Direct Rendering Infrastructure: Architecture

José Manuel Rios Fonseca

13th June 2005

Contents

List of Figures

List of Tables

Acknowledgments

This work was initially done for the subject of Architecture of Software and Systems, part of the Informatics Engineering Master course at the Faculty of Engineering of the university of Porto.

Trademarks

Chapter 1

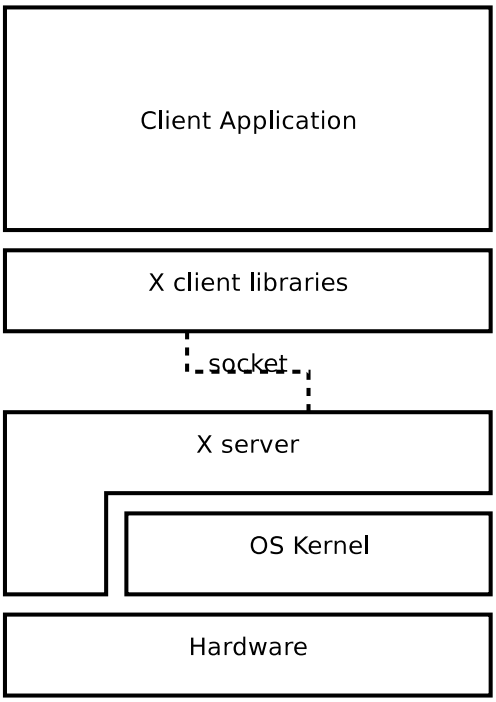
Introduction

About DRI. The Direct Rendering Infrastructure (DRI) is a framework for allowing direct access to graphics hardware under the X Window System in a safe and efficient manner, in order to create fast OpenGL implementations.

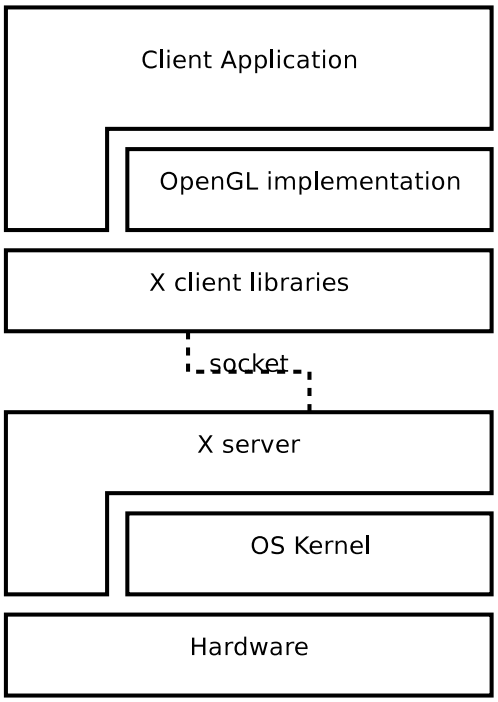
In the X Window System rendering is traditionally done via the X Server – a separate process with sole access to the graphics hardware. The application is linked with X client libraries which communicates to the X via a socket (Fig. 1.1a). This layered architecture allows for flexibility and encapsulation. The X application and X server not only run in a different process, but can be on two machine remotely connected by a network – X applications (or nowadays, X graphical toolkits) are designed to avoid round-trips to the X server to the maximum extent possible in order to keep user interactivity snappy in those scenarios. The X protocol is easily extensible, so 3D primitives can be encoded by the OpenGL API, transmitted to the X server by the X client libraries, and then rendered to the graphics hardware (Fig. 1.1b). Nevertheless the average bandwidth requirement of 3D rendering is much larger than 2D. The need of passing large amounts vertex and texture data to the hardware severely impacts the latency and interactivity of the 3D application. This is were DRI steps in the picture. DRI main goal is to provide a high-bandwidth low-latency communication channel between the application and the graphics hardware (Fig. 1.1c), allowing the OpenGL implementation to drive the hardware to its full potential.

About this document. This document aims to describe and analyze the DRI architecture at multiple levels of detail.

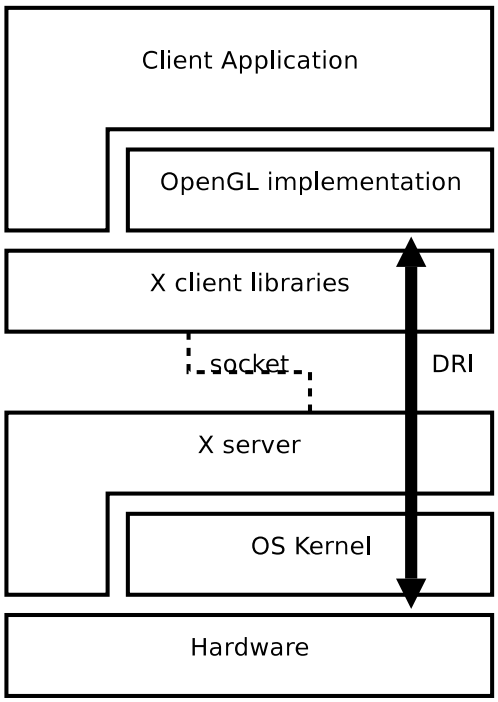
Chapter 2 describes the design requirements, the overall architecture and the roles of the components. In chapter 3 it is described the use, context and implementation of some relevant software patterns. Finally some ideas for future enhancements to the architecture are given in chapter 4.



1.1a: 2D rendering



1.1b: Indirect 3D rendering



1.1c: Direct 3D rendering

Figure 1.1: X Window System rendering overview

Chapter 2

Architecture

2.1 Design goals

Table 2.1 describes the main goals, their rationale and the implications in DRI design.

