# NXT “Memory barriers”

cwallez@google.com, kainino@google.com



## API analysis on the gpuweb issue tracker

TL;DR of the analysis

What we call “memory barriers” regroups two things in hardware:

- Memory hazards where two parts of the GPU pipeline access the same memory. This can even be hazards of a stage with itself.
- Image swizzling and transitions between different types of swizzling. Called “image layout transitions” in Vulkan.

The different native APIs provide the following controls:

- D3D12: at any time, each resource can be in one writable usage or a combination of read-only usage. The API provides a way to transition a subresource between usages, causing both synchronization to avoid memory hazard, and swizzling changes to occur.
- Metal: everything is implicit, but the driver doesn’t emit “memory barriers” inside MTLRenderEncoders (only between render encoders).

- Vulkan: A vkPipelineBarrier allows specifying synchronization for memory hazards per-resource or globally. The same command independently specifies image swizzling transitions. Some implicit swizzling transitions happen at renderpass boundary.

# Usage transitions in NXT

Error management is briefly mentioned but we are intentionally not detailing it here. As usual, please don't mind the cosmetics of the API.

## Overview

For WebGPU we were looking for an implementation of "memory barriers" that is efficient to validate while still providing most of the control D3D12 and Vulkan provide. We settled on exposing validated "usage transitions" similar to D3D12 memory barriers.

More specifically each resource has the following two properties:

- A set of allowed usages that are immutable and set when the resource is created.
- A set of current usages. These can either be a single writable usage, or a combination of readable usages (note: this resembles the [readers-writer synchronization primitive]). An initial set of current usages can be provided during resource creation.

Example of usages are the following (current enums in NXT):

```
enum class BufferUsageBit : uint32_t {
    None = 0x00000000,
    MapRead = 0x00000001, // Read-only
    MapWrite = 0x00000002, // Writable
    TransferSrc = 0x00000004, // Read-only
    TransferDst = 0x00000008, // Writable
    Index = 0x00000010, // Read-only
    Vertex = 0x00000020, // Read-only
    Uniform = 0x00000040, // Read-only
    Storage = 0x00000080, // Writable
};

enum class TextureUsageBit : uint32_t {
    None = 0x00000000,
    TransferSrc = 0x00000001, // Read-only
    TransferDst = 0x00000002, // Writable
    Sampled = 0x00000004, // Read-only
    Storage = 0x00000008, // Writable
    OutputAttachment = 0x00000010, // Writable
    Present = 0x00000020, // Special, read-only but is exclusive like writable usages
};

// Plus additional code to make the enum classes act like flags

nxt::Device device;
nxt::Texture texture = device.CreateTextureBuilder()
```

```
    .SetAllowedUsage(nxt::TextureUsageBit::TransferDst | nxt::TextureUsageBit::Sampled)
    .SetInitialUsage(nxt::TextureUsageBit::TransferDst)
    // Other options
    .GetResult();
```

There are two types of commands in NXT:

- **Buffered commands** that live inside a command buffer and for which state-tracking side-effects happen when the command buffer is submitted to a queue (and in the order in which the commands appear in the command buffer)
- **Immediate commands** that act as if executed right away and update the state-tracking immediately. Submitting a command buffer to a queue is an immediate command.

When a command makes use of a resource, the usage must be part of the resource's current usages; otherwise, an error is produced. For immediate commands, the error (if any) will be produced immediately. For buffered commands, the error can be produced (a) immediately, (b) when the command buffer is finished being recorded, or at (c) queue submission.

There are commands to "transition" resources to a new set of current usages, since NXT does state-tracking. Each command has an immediate version as well as a buffered version.

```
//--- Prototype of the transition commands currently in NXT
nxt::Buffer::TransitionUsage(nxt::BufferUsageBit usage);
nxt::Texture::TransitionUsage(nxt::TextureUsageBit usage);

nxt::CommandBufferBuilder::TransitionBufferUsage(nxt::Buffer buffer, nxt::BufferUsageBit
usage);
nxt::CommandBufferBuilder::TransitionTextureUsage(nxt::Texture texture, nxt::TextureUsageBit
usage);

//--- Example of immediate usage transition
nxt::Texture texture;

// We were rendering to the texture
texture.TransitionUsage(nxt::TextureUsageBit::Present);
swapchain.Present(texture);

//--- Example of buffered usage transition
nxt::Buffer buffer;

nxt::CommandBuffer commands = device.CreateCommandBufferBuilder()
    .TransitionBuffer(buffer, nxt::BufferUsageBit::Storage)
    .BeginComputePass()
        // Update buffer with a compute shader
    .EndComputePass()
    .TransitionBuffer(buffer, nxt::BufferUsageBit::Index | nxt::BufferUsageBit::Vertex)
    // Render pass that uses buffer as an index and vertex buffer
    .GetResult();

queue.Submit(1, &commands);
```

Other interactions:

- NXT currently has these constraints on the set of allowed usage given at resource creation (see the buffer mapping discussion for more):
    - MapRead can only be used in combination with TransferDst.
    - MapWrite can only be used in combination with TransferSrc.
- As a validation optimization, there is an immediate command to “freeze” the set of current usages of a resource (disallowing any further transitions).
- No usage transition can happen in a subpass (though this restriction is not currently implemented), and there are restrictions on transitioning textures used as attachments.
- In the future there might be implicit transitions for render target attachment.
- The Present usage is special. It cannot be current with other usages and can only be transitioned to, not transitioned from. Only the NXT implementation can transition away from present, so that it can guarantee there is no data race between the present and other commands by the application.

