Realistic Plant Simulation

Technical documentation

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

Table of contents

Table of contents **2**

Introduction **4**

1. Hardware **5**
1.1. Minimum specifications 5
1.2. Recommended specifications 5

2. Terrain generation **6**
2.1. Tessellation 6
2.2. Heightmaps 7
2.2.1. Perlin noise 7
2.2.2. Sine waves 8
2.3. Normal maps 9
2.4. Compute shaders 9

3. Plant generator **9**
3.1. Houdini 9
3.2. Houdini-Unity connector 10
3.3. Plant generator script 10
3.4.1. Houdini-Unity connector api 11
3.4.2. Placement strategies 11

4. Terrain utility **12**
4.1. Terrain generator functions 12
4.2. Plant generator functions 12
4.3. Generator options 12

5. ROS integration **13**
5.1. ROS 13
5.2. Windows Subsystem Linux 13
5.2. RVIZ 13
5.3. ROS# 13
5.4. ROS Packages 14
5.4.1. rosbridge 14
5.4.2. OpenCV_bridge 14

6. Streaming sensor data **15**
6.1. FFmpegOut 15
6.1.1. Library changes 15
6.2. RTSP Simple Server 15
6.3. Depth and Color Information 15

7. Graphics **16**
7.1. Custom Shaders 16
7.1.1 Justification 16
7.1.2 Usage 16

Create a custom shader 16
Edit a custom shader 18
Vertex Shader 19
Tessellation Vertex Shader 19
Fragment Shader 19
7.1.3 Workings 20
Shader 20
Editor 20
7.2 Post Processing 20
Effects 20

Introduction

The purpose of this project is to create a simulated environment of plants to significantly speed up the development of new robots in ROS. This simulation could for example be used to test hardware that is not available yet or to generate sensor data that can be used during development.

This project exists as a suite of editor tools within the Unity engine to generate and visualize the sensor data needed for a robot built in ROS.

1. Hardware

For this project we recommend at least a pc with the following specs:

1.1. Minimum specifications

CPU: 4 cores 8 threads at 4ghz or better
RAM: 32 GB or better
GPU: Nvidia RTX 2070 or better with at least 8gb of vram

1.2. Recommended specifications

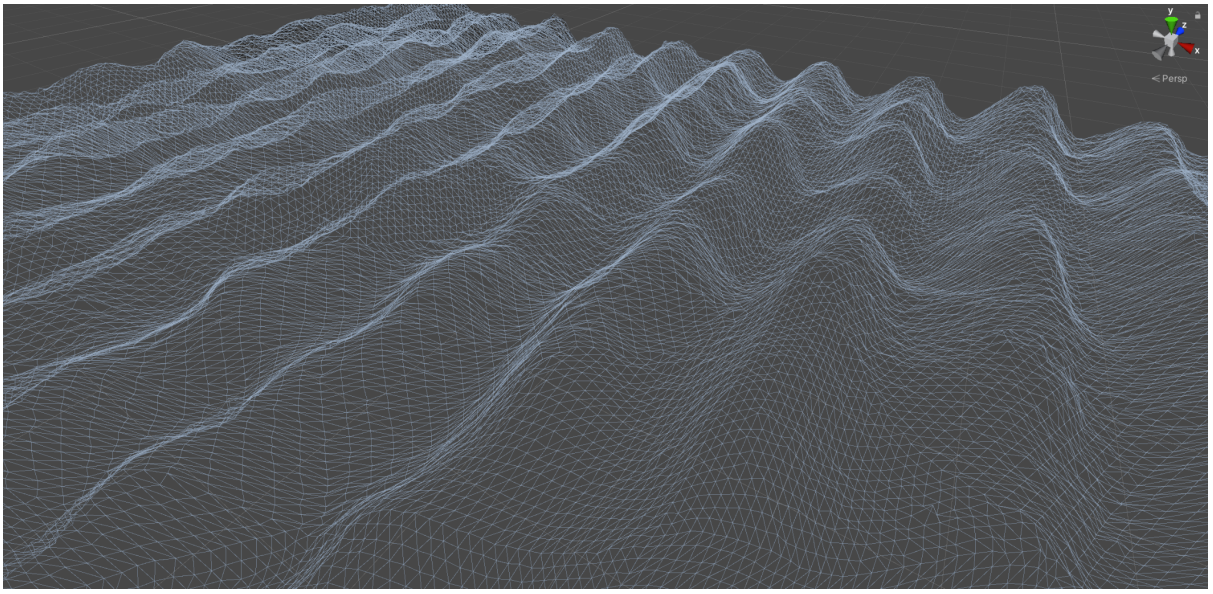
CPU: 6 cores 12 threads at 4.5ghz or better
RAM: 64 GB or better
GPU: Nvidia RTX 3080 or better with at least 10gb of vram

