience for the user. 3D virtual scenes consist of multiple 3D objects, scattered across a simulated environment. A 3D model consists of two main parts, namely geometry and material. The geometry of a 3D model is defined by points in space (vectors) that form a grid in their own local coordinate system. The material consists of multiple textures, which are used by the renderer to determine the visual appearance of the object in the scene. Among them are color and the degree to which the surface reflects light. While a program like Blender[1] is used to model and animate 3D objects and render them into an image or video offline, a game engine like Unreal needs to render hundreds or hundreds of thousands of 3D objects in a global coordinate system, while simulating them in a more or less realistic way. These simulations include lighting and physics for example to create virtual interactive environments. Traditionally, objects in the scene are rendered one at a time, applying all necessary shading and light calculations to each one individually. This approach is called Forward Rendering and is pretty straightforward, as the name implies. While it is easy to implement, it is quite inefficient in rendering thousands of objects. Unreal Engine 5 uses Deferred Rendering, a more complex approach that separates rendering the geometry and applying the shading in different passes. First the geometry of the scene is rendered into different intermediate buffers called geometry buffer (G-buffer). These buffers contain information regarding various aspects of the scene such as:

- base color: the color of each object in the scene

[1] `https://www.blender.org/`

- depth map: distance of each object to the camera position
- roughness: roughness of the objects in the scene
- metallic: how metallic the objects in the scene are
- normal orientation vector (NoV): orientation of the normal vectors of the surfaces of each object in the scene in relation to the camera
- motion vectors: how every pixel in the scene moved from the previously rendered frame to the latest
- world normal: the normal vectors of each object related to world coordinates
- world position: the world coordinates of each pixel in the frame.

In the second pass, the lighting pass, the information from the G-Buffer is used and combined to form the final, rendered frame. This approach allows a more efficient calculation for shading objects and allows for complex effects such as global illumination, but increases the memory usage of the GPU [20]. Since G-buffers are stored in the video random-access memory (VRAM) it is possible for RRSR methods to use this information for upsampling.

2.2 Artifacts

Nowadays, rendered images can be indistinguishable from real photographs given enough time and using the right methods. In a real-time scenario, artifacts can occur in the image at certain locations and in certain situations, which are difficult for the renderer to handle. We want to point out the three most prominent problems in regard to rendering low resolution frames within Unreal Engine 5.

Figure 2.1: An example frame showcasing the three artifacts: (a) aliasing, (b) dithering and (c) detail loss. For the purpose of showcasing the artifacts, we choose one of our 540p validation frames.

2.2.1 Aliasing

Aliasing is most commonly observed around the edges of objects. It appears as a pattern of jagged or stair-step lines, when geometry or patterns are too fine or detailed to be represented at the given resolution. An example can be seen at the lantern's metal housing outside the lamp, see figure 2.1 green box.

2.2.2 Dithering

Dithering is used as a technique to smooth shading or transparencies by using a pattern of pixels to approximate the correct color values. Dithering artifacts appear as speckled noise, especially around the edges of shadows, and happen while rendering with low resolution shadow maps, or textures, and no anti-aliasing method to compensate. An example can be seen in the shadow of the person, see figure 2.1 blue box.

2.2.3 Detail Loss

Detail Loss is a severe case of the aliasing effect. This means when there are geometry or texture details to fine to display at the given resolution, the details are simply lost, due to the aliasing effect. An example can be seen at the plant's stem connecting to the leaf, see figure 2.1 purple box.

2.3 Anti-Aliasing

To counteract the aliasing effect, multiple different methods have been developed over the years. We want to explain the most common methods, which are also implemented in Unreal Engine 5.

2.3.1 FXAA

Fast aproximate anti-aliasing (FXAA) is the most computational efficient anti-aliasing technique of the following methods. In contrast to the other methods it is applied after the frame has been rendered, as a post-processing effect. It tries to detect high contrast edges and blurs them together. Based on the thresholds chosen for detecting the edges, FXAA struggles with over blurring or even fails at removing the artifacts at all. While the quality of the frame is lacking, the minimal performance impact still makes it a solid choice in modern games [21].

2.3.2 SSAA

Super sampling anti-aliasing (SSAA) is the sledgehammer approach for counteracting the aliasing effect. SSAA uses multiple samples per pixel, or super samples the pixel, for blending them together and mitigate aliasing. To get the samples the frame is rendered in a higher resolution and the multiple values get averaged, e.g. with a box filter, to the display resolution [19]. While this approach reduces aliasing the most and does not introduce any other artifacts to the frame, it has a tremendous impact on the performance. This makes it almost unsuitable for real-time rendering. We chose this anti-aliasing method for our HR frames. More detail to as why is explained in 4.1.

2.3.3 MSAA

Multi sample anti-aliasing (MSAA) is similiar to SSAA, as it uses multiple samples, but is faster than a pure super sampling scheme. For MSAA the fragment of the pixel is shaded only once and shared between all samples. A fragment is all the information necessary to determine the pixel's final value. It includes, the base color, depth information, texture coordinates and normal vectors. Typical sample counts are 2, 4 and 8 [19]. MSAA is a balanced method with regard to the performance cost to frame quality ratio. Currently, MSAA is only implemented using the Forward Rendering approach in Unreal Engine 5.4 and cannot be used with the Deferred Render approach we used to generate our auxiliary buffer data.

2.3.4 TAA

Temporal anti-aliasing (TAA) is the default method for most modern game engines to handle aliasing. In contrast to SSAA and MSAA it uses multiple previously rendered frames to perform anti-aliasing. We refer to this as the accumulated temporal history. Motion vectors are used to warp the previously rendered frames to the point in time of the current frame. A small offset is applied to the warped frames, jittering them, and the pattern of how the offset is chosen is called jitter sequence. The jitter sequence typically has a non-linear behavior, e.g. utilizing the Halton sequence, to handle edge cases where aliasing would still appear. The jitter sequence is then applied before blending the samples together [19]. The most prominent artifact with this anti-aliasing method are ghosting and flickering. Ghosting describes the appearance of parts of the object from the previous frame around the edges of the object in the current frame at spots where anti-aliasing should have been applied. This temporal noise appears, when pixels moved too much in the accumulated temporal history. It can be reduced by removing these pixels from the blending process. Flickering appears in the form of several sudden shifts of pixel brightness in the same location over multiple frames. Increasing motion vector accuracy and trying different jittering sequences can increase temporal stability and reduce flickering. Since TAA uses already available information, it has an excellent performance cost to frame quality ratio [22].

2.4 Interpolation

Upsampling, traditionally, has been performed using interpolation. Interpolation is the method of constructing new data points within the range of a discrete set of known data points. In a broad sense this means we need to "guess" the color information of the newly created pixels. Interpolation can be categorized into linear and non-linear methods.

2.4.1 Linear Interpolation

Nearest neighbor interpolation is one of the simplest and fastest interpolation algorithms. The idea is that every generated pixel gets the same value as its nearest neighbor pixel. The neighbor is identified by interpolating the pixel's coordinates using the scale of the LR to HR dimensions. While this approach leads to a higher resolution, it does not handle any of the aforementioned artifacts. Bilinear interpolation uses the 4 nearest neighbors and averages their color values based on their distance to the targeted pixel (values of closer pixel are weighted higher). Bicubic interpolation takes a grid of 4x4 neighboring pixels and averages their values based on the same heuristic. It is more computationally intensive and leads to a finer transition between pixel values while preserving details in textures. Bilinear and bicubic are common interpolation methods and even implemented and used in game engines, such as Godot [23]. Both approaches can help in mitigating aliasing and dithering artifacts in a frame.

2.4.2 Non-linear Interpolation

Linear interpolation methods work well for upsampling continuous areas, but can fail at edges by making them too blurry. For such cases, non-linear interpolation methods, like unsharp masking can be used. Unsharp masking creates a blurred version and subtracts it from of the original image, creating a mask that contains only the edges in the process [24]. In the context of games, FSR for example uses a combination of linear and non-linear interpolation methods to perform RRSR.

2.5 Metrics

What makes images look "good" or visually pleasing is actually quite hard to define, as the human visual perception system is complex and people have different opinions, based on their personal preferences. A general approach for defining an objective score is to ask for many subjective opinions and average their scores. This approach is called mean opinion score (MOS) [25] and is applicable for many quality tasks, but is very time-consuming and hard to justify to use to base comparisons on. Therefore, pure objective approaches have established themselves, as they are fast to calculate and easy to compare against each other. We will talk about the most common objective metrics in the context of super-resolution.

2.5.1 PSNR

Peak signal-to-noise ratio (PSNR) expresses the error between the original and upsampled image in decibels (dB), on a logarithmic scale. The error is computed based on the mean squarred error (MSE). MSE sums up the squared difference between the predicted value and the actual value and averages it. Let I be the original image and K be the upsampled image with the same dimensions m and n, MSE can be expressed through the formula 2.1.

$$\text{MSE} = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2 \tag{2.1}$$

PSNR divides the squared maximum value in the original image with the MSE value, calculated with regard to the upsampled image, and puts it on a logarithmic scale, see formula 2.2.

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{\text{MAX}_I^2}{\text{MSE}} \right) \tag{2.2}$$

The higher the PSNR value, the closer are the absolute pixel values of the compared images to each other [26].

2.5.2 SSIM

While PSNR seems very appealing, because of its easy to understand mathematical concept, it fails in many cases to represent image quality. A common example is to apply a slight blur to the image. The PSNR of a blurred image still is very high, yet the quality of the image is severely lacking. Structural similarity index measurement (SSIM) measures the difference between the original and upsampled image by changes in luminance, contrast and structure. It is expressed through formula 2.3.

$$\text{SSIM}(x,y) = [l(x,y)]^\alpha \cdot [c(x,y)]^\beta \cdot [s(x,y)]^\gamma \tag{2.3}$$

Where α, β and γ influence the weight of the individual changes, but are typically set to 1. Further luminance is described by formula 2.4, contrast by formula 2.5 and structure by formula 2.6.

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \tag{2.4}$$

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \tag{2.5}$$

$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \tag{2.6}$$

Where:

- μ_x and μ_y are the mean values of image x and y,
- σ_x and σ_y are the standard deviations of image x and y,
- σ_{xy} is the covariance of image x and y,
- C_1, C_2, C_3 are small constants to avoid instability when the values get too close to zero.

While SSIM considers luminance, contrast and structure differences, it offers a more accurate measurement for perceived image quality based on the human visual perception system [27].

2.5.3 LPIPS

There remain cases where both PSNR and SSIM values still fail to correlate to good image quality. Learned perceptual image patch similarity (LPIPS) [28] follows a modern trend, in which machine learning based approaches of identifying perceived image quality are unreasonably effective. Instead of doing a pixel-wise comparison like PSNR and SSIM, LPIPS uses a pretrained network's deep feature layers of the upsampled and the original image and calculates the euclidean distance between those. LPIPS includes three different network architectures in VGG, AlexNet and SqueezeNet. These architectures are trained on a large-scale perceptual similarity dataset, containing human judgement. Basically, LPIPS embeds a mean human visual perception into a metric score and has shown its effectiveness in many visual tasks, in particular super-resolution. For calculating our LPIPS scores, we used the AlexNet preset.

3 Related Work

Instead of using handwritten interpolation algorithms to upsample an image, we can try to solve the problem of super resolution by learning it relying on machine learning and using neural networks. Machine learning describes the idea of learning to solve a problem. A neural network trains on given examples and tries to find a solution for the task by generalizing from them. In a broad sense the machine learning approach tries to find and fit a function to a specific use case. While the function definition is described by the network architecture, the functions variables are adjusted through the training process.

In context of super-resolution convolution neural network (CNN) architectures are commonly used. Convolution in machine learning, describes a basic mathematical operation, in which a kernel or filter slides over an input image to produce a feature map. The image is defined by a 3D matrix with the dimensions of its width, height and RGB layers, while the kernel is a smaller 2D matrix, e.g. 3x3 or 5x5, with entries called weights. The kernel is moved from the top-left to the bottom-right corner. At each step, the overlapping values of the image and the kernel are multiplied and summed up together to produce a single output. This value is stored in the feature map corresponding to the kernel, at the same location. Sliding over the whole image results in a complete feature map. There are three more variables which influence the convolution operation:

- Stride describes the step size of the kernel, in our context how many pixels we move after each operation.
- Padding describes adding additional numbers, such as zero, around the border of the input image, for example to maintain the same spatial dimensions.
- Dilation describes how many pixels apart the values of the kernel should be applied on the image. This is typically set to 0 as we want to learn contiguous features.

