Using Vulkan graphics API to render a cloud of animated particles in Stardust Application

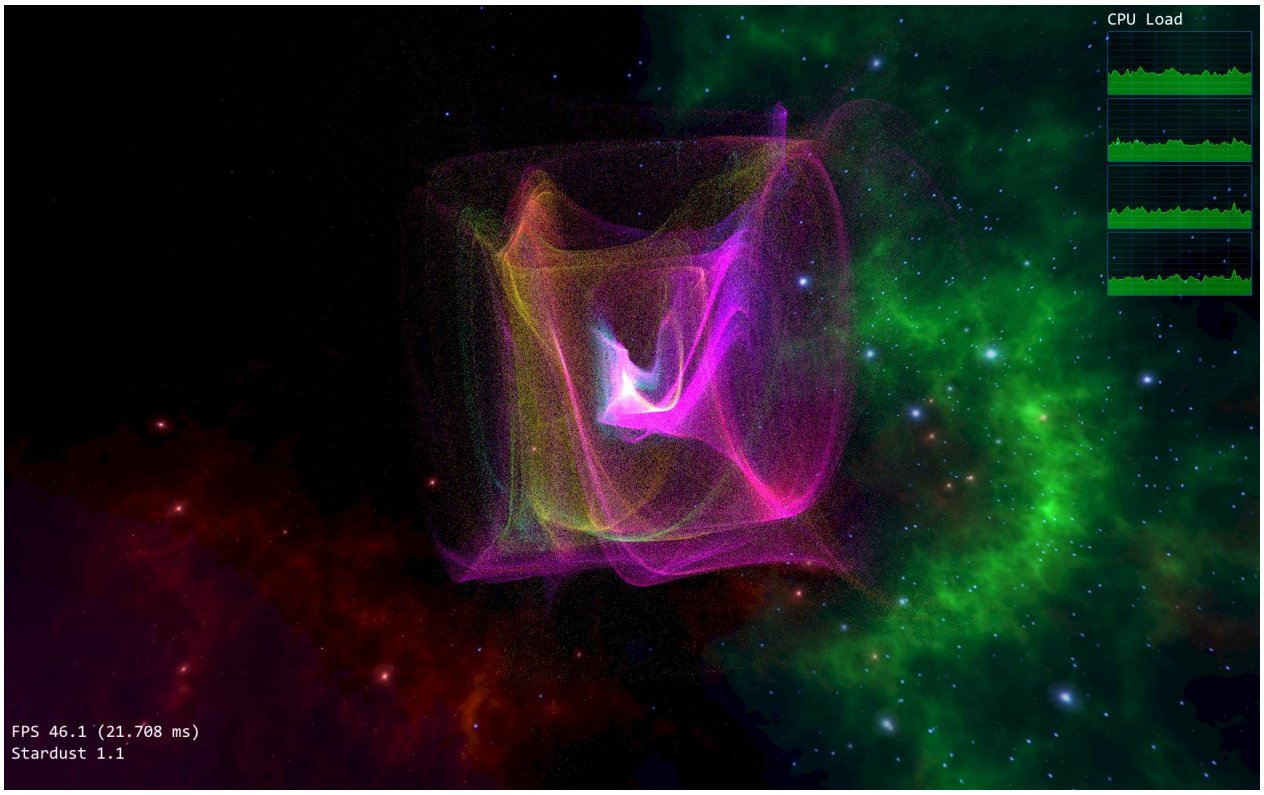
Vulkan™ API overview

Vulkan™ is the new generation, open standard API for high-efficiency access to graphics and compute on modern GPUs. This API is designed from the ground up to provide applications direct control over GPU acceleration for maximized performance and predictability. Vulkan is a unified specification that minimizes driver overhead and enables multithreaded GPU command preparation for optimal graphics and compute performance on diverse mobile, desktop, console, and embedded platforms.

The Vulkan API uses so-called command buffer objects to record and send work to the GPU. Recorded commands include commands to draw, commands to change GPU state, and so on. To record and execute command buffers, an application calls the appropriate Vulkan API functions.