2. Terrain generation

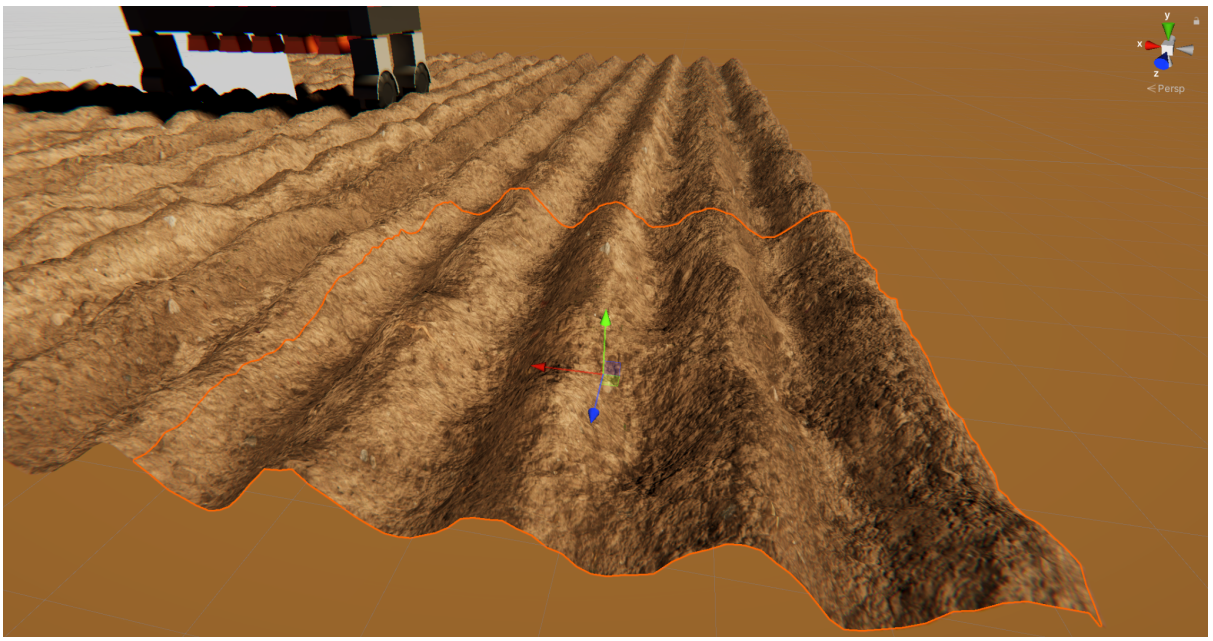
2.1. Tessellation

To procedurally generate a terrain that's suitable for generating realistic sensor data we generate procedural heightmaps and apply them to a material with a tessellation shader.

By using a tessellation shader we don't have to worry about using LOD's for terrain chunks that are far away. The tesselation shader generates the triangles and vertices for the chunk based on a heightmap texture applied to the material and adds a higher density of triangles (more detail) in area's close to the camera.



The height of the vertices generated by the tessellation shader is determined based on the grayscale value of the heightmap that the shader reads from. The resolution of the heightmap also has an impact on the tessellation, the higher resolution the texture is the more detailed our tessellation gets. For this reason we chose to generate multiple chunks, each having their own tessellation material with their own procedural heightmap.



2.2. Heightmaps

As described in 2.1. Tessellation, our tesselation shader handles the generation of the terrain mesh based on a height map texture. So to create a procedural terrain we have to generate procedural heightmap textures for this shader. We also have to keep in mind that the procedural textures align with the textures generated for the aligning chunks.

2.2.1. Perlin noise

When generating a procedural heightmap we start off with generating some noise to our texture. We do this using a perlin noise algorithm. The perlin noise function itself generates random float values based on a given x and y value. when we put the output of this noise function into the colors of a texture we get the result shown in (*Fig. 1).*

Fig 1. Perlin noise

When adding this resulting texture to our tessellation shader we can already see results. However to get more details into our terrain we use multiple layers of noise (octaves) and combine them to get better grained details in our height texture. In the images shown in (*Fig. 2 and Fig. 3*) you can see the impact of using multiple octaves.

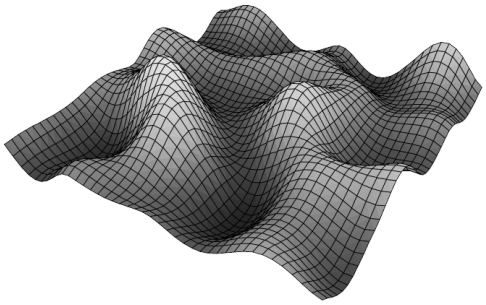


Fig 2. Single Octave

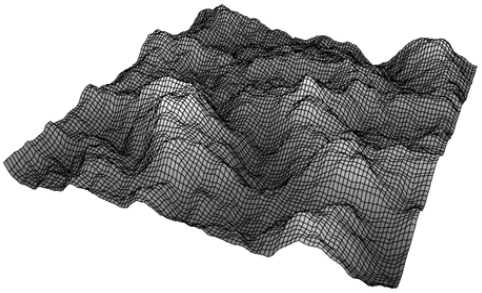
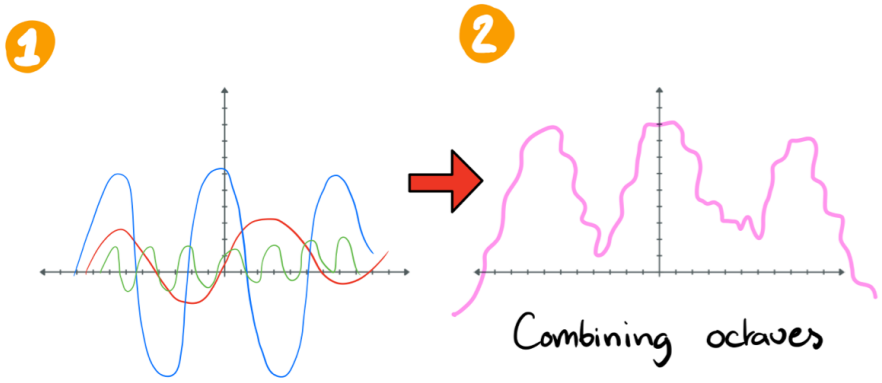