CNNs consist of multiple convolutional blocks with the goal to produce a matrix in the desired dimension, so that our image is upsampled. A convolutional block describes how the convolution operation should be applied and how many feature maps we want to create. The amount of feature maps can be adjusted, while keeping the network's architecture the same. Variables with the property to be adjustable while not changing the network architecture are called hyperparameters. As convolution operations can only model linear relationships, we combine them with activation functions. Activation functions remap the input non-linearly, enabling the network to model more complex relationships. After the network has performed the mathematical operations on the input image, the output

is compared with a ground truth, in our case the 4K image with anti-aliasing. This comparison relies on various loss functions, which in turn use objective metrics such as PSNR. The calculated loss is then propagated backwards through the network, adjusting the weights and biases of each layer to train the network. This process is controlled by an optimizer, which minimizes the loss and thus improves the prediction of the model. The learning rate determines how much the weights of the network are adjusted. Optionally, a scheduler can be used to adjust the learning rate during training so that the network converges faster and improves its stability. Finding the right network architecture and combining it with the right hyperparameters is crucial for performing super-resolution [29].

3.1 Performing in Real-Time

Achieving good image quality in real-time while maintaining low memory consumption and high inference speed is key in performing highly efficient SR. Traditionally, CNNs performed the upsampling through interpolation methods, such as bilinear, as the first step in their network architecture. Therefore, all convolution operations had to be performed in the HR space. A rather inefficient approach, as working in the higher-dimensional space adds computational complexity and memory consumption. In 2016 the concept of pixel shuffling, or sub-pixel convolution, was introduced. Pixel shuffling enables CNNs to perform all of its operations in the LR space and postpone the upsampling operation to the last step in the network, see figure 3.1.

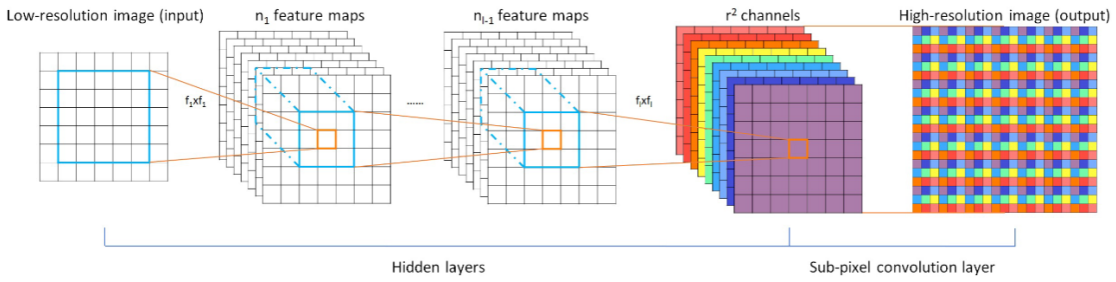

Figure 3.1: Pixel shuffling or sub-pixel convolution performs upsampling by rearranging the learned feature maps from the convolution operatio in low-resolution space into the high-resolution space [30].

The last layer of the network, the pixel shuffling layer, rearranges or shuffles the feature maps learned in LR space into the HR space. The number of feature maps required for the shuffling operation depends on the scaling factor. For example, if we scale by a factor of 2 we need $2^2 = 4$ feature maps, as we scale in both x- and y-dimension. We then rearrange the pixels of the 4 feature maps to the HR space. The inverse operation of pixel shuffling is called pixel unshuffling and rearranges HR features into multiple LR feature maps. Pixel (un)shuffling has shown to be a highly effective method in SR [30].

Performing the convolution operation depth-wise is another way to increase the efficiency. Normally, a convolution applies the filter across all input channels and then sums them up for the feature map. In a depth-wise convolution the filter is only applied on a single input channel, essentially generating one feature map for each input channel independently. By decoupling spatial and depth information, depth-wise convolutions significantly reduce the number of parameters and computations. Depth-wise convolutions are often coupled with point-wise convolutions, meaning convolutions with a kernel of size (1x1). This combination allows to still capture cross-channel information. Depth-wise separable convolutions have proven their effectiveness, especially on limited mobile hardware [31].

Recently, attention mechanisms have shown great potential in vision tasks. The most famous mechanism is the multi-head attention known for its use in the transformer model architecture, proposed by *"Attention is all you need"* [32]. The multiple heads focus on different parts of the input simultaneously, each head computing its own set of queries, keys and values. The combined results allow the model to capture different relationships within the data. The multi-head attention mechanism is primarily used in natural language processing (NLP) tasks, such as text style transfer, but can also be used in vision tasks. The Vision Transfomer (ViT) model has shown the effectiveness of the transformer architecture in vision tasks, by remodeling the image into a sequence of patches, similar to how sentences are represented as sequences of words [33]. To improve the efficiency in vision tasks, the Convolutional vision Transformer (CvT) introduces convolutions into the ViT. In particular, the linear layers of the multi-head attention block are replaced by depth-wise convolution layers, changing the behavior of the multi-head attention by adding a stronger focus on spatial relations, instead of global relations [18].

Another recent trend follows structural reparameterization to increase the performance in different SR tasks [34, 35, 36]. Similar to pixel shuffling, structural reparameterization tries to rearrange learned features into a more efficient representation. During training CNNs use a multi-branch architecture, which is restructured to a simpler form for inference, as seen in figure 3.2.

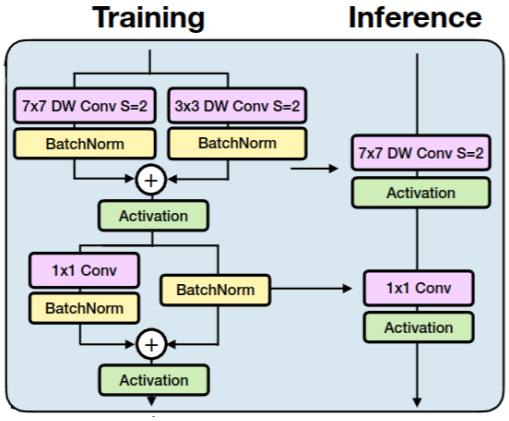

Figure 3.2: Example of structural reparameterization: The network architecture is more complex while trained and restructured to a simpler form when used for inference. [34]

Training a CNN means to adjust the weights inside the convolution blocks according to the error in the result the network produces compared to the target image. This adjustment process is called backpropagation. Inference means using the function defined by the network architecture and all the trained weights of each block are not modified anymore. The network, therefore, is only used to produce an output, omitting the step of backpropagation. Typically, the time a network needs to produce an output, as well as the memory consumption on the GPU are measured during inference, as it is the only step necessary once the training has been completed. As a result, we can further optimize the process by getting rid of all variables necessary for backpropagation.

The multi-branch architecture allows the network to fit a more complex function to perform SR. Once the training is done, the multiple branches can be optimized into a single convolution block by combining the trained weights from the different blocks and arranging them into a new single convolution block. During inference we only use the new single convolution block instead of the multi-branch design. While structural reparameterization makes the training process of the network more complex and slower, it simplifies the

network during inference and provides a performance boost for SR.

3.2 Warp Zone

In the context of VSR and RRSR, warping describes the process of rearranging the pixels from previous or past frames to the same point in time of the current frame. For this to happen, motion vectors must be calculated, describing how, e.g., the current frame's pixels have moved since the previous frame. Motion vectors are represented as 2D vectors (dx, dy), with dx as horizontal displacement and dy as vertical displacement in the frame's space. The displacement is represented by positive and negative values which embed its direction. For example, $(-3, 5)$ would mean the pixel moved 3 to the left and 5 up since the last time it had been rendered. For calculation purposes, these values are normalized to the range of $(-1, 1)$, or in the case of Unreal's motion vector calculation, they are normalized to the range of $(-0.5, 0.5)$ [37]. The flow field, our motion vector data across the whole frame, therefore, represents the displacement of every pixel in x and y direction. The warping operation utilizes the flow field, by creating a normalized grid for every pixel's location in the frame to be warped. The warping operation then applies the flow field onto the grid, moving the pixel's location according to the corresponding displacement values. This results, for example, in the previous frame to be forward warped based on the motion vector data with regard to the current frame. Warping can be applied in both directions, forward and backward. It has shown to increase SR accuracy [38] and provides a good basis for generating completely new frames [39]. For our use case, we only use forward warping.

3.3 Real-time rendering methods

In this section, we focus on current state of the art (SOTA) methods used for RRSR. Spatial temporal super sampling (STSS) [14] performs spatial as well as temporal SR, as the name implies. While spatial SR upsamples the current LR frame into HR space, temporal SR tries to generate a completely new frame by either interpolation or extrapolation. STSS performs extrapolation by utilizing the G-buffer information from the extrapolated frame. They perform a special warping operation by applying a custom shader in Unreal's Render Pipeline to inpaint the input for their extrapolated frames. Unfortunately, they use a custom version of Unreal Engine 4.27[1], an older version of Unreal based on ExtraNet's codebase [40]. Extranet [40] only performs temporal SR in the form of extrapolation utilizing Unreal Engine. Therefore, we modify and use STSS as a spatial SR method only. At the core of STSS lies a U-Net architecture, as seen in figure 3.3.

The U-Net architecture is a type of CNN which was originally developed for biomedical image segmentation tasks. Since then, it has been applied in various tasks due to its effectiveness. A U-Net architecture consists of two main parts: the encoder and the decoder. The encoder captures information about the image by applying convolutions and further downsampling the image. Downsampling can happen through a maxpool operation or applying a convolution with a stride greater than one, further reducing the dimensions. While reducing the feature maps dimension, the U-Net architecture doubles the amount of feature maps. The decoder constructs the output image from the encoded features by progressively upsampling the feature maps to the original image size. The core idea in this architecture is the skipping of connections, as it allows the decoder to use shallow feature maps from the first encoder convolution blocks. The multiple down- and upsampling operations form a U-like appearance, hence the name [17].

[1] `https://github.com/fuxihao66/UnrealEngine/tree/UE425ExtraNet_active`

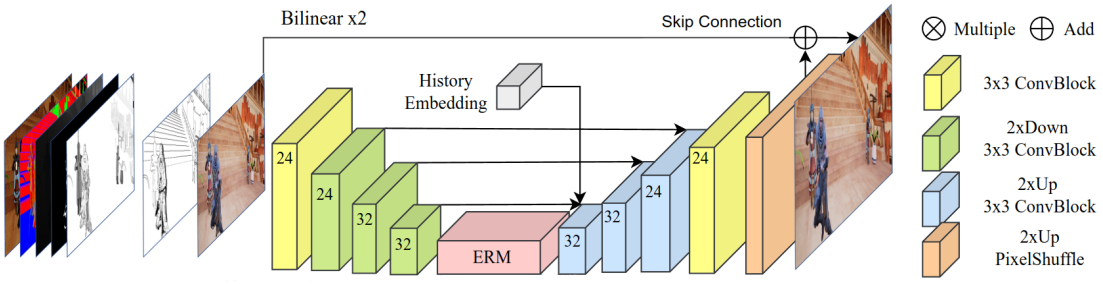

Figure 3.3: Network architecture of Spatial Temporal Super Sampling (STSS) following a classic U-Net style. In addition, the low resolution frame gets bilinear upsampled and added to the U-Net's output. A second history encoder is used to embed information from the two previous frames [14].

ExtraSS [15], the successor to ExtraNet, also performs spatial and temporal SR by extrapolating every second frame. It uses a similar U-Net architecture like STSS, but claims to perform a different warping operation called G-buffer guided warping. Before feeding the warped frames into the network, a second U-Net called Flow-based Refinement is used to handle artifacts introduced in the warping operation. Using the motion vectors provided by the engine has multiple shortcomings. Motion vectors do not track particle systems in the scene, occluded objects cannot be fully tracked, and translucent objects, like glass surfaces, lead to artifacts after the warping operation. As we did not implement the G-buffer guided warping strategy and the authors did not publish their code base, we refrain from using this network as another comparison.

