Discardable GPU Memory V2

ericrk@

Overview

This is a re-design of the initial GPU discardable proposal, available here.

Problem

In Chrome, browser and renderer processes make heavy use of GPU memory. Each browser or renderer has its own 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 memory. Not only is the upper limit on 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 Memory”. This is memory that:

- Can be unlocked by a client (renderer/browser), allowing the GPU process to delete it at-will.
- Is stored in a single cache in the GPU process, allowing for a global eviction policy.

These features allow the GPU process to maintain a single cache of GPU memory from multiple clients, deleting objects as necessary to enforce a global GPU memory limit.

Client API Design

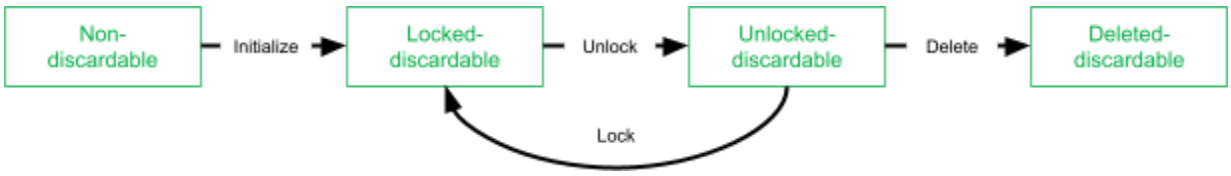
Overview

Note: *The initial implementation of Discardable GPU memory will deal with GL textures only. However, the implementation is designed to be extendable to other GL objects (buffers, render targets), as well as GPU memory buffers.*

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. The only operation which may be performed on an unlocked-discardable texture is glDeleteTextures.

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 glInitializeDiscardableTextureCHROMIUM(GLuint texture_id);

Transitions the provided texture from the non-discardable to the locked-discardable state.

void glUnlockTextureCHROMIUM(GLuint texture_id);

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

bool glLockTextureCHROMIUM(GLuint texture_id);

Transitions the provided textures to the locked-discardable state. Returns true if the texture was successfully locked, otherwise returns false, indicating that the texture was deleted.

Implementation

Goals

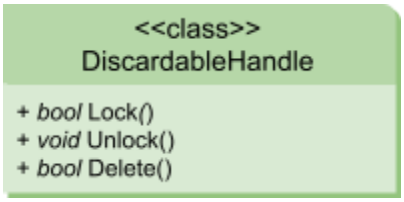
- Lock calls are very common and block further execution. These calls should be fast, even in the cases where Lock fails.
- Unlock calls must be synchronized with the GL command buffer, ensuring that unlock is delayed until all commands which may reference the locked memory are completed.

Key Components

In order to implement the GPU discardable APIs outlined above, we will implement:

- **DiscardableHandle** - A wrapper around shared-memory representing the locked/unlocked state of a discardable allocation.
- **ClientDiscardableManager**- A client (non-GPU) process component responsible for creating DiscardableHandles from a pool of shared-memory.
- **ServiceDiscardableManager** - A GPU process component which tracks DiscardableHandles and sizes of current allocations, deleting unlocked allocations as necessary.
- **GL Command Buffer functions** - Implementations of Initialize/Lock/Unlock outlined above.

DiscardableHandle

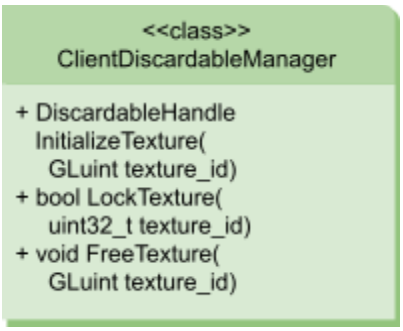


Backed by a single int in shared memory. This int represents the possible states of a discardable allocation. Due to lock being handled on the client process and unlock on the GPU process, there may be cases where a handle is locked multiple times before being unlocked. In these cases, the lock count is represented by values 2+.

Shared Memory Value	State
0	Deleted
1	Unlocked
2+	Locked

State is transitioned using atomic operations on shared memory.

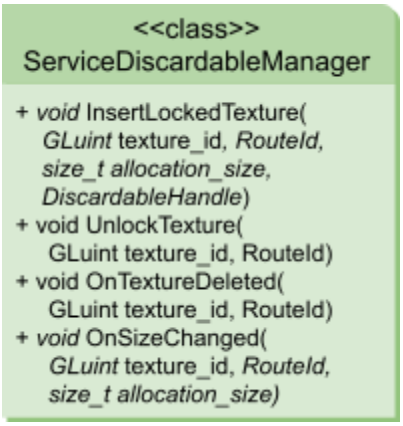
ClientDiscardableManager



Allocates a block of shared memory from which DiscardableHandles are allocated. Maintains a free-list to allow for efficient re-use of this memory as handles are created/destroyed.

ServiceDiscardableManager

Tracks each discardable allocation, storing its size and DiscardableHandle. Also tracks overall size of discardable memory. When allocated memory goes over-budget, the ServiceDiscardableManager attempts to delete elements in LRU order.



