DEPRECATED: Please see Discardable GPU Resources V2

~~Discardable GPU Textures~~

ericrk@

Overview

Problem

In Chrome, browser and renderer processes make heavy use of GPU texture memory. Each browser or renderer has its own texture caches, each cache with its own memory limits. As there is no central controller, cache limits must be chosen conservatively, assuming that many caches may be live simultaneously. This leads to inefficient use of texture memory. Not only is the upper limit on texture memory mostly unconstrained, but a renderer or browser performing heavy GPU work is given the same cache limits as one performing light work.

Proposed Solution

This document proposes the concept of “Discardable GPU Textures”. These are textures that:

- Can be unlocked by a client (renderer/browser), allowing the GPU process to delete them at-will.
- Are stored in a single LRU cache in the GPU process.

These features allow the GPU process to maintain a single cache of textures from multiple clients, deleting textures as necessary to enforce a global GPU memory limit. Additionally, the use of an LRU cache allows for clients which make heavy use of GPU resources to automatically take a larger share of the available memory, improving performance.

Client API Design

Overview

Every texture in use by Chrome will now be in one of three states.

- **Non-discardable** - a “traditional” GL texture which is not associated with the GPU discardable system. Non-discardable textures do not count against GPU discardable memory limits.

- **Locked-discardable** - a GL texture which is associated with the GPU discardable system and which counts against GPU discardable memory limits. A locked-discardable texture is considered in-use, and will not be automatically deleted by the GPU process.
- **Unlocked-discardable** - a GL texture which is associated with the GPU discardable system and which counts against GPU discardable memory limits. An unlocked-discardable texture is considered not-in-use, and may be automatically deleted by the GPU process.

A GL texture can transition between these states as follows:

The GPU process can delete unlocked-discardable textures as necessary to keep Chrome below a global memory limit. Clients must handle cases where re-locking an unlocked-discardable texture fails.

GL API

void glUnlockTexturesCHROMIUM(GLuint* texture_ids, uint32_t texture_count);

Transitions the textures associated with the provided texture_ids from the locked-discardable to the unlocked-discardable state. May only be called on textures which are in the locked-discardable state.

void glLockTexturesCHROMIUM(GLuint* texture_ids, uint32_t texture_count, bool* results);

Transitions the textures associated with the provided texture_ids to the locked-discardable state. May be called on textures which are in the non-discardable or unlocked-discardable states. Locking is guaranteed to succeed for textures that are in the non-discardable state. For other textures, the success of locking is passed out in that texture's index in the results array.

Implementation

Goals

- We must avoid race conditions around re-locking and deleting textures.
- Lock/Unlock calls are very common and block further execution. These calls should be fast, even in the cases where Lock fails.

- Deletion is a less common scenario, and must be correct, but not necessarily fast.

Overview

Locking and unlocking a texture is driven by client calls to glLockTextureCHROMIUM or glUnlockTextureCHROMIUM. On the other hand, deleting textures is driven by GPU Discardable Texture memory limits within the GPU process.

Despite originating in different processes, the handling of these three operations must be synchronized in order to prevent race conditions (for example, if the GPU process deletes a texture that a renderer has just locked). To achieve this, we move final control of all three operations to the client (renderer/browser) process:

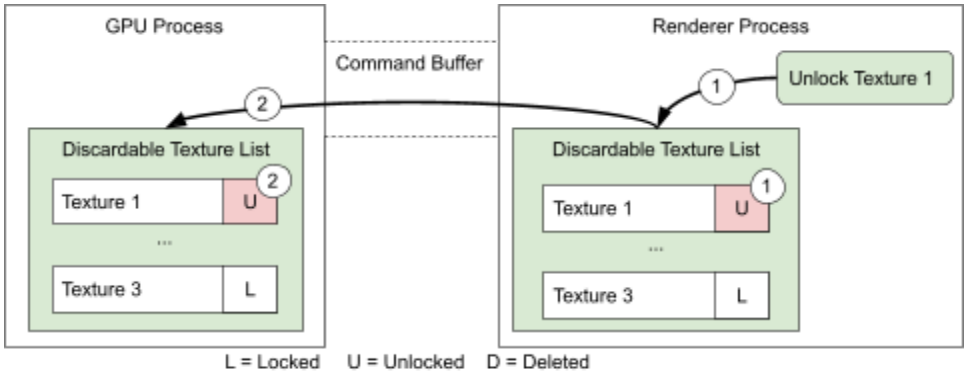
- Lock/Unlock commands are immediately handled in the client process, allowing for very low latency.
- Deletion commands are sent via IPC from the GPU process to the client process, preventing the client's view from being out of sync.

This optimizes for fast lock/unlock, while ensuring correctness.