Table 2.1: Design goals

Goal	Rationale	Implications
• Allow high performance utilization of graphics hardware.		• Maximize the hardware communication bandwidth.
• Support of a variety of different graphics hardware designs.	• Functionality present in graphics hardware can vary from the simple triangle rasterization present in first-generation devices, to the interpretation of vertex and fragment shader programs present in the new generation devices. The hardware programming and communication varies not only across hardware generations, but also across the different hardware vendors, from the straightforward three vertex PIO triangle setup, to intricate nested[1] DMA transfers.	• DRI must not assume a particular design or family of designs. It needs to smoothly provide software fallbacks for functionality not present in the graphics hardware, and flexible DMA buffering.

[1] Whereby a DMA transfer spawns child DMA transfers

Table 2.1: (continuation)

Goal	Rationale	Implications
• Support of multiple, simultaneous rendering by many client programs.	• A OpenGL program can have multiple rendering contexts (rendering windows).	• The simplifications of assuming a single full-screen window are not possible.
• Security to prevent malicious misuse of the system.	• Current graphics hardware bus mastering abilities often allow to read/write to anywhere in the system memory.	• When present, such functionality must not be exposed to avoid root exploits.
• Reliability to prevent hardware lockups or system deadlocks.		
• Portability to allow implementations on other operating systems and system architectures.		• OS-specific code should be minimized by reducing it to a portability layer.
• Compliance with the OpenGL and GLX specifications.		
• Integration with the XFree86 project.		
• Open-source implementation.		

2.2 Main components

DRI main components are dictated by the different spaces were control and data flow. These spaces are the client application, the X server, and the OS kernel. They have distinct scope and characteristics, which are summarized in Table 2.2. The possible communication channels between the spaces are illustrated in 2.1[4].

The main DRI components work as plug-ins in each of these spaces. For every space there is a part of the plugin which is shared for all DRI drivers (frozen spot) and a part which varies for each driver (hot spot). These are listed in table 2.3. The boundary between the hot and frozen spots is not always clearly cut. Although some in cases the separation is done by using different binaries binaries, in other separation is visible only in the source code.

Table 2.2: Spaces features

Space	Forces
Client	
	+ where vertex and texture data is first generated and is more easily accessible
	+ MMIO and DMA allowed
	- multiple and concurrent instances
	- virtual address space accessible to malicious clients
	- no root privileges
	- PIO not allowed
	- requires polling for DMA completion or VSYNC events
X Server	
	+ where graphics hardware is detected and initialized
	+ root privileges
	+ PIO, MMIO and DMA allowed
	+ highly portable code
	- imposes an indirection layer for vertex and texture data flow
	- requires polling for DMA completion or VSYNC events
OS Kernel	
	+ physical memory access
	+ PIO, MMIO and DMA allowed
	+ allows the use IRQ handlers for DMA completion or VSYNC events
	+ establishing of memory maps
	- very low portability across different OS's
	- use of floating point operations highly unrecommended

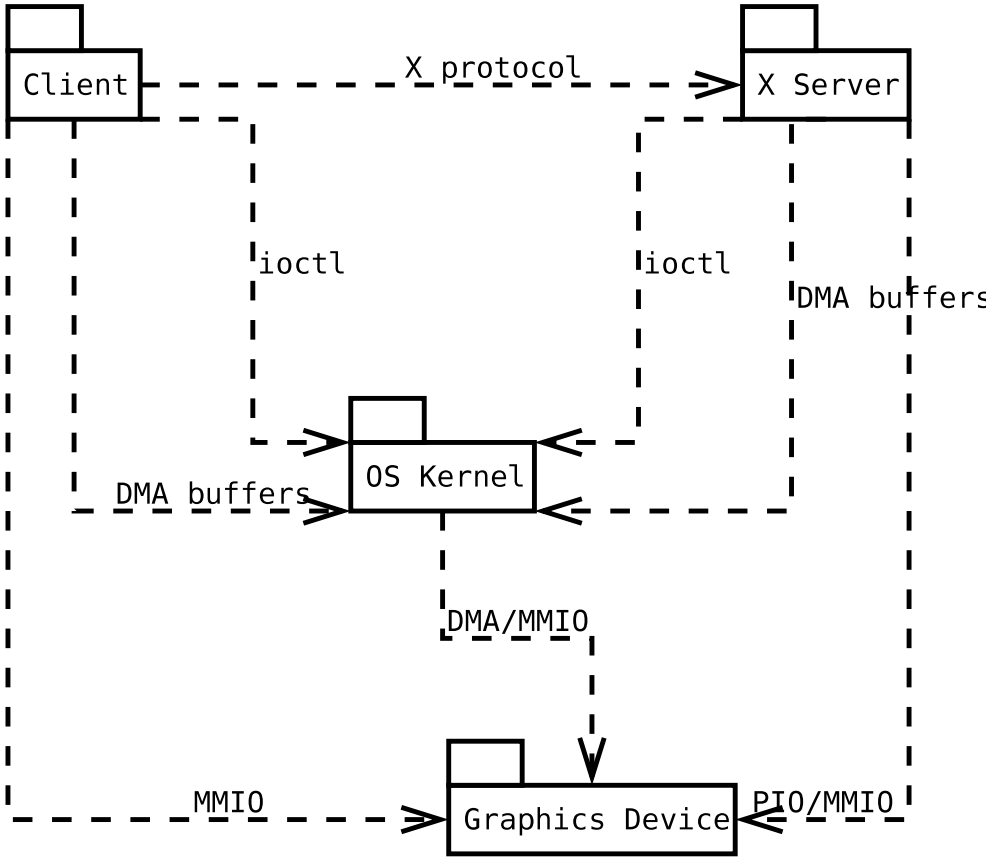


Figure 2.1: Spaces communication pathways

Table 2.3: Infrastructure extensibility spots

Space	Frozen spots	Hot spots
Client	libGL	libGL driver (also known as DRI 3D driver)
X Server	DRI X extension	DDX driver
OS Kernel	Direct Rendering Manager (DRM) core module	DRM module

