Reflection probe interpolation

Jakob Petek

Faculty of information and computer science, University of Ljubljana

Abstract
Realistic reflection rendering plays a crucial role in enhancing the visual quality of interactive applications such as video games and virtual reality experiences. Traditionally, the use of precomputed reflection probes with baked images has been a popular approach to capture static environments. However, dynamically changing scenes pose a challenge for real-time reflection updates, demanding efficient techniques that strike a balance between visual fidelity and computational cost.
In this paper, we propose a method for enhancing quality of traditional reflection capturing method that uses Delaunay triangulation and a neural network. We introduce a pipeline that could potentially be used to enhance interpolation images of the captured environment.
To evaluate the performance and visual quality of our method, we conducted a series of experiments in Python as well as suggest one possible way of incorporating the pipeline in Unreal Engine 5. The results showcase the potential of using the suggested pipeline as a possible replacement or an enhancement of the traditional reflection probe linear interpolation method.

Keywords: reflection probe capture, real-time rendering, triangulation, neural network autoencoder

1. Introduction

In real time computer graphics applications achieving realistic and visually appealing reflective materials is a challenging task. Reflective materials play a crucial role in creating immersive environments and adding depth to virtual scenes. While today's dynamic reflections such as ray tracing or Lumen reflections [Gam23a] (Unreal Engine's new global illumination system) can already be seen in certain applications and tech demos, for the most part, are not yet used, due to the need for high-end hardware. These algorithms work by tracing rays of light to simulate the indirect bounce of light within a scene, which is computationally expensive, as a large amount of rays are needed for accurate results. Therefore in the past and today different approaches are used in order to simulate reflective materials. One of the more simple ones is, so called, reflection capture probe.

In this seminar we take a look at how these probes work in order to capture reflections and display them on materials and a possible way of improving their quality by using a neural network to improve interpolation results.

2. Related work

Besides the already stated ray tracing algorithms and the static reflection probe capture, which will be explained in the following, there are several other methods used for capturing or computing reflections. Two such worth mentioning are screen space reflections and planar reflections, which are both dynamic but generally less computational heavy [Gam23b].

Screen space reflection (SSR) is a technique for reusing screen space data to calculate reflections and is commonly used to create more subtle reflections such as on wet floor surfaces or in puddles. It is a dynamic post process effect that approximates reflections and is limited to only reflecting what is visible on the screen. This is also its greatest weakness, as objects off screen or objects that are occluded by other objects will not be represented in SSR, which can cause rendering artifacts in the reflections.

One other approach usually used as replacement/improvement for SSR is planar reflection, which comes at a higher rendering cost. It works by rendering the scene again from the direction of the reflection either in real time (each frame) or pre-computed and projecting it back on the object or surface. Planar reflections can reflect objects that are off-screen regardless of camera view.

3. Methodology

However, if we cannot afford calculating reflections at run-time then reflection capturing probes are a simple and cheap solution to our problem. It works by capturing a cubemap from the location in the scene and bakes it onto an image. That image is then (same as planar reflection) used and projected on surface of an object. The obvious issue is that having only one capture probe is not enough for the whole scene, especially if we have objects that reflect environment and can be moved at run-time. Its material therefore must be updated each frame, which would mean that we would have to capture an image at every frame and project it back to the object. Since that would be too computationally heavy, we instead capture multiple images at multiple locations and, based on current object location, linearly interpolate two (Figure 1 and Figure 2) of the closest captured images to generate a new image. A simple example is shown below.

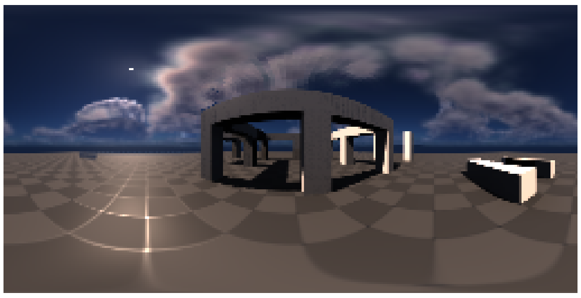
Figure 1: Example of capturing an image in a game level at location 1.

