Intel® Graphics Performance Analyzers Assist Snow Simulation in *Frostpunk**

By Szymon Jabłoński, 11 bit studios, and Bartosz Boczula, Intel

Warsaw, Poland-based 11 bit studios put a lot of care and thought into the design and implementation of their snow simulation and snow rendering system. Their in-house Liquid engine has several custom terrain and shading features that efficiently support the look and feel of a world with dynamic weather. This article begins with a brief discussion of some implementation details of the snow simulation and rendering. Following that, we discuss the collaboration between 11 bit studios and Intel to improve the GPU performance of the snow rendering on Intel® Graphics while maintaining the visual fidelity required to make snow look and behave as needed for immersive gameplay and storytelling.

The game was optimized for Intel® Iris® Pro Graphics 580, using the Intel® NUC kit NUC6i7KYK—the platform commonly known as Skull Canyon (details here)—with 6th generation Intel® Core™ i7 four-core (eight threads) processors and the Microsoft® Windows® 10 OS. The primary tool

used for the performance optimizations outlined in this article was Intel® GPA, available for free here.

Thanks to great cooperation with the developers from 11 bit studios, we were able to improve the average frame rate from ~17 frames-per-second (~58 ms) to ~35 fps (~29 ms) in the scenario we've tested, making the game very enjoyable to play. In this article, we will fully describe the snow simulation and rendering system, and then show several GPU optimizations that were crucial to the game's overall playability.

Realistic Snow Simulation is Vital

Snow plays a very important role in the frozen world of *Frostpunk**. Players rule the last city of survivors left on Earth, where the intense cold is a constant foe. Heat sources may temporarily keep the relentless snow at bay, the city's inhabitants can clear it away as they navigate the world, but it will always continue to fall. In *Frostpunk*, both the snow compute simulation and the snow rendering are performed entirely on the GPU. The GPU snow compute simulation is divided into three stages: snow initialization, snow melting, and snow falling.

Snow Initialization

Simulation is dependent on two dynamic textures: a snow heightmap, and a snowfall mask. The snowfall mask is used to prevent snowfall in defined areas, such as under buildings, inside heat zones, and on streets (See Figure 1). At the beginning of the game, the snow heightmap is initialized with input data from game assets, and the snowfall mask is cleared. To achieve a good visual result, the snow heightmap uses an *R16F* format and the snowfall mask uses an *R8G8* format. *R16F* is 16-bit floating point, and offers higher fidelity representation of snow height. *R8G8* is simple snow on/off, and doesn't need float representation (even *R8G8* might be overkill for representing snowfall on/off areas).

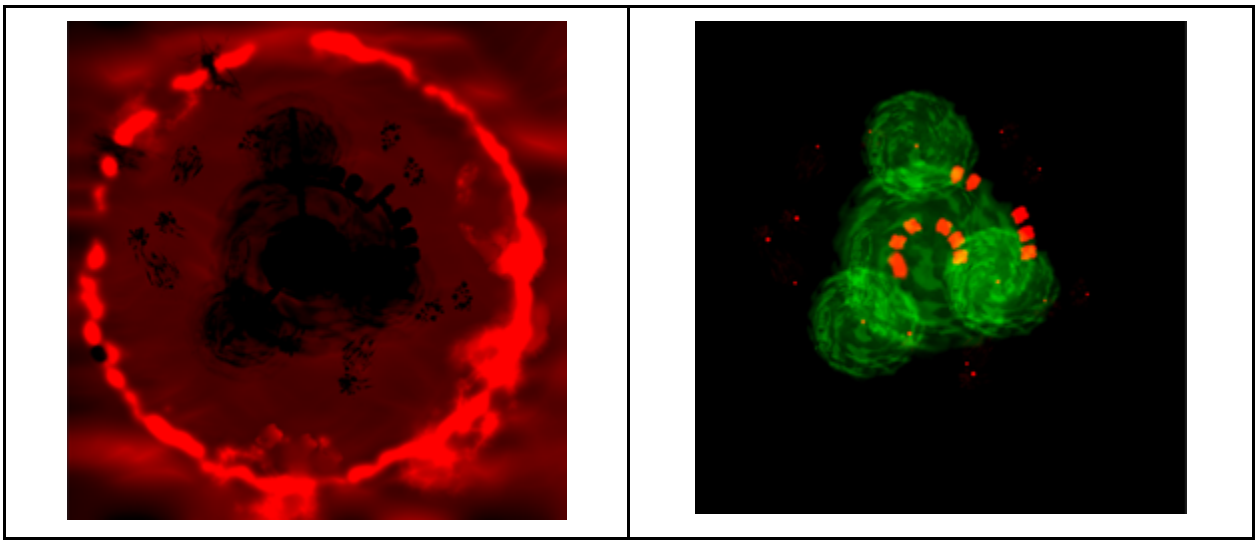


Figure 1. Textures used in snow simulation. On the left a snow heightmap texture, on the right snowfall mask (to indicate areas where snowfall should be prevented). Both textures are 1K; the first one uses R16 format and the second one uses R8G8 format.

Snow Melting

Throughout the game, there are several different types of events that can make the snow melt. So called “snowmelt” events are implemented as relatively cheap, batched draw-calls of “melt quads”. Melt quads simulate snowmelt by using additive blending with positive or negative intensity, and rely on additional textures and procedural noise functions to create varied melting effects. Melt quads are also used to update the snowfall mask texture.

- *Citizen/Automaton Movement*. Each citizen, and each leg of an Automaton, has a melt quad attached to it, which uses an additional texture that defines parameters such as size, fall-off, noise, and suppression, with random variations. See Figures 2 and 3.
- *Building/Street Placement*. Each building and street segment also has a melt quad attached. Snow around buildings is melted immediately after construction is started. See Figure 4. This is kept in the red channel of snowfall mask.
- *Steam Generator Operation*. Heat sources also have melt quads attached, but unlike buildings and streets, the snow around steam generators melts more slowly and dynamically. See Figure 5. This is kept in the green channel of snowfall mask.

