Beyond Porting

How Modern OpenGL can Radically Reduce Driver Overhead
Who are we?

- Cass Everitt, NVIDIA Corporation
- John McDonald, NVIDIA Corporation
What will we cover?

- Dynamic Buffer Generation
- Efficient Texture Management
- Increasing Draw Call Count
Dynamic Buffer Generation

Problem

- Our goal is to generate dynamic geometry directly in place.
- It will be used one time, and will be completely regenerated next frame.
  - Particle systems are the most common example
  - Vegetation / foliage also common
void UpdateParticleData(uint _dstBuf) {
    BindBuffer(ARRAY_BUFFER, _dstBuf);
    access = MAP_UNSYNCHRONIZED | MAP_WRITE_BIT;
    for particle in allParticles {
        dataSize = GetParticleSize(particle);
        void* dst = MapBuffer(ARRAY_BUFFER, offset, dataSize, access);
        (*(Particle*)dst) = *particle;
        UnmapBuffer(ARRAY_BUFFER);
        offset += dataSize;
    }
};

// Now render with everything.
void UpdateParticleData(uint _dstBuf) {
    BindBuffer(ARRAY_BUFFER, _dstBuf);
    access = MAP_UNSYNCHRONIZED | MAP_WRITE_BIT;
    for particle in allParticles {
        dataSize = GetParticleSize(particle);
        void* dst = MapBuffer(ARRAY_BUFFER, offset, dataSize, access);
        (*(Particle*)dst) = *particle;
        UnmapBuffer(ARRAY_BUFFER);
        offset += dataSize;
    }
}

// Now render with everything.

This is so slow.
Driver interlude

First, a quick interlude on modern GL drivers

In the application (client) thread, the driver is very thin.

- It simply packages work to hand off to the server thread.

The server thread does the real processing

- It turns command sequences into push buffer fragments.
Healthy Driver Interaction Visualized

Application
Driver (Client)
Driver (Server)
GPU

- Thread separator
- Component separator

State Change
Action Method (draw, clear, etc)
Present
Avoids an application-GPU sync point (a CPU-GPU sync point)

But causes the Client and Server threads to serialize

- This forces all pending work in the server thread to complete
- It’s quite expensive (almost always needs to be avoided)
Q: What’s better than mapping in an unsynchronized manner?
A: Keeping around a pointer to GPU-visible memory forever.
Introducing: ARB_buffer_storage
ARB_buffer_storage

- Conceptually similar to ARB_texture_storage (but for buffers)
- Creates an immutable pointer to storage for a buffer
  - The pointer is immutable, the contents are not.
  - So BufferData cannot be called—BufferSubData is still okay.
- Allows for extra information at create time.
- For our usage, we care about the PERSISTENT and COHERENT bits.
  - PERSISTENT: Allow this buffer to be mapped while the GPU is using it.
  - COHERENT: Client writes to this buffer should be immediately visible to the GPU.

http://www.opengl.org/registry/specs/ARB/buffer_storage.txt
ARB_buffer_storage cont’d

- Also affects the mapping behavior (pass persistent and coherent bits to MapBufferRange)
- Persistently mapped buffers are good for:
  - Dynamic VB / IB data
  - Highly dynamic (~per draw call) uniform data
  - Multi_draw_indirect command buffers (more on this later)
- Not a good fit for:
  - Static geometry buffers
  - Long lived uniform data (still should use BufferData or BufferSubData for this)
Armed with persistently mapped buffers

// At the beginning of time
flags = MAP_WRITE_BIT | MAP_PERSISTENT_BIT | MAP_COHERENT_BIT;
BufferStorage(ARRAY_BUFFER, allParticleSize, NULL, flags);
mParticleDst = MapBufferRange(ARRAY_BUFFER, 0, allParticleSize,
                                flags);
mOffset = 0;

// allParticleSize should be ~3x one frame’s worth of particles
// to avoid stalling.
void UpdateParticleData(uint _dstBuf) {
    BindBuffer(ARRAY_BUFFER, _dstBuf);
    access = MAP_UNSYNCHRONIZED | MAP_WRITE_BIT;
    for particle in allParticles {
        dataSize = GetParticleSize(particle);
        void* dst = MapBuffer(ARRAY_BUFFER, offset, dataSize, access);
        (*(Particle*)dst) = *particle;
        offset += dataSize;
    }
}