Fig 3. Multiple octaves



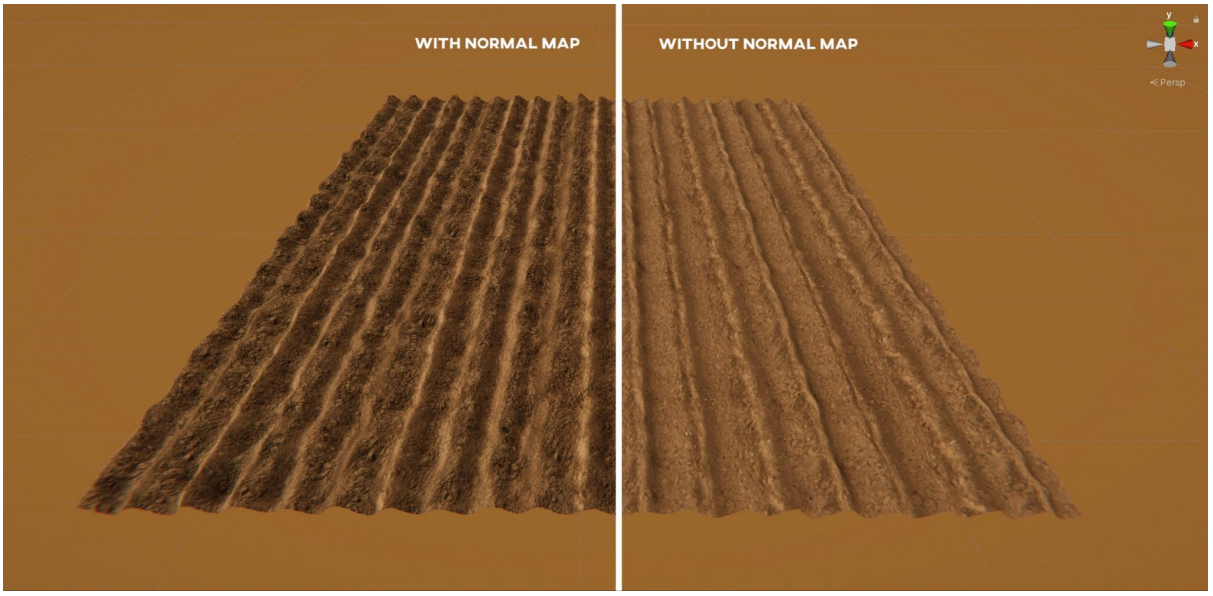
2.2.2. Sine waves

In the example images of the terrain we received from our client we saw the farmland having somewhat straight bumps and grooves in the soil where the plants are placed by farming machines. The terrain we get from our perlin noise is already looking like farmland, however we still want to have the bumps and grooves on it like we've seen in the example images.

To do this we simply combine our perlin noise outputs with the output of a sine wave function as shown in (Fig. 4). To give the user the freedom to change the farmland the way he likes it we exposed all the perlin noise and sine wave settings to the ***TerrainGenerator*** component.

When our tesselation shader reads the values from our height map it creates triangles and vertices and sets the height of the vertice to the interpolated value of the pixels that align with that vertice. Because of this our chunks need to interpolate correctly on their edges as well, because the shader can’t read from textures outside of our terrain, we have to make sure our textures have the color of their pixels the same on both of the aligning ends of the texture.

2.3. Normal maps



After applying the procedural heightmap to our ***TerrainChunk*** we had to update our normal map as well. Because of the changes in the mesh our current normal map is not correct anymore.

So instead of using the old normal map we generate a normal map from our heightmap. However, our heightmap does not include the normals for the details that are in the base map of our materials. To fix this we combine the original normal map of the material with our procedural one by blending it.

2.4. Compute shaders

The computations to generate the procedural mappings for each chunks take a long time. Even though the terrain is pregenerated and it doesn't matter too much how long the generation of the chunks takes we chose to make compute shaders for the computations of our procedural mappings. By doing so and letting our gpu handle the workload we go from seconds to milliseconds for generating each chunk.

3. Plant generator

3.1. Houdini

To procedurally generate plants we use SideFX Houdini. This is a 3D animation software application that is built on the idea of procedural generation.

Using L-systems within Houdini we can generate many variations of the same type of plant. These L-systems can be exported from Houdini to a HDA file. This file can be imported into the Unity project.

3.2. Houdini-Unity connector

To be able to import a Houdini file into unity we need the Unity connector for Houdini. This plugin is able to modify parameters of the plants so that we can keep generating new variations of them.

We created various scripts to do this generation automatically at editor time. These scripts are able to modify Houdini variables and generate different types of plants based on positioning strategies.

3.3. Plant generator script

The plant generator script that we wrote has the possibility to spawn the Houdini plants automatically, this can be assigned in the Unity inspector.

This means that new plants can be added to the project without changing any code.