Neural super-resolution with radiance demodulation (NSRRD) [13] follows a different idea. The creators demodulate the lighting pass from the rendered frame and perform spatial SR only on the lighting pass. As the last step in their network, they modulate the upsampled lighting pass with the base color buffer rendered in HR space. They show that rendering G-buffer information in HR space theoretically does not affect the performance of the rendering process dramatically. At the core of their architecture lies a lightweight recurrent convolutional long short-term memory (ConvLSTM) module. Long short-term memory (LSTM) blocks are used in recurrent neural networks (RNN) to keep information over a long time and help eliminate the vanishing gradient problem. The vanishing gradient phenomenon occurs when a network has a long sequence of layers, making it a deep network. The loss which is back propagated through all layers becomes so small, that it starts vanishing. LSTMs are primarily used in natural language processing tasks, as the context of words needs to maintained for a long time. The network utilizes a memory cell containing a recurrent hidden state to maintain the context. Instead of using fully connected layers, ConvLSTM incorporates convolutional layers to capture spatial and temporal features, which makes it suitable for SR [41].

The full architecture of NSRRD can be seen in figure 3.4. In addition to the two warped previous frames and the LR lighting pass, the previous SR output is warped and fed into the network through a pixel unshuffle as well. The warped previous frames are concatenated with a motion mask, created from so-called dual motion vectors. The idea behind dual motion vectors is the addition of the current to the previous motion vector information. The mask is then created from the difference between the dual motion vectors and the previous motion vector data, highlighting parts which had previously been occluded. After the ConvLSTM, which internally also utilizes the previous state, a small U-Net architecture is used to upsample the lighting pass frame into HR. As NSRRD uses the Unity game engine for generating its data and a custom shader for obtaining the dual motion vector data, as

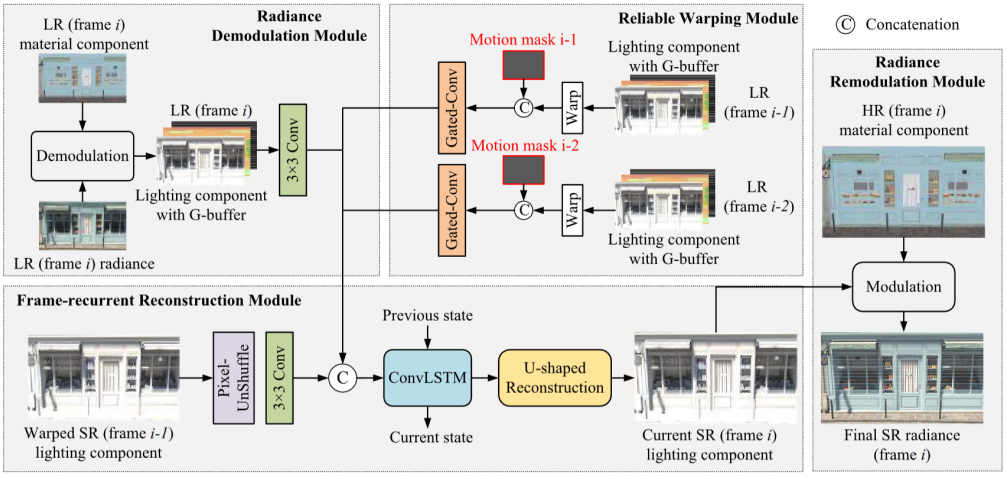

Figure 3.4: Neural Super-Resolution with Radiance Demodulation upsamples the demodulated lighting pass frame and modulate that with the base color frame in high resolution space. It uses a recurrent convolutional long short-term memory cell in combination with a small U-Net [13].

well as lighting pass information and G-buffers in HR space, the idea cannot directly be tested on our data set. Yet, we try a modified version of their architecture without the demodulation and the motion mask.

FuseSR [16], similar to NSRRD, uses the auxiliary G-buffer information in HR space and demodulates the rendered frame into the lighting pass information. Instead of performing SR only on the lighting pass and modulating it back in the last step, they let their network handle the modulation step. They propose a so-called H-Net architecture which they describe as a fusion network. It aligns the G-buffer information in HR space, and the current as well as previous frame input and feeds that into their architecture. The G-buffer information is downsampled by pixel unshuffeling. The fusion network consists of multiple residual blocks, a long sequence of convolution layers with skip connections to reduce the vanishing gradient problem. Considering FuseSR focuses on upsampling to 4k, using factors 4 and 8, and we rely on a factor of 2, we refrained from using their architecture.

4 Methodology

As far as we know, only [14] and [12] provide a dataset suitable for RRSR, and their datasets neither contain 1080p LR frames nor do they include additional G-buffers or 4k HR frames. Therefore, we created our own dataset called **U**nreal Engine **S**tylized **M**otion **M**atching (USMM) using Unreal Engine 5.4. In addition, we designed a lightweight network called **U**nreal **R**eal-**T**ime **S**uper **R**esolution (URTSR), which is capable of upsampling rendered 1080p frames to 4k in real-time. In this section, we illustrate how we created USMM, how we designed URTSR and what we have implemented so far.

4.1 Generating the data set

In contrast to a SISR dataset, like DIV2K [42], or a VSR dataset, such as REDS [43], a RRSR dataset contains auxiliary buffers, including base color, depth map and motion vector information, to name a few. In addition, the LR images from DIV2K and LR frames from REDS are typically acquired using a degradation kernel like bicubic downsampling, while a real-time renderer allows us to render the frames in different resolutions. This leads to a different set of artifacts in the LR frames, such as aliasing, dithering and detail loss, which are counteracted in the HR frame by applying anti-aliasing methods. As we want to obtain auxiliary G-buffers, we need to use a renderer capable of deferred rendering. For this purpose, we choose the Unreal Engine, a SOTA commercial game engine. More precisely, we use the newest version in form of Unreal Engine 5.4, as it provides new features such as Lumen, a global illumination system, and Nanite, an automatic level of detail (LoD) system, which enables us to render up to highly realistic environments used in the most elaborate games, called AAA games.

Video games come in a variety of camera work, with different content shown from different perspectives. Examples are real-time strategy games like Star Craft 2 [44], which is displayed from a top-down perspective or first-person shooter games like Doom Eternal [45]. We decided to focus on games with a third-person perspective, such as Elden Ring [46]. To simulate their gameplay, we choose two different kinds of sequences: (a) overview camera shots and (b) third person camera shots. Overview camera shots are used in games to give an overview of the level, like an aerial view of the track before the start of a race in a racing game, or to highlight certain objects, to motivate the action of acquiring a certain item to proceed in the game. The third person camera shots we used are simulating the gameplay in third person games, as the camera is focused on the character the player is controlling and moving according to the received input. More detailed information is provided later

in the chapter. We think data collected from these sequences are sufficient to recreate gameplay suitable for RRSR as our data is comparable to [14] and [12].

Similar to FuseSR [16], we wanted to use the freely available City Sample[1] project, and in addition the Electric Dreams Environment[2], the Hillside Sample[3] and the Valley of the Ancient[4] demos, to generate our (LR, HR) frame pairs. These sample projects do an excellent job in reflecting AAA game content, as they incorporate high fidelity environments with realistic lighting. Unfortunately, we encountered multiple problems leading to severe differences in the (LR, HR) frame pairs, which made them unsuitable for training our network. We elaborate the limitations we faced in 6.3. Therefore, we used more simple stylized content assets and combined them with Unreal Engine's Game Animation Sample project[5] instead. Stylized content refers to a type of visual design that deliberately deviates from highly realistic representations. This can be expressed in exaggerated character proportions, simplified shapes used for geometry, or environments in unusually bright color schemes. Examples for stylized games include but are not limited to The Legend of Zelda Tears of the Kingdom [6] or Overwatch 2 [47]. The big advantage of stylized content for us specifically is the fact that far less resources are needed, thus making it work more smoothly with our setup. For example, the City Sample project requires 64 gigabyte (GB) of random-access memory (RAM) and takes hours to compile when strated for the first time. Even the size of the package alone makes a big difference, as the City Sample project needs around 100 GB of storage space, whereas all our stylized assets combined only take up around 10 GB. We used the first three of the following stylized environments for generating our training and validation data and the fourth environment only for additional testing.

In particular, we used:

- Stylized Egypt[6], an Egyptian town with a pyramid.
- Stylized Fantasy Provencal[7], a medieval town with windmills and a castle.
- Stylized Eastern Village[8], an eastern village with a big palace.
- Stylized Town[9], a street in a modern French town.

As all of these asset packs come with fully modeled environment and animated objects, e.g. the windmills in stylized provencal, we can use them for our overview camera shots. However, for the third person camera shots, we are still missing a fully animated character, which can react to our input. Therefore, we combined these environments with the Game Animation Sample project. The Game Animation Sample project showcases playable characters with an automatic animation selection system based on the player input. This system is called Motion Matching and continuously selects the best frame of an animation from a big pool of animations. It includes a standard input controller, as well as multiple rigged characters, such as Echo from the Valley of the Ancient demo. The modularity of this project allowed us to plug and play the animated characters into our stylized environments and use them for our third person camera shots.

[1] https://www.unrealengine.com/marketplace/en-US/product/city-sample
[2] https://www.unrealengine.com/marketplace/en-US/product/electric-dreams-env
[3] https://www.unrealengine.com/marketplace/en-US/product/hillside-sample-project
[4] https://www.unrealengine.com/marketplace/en-US/product/ancient-game-01
[5] https://www.unrealengine.com/en-US/blog/game-animation-sample
[6] https://www.unrealengine.com/marketplace/en-US/product/stylized-egypt
[7] https://www.unrealengine.com/marketplace/en-US/product/stylized-fantasy-provencal
[8] https://www.unrealengine.com/marketplace/en-US/product/stylized-eastern-village
[9] https://www.unrealengine.com/marketplace/en-US/product/2852d85b8c0f4192b10046394bb82b38

For generating the overview camera shots, we used the sequencer [48] to create a Level Sequence element. We further added a camera actor to the Level Sequence in the environment we wanted the shot to take place. Every Level Sequence has a timeline tracking the game objects associated with it. In case of the overview camera shot, we only track the camera actor's transform over time of the sequence by key framing them. Once the sequence plays, the camera actor's position and rotation is interpolated, based on the key frames set on the timeline.

Generating the third person camera shots is more complicated, as we want to track the player's character movement over the sequence. Fortunately, Unreal Engine has a plugin to accomplish this in form of the Take Recorder [49]. The Take Recorder allows us to record gameplay from a live performance, so we can simply start the recording and play the character for the duration of the sequence. The recording is saved automatically and converted into a valid sequence, tracking the player character's movement.

We recorded for scenes per environment: two overview and two third person camera sequences, one for training and one for validation for each perspective. This excludes the stylized town environment, as we only generated one overview camera and one third person camera sequence for testing. An overview of the camera trajectory in each sequence of each level is shown in figure 4.1.

Figure 4.1: Overview of how the recorded sequences progress through the levels Egypt, village, town and provencal (left to right, top before bottom). The sequences are color coded: The overview camera trajectories for training are orange yellow, for validation/testing light cyan. The third person camera movements for training are light green, for validation/testing sky blue.

Overall, we recorded 6 sequences for training and validation, each, and 2 for testing. With all sequences recorded, we still need to render each LR and HR frame, as well as the auxiliary buffers frames into individual external files. To accomplish this, we use yet another plugin called Movie Render Queue (MRQ) [50]. The MRQ allows us to render a sequence (a) in different resolutions, (b) with additional render passes and (c) while special settings are applied to it. (a) allows us to render the same sequence in 540p, 1080p and 4k, giving us our (LR, HR) frame pairs. (b) enables us to render the auxiliary G-buffer information, as we can inject custom materials, which include shaders to obtain this information. (c) lets us set the anti-aliasing method, as well as forcing the highest LOD level for the geometry and a negative mip mapping bias for the textures. Mip stands for Multum In Parvo, which is latin for many things in a small space. Mip maps are precomputed versions of textures at different resolutions, simplified, they are LOD for

textures. Applying a negative mip mapping bias increases the LOD chosen for the given texture. For the LR renderings we chose no anti-aliasing method, while for the HR render we used SSAA, as we set a command increasing the render resolution internally by 100%. This means, to obtain our 4k HR frames, we internally rendered them in 7680x4320, also known as 8k, to perform SSAA. We tried different anti-aliasing methods, such as TAA, MSAA or a combination of both of them, but noticed lighting differences in certain situations, which will be discussed in 6.3.

We rendered the LR, HR and auxiliary buffer data into individual .png files, but not the motion vector data, as it needs to be treated differently. Unreal Engine stores motion vectors normalized to the entire screen in the range of [-0.5, 0.5] for the X-axis in the R-channel, and for the Y-axis in the G-channel of an image file [37]. To actually save these vectors in an image file, we add a value of 0.5 across all pixels of the frame and undo that operation later when using the file for warping the image. Unfortunately, rendering this information into a .png file triggers the tone mapper of Unreal Engine's rendering pipeline. Instead of writing the values linearly into the .png file, the tone mapper rewrites the high-definition range (HDR) values into standard definition range (SDR) values. Furthermore, a tone curve is applied, adjusting for example contrast and brightness[51]. To overcome this issue, Unreal Engine supports the OpenEXR, or .exr, file format, which allows writing values linearly. OpenEXR is an HDR file format, which supports multiple channels, more than 3 or 4, and supports 16 bits of precision.