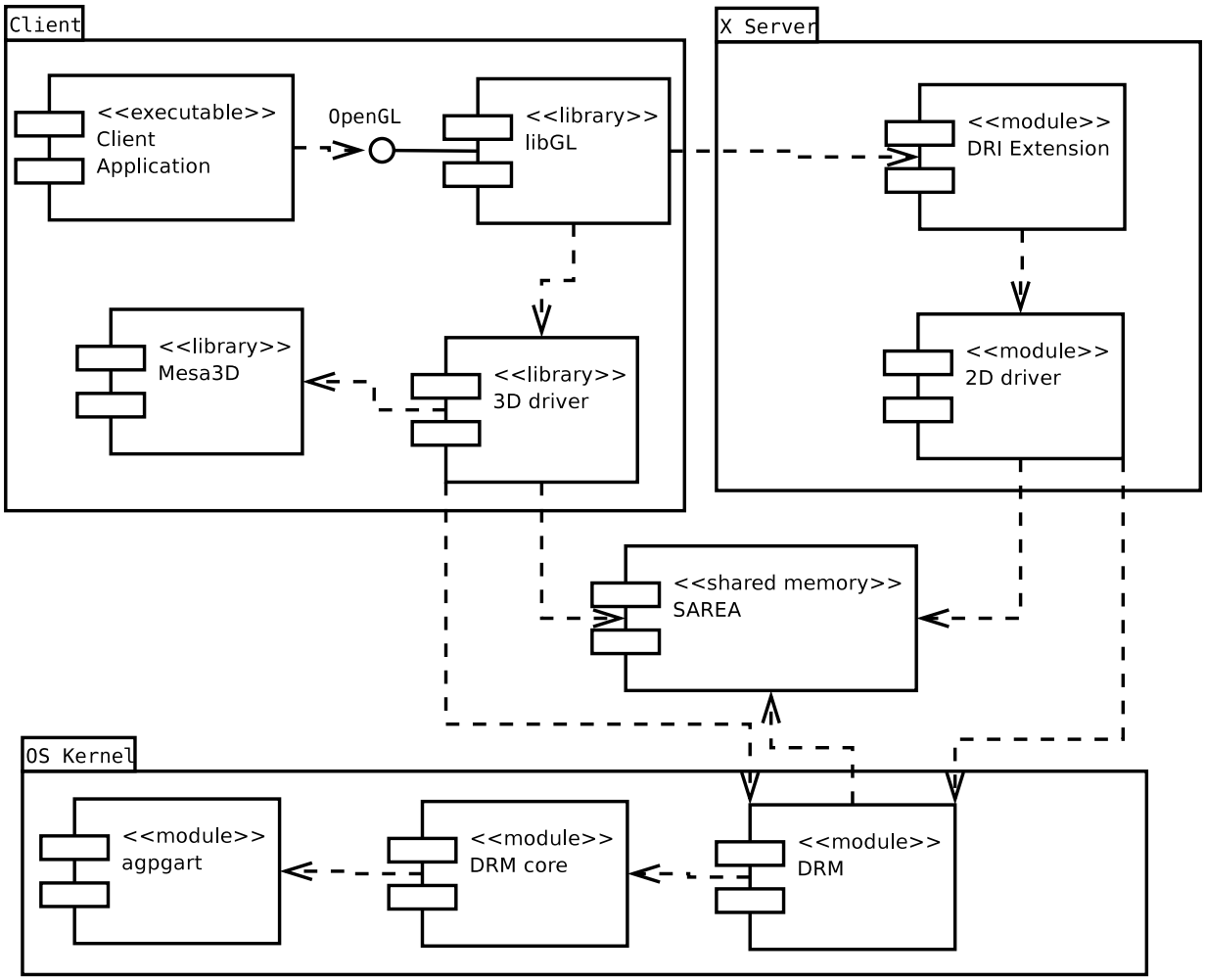


Figure 2.2: Main components

Table 2.4: Responsibilities

Component	Responsibilities
libGL	• Present a OpenGL compatible API to the client application • Implement the GLX API (the glue between OpenGL and X) • Find and load the appropriate 3D driver • Dispatch the received OpenGL API calls to the 3D driver, or fallback to the X server if no 3D driver was found
3D driver	• Implement the OpenGL API • Transform the received vertex and texture data into the hardware native format • Keep a backup of the graphics hardware state which is relevant to its drawing context • If DMA is supported by the hardware, fill in DMA buffers with the vertex and texture data and signal the DRM module to dispatch it to the hardware • Provide software fallbacks for all operations not supported in hardware
DRI extension	• Context/window setup

Table 2.4: (continuation)

2D driver	
	• Detect and initialize hardware
	• Reserve on-board memory for 3D operations
	• Synchronize 2D operations with 3D ones
	• Identify which 3D driver and DRM module to load
	• Communicate the current *cliprect* list
	• Authorize client access to the DRM module
DRM core module	
	• Thin OS kernel abstraction layer for portability
DRM module	
	• Graphical hardware lock
	• Allocate a pool of DMA buffers (in the AGP aperture if possible)
	• Memory map the DMA buffers to client virtual address space
	• Dispatch the DMA buffers written by the clients
SAREA	
	• Store dirty hardware specific state
	• Store *cliprects*

The distribution of the responsibilities to the several components was done according to the forces present in each space and the goals. Table 2.4 lists this distribution. In summary, we shift the most of OpenGL implementation load to the libGL driver in the client space, leaving to the DDX and DRM the low level task such as setup and communication.

Figure 2.3 and 2.4 respectively show the action and sequence diagram of a typical operation.