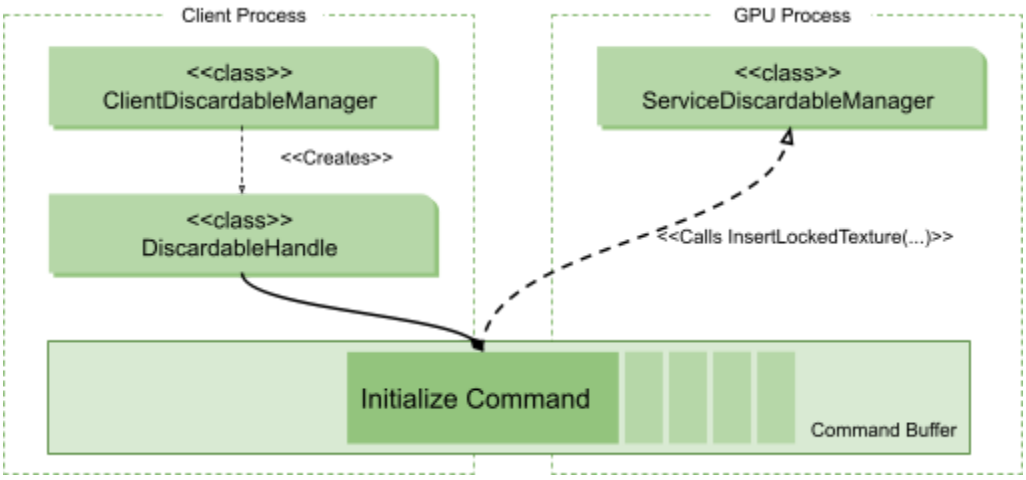
glInitializeDiscardableTextureCHROMIUM

In the client process:

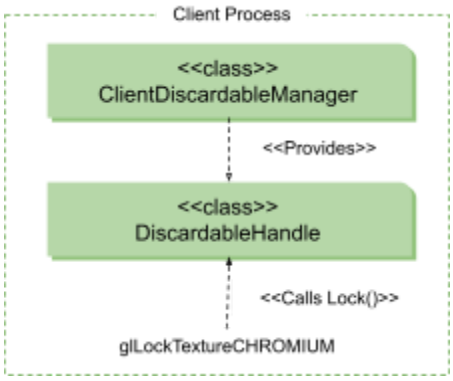
1. Requesting a new DiscardableHandle from the client process' ClientDiscardableManager.
2. Sends the handle and the texture ID over the command buffer to the GPU process.

In the GPU process:

3. Receives the DiscardableHandle and texture ID.
4. Queries the TextureManager for the texture's current size.
5. Registers the texture ID, DiscardableHandle, and current size with the ServiceDiscardableManager.



glLockDiscardableTextureCHROMIUM



As lock returns a success status synchronously, it is important that it execute quickly, not requiring a round-trip to the GPU process.

In the client process:

1. Requests the handle for the given texture from the ClientDiscardableManager.
2. Immediately attempts to lock the handle, returning the success or failure status.

glUnlockDiscardableTextureCHROMIUM

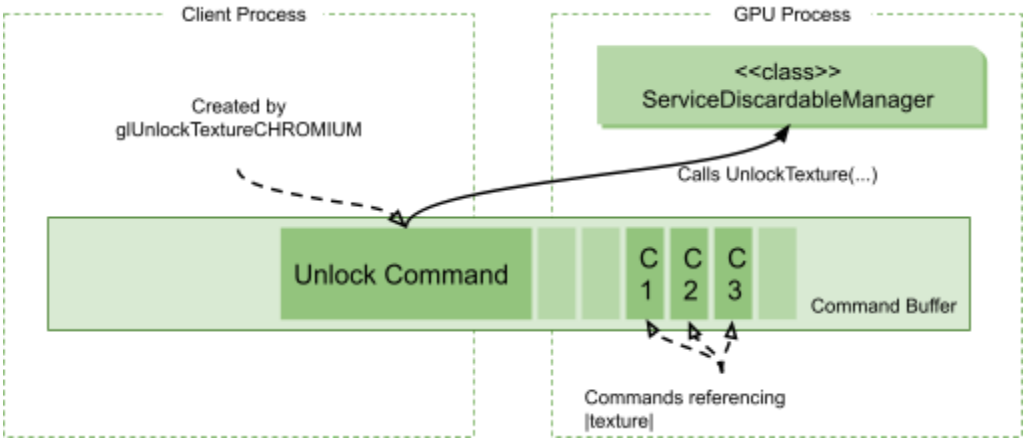
As commands in the command-buffer may rely on the locked data, unlock must not happen immediately in the Renderer process. Instead, it is sent over the command buffer to the GPU process, ensuring proper ordering with dependent commands.

In the client process:

1. Sends the texture ID to the GPU process.

In the GPU process

2. Receives the texture ID and notifies the ServiceDiscardableManager to unlock the texture. The ServiceDiscardableManager unlocks the texture, decrementing its DiscardableHandle value by 1. If the handle reaches the “unlocked” state (1), the manager updates the texture’s position in its LRU list.



Challenges

Eviction Strategy

When choosing which unlocked elements to evict in the ServiceDiscardableManager, a naive approach is to use an LRU. Based on feedback, it sounds like this may lead to cache thrashing in the case where we are just-over-budget frame after frame. We may want to consider a more randomized eviction strategy.

Size Changes

In the case of textures, an allocation can change size as levels are added. We need to communicate size changes from the TextureManager to the ServiceDiscardableManager as they happen.

Resource Deletion

As the ServiceDiscardableManager may be deleting resources across many GL contexts, we should take care to batch the deletes by context, in order to minimize context switches.

Future Work

Other GL Resource Types

Buffers, render targets, etc... can be handled in much the same way as textures, by adding methods to ServiceDiscardableManager/HandleManager to support these resource types.

GPU Memory Buffers

The system outlined here could be expanded to support GPU Memory Buffers. As GMBs are already created/destroyed independently of the GL Command Buffer, it may make sense to offer an alternative API for locking/unlocking, built on the same DiscardableHandle system. This would leave synchronization with GL commands up to the caller, much the same way GMB deletion is handled.