Realistic plant simulation

Research document

Glyn Leine	445021
Maurijn Besters	462030
Robin Dittrich	462270
Rowan Ramsey	459575

Table of contents

Table of contents 2

Introduction 3

1. How can we efficiently, realistically, and procedurally generate plants to populate a field in our simulation? 4

1.1. SideFX Houdini 4

1.2. L-systems 4

2. How can we render the scene realistically and efficiently? 5

2.1. Path tracing 5

2.2. Rasterization + Traditional Effects 5

2.3. Rasterization + Raytraced Effects 6

2.4. Optix denoiser 6

3. How can we simulate camera sensors in the simulation and how do we use the data they create? 7

3.1. Unity render streaming 7

3.2. FMETP Stream 7

3.3. FFmpegOut 8

3.4. Development 8

3.5. How does the streaming work 8

3.6. Simulation of camera properties 9

4. How do we facilitate the communication between Unity and ROS? 10

4.1. Unitys TCP Server Solution 10

4.2. ROS# 10

4.3. Development 11

5. How could our robot affect the simulated plants? 12

5.1. Nvidia Flex 12

5.2. Unity Cloth 13

5.3. Obi for Unity 13

5.4. Custom solution 13

6. How do we simulate a realistic looking field? 14

6.1. Mesh baking 14

6.2. Tessellation 14

6.2.1. Height mapping 14

6.2.1.a. Precomputed heightmap 14

6.2.1.b. Runtime heightmap calculation 15

6.2.1.c. Hybrid of runtime and precomputed 15

6.2.2. Interaction 15

6.2.2.a. Per frame canvas drawing 15

6.2.2.b. Spline memory track drawing 15

Bibliography 16

Introduction

This document contains all of the research we’ve done for the Realistic plant simulation project given to us by Saxion as part of the minor Immersive Media / Project IMT&S.

1. How can we efficiently, realistically, and procedurally generate plants to populate a field in our simulation?

For this project having a “random” and variable data set in our plant population is key to creating realistic data. Procedural plant generation provides the user full control over the age, size, and color of an individual plant, which is ideal for experimenting in different scenarios

1.1. SideFX Houdini

Houdini is a 3D animation software application developed by Toronto-based SideFX, who adapted it from the PRISMS suite of procedural generation software tools. Houdini's exclusive attention to procedural generation distinguishes it from other 3D computer graphics software.
(Wikipedia contributors, 2021)

We use houdini as a means of generating lots of different types of plants. Houdini has native support for L-systems. This means that we can easily create lots of different variations of the same plant while only modifying a few parameters.

Houdini has developed a plugin for Unity which allows us to easily interact with the houdini asset(HDA files) files. With this plugin we are able to import the HDA files and then modify the parameters from Unity. This means we can automate the process of generating plants, easing the user experience.

1.2. L-systems

L-systems (Lindenmayer-systems, named after Aristid Lindenmayer, 1925-1989), allow definition of complex shapes through the use of iteration. They use a mathematical language in which an initial string of characters is matched against rules which are evaluated repeatedly, and the results are used to generate geometry. The result of each evaluation becomes the basis for the next iteration of geometry, giving the illusion of growth.
(SideFX, n.d.)

We use L-systems with our project to generate many variations of the same plant. We can modify different values to change the age, height, width, color and/or the amount of buds/fruits of a plant. With help from an external consultant we were able to create some initial procedural meshes that fit our current needs very well.

2. How can we render the scene realistically and efficiently?

In order to supply the virtual camera sensors with realistic data we need to be able to render the simulation as realistically as possible for each camera. In order to render multiple cameras and still stay within our soft real-time latency budget our rendering also needs to be incredibly efficient and performant.

A perfect solution for our problem doesn’t exist yet, so we had a look at several approaches and their advantages and drawbacks.

2.1. Path tracing

Path tracing is a method for generating digital images by simulating how light would interact with objects in a virtual world. The path of light is traced by shooting rays (line segments) into the scene and tracking them as they bounce between objects. (Walt Disney Animation Studios, 2016)

Path tracing allows us to practically simulate the complete interaction of light with the scene. This makes path traced images hyper realistic in their lighting. Path tracing also allows us to simulate lighting effects that are simply impossible using other approaches. The major drawback of path tracing however is that it is incredibly costly. The math for checking collision between 3D oriented rays and 3D oriented triangles is much more complex and costly than rasterization. Rasterization also has much better hardware acceleration support on current GPUs.

