Realistic plant simulation

User manual

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

Table of contents

Requirements

Hardware

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

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

Software

- Unity 2020.3.11f1
- Houdini 18.0.597
- ROS Noetic
- WSL2
- Windows 10 or higher (DirectX and Nvidia drivers)

Project Setup

ROS

ROS Installation

Step 1: Getting your Ubuntu environment

ROS only runs natively on Linux operating systems, but because of some API specific features Unity uses we must be running a version of Windows 10. Therefore you must install WSL2(Windows Subsystem Linux 2), and the latest version of Ubuntu on that. WSL2 setup and installation.

Step 2: Installing ROS

For our purposes we will be installing ROS1 Noetic. Follow this tutorial to do this.
We recommend installing the `ros-noetic-desktop-full` config.

If you would like to become more familiar with the ROS environment here are some tutorials that do a great job of teaching you.

General Setup

Step 1: Starting ROS

Once you have ROS installed, it is time to set up our workspace.
First make a new directory and call it ‘ros_unity_ws’,
```
mkdir -p ~/ros_unity_ws/src
```
cd into the ‘src’ file of the new directory,
```
cd ~/ros_unity_ws/src
```
and then clone our the repo for the ros_unity_com package,
```
git clone https://github.com/Ragingram2/ros_unity_com-Package.git
```

Now to build the workspace make sure you are in the main section of workspace and enter `catkin_make` build command:
```
cd ~/ros_unity_ws
catkin_make
```

Once that is done we must set up the ROS environment. Enter `source devel/setup.bash` into the console. This command must be done every time you open a new console and want to develop on your package.

Step 2: Installing Required Packages

For this project we require two packages to be installed, cv_bridge and rosbridge_suite. These packages are already set up as dependencies in our config, therefore all you should need to do is run the following command:

```
rosdep install --from-paths src --ignore-src -r -y
```

Now you should be able to start running our nodes.

Running: Rosbridge

For a node to communicate with Unity you must first run the rosbridge node: `roslaunch ros_unity_com rosbridge.launch`

Running: Position Publisher

To use the Position Publisher node use the command: `roslaunch ros_unity_com position_pub_test.launch`

Running: Stream Listener

To use the Stream Listener node use the command: `roslaunch ros_unity_com streams_test.launch`

Houdini

To use the plant generator in the project you need to install the Houdini engine. This has a few weird nuances so follow this tutorial precisely.

Installing the Houdini launcher

- First install the Houdini launcher from the SideFX website: https://www.sidefx.com/download/
  To do this. You need to create an account.
- After creating your account go back to the downloads page and at the very bottom there is a link to download the new launcher (beta). Download and install that.

Installing Houdini

- To install houdini from the launcher, start the launcher as administrator.
- Go to the Houdini tab on the left. On the top right there should be a button labelled "***Install Houdini***".
- A window should pop up asking what version of Houdini you want to install. Click on "***Choose Another Version***".
- Find version 18.0.597 and click on ***install***.

- Houdini will now install. This takes some time

Getting licenses

This step is crucial to getting your Houdini instance working right.

- Go to the SideFX buy page: https://www.sidefx.com/buy/
- Then buy the Houdini indie version that is free.
- Fill in your details and go back to the buy page after you finish.
- Go to the Houdini Engine tab and buy the Houdini Engine for Unity.

- After you finish buying these licenses go back to the Houdini Launcher and log into your houdini account using the ***login*** button on the top right.

- After logging in go to the ***Licenses*** tab on the left.
- Make sure the license server is installed. If it is not you can install it using the "***Install License Server***" button on the top right. It doesn't matter which version you use. so just use the latest production version.
- Click on the ***License Administrator***.

Setting up licenses

- The license administrator should pop up after a few seconds. Make sure you are logged in by looking at the top right. If you need to log in, go to ***file*** and click on ***login***.
- Go to ***file*** and click on ***Install licenses*** and install all the licenses that are available by clicking on ***install***. This can take some time.

Common problems and fixes

- After startup you can experience a Houdini Engine Error in Unity. To fix this go to the ***HoudiniEngine*** button at the top, go to ***session*** and click on ***restart session***. This should fix the problem. If not, make sure all your licenses are installed.

Unity

Unity installation

Download Unity Hub and install it.
After installing, open the Unity Hub and then install the recommended Unity version by going to The download archive and clicking on the Unity Hub button next to the recommended version.

To download the project you need to set up Git LFS. ***Do not*** download the project as a .ZIP file from GitHub. That does not work!

Download Git Large File Storage from this link and install it.
Then open up a new command prompt after installing, and run “`git lfs install`”.

You can now clone the project from github using git clone.
Then run the command “`git lfs pull`”.

The large files should now be downloaded.

Go to the Unity Hub. Make sure you have a license installed. Otherwise Unity might not work.
Then go to ***Projects*** and click on ***Add***. Navigate to the location of the downloaded project. And select the RealisticPlantSim folder within the repository.