// Now render with everything.
void UpdateParticleData() {

    for particle in allParticles {
        dataSize = GetParticleSize(particle);

        mParticleDst[mOffset] = *particle;
        mOffset += dataSize; // Wrapping not shown
    }
}

// Now render with everything.
Test App
Performance results

- 160,000 point sprites
- Specified in groups of 6 vertices (one particle at a time)
- Synthetic (naturally)

<table>
<thead>
<tr>
<th>Method</th>
<th>FPS</th>
<th>Particles / S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map(UNSYCHRONIZED)</td>
<td>1.369</td>
<td>219,040</td>
</tr>
<tr>
<td>BufferSubData</td>
<td>17.65</td>
<td>2,824,000</td>
</tr>
<tr>
<td>D3D11 Map(NO_OVERWRITE)</td>
<td>20.25</td>
<td>3,240,000</td>
</tr>
</tbody>
</table>
Performance results

- 160,000 point sprites
- Specified in groups of 6 vertices (one particle at a time)
- Synthetic (naturally)

<table>
<thead>
<tr>
<th>Method</th>
<th>FPS</th>
<th>Particles / S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map(UNSYNCHRONIZED)</td>
<td>1.369</td>
<td>219,040</td>
</tr>
<tr>
<td>BufferSubData</td>
<td>17.65</td>
<td>2,824,000</td>
</tr>
<tr>
<td>D3D11 Map(NO_OVERWRITE)</td>
<td>20.25</td>
<td>3,240,000</td>
</tr>
<tr>
<td>Map(COHERENT</td>
<td>PERSISTENT)</td>
<td>79.9</td>
</tr>
</tbody>
</table>

Room for improvement still, but much, much better.
The other shoe

- You are responsible for not stomping on data in flight.
- Why 3x?
  - 1x: What the GPU is using right now.
  - 2x: What the driver is holding, getting ready for the GPU to use.
  - 3x: What you are writing to.
- 3x should ~ guarantee enough buffer room*…
- Use fences to ensure that rendering is complete before you begin to write new data.
Use FenceSync to place a new fence.

When ready to scribble over that memory again, use ClientWaitSync to ensure that memory is done.

- ClientWaitSync will block the client thread until it is ready
- So you should wrap this function with a performance counter
- And complain to your log file (or resize the underlying buffer) if you frequently see stalls here

For complete details on correct management of buffers with fencing, see Efficient Buffer Management [McDonald 2012]
Efficient Texture Management

Or “how to manage all texture memory myself”
Problem

- Changing textures breaks batches.
- Not all texture data is needed all the time
  - Texture data is large (typically the largest memory bucket for games)
- Bindless solves this, but can hurt GPU performance
  - Too many different textures can fall out of TexHdr$
  - Not a bindless problem per se
Reserve – The act of allocating virtual memory
Commit – Tying a virtual memory allocation to a physical backing store (Physical memory)
Texture Shape – The characteristics of a texture that affect its memory consumption
   Specifically: Height, Width, Depth, Surface Format, Mipmap Level Count
Old Solution

- Texture Atlases
- Problems
  - Can impact art pipeline
  - Texture wrap, border filtering
  - Color bleeding in mip maps
Texture Arrays

- Introduced in GL 3.0, and D3D 10.
- Arrays of textures that are the same shape and format
- Typically can contain many “layers” (2048+)
- Filtering works as expected
- As does mipmapping!
Sparse Bindless Texture Arrays

- Organize loose textures into Texture Arrays.
- Sparsely allocate Texture Arrays
  - Introducing ARB_sparse_texture
  - Consume virtual memory, but not physical memory
- Use Bindless handles to deal with as many arrays as needed!
  - Introducing ARB_bindless_texture
Applications get fine-grained control of physical memory for textures with large virtual allocations