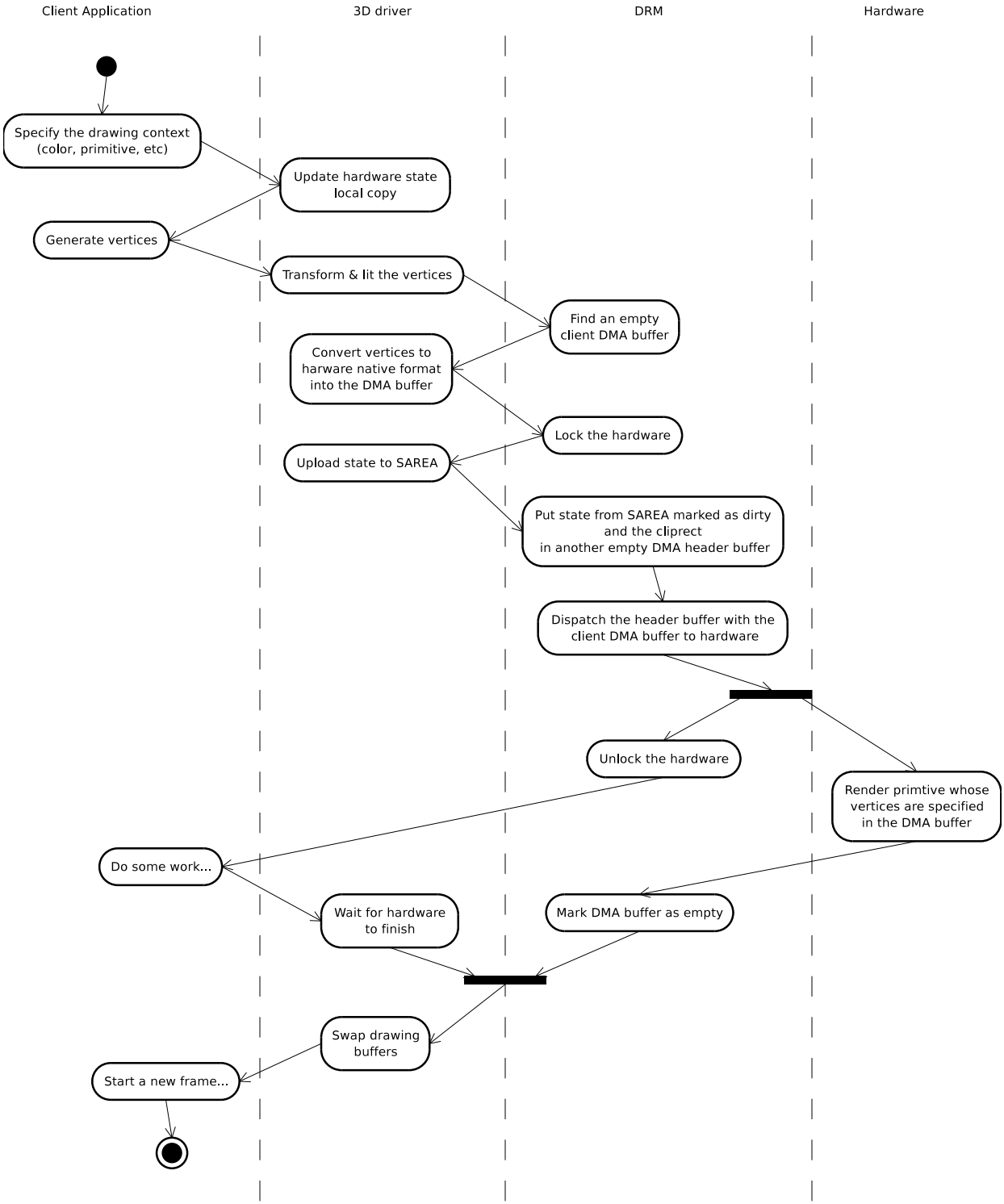


Figure 2.3: Operation action diagram

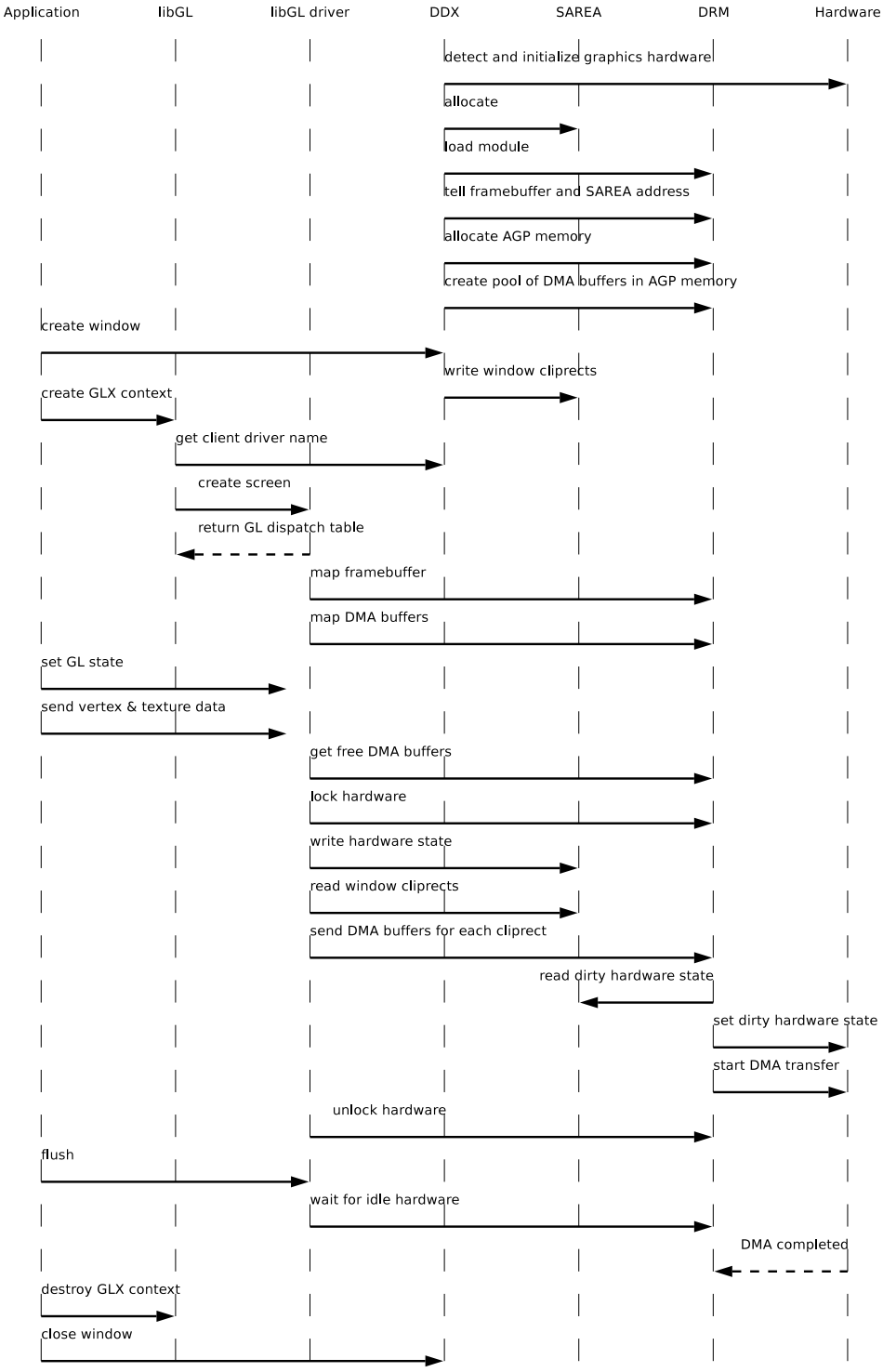


Figure 2.4: Operation sequence diagram

Chapter 3

Design patterns

Although code is written in C, the OOP paradigm is more or less present em several sub-components (most notably on Mesa, where objects and inheritance are used to model drivers, textures, and contexts).

3.1 Inheritance

There are two C idioms used to implement inheritance – by structure reference and by aggregation. They mostly differ in the order of initialization and memory allocation.

3.1.1 Inheritance by reference

In the inheritance by reference idiom, the base class data structure has an extra attribute which is a pointer to the subclass data structure.

Listing 3.1 and 3.2 respectively describe the implementation of the base and derived classes shown in 3.1. Virtual methods are implemented as function pointers. If the destructor needs to be virtual (quite often the case) then it is implemented like the virtual `aMethod()` shown.

Listing 3.1: Inheritance by reference base class implementation

```
struct BaseClass
{
    int an_attribute;
    (*aMethod)();
    void *pderived;
};

struct BaseClass *BaseClass_create()
{
    struct BaseClass *pbase;
    if((pbase = malloc(sizeof(struct BaseClass))) == NULL)
        return NULL;
```

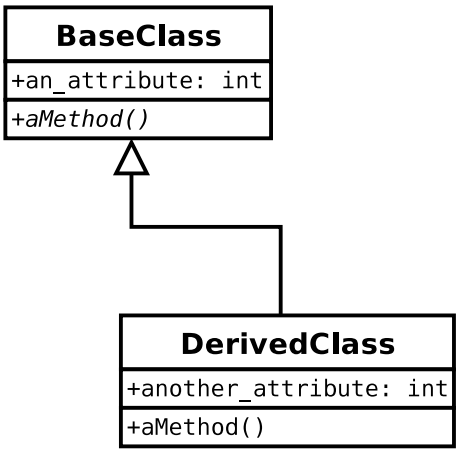


Figure 3.1: Inheritance example.

```
    memset(pbase, 0, sizeof(struct BaseClass));
    pbase->aMethod = BaseClass_aMethod;
    return pbase;
}

void BaseClass_destroy(struct BaseClass *pbase)
{
    if(pbase->pderived)
        free(pbase->pderived);
    free(pbase)
}

BaseClass_aMethod(struct BaseClass *pbase, ...)
{
    ...
}
```

Listing 3.2: Inheritance by reference derived class implementation

```
struct DerivedClass
{
    int another_attribute;
};

int DerivedClass_create(struct BaseClass *pbase)
{
    struct DerivedClass *pderived;
    if((pderived = malloc(sizeof(struct DerivedClass))) == NULL)
        return 0;

    memset(pderived, 0, sizeof(struct DerivedClass));
    pbase->aMethod = DerivedClass_aMethod;
    pbase->pderived = pderived;
    return 1;
```

```
}

DerivedClass_aMethod(struct BaseClass *pbase, ...)
{