Nvidia’s DLSS (Deep Learning Super Sampling) would allow us to do path tracing at half, or lower, resolution and thus have much better performance scaling. But DLSS isn’t supported in the currently stable version of Unity.

2.2. Rasterization + Traditional Effects

Real-time computer graphics have long used a technique called “rasterization” to display three-dimensional objects on a two-dimensional screen. It’s fast. And, the results have gotten very good, even if it’s still not always as good as what ray tracing can do. (Caulfield, 2018)

As mentioned before, rasterization is a lot more performant on current GPUs than Path tracing. basic rasterization has no problem giving us the performance we might need. However the drawback that it has is that it is a lot less accurate due to that a lot of lighting effects are effectively faked to look similar, but aren’t even close to the complete simulated interaction. Next to that some lighting effects are straight up either impossible to do properly through rasterization or impossible to do so performantly.

2.3. Rasterization + Raytraced Effects

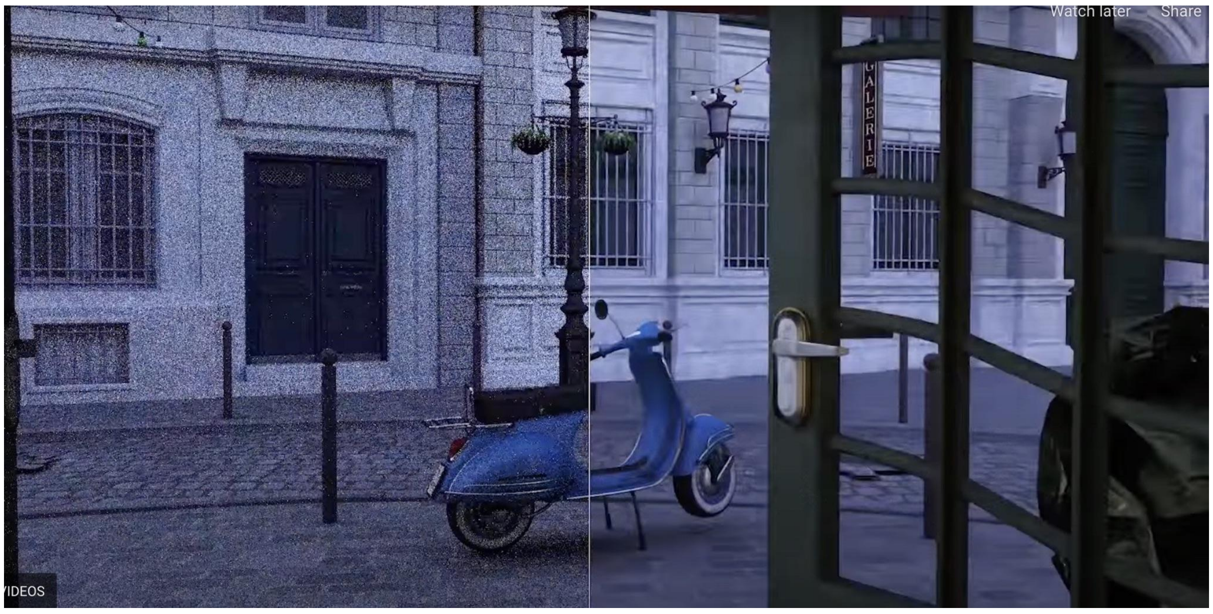
Effects such as screen space reflections are widely used in rasterized applications, and despite their impact, these techniques come with a number of limitations around reflecting off-screen pixels or geometry such as particle systems lacking in depth data. By overcoming these drawbacks, ray traced reflections become a compelling solution. Furthermore, ray tracing enables the modelling of both opaque and transparent reflections without having to compromise performance. (Mihut & Nvidia, 2020)

Using a hybrid or rasterization for direct lighting and lens effects and then using ray tracing for secondary bounces for increased accuracy on additional lighting effects seems to provide a good middle ground between realism and performance. Ray traced shadows, global illumination, subsurface interaction, ambient occlusion, refraction, and reflections allow for far more realistic light interaction modelling with the scene. It comes at a performance cost over basic rasterization, but it’s not nearly as performance heavy as path tracing.

2.4. Optix denoiser

Optix is an AI-accelerated denoiser built by Nvidia which used GPU-accelerated artificial intelligence to drastically reduce the time to render a high fidelity image by denoising it.