This should add the project to the Unity Hub. If this didn’t work. Make sure you select the folder that contains the Assets folder.

Open the project. It should start importing stuff. This might take a while for the first time.
Once the project is open, navigate to the ***Scenes*** folder and select the “***Farmland-test***” scene.

How to use

Within our project there's an example scene under ***"Assets/Scenes/Farmland-test.unity"*** which includes all of the scripts that are necessary to run and configure a simulation. We recommend using this scene as a starting point to configure your own simulation.

Terrain utility

Within the test scene there's a gameObject called "***TerrainGenerator***" this GameObject contains a component called "***Terrain Utility***".

The ***Terrain Utility*** script controls the most basic settings of the terrain which will be changed the most during different simulations. This script allows the user to regenerate the terrain and the plants in the inspector.

Terrain size X & Terrain size Z

The terrain ***width*** and ***length*** specify the size of the terrain that will be generated.

Chunks per frame

This variable defines how many chunks need to finish generating before calling the frame update method to redraw everything on screen, This helps achieve a better performance while generating a lot of chunks.

Plants per unit

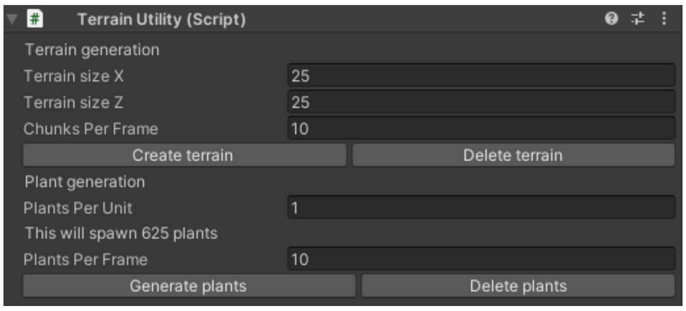
This value determines how many plants per 1x1 unit there should spawn.

Plants per frame

Defines how many plants should be generated before calling the frame update method.

Generate terrain and plants

The Create and delete terrain buttons will start generating the terrain in editor mode.
The same thing will happen for the plants when you click on the "***Generate plants***" and the "***Delete Plants***" button.



Terrain Utility (Script)

Terrain generation

Terrain size X	25
Terrain size Z	25
Chunks Per Frame	10

Plant generation

Plants Per Unit	1

This will spawn 625 plants

Plants Per Frame	10

Terrain generation

Within the test scene there's a gameObject called "***TerrainGenerator***" this GameObject contains a component called "***TerrainGenerator***" which has all the parameters needed for creating your own procedural terrain.

Terrain settings

Texture Width & Texture Height:

These values set the resolution width and height for the heightmap and normal textures of each chunk generated for the terrain.

Terrain Material:

The material attached here is the material that gets applied to each and every chunk.
Keep in mind that this material needs to use the "***HDRP/Custom/Terrain***" shader to work correctly with the terrain.

Base Height Texture & Base Height Normal Texture

The textures in here are the ***base height texture*** and ***base normal*** that will be combined/blended with the procedural height texture and normal map which will then be applied to the terrain chunk's material.

Normal Strength

The strength of the normal algorithm to generate normals from the procedural height maps.

Maximum Chunk Size

The maximum chunk size determines the maximum size one chunk can be in both width and length. (when generating terrain, the remaining width and length will consist of smaller chunks)

Noise settings

The procedural terrain is generated using a perlin noise algorithm. The Values within this part of the component control the settings for this perlin noise algorithm.

Seed

The seed used by the perlin noise function.

Perlin scale

The perlin scale sets the scale of the perlin noise

Perlin octaves

Perlin noise combines multiple functions called 'octaves' to produce natural looking surfaces. Each octave adds a layer of detail to the surface. The value of this parameter determines the amount of octaves in this generation.

Persistence

The persistence value determines how much each octave contributes to the outcome of the noise. A persistence of 1 means that all octaves contribute equally.

Lacunarity

Lacunarity determines the smoothness of the octaves, Lacunarity of 1 means that each octave will have the same level of detail.

Perlin Base Amplitude

The perlin noise base amplitude sets the minimum amplitude of the noise generated in the noise map.

XOffset & YOffset

These two values set the offset of the perlin noise, an xOffset of 1 basically means that the perlin noise has moved 1 unit to the right.

Include Sine Wave

When enabled the perlin noise will be combined with a sine wave to get a more wavy looking terrain.

Sine Amplitude

Sets the amplitude (height) of the sine wave in units.

Sin Period

The sine period controls the width of the waves of the farmland. a period of 1 means the waves will be 1 unit in width.

Perlin noise Weight

The ***perlin noise weight*** controls the blending factor with the sine wave, a weight of 0 means no contribution from our perlin noise, and a factor of 1 means the perlin noise will overcontribute the sine wave.