The main houdini plant is a GameObject that contains the Houdini scripts.
Using the ***Randomizers*** it is possible to let the plant spawner vary different aspects of the Houdini asset. The ***Houdini key*** can be entered and a random float/int will be chosen between the ***min*** and ***max*** values that are entered.

After randomizing the Houdini variables, the plant generator script creates a new empty game object that will hold the plant meshes. It then copies only the meshes from the Houdini plant object and places them into the empty gameobject
Doing this eliminates some of the lag and issues that come with copying the whole plant or instancing the entire houdini plant multiple times.

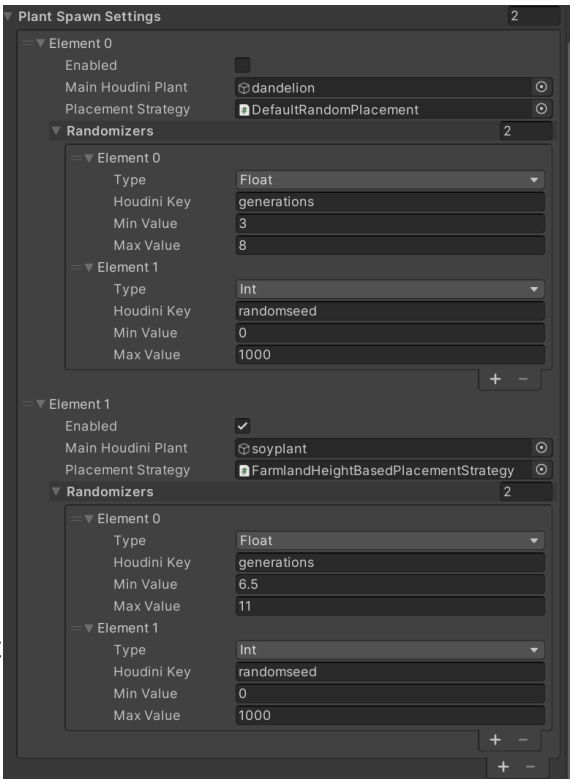


Fig 5. Plant spawn settings

3.4.1. Houdini-Unity connector api

To do this we use the Houdini-Unity connector api.

```
/// <summary>
/// Randomizes the houdini variables of one plant according to the randomizer settings
/// </summary>
/// <param name="spawnSettings"></param>
/// <returns></returns>
1 reference
private IEnumerator randomizeHoudiniVars(PlantSpawnSettings spawnSettings)
{
    HEU_HoudiniAssetRoot assetRoot = spawnSettings.mainHoudiniPlant.GetComponent<HEU_HoudiniAssetRoot>();
    foreach (PlantVariationSetting variation in spawnSettings.randomizers)
    {
        switch(variation.type)
        {
            case PlantVariationTypes.Int:
                HEU_ParameterUtility.SetInt(assetRoot._houdiniAsset, variation.houdiniKey, Random.Range((int)variation.minValue, (int)variation.maxValue));
                break;
            case PlantVariationTypes.Float:
                HEU_ParameterUtility.SetFloat(assetRoot._houdiniAsset, variation.houdiniKey, Random.Range(variation.minValue, variation.maxValue));
                break;
        }
    }
    assetRoot._houdiniAsset.RequestCook(bCheckParametersChanged: true, bAsync: false, bSkipCookCheck: true, bUploadParameters: true);
    yield return null;
}
```

Fig 6. Houdini api code

To randomize the houdini values on the main plant. We first need to get the ***HoudiniAssetRoot*** component. This manages the houdini instance on that plant.
With this ***assetRoot*** we can change parameters in the houdini plant (ints or floats) and cook the changed asset.

After this cooking process the plant meshes can be copied and instanced onto the field.

3.4.2. Placement strategies

To change the spawning behavior of plants it is possible to assign placement strategy scripts per plant. These scripts can be created according to the way someone wants to spawn plants. For example if the user wants plants to spawn on the top of the terrain bumps he/she could implement a placement strategy for that behavior and assign it to specific plants where it is applicable.

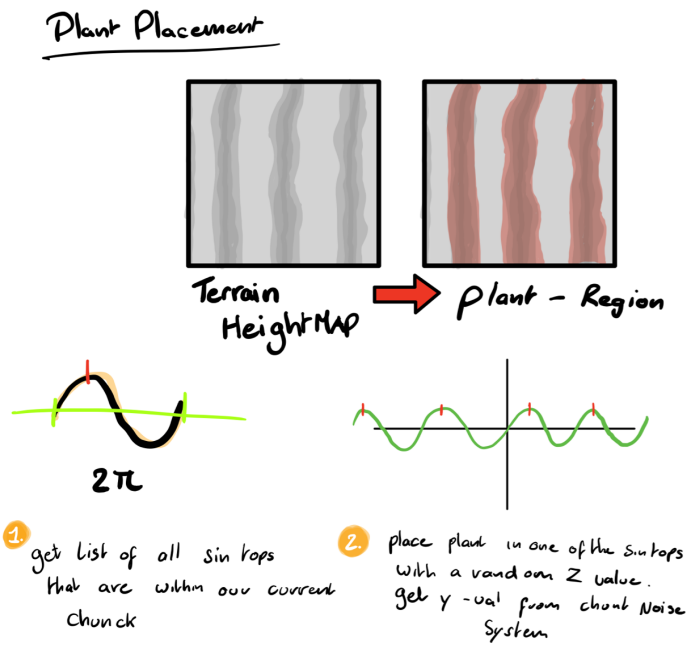


Fig 7. Terrain height based placement strategy

4. Terrain utility

To make it easier for the end user to generate the terrain and plants there is a terrain utility script that has some fields that automatically edit the plant generator and terrain generator.

