Texture Refactor in Ogre 2.1+

by Matías Nazareth Goldberg

Texture Refactor

- NEVER STALL
- Automatic batching
- Explicit Residency
- Managing Residency
- Async Texture Uploading
- Fix RenderTarget ↔ HardwarePixelBuffer ↔ Texture mess.
- Fix MSAA & MRT.

NEVER STALL

- If a texture isn't ready, show a 4x4 blank texture instead.
- If the mips could be loaded first, show the first 64x64 mips.
- When the texture is done loading; it is modified so that its internal API object points to the actual texture object.

Stalls can only happen because of two reasons:

- User requested to map for reading a texture that isn't done loading (they can query whether it's ready though)
- User specifically requested to stall (e.g. level loading, user wants to prevent showing a blank texture)

Automatic batching

- Similar to what HlmsTextureManager does.
- Textures will have two modes:
  - Automatic
  - Manual

Automatic batching

Manual

The Texture is exactly the type requested (e.g. 2D texture won't get a 2D array instead)

While a texture is transitioning to Resident, no 64x64 is used, but the 4x4 dummy one will be used instead (blank texture).

Automatic batching

Automatic

The Texture can be of a different type. Most normally we'll treat 2D textures internally as a slice to a 2D array texture

Ogre will keep three API objects:

- A single 4x4 texture. Blank.
- An array of 2D textures of 64x64. One of its slices will contain the mips of the texture being loaded
- An array of 2D textures in which one of its slices the fully resident texture will live.

Each time we change the internal API object, HlmsDatablocks need to be notified so it can pack the arrays, update the slices to the GPU, and compute the texture hashes.

- All of that (except updating slices to the GPU) can be done in a worker thread, then all the values swapped w/ the Datablock's.

Explicit Residency

- Current main problem is textures need to be parsed upfront.
- Blows loading time.
- Can cause out of GPU memory errors.

Explicit Residency

- Current main problem is textures need to be parsed upfront.
- Blows loading time.
- Can cause out of GPU memory errors.

3 MODES:

On storage

On Sys. RAM

Resident

Explicit Residency

- Current main problem is textures need to be parsed upfront.
- Blows loading time.
- Can cause out of GPU memory errors.

3 MODES:

On storage | **On Sys. RAM** | **Resident**

- Texture is on storage (i.e. sourced from disk, from listener)
- A 4x4 blank texture will be shown if user attempts to use this Texture.
- No memory is consumed.

Explicit Residency

- Current main problem is textures need to be parsed upfront.
- Blows loading time.
- Can cause out of GPU memory errors.

3 MODES:

On storage | **On Sys. RAM** | **Resident**

- Texture is on System RAM.
- If the texture is fully not resident, a 4x4 blank texture will be shown if user attempts to use this Texture.
- If the texture is transitioning to Resident, a 64x64 mip version of the texture will be shown. Starts blank, gets progressively filled. If not enough resources we may just display the 4x4 blank dummy.
- Explicit APIs (D3D12 / Vulkan) may use true OS residency funcs.

Explicit Residency

- Current main problem is textures need to be parsed upfront.
- Blows loading time.
- Can cause out of GPU memory errors.

3 MODES:

On storage | **On Sys. RAM** | **Resident**

- It's loaded on VRAM.
- Ready to be used by GPU.
- May keep a copy on system RAM (user tweakable)
- Texture may transition directly from on storage to Resident.
- However, Sys. RAM is a very attractive option for 64-bit editors (keep everything on Sys. RAM; load to GPU based on scene demands)

Managing Residency

- User tells us priorities via a “rank” system.
  1. Must be loaded first. Must be kept always resident. Rank = 0
  2. Can be paged out if necessary. Rank > 0
  3. Naming convention: avoid “high” rank and “low” rank since it’s confusing. Prefer “top rank” for high priority, and “bottom rank” for low priority. (better suggestions are welcomed!)
- Every time a texture is used the frame count it was used in is saved. We’ll refer to this as textures “getting touched”.
- The older a texture remains unused, the more likely it will be paged out (if memory thresholds are exceeded, and multiplied by rank).
- The minimum distance to camera could be saved as well. More distant textures may be paged out and replaced with 64x64 mips.

Managing Residency

- User tells Ogre how to deal paged out textures:
  - Textures should be saved to Sys. RAM or...
  - Discarded (will read from disk next time, speed will depend on whether OS still has a cached copy of the file) or...
  - Always keep a Sys. RAM copy to avoid readback.