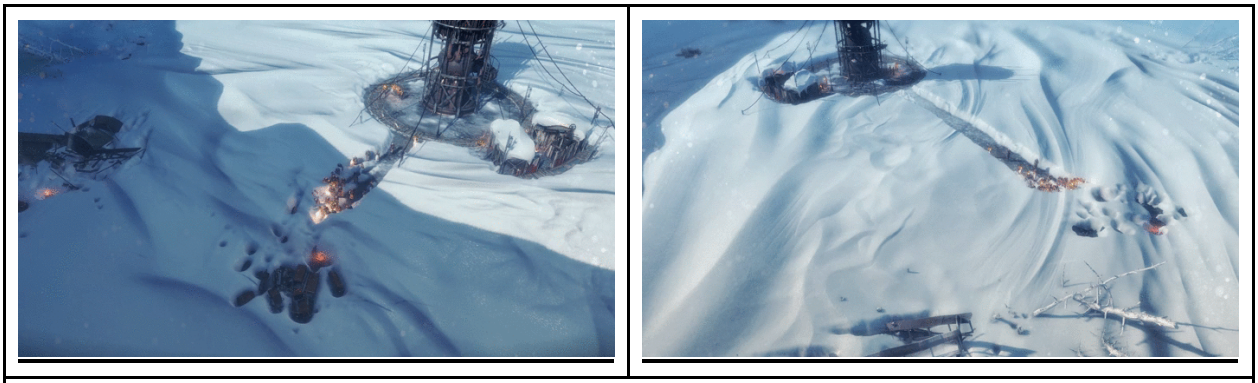


Figure 2. Examples of snow melting through citizen movement.

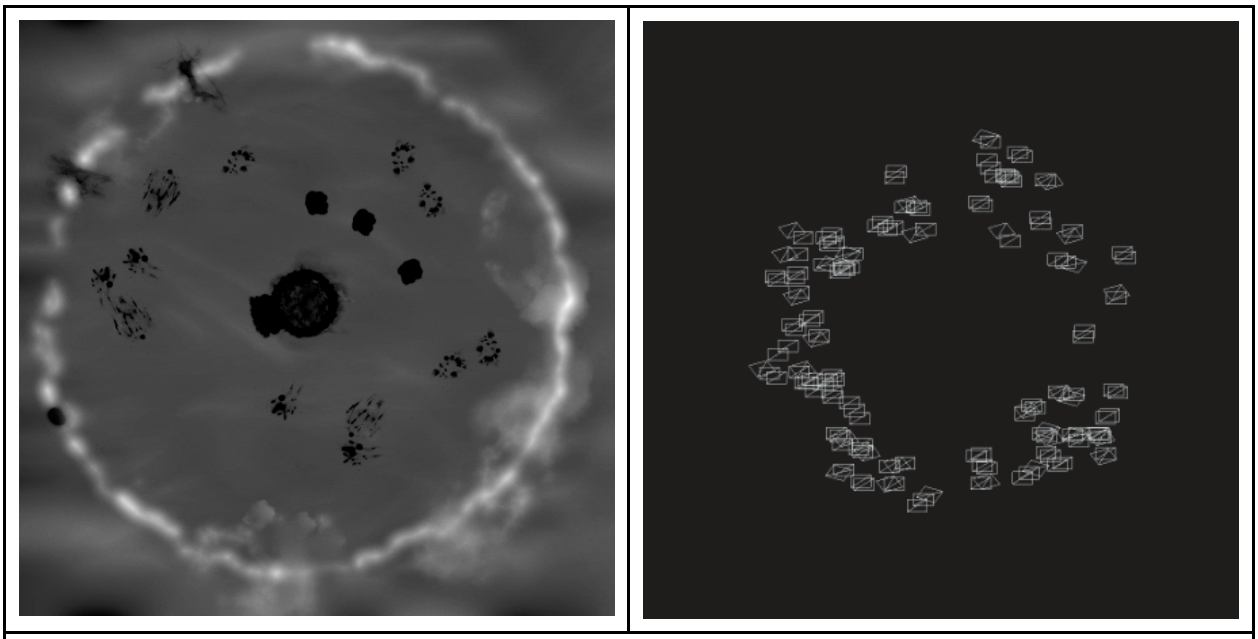


Figure 3. Citizen movement simulation. On the left, snow heightmap changing to reflect melted snow. On the right, an animation of Input Assembler view of batched citizen melt quad draw-calls.