# Mapping to each API

## D3D12

The semantics of the usage transitions are very close to the D3D12 memory barriers. The only special thing is that usage transitions are buffered so that they can be combined as a single “ResourceBarriers” call to help the D3D12 driver minimize the number of barriers.

## Metal

All NXT usage transitions are translated to no-ops; synchronization and layout transitions will be emitted implicitly by the Metal driver. There is no transition in NXT that Metal isn’t able to do implicitly.

## Vulkan

Things are similar to D3D12 except that synchronization, and potentially a layout transition, is deduced from the NXT usage transition.

# Validation

At resource creation, the application provides the set of allowed usages and the set of initial current usages; these are validated immediately. Some restrictions might be added, for example for depth-stencil textures, such that they can’t have some usages that wouldn’t work on the backing APIs.

Validation of immediate commands is done, well… immediately.

The validation of buffered commands is more complex. Essentially when a command would use a resource in the command buffer, it check if a previously buffered transition would either conflict (error case), or guarantee the usage. If there is no previously buffered transition the command buffer remembers which usages the resource must be in.

Then when the command buffer is submitted to the queue, command buffers are processed in order and first checked if they can “stitch” with the current resources’ usage, then their transitions are applied. “Stitching” validation contains the following constraints:

1. Resources used in the command buffer are still alive.
2. Resources transitioned in the command buffer haven’t had their usage frozen.
3. Resources used before they are transitioned in the command buffer have the correct current usage.

Note that any WebGPU will have to check for 1), so adding 2) and 3) on top of it doesn’t add much overhead (the cache line of the resource has already been accessed, and we might even be able to pack the state for all 3 into a single bitfield, for example). ~Jedi hand wave~

# Interactions with multiple queues

NXT’s usage transition model extends naturally to contexts with multiple queues.

In a single-queue context, we avoid data races by using the single-write-multiple-read pattern. This usage model extends naturally to multi-queue contexts, to prevent data races between queues.

In addition to the allowed-usage set and current-usage set, resources would gain an allowed-queue set and current-queue set. (These sets can default to the single “universal queue.”) Valid state for the current-usage and current-queue sets would be only:

- One write usage, and one queue.
- Any set of read usages, and any set of queues.

A resource’s current-queue set would be changed via immediate transition commands only. The transitions then implicitly define the synchronization between queues. The command buffer “stitching” validation (above) would be augmented with checks that the resources have the queue in their current queue set.

# Open questions

## Interactions with render passes

Vulkan has implicit texture swizzling transitions when executing render passes, as they can be optimized out on tiler GPUs. We might want to have a similar behavior.

If WebGPU has Vulkan style tile control, usage transitions inside render passes won't be possible because they semantically apply instantly. Vulkan allows setting global memory barriers, but NXT needs to be able to update its state tracking. Things are complicated but we might have a solution where we have "BeginTransition" and "EndTransition" buffered commands that are validated.

### Mipmap generation

Currently resources have to be completely in one set of usages. This makes it impossible to render custom mipmaps for a texture as it would need to be both in OutputAttachment (writable) and Sampled usages. For this use case we are thinking of allowing a per-mip-level usage for textures with a BIG (CPU validation) performance caveat to applications that they should strive to have all mips be in the same usage.

### "UAV" barriers

Storage buffers are both readable and writable, and it will be common to have multiple subsequent compute dispatches that all use the same buffer both for reading and writing. We want to have way to prevent memory hazard if the application requires it. Usage transitions would be suboptimal and we need the equivalent of a D3D12 "UAV barrier". This could be a buffered command only allowed in compute passes.

# Alternatives

## Vulkan-style memory barriers without validation

Pros:

- Potential to extract a couple more % of GPU perf from the API.
- Matches an existing an API and could make it easier to port native applications
- Reduces validation overhead on all backends

Cons:

- Known to be "OMG" level of difficult to use correctly, even for GPU experts
    - Getting code running in the first place will be hard
    - Mistakes might be invisible and cause the application to fail on untested hardware
- Would be expensive to add back validation. Using validation layers during development will be a crutch: even the Vulkan validation layers don't fully validate memory synchronization
- Not clear how secure unvalidated layout transitions are
- Same for unvalidated memory synchronization (probably ok but the Vulkan spec doesn't provide guarantees)

## Implicit memory barriers (like Metal)

Pros:

- API less verbose than NXT
- Slightly less *validation* overhead on all backends

Cons:

- Arguably no simpler than what's presented here
- Make it harder to understand exactly what happens
- *Implementation* overhead increases a lot on D3D12 and Vulkan
- Potentially suboptimal barriers in D3D12 and Vulkan backends
- Tile control and other future features will certainly require some form of explicit memory barriers