Each training sequence is 40 seconds long, captured with a framerate of 60 fps, resulting in 2400 frames that have to be rendered per sequence. For 6 training sequences, this totals to 14400 (LR, HR) frame pairs. Each validation/testing sequence is 5 seconds long, again with a framerate of 60 fps, providing us with 300 frames per sequence. With 6 validation sequences, this totals to 1800 (LR, HR) frame pairs. The 2 testing sequences, result in 600 (LR, HR) frame pairs. In total, we created 16800 (LR, HR) frame pairs, where LR contains the rendered and final frame in LR, as well as G-buffer information for base color, depth map, roughness, metallic, NoV, motion vectors in .exr format, world normal and world position. As we rendered each sequence three times in 540p (LR), 1080p (LR) and 4k (HR), we generated around 550 GB of data. An example of one (LR, HR) frame pair can be seen in figure 4.2.

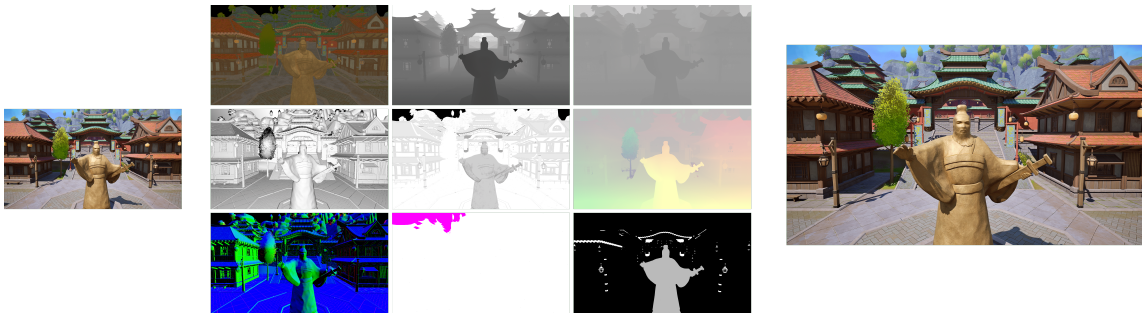

Figure 4.2: An example of a (LR, HR) frame pair from our dataset. From left to right: final rendered frame in 1080p, G-buffer frames in 1080p and final rendered frame in 4k. For the G-buffer block, from top left to bottom right: base color, depth map, depth map normalized, NoV, roughness, motion vectors, world normal, world position and metallic.

We provide a separate repository containing the complete Unreal Engine project used for generating our USMM dataset at `https://gitlab.informatik.uni-wuerzburg.de/Brandner/generate_ue_stylized_data`. In addition, we provide a small content plugin containing the setting files for the MRQ to render the (LR, HR) frame pairs in 540p, 1080p

and 4k respectively, as well as the custom materials to obtain the G-buffer information. This plugin allows users to generate the aforementioned data from any sequences in other Unreal Engine 5 projects [52]. A step-by-step guide for obtaining (LR, HR) frame pairs in our project can be found in the appendix A.1.

4.2 Designing the Network

Since we want to perform SR on 1080p or 540p to 4k in a 60 fps setting, we need a network capable of upsampling our input data in far less than 16ms, as the LR frame also needs to be rendered by the engine.

Therefore, we propose our network called URTSR, with the architecture depicted in figure 4.3. URTSR follows a shallow U-Net architecture. We use two encoders, one for the LR frame and the G-buffer information, such as base color, depth map, NoV, and one for the two previously rendered frames, called history encoder. Both encoders follow a similar architecture.

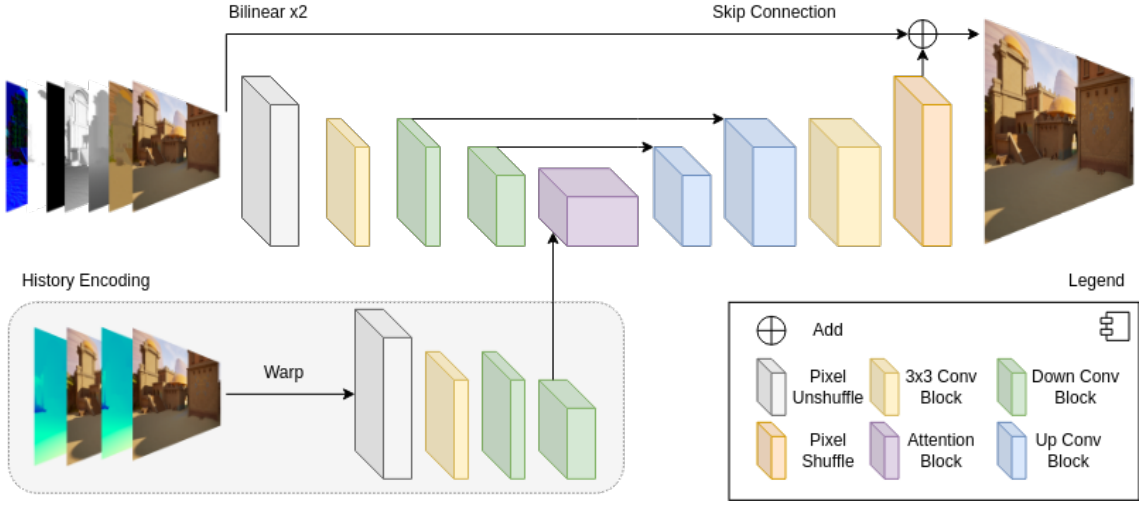

Figure 4.3: Our proposed network URTSR follows a shallow U-Net architecture. We use two encoders: The first encoder uses the current (LR, G-Buffer) frame pair data. And the second history encoder utilizes the latest 2 rendered LR frames forward warped based on the motion vector data from the subsequent frame. The information from both encoders is combined and fed into the bottom layer, containing an attention block. Combining the feature maps from the previous down convolution blocks, we increase the feature map dimensions with up convolution blocks. A final Pixel Shuffle layer upsamples to 4k. This is combined with the bilinear upsampled LR frame from the beginning, to form our network's output.

Similar to many methods proposed in [1], we begin with a pixel unshuffle layer to reduce the feature map dimensions. For the LR and G-buffer input, the feature maps are fed into a convolution block with a kernel size of (3x3). The convolution block performs two convolution operations with a rectified linear unit (ReLU) [53] activation in between. We further decrease the feature map dimensions with two subsequent down convolution blocks. The down convolution block, first, performs a convolution operation with a stride size of two, cutting the feature map dimensions in half and afterwards performs the same double convolution operation with ReLU in between.

For the history encoder, we use the two previously rendered LR frames and forward warp them based on the motion vector data from the subsequent frames. This means, we warp

the previous LR frames to the same point in time as the current LR frame, by utilizing the data in the current motion vector frame and, for the oldest frame, additionally the motion vector data from the previous frame, as we need to apply two warping operations. On top of the three artifacts we have identified in the form of aliasing, dithering and loss of detail, the warping process induces further local artifacts to the frame. Figure 4.4 showcases two prominent problems: First, edges around game object can be drawn a second time, as the motion vector data can be inaccurate. Second, black holes are drawn at the edges of the frame itself, as the motion vectors simply have no information from outside the frame. The history encoder follows a similar structure as our first encoder, but instead of performing standard convolution operations, it performs gated convolutions. In a gated convolution, two convolutions are performed with the same kernel, stride, padding and dilation, one outputting the desired number of feature maps, while the other produces a single feature map that is used as a mask. The output of the first convolution gets multiplied by the mask of the second convolution, creating a gating mechanism. The gated convolution is primarily used in inpainting tasks, where parts of the image need to be corrected due to significant artifacts [54]. The idea behind the gating convolution operation is to be particularly helpful for removing the additional artifacts.

Figure 4.4: Forward warping the previous frame to the current time leads to two artifacts: (a) edges of game objects are drawn a second time and (b) black holes are drawn at the edges of the frame itself

The feature maps of both encoders are concatenated and fed into the bottom layer of our U-Net architecture. The bottom layer consists of a modified multi-head attention block, utilizing convolution layers instead of linear layers, as proposed in [18]. The feature maps from the convolution attention block are bilinear upsampled and concatenated with the input of the previously used down convolution blocks via a skip connection. Furthermore, the up convolution block performs two (3x3) convolution operations with a ReLU activation in between. The second up convolution block concatenates the resulting feature maps with the input from the first down convolution block via a skip connection and then passes this information to the other (3x3) convolution block. Finally, a pixel shuffle layer restructures the information of 48 feature maps with a dimension of (960, 540) to 3 times (3840, 2160). The upsampled feature maps are then added to the bilinear upsampled LR frame input to produce the 4k output of our network.

The output of our network is compared to the anti-aliased 4k HR frame at the same point in time, our ground truth. We use the loss of [14], which is a combination of the L1 loss and LPIPS, defined in 4.1.

$$L_{ours} = \text{L1}(out, gt) + 0.1 * \text{LPIPS}(out, gt) \tag{4.1}$$

We use the adaptive moment estimation (Adam) optimizer, as it requires less tuning of

the learning rate and has shown to perform well on large datasets. Adam uses a decay approach to avoid oscillations and smoothly move towards the minimum. The decay rate is controlled by two hyperparameters, β_1 and β_2. We initialize β_1 and β_2 with their default values of 0.9 and 0.99 respectively [55]. For the scheduler we use StepLR. StepLR reduces the learning rate by a fixed amount after reaching a certain amount of epochs. The step-wise decay allows network models to take larger steps in the early epochs and smaller step towards later epochs, fine-tuning the network.

4.3 Implementation

In this section we want to share some design approaches we implemented, which helped with the modularity of our codebase. We used Python and Pytorch [56] to implement and train our proposed URTSR architecture. We based our training on the dataset class of Pytorch and modified it to load and return (a) the LR frame, (b) the corresponding G-buffer frames, (c) a variable amount of frames previously rendered, (d) the motion vector frames to warp these frames to the current point in time and (e) the ground truth HR frame. To have fine control over which G-buffer data and how many previous frames are loaded, we used YAML Ain't Markup Language (YAML) configuration files. Similar to the approach of BasicSR [57], where hyperparameters like the learning rate or the crop size are exposed to more easily set up different training configurations, we added options for loading the auxiliary G-buffer frames or perform a forward warping operation. An example of such a .yaml configuration file is shown in A.2.1.

To check the correctness and gain insight of the implementation of our network model files, each model file can be executed by itself. We added three behaviors:

- A summary of all the network layers with a precise description of how many parameters and multiplication plus additions are used. We use the summary function from the torchinfo package.
- Measuring the average inference time without back propagation.
- Measuring the memory footprint on the VRAM without back propagation.

All three behaviors are inherited from a base class. By providing dummy input tensors for the LR frame, the G-buffers and the history, we can check if the model compiles, how many total parameters it uses, what average forward pass time it has and how much memory needs to be allocated on the GPU. Our single file approach allows to debug the network model file more easily, check inference speeds of single blocks used in the network and overall helps building the network block by block. An example output of running our model files is shown in A.2.2.

Pytorch allows integrating tensorboard to visualize the training process of different network runs [58]. We integrated this library and track the loss for every training epoch. After 10 epochs we perform a validation step on the corresponding validation dataset to our training dataset. For the validation step, we track the three metrics PSNR, SSIM and LPIPS, as well as write and display one random example to the board. This example visualizes the input in form of the LR, G-buffer, history frames for our network, as well as the ground truth and the output of our network in HR space. We show an example run in A.2.3.

We believe the proposed approaches provide good control, make it easy to share network files and help visualizing the training process.

5 Evaluation

In this section we show the results of the objective metrics PSNR, SSIM and LPIPs across our four environments and 8 sequences. We compare the statistics of the different implemented methods, as well as the effect of different configurations on our URTSR network's results.

We implemented the following network models:

- RTSRN [1], a SISR method for reference.
- STSS [14], we modified STSS to only perform spatial SR.
- NSRRD [13], we removed the demodulation part and perform SR on the rendered LR frame, instead of the lighting pass.
- URTSR, our own method to perform highly efficient RRSR.