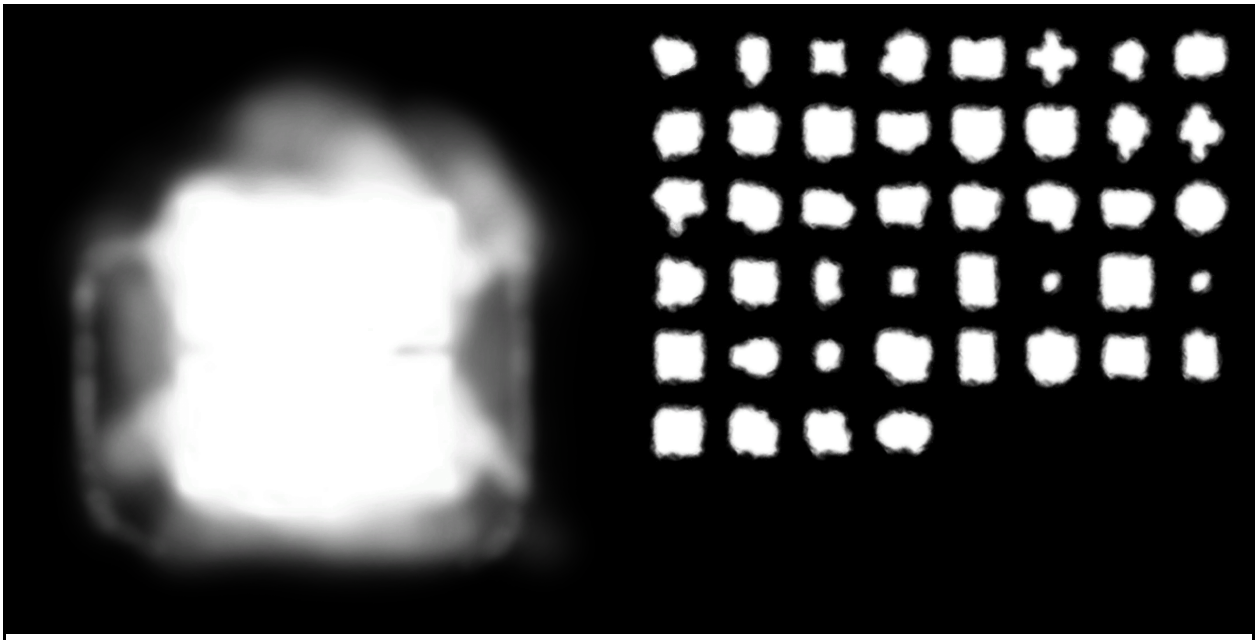


Figure 4. Example of textures used in building melt-quads. On the left, a texture used for Ambulatorium building. On the right, texture atlas used for common buildings.

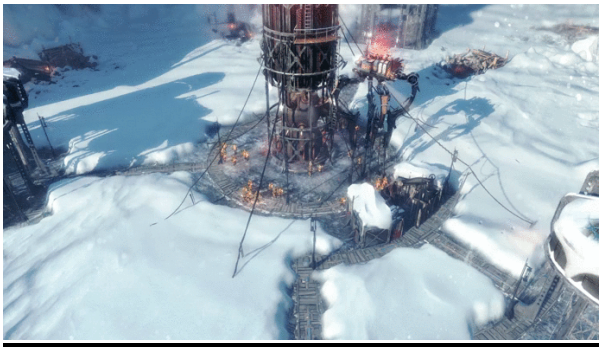


Figure 5. Example of snow melting though working steam generator and Automaton movement.

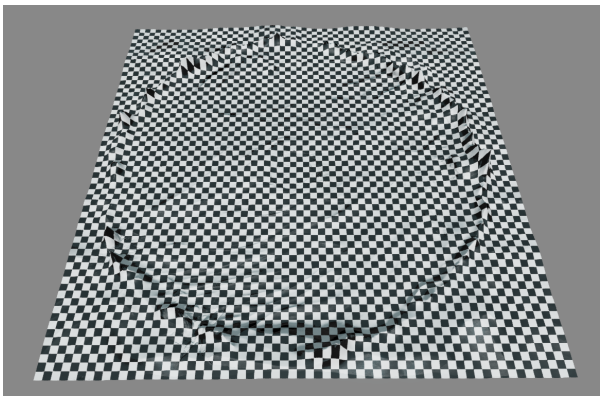
Snow Falling

Snow constantly falls, covering previously melted areas of the world. This operation is relatively simple, constantly increasing the values in the snow heightmap until they reach their original initialized value. The dynamic height-map is initialized with data from game assets (original snow heightmap sculpted by artists). This is the start point, and also the end-goal, of the snow falling. The snowfall pixel shader first applies new snow, and then clamps the value in the snow heightmap to ensure it stays within the proper range (more than 0.0, and less than the maximum read from assets). For increased visual quality, additional textures and parameters are used to achieve non-homogeneous snow growth. The snowfall mask is used to prevent snow growth in areas where snow cannot fall, such as on streets, around buildings, and inside active heat zones (created by steam generators).

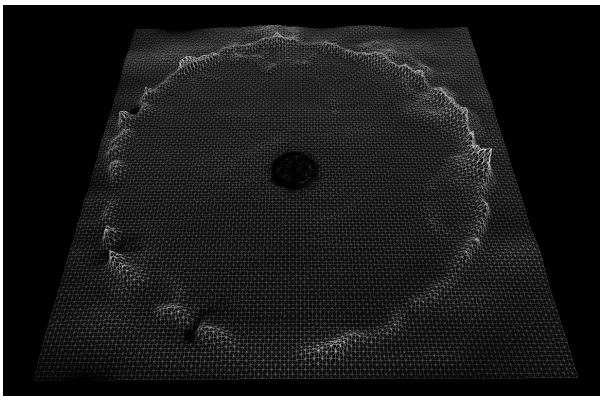
Meshes and Decals Create Snow Rendering System

The G-Buffer pass in 11 bit studios' Liquid engine consists of three stages: rendering solid meshes, solid softness meshes, and decals. Solid softness meshes are meshes that support smooth blending with solid meshes from the previous stage. The snow terrain is rendered with the softness meshes to achieve smooth blending between snow and environment objects.

The dynamic terrain for the snow system is fully controlled by the GPU. The supporting geometry consists of *N* x *N* terrain tiles. Terrain tiles are rendered with the help of geometry instancing. See Figure 6.



N x N tiles division (black & white checkerboard pattern)



Coarse geometry wireframe without tessellation

Figure 6. Terrain-tiles structure for snow system.

The whole terrain is divided into *D* x *D* areas to support CPU frustum culling. See Figure 7.