Optix might allow us to use the path tracer component in our simulation. the path tracer component currently takes too much samples to get a clear and acceptable output. we hope the Optix denoiser can help us use less samples on the path tracer and denoise them using optix so that it can render at least 20 frames per second.



3. How can we simulate camera sensors in the simulation and how do we use the data they create?

To grab the footage from the Unity engine and send it to the robot software (ROS) we need some kind of streaming setup between the two applications. We have looked at a few different options for streaming this data.

The hardest part is converting the frames from virtual cameras within unity to streamable data. To achieve this we have looked at a few plugins and bits of code from the asset store and github.

3.1. Unity render streaming

Unity Render Streaming is a solution that provides Unity's high quality rendering abilities via browser. It's designed to meet the needs of tasks like viewing car configurators or architectural models on mobile devices.
This solution's streaming technology takes advantage of WebRTC, and developers can even use the WebRTC package to create their own unique solutions.
(Unity Technologies, n.d.)

Unity render streaming is a relatively new package developed by Unity Technologies. We have tried to get it working with ROS. But unfortunately the codebase is very closed source and complicated so we couldn’t get it to work properly so we stopped trying to get this to work.

3.2. FMETP Stream

FMETP STREAM is a plugin for Unity3D, which aims for sharing your game view, remote assistance and VR Interactive application. In local area network, it can achieve low-latency live streaming between Unity3D apps.
(Frozen Mist, n.d.)

We have also looked into the FMETP Stream package by Frozen Mist. This is a paid package from the Unity Asset Store. The problem with this package is that it costs money and is under a Single Entity license if you buy it. So this means you can only use it for one person.

3.3. FFmpegOut

FFmpegOut is a Unity plugin that allows the Unity editor and applications to record video using FFmpeg as a video encoder.
(Takahashi, n.d.)

FFmpegOut is a recording package created by Keijiro Takahashi that can record camera's within Unity and saves the recordings to a file. The package uses a media library known as FFmpeg. This library also has the capability to stream video to a port over RTSP(Real-Time Streaming protocol). Using the package provided by Keijiro Takahashi we were able to alter some settings for FFmpeg so instead of saving to a file it would stream it to a port.

3.4. Development

We took a look at the possibility of streaming video from the FFmpeg library and saw that that would be possible to achieve. So we did a few tests to see if we could adapt Keijiro's library to stream the footage instead of saving it.

Eventually after a few hours of trying different settings with ffmpeg and using an intermediate RTSP server (RTSP Simple Server, https://github.com/aler9/rtsp-simple-server) we got the library to stream a camera from Unity to a VLC player.

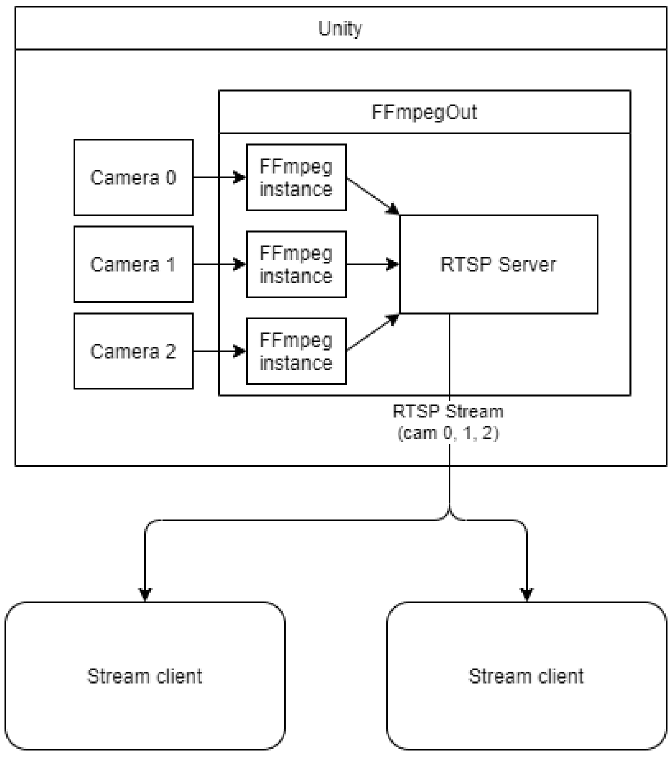
After a lot of experimenting with settings and adapting the library to fully automate the process of setting up the RTSP server and initiating the stream we now have a fully working script you can put on multiple cameras in a Unity scene.

