OpenGL Insights

Edited by

Patrick Cozzi and Christophe Riccio

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

Asynchronous Buffer Transfers 28

Ladislav Hrabcak and Arnaud Masserann

28.1 Introduction

Most 3D applications send large quantities of data from the CPU to the GPU on a regular basis. Possible reasons include

- streaming data from hard drive or network: geometry, clipmapping, level of detail (LOD), etc.;
- updating skeletal and blend-shapes animations on the CPU;
- computing a physics simulation;
- generating procedural meshes;
- data for *instancing*;
- setting uniform parameters for shaders with *uniform buffers*.

Likewise, it is often useful to read generated data back from the GPU. Possible scenarios are

- video capture [Kemen 10];
- physics simulation;
- page resolver pass in virtual texturing;
- image histogram for computing HDR tonemapping parameters.

While copying data back and forth to the GPU is easy, the PC architecture, without unified memory, makes it harder to do it fast. Furthermore, the OpenGL API specification doesn't tell how to do it efficiently, and a naive use of data-transfer functions wastes processing power on both the CPU and the GPU by introducing pauses in the execution of the program.

In this chapter, for readers familiar with buffer objects, we are going to explain what happens in the drivers and then present various methods, including unconventional ones, to transfer data between the CPU and the GPU with maximum speed. If an application needs to transfer meshes or textures frequently and efficiently, these methods can be used to improve its performance. In this chapter, we will be using OpenGL 3.3, which is the Direct3D 10 equivalent.

28.1.1 Explanation of Terms

First, in order to match the OpenGL specification, we refer to the GPU as the *device*.

Second, when calling OpenGL functions, the drivers translate calls into commands and add them into an internal queue on the CPU side. These commands are then consumed by the device asynchronously. This queue has already been refered to as the command queue, but in order to be clear, we refer to it as the *device command queue*.

Data transfers from CPU memory to device memory will be consistently referred to as *uploading* and transfers from the device memory to CPU memory as *downloading*. This matches the client/server paradigm of OpenGL.

Finally, *pinned memory* is a portion of the main RAM that can be directly used by the device through the PCI express bus (PCI-e). This is also known as *page-locked memory*.

28.2 Buffer Objects

There are many buffer-object targets. The most well-known are `GL_ARRAY_BUFFER` for vertex attributes and `GL_ELEMENT_ARRAY_BUFFER` for vertex indices, formerly known as vertex buffer objects (VBOs). However, there are also `GL_PIXEL_PACK_BUFFER` and `GL_TRANSFORM_FEEDBACK_BUFFER` and many other useful ones. As all these targets relate to the same kind of objects, they are all equivalent from a transfer point of view. Thus, everything we will describe in this chapter is valid for any buffer object target.

Buffer objects are linear memory regions allocated in device memory or in CPU memory. They can be used in many ways, such as

- the source of vertex data,
- texture buffer, which allows shaders to access large linear memory regions (128–256 MTexels on GeForce 400 series and Radeon HD 5000 series) [ARB 09a],

- uniform buffers,
- pixel buffer objects for texture upload and download.

28.2.1 Memory Transfers

Memory transfers play a very important role in OpenGL, and their understanding is a key to achieving high performance in 3D applications. There are two major desktop GPU architectures: discrete GPUs and integrated GPUs. Integrated GPUs share the same die and memory space with the CPU, which gives them an advantage because they are not limited by the PCI-e bus in communication. Recent APUs from AMD, which combine a CPU and GPU in a single die, are capable of achieving a transfer rate of 17GB/s which is beyond the PCI-e ability [Boudier and Sellers 11]. However, integrated units usually have mediocre performance in comparison to their discrete counterparts. Discrete GPUs have a much faster memory on board (30–192 GB/s), which is a few times faster than the conventional memory used by CPUs and integrated GPUs (12–30 GB/s) [Intel 08].

The direct memory access (DMA) controller allows the OpenGL drivers to asynchronously transfer memory blocks from user memory to device memory without wasting CPU cycles. This asynchronous transfer is most notably known for its widespread usage with pixel buffer objects [ARB 08], but can actually be used to transfer any type of buffer. It is important to note that the transfer is asynchronous from the CPU point of view only: Fermi (GeForce 400 Series) and Nothern Islands (Radeon HD 6000 Series) GPUs can't transfer buffers and render at the same time, so all OpenGL commands in the command queue are processed sequentially by the device. This limitation comes partially from the driver, so this behavior is susceptible to change and can be different in other APIs like CUDA, which exposes these GPU-asynchronous transfers. There are some exceptions like the NVIDIA Quadro, which can render while uploading and downloading textures [Venkataraman 10].

There are two ways to upload and download data to the device. The first way is to use the `glBufferData` and `glBufferSubData` functions. The basic use of these functions is quite straightforward, but it is worth understanding what is happening behind the scenes to get the best functionality.

As shown in Figure 28.1, these functions take user data and copy them to pinned memory directly accessible by the device. This process is similar to a standard `memcpy`. Once this is done, the drivers start the DMA transfer, which is asynchronous, and return from `glBufferData`. Destination memory depends on usage hints, which will be explained in the next section, and on driver implementation. In some cases, the data stay in pinned CPU memory and is used by the GPU directly from this memory, so the result is one hidden `memcpy` operation in every `glBufferData` function. Depending on how the data are generated, this `memcpy` can be avoided [Williams and Hart 11].

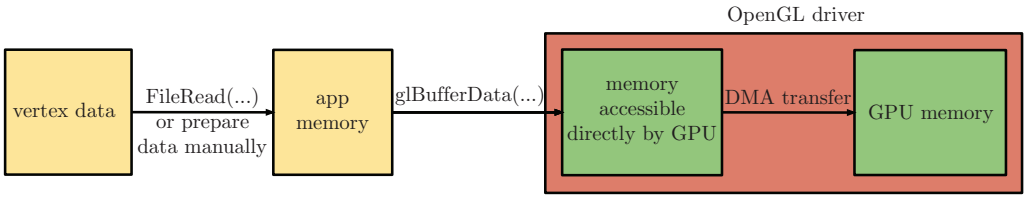

Figure 28.1. Buffer data upload with `glBufferData` / `glBufferSubData`.

A more efficient way to upload data to the device is to get a pointer to the internal drivers' memory with the functions `glMapBuffer` and `glUnmapBuffer`. This memory should, in most cases, be pinned, but this behavior can depend on the drivers and available resources. We can use this pointer to fill the buffer directly, for instance, using it for file read/write operations, so we will save one copy per memory transfer. It is also possible to use the `ARB_map_buffer_alignment` extension, which ensures that the returned pointer is aligned at least on a 64-byte boundary, allowing SSE and AVX instructions to compute the buffer's content. Mapping and unmapping is shown in Figure 28.2.

The returned pointer remains valid until we call `glUnmapBuffer`. We can exploit this property and use this pointer in a worker thread, as we will see later in this chapter.