    struct DerivedClass *pderived = (struct DerivedClass *)pbase->pderived;
    ...
    pderived->another_attribute = ...
    ...
}
```

This idiom allows for the initialization of the derived class to be deferred. It can be useful when the base class is not abstract, and no method is overridden by the derived class – only new attributes are added – avoiding the need to use patterns such as *Abstract Factory* or *Factory Method*.

This idiom results in more memory allocations, and an additional indirection is required to obtain the derived class pointer from the base class.

3.1.2 Inheritance by aggregation

In the inheritance by aggregation idiom, the derived class data structure contains the base class data structure as its first attribute.

Again, virtual methods are implemented as function pointers.

Listing 3.3 and 3.4 show the implementation of the base and derived classes respectively.

Listing 3.3: Inheritance by reference base class implementation

```
struct BaseClass
{
    int an_attribute;
    (*aMethod)();
};

int BaseClass_init(struct BaseClass *pbase)
{
    memset(pbase, 0, sizeof(struct BaseClass));
    pbase->aMethod = DerivedClass_aMethod;
    return 1;
}

void BaseClass_deinit(struct BaseClass *pbase)
{
    ...
}

BaseClass_aMethod(struct BaseClass *pbase, ...)
{
    ...
}
```

Listing 3.4: Inheritance by reference derived class implementation

```
struct DerivedClass
{
    struct BaseClass base;
    int another_attribute;
};

struct DerivedClass * DerivedClass_create(struct BaseClass *pbase)
{
    struct DerivedClass *pderived;
    if((pderived = malloc(sizeof(struct DerivedClass))) == NULL)
        return NULL;

    memset(pderived, 0, sizeof(struct DerivedClass));
    if(!BaseClass_init(&pderived.base))
    {
        free(pbase);
        return NULL;
    }
    pbase->aMethod = DerivedClass_aMethod;
    return pderived;
}

void DerivedClass_destroy(struct DerivedClass *pderived)
{
    DerivedClass_destroy
    free(pderived);
}

DerivedClass_aMethod(struct BaseClass *pbase, ...)
{
    struct DerivedClass *pderived = (struct DerivedClass *)pbase;
    ...
    pderived->another_attribute = ...
    ...
}
```

With this idiom the base and derived classes are allocated simultaneously and share the same base pointer.