Managing Residency

- Keep list of commands in main thread to maintain order.
  - If user loads a texture, then uploads, then uploads some more, they need to be ordered.
- Reading must be done via AsyncTickets. Mapping immediately will obviously stall.

Managing Residency

- Immutable textures (D3D11)
  - Only Rank 0 textures can be made immutable (since paging them in/out would be problematic and must be done at per texture array granularity)
  - User can specify it started a “loading screen” (i.e. not streaming) so that Ogre will defer Rank 0 textures upload until it fills all the slices for an array.
  - Alternatively, user can specify a list of textures it wants to load as immutable into a single array.

Managing Residency

- How to touch textures to update distance to camera and frame used?
  - 100.000 objects on scene w/ 5 textures each = 500.000 touches per frame!
- Touch HlmsDatablocks instead!
- Worker thread will periodically iterate through all active datablocks in scene checking for touched datablocks and updating its textures w/ the datablock's touch.
- This implies HlmsDatablocks and worker thread need to be sync.
  - No need for a mutex!
  - When its textures change, HlmsDatablock can send a message with an array w/ the new textures.
  - Worker thread will work with the old array until the message arrives.
- What about textures without datablocks?
  - You *need* the Hlms to render.
  - Hlms implementations may touch a texture if not part of datablock

Async Texture Uploading

- Async streaming done via worker thread.
- All textures always uploaded from worker thread unless specifically requested (i.e. user-managed ones).
- Use double buffer scheme.
- While a texture is not yet resident, we show either the 64x64 or 4x4 texture. Until it's ready.
- Preallocate a bunch of StagingTextures for the worker thread.
  - On GLES2, use a CPU-side buffer to emulate lack of ptr mapping.

Async Texture Uploading

- Async streaming done via worker thread.
- Use double buffer scheme.
- On GLES2, use a CPU-side buffer to emulate lack of ptr mapping.

Main Thread

1. **Main thread maps big staging textures.**
2. **Swap w/ worker thread.**
3. **Unmap & execute commands**
4. **Update textures (to point to dummy/real texture API object)**
5. **Update HlmsDatablock's texture hash.**
6. **Repeat 1.**

Worker thread

1. **Swap w/ main thread**
2. **Iterate trough all HlmsDatablocks**
3. **memcpy unpaged textures to gpu ptr.**
4. **Create copy & other cmds for main thread to execute**
5. **Repeat 1.**

Async Texture Uploading

- Creating and destroying datablocks also needs to be sync'ed.
  - Probably use message passing as well, and delay destruction until worker thread informs he's aware the datablock will be removed.
- Best place to touch a datablock is in MovableObject::updateAllBounds
  - Profiling says there's spare cycles in that function :)
  - Even if an object fails frustum test, it could be right behind us. Distance to camera and knowing a texture belongs to a currently active object is far more important.
  - Happens once per frame. Frustum tests can happen multiple times.

RenderTarget ↔ HardwarePixelBuffer ↔ Texture.

- Get rid of HardwarePixelBuffer
- A 'texture' contains all mip levels, all faces/slices.
- Use Metal's approach to render targets:
  - Textures to be used as RTT must be marked as such.
  - Create immutable RenderView:
  - Can be attached N textures with mip/slice/face for colour.
  - Can be attached a texture for depth.
  - Can be attached a texture for stencil.
  - Depth can be sourced explicitly, or from a pool assigned at creation.
    - Previously the the depth was grabbed from pool on first use.
  - Trying to attach a non-marked RTT Texture to a RenderView throws
- Makes resource transitions (D3D12 / Vulkan) easier by dealing with Textures all the time instead of having to deal with RenderTargets AND Textures.

Compositor

- Textures are no longer declared as MRT. i.e. only one format.
- To do MRT, a RenderView can be created explicitly from script.
- If the texture is used directly, a RenderView is implicitly created.
- Support selecting array slices for 3D and 2D array textures.

Script examples:

```
texture 2d texture0 640 480 PF_R8G8B8A8
texture 2d texture1 640 480 PF_R8G8B8A8
texture 2d texture2 640 480 PF_R8G8B8A8

renderview myMrt texture0 texture1 texture2
```

Matías Nazareth Goldberg
@matiasgoldberg