Finally, there are also `glMapBufferRange` and `glFlushMappedBufferRange`, similar to `glMapBuffer`, but they have additional parameters which can be used to improve the transfer performance and efficiency. These functions can be used in many ways:

- `glMapBufferRange` can, as its name suggests, map only specific subsets of the buffer. If only a portion of the buffer changes, there is no need to reupload it completely.

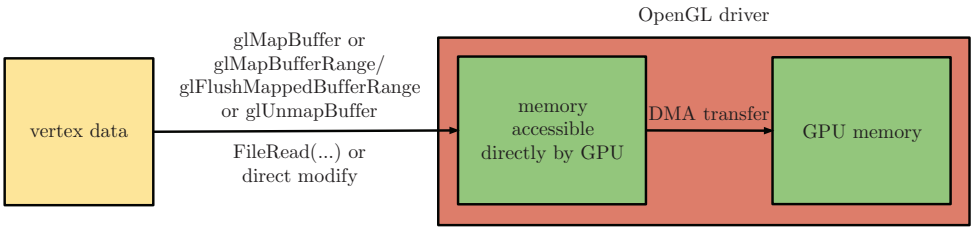

Figure 28.2. Buffer data upload with `glMapBuffer` / `glUnmapBuffer` or `glMapBufferRange` / `glFlushMappedBufferRange`.

- We can create a big buffer, use the first half for rendering, the second half for updating, and switch the two when the upload is done (manual double buffering).
- If the amount of data varies, we can allocate a big buffer, and map/unmap only the smallest possible range of data.

28.2.2 Usage Hints

The two main possible locations where the OpenGL drivers can store our data are CPU memory and device memory. CPU memory can be page-locked (pinned), which means that it cannot be paged out to disk and is directly accessible by device, or paged, i.e., accessible by the device too, but access to this memory is much less efficient. We can use a hint to help the drivers make this decision, but the drivers can override our hint, depending on the implementation.

Since Forceware 285, NVIDIA drivers are very helpful in this area because they can show exactly where the data will be stored. All we need is to enable the `GL_ARB_debug_output` extension and use the `WGL_CONTEXT_DEBUG_BIT_ARB` flag in `wglCreateContextAttribs`. In all our examples, this is enabled by default. See Listing 28.1 for an example output and Chapter 33 for more details on this extension.

It seems that NVIDIA and AMD use our hint to decide in which memory to place the buffer, but in both cases, the drivers uses statistics and heuristics in order to fit the actual usage better. However, on NVIDIA with the Forceware 285 drivers, there are differences in the behavior of `glMapBuffer` and `glMapBufferRange`: `glMapBuffer` tries to guess the destination memory from the buffer-object usage, whereas `glMapBufferRange` always respects the hint and logs a debug message (Chapter 33) if our usage of the buffer object doesn't respect the hint. There are also differences in transfer rates between these functions; it seems that using

```
Buffer detailed info: Buffer object 1 (bound to GL_TEXTURE_BUFFER, usage hint is ↩
    GL_ENUM_88e0) has been mapped WRITE_ONLY in SYSTEM HEAP memory (fast).
Buffer detailed info: Buffer object 1 (bound to GL_TEXTURE_BUFFER, usage hint is ↩
    GL_ENUM_88e0) will use SYSTEM HEAP memory as the source for buffer object ↩
    operations.
Buffer detailed info: Buffer object 2 (bound to GL_TEXTURE_BUFFER, usage hint is ↩
    GL_ENUM_88e4) will use VIDEO memory as the source for buffer object operations.
Buffer info:
Total VBO memory usage in the system:
 memType: SYSHEAP, 22.50 Mb Allocated, numAllocations: 6.
 memType: VID, 64.00 Kb Allocated, numAllocations: 1.
 memType: DMA_CACHED, 0 bytes Allocated, numAllocations: 0.
 memType: MALLOC, 0 bytes Allocated, numAllocations: 0.
 memType: PAGED_AND_MAPPED, 40.14 Mb Allocated, numAllocations: 12.
 memType: PAGED, 142.41 Mb Allocated, numAllocations: 32.
```

Listing 28.1. Example output of `GL_ARB_debug_output` with Forceware 285.86 drivers.

Function	Usage hint	Destination memory	Transfer rate (GB/s)
`glBufferData` / `glBufferSubData`	`GL_STATIC_DRAW`	device	3.79
`glMapBuffer` / `glUnmapBuffer`	`GL_STREAM_DRAW`	pinned	n/a (pinned in CPU memory)
`glMapBuffer` / `glUnmapBuffer`	`GL_STATIC_DRAW`	device	5.73

Table 28.1. Buffer-transfer performance on an Intel Core i5 760 and an NVIDIA GeForce GTX 470 with PCI-e 2.0.

`glMapBufferRange` for all transfers ensures the best performance. An example application is available on the OpenGL Insights website, www.openglinsights.com, to measure the transfer rates and other behaviors of buffers objects; a few results are presented in Tables 28.1 and 28.2.

Pinned memory is standard CPU memory and there is no actual transfer to device memory: in this case, the device will use data directly from this memory location. The PCI-e bus can access data faster than the device is able to render it, so there is no performance penalty for doing this, but the driver can change that at any time and transfer the data to device memory.

Transfer	Source memory	Destination memory	Transfer rate (GB/s)
buffer to buffer	pinned	device	5.73
buffer to texture	pinned	device	5.66
buffer to buffer	device	device	9.00
buffer to texture	device	device	52.79

Table 28.2. Buffer copy and texture transfer performance on an Intel Core i5 760 and an NVIDIA GeForce GTX 470 with PCI-e 2.0 using `glCopyBufferSubData` and `glTexImage2D` with the `GL_RGBA8` format.

28.2.3 Implicit Synchronization

When an OpenGL call is done, it usually is not executed immediately. Instead, most commands are placed in the device command queue. Actual rendering may take place two frames later and sometimes more depending on the device's performance and on driver settings (triple buffering, max prerendered frames, multi-GPU configurations, etc.). This lag between the application and the drivers can be measured by the timing functions `glGetInteger64v(GL_TIMESTAMP,&time)` and `glQueryCounter(query,GL_TIMESTAMP)`, as explained in Chapter 34. Most of

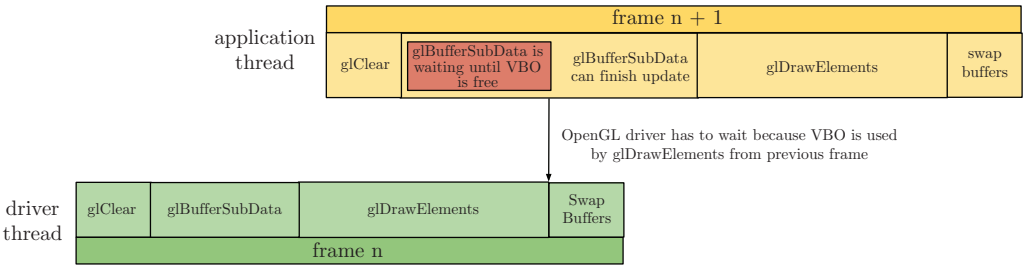

Figure 28.3. Implicit synchronization with `glSubBufferData`.

the time, this is actually the desired behavior because this lag helps drivers hiding latency in device communication and providing better overall performance.

