Introduction to FidelityFX Variable Shading

Motivation

One of the major performance bottlenecks, when rendering a frame in modern games, is the high computational power required to shade every pixel at high resolutions. A side effect of rendering at high resolution is that the world space position of neighboring fragments within the same primitive are typically quite close, and the color value resulting from adjacent pixel shader executions in the same primitive often produce visibly similar results.

Geometry edges typically contain more important visual information than the insides of primitives. Multi-sample anti-aliasing (MSAA) takes advantage of this by keeping the number of pixel shader executions low, relative to the increase in fragments used to compute the final frame.

Variable Rate Shading (VRS) provides a similar way to reduce the number of pixel shader executions and reused the same output value for multiple neighboring pixels, if they are covered by the same primitive. This can significantly reduce the amount of ALU and bandwidth required to compute a frame and improve frame rate without a noticeable loss of visual quality, which means the GPU time saved can then be repurposed for effects that have a bigger impact on the final frame, if you have room in your frame budget.

One way to think about how VRS works is to think of MSAA: if a 4K frame is rendered with 2x2 VRS for everything, the required computational power and the final quality of the image would be comparable to rendering a 1080p image with 4xMSAA enabled. However, VRS provides additional features to enable more fine-grained control over which parts of the frame should get rendered with lower resolution.

Implementing Variable Rate Shading

Cases allowing VRS

Multiple cases exist where VRS can be applied without significantly impacting the quality of the final image:

- Distant objects are usually more defined by their geometry, rather than by texture detail (using lower MIP levels) and color variance of distant pixels is often further reduced by fog, atmospheric effects or semi-transparent geometry (e.g. particle systems).
- Some objects may be known to be out of focus or known to be blurred by postprocessing effects like motion blur or bloom.
- Some objects may get distorted or blurred by being behind haze or semi-transparent objects like water or frosted glass.
- Some objects might be known to have little detail variance (such as rendering for toon shaded games) or due to being in very dark parts of the scene (e.g. the unlit or shadowed parts of objects in scenes with little ambient light).

- In fast moving scenes, high framerate and low input lag are important, but small details are less likely to get noticed by the player, so aggressively using VRS can help to achieve the performance goals.

In addition to the cases mentioned above, some pixels might be known to be of little interest to the player, either through eye-tracking (foveated rendering) or other systems, or because the game design aims to steer the focus of the player to certain parts of the screen. As an example, the game might choose to reduce the shading rate on the background geometry but make sure all enemies are rendered at highest quality.

Due to saving computational power by focusing usage of GPU resources where it matters most, VRS can be used to make sure target frame times are achieved (similar to dynamic resolution scaling but with more fine-grained control over where detail needs to be preserved), as well as for power saving on portable devices without noticeably sacrificing image quality.

VRS Control Options

VRS support in DirectX12 comes in 2 tiers:

- Tier 1 allows setting a shading rate per draw call. This way the shading rate can be adjusted based on object type (importance), distance to camera, movement speed or screen space position of the object.
- Tier 2 adds two additional techniques for more fine-grained control over the shading rate:
    - `SV_ShadingRate` can be exported from the vertex or geometry shader to control shading rate at a per-primitive level. This way, for example, primitives facing away from the main light source, fast moving parts of a skinned mesh, or simply based on artist specification in a vertex attribute, can get instructed to be shaded at a lower rate.
    - A VRS control image containing information which VRS rate to use on a per screen tile basis, can be bound to specify a VRS rate per screen region. This can be used to control the shading independent of the geometric granularity of the objects rendered, based on screen region, user focus area, or by analyzing the previous frame's final back buffer to compute which regions of the screen are likely not to suffer much from reduced shading rates.

All three modes can be used simultaneously: Tier 2 defines a combiner tree which determines how the result of each state should be combined with the result of the previous state to compute the final shading rate. The options for each combiner are: passthrough the previous state (i.e. disable the current stage), override (ignore previous stages), min, max and sum.