Inspired by Mega Texture

Primary expected use cases:
- Sparse texture data
- Texture paging
- Delayed-loading assets

http://www.opengl.org/registry/specs/ARB/sparse_texture.txt
Textures specified by GPU-visible “handle” (really an address)
   Rather than by name and binding point
Can come from ~anywhere
   Uniforms
   Varying
   SSBO
   Other textures
Texture residency also application-controlled
   Residency is “does this live on the GPU or in sysmem?”
https://www.opengl.org/registry/specs/ARB/bindless_texture.txt
Advantages

- Artists work naturally
- No preprocessing required (no bake-step required)
  - Although preprocessing is helpful if ARB_sparse_texture is unavailable
- Reduce or eliminate TexHdr$ thrashing
  - Even as compared to traditional texturing
- Programmers manage texture residency
- Works well with arbitrary streaming
- Faster on the CPU
- Faster on the GPU
Disadvantages

- Texture addresses are now structs (96 bits).
  - 64 bits for bindless handle
  - 32 bits for slice index (could reduce this to 10 bits at a perf cost)
- ARB_sparse_texture implementations are a bit immature
  - Early adopters: please *bring us your bugs*.
- ARB_sparse_texture requires base level be a multiple of tile size
  - (Smaller is okay)
  - Tile size is queried at runtime
  - Textures that are power-of-2 should almost always be safe.
Implementation Overview

- When creating a new texture...
- Check to see if any suitable texture array exists
  - Texture arrays can contain a large number of textures of the same shape
  - Ex. Many TEXTURE_2Ds grouped into a single TEXTURE_2D_ARRAY
- If no suitable texture, create a new one.
Texture Container Creation (example)

- GetIntegerv( MAX_SPARSE_ARRAY_TEXTURE_LAYERS, maxLayers );
  - Choose a reasonable size (e.g. array size ~100MB virtual )

- If new internalFormat, choose page size
  - GetInternalformativ( ..., internalformat, NUM_VIRTUAL_PAGE_SIZES, 1, &numIndexes);
  - Note: numIndexes can be 0, so have a plan
  - Iterate, select suitable pageSizeIndex

- BindTexture( TEXTURE_2D_ARRAY, newTexArray );
- TexParameteri( TEXTURE_SPARSE, TRUE );
- TexParameteri( VIRTUAL_PAGE_SIZE_INDEX, pageSizeIndex );

- Allocate the texture’s virtual memory using TexStorage3D
Specifying Texture Data

- Using the located/created texture array from the previous step
- Allocate a layer as the location of our data
- For each mipmap level of the allocated layer:
  - Commit the entire mipmap level (using `TexPageCommitment`)
  - Specify actual texel data as usual for arrays
    - `gl(Compressed|Copy|)TexSubImage3D`
  - PBO updates are fine too

Allocated layer

uncommitted layer
Freeing Textures

To free the texture, reverse the process:

- Use `TexPageCommitment` to mark the entire layer (slice) as free.
- Do once for each mipmap level
- Add the layer to the free list for future allocation
Combining with Bindless to eliminate binds

At container create time:
- Specify sampling parameters via `SamplerParameter` calls first
- Call `GetTextureSamplerHandleARB` to return a GPU-visible pointer to the texture+sampler container
- Call `MakeTextureHandleResident` to ensure the resource lives on the GPU

At delete time, call `MakeTextureHandleNonResident`

With bindless, you explicitly manage the GPU’s working set
Using texture data in shaders

When a texture is needed with the default sampling parameters

Create a GLSL-visible TextureRef object:

```glsl
struct TextureRef {
    sampler2DArray container;
    float slice;
};
```

When a texture is needed with custom sampling parameters

Create a separate sampler object for the shader with the parameters

Create a bindless handle to the pair using `GetTextureSamplerHandle`, then call `MakeTextureHandleResident` with the new value

And fill out a TextureRef as above for usage by GLSL
C++ Code

- Basic implementation (some details missing)
- BSD licensed (use as you will)

https://github.com/nvMcJohn/apitest/blob/pdoane_newtests/sparse_bindless_texarray.h
https://github.com/nvMcJohn/apitest/blob/pdoane_newtests/sparse_bindless_texarray.cpp
Increasing Draw Call Count

Let’s draw *all the calls!*
All the Draw Calls!

