REAC 2025

Questions for Mishima Hitoshi

Question	Answer
Is there not a potential performance issue with using such small 8x4 regions for the visibility buffer resolve. If I'm not mistaken using triangle for these will cause unfilled quads along the triangle diagonals. Did you consider using Rect primitives for them instead on console to avoid this issue?	As you have mentioned, it's true that we have seen some performance issues along the diagonals. With regards to using rects instead, we simply haven't had a chance to properly measure the performance differences yet. If it's at all similar to the fullscreen triangles we use for post processing, then I would expect a few % improvements in performance. Unfortunately DirectX doesn't offer rect as an option for topology. We tried addressing this before by replacing our two triangles with a single larger triangle and using clip distance to trim said larger triangle, but saw no measurable differences in performance. Additionally, we've experimented with a compute shader version of this in our internal engine tests but it still couldn't beat the pixel shader's speed.
you credit quite a few English presentations in your slides, do you find access / translation to these to be a problem? I imagine in the reverse there are also a ton of quality talks in Japanese that we are missing out on because we aren't being exposed to them. (This is perhaps just a general question for the chat too) Are there any resources attempting to bridge that gap?	There isn’t any particular gap when it comes to technical matters. However, there is the issue of good papers in non-mainstream languages getting overlooked. I believe that advancements in information science will help resolve this problem. Things have definitely gotten much easier compared to before.
I had a hard time following the pre-transformed vertex section. Was this run as a prepass in a compute shader, or done when the mesh shader runs? Are all the verts streamed out to a large buffer? If so, did you/do you have any concerns about bandwidth and memory usage? (Also thanks for the shout out)	First, we use mesh shaders to render the Visibility Buffer. Second, using a compute shader, we perform a pre-transform of vertices for the GBuffer based on the Primitive IDs from the Visibility Buffer. Finally, in the GBuffer pass, we reference both the Visibility Buffer and the PreTransformed vertices. Here, the PreTransformed vertices are specifically for the GBuffer. The second step might seem unnecessary, but computing vertex transforms for all three vertices per pixel would be too computationally expensive. The issue lies in memory consumption. If compacted efficiently, it can be reduced to 1/32 or 1/64 (depending on wave count), but in the worst case—where each pixel contains a triangle—it becomes pixel resolution × 3 vertices ×

Question	Answer
	vertex size. Fortunately, our engine uses alias memory, so although it’s an extremely large memory block, it's only consumed for a specific part of the rendering frame.
Do y'all have a special approach for far sun shadows, i.e. beyond the final cascade?	Yes. SDF Shadow has been adopted.
Do you have fully skinned foliage or some are mesh hierarchies with a skeleton that you deform to circumvent the vertex/meshlet skinning pass? (I’m thinking trees, for example)	We're not using foliage with skeletal animation, but we do have features like movement based on a rotation origin called a "pivot", as well as some SpeedTree features. In terms of functionality, rendering Skinned Meshes is supported. However, the foliage system itself does not support skeletal animation.
did you have any issues with “small” clusters, for example what happens if you have a lot of clusters with less than 128 triangles each?	The main issue lies with foliage. The polygon density of grass and leaves is low, and the cluster bounding boxes are not sufficiently small relative to the size of the instance bounding boxes. For such meshlet clusters, running cluster culling tests may be pointless. In the main branch of the engine, we’ve implemented a mechanism that bypasses the cluster culling test based on the size difference between the instance and cluster bounding boxes.
In MH:World the season system changes are very abrupt, changing almost instantly. Was this an artist or technical choice?	Unfortunately this was a choice made by the game development team so I am unaware of the reasons for this decision.
I'm not sure I understand the base pass rasterization. It looks like it's done on a per-tile basis? So is it that the SW rasterization is performed fully in the pixel shader?	By "base rasterization", are you referring to the visibility buffer? The visibility buffer stores data such as the Zprepass (32-bit), cluster indices (25-bit), and triangle indices (7-bit). These are generated using either a hardware or software rasterizer. Unlike tiled approaches, this processing is done on a single full-screen buffer. The GBuffer is generated using a pixel shader.
vendors recommend to output 124 triangles instead of 128 in mesh shaders, did you notice any performance differences because of it?	We may be operating on older information, but I have heard that NVIDIA recommends outputting 126 triangles. We tested 126 triangles on NVIDIA hardware and, while it definitely seemed faster, the performance difference wasn't too significant. Of course this could have just been a problem with our implementation.

Question	Answer
Have you considered having different instance encodings depending on the instance type, e.g: an instance coming from a GPU placement system where you could afford to compress the world matrix VS an instance coming from an artist-placed entity that should retain a precise transform	At this point, we haven't implemented that level of optimization. However, when placing a truly massive number of instances, using float4x3 can be quite large, so switching to a more compact format might be a good idea. Personally, I sometimes wonder whether it's better to store the data using fixed-point floats. In older engines—like those from the PS3/Xbox 360 era—grass positions were sometimes stored using 8-bit formats.
RE4 has way better performance in mid-level hardware than both DD2 and MHWilds. What would you say is the limiting factor in the open world games? Do they need more refined occluders which is not possible for such vast spaces?	I apologize, but we will refrain from responding as it is difficult to comment on this matter.
Have you considered directly shading the visibility buffer (read triangle id, reconstruct triangle , interpolate and shade) instead of generating an intermediate gbuffer for deferred shading?	No, we haven't tried that yet. That said, our engine does have a pass that performs forward shading. It might be interesting to try it there and see if it improves performance. Currently, the forward shading portion using special rasterization is implemented by transforming vertices with a mesh shader and applying forward shading directly in the pixel shader.
You mentioned all large meshes use a single mesh shader in DD2, does this mean you have a unified meshlet data format?	Yes, that's correct. The meshlet data is standardized. More precisely, we use dynamic branching based on flags in the meshlet header to reconstruct the data. As for vertex tangents, there's a project setting that allows you to either completely remove them or keep them. If the project is configured to not use tangents at all, the meshlet structure becomes static in that regard, and the shader is compiled accordingly.
What's the process for game teams for requesting + receiving engine features look like if they're prohibited from changing code? Is it "more" expected that content teams work around limitations virtually all of the time	There are three types of requests: requests submitted in batches by the TAs in the development department, joint development based on long-term interdepartmental plans, and emergency responses for specific game titles. Most requests come from TAs, but emergency responses often take higher priority, which can cause delays. In such cases, the engine team and the title team usually coordinate to implement special features or optimizations.

Question	Answer
	On the title side, we’re gradually expanding the range of what they can use. For shaders, some users are now writing compute shaders and outputting to ByteAddressBuffers or Textures. These results can then be plugged into the Shader Editor as textures or buffers. Recently, we’ve also added functionality to define custom shading pipelines—for example, to achieve Toon Shading or other visual styles.