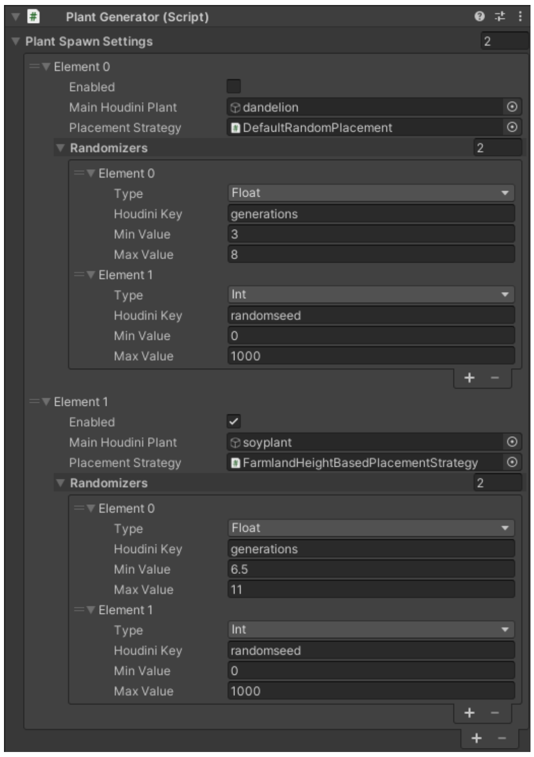
Plant generation

To generate plants in the scene we have a plant generator script. This script can be used to instantiate Houdini plants with varying parameters in the scene.

To start generating plants keep in mind that the terrain already needs to exist. The plant generator won't work if there is no terrain present.

To use this script create a new item in the Plant Spawn Settings list. Then assign the ***Main Houdini Plant***. This needs to be the gameobject that contains the main houdini script of the plant.

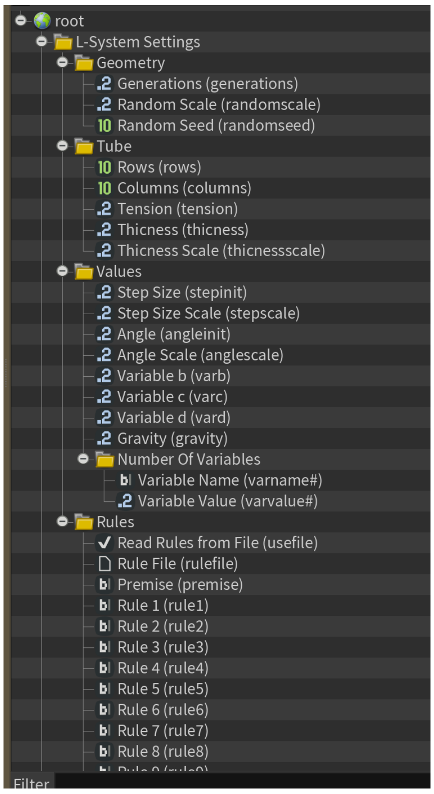
Then assign a ***placement strategy***. This decides how the plant should be placed in the scene. The placement strategies are described in the paragraph below.



Plant Generator (Script)
Plant Spawn Settings 2
Element 0
Enabled
Main Houdini Plant: dandelion
Placement Strategy: DefaultRandomPlacement
Randomizers 2
Element 0
Type: Float
Houdini Key: generations
Min Value: 3
Max Value: 8
Element 1
Type: Int
Houdini Key: randomseed
Min Value: 0
Max Value: 1000
Element 1
Enabled: ✓
Main Houdini Plant: soyplant
Placement Strategy: FarmlandHeightBasedPlacementStrategy
Randomizers 2
Element 0
Type: Float
Houdini Key: generations
Min Value: 6.5
Max Value: 11
Element 1
Type: Int
Houdini Key: randomseed
Min Value: 0
Max Value: 1000

Next you can assign the ***Randomizers***. These randomizers can change different Houdini parameters based on the Houdini Key, a ***min*** and a ***max*** value.
These values decide how different the plants will look. The keys are assigned in the Houdini plant file and need to be exactly matched in Unity.

The values below the plant spawn settings shouldn't be changed. These get set by the ***Terrain Utility*** script.

