Realistic plant simulation

Project report

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

Summary

This project contains the research, process and result of the Realistic Plant Simulation project for the minor Immersive Media and project IMT&S.

This project is aimed at improving the development cycle of agricultural robots. The application is a simulation environment where these robots can be tested virtually, so the software can be tested before bringing the hardware out in the real world. This can improve the development time of these robots.

Table of contents

1. Introduction

The Realistic Plant Simulation (RPS) application is built as a suite of editor tools within Unity. These tools offer the possibility to the ROS comunity to test their code within a virtualized environment inside of Unity. This is less time consuming and more graphically realistic than the current ways of testing their code using programs like Gazebo or going out to a field with an actual robot.

The end goals of the project are:

1. A field of crops waving in the wind that are procedurally generated.
2. This simulation can be virtually observed by one or multiple cameras or sensors.
3. The plants must look realistic but also run in real-time (minimum of 20fps)
4. The simulation can interface with virtual sensors so that the simulation data can be fed back to ROS.

The assignment is provided by Saxion Mechatronics.

2. Context

2.1. Problem analysis

Currently Saxion Mechatronics is using Gazebo to simulate robots. The problem with Gazebo is that it is very slow and not very accurate. This applies to both physics and graphics.

The idea of this project is to introduce a specialized alternative to Gazebo that runs faster and has better visual fidelity. This specialized simulation environment should be tailored to agricultural robots.

The visual fidelity that Gazebo has is not meant for testing robots that use computer vision to identify things in the camera view. This means that these robots cannot be tested in a simulation environment effectively.

2.2. Desired outcome

To address all of these issues an application needs to be created that can implement all of these requirements and possibly more in the future. Because of these requirements we chose to create the project in Unity.

This application needs to contain at least the following features:

- **A virtual field of realistic looking plants**
    - To correctly simulate agricultural robots there needs to be a virtual environment for it to be simulated in.
    - This environment will exist in a virtual farmland field with plants. The field and plants will both be procedurally generated at editor time.
- **A virtual robot that can move through the environment and be controlled by outside scripts/connections**
    - To do something with the virtual environment there needs to be a robot that can move around the scene and get visual data from it.
    - This robot must also be able to be controlled by external scripts or connections such as ROS
- **A connection to ROS**
    - The unity project needs to be able to connect to ROS to be able to interface with actual robot software.
    - It needs to have two way communication to achieve the desired result. Data must be able to be sent and received over the different types of methods ROS supplies.
    - It must also be possible for the ROS connection to receive video data from Unity. This means color and depth data from 1 or more Unity cameras
- **Modularity**
    - It is important that the application is modular so many different types of robots and connections can be simulated.
- **Interaction**
    - The robot needs to be able to interact with the scene. The interaction can mean leaving tracks behind in the sand, or interacting with the plants to cut them etc.
    - This interaction is important for the realism and usability of the project.

2.3. Stakeholders

For this project there are a few stakeholders. Below is described who is a stakeholder and what their role is for this project.

- **Wilco Bonestroo**
    - Wilco is a researcher for robotics software at Saxion Mechatronics. His research focuses on the software (architecture) for robotics ranging from autonomous ground robots and aerial vehicles to industrial robots.
    - His role in this project is to act as our “customer” and provide us with information and discuss features that are important to have.
- **Abeje Mersha**
    - Abeje is a professor of Unmanned Robotic Systems at Saxion Mechatronics. His research is focussed on unmanned robots (aerial robots, ground robots, surface and underwater robots).
    - Due to Wilco’s unforeseen personal circumstances he wasn’t able to continue supervising the project, so Abeje took over the project after the halfway point.
- **Saxion Mechatronics**
    - As our main target audience, Saxion Mechatronics is also a stakeholder in this project. The Saxion Mechatronics research group focuses on the development of robotic and mechatronic technologies.
    - Saxion Mechatronics is also an important stakeholder. They are gonna use this project to develop their agricultural robots. So it is important that we make the project as accessible to them as possible.

3. Research & design methods

3.1. Scrum

In this project we will be using scrum to divide up the work into smaller pieces. We are using Trello as a planning board to host the issues and keep track of them.
The duration of each sprint is two weeks. At the start of the sprint we will divide up the epic's into smaller issues that can be put into the backlog of the next sprint.