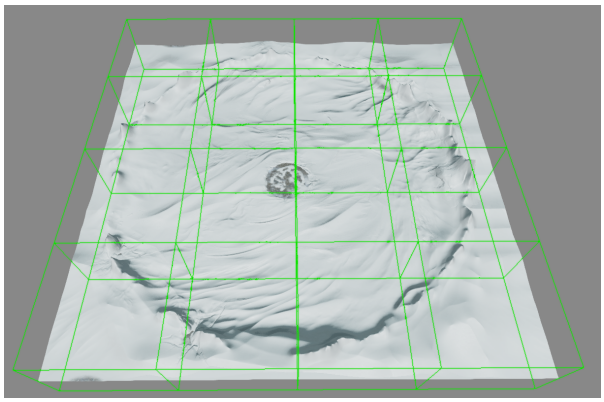
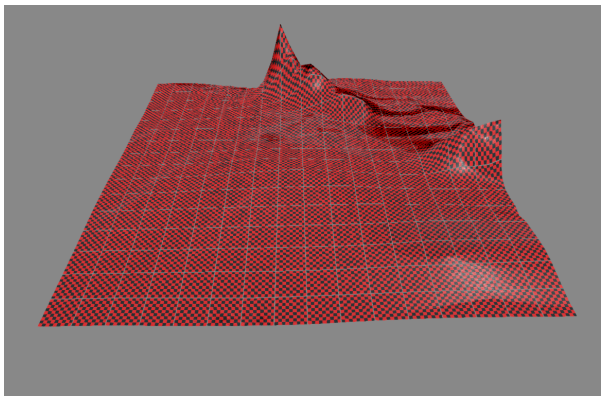
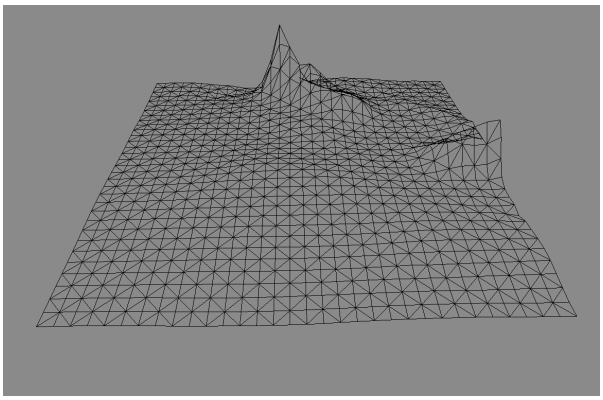


Figure 7. Additional terrain subdivision for CPU frustum culling. Green lines are visualization of subdivision areas bounding boxes used for frustum culling.

Each terrain tile consists of M x M tessellation patches tessellated in tessellation shaders. Both the number of terrain tiles and density of tessellation patches are configurable in the terrain properties. See Figure 8.



M x M patches division (black & red checkerboard pattern) for each tile for selected snow terrain area



Coarse geometry wireframe preview for selected snow terrain area

Figure 8. Terrain tessellation patches structure for selected terrain area.

The terrain system depends on the following collection of textures:

- *Snow heightmap*. Modified dynamically by the snow simulation, the heightmap is used as the terrain displacement map for geometry displacement and normal calculations.
- *Material layer map*. Used to apply texture splatting to snow ground.
- *Material blend map.* Used to store weight of each painted material.
- *Materials texture arrays*. Used to store materials for terrain painting.
- *Density map*. Used to store information about terrain areas that could be tessellated in shaders.
- *Softness map*. Used to store information about terrain areas that could be blended with solid objects.
- *Detail map*. Used to store additional normal/roughness/metalness maps in high resolution. This texture is especially important for naturally flat snow objects.

The vertex shader is used to apply basic transformation to the flat terrain patches. Hull and domain shaders perform adaptive tessellation based on density maps and the distance LOD evaluation. They also apply the terrain displacement, and perform GPU culling of terrain patches. The pixel shader is responsible for a number of things, including generating geometry normals, applying and blending materials. Calculating normal vectors in the pixel shader (compared to performing calculation in domain shader) significantly increases the frame-time cost, but it results in amazing visual effects, and detailed terrain with low-resolution input geometry. See Figure 9 for a sample output.

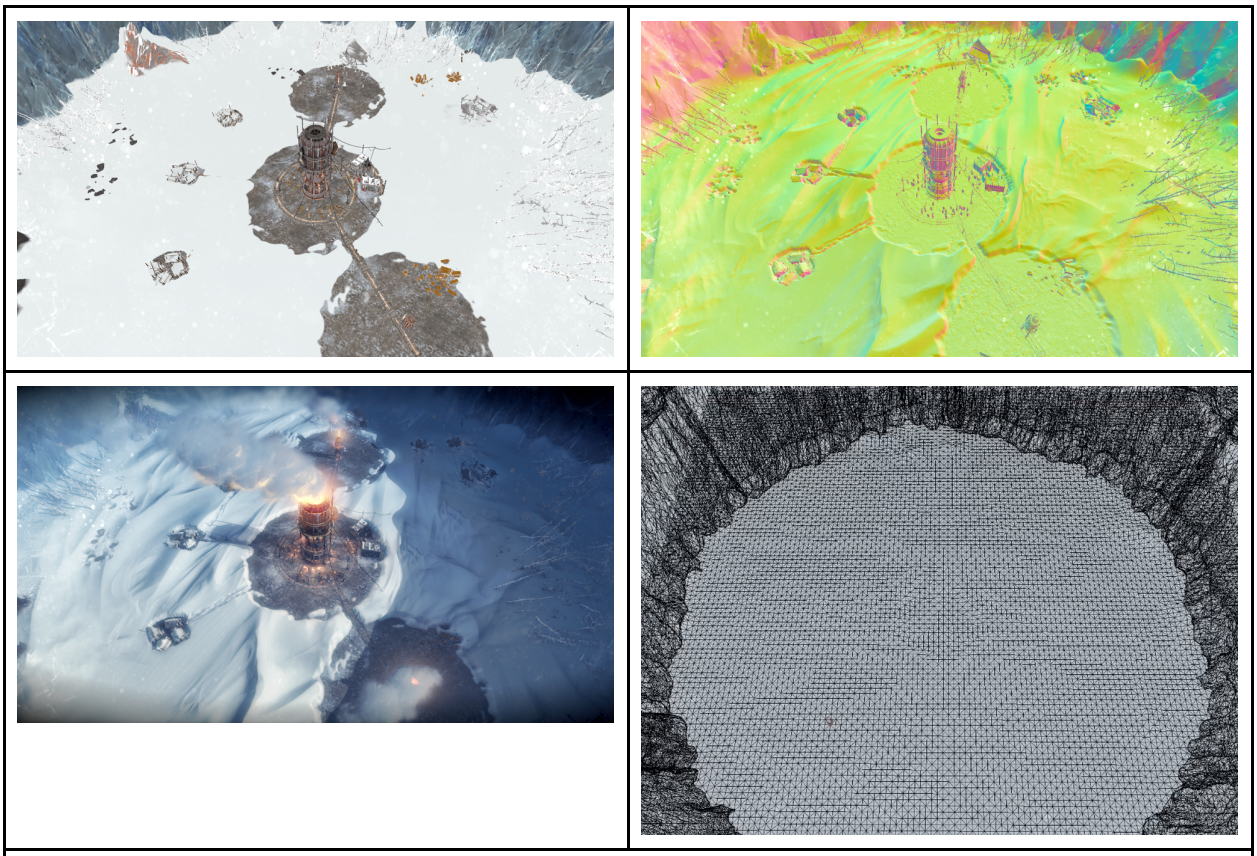


Figure 9. Example output of snow rendering
Top Left: Albedo, Top Right: Normal map
Bottom Left: Final render result
Bottom Right: Wireframe preview with distance based adaptive tessellation