We compare our implemented network models with the interpolation methods bilinear and bicubic on the objective metrics PSNR, SSIM and LPIPS. In particular, we compare the methods on each of our scenes separately, this includes egypt (E), fantasy provencal (FP), eastern village (EV) and town (T^*). As described in 4.1 we rendered one overview camera shot ($_o$) and one third person camera shot ($_{tp}$) for every scene. Note, for egypt, fantasy provencal and eastern village we trained the network models only on the corresponding training data, so we state single scene performance. As there is no corresponding training sequence for town, we trained on all training sequences, therefore we state multi scene performance and test the generalization ability of each method. For single sequence performance we trained each network method for 200 epochs, with 2.400 training examples per epoch this results in 480.000 iterations. For multi sequence performance we trained each network method for 100 epochs, with 14.000 training examples per epoch this results in 1.400.000 iterations. We randomly select frames, crop and rotate them for data augmentation. Except for NSRRD, as the recurrent nature of the ConvLSTM prohibits us from shuffling the input frames. Due to time constraints we do not provide multi sequence performance for NSRRD. We used the following hyperparameter settings across all network methods:

- batch size set to 4, crop size to 256
- learning rate set to 0.001
- Adam optimzer, default β_1 to 0.9 and β_2 to 0.99

- StepLR, step size set to 50, gamma to 0.9 and starting the decay at epoch 20

The values for PSNR can be seen in table 5.1, for SSIM in 5.2 and for LPIPS in 5.3. For PSNR and SSIM higher values are better, while for LPIPS lower values are better. In context to the table's content, the three best values are highlighted using the following colors: (1), (2), (3).

Table 5.1: PSNR values across different methods on our validation and testing scenes

PSNR ↑	**Method**					
Scene	**bilinear**	**bicubic**	**RTSRN**	**STSS**	**NSRRD**	**URTSR**
E_o	36.76	36.63	37.55	37.13	35.19	36.20
E_{tp}	34.69	34.56	35.60	35.48	35.00	34.60
FP_o	26.69	26.10	27.41	27.29	27.17	26.79
FP_{tp}	27.54	27.00	28.35	28.13	28.01	27.79
EV_o	24.15	23.69	24.98	24.77	24.68	24.41
EV_{tp}	27.75	27.70	29.06	29.17	28.84	28.18
T_o^*	30.77	30.71	31.02	31.24	-	30.63
T_{tp}^*	29.13	28.94	29.66	29.71	-	29.22

Table 5.2: SSIM values across different methods on our validation and testing scenes

SSIM ↑	**Method**					
Scene	**bilinear**	**bicubic**	**RTSRN**	**STSS**	**NSRRD**	**URTSR**
E_o	0.9637	0.9628	0.9681	0.9661	0.9636	0.9611
E_{tp}	0.9584	0.9584	0.9646	0.9639	0.9605	0.9558
FP_o	0.8028	0.7987	0.8238	0.8231	0.8183	0.8070
FP_{tp}	0.8780	0.8756	0.8952	0.8935	0.8892	0.8842
EV_o	0.7391	0.7490	0.7739	0.7666	0.7636	0.7510
EV_{tp}	0.8835	0.8895	0.9125	0.9141	0.9068	0.8922
T_o^*	0.8837	0.8882	0.8957	0.8981	-	0.8881
T_{tp}^*	0.8642	0.8687	0.8805	0.8816	-	0.8710

Table 5.3: LPIPS values across different methods on our validation and testing scenes

LPIPS ↓	**Method**					
Scene	**bilinear**	**bicubic**	**RTSRN**	**STSS**	**NSRRD**	**URTSR**
E_o	0.1349	0.1128	0.0432	0.0464	0.0503	0.0732
E_{tp}	0.1245	0.1079	0.0449	0.0471	0.0528	0.0704
FP_o	0.2033	0.2021	0.1130	0.1166	0.1186	0.1406
FP_{tp}	0.1507	0.1455	0.0782	0.0796	0.0833	0.0992
EV_o	0.2623	0.2591	0.1269	0.1402	0.1362	0.1631
EV_{tp}	0.1749	0.1681	0.0682	0.0700	0.0759	0.1004
T_o^*	0.1980	0.1896	0.1002	0.0993	-	0.1240
T_{tp}^*	0.2129	0.2086	0.1094	0.1108	-	0.1367

We also look at the statistics of each method. In particular we look at (a) the inference speed in miliseconds, (b) VRAM usage in megabyte, (c) total number of parameters and (d) MACs, the number of multiply-accumulate operations measuring the computational

complexity, represented in billions. All network models were trained and tested on a system with an Nvidia 4080 (GPU), an Intel 14700K (CPU) and 32 GB 5600 (RAM). The statistics for each method can be seen in table 5.4.

Table 5.4: Statistic values across different methods on our validation and testing scenes

STATS ↓	**Method**					
	Details	**bilinear**	**bicubic**	**RTSRN**	**STSS**	**NSRRD**	**URTSR**
	speed (ms)	0.27	0.50	56.13	29.31	62.30	12,88
	VRAM (MB)	-	-	2335	1977	4853	725
	# of params	-	-	193,356	231,567	238,933	216.642
	MACS (G)	-	-	200.47	45.02	142.57	25,61

At last, we provide a small ablation study, testing our URTSR architecture with different configurations to see how this changes the behavior of our architecture and influences our results on the scene E_o. The different configurations include:

- barebone, testing our network as a pure VSR Method
- with G-buffers, we include the auxiliary buffer data
- history frames warped, the two previous frames get forward warped to the current frame's time
- G-buffers and warped, adding the auxiliary buffers and performing the warping operation

The results can be seen in table 5.5.

Table 5.5: Ablation study for comparing the different configurations of our method URTSR and their impact on the objective metrics PSNR, SSIM and LPIPS.

Metrics	**Configuration**			
	barebone	**with G-buffers**	**history frames warped**	**G-buffers and warped**
PSNR ↑	36.54	36.5	36.58	36.54
SSIM ↑	0.9589	0.9584	0.9594	0.9588
LPIPS ↓	0.0807	0.0868	0.0736	0.0917

6 Discussion

> "Experience is simply the name we give our mistakes."
> — Oscar Wilde

In this section we want to talk about the results we achieved, the mistakes we made along the way, the limitations we faced and finally what ideas could be pursued in the future.

6.1 Results

Based on the values we get from the interpolation methods bilinear and bicubic, we see that the environments present vastly different levels of difficulty for RRSR. Egypt, E_o and E_tp, for example is far easier for all our methods to achieve good results, while eastern village, EV_o, seems particular hard. Looking at the content rendered in the different scenes, we see a pattern. Scenes including game objects with a lot of detail, or high frequency, like grass, trees, flowers and bushes, tend to perform quite bad. These game objects are treated as instanced static meshes and are typically bundled as foliage. Foliage game objects are drawn multiple times in the same scene, with slight adjustments to their transform, and therefore can be optimized to increase render performance of the game. Because of the high frequency details, foliage game objects are particularly affected from the aliasing, dithering and detail loss rendering artifacts.

In most cases the neural methods are on pair with the interpolation methods in regards to PSNR and SSIM. With RTSRN and STSS slightly outperforming on PSNR with 1 point and SSIM with 0.1. Yet all neural methods vastly outperform the interpolation methods on LPIPS, cutting their values in half on average. Comparing only the interpolation methods, we see that bilinear outperforms bicubic in almost every scene in the metrics PSNR and SSIM, yet bicubic outperforms bilinear across all scenes in LPIPS. We, therefore, believe LPIPS is the most expressive metric for our use case and for RRSR content in general.

Comparing the neural methods, we draw multiple insights. Based on the reference SISR method RTSRN, NSRRD performs especially bad for being a dedicated RRSR method. Since we do modify the original architecture away from the demodulation idea, this might come as no surprise. Getting the lighting pass data is actually possible with the MRQ in Unreal [37]. But since NSRRD also needs the unlit buffer frame anti-aliased in HR space, we did not generate the additional data because of time and space restrictions. Looking at the inference speed and the memory footprint of this method, we think their architecture with the ConvLSTM block seems unsuitable for our restrictive use case.

STSS's U-net architecture performs very well as it comes very close in all metrics to RTSRN, but roughly cuts the inference speed in half with 29.31 miliseconds. Yet the memory footprint with 2 GB VRAM usage is quite high in a gaming context, where VRAM is an especially scarce resource.

Our method URTSR is beaten in every metric by the other neural methods in regard to PSNR, and SSIM sometimes even from bilinear and bicubic. Yet it outperforms the interpolation methods in every scene when it comes to LPIPS. Still, with an inference speed of 12.88 miliseconds and a VRAM footprint of 725 MB, it is the only method suitable for our restrictive use case. It achieves respectable image quality in removing dithering artifacts and some, but not all aliasing. Looking at figure 6.1, we can see a visual comparison of the different methods. We compare all methods with the input frame as reference and the ground truth frame. In the blue cutout, we compare how well the methods mitigated the dithering artifact around the shadows of the flowers. The interpolation methods bilinear and bicubic clearly struggle with removing the artifacts, while all neural methods do a far better job. Yet, no method is capable of removing dithering completely. In our opinion STSS does the best job here, closely followed by RTSRN. In the orange cutout, we compare how well the methods mitigated the aliasing artifacts of the windmill. Again, the interpolation methods are not capable of removing aliasing in this example. Our method URTSR does a better job, though aliasing is still clearly visible. In our opinion RTSRN mitigates aliasing the best, followed closely by STSS.

Figure 6.1: We compare the frames resulting from the different methods we implemented so far with the reference input and the ground truth frame. The blue example compares how well our methods mitigated dithering. In the orange example we compare how well our methods mitigated aliasing.

Regarding the ablation study from table 5.5, we can see that warping the previous frames produces the best results across all metrics. Adding the buffer information seems to distract our architecture and leads to worse results than using it as a barebone VSR method. Doing both, warping the previous frames and include the auxiliary buffer data, leads us to a net zero. It seems the restrictive design of our architecture and working immediately in a down scaled feature dimension implies that our network is not fully suited for RRSR. Further

investigations on how to utilize the G-buffer information, like in dedicated blocks/encoders, should therefore be pursued.

6.2 Failed tries

We tested structural reparametrization as it yields good performance for high inference speed. We implemented the RepNet architecture as proposed in [1]. Unfortunately, given our dataset and hyperparameters RepNet does not converge. We changed the hyperparameters as proposed in the paper, using a higher batch size of 32 and the L2 loss[1]. The changes lead RepNet to converge, but the results were lacking. PSNR and SSIM were worse than bilinear and LPIPS was very high compared to our other neural based methods. Still, we tried to use reparametrization blocks in our URTSR architecture, substituting the (3x3) conv blocks, as well as having a variable amount of them in the bottom layer, instead of the attention block. Both tries resulted in our network to get unstable, after 60 epochs the loss turned to not a number. Due to time constraints, we did not further investigate structural reparametrization.

We also tried the efficient video restoration network (EVRNet) architecture [59], a pure VSR method. The idea of having the same U-net encoder decoder blocks, but for different purposes in differential, alignment and fusion, seemed interesting. Yet, again, EVRNet did not converge given our dataset and hyperparameters, EVRNet uses a very low learning rate of 0.000001 and a custom scheduler. The scheduler first increases the learning rate over the first 100 epochs, and later decreasing it again. Implementing this behavior lead EVRNet to converge, yet the results where mediocre. We implemented a version of our network with the differential, alignment and fusion block design, but the results were lacking. In addition, the original architecture is far too slow, as the inference speed is roughly around 200 milliseconds, given our hardware specifications. We deemed the architecture to be unsuitable for our use case.

6.3 Limitations

As mentioned in 4.1 we want to highlight some limitations we faced, while rendering our (LR, HR) frame pairs. Since we use the MRQ to render the same sequences sequentially in different resolutions, it is essential that the sequences must be deterministic. This means, everything that includes some kind of randomness, needs to be excluded from the rendered scene. Specifically, this applies to: Particle systems, as the particles are randomly initialized and behave with some random variance. Unreal's physics system Chaos [60] is not deterministic, all objects physics simulated can behave slightly different. For example, we used a character with cloth simulation from the Game Animation sample project, which lead to differences in his coat behavior in the (LR, HR) frame pairs. In addition, we recommend disabling the texture streaming option in the project setting, as the automatic streaming leads to low LoD levels for textures for a couple of frames. The same applies to the setting for automatic pool growth, which should also be deactivated.