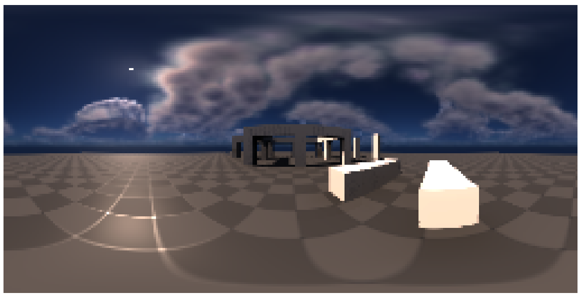
Figure 2: Example of capturing an image in a game level at location 2.

The following result (Figure 3) is not of best quality but often sufficient, given its limitations.

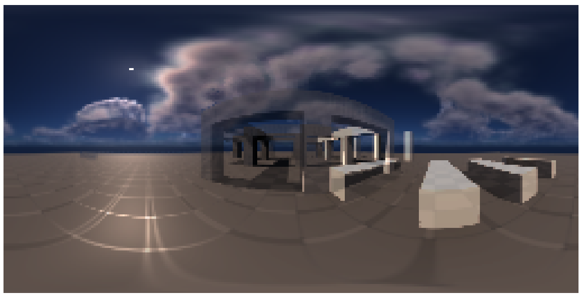
Figure 3: Generating a linear interpolated image from images above during run-time, based on object location. The location of the image is set as halfway between the two.

The algorithm is quite straightforward as the current pixel in interpolated image is the weighted sum of pixels from the other two images. The weights (0 - 1) are set based on the distances to the other two reflection probes.

4. Experiments

In order to improve the interpolation results, we expended the traditional pipeline and introduced two general improvements.

4.1. Bi linear interpolation and Delaunay triangulation

Instead of linearly interpolating between two closest reflection probes we instead used a bi linear method and Delaunay triangulation. The idea is to create a triangulation grid of reflection probes so that based on current object location, we interpolate, not with the 2 closest reflection probes, but instead, the 3 reflection probes that form a triangle in which the object currently lays. Furthermore, instead of using linear interpolation we use bi linear interpolation which is basically same as linear interpolation, but it uses 3 images instead of 2. The algorithm for calculating pixel value is otherwise the same.

4.2. Autoencoder neural network

Since bi linear interpolation is not a good enough improvement we decided to expand the pipeline by using an autoencoder neural network, whose job is to clean up the bi linear image and generate a smooth image representing the actual scene capture in the desired location.

Although there are some options when it comes to picking up the neural network architecture fit for the required job, like a conditional generative adversarial network (CGAN) we ultimately chose an autoencoder, mainly due to hardware limitations.

An autoencoder is an unsupervised learning neural network architecture which aims to learn a compressed representation of the input data and then reconstructs the original data from this compressed representation [GBC16]. The autoencoder architecture consists of an encoder network that maps the input data to a lower dimensional representation, and a decoder network that reconstructs the input data from the encoded representation.

The purpose of an autoencoder is to capture the most important features or patterns in the data, effectively learning a compressed representation that retains useful information. By reconstructing the input data from this compressed representation, the autoencoder is encouraged to capture and preserve the salient features of the data. It is used in many applications such as image noise removal, anomaly detection and feature extraction, to name a few.

5. Results and discussion

Given that this pipeline would be used in real time application, we have decided to use Unreal Engine 5 (hereinafter UE5) as our gateway for rendering and Python for model training, as training in UE5 is not yet possible.

5.1. Data acquisition

The data required to train the model has been captured in UE5 and exported using the UE5 Python API as exporting them using blueprints or C++, was too tedious. We have created a simple level consisting of primitive geometry (mostly boxes) (Figure 4) and captured the images using the UE5 built-in scene capture cube actor, which as name suggests captures the environment in a cubemap and renders it to a render target, which can then be used to export the image as .hdr. Using a Python script we were able to move the scene capture cube actor and capture hdr images in 128x256 resolution across the whole level as well as their captured locations (x and y). The total amount of captured images was 4900 or 70x70 images per axis. The z axis was set to 170 cm and did not change. The reason the image quality was set so low, was due to hardware limitations for model traning.