Performance Analysis with Intel® GPA

Much of our performance analysis was performed using Intel GPA (link at the top of the article). GPA is a suite of graphics application performance analysis tools, including tools for capturing and playing back a single frame of your game. Intel® GPA can be used on both integrated and discrete GPU hardware. When used on Intel integrated graphics, GPA provides a collection of detailed hardware metrics that supports an extremely useful bottleneck analysis. Throughout this optimization discussion, we will reference several Intel GPA features. (Additional detail on these features can be found in the Intel GPA User Guides.)

Looking at a single frame from *Frostpunk*, the first thing we noticed was that the very first draw-call, which was responsible for snow simulation, took almost 3ms of GPU time on Skull Canyon.

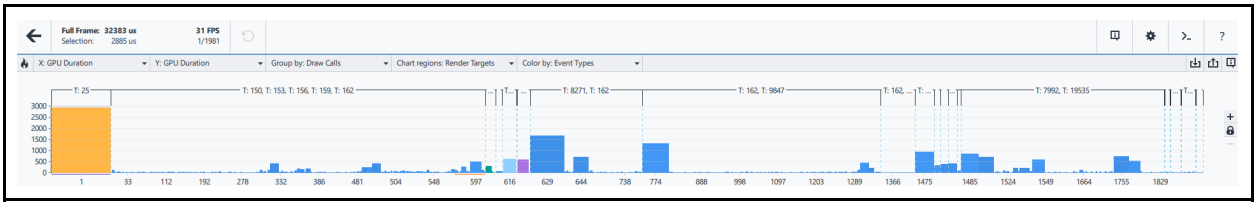


Figure 10. Snippet from Intel GPA. It represents all of the draw calls in the frame, and the height of each bar represents time taken to execute given draw call. The orange bar is a draw call responsible for snow melting and falling in *Frostpunk*.

Knowing what this snow simulation actually does, and knowing that our goal was to achieve at least 30 fps on this hardware, this draw call seemed too expensive.

Here is the DirectX* bytecode of the pixel shader used in this draw call:

```
36 ps_5_0
37 dcl_globalFlags refactoringAllowed
38 dcl_sampler s0, mode_default
39 dcl_sampler s1, mode_default
40 dcl_sampler s2, mode_default
41 dcl_sampler s3, mode_default
42 dcl_resource_texture2d (float,float,float,float) t0
43 dcl_resource_texture2d (float,float,float,float) t1
44 dcl_resource_texture2d (float,float,float,float) t2
45 dcl_resource_texture2d (float,float,float,float) t3
46 dcl_input_ps linear v1.xy
47 dcl_input_ps linear v2.yzw
48 dcl_input_ps linear v3.xyzw
49 dcl_output o0.xyzw
50 dcl_temps 1
51 add r0.xy, v3.wwww, v3.xyxx
52 mul r0.xy, r0.xyxx, v3.zzzz
53 mad r0.xy, v1.xyxx, v2.yyyy, r0.xyxx
54 sample_indexable(texture2d)(float,float,float,float) r0.x, r0.xyxx, t3.wxyz, s3
55 sample_indexable(texture2d)(float,float,float,float) r0.yz, v1.xyxx, t2.zxyw, s2
56 max r0.y, r0.z, r0.y
57 mad_sat r0.x, r0.x, v2.z, -r0.y
58 sample_indexable(texture2d)(float,float,float,float) r0.y, v1.xyxx, t0.yxzw, s0
59 add r0.x, r0.x, r0.y
60 max r0.x, r0.x, l(0.000000)
61 sample_indexable(texture2d)(float,float,float,float) r0.y, v1.xyxx, t1.yxzw, s1
62 min o0.x, r0.y, r0.x
63 mov o0.yzw, v2.yyzw
64 ret
65 // Approximately 14 instruction slots used
66
```

We can see that this shader is relatively simple: four sample instructions, and some simple math in-between. How can such a simple shader take 3ms to execute? Well, Intel GPA has a few handy features to help us dig deeper (See Figure 11).