At the end of each sprint we will do a retrospective to fish out the good and bad parts of the last sprint.

3.2. Design/Research methods

3.2.1. SWOT

We will analyze our potential benefits and shortcomings using a SWOT diagram in order to further understand our market position and what we should keep in mind during development of the product.

(SWOT Analysis - ICT Research Methods, 2018)

3.2.2. Design thinking process

Empathize

By interviewing mechatronics students and having discussions with our client we hope to discover the needs and wants of our user base so they can have a better product.

Define

After review and discussion of the needs and wants of the community, we will write a problem statement. This statement will outline what we must solve for the community.

Ideate

Using our problem statement we will brainstorm multiple solutions and carefully analyze their validity, and effectiveness for the problem.

Prototype

We will continuously keep prototyping iteratively in order to stay flexible and to be able to apply research and acquired knowledge as we develop.

Test

We will consistently conduct sample tests using the supposed user-base to gather feedback about what they think of our product.

Assess

We will analyze the test results and incorporate any improvements and insights we discover into the next development sprint in order to be able to test it again with the next prototype.

3.2.3. Literature Study

We will be reading papers and other resources focused on video streaming and video compression. Alongside this research we will also be comparing different methods of inter-application communication with ROS and Unity.
(Literature Study - ICT Research Methods, 2018)

3.2.4. Prototyping

We will be dividing each feature of the final product into chunks and developing standalone prototypes for each so that we can test specific features and design concrete implementations.
(Prototyping - ICT Research Methods, 2018)

4. Research process and results

4.1. 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

4.1.1. 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.

4.1.2. 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.

4.2. 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.

4.2.1. 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.

4.2.2. 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.

4.2.3. Rasterization + Raytraced Effects

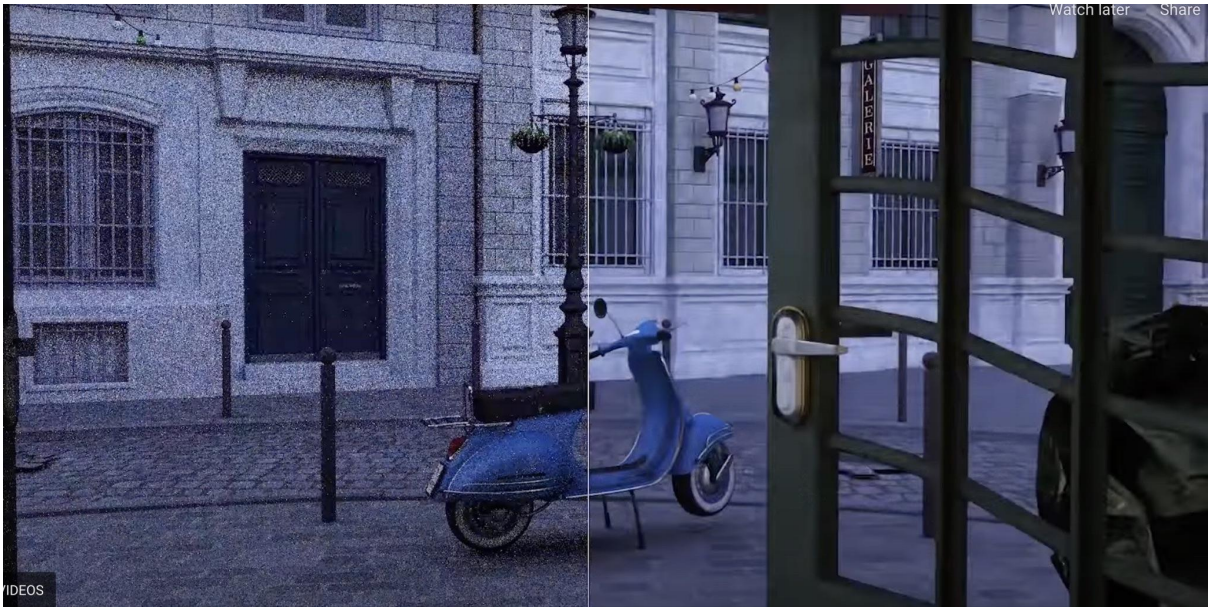
Effects such as screen space reflections are widely used in rasterized applications, and despite their impact, these techniques come with a number of limitations around reflecting off-screen pixels or geometry such as particle systems lacking in depth data. By overcoming these drawbacks, ray traced reflections 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.