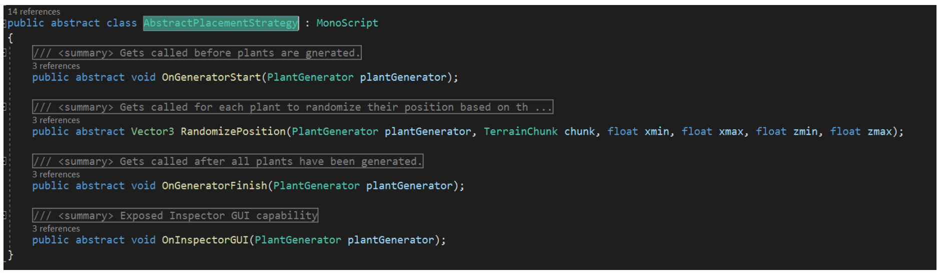


root
L-System Settings
Geometry
Generations (generations)
Random Scale (randomscale)
Random Seed (randomseed)
Tube
Rows (rows)
Columns (columns)
Tension (tension)
Thicness (thicness)
Thicness Scale (thicnessscale)
Values
Step Size (stepinit)
Step Size Scale (stepscale)
Angle (angleinit)
Angle Scale (anglescale)
Variable b (varb)
Variable c (varc)
Variable d (vard)
Gravity (gravity)
Number Of Variables
Variable Name (varname#)
Variable Value (varvalue#)
Rules
Read Rules from File (usefile)
Rule File (rulefile)
Premise (premise)
Rule 1 (rule1)
Rule 2 (rule2)
Rule 3 (rule3)
Rule 4 (rule4)
Rule 5 (rule5)
Rule 6 (rule6)
Rule 7 (rule7)
Rule 8 (rule8)
Filter

Plant placement strategies

Within the Plant spawn settings each plant has a field in which you can add placement strategy. Placement strategies are scripts that control how plants are placed on the terrain. Each plant can have a different placement strategy.

To create a placement strategy create a new script and let it inherit from "***AbstractPlacementStrategy***". You will see that your strategy class needs to implement 4 new methods:



```
14 references
public abstract class AbstractPlacementStrategy : MonoScript
{
    /// <summary> Gets called before plants are gnerated.
    3 references
    public abstract void OnGeneratorStart(PlantGenerator plantGenerator);

    /// <summary> Gets called for each plant to randomize their position based on th ...
    3 references
    public abstract Vector3 RandomizePosition(PlantGenerator plantGenerator, TerrainChunk chunk, float xmin, float xmax, float zmin, float zmax);

    /// <summary> Gets called after all plants have been generated.
    3 references
    public abstract void OnGeneratorFinish(PlantGenerator plantGenerator);

    /// <summary> Exposed Inspector GUI capability
    3 references
    public abstract void OnInspectorGUI(PlantGenerator plantGenerator);
}
```

OnGeneratorStart:

This method gets called before the plant is placed and passes the Generator settings.

RandomizePosition:

This method gets called when actually spawning the plant. This is the part where you will have to add your own logic for the placement. It expects a Vector3 which is the actual position the plant will be spawned. Keep in mind that this Vector3 is in world space.

OnGeneratorFinish:

This method gets called after the plant is placed and passes the Generator settings.

OnInspectorGUI:

In this method you will have to add anything that needs to be drawn in the inspector window. You could add custom variables for this strategy to show up in the PlantGenerator script.

Occlusion Culling

For performance reasons we want to make sure our occlusion culling works perfectly. Because of some issues with unity we have to do this manually after we have generated the terrain and our plants.

To do this select the terrain object from the hierarchy and mark it as “***static***” if unity asks if you want to mark all of his children as static as well, click ***yes***. Make sure all of the children are correctly marked as static by checking one of the plants

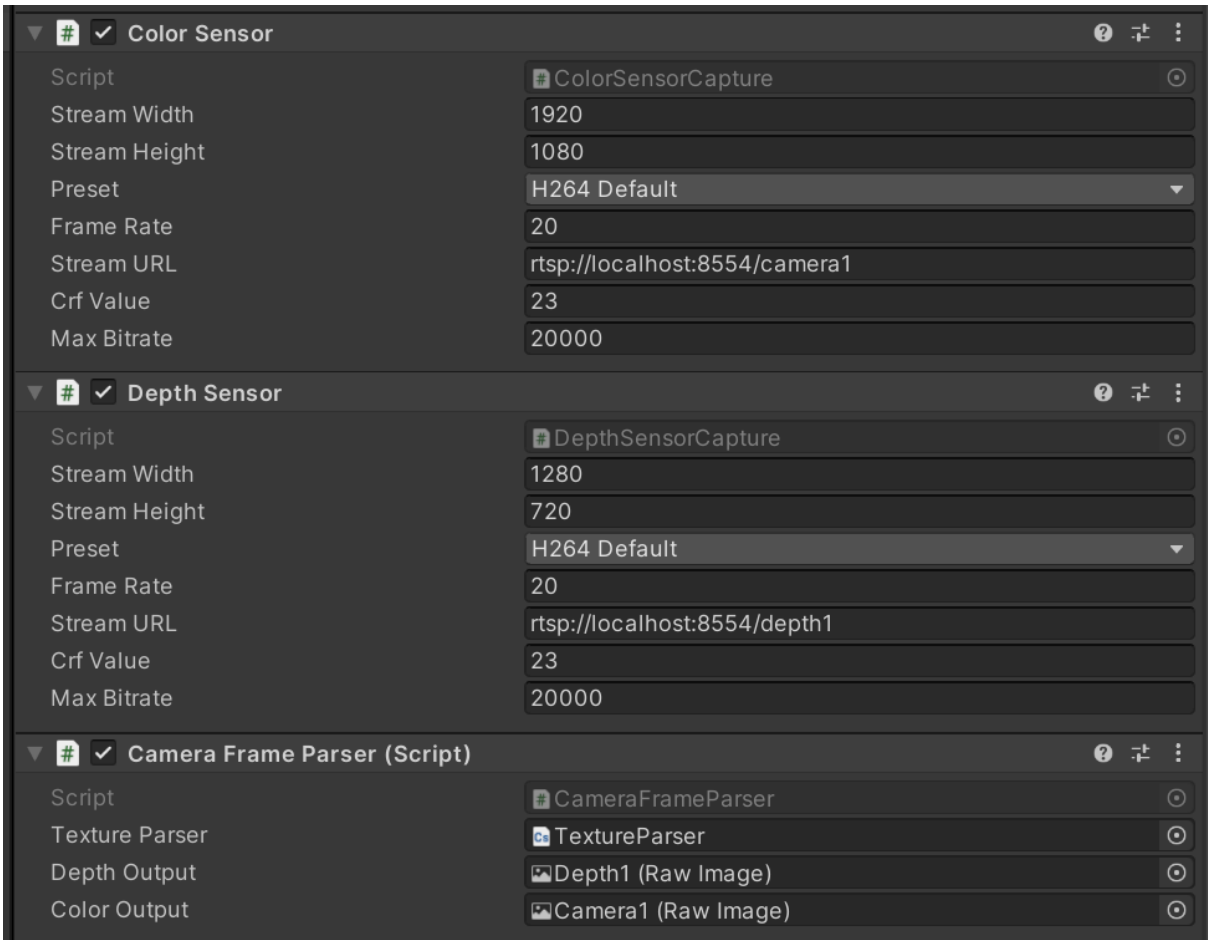
After marking the terrain and the plants as static go to “***Window->Rendering->Occlusion Culling***” and click on the ***bake*** button at the right bottom corner that shows up in the inspector tab.