Stardust demo overview

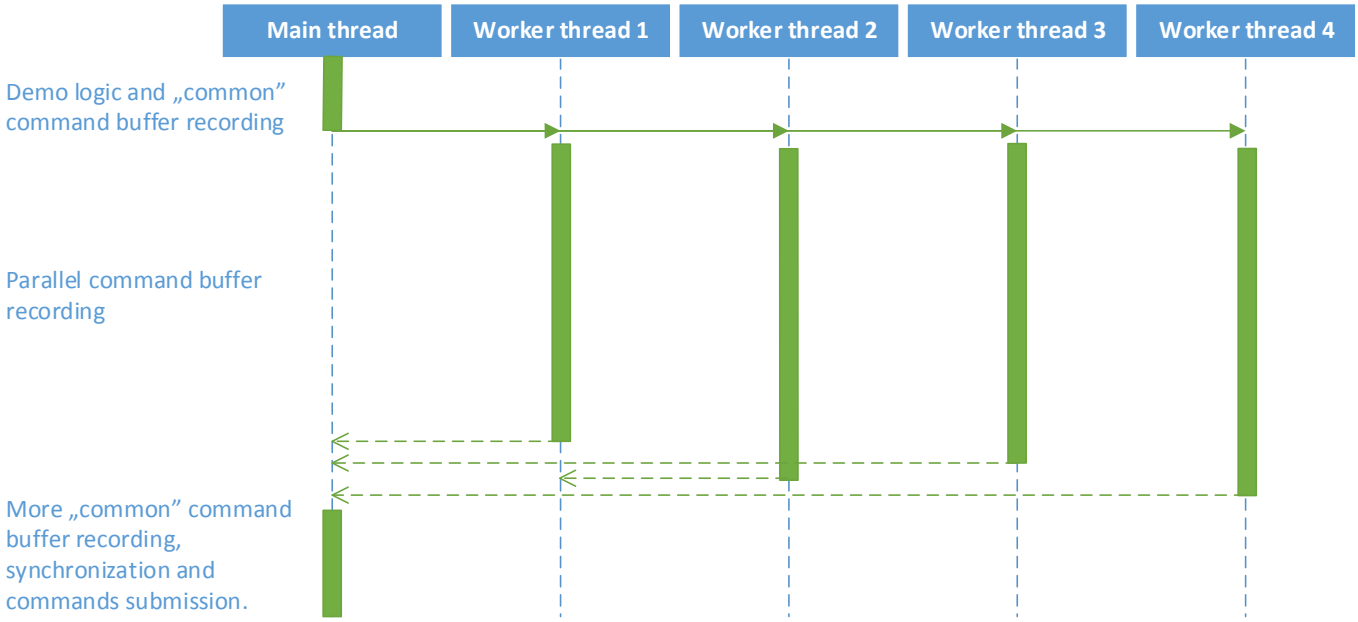
The Stardust sample application uses the Vulkan graphics API to efficiently render a cloud of animated particles. To highlight Vulkan's low CPU overhead and multithreading capabilities, particles are rendered using 200,000 draw calls. The demo is not using instancing; each draw call uses different region of a vertex buffer so that each draw could be a unique piece of geometry. Furthermore, all CPU cores are used for draw call preparation to ensure that graphics commands are generated and delivered to the GPU as fast as possible. The per-core CPU load graph is displayed in the upper-right corner.



GPU work generation and submission

Stardust sends 2,000,000 point primitives with 200,000 10-point draw calls to the GPU each frame. Please note that this is an exaggerated, extreme scenario that is intended only to demonstrate how thin and fast the Vulkan driver is. Despite so many draw calls, the CPU load is quite low as shown on the green graphs in the image above.

Intel's Stardust demo records and submits several command buffers each frame. All those command buffers together include 200,000 draw calls and other GPU commands. Command buffers are recorded in parallel by N CPU worker threads, where N is equal to the number of logical CPU cores present on a machine. Each worker thread records one command buffer that includes a subset of all draw calls and other commands. After all worker threads complete, the main thread submits all of the command buffers with one API call. In that way CPU load is distributed evenly across all cores, and the GPU is fed with commands as fast as possible.



GPU work generation and submission on a four-core CPU.

Particle animation

The position of each particle is computed entirely on the GPU in the vertex shader stage. The following steps are performed to compute particle positions:

1. A per-particle input "seed" value is used to compute the initial particle position.
2. A "matrix seed" value is used to compute two sets of one rotation, one translation, and one scaling transformation matrix. All three matrices in each set are then concatenated and linearly time-interpolated to form one final transformation matrix.
3. The initial particle position is multiplied by the computed matrix and then additional non-linear transformations are performed. This step is repeated a number of times. A 1D texture coordinate is also computed in this step.
4. The particle position computed in the previous step is multiplied by the concatenated view and projection matrices.

Animation of the particle cloud is achieved by updating the "matrix seed" value periodically on the CPU.

Particle coloring

To achieve nice color effects, the particles are rasterized with additive blending enabled. Particle color is computed in the fragment shader by sampling and interpolating two "palette" textures. Interpolation between two color samples is based on the time to get a smooth color change.