Technical Details

VRS works similarly to MSAA, where the pixel shader usually gets executed once per pixel and the resulting value gets written to all fragments of a pixel which are covered by the primitive. VRS executes one pixel shader thread per VRS coarse pixel, which can contain 1, 2 or 4 pixels (or up to 16 if additional shading rates are supported), and then writes the result to all pixels within the coarse pixel region that are covered by the primitive.

This can significantly improve performance in cases where the scene mainly consists of large polygons using expensive pixel shaders. Performance for draw calls containing lots of very small primitives will profit less from VRS, but at the same time visual quality will not be impacted. Also, since VRS does not reduce the amount of pixels which get written to the rendertargets, fillrate will not get reduced by enabling VRS, so using VRS in fillrate bound passes is not recommended.

Likewise, since depth and stencil values are usually computed and written before the pixel shader gets executed. In most cases depth-only passes do not have a pixel shader, or at least not an expensive one, so depth-only passes (like shadow map rendering) will not profit from VRS.

When working with depth values and VRS, it's important to note that the sample position of a pixel shader may be different than the pixel centre for which the depth value got stored in the depth buffer. When rendering geometry that reads the depth buffer, like decals or soft particles, it has to be taken into account that the difference between values read from the depth buffer, and the depth value of the pixel passed from the vertex shader, may be considerably larger than without VRS enabled.

Finally, since the centre of the VRS tile may be outside the rendered geometry, centroid interpolation should be enabled in the pixel shader when using VRS, otherwise this can cause artefacts like sampling outside the region of a texture map that is intended to be used for a primitive. Using centroid interpolation will ensure the barycentric coordinates, used to interpolate vertex attributes, are always within the area covered by the primitive.

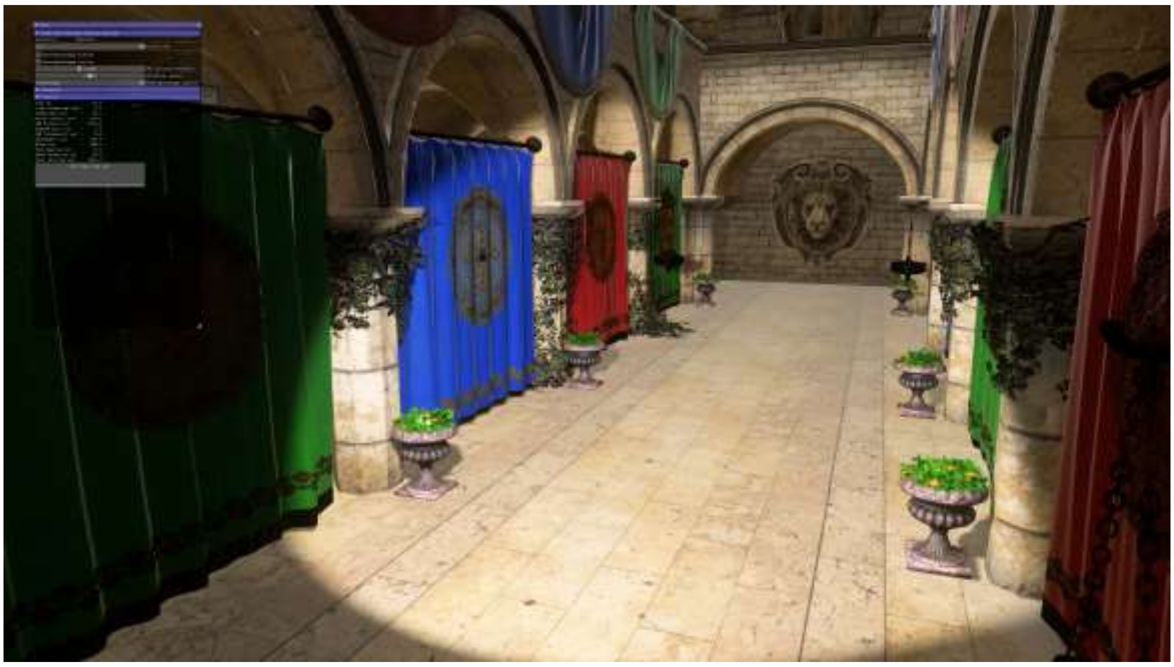
Should the information on which pixels of a VRS tile are covered by the geometry be required in the pixel shader, e.g. for alpha-tested geometry, they can be evaluated by reading `SV_Coverage`. Exporting the coverage mask from the pixel shader is also possible, however it is not recommended for performance reasons.