Keep in mind that you will have to do this everytime the terrain changes!

Camera streaming

Setup a Camera for Streaming

If you would like to stream color and/ord depth information from a camera in Unity you must ensure that the camera has the "***Color Sensor***" and/or "***Depth Sensor***" and the "***Camera Frame Parser***" scripts attached to the camera, and that the ***Depth Output*** and ***Color Output*** are set to the proper UI elements in the "***Canvas***" gameobject. Also ensure that the "***TextureParser.compute***" is set properly.
To open up an existing camera, you can find them under the ***Robot*** gameobject.
See below



Color Sensor

Script	ColorSensorCapture
Stream Width	1920
Stream Height	1080
Preset	H264 Default
Frame Rate	20
Stream URL	rtsp://localhost:8554/camera1
Crf Value	23
Max Bitrate	20000

Depth Sensor

Script	DepthSensorCapture
Stream Width	1280
Stream Height	720
Preset	H264 Default
Frame Rate	20
Stream URL	rtsp://localhost:8554/depth1
Crf Value	23
Max Bitrate	20000

Camera Frame Parser (Script)

Script	CameraFrameParser
Texture Parser	TextureParser
Depth Output	Depth1 (Raw Image)
Color Output	Camera1 (Raw Image)

Before streaming ensure that the IP and port of the stream are set. The color and depth information may be streamed on the same IP and port as long as the directory paths are different. See Above.

ROS Communication

To run any demo that connects with ROS, ensure that the ***ROSConnector*** game object exists in the scene and that the IP and port match up with the IP and port listed in the ”***rosbridge.launch***” file in ROS, and that the rosbridge websocket node is running as well.

ROS Driving Demo

To run the ROS Driving Demo make sure that the “***Position Publisher***” and “***Position Subscriber***” scripts are attached and enabled to the ***ROSConnector*** game object. Now navigate to the robot model in the hierarchy, and ensure that the “***ROS Driving***” script is attached to it. If all of these scripts are in the scene and the rosbridge websocket node is running then we can start the simulator.

Once the console logs a successful connection, create a new ROS terminal and launch the “***position_pub_test.launch***” file. After a few moments the robot should start rotating and moving towards randomly chosen points in the scene. If you would like to adjust the rotational/linear speed of the robot, navigate to the robot model and adjust the values on the “***ROS Driving script***” (These values **can** be adjusted during playmode).