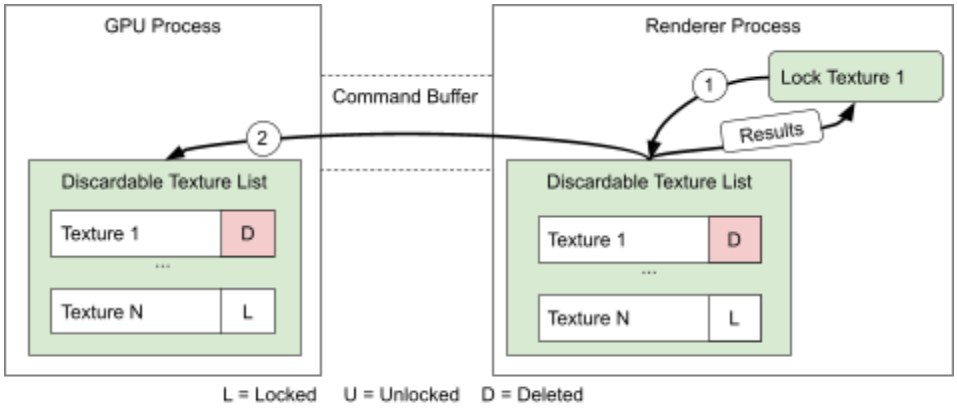
Details

Both the Renderer and the GPU process maintain a list of discardable textures as well as their current states. The GPU process additionally maintains LRU information so that textures can be deleted in LRU order.

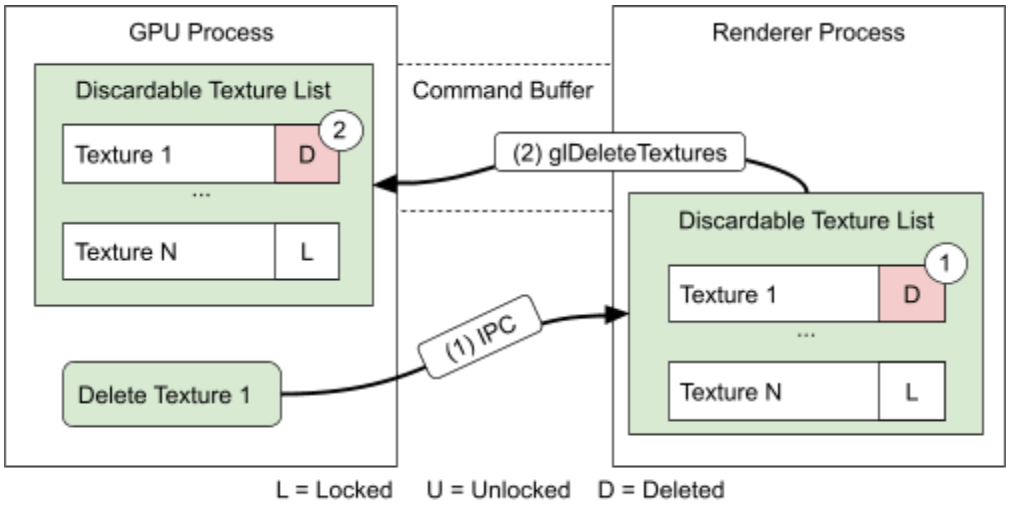
Unlock commands flow over the GPU command buffer from the renderer to the GPU process, updating state along the way:



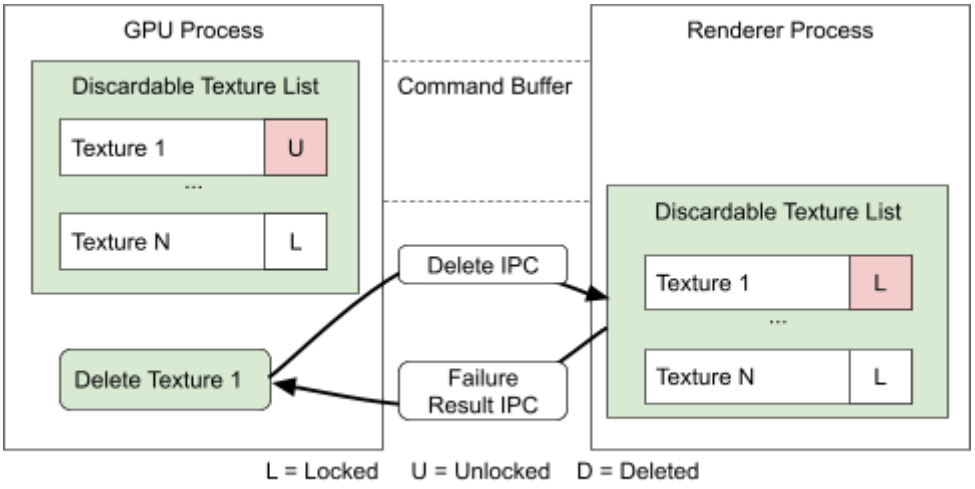
Lock commands are similar to Unlock, however they must pass results back to the caller. In order to minimize latency, results are handled within the renderer process:



Deletion is triggered by the GPU process, which sends a deletion request IPC to the renderer. The renderer then deletes the texture if possible. This avoids synchronization problems with lock/unlock.



If the GPU process is out of sync with the renderer process, deletion may fail. In these cases a failure IPC is sent back to the GPU.



In cases where a deletion request fails, the GPU process will move the texture to the end of the LRU list and attempt to delete additional textures. This should roughly maintain correct LRU order without needing to wait for the command buffer to synchronize.