FidelityFX Variable Shading

The FidelityFX Variable Shading Sample is based on the AMD glTFSample (https://github.com/GPUOpen-LibrariesAndSDKs/glTFSample) and uses the AMD Cauldron framework (https://github.com/GPUOpen-LibrariesAndSDKs/Cauldron).

The glTFSample has been extended to add VRS through FidelityFX Variable Shading, and the UI has been extended to provide a platform to experiment with the VRS and FidelityFX Variable Shading setting and easily prototype ideas for how to utilize VRS, in scenes that are representative of games.

Sponza scene



The sample uses the Sponza scene, a scene commonly used for visualizing new visual effects. The scene was chosen since it contains a high variance of texture details and lighting conditions, making it suitable for visualizing where VRS can be applied and how the algorithm works.

Controls & Options

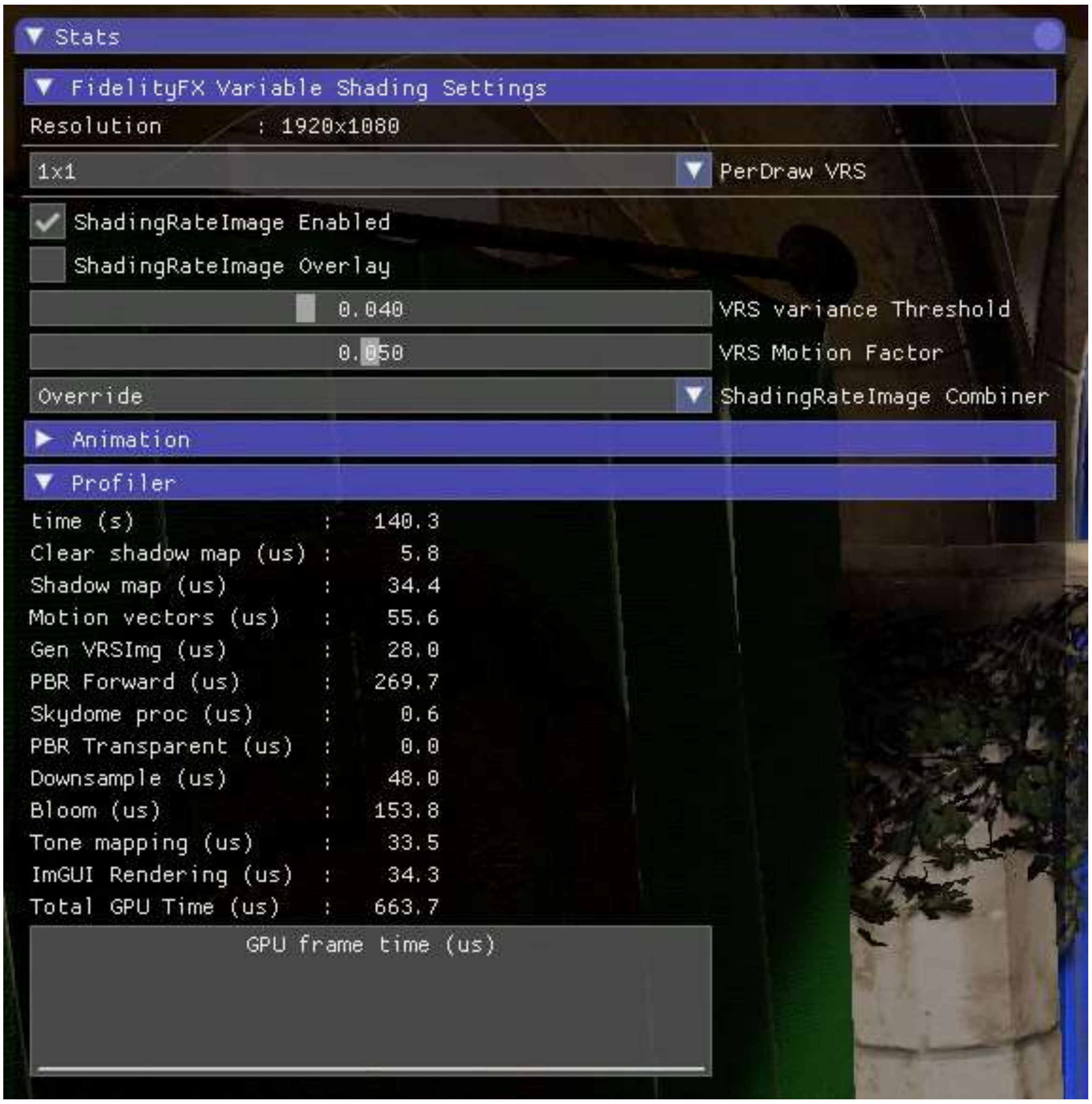
Allow additional VRS Rates

In addition to the 2 tiers, VRS in DirectX12 supports some extra capabilities. These indicate which granularity is used for the VRS image (one shading rate value per 8x8 screen pixels on AMD) and which shading rates are supported. AMD's RDNA2 architecture supports 1x1, 1x2, 2x1 and 2x2 shading rates. Additional shading rates, which are unsupported by RDNA2, are 2x4, 4x2 and 4x4. Disallowing additional VRS rates (on cards that do support them) can be helpful to test consistency of an implementation independently of the feature.

VRS Techniques and Combiner