4.2.4. 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 many 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.



4.3. 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.

4.3.1. 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.

4.3.2. 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.

4.3.3. 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.

4.3.4. 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.

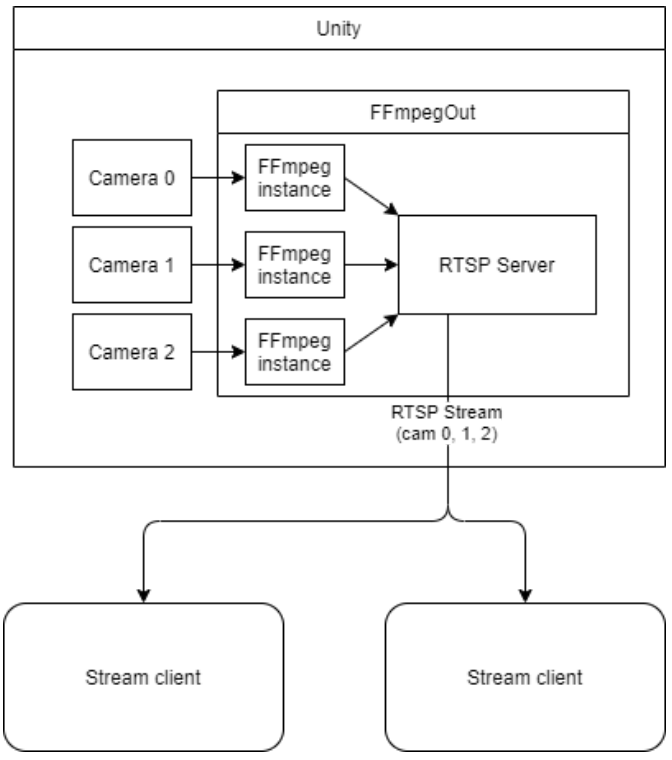
After a lot of experimenting with settings and adapting the library to fully automate the process of setting up the RTSP server and initiating the stream we now have a fully working script you can put on multiple cameras in a Unity scene.

4.3.5. 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.



4.3.6. 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.

4.4. 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

4.4.1. 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.

4.4.2. 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.

4.4.3. 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.

4.5. 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.

4.5.1. 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.

4.5.2. 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.

4.5.3. 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.

4.5.4. 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.

4.6. 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.

4.6.1. 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.

4.6.2. 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.

4.6.2.1. 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.

4.6.2.1.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.

4.6.2.1.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.

4.6.2.1.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.

4.6.2.2. 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.

4.6.2.2.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.

4.6.2.2.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.

5. Results

5.1. ROS Connection

To allow Unity and ROS to communicate we used the ROS# Unity packages paired with the rosbridge_suite ROS package. These two packages communicate with each other over the network, serializing ROS topics and Unity's messages into JSON and transmitting them to each other. ROS# also implements an API to create C# scripts for each of the ROS protocols(Services,Messages,Actions).

Because ROS is easier to use and develop on Linux distros, but we are developing on Windows, we used the WSL(Windows Subsystem Linux) to run an Ubuntu virtual machine on our Windows machines. This allows us to run full fat versions of ROS and RVIZ(ROS Visualization), and run Unity on the same machine, making network communication simple.

5.2. Camera Streaming

To get camera data to ROS or some other client we implemented a camera streaming stack in the project. This can send video streams from Unity camera to any client that supports RTSP stream playback.

To use this video data in ROS we implemented a library which can decode the stream into an Image topic.

5.3. Terrain

To create a realistic terrain we created a script that generates terrain procedurally based on some parameters the user can change. We did this because not every terrain the robot should be tested on looks the same.

By generating our terrain based on these parameters we still make our terrain procedural but it can be diversified for different robots and create different conditions to test the robot with.

5.4. Plants

To get plants into the scene we implemented a script around the Houdini Engine connector for Unity. This script allows the user to spawn many different types of plants and vary the settings according to parameters the user sets up.

The plants can be made inside of Houdini using L-system rules. These can then be exported to a file and imported into Unity.

There is also a way to change the spawning placement behavior per plant. This makes it possible to distinguish between the spawning behavior of crops and weeds for example.

5.5. Graphics

5.5.1. Customizability

To allow users to create custom visual effects or interactivity we created a tool that allows anyone to create custom shaders for the Unity HDRP. The tool allows anyone to create custom vertex shaders, custom tessellation shaders and custom fragment shaders.