While working with the City Sample and Electric Dreams Environment, we noticed problems specific to these projects. When looking at the image 6.2, we noticed that the lights in the City Sample project, such as the highlighted lantern, only affect the wall in the HR frame, but not in the LR frame. In the Electric Dreams Environment project, foliage, such as small leaves, are present in the HR frame but not in the LR frame. Since these sample projects showcase new features and are very complex, we are not sure why these problems exist in them. That's why we decided to move away from these realistic yet complex projects and move on to more simple and easy stylized projects.

[1] `https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html`

(a) Lantern example rendered in 1080p.

(b) Lantern example rendered in 4k.

Figure 6.2: Looking at the differences between LR (a) and HR (b) frame pair. The lantern's light reflected on the wall in the house behind, is only present in the HR frame and not in the LR frame.

Next to the FXAA, MSAA and TAA options, the MRQ allows to set a variable spatial and temporal sample count to perform anti-aliasing. This can be seen as a combination of MSAA and TAA, as increasing the spatial sample count, refers to render the same frame with a small offset applied to it, while increasing the temporal sample count, refers to using the previous rendered frames to perform anti-aliasing. Using a combination of both is recommended according to the documentation [61]. As this indeed leads to high quality rendered frames with anti-aliasing applied to them, we noticed small lighting differences in the (LR, HR) frame pairs. Since this behavior is not documented, we have asked a question in the official Unreal forum[2]. That's why we have resorted to SSAA.

6.4 Future Work

Based on the related work, it should come as no surprise we aimed to achieve spatial and temporal SR. Similar to [14, 15, 39] we tried to use extrapolation to generate a frame in between two actually rendered frames. Frame generation comes in two flavors: (a) interpolation and (b) extrapolation. Why interpolation generates a frame in between two already rendered frames, extrapolation predicts the next frame based solely on previous rendered frames. In the context of videos, frame interpolation already has proven to be highly effective [62]. DLSS 3 [63] and FSR 3 [64] also show that frame interpolation can be used in a gaming context. Yet, frame interpolation introduces a severe problem in the form of additional latency. As games are interactive simulations low latency is required to give the player a high responsive experience. Normally high frame rates lead to low latency, but to effectively use frame interpolation in a gaming context, the interpolated frame needs to be displayed before the latest rendered frame. To achieve this the rendering logic of a game engine needs to be adjusted. This means the latest rendered frame is held back, to display the interpolated frame first. While the game now runs in a higher frame rate, this modification actually leads to a higher latency experienced for the player. Let's say a game runs at 60 fps, and we interpolate every second frame so 120 fps are displayed.

[2] `https://forums.unrealengine.com/t/movie-render-queue-get-similiar-low-resolution-and-high-resolution-images/1934041`

Since the latest rendered frame is hold back, the game is displayed with a slight delay of $\frac{1}{60}\,\text{s} = 16.\overline{6}\,\text{ms}$, this delay makes the game feel like 30 fps since $16.\overline{6}\,\text{ms} \times 2 = 33.\overline{3}\,\text{ms} = \frac{1}{30}\,\text{s}$. This problem does not occur with frame extrapolation and the render pipeline does not need to be adjusted either. It should therefore be preferred, especially at frame rates lower than 60, where the delay is more noticeable.

We tried different approaches to achieve acceptable extrapolation: Letting a network learn the temporal flow behavior by itself, for example we modified flow-agnostic video representation (FLAVR) [62] to perform extrapolation, instead of interpolation. Or pre-warping the LR frame based on the motion vector data of the next frame and then feeding it into our RRSR network. Unfortunately every approach resulted in blurry results. More time needs to be invested to find a solution given our data and our restrictive use case.

We are not discouraged by this and would like to present a few interesting ideas here. Similar to how frame interpolation modifies the render pipeline, we could modify it for extrapolation to further reduce the latency. The idea would be to increase the internal logic frame rate to the increased displayed frame rate. Then embed the input given by the player to the network, so that it predicts the extrapolated frame with more accuracy. Let's say the player moves the character for the last couple of frames only to the left, so every extrapolated frame would try to display the player more to the left, now suddenly the player moves into the opposite direction, leading to a divergence of the extrapolated frame and the latest rendered frame. Now if we let the extrapolated frame react to the player's input, by increasing the pooling rate, could fix this problem and in theory this would lead to lower latency experienced by the player.

Based on that, we propose an even more difficult to realize idea. The best aspect of RRSR for us is that no matter the content rendered, the network always needs the same time to achieve SR. This is especially useful in games with highly realistic environments, where the frame rate is most of the time bound by the GPU and therefore bound by how visually demanding the rendered scene is. What if we perform frame generation not based by previous and future rendered frames, but instead based by the internal state of the game. Removing or replacing the traditional rendering pipeline all together. In theory this would mean no matter the content rendered, games would be more likely bottle necked by the central processing unit (CPU)'s logic thread, instead of the GPU.

7 Conclusion

We investigated real-time rendering super resolution in Unreal Engine 5. We created a novel stylized dataset with Unreal Engine 5, suitable for real-time rendering super resolution. The dataset contains 16800 (LR, HR) frame pairs. The LR input includes 1080p and 540p aliased rendered frames, with auxiliary G-buffer information, like base color, depth map and motion vectors. While the HR ground truth comprises 4k anti-aliased rendered frames. The dataset is 550 GB in total and can be used for times 2 and times 4 super resolution. In addition, we provide a custom plugin to generate (LR, HR) frame pairs with auxiliary buffers within any projects using Unreal Engine 5. We designed a shallow U-Net architecture utilizing attention mechanism called URTSR. While it does not outperform related work in image quality, it is the only method we have implemented that is capable of performing super resolution from 1080p to 4k in a 60 fps real gaming use case. It shows significant image quality increase, in particular in LPIPS, over traditional interpolation methods, like bicubic. Yet it can not fully utilize additional information provided by auxiliary buffers, further investigations should therefore be pursued.

List of Figures

List of Tables

Acronyms

LR low-resolution

HR high-resolution

SR super resolution

1080p 1920x1080

4k 3840x2160

fps frames per second

RRSR real-time rendering super resolution

DLSS Deep Learning Super Sampling

FSR Fidelity Super Resolution

XeSS Xe Super Sampling

GPU graphics processing unit

SISR single image super resolution

VSR video super resolution

540p 960x540

270p 480x270

G-buffer geometry buffer

NoV normal orientation vector

VRAM video random-access memory

FXAA Fast aproximate anti-aliasing

MSAA Multi sample anti-aliasing

TAA Temporal anti-aliasing

SSAA Super sampling anti-aliasing

MOS mean opinion score

PSNR Peak signal-to-noise ratio

MSE mean squarred error

SSIM Structural similarity index measurement

LPIPS Learned perceptual image patch similarity

CNN convolution neural network

NLP natural language processing

ViT Vision Transfomer

CvT Convolutional vision Transformer

SOTA state of the art

STSS Spatial temporal super sampling

NSRRD Neural super-resolution with radiance demodulation

ConvLSTM convolutional long short-term memory

LSTM Long short-term memory

RNN recurrent neural networks

USMM **U**nreal Engine **S**tylized **M**otion **M**atching

URTSR **U**nreal **R**eal-**T**ime **S**uper **R**esolution

LoD level of detail

GB gigabyte

RAM random-access memory

MRQ Movie Render Queue

HDR high-definition range

SDR standard definition range

ReLU rectified linear unit

Adam adaptive moment estimation

YAML YAML Ain't Markup Language

FLAVR flow-agnostic video representation

CPU central processing unit

EVRNet efficient video restoration network

Bibliography

[1] M. V. Conde, E. Zamfir, R. Timofte, D. Motilla, C. Liu, Z. Zhang, Y. Peng, Y. Lin, J. Guo, X. Zou, *et al.*, "Efficient deep models for real-time 4k image super-resolution. ntire 2023 benchmark and report," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 1495–1521, 2023.

[2] NVIDIA, "Nvidia dlss." `https://www.nvidia.com/de-de/geforce/technologies/dlss/`.

[3] AMD, "Amd fidelityfx super resolution." `https://www.amd.com/en/technologies/fidelityfx-super-resolution`.

[4] Intel, "Xess super sampling, ai-enhanced upscaling." `https://www.intel.com/content/www/us/en/products/docs/discrete-gpus/arc/technology/xess.html`.

[5] R. Entertainment, "Alan Wake 2." `https://www.alanwake.com/`, 2023.

[6] Nintendo, "The Legend of Zelda: Tears of the Kingdom." `https://www.zelda.com/tears-of-the-kingdom/`, 2023.

[7] G. Foster, "Playstation's spectral super resolution (pssr) unveiled." `https://insider-gaming.com/playstations-spectral-super-resolution-pssr/`.

[8] AMD, "Radeon super resolution." `https://www.amd.com/en/products/software/adrenalin/radeon-super-resolution.html`.

[9] NVIDIA, "How to enable nvidia image scaling." `https://nvidia.custhelp.com/app/answers/detail/a_id/5280/~/how-to-enable-nvidia-image-scaling`.

[10] Unknown, "Lossless scaling on steam." `https://store.steampowered.com/app/993090/Lossless_Scaling/`.

[11] Microsoft, "Auto sr: Ai-powered super resolution for directx." `https://devblogs.microsoft.com/directx/autosr/`.

[12] A. Mercier, R. Erasmus, Y. Savani, M. Dhingra, F. Porikli, and G. Berger, "Efficient neural supersampling on a novel gaming dataset," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 296–306, 2023.

[13] J. Li, Z. Chen, X. Wu, L. Wang, B. Wang, and L. Zhang, "Neural super-resolution for real-time rendering with radiance demodulation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4357–4367, 2024.

[14] R. He, S. Zhou, Y. Sun, R. Cheng, W. Tan, and B. Yan, "Low-latency space-time supersampling for real-time rendering," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, pp. 2103–2111, 2024.

[15] S. Wu, S. Kim, Z. Zeng, D. Vembar, S. Jha, A. Kaplanyan, and L.-Q. Yan, "Extrass: A framework for joint spatial super sampling and frame extrapolation," in *SIGGRAPH Asia 2023 Conference Papers*, pp. 1–11, 2023.

[16] Z. Zhong, J. Zhu, Y. Dai, C. Zheng, G. Chen, Y. Huo, H. Bao, and R. Wang, "Fusesr: Super resolution for real-time rendering through efficient multi-resolution fusion," in *SIGGRAPH Asia 2023 Conference Papers*, pp. 1–10, 2023.

[17] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pp. 234–241, Springer, 2015.

[18] H. Wu, B. Xiao, N. Codella, M. Liu, X. Dai, L. Yuan, and L. Zhang, "Cvt: Introducing convolutions to vision transformers," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 22–31, 2021.

[19] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*. Boca Raton, FL: A K Peters/CRC Press, 4th ed., 2018.

[20] J. de Vries, "Forward and deferred rendering." `https://learnopengl.com/Advanced-Lighting/Deferred-Shading`, 2024.

[21] NVIDIA, "Fxaa white paper." `https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf`.

[22] L. Yang, S. Liu, and M. Salvi, "A survey of temporal antialiasing techniques," in *Computer graphics forum*, vol. 39, pp. 607–621, Wiley Online Library, 2020.

[23] G. Engine, "Resolution scaling." `https://docs.godotengine.org/en/stable/tutorials/3d/resolution_scaling.html`. Accessed: 2024-08-06.

[24] G. Deng, "A generalized unsharp masking algorithm," *IEEE transactions on Image Processing*, vol. 20, no. 5, pp. 1249–1261, 2010.

[25] S. Winkler, *Digital video quality: vision models and metrics*. John Wiley & Sons, 2005.

[26] R. C. Gonzalez, *Digital image processing*. Pearson education india, 2009.

[27] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[28] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang, "The unreasonable effectiveness of deep features as a perceptual metric," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 586–595, 2018.

[29] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[30] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1874–1883, 2016.

[31] A. Howard, "Mobilenets: Efficient convolu-tional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[32] A. Vaswani, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[33] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[34] X. Ding, X. Zhang, N. Ma, J. Han, G. Ding, and J. Sun, "Repvgg: Making vgg-style convnets great again," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 13733–13742, 2021.

[35] P. K. A. Vasu, J. Gabriel, J. Zhu, O. Tuzel, and A. Ranjan, "Fastvit: A fast hybrid vision transformer using structural reparameterization," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 5785–5795, 2023.

[36] Z. Huang, A. Huang, X. Hu, C. Hu, J. Xu, and S. Zhou, "Scale-adaptive feature aggregation for efficient space-time video super-resolution," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 4228–4239, 2024.