Problem

- You want more draw calls of smaller objects.
- D3D is slow at this.
- Naïve GL is faster than D3D, but not fast enough.
XY Problem

Y: How can I have more draw calls?
X: You don’t really care if it’s more draw calls, right?
Really what you want is to be able to draw more small geometry groupings. More objects.
Well why didn’t you just say so??

First, some background.

- What makes draw calls slow?
- Real world API usage
- Draw Call Cost Visualization
Some background

What causes slow draw calls?
- Validation is the biggest bucket (by far).
- Pre-validation is “difficult”
- “Every application does the same things.”
  - Not really. Most applications are in completely disjoint states
  - Try this experiment: What is important to you?
  - Now ask your neighbor what’s important to him.
Why is prevalidation difficult?

- The GPU is an exceedingly complex state machine.
  - (Honestly, it’s probably the most complex state machine in all of CS)
- Any one of those states may have a problem that requires WAR
- Usually the only problem is overall performance
  - But sometimes not. 😊
- There are millions of tests covering NVIDIA GPU functionality.
How can app devs mitigate these costs?

- Minimize state changes.
- All state changes are not created equal!

Cost of a draw call:

Small fixed cost + Cost of validation of changed state
Feels limiting…

- Artists want lots of materials, and small amounts of geometry
- Even better: What if artists just didn’t have to care about this?
  - Ideal Programmer->Artist Interaction
  - “You make pretty art. I’ll make it fit.”
Relative costs of State Changes

- Render Target: ~60K / s
- Program: ~300K / s
- ROP: ~1.5M / s
- Texture Bindings
- Vertex Format: ~1.5M / s
- UBO Bindings
- Vertex Bindings
- Uniform Updates: ~10M / s

Note: Not to scale
Real World API frequency

API usage looks roughly like this…

- Increasing Frequency of Change
  - Render Target (scene)
  - Per Scene Uniform Buffer + Textures
  - IB / VB and Input Layout
  - Shader (Material)
  - Per-material Uniform Buffer + Textures
  - Per-object Uniform Buffer + Textures
  - Per-piece Uniform Buffer + Textures
  - Draw
Draw Calls visualized (cont’d)

Read down, then right
Black—no change

- **Render Target**
- **Program**
- **ROP**
- **Texture**
- **UBO Binding**
- **Vertex Format**
- **Uniform Updates**
- **Draw**
Goals

- Let’s minimize validation costs without affecting artists
- Things we need to be fast (per app call frequency):
  - Uniform Updates and binding
  - Texture Updates and binding
- These happen most often in app, ergo driving them to ~0 should be a win.
Textures

- Using Sparse Bindless Texture Arrays (as previously described) solves this.
  - All textures are set before any drawing begins
  - (No need to change textures between draw calls)
- Note that from the CPU’s perspective, *just* using bindless is sufficient.

That was easy.
Eliminating Texture Binds -- visualized

Increasing Frequency of Change
- Render Target (scene)
- Per Scene Uniform Buffer + Textures
- IB / VB and Input Layout
- Shader (Material)
- Per-material Uniform Buffer + Textures
- Per-object Uniform Buffer + Textures
- Per-piece Uniform Buffer + Textures
- Draw
Increasing Frequency of Change
- Render Target (scene)
- Per Scene Uniform Buffer
- IB / VB and Input Layout
- Shader (Material)
- Per-material Uniform Buffer
- Per-object Uniform Buffer
- Per-piece Uniform Buffer
- Draw

- Render Target
- Program
- ROP
- Texture
- UBO Binding
- Uniform Updates
- Draw
- Vertex Format
Buffer updates (old and busted)

- Typical Scene Graph Traversal

```python
for obj in visibleObjectSet {
    update(buffer, obj);
    draw(obj);
}
```
Buffer updates (new hotness)

Typical Scene Graph Traversal

```python
for obj in visibleObjectSet {
    update(bufferFragment, obj);
}
```

```python
for obj in visibleObjectSet {
    draw(obj);
}
```
Rather than one buffer per object, we share UBOs for many objects.

ie, given struct ObjectUniforms { /* … */ };

// Old (probably not explicitly instantiated,
// just scattered in GLSL)
ObjectUniforms uniformData;

// New
ObjectUniforms uniformData[ObjectsPerKickoff];

Use persistent mapping for even more win here!