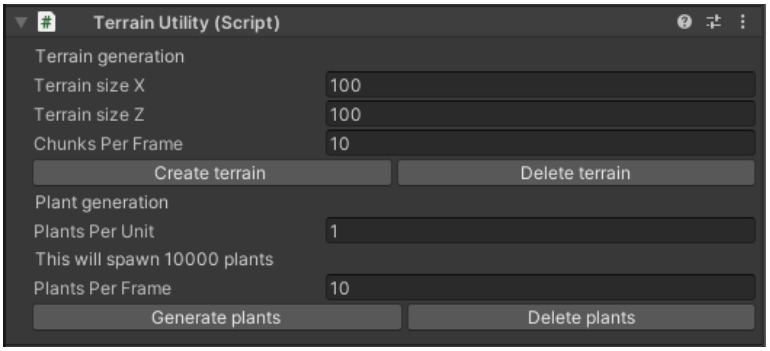


Fig 8. Terrain utility script

4.1. Terrain generator functions

The terrain utility has a few input fields and buttons to make generating the terrain easier. There are two fields for the terrain size (***Terrain size X*** and ***Terrain size Z***). These fields change parameters in the terrain generator script. and also set up the amount of plants to be generated based on this size and the plants per unit.

4.2. Plant generator functions

The terrain utility also has some options for the plant generator. In combination with the terrain size and the plants per unit input, the terrain utility can calculate the amount of plants it needs to spawn for the entire terrain.

4.3. Generator options

To generate the terrain and plants we use coroutines to not make the editor freeze up. This means that everytime we want to update the editor screen we also need to redraw the scene view. This costs us precious CPU time.

To combat this problem we added a feature that can batch some of the work together. Only after that batch of work has been completed we redraw the editor. This way it wont freeze up and the amount of time the generators take is still acceptable.

This behavior can be changed by changing the ***Chunks Per Frame*** or ***Plants Per Frame*** option.

5. ROS integration

5.1. ROS

Since most robots use the ROS(Robotic Operating System) middleware to communicate with their other physical parts, we will be using the same middleware to communicate with the brains of a robot, to simulate actuation or sensor commands. ROS uses the local area network of the host machine to transmit TCP-like packages. ROS defines three types of protocols for information transmission. Messages, requests and responses, and actions.

The version of ROS we are implementing is ROS 1 Noetic Ninjemys. Noetic is the last version of ROS 1 and its EOL date is May 2025. To learn more about ROS and how it works refer to http://wiki.ros.org/.

5.2. Windows Subsystem Linux

ROS and its supported tools all require Linux to run. To accelerate development we used the WSL(Windows Subsystem Linux). WSL is a kernel level virtual machine that is hosted in Windows. It allows us to easily run Linux only commands and programs, and view/interact with them on Windows.

Because WSL is in preview right now, we had to install a Windows Insider Preview Build. This allows us to also take advantage of the new GUI api installed with WSL2. This means we can open up programs from our Linux distro and have a Linux window open up on our Windows desktop.

5.2. RVIZ

RVIZ is a visualization program that allows us to see all the information we are passing over the ROS network. Positions, Camera sensor data, rotations, odometric data, etc.

5.3. ROS#

Unity being our development platform means that all of our code written for the simulation environment will be written in C#. To make this development process easier we implemented the ROS# library. This library allows us to communicate with our ROS instance running on a local WSL VM, using plain C# Unity scripts. Using ROS# We have implemented some basic publisher and subscriber scripts, for robot position, and camera position & orientation. To learn more about ROS# and how it works refer to https://github.com/siemens/ros-sharp.

5.4. ROS Packages

ROS has a large assortment of community developed packages. These packages give ROS specific capabilities, depending on your use case. Below are the ROS packages that we require for this project.

5.4.1. rosbridge

Rosbridge is a ROS package that allows our Unity code to communicate with a ROS node running on a networked machine. It acts as a TCP endpoint server that translates TCP packets sent by ROS# to ROS, into ROS protocols.

5.4.2. OpenCV_bridge

OpenCV_bridge is a ROS1 package that acts as a wrapper and API for OpenCV. We use OpenCV to read from our color and depth streams and parse each frame into image topics. These topics are then published for use in RVIZ

6. Streaming sensor data

To get camera and depth data from Unity into ROS we have implemented a way to stream camera data using RTSP streaming. This makes it possible to stream locally or over a network.

6.1. FFmpegOut

To get the streaming working we modified a library we found on github. This is the FFmpegOut library made by Keijiro Takahashi. This library is meant for grabbing the video output of a unity camera and saving it to a video file. It has many options for video codecs. It uses the FFmpeg tool to transcode the video stream to a video file.

6.1.1. Library changes

To get this library to stream the output of the unity video camera to a usable stream format we needed to modify it a little bit. FFmpeg is pretty easy to get working with RTSP streaming. It has a lot of options and we experimented to get the right combination of options to achieve a good video quality and stable connection.