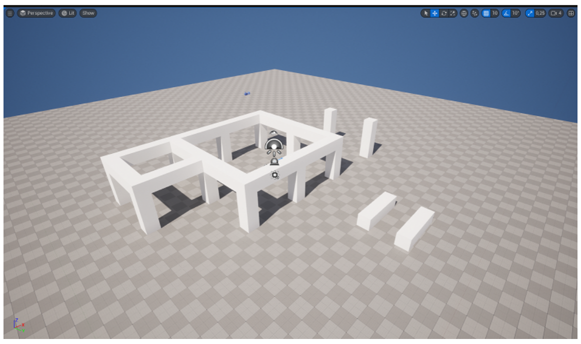

Figure 4: View of the level used to train the neural network in Unreal Engine's editor.

5.2. Preprocess data

Captured images along with their locations were then imported to Python were we started the preprocessing step. Here we did all the necessary steps needed for model training. Mainly, that comes down to changing resolution of all images to 256x256, normalizing pixel value from 8 bit unsigned integer to float values in range from 0 to 1 and generating reflection probe triangulation as well as generating bi linear images, which would be our input to the neural network (train data).

Generating a triangular grid was fairly straightforward. We have decided to take 10 of the captured images as our reflection sphere captures. This means that using these 10 images we have to create all the rest of 4900 images using bi linear interpolation. Figure 5 shows image locations used as reflection captures (red color) and in blue color all the images captured in a level. Since the plot is small the step between image locations can't be seen.

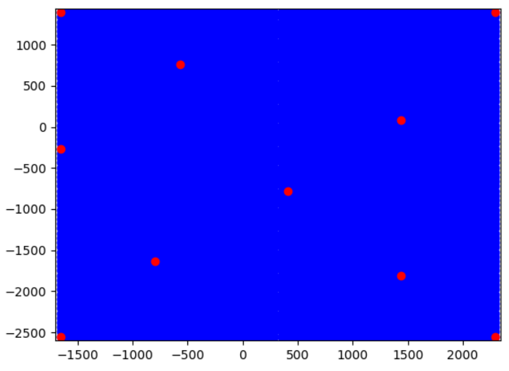

Figure 5: View on the level from top side in Python. The red circles represent reflection capture images, while the blue represent all images captured inside the engine.

Using these 10 positions we then made the triangulation so that we could calculate a new image at any location based off the three reference images that form a triangle (Figure 6).

At this stage we were able to generate 4900 bi linear images which we then feed to neural network along with our ground truths, captured in the engine.

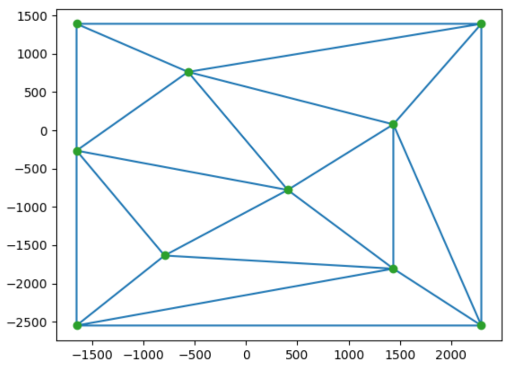

Figure 6: Level triangulation using 10 reflection captures.

5.3. Model training

The overall architecture of the autoencoder is fairly simple consisting of 14 layers. The encoder gradually downsamples the input image through convolutional and max pooling layers and then the decoder, does the opposite, by upsampling the image back to the original size using upsampling and convolutional layers. The mean squared error is used as loss function in regression. It calculates the average of the squared differences between the predicted pixel values and the actual pixel values. For the optimization algorithm adaptive moment estimation was used.

Using Tensorflow and Keras libraries we firstly attempted to train the model on the whole database, but ultimately failed as we ran out of VRAM memory on the GPU, so we have instead split the data into two separate batches, however we soon realized that the training would take too much time so we reduced the data to 100 images.

Realizing that, that would probably not be enough data for the whole level we instead picked only 70 images across the center of the level and trained the model on center only. The training time was not recorded but took at least 10 hours with at least 50.000 epochs with batch size of 8 images. We stopped the training after the loss function did not improve in the last 2 hours. The final result was an accuracy of around 95%.

5.4. Port to Unreal Engine

Porting the trained model to UE5 is not as straightforward, since it requires the model to be in ONNX format. Unfortunately the library version for CUDA tensorflow does not support the conversion to ONNX, however we were able to convert it by creating a new Python environment with older tensorflow version. Regardless of that we were unable to run the model in real time in UE5.