For large amounts of data (bones) consider SSBO.

Introducing ARB_shader_storage_buffer_object
SSBO?

- Like “large” uniform buffer objects.
  - Minimum required size to claim support is 16M.
- Accessed like uniforms in shader
- Support for better packing (std430)
- Caveat: They are typically implemented in hardware as textures (and can introduce dependent texture reads)
  - Just one of a laundry list of things to consider, not to discourage use.
- http://www.opengl.org/registry/specs/ARB/shader_storage_buffer_object.txt
Eliminating Buffer Update Overhead

Increasing Frequency of Change
- Render Target (scene)
- Per-Scene Uniform Buffer
- IB / VB and Input Layout
- Shader (Material)
- Per-material Uniform Buffer
- Per-object Uniform Buffer
- Per-piece Uniform Buffer
- Draw

- Render Target
- Program
- ROP
- Texture
- UBO Binding
- Uniform Updates
- Draw
- Vertex Format
Sweet!

- Increasing Frequency of Change
  - Render Target (scene)
  - IB / VB and Input Layout
  - Shader (Material)
  - Draw (* each object *)

- Hrrrrmmmmmmmm....

<table>
<thead>
<tr>
<th>Render Target</th>
<th>Texture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>UBO Binding</td>
</tr>
<tr>
<td>ROP</td>
<td>Vertex Format</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uniform Updates</th>
<th>Draw</th>
</tr>
</thead>
</table>
So now...

- It’d be awesome if we could do all of those kickoffs at once.
- Validation is already only paid once
- But we could just pay the constant startup cost once.
- If only......
So now…

- It’d be awesome if we could do all of those kickoffs at once.
- Validation is already only paid once
- But we could just pay the constant startup cost once.
- If only…….

- Introducing ARB_multi_draw_indirect
ARB_multi_draw_indirect

- Allows you to specify parameters to draw commands from a buffer.
  - This means you can generate those parameters wide (on the CPU)
  - Or even on the GPU, via compute program.

http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt
void MultiDrawElementsIndirect(enum mode,
    enum type
    const void* indirect,
    sizei primcount,
    sizei stride);
const ubyte * ptr = (const ubyte *)indirect;
for (i = 0; i < primcount; i++) {
    DrawArraysIndirect(mode,
                        (DrawArraysIndirectCommand*)ptr);
    if (stride == 0)
        { ptr += sizeof(DrawArraysIndirectCommand); }
    else { ptr += stride; }
}
typedef struct {
    uint count;
    uint primCount;
    uint first;
    uint baseInstance;
} DrawArraysIndirectCommand;
Knowing which shader data is mine

- Use ARB_shader_draw_parameters, a necessary companion to ARB_multi_draw_indirect
  - Adds a builtin to the VS: DrawID (InstanceID already available)
    - This tells you which command of a MultiDraw command is being executed.
    - When not using MultiDraw, the builtin is specified to be 0.
  - Caveat: Right now, you have to pass this down to other shader stages as an interpolant.
  - Hoping to have that rectified via ARB or EXT extension “real soon now.”

http://www.opengl.org/registry/specs/ARB/shader_draw_parameters.txt
Applying everything

- CPU Perf is massively better
  - 5-30x increase in number of distinct objects / s
- Interaction with driver is decreased ~75%
- Note: GPU perf can be affected negatively (although not too badly)

As always: Profile, profile, profile.
Visualized Results

- Render Target
- Texture
- UBO Binding
- Vertex Format
- Uniform Updates
- Draw
- MultiDraw
Where we came from

- Render Target
- Program
- ROP
- Texture
- UBO Binding
- Vertex Format
- Uniform Updates
- Draw
Conclusion

Go forth and work magnify.
Questions?

- jmcdonald at nvidia dot com
- cass at nvidia dot com