The user interface options for technique combiners allow the user to directly modify all options the VRS API provides in DirectX12.

PerDrawVRS resembles VRS tier 1 and allows the user to set the base shading rate. In the sample this rate will be applied to all draw calls, although it is recommended to modify it on a per draw basis when using VRS tier 1. Please check out the VRS tier 1 sample regarding this feature.



```
▼ Stats
▼ FidelityFX Variable Shading Settings
Resolution        : 1920x1080
1x1                                        PerDraw VRS
✓ ShadingRateImage Enabled
  ShadingRateImage Overlay
               0.040                       VRS variance Threshold
               0.050                       VRS Motion Factor
Override                                   ShadingRateImage Combiner
▶ Animation
▼ Profiler
time (s)              :   140.3
Clear shadow map (us) :     5.8
Shadow map (us)       :    34.4
Motion vectors (us)   :    55.6
Gen VRSImg (us)       :    28.0
PBR Forward (us)      :   269.7
Skydome proc (us)     :     0.6
PBR Transparent (us)  :     0.0
Downsample (us)       :    48.0
Bloom (us)            :   153.8
Tone mapping (us)     :    33.5
ImGUI Rendering (us)  :    34.3
Total GPU Time (us)   :   663.7
            GPU frame time (us)
```

The final section controls usage of the FidelityFX Variable Shading technique, which generates a VRS image surface based on the luminance information in the last frame, combined with motion vectors.