3.2 Abstract Factory and Factory Method

The *Abstract Factory* pattern is used to instantiate the driver, such as the case of the `DriverAPI` class in the fig. 3.2.

The *Factory Method* is also often used, e.g., when creating textures objects in order to allow the driver create an specialized object.

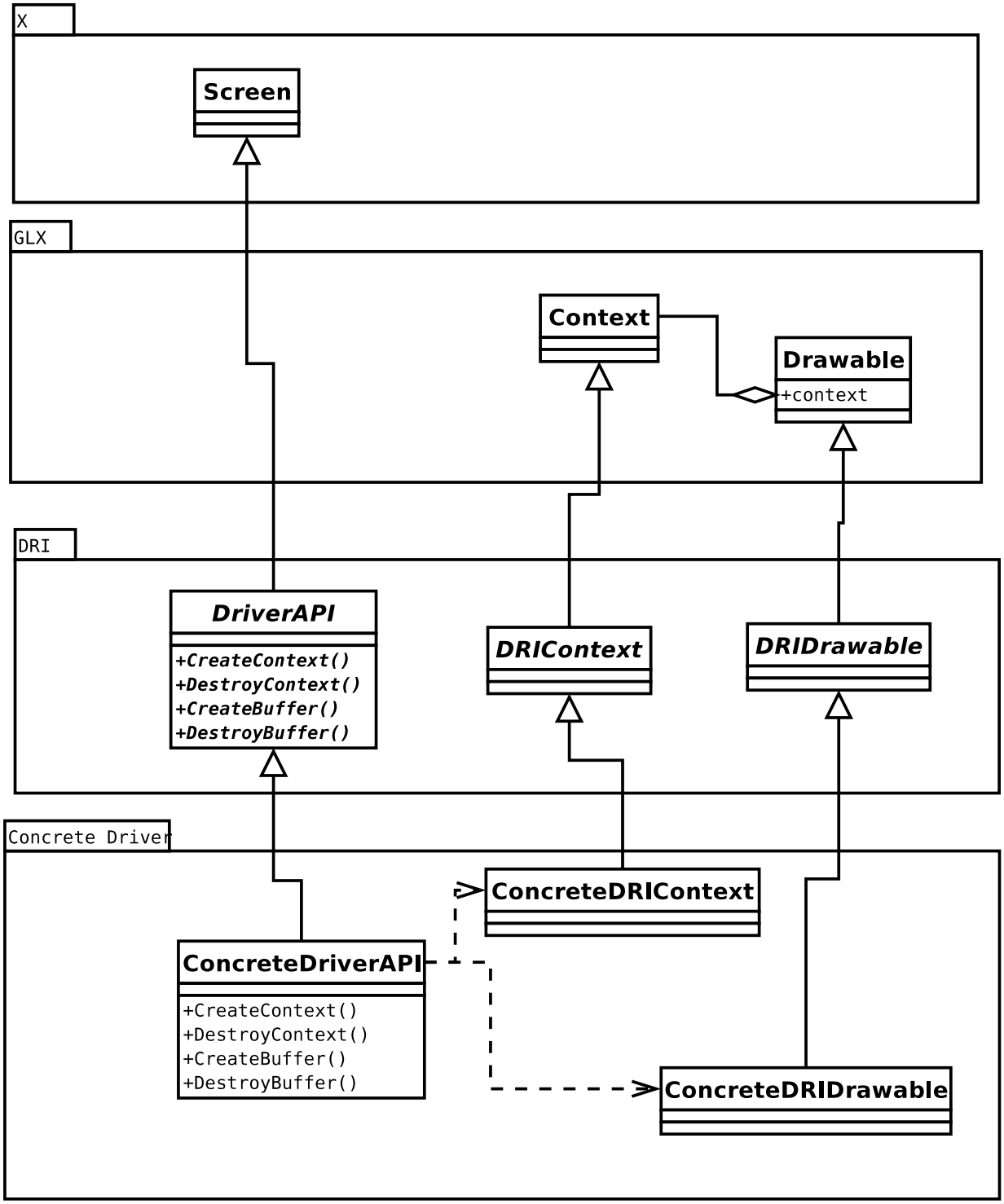


Figure 3.2: Driver class diagram.

3.3 Template

Many graphical algorithms have a common skeleton, but slight variations in some of the steps – the main force for using the Template pattern.

See for instance the interpolation example in listing 3.5, which varies according which vertices components (colors, textures coordinates, ...) are to be interpolated. Here the variations are implemented with *if*-statements, but hooks and callbacks could be used instead. Regardless of which one, both impose a runtime overhead for this flexibility. This is unacceptable as, depending on the OpenGL context, some of these functions can be the performance bottleneck.

Listing 3.5: Vertex interpolation function

```
extern int color_enabled;
extern int textures_enabled;

void interpolate(float k, vertex_t *v1, vertex_t *v2, vertex_t *vr) {
  float omk = 1.0 - k;
  vr->x = k*v1->x + omk*v2->x;
  vr->y = k*v1->y + omk*v2->y;
  vr->z = k*v1->z + omk*v2->z;
  if(color_enabled) {
    vr->r = k*v1->r + omk*v2->r;
    vr->g = k*v1->g + omk*v2->g;
    vr->b = k*v1->b + omk*v2->b;
  }
  if(textures_enabled) {
    vr->u = k*v1->u + omk*v2->u;
    vr->z = k*v1->v + omk*v2->v;
  }
  ...
}
```

The solution found to this problem was shifting the variability from runtime to compile-time by (ab)using the CPP (C Pre-Processor). The generic template is written in a separate header and variability achieved using CPP directives and macros, as shown for the example above in listing 3.6. All the variations are instantiated in a separate C file by repeatedly including the template header, as shown in listing 3.7. At runtime the appropriate variation of the function is chosen only when the GL context changes.