3.5. How does the streaming work

The FFmpegOut script can be placed on every camera in the scene that should be streamed. This automatically sets up the streaming server once and makes the streams available to view on the localhost or even (when opened in the firewall) on the local network.

The user can change a few variables to change the quality, frame rate and bitrate of the streams. It depends on the quality and network needs of the user what quality and bitrate he/she should use.

The stream(s) can be watched by multiple clients at the same time.



3.6. Simulation of camera properties

Because we use the HDRP rendering pipeline in Unity we can simulate a lot of different properties that a real camera has such as accurate depth of field effects, Iso and shutter speed effects.
This means that we can accurately simulate a real camera sensor inside of Unity. This is important because the data going to ROS needs to be as accurate as it can be.

4. How do we facilitate the communication between Unity and ROS?

The Robotic Operating Software (ROS) uses a set of protocols to communicate over its network to “Nodes” that have subscribed to specific protocols. The challenge is, how do we send and receive these protocols to and from an external program (in our case Unity Engine). Here are a few tools that can help us perform this exact task

4.1. Unitys TCP Server Solution

Simulation plays an important role in robotics development, and we’re here to ensure that roboticists can use Unity for these simulations. We're starting off with a set of tools to make it easier to use Unity with existing ROS-based workflows (Unity Technologies, n.d)

The Unity-Robotics-Hub package is a set of tools that set up servers and communication with a ROS program. Unity sets up a local server that connects directly with another server on a ROS machine. These two servers communicate messages directly and translate them to their respective hosts.

This setup had too many complicated parts and required too much network communication. We believed this setup would dig too far into our latency budget and so we decided not to go with this solution.

4.2. ROS#

ROS# is a set of open source software libraries and tools in C# for communicating with ROS from .NET applications, in particular Unity3D (Siemens, Dr. Martin Bischoff)

ROS# provides API in Unity3D to send ROS protocols to ROS over the network to non-ROS applications. ROS# can’t communicate directly with ROS so it uses a ROS package called rosbridge_suite. This package acts as a translation server for the the JSON messages that ROS# sends. Translating the JSON to usable message types for ROS, and vice versa.

4.3. Development

In ROS we use the roscpp C++ library along with a series of other packages, to create some example nodes for position publisher, and stream reading. This is easily done by creating a new package in ROS and setting up all the dependencies for C++ in the config files. For the position publisher I simply followed the example publisher script provided by ROS, and modified it to return a new randomized Vector3 when we receive a message from Unity to get a new position. Similarly for the stream reader, we mostly use the opencv_bridge package to read and republish the frames from our RTSP stream.

In ROS# things are just as simple but in C#. We create a script that inherits from Unity_Subscriber/Unity_Publisher, and create some logic to listen to specific topics or publish messages over a specific topic.

Things have not always been so simple however. Initially we decided to use ROS2, which is the most recent and LTS version of ROS, because it was new and would be supported for longer. However because it was so new there weren't many tutorials or helpful resources to help us work with ROS2. Sadly we did not realize this soon enough and spent almost half the project struggling with ROS2 not working with anything else we tried. We still aren’t sure what exactly caused these errors or how to fix them, but we believe with further development of ROS2 and the ROS1 packages these errors should be fixed.

5. How could our robot affect the simulated plants?

To achieve the most realistic robot plant interaction, softbody plants are recommended. Softbody can deform in realistic ways based on external forces applied by the world, and the scene. Softbodies are complicated and hard to simulate properly and efficiently, but are the best solution for this project.

5.1. Nvidia Flex

“FleX is a particle based simulation technique for real-time visual effects”(“NVIDIA FleX.” *NVIDIA Developer*). Flex is capable of simulating rigid/deformable bodies, phase transition (Solid->Liquid->Gas), Particles, Fluids, Cloth, Rope, Adhesion, and Gases in realtime on common consumer grade hardware. Flex uses a similar method to Mayas nCloth, and Softimage’s Lagoa, but in such a way that the simulations can run in realtime instead of being computed offline.

FleX is a very promising solution but because we are using the High Definition Render Pipeline for Unity, FleX does not work in our project. FleX relies on specific shaders and features that are different/do not exist in the HDRP. If someone were to want to further develop this project they could either upgrade the FleX plugin to HDRP or possibly create a custom solution, more on that later.