However, when using `glBufferSubData` or `glMapBuffer[Range]`, nothing in the API itself prevents us from modifying data that are currently used by the device for rendering the previous frame, as shown in Figure 28.3. Drivers have to avoid this problem by blocking the function until the desired data are not used anymore: this is called an *implicit synchronization*. This can seriously damage performance or cause annoying jerks. A synchronization might block until all previous frames in the device command queue are finished, which could add several milliseconds to the performance time.

28.2.4 Synchronization Primitives

OpenGL offers its own synchronization primitives named *sync objects*, which work like fences inside the device command queue and are set to *signaled* when the device reaches their position. This is useful in a multithreaded environment, when other threads have to be informed about the completeness of computations or rendering and start downloading or uploading data.

The `glClientWaitSync` and `glWaitSync` functions will block until the specified fence is signaled, but these functions provide a timeout parameter which can be set to 0 if we only want to know whether an object has been signaled or not, instead of blocking it. More precisely, `glClientWaitSync` blocks the CPU until the specified sync object is signaled, while `glWaitSync` blocks the device.

28.3 Upload

Streaming is the process in which data are uploaded to the device frequently, e.g., every frame. Good examples of streaming include updating instance data when using

instancing or font rendering. Because these tasks are processed every frame, it is important to avoid implicit synchronizations. This can be done in multiple ways:

- a round-robin chain of buffer objects,
- buffer respecification or "orphaning" with `glBufferData` or `glMapBufferRange`,
- fully manual synchronization with `glMapBufferRange` and `glFenceSync` / `glClientWaitSync`.

28.3.1 Round-Robin Fashion (Multiple Buffer Objects)

The idea of the round-robin technique is to create several buffer objects and cycle through them. The application can update and upload buffer N while the device is rendering from buffer $N-1$, as shown on Figure 28.4. This method can also be used for download, and it is useful in a multithreaded application, too. See Sections 28.6 and 28.7 for details.

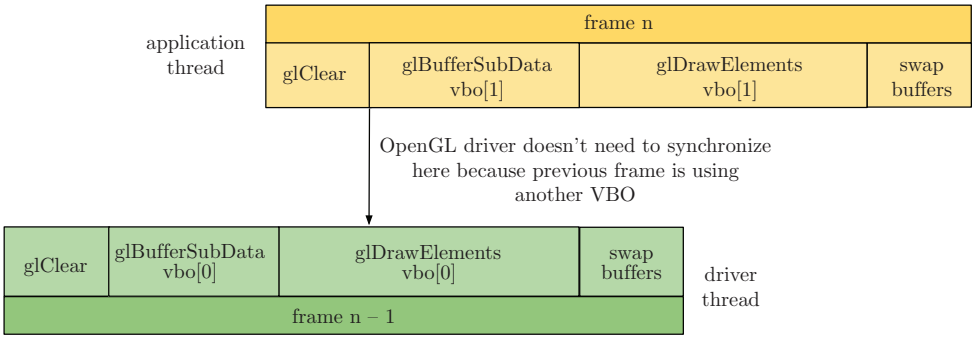

Figure 28.4. Avoiding implicit synchronizations with a round-robin chain.

28.3.2 Buffer Respecification (Orphaning)

Buffer respecification is similar to the round-robin technique, but it all happens inside the OpenGL driver. There are two ways to respecify a buffer. The most common one is to use an extra call to `glBufferData` with NULL as the data argument and the exact size and usage hint it had before, as shown in Listing 28.2. The driver will detach the physical memory block from the buffer object and allocate a new one. This operation is called *orphaning*. The old block will be returned to the heap once it is not used by any commands in the command queue. There is a high probability that

```
glBindBuffer(GL_ARRAY_BUFFER, my_buffer_object);

glBufferData(GL_ARRAY_BUFFER, data_size, NULL, GL_STREAM_DRAW);
glBufferData(GL_ARRAY_BUFFER, data_size, mydata_ptr, GL_STREAM_DRAW);
```

Listing 28.2. Buffer respecification or orphaning using `glBufferData`.

this block will be reused by the next `glBufferData` respecification call [OpenGL Wiki 09]. What's more, we don't have to guess the size of the round-robin chain, since it all happens inside the driver. This process is shown in Figure 28.5.

The behavior of `glBufferData` / `glBufferSubData` is actually very implementation dependent. For instance, it seems that AMD's driver can implicitly orphan the buffer. On NVIDIA, it is slightly more efficient to orphan manually and then upload with `glBufferSubData`, but doing so will ruin the performance on Intel. Listing 28.2 gives the more "coherent" performance across vendors. Lastly, with this technique, it's important that the size parameter of `glBufferData` is always the same to ensure the best performance.

The other way to respecify the buffer is to use the function `glMapBufferRange` with the `GL_MAP_INVALIDATE_BUFFER_BIT` or `GL_MAP_INVALIDATE_RANGE_BIT` flags. This will orphan the buffer and return a pointer to a freshly allocated memory block. See Listing 28.3 for details. We can't use `glMapBuffer`, since it doesn't have this option.

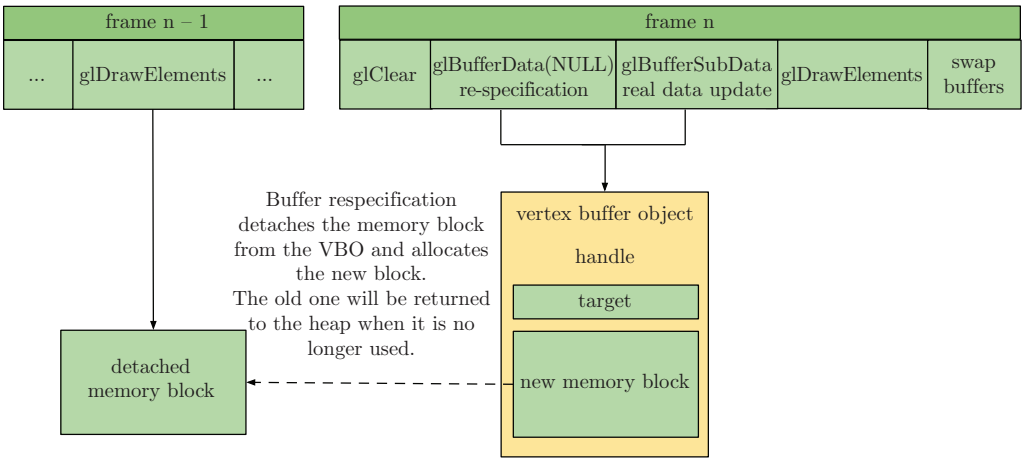

Figure 28.5. Avoiding implicit synchronizations with orphaning.

```
glBindBuffer(GL_ARRAY_BUFFER, my_buffer_object);
void *mydata_ptr = glMapBufferRange(
  GL_ARRAY_BUFFER, 0, data_size,
  GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT);

// Fill mydata_ptr with useful data

glUnmapBuffer(GL_ARRAY_BUFFER);
```

Listing 28.3. Buffer respecification or invalidation using `glMapBufferRange`.