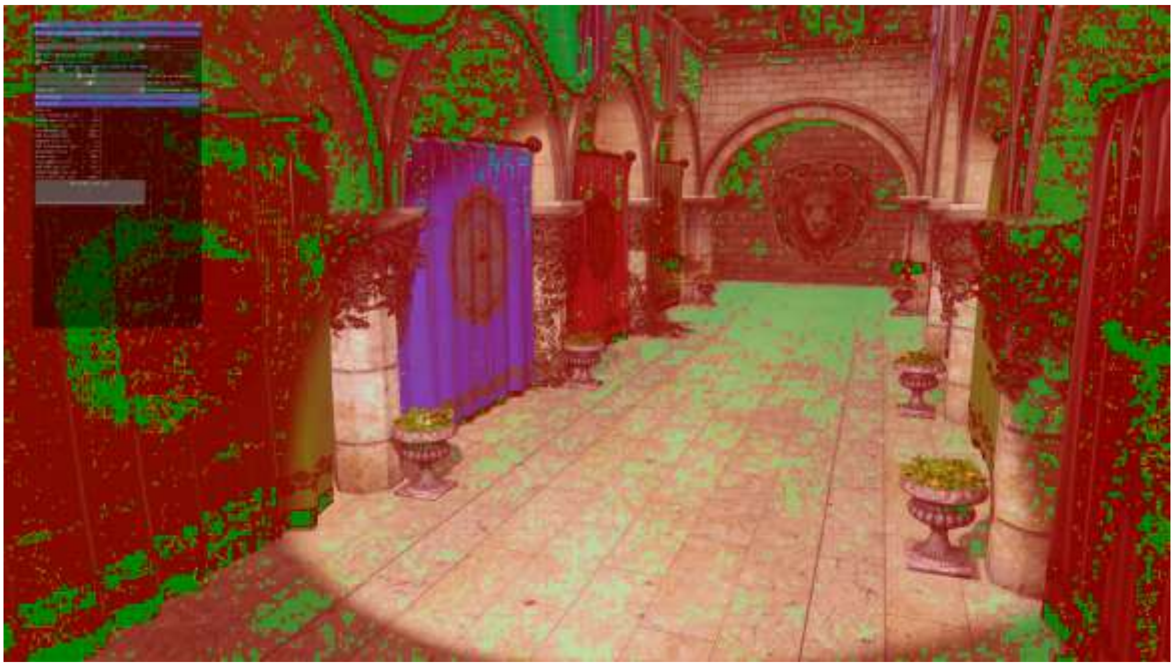
"VRS Variance Threshold" defines a value against which luminance variance gets compared in the compute shader generating the VRS image. Tuning this value allows you to apply VRS more aggressively at the cost of visual quality, or vice versa.

"VRS Motion Factor" sets a factor by which the motion of the pixel since the last frame gets scaled to modify the shading rate. This allows VRS to be used more aggressively if the camera or some objects are moving very fast.

The combiner controls how the VRS rate stored in the VRS image should be combined with the output of the previous combiner to compute the final VRS rate.

VRS Image Overlay

When generating a VRS image in the compute shader, it is a good idea to implement a way to visualize the VRS image over the rendered scene, in order to easily identify if the VRS image matches what you would expect (i.e. where in the image regions exist that allow lower resolution to be applied). Toggling VRS and the overlay also allows the user to spot regions where VRS is being applied too aggressively.



Implementation

Prerequisites

To start using VRS, Windows 10 build 1903 or later, plus a recent windows SDK, as well as a GPU supporting VRS are required. RDNA2-based GPUs from AMD fully support VRS tier 2 (without the optional additional shading rates).

Details

The first thing required when implementing VRS is to query the following information:

- Which tier is supported? On AMD RDNA2 hardware, VRS tier 2 is supported.
- What is the tile size for the VRS image? On AMD RDNA2 hardware the tile size is 8x8.
- Are additional shading rates supported? On AMD RDNA2 hardware additional shading rates are not supported.

Next, assuming image-based VRS is going to be used, the application needs to create an R8_UINT image of dimensions matching the render target resolution divided by the VRS tile size. This image should get cleared to `SHADING_RATE_1x1` by default.