5.2. Unity Cloth

Unity includes a simple cloth physics component. This component only interacts with spherical and capsule colliders and doesn’t output any force on the world. The component uses the vertices of a mesh to create constraints and spring forces between them. You can modify the spring forces and vertex “size” to adjust rigidity of the mesh.

The major problem with this solution is that the Unity cloth component is far too limited for our purposes. Because of the limited collider interaction and because the cloth cannot apply forces to the world this solution will not work for us.

5.3. Obi for Unity

Obi is a paid asset that hosts a suite of physics solvers built for Unity. Obi can do Fluids, Softbodies, Rope, and Cloth. It uses a particle based method similar to FleX, the only caveat being that all the solvers are run on CPU. Obi also sports editor tooling, sim preview, mid-sim saving, and a plethora of other useful features and tools.

While very impressive and very performant, in the current state of the project a CPU based physics solver would not fit in our performance budget. We already do a massive amount of processing on the CPU and can't afford Obi hogging up our resources slowing everything down.

5.4. Custom solution

A custom physics solver for softbody physics is the most difficult of all the aforementioned solutions. Research must be done into softbody solvers, and then implemented into GPU code for the most performance. While difficult it is also the most modular and customizable solution. You can decide how and when softbodies are being solved, and you have complete control over the quality of the sim.

6. How do we simulate a realistic looking field?

Besides rendering and simulating realistic vegetation, we also need to make sure that the farmland terrain the robot moves around on is realistically displayed. To do so properly means doing more than just lighting it realistically. Soft ground terrain can deform when touched and interactors can thus leave imprints. But weather, time of day, temperature and moisture can also affect the behaviour and light interaction of the terrain.

To simulate a realistically looking terrain we thus need a sophisticated solution that can solve all the situations we might encounter. The solution also needs to do so with a small runtime overhead to remain within our soft real-time latency budget.

6.1. Mesh baking

With mesh baking we can procedurally generate a 3D model that we then use for rendering. It is quite an intensive task to perform due to the fact that the process could potentially end up dealing with large chunks of data as well as large amounts of compute tasks to be executed on that large data. Using compute shaders could make the process a little faster, however generating an optimised mesh, and thus not having any duplicate vertices, is not very easily parallelizable. This makes mesh baking unsuitable for runtime purposes. However mesh baking isn't meant to be executed on runtime, it's supposed to be done once at or before start-up and the result is then supposed to be reused at runtime. The downside with that approach is however that the mesh can thus not be dynamically modified at runtime. This makes for a very static and non-interactable appearance.

6.2. Tessellation

Using tessellation we can generate additional geometry at runtime using tessellation hardware acceleration. This specialised hardware allows us to not only add more geometry only closer to the camera, resulting in less rendered geometry outside of the view or further away from the camera, but also allows us to place that geometry anywhere we want while it's being generated. This allows us to efficiently create deforming detailed geometry that's also easier to render than a prebaked mesh.

6.2.1. Height mapping

In order to create that additional detail we need a heightmap. This heightmap encodes the position offset any additional geometry should have from the original geometry. Using heightmaps we can create more detail in the terrain than there originally used to be.

6.2.1.a. Precomputed heightmap

Like a prebaked mesh we can also precompute the heightmap. This could be done on either CPU or GPU. Due to the heightmap simply being a texture, generating the heightmap is a lot more parallelizable than generating a mesh. The benefit of precomputing a heightmap is that besides reading the heightmap it takes no extra runtime overhead. The drawback is that like a prebaked mesh, a precomputed heightmap is static and typically not modified at runtime.

6.2.1.b. Runtime heightmap calculation

It is possible to also generate height data at runtime by embedding the heightmap generation code into the tessellation shader. This makes the tessellation more intensive to run but also would take less memory due to being able to skip having the height texture at all. This might also be easier to implement due all the code being in one place. This of course also allows for easy runtime modification of the height data since it was never stored in the first place.

6.2.1.c. Hybrid of runtime and precomputed

The best solution would be to mix the two above mentioned solutions. Use precomputation for high detail resolution data that doesn't need to change and is more intensive to calculate at runtime, and then use runtime calculate height information mixed with the precomputed data to modify the original heightmap. This allows us to save some runtime overhead, but still keep dynamic modifiability.