[37] E. Games, "Cinematic render passes in unreal engine." `https://dev.epicgames.com/documentation/en-us/unreal-engine/cinematic-render-passes-in-unreal-engine`, 2024. Accessed: 2024-08-09.

[38] A. Ranjan and M. J. Black, "Optical flow estimation using a spatial pyramid network," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[39] Z. Wu, C. Zuo, Y. Huo, Y. Yuan, Y. Peng, G. Pu, R. Wang, and H. Bao, "Adaptive recurrent frame prediction with learnable motion vectors," in *SIGGRAPH Asia 2023 Conference Papers*, pp. 1–11, 2023.

[40] J. Guo, X. Fu, L. Lin, H. Ma, Y. Guo, S. Liu, and L.-Q. Yan, "Extranet: real-time extrapolated rendering for low-latency temporal supersampling," *ACM Transactions on Graphics (TOG)*, vol. 40, no. 6, pp. 1–16, 2021.

[41] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W.-c. Woo, "Convolutional lstm network: A machine learning approach for precipitation nowcasting," *Advances in neural information processing systems*, vol. 28, 2015.

[42] E. Agustsson and R. Timofte, "Ntire 2017 challenge on single image super-resolution: Dataset and study," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pp. 126–135, 2017.

[43] S. Nah, S. Baik, S. Hong, G. Moon, S. Son, R. Timofte, and K. M. Lee, "Ntire 2019 challenge on video deblurring and super-resolution: Dataset and study," in *CVPR Workshops*, June 2019.

[44] B. Entertainment, "StarCraft II." `https://starcraft2.com/`, 2010.

[45] id Software and B. Softworks, "Doom Eternal." `https://bethesda.net/en/game/doom`, 2020.

[46] I. FromSoftware and B. N. Entertainment, "Elden Ring." `https://en.bandainamcoent.eu/elden-ring/elden-ring`, 2022.

[47] B. Entertainment, "Overwatch 2." `https://overwatch.blizzard.com/en-us/`, 2022.

[48] E. Games, "Unreal Engine Sequencer Movie Tool Overview." `https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-sequencer-movie-tool-overview`, 2024.

[49] E. Games, "Take Recorder in Unreal Engine." `https://dev.epicgames.com/documentation/en-us/unreal-engine/take-recorder-in-unreal-engine?application_version=5.4`, 2024.

[50] E. Games, "Render Cinematics in Unreal Engine." `https://dev.epicgames.com/documentation/en-us/unreal-engine/render-cinematics-in-unreal-engine?application_version=5.4`, 2024.

[51] E. Games, "Unreal engine color pipeline opencolorio." https://dev.epicgames.com/community/learning/courses/qEl/unreal-engine-technical-guide-to-linear-content-creation-pipeline-development/KJZk/unreal-engine-color-pipeline-opencolorio, 2024. Accessed: 2024-08-09.

[52] E. Games, "Plugins in unreal engine." https://dev.epicgames.com/documentation/en-us/unreal-engine/plugins-in-unreal-engine?application_version=5.4, 2024. Accessed: 2024-08-09.

[53] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[54] J. Yu, Z. Lin, J. Yang, X. Shen, X. Lu, and T. S. Huang, "Free-form image inpainting with gated convolution," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 4471–4480, 2019.

[55] D. P. Kingma, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[56] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[57] X. Wang, L. Xie, K. Yu, K. C. Chan, C. C. Loy, and C. Dong, "BasicSR: Open source image and video restoration toolbox." https://github.com/XPixelGroup/BasicSR, 2022.

[58] PyTorch, "TensorBoard with PyTorch." https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html, n.d.

[59] S. Mehta, A. Kumar, F. Reda, V. Nasery, V. Mulukutla, R. Ranjan, and V. Chandra, "Evrnet: Efficient video restoration on edge devices," in *Proceedings of the 29th ACM international conference on multimedia*, pp. 983–992, 2021.

[60] E. Games, "Physics in Unreal Engine." https://dev.epicgames.com/documentation/en-us/unreal-engine/physics-in-unreal-engine, 2024.

[61] E. Games, "Cinematic Rendering Image Quality Settings in Unreal Engine." https://dev.epicgames.com/documentation/en-us/unreal-engine/cinematic-rendering-image-quality-settings-in-unreal-engine#anti-aliasing, 2024.

[62] T. Kalluri, D. Pathak, M. Chandraker, and D. Tran, "Flavr: Flow-agnostic video representations for fast frame interpolation," in *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pp. 2071–2082, 2023.

[63] H. C. Lin and A. Burnes, "Nvidia dlss 3." https://www.nvidia.com/en-us/geforce/news/dlss3-ai-powered-neural-graphics-innovations/.

[64] A. G. Open, "Amd fidelityfx™ super resolution 3 is now available in games." https://gpuopen.com/fsr3-in-games-technical-details.

A Appendix

A.1 Step-by-step guide for obtaining rendering data in Unreal Engine 5.4

In this appendix we want to illustrate the necessary steps to reproduce or generate completely new rendered data with Unreal Engine 5.4.

Generate a new project through the Epic Game Launcher. Unzip our GenerateData.zip file and put the folder into your projects directory under the Plugins folder. If no Plugins folder is present, simply generate one.

Open the project in Unreal Engine and navigate to the Plugins window, this can be found under *Edit* > *Plugins*. You should see a list of plugins, that can be installed or activated. Install the Movie Render Queue plugin and activate our GenerateData plugin, it should look like figure A.1.

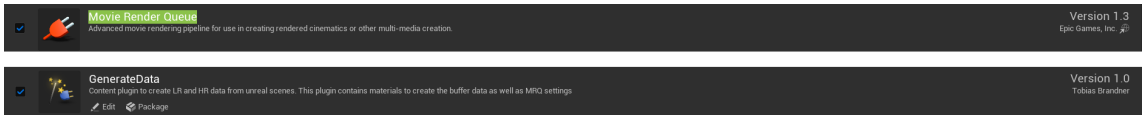

Figure A.1: The Movie Render Queue plugin as well as our GenerateData plugin should be listed in the Plugins window inside the Unreal project.

Once the plugins are active, the engine might ask to restart the project. As next, we describe the steps to generate the rendered data for an overview camera shot:

1. First add a Level Sequence to your active Level.
2. Place a camera actor through the Place Actor window in the scene and drag it into the Sequencer window.
3. The camera actor should now be tracked in the Sequencer window's timeline. Take posession of the camera and key frame the camera's transform over the sequence's timeline. Note: Move the time indicator else the key frame is always set at the same time spot.
4. Once your sequence is finished, save it and move to the Movie Render Queue window (found under *Windows* > *Sequencer*) and open it. Add one of the MRQ presets we provide, see figure A.3. Apply the setting and render the data locally. A window

should pop up, giving you a preview of how the render will look like, an example can be seen in figure A.4.

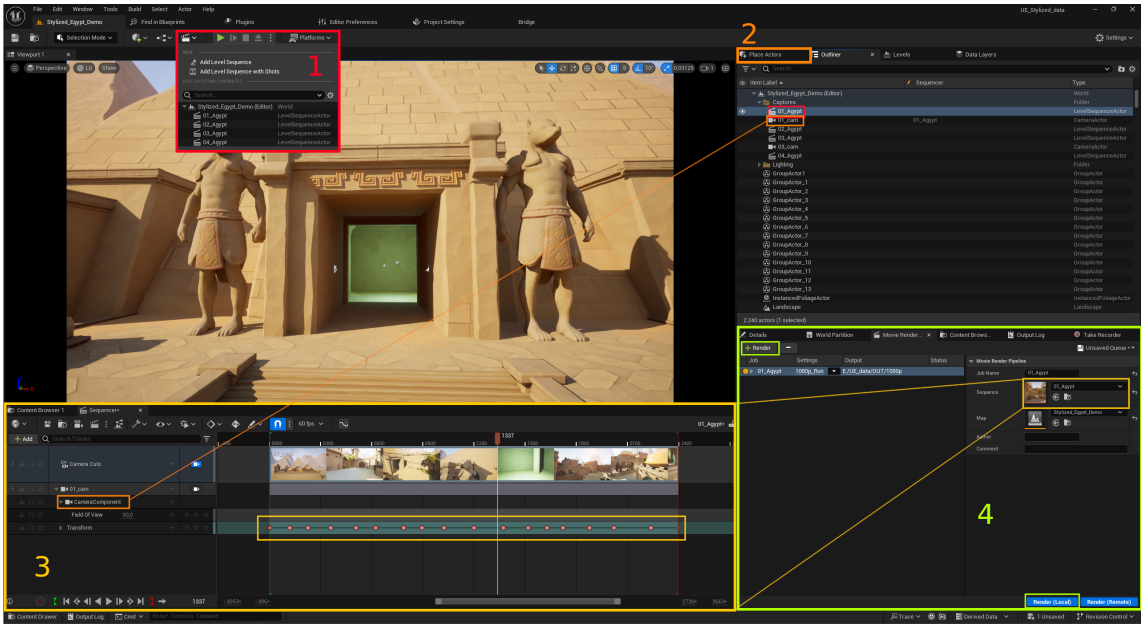

Figure A.2: Step-by-step guide to generate an overview camera shot. There are four main steps to generate the data: (1) Add a level sequence (2) Place a camera actor in the scene and drag it into the sequencer window (3) Control the camera and key frame the camera's transform over the sequence's timeline (4) In the Movie Render Queue window open the level sequence, add a configuration from our plugin and render the data

A visual representation of this description can be seen in figure A.2. Be careful to set the output directory for the external .png and .exr files are saved somewhere you have enough space.

Generating the third person camera data follows a similiar approach, again with a visual representation displayed in figure A.5:

1. Add a Level Sequence element to the active scene.
2. Switch into the Take Recorder Window (found under *Windows* > *Sequencer*) and open the newly created Level Sequence.
3. Click on the Source button and scroll to the Player preset. This will track the character controlled by the input of the player.
4. Start the recording. The level should now start and give you control over the main character in the scene. For the time of the sequence the character and your input will be recorded. The recording will automatically be saved inside the Level Sequence.

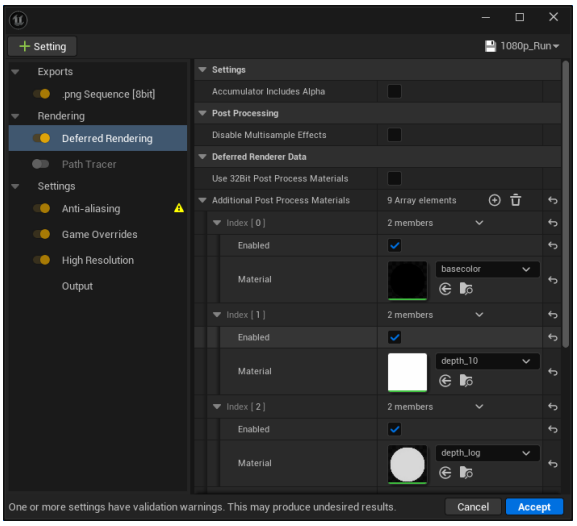

Figure A.3: Movie Render Pipeline settings we provide through our custom plugin GenerateData. In the Deferred Render setting are our custom material shaders applied to additionally render the auxiliary G-buffer data.

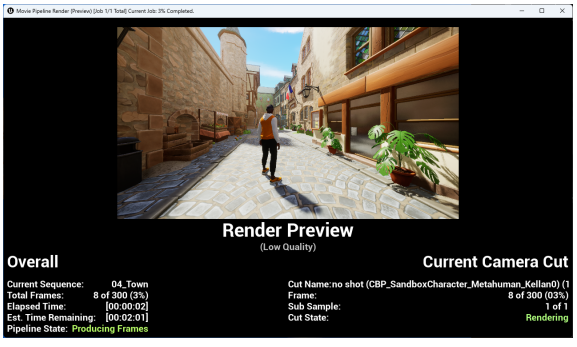

Figure A.4: Example of how a successful render of our dataset. A popup window displays a preview of how the rendered sequence will look like.