Listing 3.6: `interpolate_tmp.h` – vertex interpolation function template

```
void TAG(interpolate)(float k, vertex_t *v1, vertex_t *v2, vertex_t *vr) {
  float omk = 1.0 - k;
#if HAS_XYZ
  vr->x = k*v1->x + omk*v2->x;
  vr->y = k*v1->y + omk*v2->y;
  vr->z = k*v1->z + omk*v2->z;
#endif
#if HAS_COLOR
```

```
  vr->r = k*v1->r + omk*v2->r;
  vr->g = k*v1->g + omk*v2->g;
  vr->b = k*v1->b + omk*v2->b;
#endif
#if HAS_TEXTURE
  vr->x = k*v1->x + omk*v2->x;
  vr->y = k*v1->y + omk*v2->y;
  vr->z = k*v1->z + omk*v2->z;
#endif
  ...
}

#undef TAG
#undef HAS_XYZ
#undef HAS_COLOR
#undef HAS_TEXTURE
```

Listing 3.7: Instantiation of the vertex interpolation template

```
#define TAG(x) x##_xyz
#define HAS_XYZ
#include "interpolate_tmp.h"

#define TAG(x) x##_xyzuv
#define HAS_XYZ
#define HAS_TEXTURE
#include "interpolate_tmp.h"

...

void (*interpolate)(float k, vertex_t *v1, vertex_t *v2, vertex_t *vr);

void glEnable(GLenum cap)
 if(cap == GL_TEXTURE_2D)
  interpolate = interpolate_xyzuv;
 elif ...
  ...
 else
  interpolate = interpolate_xyz;
}
```

Although the variability in this example was controlled by using yes/no macros, macros containing code which is expanded in the template, can be and are used in other more complex situations.

Chapter 4

Possible enhancements

Roughly around 30% to 50% of driver code is similar to other drivers, apart from small modifications. To develop a 3D open-source driver takes around 2 men-year. So there is a strong motivation for code reuse, but also much inertia due to the code size and lack of refactoring tools.

More code reuse could be achieved by incrementally:

- bringing the infrastructure closer the OOP paradigm (more inheritance and inclusion of objects);
- using/developing better tools for automatic code generation (such as a smarter template engine);

specially in the driver code.

Bibliography

[1] Rickard E. Faith. *The Direct Rendering Manager: Kernel Support for the Direct Rendering Infrastructure*. Rickard E. Faith, May 1999.

[2] Rickard E. Faith and Kevin E. Martin. *A Security Analysis of the Direct Rendering Infrastructure*. Precision Insight Inc., May 1999.

[3] Rickard E. Faith, Jens Owen, and Kevin E. Martin. *Hardware Locking for the Direct Rendering Infrastructure*. Precision Insight Inc., May 1999.

[4] Kevin E. Martin, Rickard E. Faith, Jens Owen, and Allen Akin. *Direct Rendering Infrastructure, Low-Level Design Document*. Precision Insight Inc., May 1999.

[5] Jens Owen. The dri project history. http://dri.freedesktop.org/wiki/DriHistory.

[6] Jens Owen and Kevin Martin. *DRI Extension for supporting Direct Rendering: Protocol Specification*. Precision Insight Inc., May 1999.

[7] Jens Owen and Kevin E. Martin. *A Multipipe Direct Rendering Architecture for 3D*. Precision Insight Inc., September 1998.

[8] Jens Owen and Liam Smit. Dri control flow. http://dri.sourceforge.net/doc/dri_control_flow.html.

[9] Jens Owen and Liam Smit. Dri data flow. http://dri.sourceforge.net/doc/dri_data_flow.html.

[10] Brian Paul. *Introduction to the Direct Rendering Infrastructure*, August 2000.

[11] Dri wiki. http://dri.freedesktop.org/wiki/.

Appendix A

Data transfer modes

A.1 Programmed Input/Output (PIO)

Programmed Input/Output (PIO) is a data transfer to/from a I/O address (usually a register) a byte/word at a time using a dedicated processor instruction (such as Intel `in` and `out` instructions).

Listing A.1 shows an example of using PIO to program the drawing of a triangle on a hypothetical hardware.

Listing A.1: PIO example

```
void draw_triangle(int x1, int y1, int z1,
                   int x2, int y2, int z2,
                   int x3, int y3, int z3)
{
    while(!(intw(REG_TRI_STATUS) & MASK_TRI_IDLE))
        usleep(1); // wait for idle
    outw(x1, REG_TRI_X1);
    outw(y1, REG_TRI_Y1);
    outw(z1, REG_TRI_Z1);
    outw(x2, REG_TRI_X2);
    outw(y2, REG_TRI_Y2);
    outw(z2, REG_TRI_Z2);
    outw(x3, REG_TRI_X3);
    outw(y3, REG_TRI_Y3);
    outw(z3, REG_TRI_Z3);
    outw(inw(REG_TRI_STATUS) | MASK_TRI_DRAW, REG_TRI_STATUS);
}
```

A.2 Memory Mapped Input/Output (MMIO)

Memory Mapped Input/Output (MMIO) is a data transfer to/from a range of I/O addresses (such the graphics card memory, or register) which has been memory-mapped

into the virtual address space using regular memory access instructions (such as Intel `mov` instructions).

Listing A.2 shows the same example using MMIO.

Listing A.2: MMIO example

```
extern int* pmmio; // pointer to MMIO region

void draw_triangle(int x1, int y1, int z1,
                   int x2, int y2, int z2,
                   int x3, int y3, int z3)
{
    while(!(pmmio[REG_TRI_STATUS] & MASK_TRI_IDLE))
        usleep(1); // wait for idle
    pmmio[REG_TRI_X1] = x1;
    pmmio[REG_TRI_Y1] = y1;
    pmmio[REG_TRI_Z1] = z1;
    pmmio[REG_TRI_X2] = x2;
    pmmio[REG_TRI_Y2] = y2;
    pmmio[REG_TRI_Z2] = z2;
    pmmio[REG_TRI_X3] = x3;
    pmmio[REG_TRI_Y3] = y3;
    pmmio[REG_TRI_Z3] = z3;
    pmmio[REG_TRI_STATUS] |= MASK_TRI_DRAW;
}
```

A.3 Direct Memory Access (DMA)

Direct Memory Access (DMA) is a bulk data transfer between the system memory and peripheral device without the processor intervention.