6.2. RTSP Simple Server

To get the RTSP stream from FFmpeg to a client you need a RTSP streaming server. We are using the rtsp-simple-server library created by Alessandro Ros, which is an easy to use and simple library written in the Go programming language.

We included some modifications to the FFmpegOut library that automatically starts up this RTSP server when the first stream tries to start. This means that a instance of this server will automatically be created at runtime and the user doesn’t need to do anything to set up this stream.

6.3. Depth and Color Information

Depth and color information are transmitted separately on different RTSP streams. We take the rendered frame from our camera and parse it, using a compute shader, into two seperate textures. One depth, and the other color. These are then pushed to the FFmpegOut pipeline and streamed over the network.

7. Graphics

7.1. Custom Shaders

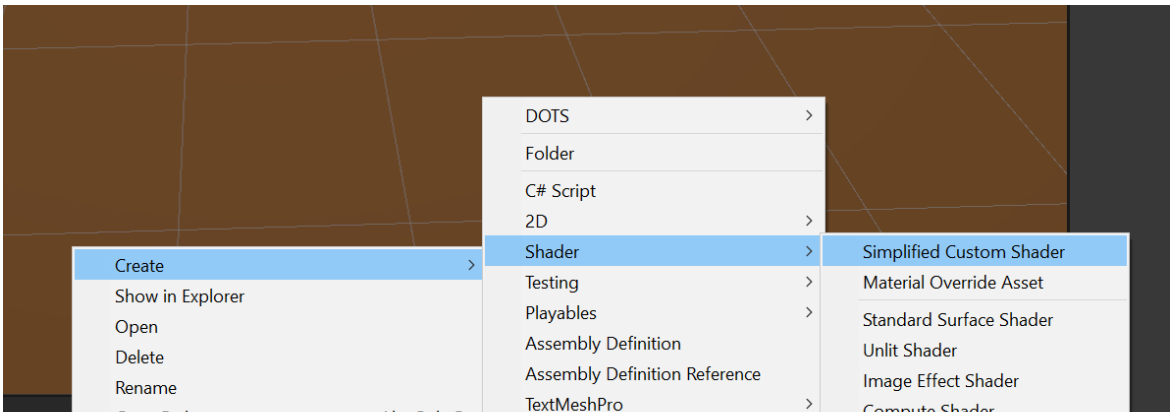
7.1.1 Justification

In the current version of the Unity HDRP, custom shaders are technically not supported. We do have access to the HDRP shader code however, and this allows us to inject our own custom shader API. Custom shaders are very desired because they allow us to have much more control over how certain things get rendered. They allow us to add custom effects or different parameterisations. Writing custom shaders for HDRP from scratch is a lot of work and not very easy, so some kind of simplification and tool to help the creation of custom shaders would be very useful.

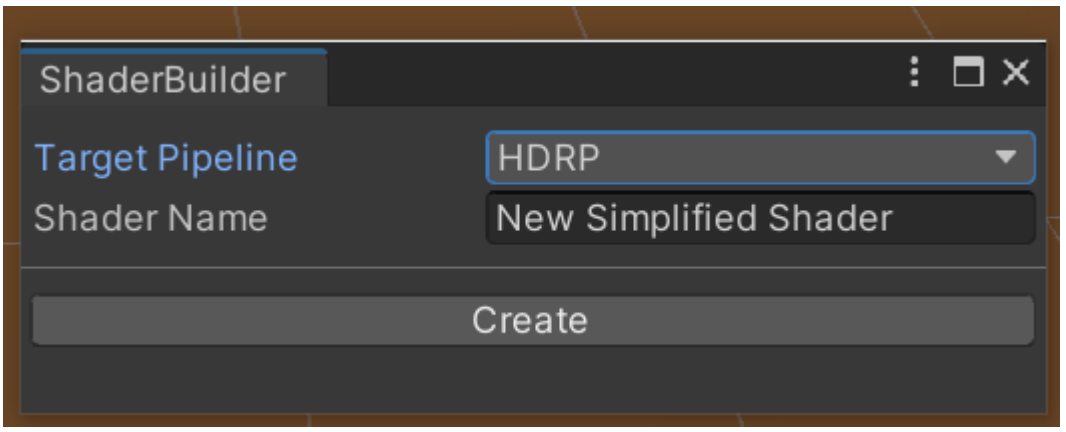
7.1.2 Usage

Create a custom shader

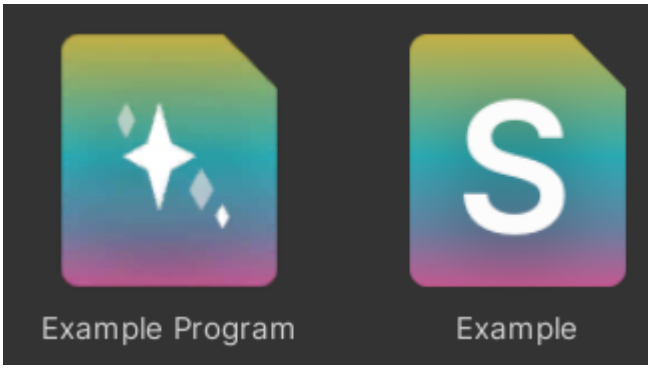
1. Find a folder to create the custom shader in.