The reason we weren't able to run the model was not because of the fact that machine learning models in UE5 are still in beta and can cause crashes but it was the way cubemaps are stored in Unreal.

For us to be able to run the model we would have to either change the whole architecture of the model and data used for training or perform conversion from cubemaps to equirectangular Images and back to cubemaps. Both of the options are out of scope for this seminar. The main problem is that cubemaps are stored as 6 cube images instead of one equirectangular image. The model we trained used images that are equirectangular, which is essentially a rectangular projection of a spherical image (Figure 7). But accessing UE5 cubemaps, we can only get pixel data for each of the 6 images. Therefore we would firstly have to convert the cubemap to equirectangular shape, feed it to the model and then convert the output of the model back to cubemap. Under the hood Unreal Engine does convert the cubemap to equirectangular image, however that is not accessible as it is most likely part of the rendering pipeline, which, in order to access, would require building the engine from source.

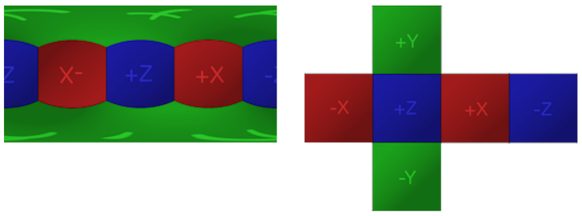

Figure 7: Difference between cubemap and equirectangular projection.

For these reasons we didn't test our model in UE5.

5.5. Final results

Since we weren't able to evaluate the model in Unreal engine, we have instead evaluated it in Python in two ways: comparing it to an image it wasn't trained on (Figure 8) and creating a sequence imitating a result we would get in Unreal Engine (GIF is available on GitHub [Pet23]).

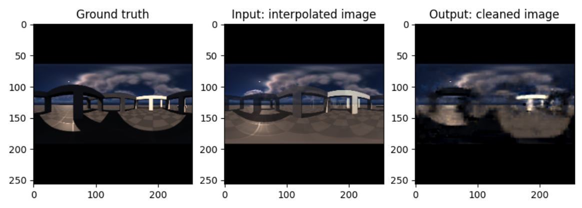

Figure 8: Model evauation on an image it wasn't trained on. We can see that the general shape was captured however the finer detail was not.

As we can see from figure 8 the output image from the model is not so clean, however the model does manage to add shadows that weren't in the input image. Most of the finer details, like box edges, sky and tiles are blurred, but the general shape is somewhat captured.

Similar results can be seen on .gif animation that is available on GitHub. On the animation we can also see another problem seen as abrupt cutting, which is most likely the effect of moving from one triangle to another, and with that, using different reflection probe images.

6. Conclusion

In my opinion the created images (and with it the suggested pipeline) is not suitable for real time applications, since generally the neural networks are slow and compete for GPU resources during run time. On top of that the results are on par at best and overall aren't better then just plain simple interpolation (comparing the sequence) as we generally get quite noisy output data.

These problems could be improved with perhaps a longer train time, bigger database or different layers/parameters in the autoencoder. Perhaps different neural network architecture would give us better results or a different image merging method. Moreover results given by the autoencoder loos function could indicate that the model might simply be unable to learn what we are trying to teach it.

In conclusion, while our proposed pipeline may not currently be the optimal solution for real-time applications, the existence of opportunities for improvement underscores the significance of neural networks in this domain. Our research perhaps serves as a foundation for future advancements, encouraging the exploration of novel techniques and approaches that can harness the power of neural networks to elevate the visual quality and realism of real-time reflections in interactive applications.

References

[Gam23a] GAMES E.: *Lumen Global Illumination and Reflections*. Retrived 20.5.2023. `https://docs.unrealengine.com/5.0/en-US/lumen-global-illumination-and-reflections-in-unreal-engine/`. 1

[Gam23b] GAMES E.: *Reflections Environment*. Retrived 20.5.2023. `https://docs.unrealengine.com/5.0/en-US/reflections-environment-in-unreal-engine/`. 1

[GBC16] GOODFELLOW I., BENGIO Y., COURVILLE A.: *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. 3

[Pet23] PETEK J.: *GitHub repository*. Retrived 20.5.2023. `https://github.com/Friday202/ReflectionProbeInterpolation`. 4