However, we found that, at least on NVIDIA, `glBufferData` and `glMapBufferRange`, even with orphaning, cause expensive synchronizations if called concurrently with a rendering operation, even if the buffer is not used in this draw call or in any operation enqueued in the device command queue. This prevents the device from reaching 100 percent utilization. In any case, we recommend not using these techniques. On top of that, flags like `GL_MAP_INVALIDATE_BUFFER_BIT` or `GL_MAP_INVALIDATE_RANGE_BIT` involve the driver memory management, which can increase the call duration by more than ten times. The next section will present unsynchronized mapping, which can be used to solve all these synchronization problems.

28.3.3 Unsynchronized Buffers

The last method we will describe here gives us absolute control over the buffer-object data. We just have to tell the driver not to synchronize at all. This can be done by passing the `GL_MAP_UNSYNCHRONIZED_BIT` flag to `glMapBufferRange`. In this

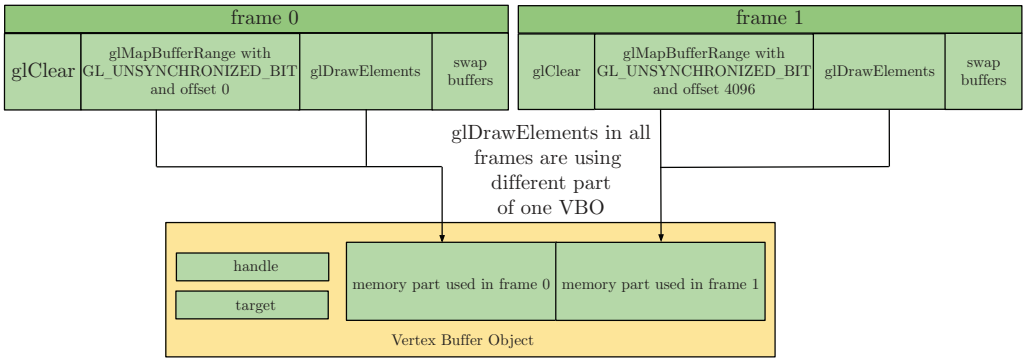

Figure 28.6. Possible usage of unsynchronized `glMapBufferRange`.

```
const int buffer_number = frame_number++ % 3;

// Wait until buffer is free to use, in most cases this should not wait
// because we are using three buffers in chain, glClientWaitSync
// function can be used for check if the TIMEOUT is zero
GLenum result = glClientWaitSync(fences[buffer_number], 0, TIMEOUT);
if (result == GL_TIMEOUT_EXPIRED || result == GL_WAIT_FAILED)
{
  // Something is wrong
}

glDeleteSync(fences[buffer_number]);
glBindBuffer(GL_ARRAY_BUFFER, buffers[buffer_number]);
void *ptr = glMapBufferRange(GL_ARRAY_BUFFER, offset, size, GL_MAP_WRITE_BIT | ↩
    GL_MAP_UNSYNCHRONIZED_BIT);

// Fill ptr with useful data
glUnmapBuffer(GL_ARRAY_BUFFER);

// Use buffer in draw operation
glDrawArray(...);

// Put fence into command queue
fences[buffer_number] = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);
```

Listing 28.4. Unsynchronized buffer mapping.

case, drivers just return a pointer to previously allocated pinned memory and do no synchronization and no memory re-allocation. This is the fastest way to deal with mapping (see Figure 28.6).

The drawback is that we really have to know what we're doing. No implicit sanity check or synchronization is performed, so if we upload data to a buffer that is currently being used for rendering, we can end up with an undefined behavior or application crash.

The easiest way to deal with unsynchronized mapping is to use multiple buffers like we did in the round-robin section and use `GL_MAP_UNSYNCHRONIZED_BIT` in the `glMapBufferRange` function, as shown in Listing 28.4. But we have to be sure that the buffer we are going to use is not used in a concurrent rendering operation. This can be achieved with the `glFencSync` and `glClientWaitSync` functions. In practice, a chain of three buffers is enough because the device usually doesn't lag more than two frames behind. At most, `glClientWaitSync` will synchronize us on the third buffer, but it is a desired behavior because it means that the device command queue is full and that we are GPU-bound.

28.3.4 AMD's pinned_memory Extension

Since Catalyst 11.5, AMD exposes the `AMD_pinned_memory` extension [Mayer 11, Boudier and Sellers 11], which allows us to use application-side memory allocated

```
#define GL_EXTERNAL_VIRTUAL_MEMORY_AMD 37216 // AMD_pinned_memory

char *_pinned_ptr = new char[buffer_size + 0x1000];
char *_pinned_ptr_aligned = reinterpret_cast<char *>(unsigned(_pinned_ptr + 0xfff) &↩
     (~0xfff));

glBindBuffer(GL_EXTERNAL_VIRTUAL_MEMORY_AMD, buffer);
glBufferData(GL_EXTERNAL_VIRTUAL_MEMORY_AMD, buffer_size, _pinned_ptr_aligned, ↩
    GL_STREAM_READ);
glBindBuffer(GL_EXTERNAL_VIRTUAL_MEMORY_AMD, 0);
```

Listing 28.5. Example usage of `AMD_pinned_memory`.

with `new` or `malloc` as buffer-object storage. This memory block has to be aligned to the page size. There are a few advantages when using this extension:

- Memory is accessible without OpenGL mapping functions, which means there is no OpenGL call overhead. This is very useful in worker threads for geometry and texture loading.
- Drivers' memory management is skipped because we are responsible for memory allocation.
- There is no internal driver synchronization involved in the process. It is similar to the `GL_MAP_UNSYNCHRONIZED_BIT` flag in `glMapBufferRange`, as explained in the previous section, but it means that we have to be careful which buffer or buffer portion we are going to modify; otherwise, the result might be undefined or our application terminated.

Pinned memory is the best choice for data streaming and downloading, but it is available only on AMD devices and needs explicit synchronization checks to be sure that the buffer is not used in a concurrent rendering operation. Listing 28.5 shows how to use this extension.