6.2.2. Interaction

As mentioned in 7.2.1 we require runtime modifiable heightmaps. We need this in order to model runtime terrain interaction. How we modify the heightmap could be done in multiple ways.

6.2.2.a. Per frame canvas drawing

Like drawing on a canvas with a paintbrush we could paint onto the runtime modified heightmap with the tracks of the robot. This is simple to do and not performance intensive, but drawing the tracks per frame doesn't allow modification of tracks from previous frames; There's no temporal adjustment of the tracks. This means that we can't easily mimic inter-frame interaction between the robot and the terrain and thus might result in lower accuracy.

6.2.2.b. Spline memory track drawing

Another approach we could take is to instead remember all the previous positions that the tracks have been in before, and then draw the entire track each frame. This allows modification of any part of the track at any point in time. Which means that previous frames are allowed to be adjusted by later frames once they've gathered more data about what was going to happen, and thus we can also interpolate the inter-frame interaction. The major downsides of the approach being that it's both more labour intensive to implement as well as that it's more intensive to execute at runtime. The end result of the effect might be more accurate however.

Bibliography

Caulfield, B. (2020, May 22). *What's the Difference Between Ray Tracing, Rasterization? | NVIDIA Blog*. The Official NVIDIA Blog. https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/

Electronic Arts. (2019, May 16). *Physically Based Sky, Atmosphere & Cloud Rendering - Frostbite*. Electronic Arts Inc. https://www.ea.com/frostbite/news/physically-based-sky-atmosphere-and-cloud-rendering

Elkady, A. (2012, May 7). *Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography*. Hindawi. https://www.hindawi.com/journals/jr/2012/959013/

ffmpeg Documentation. (n.d.). FFmpeg. Retrieved April 16, 2021, from https://www.ffmpeg.org/ffmpeg-all.html

Frozen Mist. (n.d.). *Frozen Mist Adventure*. Frozen Mist Adventure. Retrieved April 16, 2021, from https://www.frozenmist.com/page-product/fmetp-stream.html

Greene, C. (2020, November 19). *Robotics simulation in Unity is as easy as 1, 2, 3!* Unity Blog. https://blogs.unity3d.com/technology/robotics-simulation-in-unity-is-as-easy-as-1-2-3

Laude, T. (2019, September 20). *A Comprehensive Video Codec Comparison | APSIPA Transactions on Signal and Information Processing*. Cambridge Core. https://www.cambridge.org/core/journals/apsipa-transactions-on-signal-and-information-processing/article/comprehensive-video-codec-comparison/C1BEFBC1983E06BE4D67B239AC0F26A4

Maruyama, Y., Kato, S., & Azumi, T. (2016, October 1). *Exploring the performance of ROS2*. ACM Digital Library. https://dl.acm.org/doi/10.1145/2968478.2968502

Müller, M. (2007, April 1). *Position based dynamics*. ScienceDirect. https://linkinghub.elsevier.com/retrieve/pii/S1047320307000065

NVIDIA. (2020, November 24). *NVIDIA FleX*. NVIDIA Developer. https://developer.nvidia.com/flex

Open Robotics. (n.d.). *rosbridge_suite - ROS Wiki*. ROS Wiki. Retrieved June 24, 2021, from http://wiki.ros.org/rosbridge_suite

Scott, K. (2020, February 20). *About the ROS-Agriculture category*. ROS Discourse. https://discourse.ros.org/t/about-the-ros-agriculture-category/12890

Siemens. (n.d.). *siemens/ros-sharp*. GitHub. Retrieved June 24, 2021, from https://github.com/siemens/ros-sharp

Takahashi, K. (n.d.). *keijiro/FFmpegOut*. GitHub. Retrieved April 16, 2021, from https://github.com/keijiro/FFmpegOut

Unity Technologies. (n.d.). *Unity-Technologies/UnityRenderStreaming*. GitHub. Retrieved April 16, 2021, from https://github.com/Unity-Technologies/UnityRenderStreaming

Walt Disney Animation Studios. (2016, August 9). *Disney's Practical Guide to Path Tracing*. YouTube. https://www.youtube.com/watch?v=frLwRLS_ZR0

Wikipedia contributors. (2021, June 11). *Deep learning super sampling*. Wikipedia. https://en.wikipedia.org/wiki/Deep_learning_super_sampling