5.5.2. Fidelity

To create a realistic looking image we use lots of different visual effects. Some of these effects are achieved using hardware-accelerated real-time raytracing for extra accuracy. Most of the raytraced effects have less accurate non-raytraced alternatives in case performance is more important. All effects, where applicable, have been configured to use the physical camera settings in order to model the real life light interaction with the camera lenses and sensor. We also utilize tessellation to achieve a higher detail density in the terrain.

6. Conclusion

The current application has most of the desired functionality. Due to unforeseen circumstances with our customer we couldn't fit in all of the functionality they wanted and had to make some choices of our own.

Because the application does not have all of the features we wanted to implement. We are making sure this project can be carried over to another team. To do this we made a technical documentation and described our recommendations in 7. Recommendations.

The progress of the project was good and the team worked together well. Communication between team members was great and if anybody had problems with something the rest of the team was available straight away to help them.

At the start of the project progress went a little bit slower than we would have hoped. This was mostly due to a lack of information from our customer and vague requirements. We eventually got that sorted out. And when we had our requirements well documented and a project set up, we could get to work more efficiently.

The project went on pretty well. We had to drop some wished-for features because of limited time. An example of this would be scene interaction. This means that the robot should be able to change stuff in the scene. Unfortunately this proved way too difficult within our limited time budget so we decided to drop this feature. The rest of the project went pretty well. We had clear requirements and made some good features to make sure all the requirements were met.

Our client was very enthusiastic about our product. Although there is still room for improvement. We can conclude from this that the project was a success.

7. Recommendations

For the functionality the project still needs to be fully usable we are recommending the following things to add/change.

7.1. Scene interaction

Scene interaction refers to what forces the scene and robot model apply to each other and how that can be visualized realistically.

7.1.1. Ground interaction

Ground interaction is any sort of collision, and then subsequent visualization of the results of the collision with the terrain. Currently the farmland terrain has no collider. This is because the procedural nature of the terrain makes it difficult to use a standard collider, therefore the terrains collider must also be procedural. Unity has an implementation for a procedural collider called “Mesh Collider”. This method is usually considered to be quite slow, but further testing should be done to check it’s viability in our purposes.

Along with a procedural collider, the mesh should also physically deform as the robot travels over it, leaving tracks and flattened plants. This can be done using the currently implemented tesselation shader, raymarching, or maybe some decals with heightmaps.

7.1.2. Plant interaction

Plant interaction refers to the physical collision deformation, and the slicing of our procedural plant meshes. The recommended method for doing this is a softbody physics solver.

7.2. Plant generation

There is also some stuff that can be improved with the plant generation. This can give huge performance benefits and more possibilities for generating plants.

Described below there are two ways the plant generator can be improved.

7.2.1. Custom L-system implementation

Currently we are using Houdini L-systems to generate plants. This is one way to do it. And Unfortunately for us it was the only way because we didn't have an artist that could set up custom L-systems for us.

A way this setup could be improved is to implement a custom L-system generator to generate meshes within Unity. This will speed up the generator because it could be possible to generate plants on multiple threads. There could also be more improvements to the setup of the meshes and objects within the plants so GPU instancing could be possible. This could give us a pretty large performance increase.

This is one of the limitations of the Houdini-Unity plugin. It is not possible to generate plants on other threads. So no multithreading is possible.

7.2.2. Implement vegetation plugin

As an easy alternative to the custom L-system implementation could be a plugin that can generate vegetation. For example The Vegetation Engine by BOXOPHOBIC (2020) could be used to set up the plants.

This might not be the most practical setup for generating thousands of plants. But it could be an easy solution to the problem of creating a field of realistic looking plants as the current solution doesn't give us the looks and performance results we want. This plugin or similar plugins could

The custom L-system implementation is still probably a better option. But it is a lot more difficult and time consuming to program.

7.3. General optimization

The current simulation runs at about 50ms per frame (20 frames per second) with 4 cameras with both color and depth streams without using RTX. Based on the current performance some further optimizations are quite beneficial. There are a few more directions we could look into for further optimization possibilities.

7.3.1. LOD's

Currently all plants at all distances are rendered using the same models and fidelity, even if they are further away from the camera and thus less noticeable. Reducing the fidelity of plants that are less visible would make it easier to render more plants in view within the performance budget.

