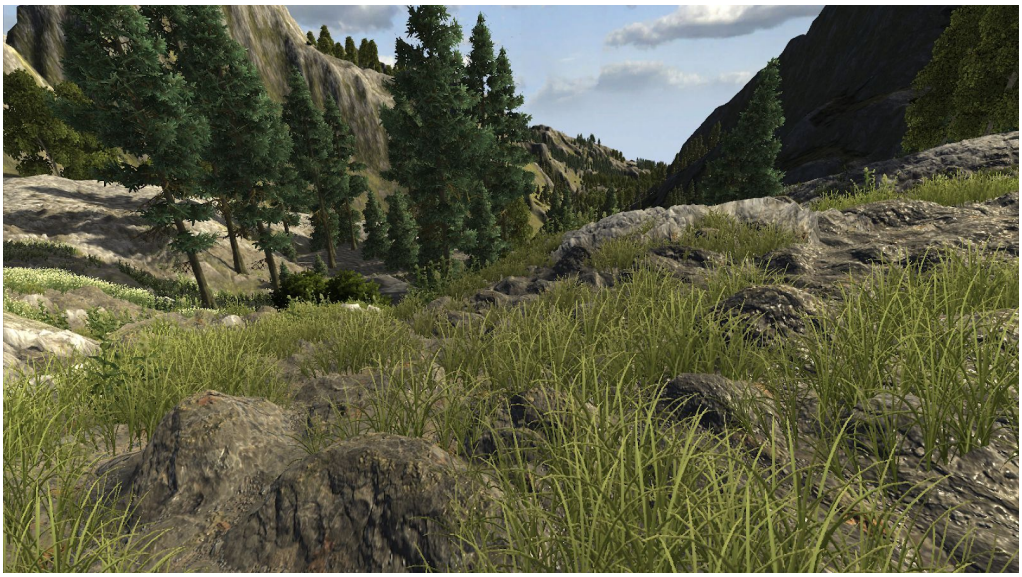
MicroSplat

Writing custom modules



Overview

MicroSplat is an incredibly flexible terrain generation system, which can be extended indefinitely with new features via new modules. The module system not only enabled this, but helps keep code clean and features reasonably independent of each other, helping features work in all combinations with minimal cross talk between features.

FeatureDescriptor

Every module has a C# class which inherits from FeatureDescriptor. FeatureDescriptor contains a number of abstract and virtual functions which can be used to inject your shader code into the shader, provide UI for the user, and serialize feature choices into the keyword structure. To create a module, create a C# class in an editor folder which inherits from FeatureDescriptor. You will want to open an existing module to compare code, but here's essentially how it all works.

First you will find a section of code which inserts a define into the build settings using the MicroSplatDefines.InitDefine function. This makes it possible to conditionally compile code when a module is installed.

Next you will find an enumeration usually called DefineFeatures, which contains a list of shader keywords. For example, lets imagine we are doing a feature called radar ping, so we might add a enum called _RADARPING. This will be defined in the shader code when radar ping is enabled, so that the shader can

```
#if _RADARPING
  DoSomething()
#endif
```

Further down you will find a bool or enum value that's used to store the state of that option locally. The Pack() and Unpack() methods will serialize this for us:

```
// in Pack()
if (radarPing)
{
  features.Add(GetFeatureName(DefineFeature._RADARPING));
}

// in Unpack()
```

```
radarPing = HasFeature(keywords, DefineFeature._RADARPING);
```

Note that the HasFeature and GetFeatureName functions need to be copied into new modules, as they cannot be included in the subclass since DefineFeatures is different for each module.

To show the option to the user, the add a toggle to the DrawFeatureGUI function:

```
radarPing = EditorGUILayout.Toggle("Radar Ping", radarPing);
```

Adding Code and properties

Now that the user can toggle your feature on and off, it's time to write some code. Often it's easier to store this in text files than to write the code in C#. MicroSplat will automatically scan the project for text files starting with the string "microsplat_" and pass them to each modules InitCompiler function. Here you can check the names and see if it's one you need, and assign them to local variables for easy printing into the shader later.

There are several functions in which you want to emit code:

WriteProperties

This function emits the property definitions for your material properties.

WritePerMaterialCBuffer

This function is for writing CBuffer data, which is an optimization on some platforms. Essentially, you want all non-texture variables declarations output here (float4 _MyVariable, etc).

WriteFunctions

This function allows you to emit functions and texture declarations you will need. In general, each feature simply calls a function from the main section of the shader. So we might have some function like “void DoRadarPing(float3 worldPosition, half4 albedo)” that we write here.

GUI

The material gui is handled in the DrawShaderGUI function. Note that the compiler may be actively compiling a shader while the GUI is drawing, so before displaying a section of code, use the mat.HasProperty function to make sure the material property exists before displaying editor code. Additionally, the MicroSplatUtilities class has a DrawRollup function which can be used to organize your modules properties under a rollout.

Other Stuff

The ComputeSampleCounts function should be filled out giving the user a rough idea of how many texture samples your feature will add to the shader. The DrawPerTextureGUI function can be used to draw per texture properties. Note that these also define a shader feature so that they are compiled out when not in use, and are stored in a 32x32 color array and pushed to the shader as a texture. Consult MicroSplatPropData for the layout and open slots. There are also function to move your module up or down in the compile or display order.

Calling your code

Now that your functions, properties, and variables can be written into the shader, you need to call your function from tha main shader. The easiest way is to insert some code into the microsplat_terrain_body.txt which looks like so:

```
#if _RADARPING
  DoRadarPing(i.worldPos, albedo, normSAO);
#endif
```

However, you could also add a unique comment string to where your function needs to be inserted and replace that in the modules OnPostGeneration function. This has the advantage of not needing to change the core module when your function signature changes.

Shader Structure

The main files of the shader are the microsplat_terrain_body.txt file and microsplat_shared.txt. You will find most structure definitions and common functions in the shared file, and the actual shader layout in the terrain_body file.

In the terrain_body file, you will find functions for sampling each of the arrays. Below that you will find a Sample function, which is the main flow of the shader. This function is passed the following data:

Input

this is the input from the vertex to fragment shader, and contains things like the original uvs, normal, world position, tangent space view direction (i.viewDir), etc.

Config

The config contains UV's for each of the 4 texture sets which will be sampled. Note these are float3's with the third index being the texture index. There's some additional information there for texture clustering as well, and a TriplanarConfig when triplanar is enabled.

MicroSplatLayer

The function returns the MicroSplatLayer, which is similar to structures like SurfaceStandardOutput in Unity's shader. It contains albedo, normal, etc.

This Sample function roughly does the following:

- Sample FX data (streams/lava)
- Setup UVs for each texture (per texture UV scale, etc)
- Setup sampling structs and cull low weighted textures (Setup function called)
- Sample various arrays (albedo/normal/etc) and setting up the RawSamples structures, which contain the unmixed samples from the textures
- Apply per-texture based effects before samples are mixed
- Mix samples
- Apply effects to mixed samples (snow, etc)
- Return final albedo/normal/etc in MicroSplatLayer structure

For most modules this is all that needs to be modified. However, if your module interacts with other features (like tessellation sampling, distance resampling, etc), it may need to have additional code in other parts of the shader.

HDRP/URP

In most cases, support for HDRP/URP is automatic. However, not all surface shader conventions are translated.