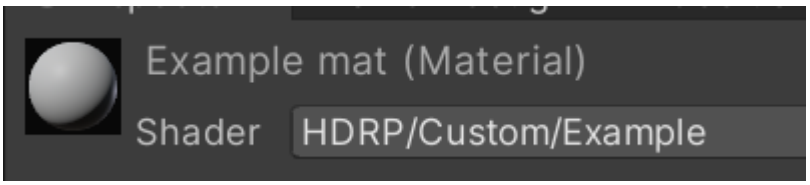
2. Right click in the project tab and click on ***Create>Shader>Simplified Custom Shader***.



3. Select HDRP in the target pipeline, and give the shader a name in the window that pops up. And click ***Create***.



4. The tool should create a brand new shader program and a Unity shader that uses that program.



5. You can use the shader by making a new material and assigning it the shader ***HDRP/Custom/{Shader Name}***.

Edit a custom shader

```
#if (SHADERPASS == SHADERPASS_FULL_SCREEN_DEBUG)
    #define DEBUG_DISPLAY
    #include "Packages/com.unity.render-pipelines.high-definition/Runtime/Debug/DebugDisplay.hlsl"
    #include "Packages/com.unity.render-pipelines.high-definition/Runtime/Debug/FullScreenDebug.hlsl"
#endif

#if (SHADERPASS == SHADERPASS_MOTION_VECTORS)
    #include "Packages/com.unity.render-pipelines.high-definition/Runtime/RenderPipeline/ShaderPass/MotionVectorVertexShaderCommon.hlsl"
#else
    #include "Packages/com.unity.render-pipelines.high-definition/Runtime/RenderPipeline/ShaderPass/VertMesh.hlsl"
#endif

#if (SHADERPASS != SHADERPASS_GBUFFER) && defined(WRITE_DECAL_BUFFER) && !defined(_DISABLE_DECALS)
    #include "Packages/com.unity.render-pipelines.high-definition/Runtime/Material/Decal/DecalPrepassBuffer.hlsl"
#endif

AttributesMesh VertexProgram(AttributesMesh input)
{
    // Vertex shader.
    return input;
}

void TesselationVertexProgram(inout VaryingsMeshToDS input, const OutputPatch<PackedVaryingsToDS, 3> patch, float3 baryCoords)
{
    // Tesselation Vertex shader.
}

void FragmentProgram(FragInputs input, float3 viewdir, inout PositionInputs posInput, inout SurfaceData surfaceData, inout BuiltinData builtinData)
{
    // Fragment shader.
}

#include "Assets/Simplified Custom Shaders/Base/HDRP/Vertex.hlsl"
#include "Assets/Simplified Custom Shaders/Base/HDRP/Tessellation.hlsl"

#if (SHADERPASS == SHADERPASS_FORWARD)
#include "Assets/Simplified Custom Shaders/Base/HDRP/FragmentForward.hlsl"
#else
#include "Assets/Simplified Custom Shaders/Base/HDRP/Fragment.hlsl"
#endif
```

Unity includes

Custom Shader Program

Simplification includes

If you open up the shader program created in the last step of ***Create a custom shader*** then you should see a shader program like shown above. You can ignore the ***Unity includes*** section and the ***Simplification includes*** section. You write your code in the ***Custom Shader Program*** section.

If you open up ***Assets>Simplified Custom Shader>Base>HDRP>CheatSheet.hlsl*** you can see the definition of all of the data structures used by Unity. It's a handy file to keep open while working on your own shaders.

```
/// <summary>
/// This is a cheat-sheet and not made actual use. Including thi
/// The definition of any commonly used unity shader data-struct
/// Author: Glyn Marcus Leine
/// </summary>

struct AttributesMesh
{
    float3 positionOS   : POSITION;
#ifdef ATTRIBUTES_NEED_NORMAL
    float3 normalOS     : NORMAL;
#endif
#ifdef ATTRIBUTES_NEED_TANGENT
    float4 tangentOS    : TANGENT; // Store sign in w
#endif
#ifdef ATTRIBUTES_NEED_TEXCOORD0
    float2 uv0          : TEXCOORD0;
#endif
#ifdef ATTRIBUTES_NEED_TEXCOORD1
    float2 uv1          : TEXCOORD1;
#endif
#ifdef ATTRIBUTES_NEED_TEXCOORD2
    float2 uv2          : TEXCOORD2;
#endif
#ifdef ATTRIBUTES_NEED_TEXCOORD3
    float2 uv3          : TEXCOORD3;
#endif
```

Vertex Shader

```
AttributesMesh VertexProgram(AttributesMesh input)
{
    // Vertex shader.
    return input;
}
```