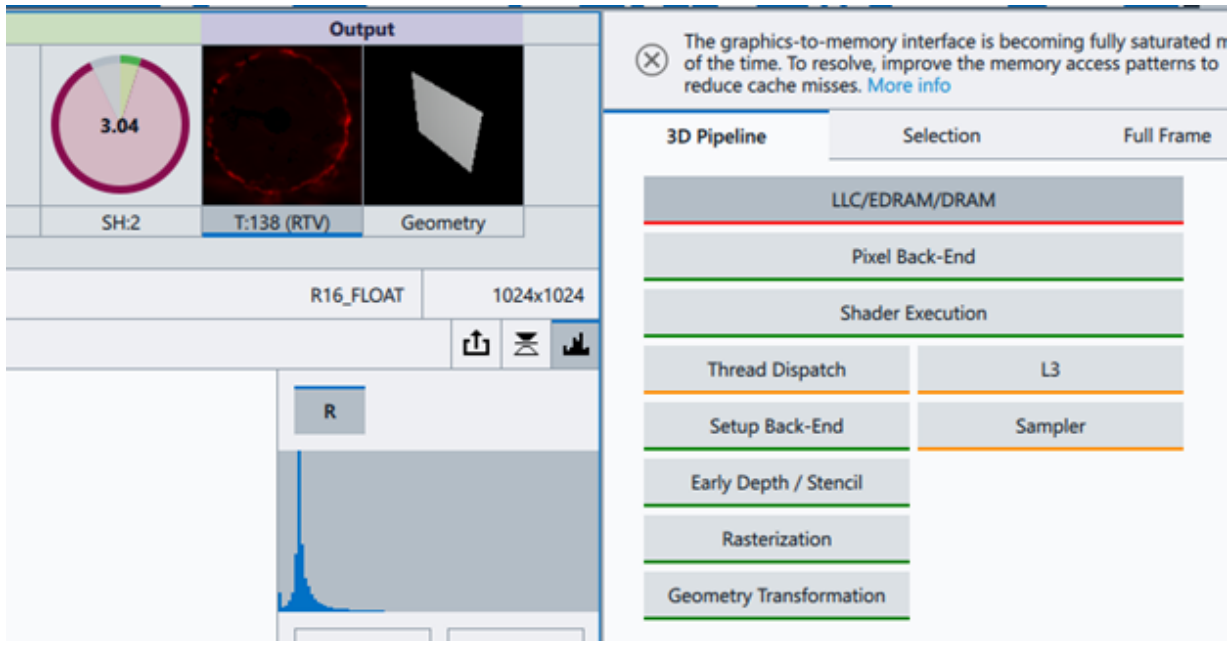


Figure 11. Snippet of Intel GPA view. On the left, characteristic of shader execution as a pie chart; on the right, Intel GPA bottleneck view.

In Intel® Architecture parlance, we refer to the parallel GPU cores as Execution Units (EUs). The pie chart in the upper left of Figure 11 shows you how the EUs spent their time during the execution of this draw call:

- Green represents Active time (EUs actually running shader instructions).
- Red represents Stalled time (time where EUs are waiting for something).
- Gray represents Idle time (no shader running on the EUs at all).

In this snow simulation draw, the EUs spent 85% of that 3ms stalled.

The diagram on the right of Figure 11 is a pipeline flow chart that uses hardware metrics and heuristics to indicate where you will likely find a bottleneck. The “Selection” and “Full Frame” tabs in this view let you view the raw metrics valued, sorted based on pipeline stage. Note that the “Shader Execution” box is green, but the “LLC/EDRAM/DRAM” box is red, indicating that the bottleneck is likely related to memory reads/writes, coming from the four sample instructions in the pixel shader. To understand why, and what optimization is necessary, we need to understand more about our GPU and system architecture.

Optimization and Intel's Memory Architecture

Intel® processor graphics does not have dedicated video memory—instead, it implements the concept of unified memory, where system DRAM memory is shared across the entire system, including the GPU and CPU.

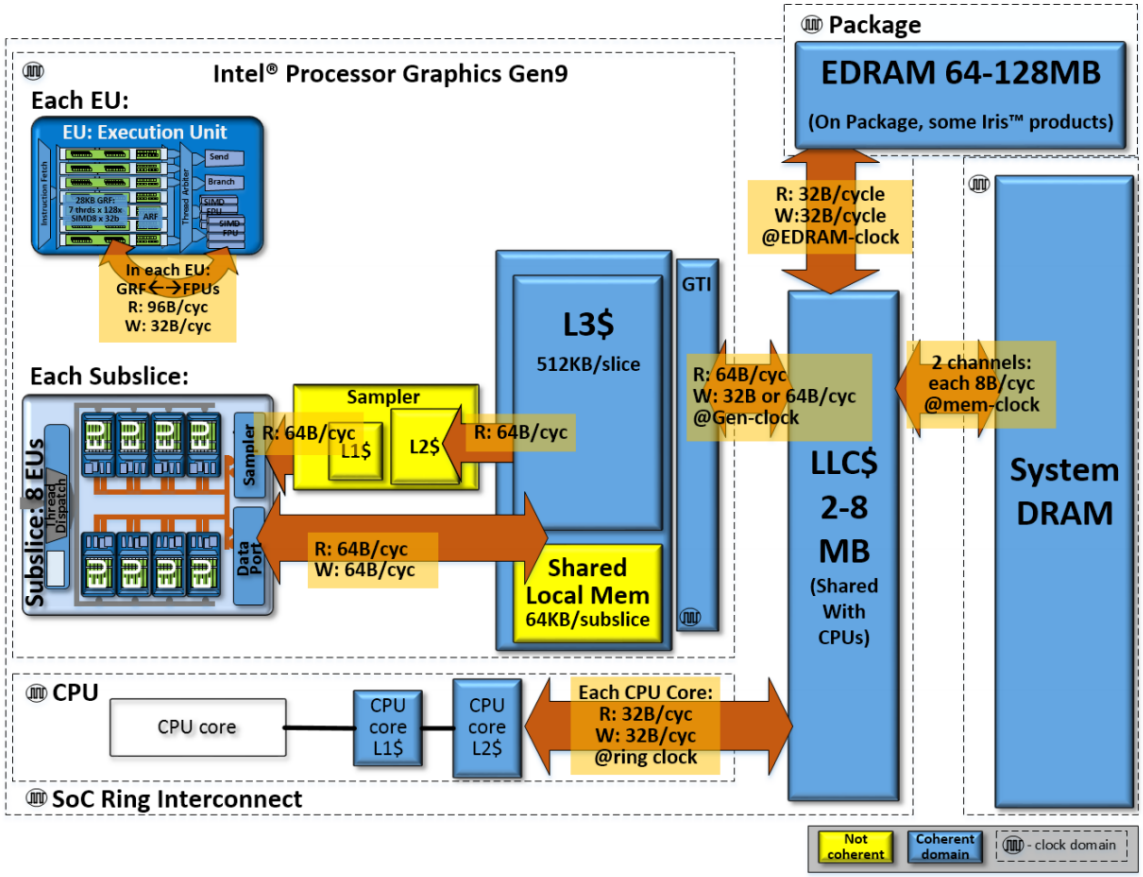


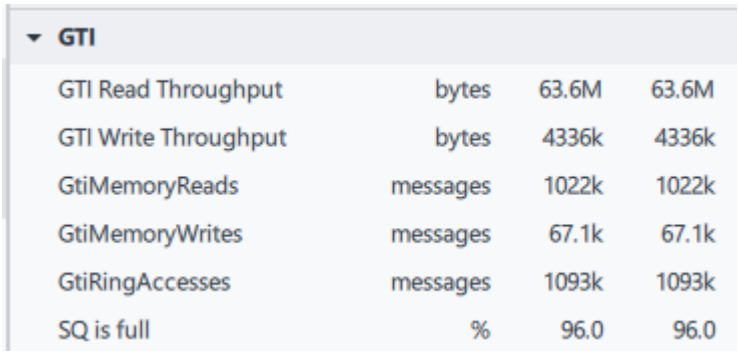
Figure 12. A view of the SoC chip level memory hierarchy, and its theoretical peak bandwidths for the compute architecture of Intel processor graphics Gen9.