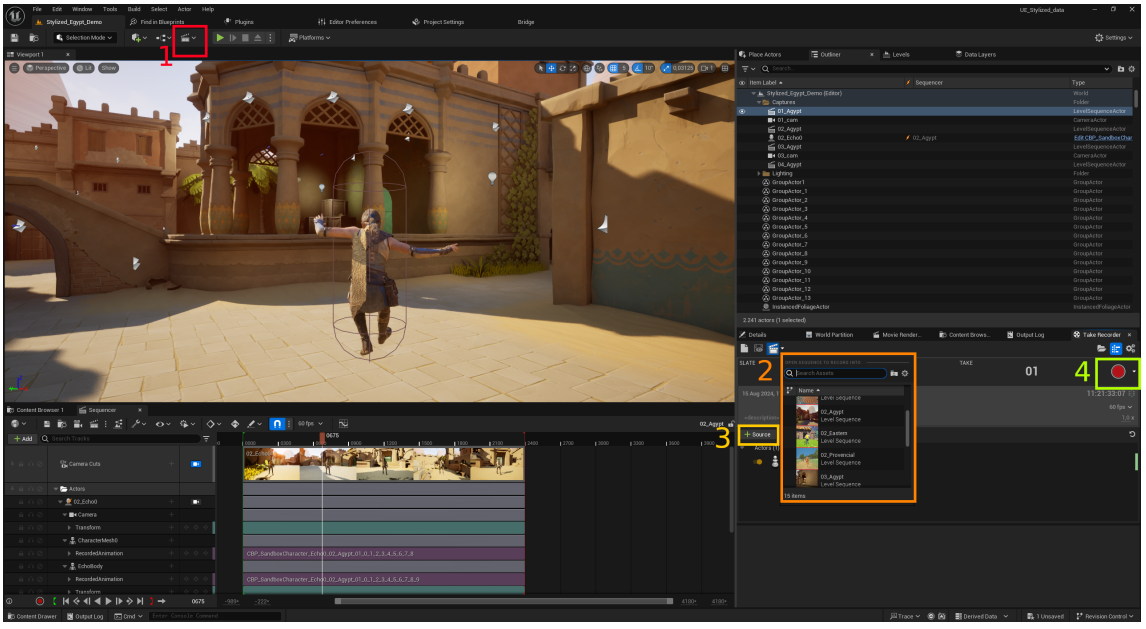

Figure A.5: Step-by-step guide to generate a third-person camera shot. There are four main steps to generate the data: (1) Add a level sequence (2) Move into the Take Recorder window. Open the created Level Sequence. (3) Add the Player as source for the recording (4) Start the recording

A.2 Implementation details

Here are some additional information provided for section 4.3.

A.2.1 YAML configuration files

```
MODEL: STSS_Original
EPOCHS: 100
SCALE: 2
BATCH_SIZE: 4
CROP_SIZE: 256
USE_HFLIP: True
USE_ROTATION: True
NUMBER_WORKERS: 8
LEARNING_RATE: 0.001
CRITERION: STSSLoss
OPTIMIZER:
  NAME: Adam
  BETA1: 0.9
  BETA2: 0.99
SCHEDULER:
  NAME: StepLR
  STEP_SIZE: 50
  GAMMA: 0.9
  START_DECAY_EPOCH: 20
DATASET:
  NAME: UE_data
  SEQUENCE: all
  HISTORY: 2
  WARP: True
  BUFFERS:
    BASE_COLOR: True
    DEPTH: True
    METALLIC: True
    NOV: True
    ROUGHNESS: True
    WORLD_NORMAL: False
    WORLD_POSITION: False
```

A.2.2 Modular network files

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
URepSS_04                                [1, 3, 3840, 2160]        --
|- PixelUnshuffle: 1-1                   [1, 32, 960, 540]         --
|- GatedConvBlock: 1-2                   [1, 24, 960, 540]         --
|   |- GatedConv: 2-1                    [1, 24, 960, 540]         --
|   |   |- Conv2d: 3-1                   [1, 24, 960, 540]         6,936
|   |   |- Sequential: 3-2               [1, 1, 960, 540]          289
|   |- ReLU: 2-2                         [1, 24, 960, 540]         --
|   |- GatedConv: 2-3                    [1, 24, 960, 540]         --
|   |   |- Conv2d: 3-3                   [1, 24, 960, 540]         5,208
|   |   |- Sequential: 3-4               [1, 1, 960, 540]          217
|   |- ReLU: 2-4                         [1, 24, 960, 540]         --
|- GatedDownConvBlock: 1-3               [1, 24, 480, 270]         --
|   |- GatedConv: 2-5                    [1, 24, 480, 270]         --
|   |   |- Conv2d: 3-5                   [1, 24, 480, 270]         5,208
|   |   |- Sequential: 3-6               [1, 1, 480, 270]          217
|   |- GatedConv: 2-6                    [1, 24, 480, 270]         --
|   |   |- Conv2d: 3-7                   [1, 24, 480, 270]         5,208
|   |   |- Sequential: 3-8               [1, 1, 480, 270]          217
|   |- ReLU: 2-7                         [1, 24, 480, 270]         --
|   |- GatedConv: 2-8                    [1, 24, 480, 270]         --
|   |   |- Conv2d: 3-9                   [1, 24, 480, 270]         5,208
|   |   |- Sequential: 3-10              [1, 1, 480, 270]          217
|   |- ReLU: 2-9                         [1, 24, 480, 270]         --
|- GatedDownConvBlock: 1-4               [1, 32, 240, 135]         --
|   |- GatedConv: 2-10                   [1, 24, 240, 135]         --
|   |   |- Conv2d: 3-11                  [1, 24, 240, 135]         5,208
|   |   |- Sequential: 3-12              [1, 1, 240, 135]          217
|   |- GatedConv: 2-11                   [1, 32, 240, 135]         --
|   |   |- Conv2d: 3-13                  [1, 32, 240, 135]         6,944
|   |   |- Sequential: 3-14              [1, 1, 240, 135]          217
|   |- ReLU: 2-12                        [1, 32, 240, 135]         --
|   |- GatedConv: 2-13                   [1, 32, 240, 135]         --
|   |   |- Conv2d: 3-15                  [1, 32, 240, 135]         9,248
|   |   |- Sequential: 3-16              [1, 1, 240, 135]          289
|   |- ReLU: 2-14                        [1, 32, 240, 135]         --
|- Sequential: 1-5                       [1, 32, 240, 135]         --
|   |- PixelUnshuffle: 2-15              [1, 24, 960, 540]         --
|   |- GatedConvBlock: 2-16              [1, 24, 960, 540]         --
|   |   |- GatedConv: 3-17               [1, 24, 960, 540]         5,425
|   |   |- ReLU: 3-18                    [1, 24, 960, 540]         --
|   |   |- GatedConv: 3-19               [1, 24, 960, 540]         5,425
|   |   |- ReLU: 3-20                    [1, 24, 960, 540]         --
|   |- GatedDownConvBlock: 2-17          [1, 24, 480, 270]         --
|   |   |- GatedConv: 3-21               [1, 24, 480, 270]         5,425
|   |   |- GatedConv: 3-22               [1, 24, 480, 270]         5,425
|   |   |- ReLU: 3-23                    [1, 24, 480, 270]         --
|   |   |- GatedConv: 3-24               [1, 24, 480, 270]         5,425
|   |   |- ReLU: 3-25                    [1, 24, 480, 270]         --
```

```
|   |- GatedDownConvBlock: 2-18          [1, 32, 240, 135]         --
|   |   |- GatedConv: 3-26               [1, 24, 240, 135]         5,425
|   |   |- GatedConv: 3-27               [1, 32, 240, 135]         7,161
|   |   |- ReLU: 3-28                    [1, 32, 240, 135]         --
|   |   |- GatedConv: 3-29               [1, 32, 240, 135]         9,537
|   |   |- ReLU: 3-30                    [1, 32, 240, 135]         --
|- Attention: 1-6                        [1, 64, 240, 135]         8
|   |- Conv2d: 2-19                      [1, 192, 240, 135]        12,480
|   |- Conv2d: 2-20                      [1, 192, 240, 135]        1,920
|   |- Conv2d: 2-21                      [1, 64, 240, 135]         4,160
|- UpConvBlock: 1-7                      [1, 32, 480, 270]         --
|   |- Upsample: 2-22                    [1, 64, 480, 270]         --
|   |- Sequential: 2-23                  [1, 32, 480, 270]         --
|   |   |- Conv2d: 3-31                  [1, 44, 480, 270]         34,892
|   |   |- ReLU: 3-32                    [1, 44, 480, 270]         --
|   |   |- Conv2d: 3-33                  [1, 32, 480, 270]         12,704
|   |   |- ReLU: 3-34                    [1, 32, 480, 270]         --
|- UpConvBlock: 1-8                      [1, 24, 960, 540]         --
|   |- Upsample: 2-24                    [1, 32, 960, 540]         --
|   |- Sequential: 2-25                  [1, 24, 960, 540]         --
|   |   |- Conv2d: 3-35                  [1, 28, 960, 540]         14,140
|   |   |- ReLU: 3-36                    [1, 28, 960, 540]         --
|   |   |- Conv2d: 3-37                  [1, 24, 960, 540]         6,072
|   |   |- ReLU: 3-38                    [1, 24, 960, 540]         --
|- GatedConvBlock: 1-9                   [1, 48, 960, 540]         --
|   |- GatedConv: 2-26                   [1, 48, 960, 540]         --
|   |   |- Conv2d: 3-39                  [1, 48, 960, 540]         10,416
|   |   |- Sequential: 3-40              [1, 1, 960, 540]          217
|   |- ReLU: 2-27                        [1, 48, 960, 540]         --
|   |- GatedConv: 2-28                   [1, 48, 960, 540]         --
|   |   |- Conv2d: 3-41                  [1, 48, 960, 540]         20,784
|   |   |- Sequential: 3-42              [1, 1, 960, 540]          433
|   |- ReLU: 2-29                        [1, 48, 960, 540]         --
|- PixelShuffle: 1-10                    [1, 3, 3840, 2160]        --
==========================================================================
Total params: 218,522
Trainable params: 218,522
Non-trainable params: 0
Total mult-adds (G): 51.59
==========================================================================
Input size (MB): 116.12
Forward/backward pass size (MB): 1434.41
Params size (MB): 0.87
Estimated Total Size (MB): 1551.41
==========================================================================
Warm up ...
Start timing ...
Average forward pass time 11.07 ms
Memory allocated (peak): 709.1767578125 MB
```

A.2.3 Tensorboard for Pytorch

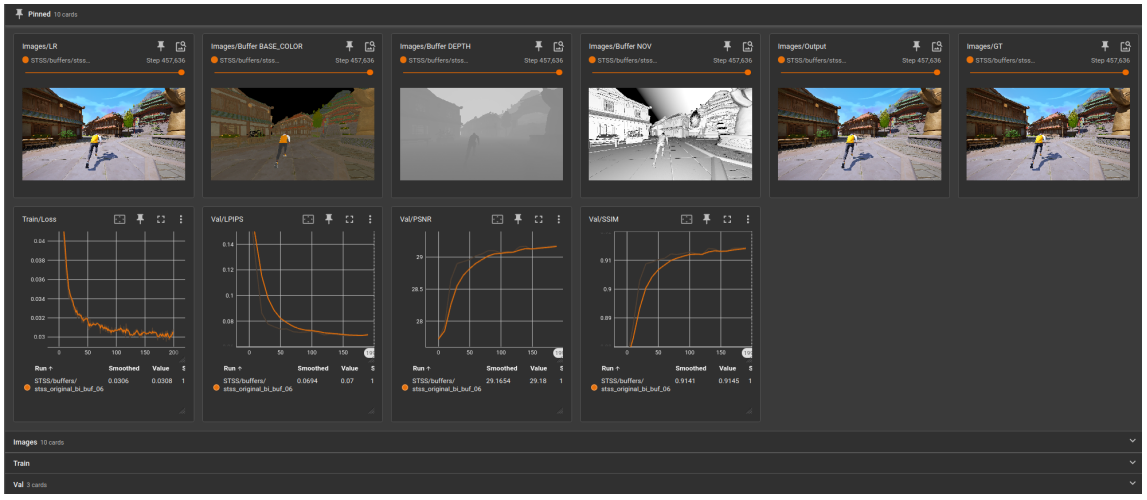

Figure A.6: Tensorboard tracks our training loss, the input of our network, like the auxiliary buffers, and the scores for PSNR, SSIM, LPIPS.

(Don't forget the Versicherung zur Leistungserbringung, which is strictly required by the Pruefungsamt. Download and fill out the latest version from the Uni Wuerzburg Homepage (uni-wuerzburg.de). Go to Studium -> Pruefungsangelegenheiten -> Formulare -> Abschlussarbeiten / Hausarbeiten -> Versicherung zur Leistungserbringung. Maybe ask your supervisor for help.) ToDo