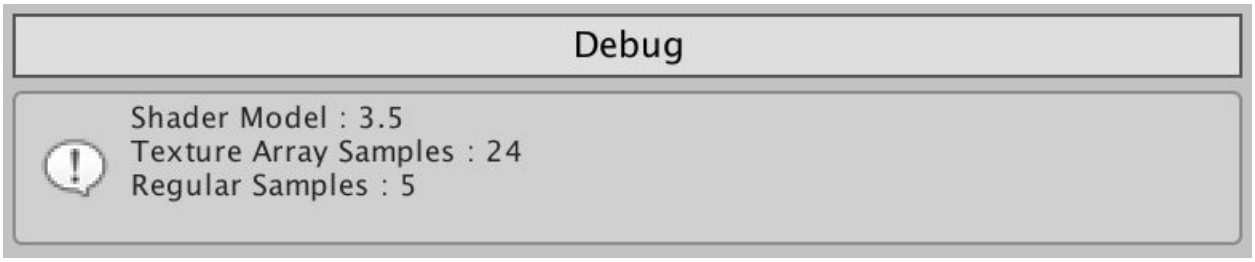
MicroSplat

Optimization Guide

MicroSplat is an extremely fast and configurable shader system, with many tradeoffs for optimization purposes. In general, any options are listed in cost order, from least costly to most costly.

You can use the debug section to see the cost of your shader in terms of texture samples as you adjust shader settings.



While all samples are not the same, generally speaking more samples will be more costly. There are also some less obvious optimization strategies listed below if your really trying to squeeze the last bits of performance on low end devices.

MicroSplat is designed to use minimal CPU time- in most cases, none. So mostly this documentation is about GPU optimization. A quick test to tell if you are bottlenecked by the CPU or GPU is to adjust your screen size- if your screen size is reduced and you do not see a change in performance, you are bottleneck by the CPU.

All Samples are not the same?

There are several factors that control the cost of a texture sample. The major one is the size of the texture, but note that is view dependent. When a texture is far away, it uses a lower res version of itself which is much cheaper to sample (mip mapping). The second is based on the cache. When a texture is sampled, a section of it is loaded into GPU memory based on the size of the texture and UV coordinate of the sample. If it's compressed, a very small section of it is decompressed. Assuming the texture is filtered (bilinear or trilinear), several subsamples are taken and blended for the final result.

If a subsequent sample is needed for a nearby pixel, that data is usually already in the cache and possibly decompressed, making that sample much, much faster than a previous one.

Set the Max Texture Count correctly

One secret to MicroSplat speed is how it approaches sampling textures. There are two dominant strategies used in terrain shaders in Unity- the first approach is what Unity uses, which is to draw the terrain one time for every 4 textures in use on the terrain. Each pass is given a splat map for the 4 weights of that texture, and the result is blended in with the previous pass.

The second approach is what other texture array shaders seems to use (except MegaSplat, which is it's own technique). The terrain is drawn only once, but if a terrain has 16 textures on it, it would sample 4 splat maps for the weights, then all 16 texture sets using whatever features are enabled. With something like triplanar and distance resampling enabled, this would be 4 + (16 * 2 * 3 * 2) samples per pixel, or 196 samples per pixel.

MicroSplat uses a very different and new technique. Each pixel samples the splat maps, then sorts and culls the weights so that only the 4 highest weighted textures are sampled per pixel. Then only those texture sets need to be sample. In the same 16 texture terrain, with triplanar and distance resampling, MicroSplat would use 4 + (4 * 2 * 3 * 1.5) samples, which is 40 texture samples in total. If you boosted that to 32 textures, it would only add 4 more samples to that for the weights, where as the traditional techniques would double the sample count to 392 samples per pixel, and insane amount.

The primary cost of this technique is sorting the samples. This is costly for 2 reasons- the first is that sorting on a GPU requires a lot of branching. The second is that this work all has to be completed before textures can start to be sampled. However, the massive memory bandwidth savings are almost always a huge win vs this cost.

If you set this to 16 textures but are only using 4 terrain textures, it will still try to sort the 16 weights, even though you are not using the other textures. So setting this to the lowest value you can may provide a speedup.

Note that when set to a max of 4 textures, sorting can sometimes be bypassed altogether, but this will depend on other optimizations which rely on the textures being sorted. Features like distance resampling (which only operates on the top 2 most dominant textures in fast mode), Blend Quality not being on Best (other modes skip sampling lower weighted textures), will prevent this optimization and cause the samples to be sorted.