7.3.2. Calculate terrain in shader

In the current implementation of our terrain generation we use compute shaders to generate terrain for each chunk. however by doing so the compute shader needs to set up a sampler for each chunk which is unnecessary. Instead of sending the height data to the shader we can also make a shader that generates this itself based on some position data of the chunk. This means all chunks have the exact same shader and the exact same mesh which also means that we can enable GPU instancing on these chunks. This would have a massive impact on our gpu VRAM that's needed as well. This will probably allow us to create bigger terrains than we currently can because there's more memory left to generate chunks.

7.3.3. DLSS

NVidia recently announced their new DLSS feature for their RTX graphics cards.
Deep Learning Super Sampling or DLSS is an AI based upscaling technology that can be used for real time use. By lowering the camera resolution and upscaling the output using DLSS we can achieve a far better framerate. Unfortunately this feature isn't implemented in Unity yet. Unity announced they are going to implement it in the newest 2021.2 version which hasn't come out yet.

7.3.4. Instanced rendering of plants & terrain

If the custom L-system is implemented correctly. It could also be possible to instance all of the leaves on the plants. This can massively cut down on draw-calls and thus improve performance. To do this every leaf needs to be rendered using the same mesh and material. This way all of the leaves can be rendered with one draw-call.

The same could be done to the terrain. If we displace the terrain in a shader as described in 7.3.2. Calculate terrain in shader, it should be possible to instance the terrain as well.

Putting this all together it can give us a lot of improvement in performance because the GPU can render all of these things in one pass.

7.3.5. Separation from the Unity Editor

The Unity editor takes up a large amount of the performance allotted to us by Unity. This becomes a big problem while we are trying to render high resolution terrains and 1000s of procedural plants, alongside countless other visual enhancements. While the simple solution for this would be to build the project as a standalone application, this is not possible due to the amount of Editor specific code we use in our project. Unity does not allow editor specific code to be built into the standalone application.Therefore we recommend research into separating the current code from the Unity Editor allowing it to be built as a standalone application giving us a lot more wiggle room for performance.

From here there are two possible paths. Either the workflow for the simulator is altered, meaning that to run a test you set up the scene in Unity with plants and terrain, setup the robot, and build the project into a separate application to run a test. Or you could design and implement a runtime editor for terrain and plant generation, and a runtime C# compiler for custom scripts and sensors.

The build workflow option would be the simplest to implement but the resulting workflow would be very slow, and the inherent rigidity of a standalone application would make debugging very difficult.

The runtime editor solution would be the most complicated but would result in a faster simulation (to a degree), and a smoother workflow. But the work needed to design, test and reiterate the design would be large. And a runtime C# compiler/interpreter would also not be an easy feat.

7.4. Graphical improvements

Although we do a lot already in order to make a convincingly realistic image, it's not as realistic as possible. With further optimisations and better hardware more rendering features could be opened up to fit within the performance budget.

7.4.1. Path tracing

When we started the project we tried to use the HDRP path trace option to get more realistic looking images of our simulation. We hoped this could be used for our camera's to render the plants and the terrain. Even though the current path tracing implementation isn't completely implemented by unity we could already see that the results looked way more realistic. There are several reasons why we can't use it yet.

The first reason is that the path tracer renders samples progressively which means it gets less noisy every frame. There's currently no way in the unity API to talk to this path tracer component to get any information on how far it's progress is with rendering the image.

That takes us to the second problem, it takes too long for a single frame to render. After a few seconds it gets denoised and the frame looks (acceptable). We tried denoising the image with the Nvidia Optix Denoiser, however this still took too long to render a single image.

Maybe a combination of DLSS and a 3000 series graphics card might achieve this. For now it's just out of our abilities.

7.4.2. Environmental effects

Besides more accurate effects, having more effects can also increase the realism of the simulation. Spending attention to details like dust getting blown in the wind, morning dew/mist, or little rain droplets after rain showers could go a long way in pushing the realism to another level. Environmental effects play a big role in visuals when trying to simulate an external setting.

List of literature

BOXOPHOBIC. (2020, August 17). *The Vegetation Engine | Utilities Tools*. Unity Asset Store. https://assetstore.unity.com/packages/tools/utilities/the-vegetation-engine-159647

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