The Intel processor graphics features a cache hierarchy, where each texture sampler unit has its own local cache hierarchy, and there is a L3 cache, shared by all GPU EU cores and samplers. If the GPU needs data that is not resident in the L3 cache, it must issue a request to access shared system-memory to retrieve that data. There is a Last Level Cache memory (LLC) that is shared with the CPU, and then eventually system DRAM. (Some products also feature 64-128 MB of EDRAM between the LLC and DRAM).

So what happens when an EU samples a texture resource? Due to the parallel nature of the GPU and typical spatial locality, the sampler doesn't just fetch just one texel worth of data. It fetches an entire cache line worth of data, which is 64 bytes on Intel GPUs. So when sampling the R16F snow heightmap, a texture fetch will also fetch data for 32 neighboring texels. With

many pixel shaders running in parallel, each requesting data from the same texture, we should hope that the data we're requesting is already in cache more often than not.

Let's go back to Intel® GPA. Figure 13 shows some of the detailed memory metrics for this draw call. Please note that the data is presented in two columns. In Intel GPA you can do various experiments, and the data in the column on the left represents measurements after applying the experiments and column on the right represents original values. In our case, we haven't done any experiments, so the data in both columns is the same.

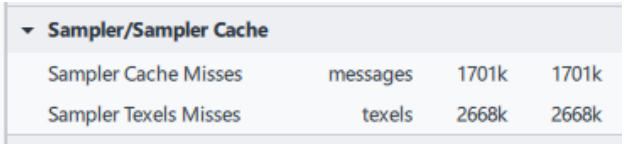


GTI			
GTI Read Throughput	bytes	63.6M	63.6M
GTI Write Throughput	bytes	4336k	4336k
GtiMemoryReads	messages	1022k	1022k
GtiMemoryWrites	messages	67.1k	67.1k
GtiRingAccesses	messages	1093k	1093k
SQ is full	%	96.0	96.0

Figure 13. Detailed memory metrics for the snow simulation draw call in Intel GPA.

GTI is the name of the interface that Intel Graphics uses to go out to system memory. The GTI Read Throughput is a bit scary—more than 63 megabyte (MB) of data is being read from outside of the GPU just for this draw call—streaming either from DRAM or LLC to GTI. It is very likely that the Execution Units spend so much time stalled because the data they are requesting is not locally available in either the local sampler caches or the GPU's L3 cache. They are stalled waiting for data to be fetched through the GTI from outside the GPU.

The sampler cache metrics in Figure 14 confirm this theory, showing ~1.7M cache misses:

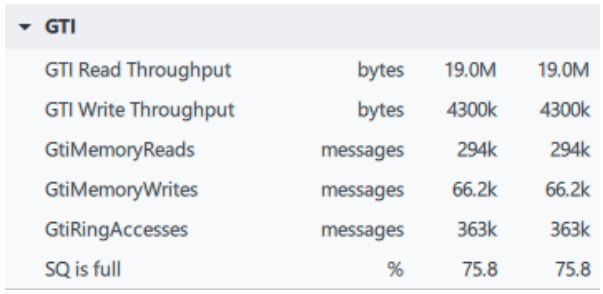


Sampler/Sampler Cache			
Sampler Cache Misses	messages	1701k	1701k
Sampler Texels Misses	texels	2668k	2668k

Figure 14. Sampler Cache metrics in Intel® GPA

If we have a good cache hierarchy, and we're fetching 32 neighboring texels at a time, how can the number of cache misses be so high? One possibility is that if the texture we're sampling is very large, then the relative UVs could point to texels that are actually quite far from each other, and thus each cache line would only contain a small amount of useful data. In such a case, we would be needlessly filling up our sampler caches and our L3 cache with adjacent, but unused data.

One of the snow simulation textures being sampled was a 4K texture, while each of the others was only 1K. Per our analysis-driven suspicion, this large texture is too high frequency relative to the lower frequency of the shader sampling. We provided this feedback and insight to 11 bit studios, and about how expensive this draw was on this hardware, and they quickly swapped in a 1K replacement for that 4K texture. Figures 15 and 16 show the GTI and sampler cache metrics for this draw after this change was applied. This change didn't affect final image too much.



GTI			
GTI Read Throughput	bytes	19.0M	19.0M
GTI Write Throughput	bytes	4300k	4300k
GtiMemoryReads	messages	294k	294k
GtiMemoryWrites	messages	66.2k	66.2k
GtiRingAccesses	messages	363k	363k
SQ is full	%	75.8	75.8

Figure 15. Detailed memory metrics for the snow simulation draw call in Intel GPA, after the 1K texture substitution from 11 bit studios.

Sampler/Sampler Cache			
Sampler Cache Misses	messages	827k	827k
Sampler Texels Misses	texels	2921k	2921k

Figure 16. *S*ampler Cache metrics in Intel GPA after the 1K texture substitution from 11 bit studios.

GTI read throughput decreased from 63M to 18M. There are now 3x fewer memory reads through GTI and about 2x fewer cache misses, resulting in a 2.4ms savings in total frame-time.

After improving the cache behavior, the bottleneck for snow simulation changed to "Thread Dispatch", and partially into the Sampler unit (See Figure 17). Intel GPA is hinting that the shader logic is potentially performing inefficiently, and to improve this, one can reduce shader thread payload (e.g. register usage). If you want to find out more about different bottlenecks, have a look at the Intel Graphics API Performance Guide for Gen9 (link in the references).