Branch Samples

The optimization above does a wonderful job of statically culling extra samples that aren't needed. But we can go one step further by enabling the branch samples option. This option uses dynamic branching on the GPU to cull additional samples based on the weights of the current pixel. If you are using features which multiply the number of samples, such as triplanar, texture clustering or stochastic sampling, or distance resampling, then using Branch Samples can be a rather large win. This will depend both on the GPU your using, the features you have enabled, and the painting on your terrain, and even things like contrast settings on the height blending. However, in the included demo environment with Stochastic, Triplanar and Distance resampling enabled, frame rate improves from 26fps to 56fps with Branch Samples enabled on my MacBook Pro with a Radion 560X graphics card.

There are two options available, Basic and Aggressive. In Basic mode samples with 0 weight from the terrain's splat maps are culled; essentially any area which just shows one or two textures will only sample those texture sets. In Aggressive mode, additional samples performed in triplanar and stochastic modes are culled based on their individual weights; in other words, if a stochastic sample or triplanar projection isn't going to be used on this pixel and that can be predicted, the sample will be culled.

A detailed video of these optimizations is available here:

http://shorturl.at/quNPS

Source Texture Size

On the texture array config, there is a source texture size option. By default, MicroSplat keeps the original textures on the terrain, in case other tools use them. However, these textures are not used for rendering, as the rendering is done through custom packed Texture Arrays. This option will let you generate small copies of those textures to use instead, saving considerable memory. In most cases, external code only cares about which texture you are on (for collisions or footsteps, not the actual texture data used, so a 16x16 version of that texture is perfectly adequate. There is also a 256x256 option, in case textures at a lower resolution will suffice.

Compressed Splat Maps

By default, unity uses uncompressed and read/write splat maps. This means a copy of the textures are stored on both the GPU and CPU, and can be quite large. A terrain with 16 textures on it will generate 4 splat maps, which if they are set to be 1024 in size means 4 megs of CPU memory and 4 megs of GPU memory. However, if you have multiple terrains and higher resolutions, this can add up quite quickly. Additionally, sampling uncompressed textures is more expensive than compressed (due to cache size limits), and this work has to be completed before the shader can determine which textures need to be sampled from the texture arrays.

DXT5 compression is a 4:1 ratio, and some mobile compression formats are even higher. So compressing these textures can save a lot of memory and speed up the shader. To do this, you can use the export tool on the MicroSplatTerrain component to export the splat maps from the terrain to regular textures on disk. Then you can turn on the Custom Splat Maps option in the shader and assign those textures, which can now be compressed through the standard texture importer settings. Then on your terrain, you can set the splat map size to be very small (16x16). Note that if you try to access this data through the terrain API it will use the 16x16 version of this data, and likely not be useful. You will also not be able to paint or edit textures in this mode, so it's best to do this type of work in a script that runs before the build.

Turn off unnecessary per-texture properties

Per texture properties can have real cost when you use a lot of them. They are stored in a small, cache friendly look up table, so reasonably cheap to use. However, UV scale must be completed before texture sampling can begin because it modifies the UVs. And HSL adjustments have a fair bit of math. In some cases these properties are for convenience of editing, but could easily be baked into the texture data instead (tint, normal strength, etc). So removing any of these can help speed the shader up a little bit.

Feature Cost

Turning a feature on or off will change the display of sample counts in the debug section of the shader. The most expensive features are the ones which multiply the number of sample's needed. For instance, triplanar requires 3 samples per texture set. Stochastic Sampling requires 3 as well. Combined they require 9 times the samples as without those features.

Options like Distance Resampling contain multiple versions which increase sample counts from a multiple of 1.25 to 2, and can either work only on the albedo or all channels. These can greatly help reduce the number of samples you need.

MicroTriangles

Generally speaking, a shader like MicroSplat tends to be memory bound- meaning texture samples are the primary cost, especially on lower end hardware. However, this is very view dependent as lower mip maps are faster to lookup as they fit better in the cache. However, there are other ways in which a shader can be bottlenecked.

People tend to relate 'polygon' count to performance, but generally speaking that is long a thing of the past. Even transforming a million vertices takes a fraction of the time drawing pixels does- as pixels tend to be much more expensive than vertices, and often the pixels on a screen and written to many times over (transparency, lighting, post processing, etc). What really matters is shading rate, and the vertex count can affect the shading rate of pixels.