If you look into the ***CheatSheet.hlsl*** file you can see the definition of ***AttributesMesh*** and see what values you have control over in the vertex shader. The input ***AttributesMesh*** will give you the default values that Unity will provide. The ***VertexProgram*** function will be called in between the Unity data collection, and the Unity default material shader code. So anything you change about the ***AttributeMesh*** before returning it from the function will be used in Unity's own lighting and vertex calculations.

```
struct AttributesMesh
{
    float3 positionOS   : POSITION;
#ifdef ATTRIBUTES_NEED_NORMAL
    float3 normalOS     : NORMAL;
#endif
#ifdef ATTRIBUTES_NEED_TANGENT
    float4 tangentOS    : TANGENT; // Store sign in w
#endif
#ifdef ATTRIBUTES_NEED_TEXCOORD0
    float2 uv0          : TEXCOORD0;
#endif
#ifdef ATTRIBUTES_NEED_TEXCOORD1
    float2 uv1          : TEXCOORD1;
#endif
#ifdef ATTRIBUTES_NEED_TEXCOORD2
    float2 uv2          : TEXCOORD2;
#endif
#ifdef ATTRIBUTES_NEED_TEXCOORD3
    float2 uv3          : TEXCOORD3;
#endif
#ifdef ATTRIBUTES_NEED_COLOR
    float4 color        : COLOR;
#endif

    UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

Tessellation Vertex Shader

```
void TesselationVertexProgram(inout VaryingsMeshToDS input, const OutputPatch<PackedVaryingsToDS, 3> patch, float3 baryCoords)
{
    // Tesselation Vertex shader.
}
```

The tessellation program offers you the input values for the current triangle and the output value for the current vertex. You can use the barycentric coordinates to interpolate the input values like this:

```
VaryingsToDS varying0 = UnpackVaryingsToDS(patch[0]);
VaryingsToDS varying1 = UnpackVaryingsToDS(patch[1]);
VaryingsToDS varying2 = UnpackVaryingsToDS(patch[2]);

VaryingsToDS varying = InterpolateWithBaryCoordsToDS(varying0, varying1, varying2, baryCoords);
```

Alternatively you can use the already interpolated values from ***VaryingsMeshToDS input*** and edit those values for the output. If no values of ***input*** have been changed then the unchanged interpolated values will be used by default.

Fragment Shader

```
void FragmentProgram(FragInputs input, float3 viewdir, inout PositionInputs posInput, inout SurfaceData surfaceData, inout BuiltinData builtinData)
{
    // Fragment shader.
}
```

The fragment program gets run in the fragment shader between the data collection and the Unity lighting code. You can affect the lighting code by changing the values in ***PositionInputs posInput***, ***SurfaceData surfaceData***, and/or ***BuiltinData builtinData***. The exact data in those structures can be found in the ***CheatSheet.hlsl*** file.

7.1.3 Workings

Shader

The simplified custom shaders tool comes with a standard Unity Shaderlab shader that references the custom shader code. The main shader handles the definitions of all the base parameters, shader passes, and base shader includes. The shader includes a custom shader code file called the shaderprogram.

In the shaderprogram you can write your own custom shader code to customize the default ***HDRP/LitTessellation*** shader passes. The shaderprogram also includes vertex, tessellation, and fragment shader code that will handle all the Unity default shader code invocation for Unity's built in shader features. The shader code included by the shaderprogram file will invoke the custom shader code written in the shaderprogram.

Editor

The editor code for the simplified custom shaders tool, provides an asset importer for the ***.shaderprogram*** asset, and a shader builder tool that handles the shaderprogram creation window.

7.2 Post Processing

There are 2 post processing preset profiles, one with, and one without ray tracing. Besides that they are configured to be as visually similar as possible.

Effects

Name	Summary	Ray tracing option
Ambient Occlusion	Uses scene geometry to figure out small creases where light can get stuck. These small places will be darker due to receiving less global illumination/ambient lighting.	Yes
Tonemapping	Remapping of HDR values to LDR values in order to display a larger range of light values and more accurate lighting.	No
Exposure	Remapping of light values of the final rendered image, a higher exposure results in a brighter image. For automatic	No

	exposure you can set it to use the automatic histogram, for a physically accurate exposure you can set it to use the physical camera settings on the camera itself.	
SubSurface Scattering	Simulate subsurface light interaction through ray tracing instead of traditional physically based rasterization estimations.	Always
Contact Shadows	Screen-space shadowing calculation using depth-buffer raymarching. More accurate shadows than conventional shadow mapping. Does not replace conventional shadow mapping.	No
Micro Shadows	Calculate additional shadowing from normal maps and ambient occlusion maps to estimate in-material shadows.	No
Physically Based Sky	Procedural generated skybox based on physical atmosphere light interaction. Can be used for ambient sky lighting as well.	No
Motion Blur	Mimic blurring that occurs when objects on screen move faster than the camera exposure time.	No
Vignette	Simulates the effect caused by improper lens housing, hoods, and filters, and improperly mounted secondary lenses.	No
Depth Of Field	Simulates the distance based blurring that is caused by objects being outside of the camera's focus range.	No
Panini Projection	Panini Project helps correct the visualisation of high field of view angle cameras on a flat 2D plane. Panini projection is a cylindrical projection, which means that it keeps vertical straight lines straight and vertical.	No
Chromatic Aberration	Simulates the colour separation around the edges of the image caused by cheap or lower quality lenses.	No
Screen Space Global Illumination	Simulates diffuse secondary light bounces for ambient lighting.	Yes
Screen Space Reflection	Simulates light reflection of specular secondary bounces.	Yes
Shadows	Override default shadow-mapping options	No

Visual environment	Override options for environmental lighting. Can enable the skybox to influence the lighting on objects.	No
Bloom	Bloom mimics the effect of light fringing around very bright objects that might overwhelm the camera sensor.	No
Streak	Lens flare effect that simulates the streaking caused with anamorphic lenses.	No