28.4 Download

The introduction of the PCI-e bus gave us enough bandwidth to use data download in real-life scenarios. Depending on the PCI-e version, the device's upload and download performance is approximately 1.5–6 GB/s. Today, many algorithms or situations require downloading data from the device:

- procedural terrain generation (collision, geometry, bounding boxes, etc.);
- video recording, as discussed in Chapter 31;
- page resolver pass in virtual texturing;

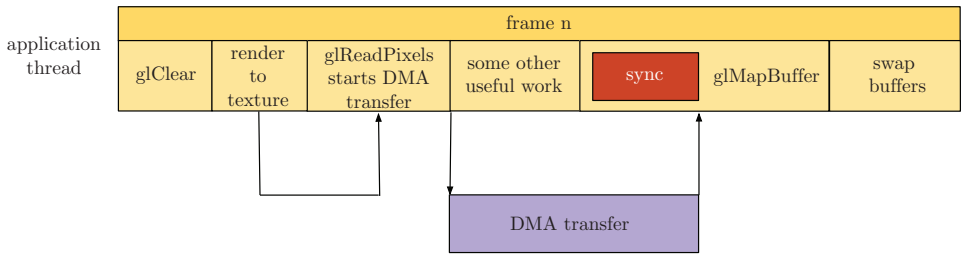

Figure 28.7. Asynchronous DMA transfer in download.

- physics simulation;
- image histogram.

The asynchronous nature of OpenGL drivers brings some complications to the download process, and the specification is not very helpful regarding how to do it fast and without implicit synchronization. OpenGL currently offers a few ways to download data to the main memory. Most of the time, we want to download textures because rasterization is the most efficient way to generate data on the GPU, at least in OpenGL. This includes most of the use-cases above.

In this case, we have to use `glReadPixels` and bind a buffer object to the `GL_PIXEL_PACK_BUFFER` target. This function will start an asynchronous transfer from the texture memory to the buffer memory. In this case, it is important to specify a `GL_*_READ` usage hint for the buffer object because the OpenGL driver will copy the data to the driver memory, which can be accessed from the application. Again, this is only asynchronous for the CPU: the device has to wait for the current render to complete and process the transfer. Finally, `glMapBuffer` returns a pointer to the downloaded data. This process is presented in Figure 28.7.

In this simple scenario, the application thread is blocked because the device command queue is always lagging behind, and we are trying to download data which aren't ready yet. Three options are available to avoid this waiting:

- do some CPU intensive work after the call to `glReadPixels`;
- call `glMapBuffer` on the buffer object from the previous frame or two frames behind;
- use fences and call `glMapBuffer` when a sync object is signaled.

The first solution is not very practical in real applications because it doesn't guarantee that we will not be waiting at the end and makes it harder to write efficient code. The second solution is much better, and in most cases, there will be no wait

```
if (rb_tail != rb_head)
{
  const int tmp_tail = (rb_tail + 1) & RB_BUFFERS_MASK;
  GLenum res = glClientWaitSync(fences[tmp_tail], 0, 0);
  if (res == GL_ALREADY_SIGNALED || res == GL_CONDITION_SATISFIED)
  {
    rb_tail = tmp_tail;
    glDeleteSync(sc->_fence);
    glBindBuffer(GL_PIXEL_PACK_BUFFER, buffers[rb_tail]);
    glMapBuffer(GL_PIXEL_PACK_BUFFER, GL_READ_ONLY);
    // Process data
    glUnmapBuffer(GL_PIXEL_PACK_BUFFER);
  }
}
const int tmp_head = (rb_head + 1) & RB_BUFFERS_MASK;
if (tmp_head != rb_tail)
{
  glReadBuffer(GL_BACK);
  glBindBuffer(GL_PIXEL_PACK_BUFFER, buffers[rb_head]);
  glReadPixels(0, 0, width, height, GL_BGRA, GL_UNSIGNED_BYTE, (void*)offset);
}
else
{
    // We are too fast
}
```

Listing 28.6. Asynchronous pixel data transfer.

because the data is already transferred. This solution needs multiple buffer objects as presented in the round-robin section. The last solution is the best way to avoid implicit synchronization because it gives exact information on the completeness of the transfer; we still have to deal with the fact that the data will only be ready later, but as developers, we have more control over the status of the transfer thanks to the fences. The basic steps are provided in Listing 28.6.

However, on AMD hardware, `glUnmapBuffer` will be synchronous in this special case. If we really need an asynchronous behavior, we have to use the `AMD_pinned_memory` extension.

On the other hand, we have found that on NVIDIA, it is better to use another intermediate buffer with the `GL_STREAM_COPY` usage hint, which causes the buffer to be allocated in device memory. We use `glReadPixels` on this buffer and finally use `glCopyBufferSubData` to copy the data into the final buffer in CPU memory. This process is almost two times faster than a direct way. This copy function is described in the next section.

28.5 Copy

A widespread extension is `ARB_copy_buffer` [NVIDIA 09], which makes it possible to copy data between buffer objects. In particular, if both buffers live in device

```
glBindBuffer(GL_COPY_READ_BUFFER, source_buffer);
glBindBuffer(GL_COPY_WRITE_BUFFER, dest_buffer);
glCopyBufferSubData(GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, source_offset, ↩
    write_offset, data_size);
```

Listing 28.7. Copying one buffer into another using `ARB_copy_buffer`.

memory, this is the only way to copy data between buffers on the GPU side without CPU intervention (see Listing 28.7).

As we pointed out at the end of previous section, on NVIDIA GeForce devices, copy is useful for downloading data. Using an intermediate buffer in device memory and reading the copy back to the CPU is actually faster than a direct transfer: 3GB/s instead of 1.5GB/s. This is a limitation of the hardware that is not present on the NVIDIA Quadro product line. On AMD, with Catalyst 11.12 drivers, this function is extremely unoptimized, and in most cases, causes expensive synchronizations.

28.6 Multithreading and Shared Contexts

In this section, we will describe how to stream data from another thread. In the last few years, single-core performance hasn't been increasing as fast as the number of cores in the CPU. As such, it is important to know how OpenGL behaves in a multithreaded environment. Most importantly, we will focus on usability and performance considerations. Since accessing the OpenGL API from multiple threads is not very well known, we need to introduce shared contexts first.

28.6.1 Introduction to Multithreaded OpenGL

OpenGL can actually be used from multiple threads since Version 1.1, but some care must be taken at application initialization. More precisely, each additional thread that needs to call OpenGL functions must create its own context and explicitly connect that context to the first context in order to share OpenGL objects. Not doing so will result in crashes when trying to execute data transfers or draw calls. Implementation details vary from platform to platform. The recommended process on Windows is depicted in Figure 28.8, using the `WGL_ARB_create_context` extensions available in OpenGL 3.2 [ARB 09b]. A similar extension, `GLX_ARB_create_context`, is available for Linux [ARB 09c]. Implementation details for Linux, Mac, and Windows can be found in [Supnik 08].