When triangles become very small on screen, less than ten pixels in size, you run into what is called the Micro-Triangle issue. This is actually why high tessellation is an issue- not the actual cost of the added vertices. This is because GPUs shade pixels in a quad of 4 pixels at a time, and the threads on a GPU handle some number of these quads. When an edge of a triangle is encountered, it will still shade all 4 of those pixels, but may not use the results from 3 of them because those 3 are on another triangle. When triangles become very small, you have many more edges, and more pixel work gets discarded. This can easily reduce the shading rate to 1/4th of what it normally would be. And when triangles get smaller than a pixel in size, it can throw all 4 samples away, and the shading rate can get much lower. In the worst cases, the shader might not be able to transform vertices fast enough to feed the pixel shader, bottlenecking even more, because it has to transform many vertices just to shade a single pixel on the screen. It is impossible to prevent micro triangles, as simply viewing a triangle edge on will produce them. But keeping triangles a reasonable size will help shading rate.

Using per-pixel normals can allow you to raise the pixel error on the terrain, and using reasonable tessellation parameters as well, will all help keep the shading rate efficient.

Texture Size

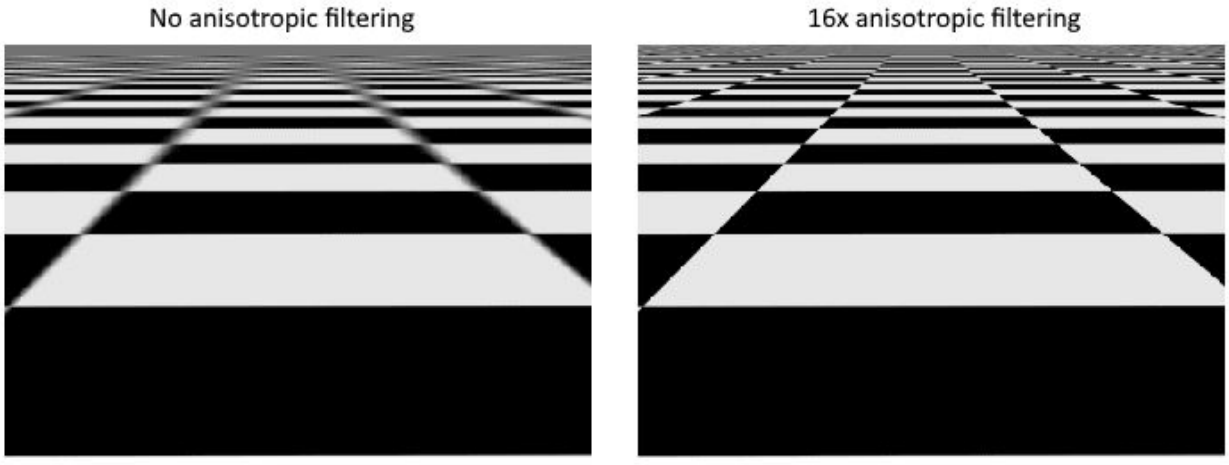
I've talked a little about texture size. I've noticed people are crazy about using 2k or 4k textures on terrain, but this isn't always best. Larger textures provide more resolution, but you might consider using smaller textures stretched over a smaller area of terrain, as this gives you the resolution boost you crave without the memory and sampling cost. Then you will have the problem of the texture tiling more- which can be overcome by techniques like Stochastic Texturing or Texture Clusters, synthesizing a surface which is higher resolution (from the standpoint of tiling) than simply increasing texture resolution would provide.

Techniques like detail noise can also increase the resolution perception of certain textures. Many textures, like sand and dirt, are effectively noise when zoomed close. So adding a detail noise to them will simulate missing detail, often at a reasonable cost compared to

higher resolution textures. These are a single sample each, as is the single noise normal option, which can be used to make the overall surface size look much larger than it really is. While it is tempting to always use stochastic sampling for it's high quality, these very cheap options are quite practical for lower end devices, or even in combination with other effects.

Finally, consider that while all textures in an array need to be the same size, the arrays themselves do not. Your diffuse/height array might be able to be much lower resolution than your normal/smoothness/ao array is and still look fine (or vice versa).

Texture Filtering



Texture filtering also has a cost. By default bilinear with an anisotropic value of 1, which is very cheap. But great quality improvements can be made using trilinear and setting anisotropic higher- especially on normals, which suffer the most from lack of these options. And like texture res, each array can have it's own settings for these values. Having trilinear or some level of anisotropy on just the normal array can greatly improve the overall quality.