ROS Stream Listener

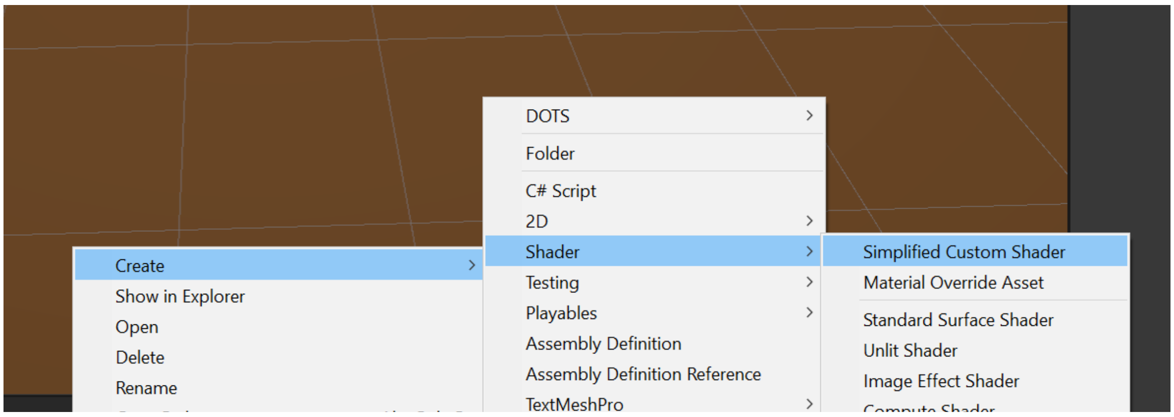
To set up the Stream Listener demo, ensure that each camera that you would like to stream is enabled in the scene, and has their settings correctly calibrated. Once that is done, launch the “***streams_test.launch***” file in a new terminal. After that node is done launching, run RVIZ and you should be able to see image topics for color and depth information.

Custom Shaders

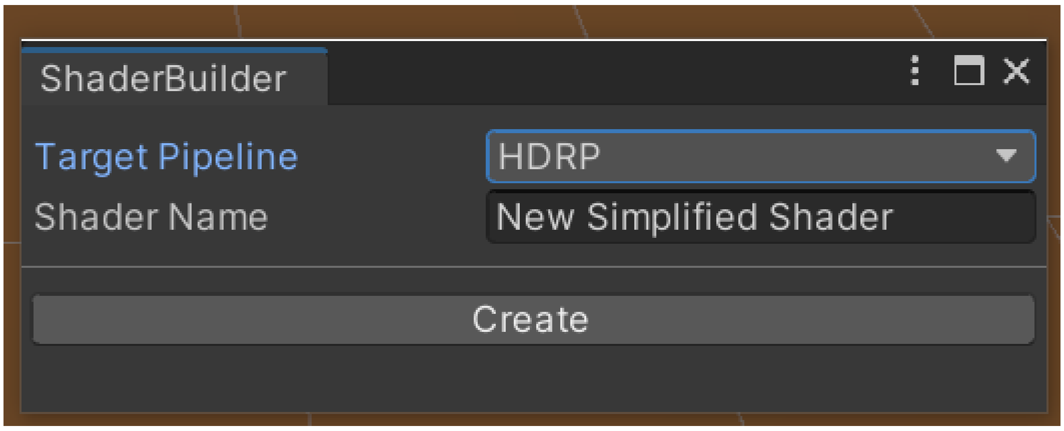
[DISCLAIMER] Custom shaders through hlsl allow a lot more control than shadergraph but also requires way more advanced knowledge of hlsl, computer graphics, and unity.
This is an advanced feature.

Create a custom shader

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



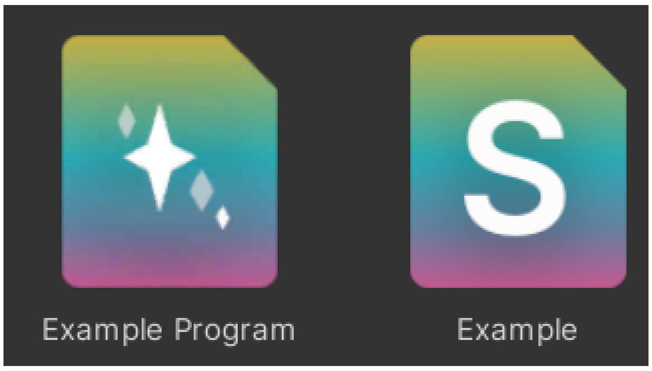
2. Right click in the project tab and click on ***Create>Shader>Simplified Custom Shader***.


ShaderBuilder

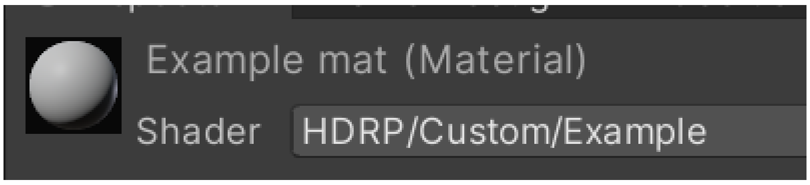
Target Pipeline: HDRP

Shader Name: New Simplified 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.

