Asynchronous Shader Input from the CPU

Status: Public. Work in progress. Seeking feedback.

Summary

This document describes an asynchronous, lock-free, and atomic method for updating inputs from the CPU to GPU, but in a way that some inputs may be dropped rather than queued.

To implement this efficiently, we need an *atomicExchange* that can be executed on the same shared memory from BOTH the CPU and GPU.

Such a feature can be used, for example, to reduce user input latency (e.g. touch or mouse position) by providing the most up-to-date user input when a frame is drawn, which isn't necessarily known when a frame's commands are queued.

Sync Points Shader Arguments can also be used in conjunction with this feature for better latency control.

Prerequisites

The CPU->GPU communication can be built on top of an atomicExchange that can be called from both the CPU and GPU, assuming other existing OpenGL features are available.

Note: atomicExchange already exists in OpenGL, but only within a shader. It does not allow the CPU to execute an atomicExchange on the same memory directly.

Some devices can support atomic operations from both the CPU and GPU. See specs for PCIE 3.0 and this whitepaper titled "Atomic Read Modify Write Primitives for I/O Devices" from Intel. Various ARM/Mips SOC support TBD.

If a device does not support an atomicExchange from the CPU, it may alternatively schedule a synchronous task on the GPU to perform the atomicExchange. Hopefully the hardware has auxiliary execution units for simple low-latency tasks such as this and doesn't need to monopolize an entire execution block.

Details

The asynchronous shader input can then be implemented using a form of triple buffering, but implemented using four buffers to avoid unnecessary copying and race conditions.

1. CPU Writable
2. CPU Staged
3. GPU Staged
4. GPU Readable

Whenever the CPU has filled the “CPU Writable” buffer with input, it can exchange that buffer with the “CPU Staged” buffer.

Then, when the GPU wants to determine if there is new input, it can exchange the “CPU Staged” and “GPU Staged” buffers. Assuming data within the buffers include a sequence number, the GPU can determine whether the new “GPU Staged” buffer is *actually* newer than the existing “GPU Readable” buffer. If it is newer, the GPU can then exchange the two GPU buffers.

Overflow is handled by dropping data: If the CPU exchanges twice before the GPU exchanges at all, then the data in the first CPU exchange is simply lost from the GPU’s perspective.

The buffers can be stored in mapped Shared Shader Buffer Objects, accessible from both the CPU and GPU. Since atomic operations cannot be performed on entire buffers, the exchange should be executed on references/indexes to the buffers instead. Care must be taken to insert the proper memory barriers before or after each exchange.

Usage

In Chrome, we plan to use a combination of input prediction and glass time to reduce latency. If the buffers representing user input contain a curve representing the nearby past and future, then we can sample a point on that curve using the glass time. By updating that curve asynchronously, we can improve the prediction quality without introducing latency jank.

Reading Sync Points in a Shader to select Glass Time

Given an input curve over time, we could also be more intelligent regarding what glass time to use. For example, if a particular sync point hasn’t passed, that could indicate we are in a high latency mode and we should advance our glass time by a vsync interval.