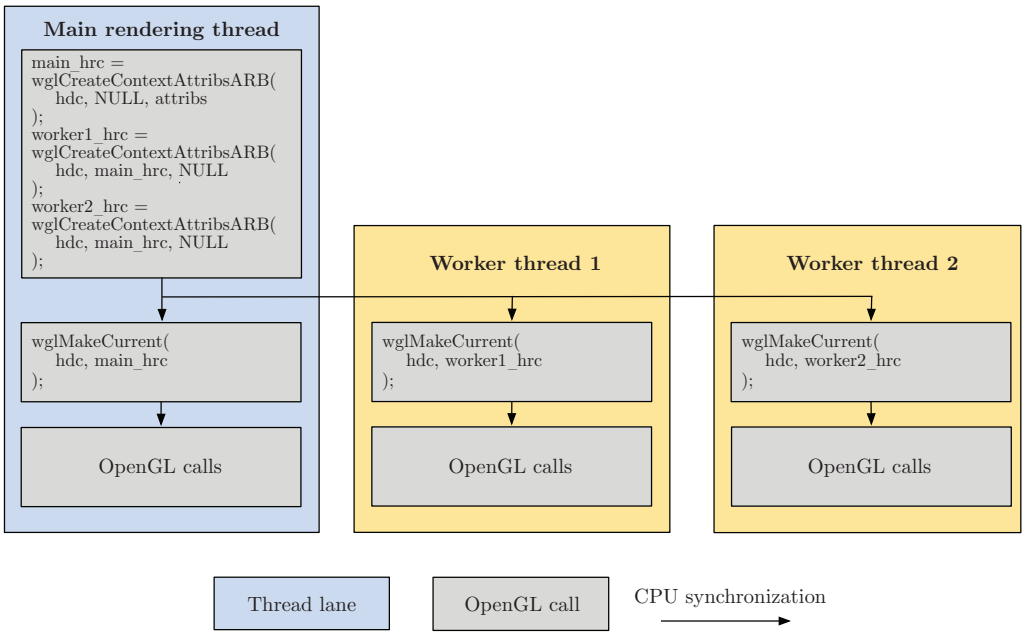

Figure 28.8. Shared-context creation on Windows.

28.6.2 Synchronization Issues

In a single-threaded scenario, it is perfectly valid to respecify a buffer while it is currently in use: the driver will put `glBufferData` in the command queue, and upon processing, wait until draw calls relying on the same buffer are finished.

When using shared contexts, however, the driver will create one command queue for each thread, and no such implicit synchronization will take place. A thread can thus start a DMA transfer in a memory block that is currently used in a draw call by the device. This usually results in partially updated meshes or instance data.

The solution is to use the above-mentioned techniques, which also work with shared contexts: multibuffering the data or using fences.

28.6.3 Performance Hit due to Internal Synchronization

Shared contexts have one more disadvantage: as we will see in the benchmarks below, they introduce a performance hit each frame.

In Figure 28.9, we show the profiling results of a sample application running in Parallel Nsight on a GeForce GTX 470 with the 280.26 Forceware drivers. The first timeline uses a single thread to upload and render a 3D model; the second

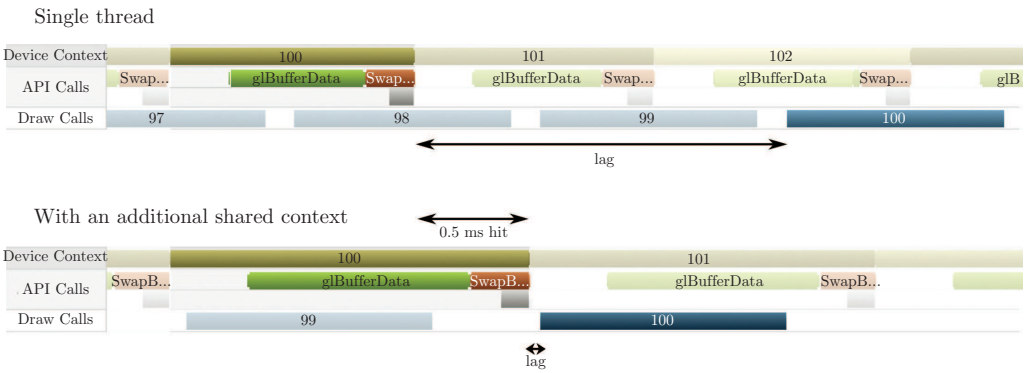

Figure 28.9. Performance hit due to shared contexts.

timeline does exactly the same thing but with an extra shared context in an idle thread. This simple change adds 0.5 ms each frame, probably because of additional synchronizations in the driver. We also notice that the device only lags one frame behind instead of two.

At least on NVIDIA, this penalty usually varies between 0.1 and 0.5 ms; this mostly depends on the CPU performance. Remarkably, it is quite constant with respect to the number of threads with shared contexts. On NVIDIA Quadro hardware, this penalty is usually lower because some hardware cost optimizations of the GeForce product line are not present.

28.6.4 Final Words on Shared Context

Our advice is to use standard working threads when possible. Since all the functionality that shared contexts offers can be obtained without them, the following do not usually cause a problem:

- If we want to speed up the rendering loop by offloading some CPU-heavy task in another thread, this can be done without shared contexts; see the next section for details.

- If we need to know if a data transfer is finished in a single-threaded environment, we can use fences, as defined in the `GL_ARB_sync` extension; see Listing 28.8 for details.

We have to point out that shared contexts won't make transfers and rendering parallel, at least in NVIDIA Forceware 285 and AMD Catalyst 11.12, so there is usually minimal performance advantage for using them. See Chapter 29 for more details on using fences with shader contexts and multiple threads.

```
glUnmapBuffer(...);

GLsync fence = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);

// Other operations

int res = glClientWaitSync(fence, 0, TIMEOUT);
if (res == GL_ALREADY_SIGNALED || res == GL_CONDITION_SATISFIED)
{
  glDeleteSync(fence);
  // Transfer finished
}
```

Listing 28.8. Waiting for a transfer completion with `GL_ARB_sync`.

28.7 Usage Scenario

In this last section, we will now present a scenario in which we will stream some scene object data to the device. Our scene is represented by 32,768 objects representing a building structure. Each object is generated in a GPU shader, and the only input is the transformation matrix, which means 2MB of data per frame for the whole scene. For rendering, we use an instanced draw call that minimizes the CPU intervention.

This scenario is implemented in three different methods: a single-threaded version, a multithreaded version without shared contexts, and a multithreaded version with shared contexts. All the source code is available for reference on the OpenGL Insights website, www.openglinsights.com.

In practice, these methods can be used to upload, for instance, frustum culling information, but since we want to measure the transfer performance, no computation is actually done: the work consists simply in filling the buffer as fast as possible.

28.7.1 Method 1: Single Thread

In this first method, everything is done in the rendering thread: buffer streaming and rendering. The main advantage of this implementation is its simplicity. In particular, there is no need for mutexes or other synchronization primitives.