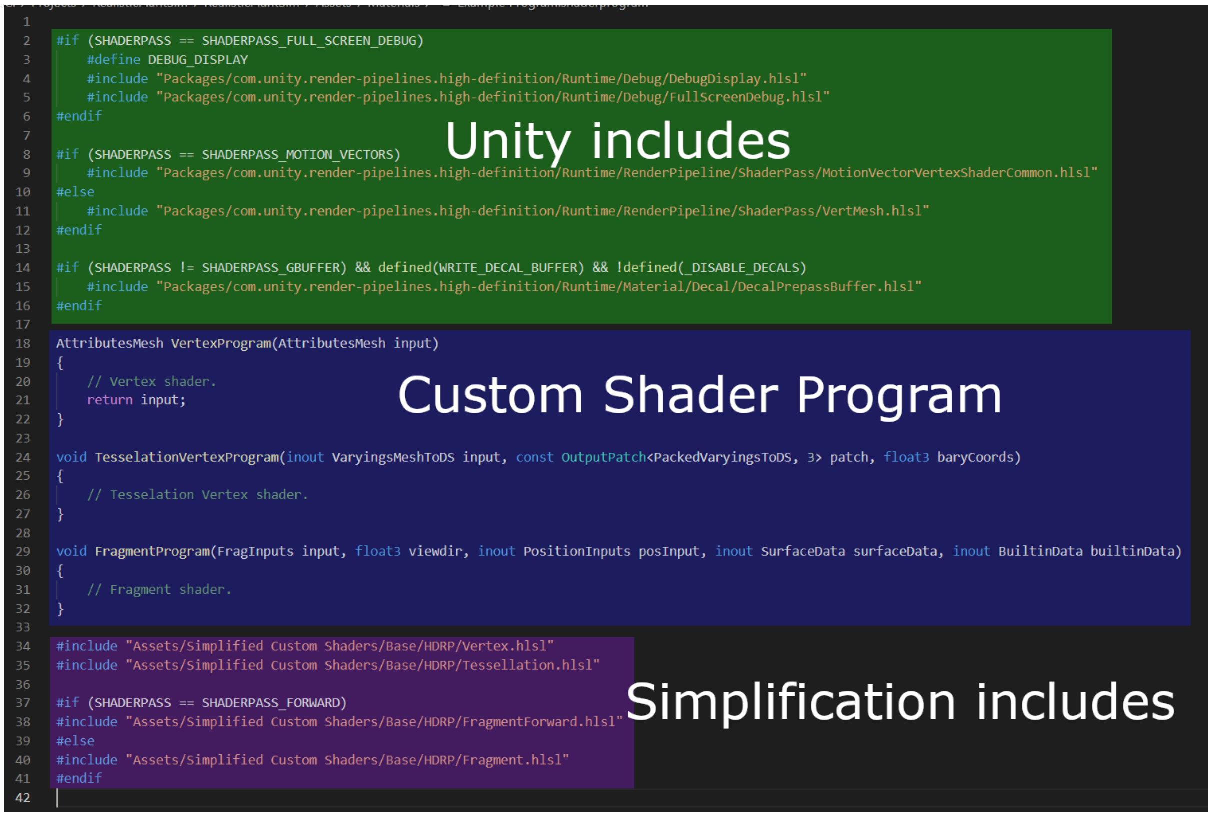


Example mat (Material)

Shader HDRP/Custom/Example

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

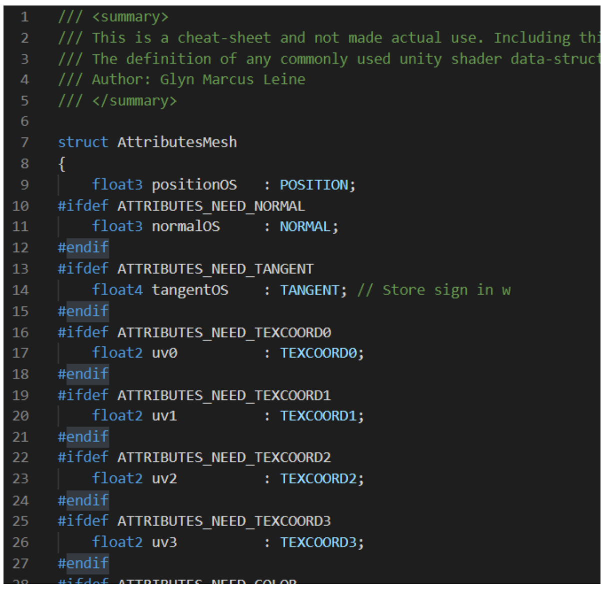
Edit a custom shader



```
#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