After giving this feedback to 11 bit studios, they came up with a few more optimizations.

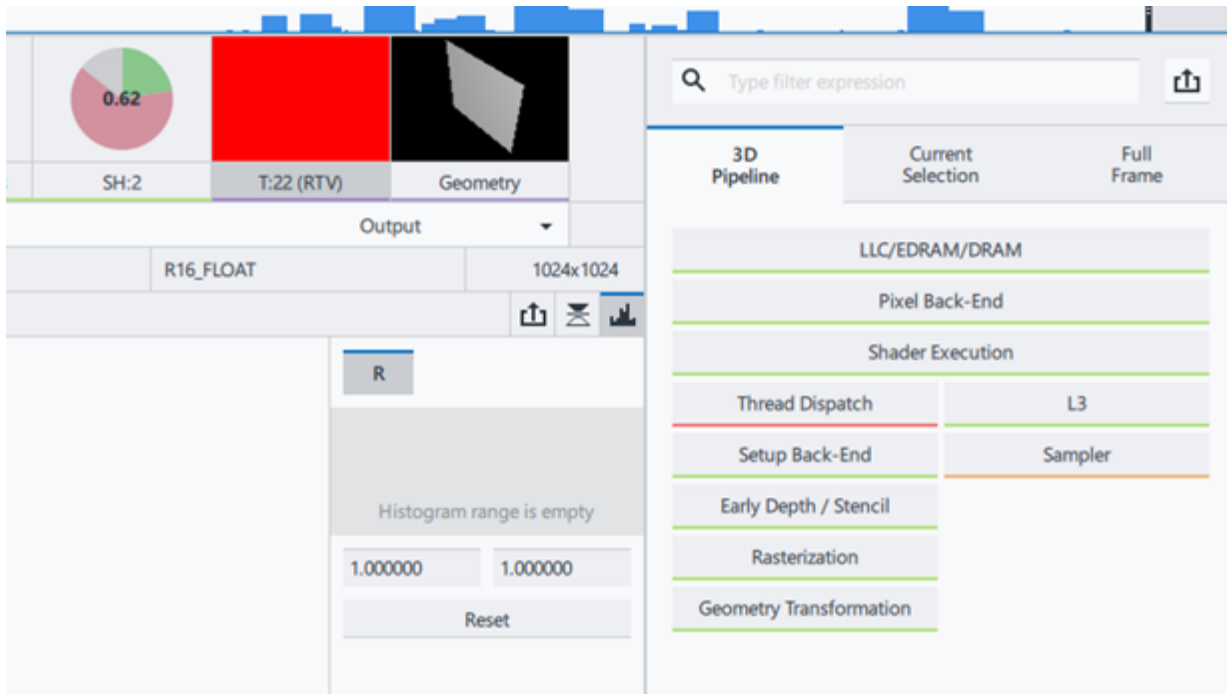


Figure 17. Pipeline bottlenecks in Thread Dispatch and Sampler

Additional Optimizations

Snow simulation was changed to no longer operate directly on a dynamic snow heightmap. Instead, the output of the snow simulation is a single dynamic *R8* format snow displacement mask texture. This texture is bound to the snow terrain shader along with a static heightmap that contains the initialized displacement data, and these are multiplied together to give the final snow height value.

This reduced the total number of samples needed to perform the snow simulation from four to two, and also removed the need to perform the clamping operation during the snowfall calculation. This significantly relieved sampler pressure in this draw.

A nice side-effect of changing the snow simulation texture formats was that the size of both game-saves and time-lapse snapshots decreased.

The snowfall algorithm was further redesigned to operate on tiles instead of the entire heightmap. The whole area is now divided into *N x N* tiles, and snowfall is only applied to one tile per frame in a round-robin fashion. In other words, instead of updating the snow for the entire map 30 times a second, now only one tile was updated every frame. Thus we decoupled the snow compute simulation, from the rendering. This solution was possible because snow is falling rather slowly, so it didn't really matter if the entire map was updated 30 times a second, or once a second. However, because of this change, the overall performance of this draw call increased drastically. The snowfall shader was also simplified by disabling the application of procedural noise for the

lowest quality profile, which removed another sampling instruction. This combined to make the per-frame cost of the snowfall draw ~17 us, which is practically negligible. If your game is doing this kind of GPU simulation for an entire map, maybe it would be worth it to decrease update frequency in a similar manner.

Finally, 11 bit studios simplified the snow rendering pixel shader to avoid the expensive normal vector calculation, assuming that the snow area was flat with a normal vector of (0, 1, 0). Using the terrain detail map in normal mapping still produced a satisfying visual result, while offering a nice performance savings on these settings.

Conclusion

Here is a screenshot of the finished game running on Skull Canyon with Intel® Iris® Pro Graphics 580.

Figure 18. Screenshot from the game running on Skull Canyon.

The optimizations to the snow system are just a few examples of the performance improvements that 11 bit studios made to *Frostpunk* over the course of our engagement with them. As shown in the screenshot above, this was achieved while preserving a high standard for visual quality—especially great-looking snow.

If you're not already using Intel GPA to optimize your code and enhance your performance benchmarking, you owe it to yourself to get started and come up to speed on all the various tools it offers. As you can see from this article, *Frostpunk* benefited greatly from a solid understanding

of the Intel GPA toolkit. We encourage you to check it out and see for yourself how great snow can look. After all, winter is coming…

Acknowledgements

First of all, big thanks to 11 bit studios for their great cooperation, and Szymon for co-authoring the article. Thanks to all of the reviewers who helped us publish this article: Adam Lake, Stephen Junkins, Jefferson Montgomery, Geoffrey Douglas and Dietmar Souch.

Resources

[Compute Architecture]

[Graphics API Performance Guide for Intel Processor Graphics Gen9]

https://store.steampowered.com/app/323190/Frostpunk/

http://www.frostpunkgame.com/

https://twitter.com/frostpunkgame