Listing A.3 shows the same example using DMA.

Listing A.3: DMA example

```
extern unsigned long bufaddr; // physical address (multiple of 4K)
extern int* buf; // virtual address
extern unsigned bufsiz, buflen;

void draw_triangle(int x1, int y1, int z1,
                   int x2, int y2, int z2,
                   int x3, int y3, int z3)
{
    int *p;
    if(buflen + 10 > bufsize) {
        while(!(intw(REG_DMA_STATUS) & MASK_DMA_IDLE))
            usleep(1); // wait for idle
        outw(bufaddr >> 12, REG_DMA_BUFADDR);
        outw(buflen, REG_DMA_BUFLEN);
        outw(inw(REG_DMA_STATUS) | MASK_DMA_START, REG_DMA_STATUS);
```

```
        buflen = 0;
    }
    p = buf + buflen;
    *p++ = COMMAND_TRI_DRAW;
    *p++ = x1; *p++ = y1; *p++ = z1;
    *p++ = x2; *p++ = y2; *p++ = z2;
    *p++ = x3; *p++ = y3; *p++ = z3;
    buflen += 10
}
```

Ring buffer. To avoid having the CPU to waiting for the graphics engine completion of a previous buffer before processing one, many graphics hardware possess a *ring buffer* – a circular DMA buffer where pointers to regular DMA buffers are queued –, as illustrated by figure A.1.

Processed buffers are collected back into the buffer pool either via an IRQ handler or by stamping. When stamping each buffer is associated with an unique and increasing number which is written to a scratch register as the buffer last command as the card processes it. A buffer can be easily determined as processing or pending by comparing its stamp with the value presently in the scratch register.

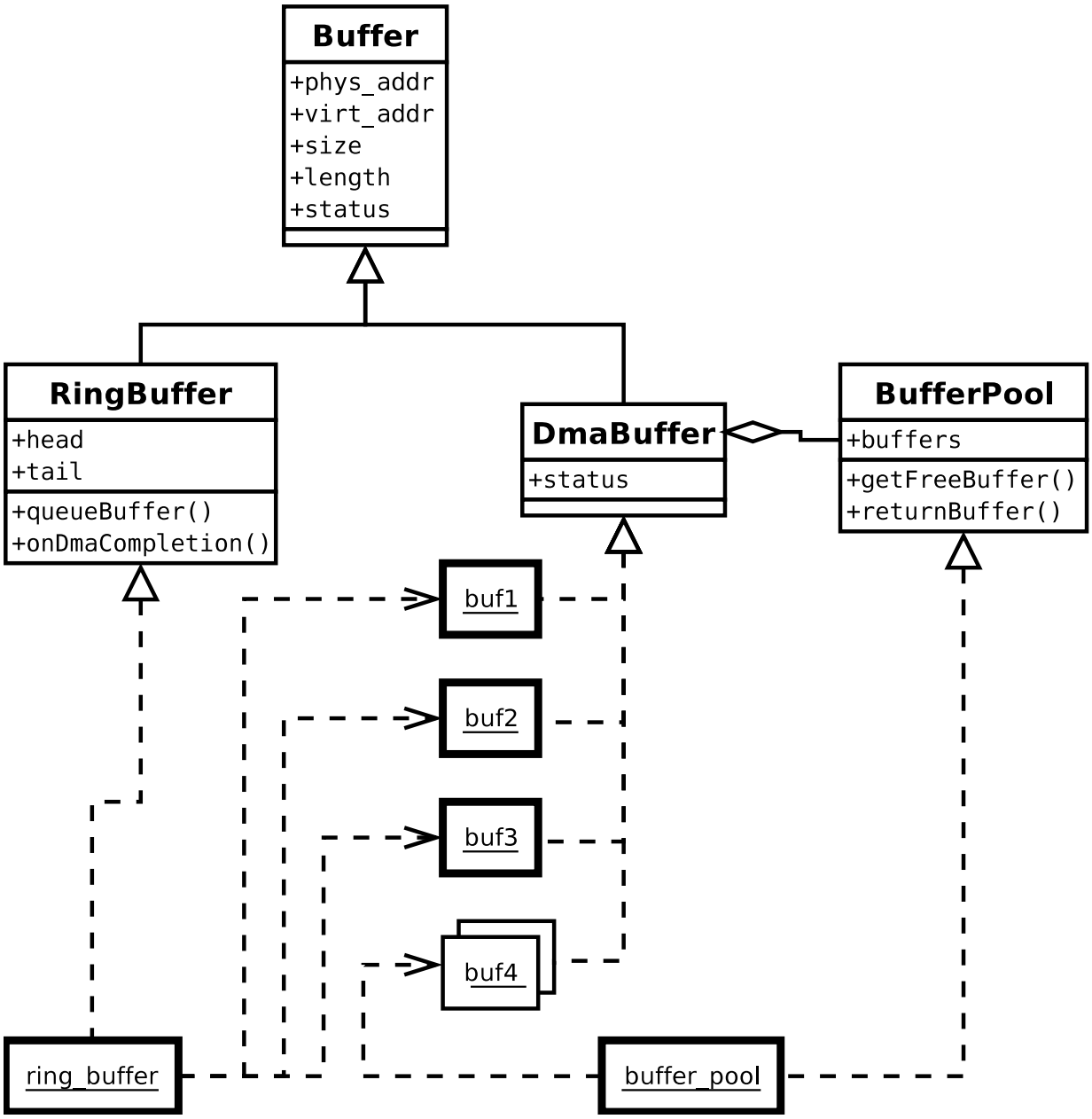


Figure A.1: DMA via a ring buffer