```

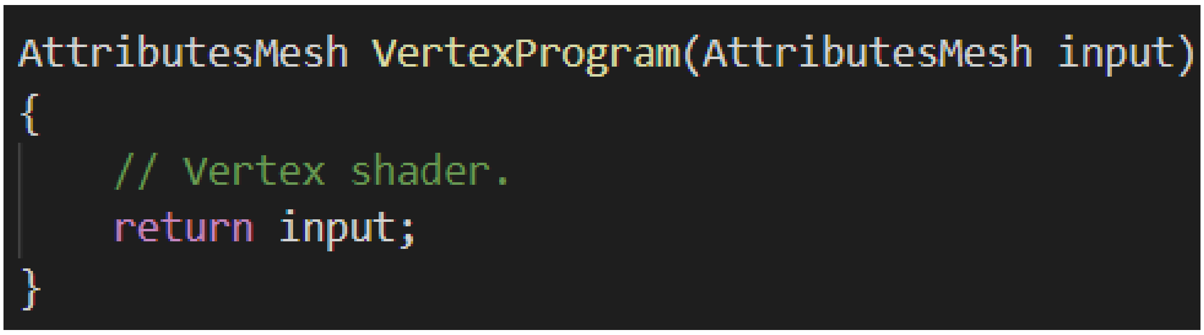
If you open up the shader program created in the last step of ***Create a custom shader*** then you should see a shader program like shown above. You can ignore the ***Unity includes*** section and the ***Simplification includes*** section. You write your code in the ***Custom Shader Program*** section.

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.



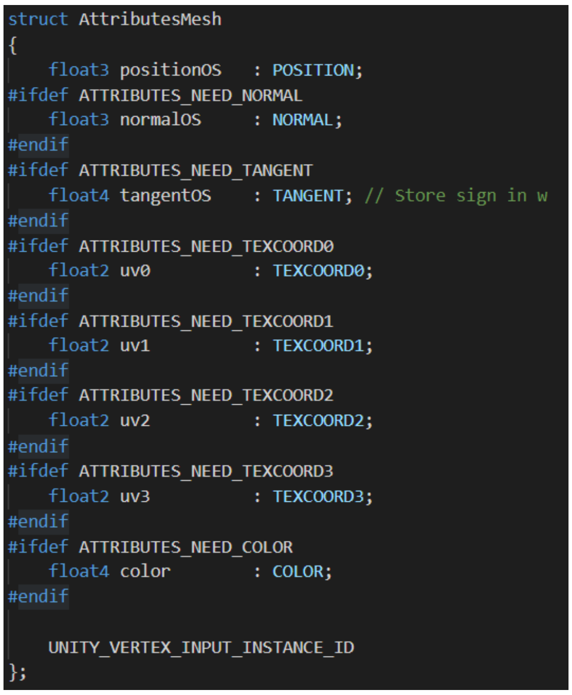
```
/// <summary>
/// This is a cheat-sheet and not made actual use. Including thi
/// The definition of any commonly used unity shader data-struct
/// Author: Glyn Marcus Leine
/// </summary>

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



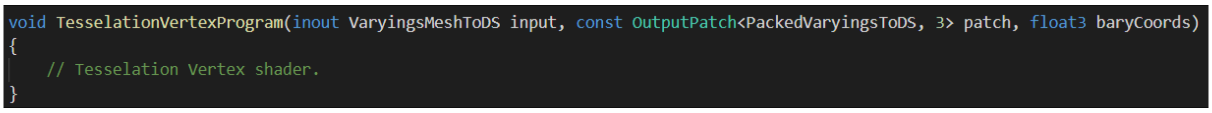
```
AttributesMesh VertexProgram(AttributesMesh input)
{
    // Vertex shader.
    return input;
}
```

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.



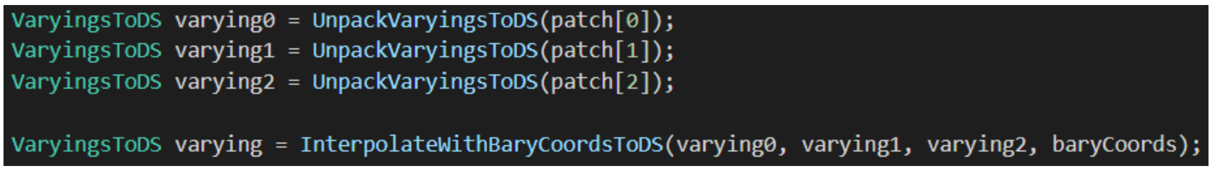
```
struct AttributesMesh
{
    float3 positionOS   : POSITION;
#ifdef ATTRIBUTES_NEED_NORMAL
    float3 normalOS     : NORMAL;
#endif
#ifdef ATTRIBUTES_NEED_TANGENT
    float4 tangentOS    : TANGENT; // Store sign in w
#endif
#ifdef ATTRIBUTES_NEED_TEXCOORD0
    float2 uv0          : TEXCOORD0;
#endif
#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.
}
```

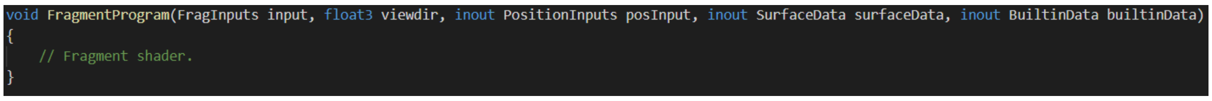
The tessellation program offers you the input values for the current triangle and the output value for the current vertex. You can use the barycentric coordinates to interpolate the input values like this:



```
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.