The buffer is streamed using `glMapBufferRange` with the `GL_MAP_WRITE_BIT` and `GL_MAP_UNSYNCHRONIZED_BIT` flags. This enables us to write transformation matrices directly into the pinned memory region, which will be used directly by the device, saving an extra `memcpy` and a synchronization compared to the other methods.

In addition, `glMapBufferRange` can, as the name suggests, map only a subset of the buffer, which is useful if we don't want to modify the entire buffer or if the size of the buffer changes from frame to frame: as we said earlier, we can allocate a big buffer and only use a variable portion of it. The performance of this single-threaded implementation method is shown in Table 28.3.

Architecture	Rendering time (ms/frame)
Intel Core i5, NVIDIA GeForce GTX 470	2.8
Intel Core 2 Duo Q6600, AMD HD 6850	3.6
Intel Core i7, Intel GMA 3000	16.1

Table 28.3. Rendering performance for Method 1.

28.7.2 Method 2: Two Threads and One OpenGL Context

The second method uses another thread to copy the scene data to a mapped buffer. There are a number of reasons why doing so is a good idea:

- The rendering thread doesn't stop sending OpenGL commands and is able to keep the device busy all the time.
- Dividing the processing between two CPU cores can shorten the frame time.
- OpenGL draw calls are expensive; they are usually more time consuming than simply appending a command in the device command queue. In particular, if the internal state has changed since the last draw call, for instance, due to a call to `glEnable`, a long state-validation step occurs [2]. By separating our computations from the driver's thread, we can take advantage of multicore architectures.

In this method (see Figure 28.10), we will use two threads: the application thread and the renderer thread. The application thread is responsible for

- handling inputs,
- copying scene instance data into the mapped buffer,
- preparing the primitives for font rendering.

The renderer thread is responsible for

- calling `glUnmapBuffer` on the mapped buffers that were filled in the application thread,
- setting shaders and uniforms,
- drawing batches.

We use a queue of frame-context objects that helps us avoid unnecessary synchronizations between threads. The frame-context objects hold all data required for a frame, such as the camera matrix, pointers to memory-mapped buffers, etc. This design is very similar to the round-robin fashion because it uses multiple unsynchronized buffers. It is also used with success in the Outerra Engine [Kemen and

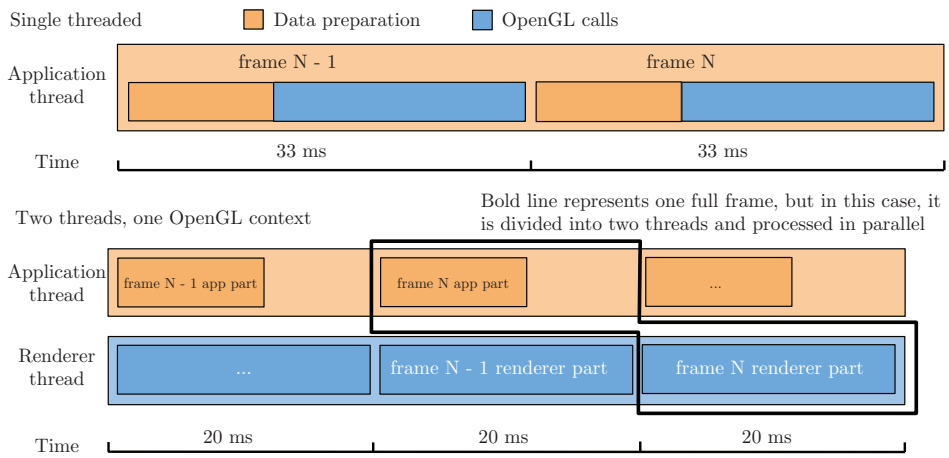

Figure 28.10. Method 2: improving the frame rate with an external renderer thread.

Hrabcak 11]. The performance results are shown in Table 28.4. For simplicity, we used only two threads here, but we can of course add more, depending on the tasks and the dependencies in the computations.

Architecture	Performance (ms/frame)	improvement vs. Method 1
Intel Core i5, NVIDIA GeForce GTX 470	2.0	×1.4
Intel Core 2 Duo Q6600, AMD HD 6850	3.2	×1.25
Intel Core i7, Intel GMA 3000	15.4	×1.05

Table 28.4. Rendering performance for Method 2.

28.7.3 Method 3: Two Threads and Two OpenGL Shared Contexts

In this last method, the scene-data copy is done in an OpenGL-capable thread. We thus have two threads: the main rendering thread and the additional rendering thread. The main rendering thread is responsible for the following tasks:

- handling inputs,
- calling `glMapBufferRange` and `glUnmapBuffer` on buffers,
- copying scene instance data into the mapped buffer,
- preparing primitives for font rendering.

Architecture	Performance		
	(ms/frame)	improvement vs. Method 1	hit due to shared contexts (ms/frame)
Intel Core i5, NVIDIA GeForce GTX 470	2.1	×1.33	+0.1
Intel Core 2 Duo Q6600, AMD HD 6850	7.5	×0.48	+4.3
Intel Core i7, Intel GMA 3000	15.3	×1.05	-0.1

Table 28.5. Rendering performance for Method 3.

The renderer thread is responsible for

- setting shaders and uniforms,
- drawing batches.

In this method, buffers are updated in the main thread. This includes calling `glMapBufferRange` and `glUnmapBuffer` because the threads are sharing the OpenGL rendering context. We get most of the benefits from the second method (two threads and one OpenGL context) as compared to the single-threaded version: faster rendering loop, parallelization of some OpenGL calls, and better overall performance than Method 1, as shown in Table 28.5. However, as mentioned earlier, there is a synchronization overhead in the driver, which makes this version slower than the previous one. This overhead is much smaller on professional cards like NVIDIA Quadro, on which such multithreading is very common, but is still present.

The performance drop of AMD in this case should not be taken too seriously, because unsynchronized buffers are not ideal with shared contexts on this platform. Other methods exhibit a more reasonable 1.1 times performance improvement over the first solution, as shown in the next section.

28.7.4 Performance Comparisons

Table 28.6 shows the complete performance comparisons of our scenarios with several upload policies on various hardware configurations. All tests use several buffers in a round-robin fashion; the differences lie in the way the data is given to OpenGL:

- InvalidateBuffer. The buffer is mapped with `glMapBufferRange` using the `GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT` flags, and unmapped normally.
- FlushExplicit. The buffer is mapped with `glMapBufferRange` using the `GL_MAP_WRITE_BIT | GL_MAP_FLUSH_EXPLICIT_BIT` flags, flushed, and unmapped. The unmapping must be done because it is not safe to keep the buffer mapped permanently, except when using `AMD_pinned_memory`.