The content of the VRS image is generated by a compute shader reading the previous frame's back-buffer. Ideally, in addition to the previous frame's back-buffer, a buffer containing motion vectors for each pixel should be present to determine the on-screen position of a pixel in the last frame. With this information, the compute shader can analyze the color variance within a VRS tile.

Experiments have shown that usually it's enough to only look at the luminance of pixels to determine if all pixels inside the current VRS tile are similar enough to reduce the shading rate. This way the complexity of the shader to compare the variance and the VGPR usage can be significantly reduced.

To confirm a pixel matches the previous frame's back buffer, the depth buffer of the current and the previous frame could be considered. This would increase the runtime cost of the generation shader due to the additional data needed to be read. Instead, the compute shader in the sample only relies on the motion vectors: if motion vectors are large, the pixel is likely to not require high detail rendering since motion won't allow the player's eye to focus on small details anyway.

Furthermore, motion blur is likely to blur any fine detail. If the motion vector is small, we can assume with high confidence that the pixel depth matches the last frame, without reading additional data.

Additionally, input parameters like glossiness could be included to ensure no reduced shading rate is used for highly reflective surfaces, since for those the color value will change significantly with movement of the camera, so an accurate prediction for a good VRS rate is not possible solely based on the previous frame.

Sampling outside the box

In an ideal case the input to the compute shader would contain an image that contains the whole scene as if VRS was completely disabled. However, the previous frame may already be affected by reduced VRS shading. As a result, in some cases where low shading rate was enabled in the previous frame, adjacent pixels contain the same color even though without VRS they would be slightly different.

This can cause a negative feedback loop (i.e. low VRS rates being applied in the last frame, e.g. due to fast movement of the camera, result in low variance within VRS coarse pixels). To counter this effect, FidelityFX Variable Shading also takes pixels outside each VRS coarse pixel into account and checks variance including neighboring regions, since the color of neighboring coarse pixels is not affected by this effect.

The FFX_VariableShading interface

The sample comes with a simple to use header file to ease the integration of VRS into any DirectX 12 title. The header file defines 6 functions:

- **`FFX_VariableShading_GetVrsImageResourceDesc`**
  This function takes width and height of the back buffer as well as the hardware-dependent tile size and returns a resource descriptor for the VRS image.

- **`FFX_VariableShading_GetDispatchInfo`**
  This function returns the threadgroup size of the dispatch of the VRS image generation shader. The shader to generate the VRS image needs to define a UAV of type uint named `imgDestination`, as well as 2 functions "`float ReadLuminance(int2 pos)`" and "`float2 ReadMotionVec2D(int2 pos)`" which are used by the shader to read luminance and motion vectors. Depending on which data is available in an engine, those functions may contain code to gather the information from various maps or decode the values fist.

`VariableShadingCode.cpp/.h` contains a helper class which handles all VRS related code in the sample. The main functions are

- **`ClearVrsMap/ComputeVrsMap`**
  Those functions handle the updating of the VRS map each frame.
- **`DrawOverlay`**
  This function renders a full-screen triangle to visualize the VRS image currently used.
- **`SetShadingRate`**
  This function updates the base shading rate and the shading rate combiners. It can also be used to temporarily disable VRS during a frame.
- **`StartVrsRendering/EndVrsRendering`**
  Those functions should be called at beginning and end of the part of the frame which is using VRS. Those functions handle the binding/unbinding of the VRS image and should only be called once per frame. Note that depth-only passes, fillrate-limited passes or post-processing passes will not benefit from VRS.

Important performance consideration

Due to the way VRS is implemented in AMD RDNA2 hardware, we advise minimizing the number of times per frame the VRS image gets bound or unbound. If VRS needs to be disabled for a few draw calls while the same depth buffer is being used, (e.g. to render alpha-tested geometry) the best practice is to leave the VRS image bound and disable VRS by modifying the combiners.