CPU	Intel Q6600		Intel i7 2630QM		Intel i5 760	
GPU	AMD HD 6850	NV GTX 460	Intel HD 3000	NV GT 525M	AMD HD 6570	NV GTX 470
Scenario 1						
InvalidateBuffer	3.6	5.0	16.1	12.6	12.0	3.5
FlushExplicit	4.9	4.9	16.1	12.5	18.4	3.5
Unsynchronized	3.6	3.7	16.1	11.2	9.0	2.8
BufferData	5.2	4.3	16.2	11.7	6.7	3.1
BufferSubData	4.4	4.3	17.3	11.6	9.5	3.1
Write	8.8	4.9	16.1	12.4	19.5	3.5
AMD Pinned	3.7	n/a	n/a	n/a	8.6	n/a
Scenario 2						
InvalidateBuffer	5.5	3.2	15.3	10.3	9.5	2.1
FlushExplicit	7.2	3.1	15.3	10.3	16.3	2.1
Unsynchronized	**3.2**	**2.9**	15.4	**9.9**	8.0	**2.0**
BufferData	4.6	3.5	15.2	10.4	**5.5**	2.3
BufferSubData	4.0	3.5	**15.1**	10.5	8.3	2.3
Write	7.4	3.1	15.3	10.3	17.0	2.1
AMD Pinned	**3.2**	n/a	n/a	n/a	8.1	n/a
Scenario 3						
InvalidateBuffer	5.3	3.8	15.2	10.6	9.4	2.4
FlushExplicit	7.4	3.7	15.2	10.6	17.1	2.3
Unsynchronized	7.5	3.2	15.3	10.2	17.9	2.1
BufferData	broken	4.5	15.3	11.0	broken	2.5
BufferSubData	4.5	3.9	**15.1**	11.0	8.6	2.5
Write	7.5	3.5	15.2	10.5	17.9	2.3
AMD Pinned	**3.2**	n/a	n/a	n/a	8.0	n/a

Table 28.6. Our results in all configurations. All values are expressed in ms/frame (smaller is better).

- **Unsynchronized.** The buffer is mapped with `glMapBufferRange` using the `GL_MAP_WRITE_BIT | GL_MAP_UNSYNCHRONIZED_BIT` flags and unmapped normally.
- **BufferData.** The buffer is orphaned using `glBufferData(NULL)`, and updated with `glBufferSubData`.
- **BufferSubData.** The buffer is not orphaned and is simply updated with `glBufferSubData`.
- **Write.** The buffer is mapped with `glMapBufferRange` using only the `GL_MAP_WRITE_BIT` flag.

Tests on the Intel GMA 3000 were performed with a smaller scene because it wasn't able to render the larger scene correctly.

The Intel GMA 3000 has almost the same performance in all cases. Since there is only standard RAM, there is no transfer and probably fewer possible variations for accessing the memory. Intel also seems to have a decent implementation of shared contexts with a minimal overhead.

NVIDIA and AMD, however, both have worse performance when using shared contexts. As said earlier, the synchronization cost is relatively constant but not negligible.

For all vendors, using a simple worker thread gets us the best performance, provided that synchronizations are done carefully. While the unsynchronized version is generally the fastest, we notice some exceptions: in particular, `glBufferData` can be very fast on AMD when the CPU can fill the buffer fast enough.

28.8 Conclusion

In this chapter, we investigated how to get the most out of CPU-device transfers. We explained many available techniques to stream data between the CPU and the device and provided three sample implementations with performance comparisons.

In the general case, we recommend using a standard worker thread and multiple buffers with the `GL_MAP_UNSYCHRONIZED_BIT` flag. This might not be possible because of dependencies in the data, but this will usually be a simple yet effective way to improve the performance of an existing application.

It is still possible that such an application isn't well suited to parallelization. For instance, if it is rendering-intensive and doesn't use much CPU, nothing will be gained from multithreading it. Even there, better performance can be achieved by simply avoiding uploads and downloads of currently used data. In any case, we should always upload our data as soon as possible and wait as long as possible before using new data in order to let the transfer complete.

We believe that OpenGL would benefit from a more precise specification in buffer objects, like explicit pinned memory allocation, strict memory destination parameters instead of hints, or a replacement of shared contexts by streams, similar to what CUDA and Direct3D 11 provide. We also hope that future drivers provide real GPU-asynchronous transfers for all buffer targets and textures, even on low-cost gaming hardware, since it would greatly improve the performance of many real-world scenarios.

Finally, as with any performance-critical piece of software, it is very important to benchmark the actual usage on our target hardware, for instance, using NVIDIA Nsight because it is easy to leave the "fast path."

Bibliography

[ARB 08] OpenGL ARB. "OpenGL EXT_framebuffer_object Specification." www.opengl.org/registry/specs/EXT/framebuffer_object.txt, 2008.

[ARB 09a] OpenGL ARB. "OpenGL ARB_texture_buffer_object Specification." www.opengl.org/registry/specs/EXT/texture_buffer_object.txt, 2009.

[ARB 09b] OpenGL ARB. "OpenGL GLX_create_context Specification." www.opengl.org/registry/specs/ARB/glx_create_context.txt, 2009.

[ARB 09c] OpenGL ARB. "OpenGL WGL_create_context Specification." www.opengl.org/registry/specs/ARB/wgl_create_context.txt, 2009.

[Boudier and Sellers 11] Pierre Boudier and Graham Sellers. "Memory System on Fusion APUs: The Benefit of Zero Copy." developer.amd.com/afds/assets/presentations/1004_final.pdf, 2011.

[Intel 08] Intel. "Intel X58 Express Chipset." http://www.intel.com/Assets/PDF/prodbrief/x58-product-brief.pdf, 2008.

[Kemen and Hrabcak 11] Brano Kemen and Ladislav Hrabcak. "Outerra." outerra.com, 2011.

[Kemen 10] Brano Kemen. "Outerra Video Recording." www.outerra.com/video, 2010.

[Mayer 11] Christopher Mayer. "Streaming Video Data into 3D Applications." developer.amd.com/afds/assets/presentations/2116_final.pdf, 2011.

[NVIDIA 09] NVIDIA. "OpenGL ARB_copy_buffer Specification." http://www.opengl.org/registry/specs/ARB/copy_buffer.txt, 2009.

[OpenGL Wiki 09] OpenGL Wiki. "OpenGL Wiki Buffer Object Streaming." www.opengl.org/wiki/Buffer_Object_Streaming, 2009.

[Supnik 08] Benjamin Supnik. "Creating OpenGL Objects in a Second Thread—Mac, Linux, Windows." http://hacksoflife.blogspot.com/2008/02/creating-opengl-objects-in-second.html, 2008.

[Venkataraman 10] Shalini Venkataraman. "NVIDIA Quadro Dual Copy Engines." www.nvidia.com/docs/IO/40049/Dual_copy_engines.pdf, 2010.

[Williams and Hart 11] Ian Williams and Evan Hart. "Efficient Rendering of Geometric Data Using OpenGL VBOs in SPECviewperf." www.spec.org/gwpg/gpc.static/vbo_whitepaper.html, 2011.