Developing for Multiple High-Performance Graphics APIs Simultaneously: A Case Study

Markus Hadwiger
msh@cg.tuwien.ac.at

Institute of Computer Graphics
Vienna University of Technology
Vienna / Austria

Abstract

In today's world of interactive computer graphics applications the choice of graphics API is crucial, but still far from clear-cut. Especially when developing for consumer-level hardware the choice of API is very important, since it will have a tremendous impact on the potential market share, as well as performance and flexibility. Nevertheless, there is no single API that is definitely better in every respect than its competitors – particularly when developing for multiple platforms.

The most important APIs, like OpenGL and Direct3D, support a multitude of graphics hardware via vendor-supplied graphics drivers. There are also proprietary APIs supporting only the hardware of a specific vendor. Although these are rapidly becoming more and more obscure, the Glide API by 3dfx Interactive is still ubiquitous in the area of entertainment software.

This paper describes the process of simultaneously developing for multiple graphics APIs. We use Parsec, a three-dimensional space-combat computer game we are currently working on, as a case study. Parsec transparently supports OpenGL, Glide, and a proprietary software-renderer through an abstract interface layer encapsulating the underlying graphics API, without compromising performance.

Keywords: graphics APIs, OpenGL, Glide, entertainment software, computer games, real-time graphics, portability.

1. Introduction

Over the last few years, the focus of graphics development in the area of entertainment software has clearly shifted from exclusive software rendering to the support of consumer-level 3-D hardware accelerators. Very prominent examples of such accelerators are boards featuring a chipset of the Voodoo Graphics family by 3dfx Interactive. Although powerful graphics boards are already widespread throughout the potential customer base for computer games, most companies still offer optional software rendering in their products. The most important reason for this is that the majority of desktop computers is still not equipped with special-purpose 3-D hardware. Nevertheless, this situation is changing rapidly and many future releases will require hardware acceleration.

A very important issue in the development of contemporary computer games is the choice of the underlying graphics API that is used to access the facilities offered by the graphics hardware. One criterion for this decision is whether to support only a single family of accelerators through a proprietary API like 3dfx's Glide, or a multitude of different hardware vendors' products through an industry-standard API like OpenGL or Direct3D. From a marketing point of view, this decision might seem to be easy. Clearly, the bigger the customer base, the better. A very important issue, however, is support and quality of an API – in particular, the availability and quality of the drivers necessary to access the target hardware. The choice of API has always been far from clear-cut and is still crucial. For this reason, many developers adopt the approach of supporting more than one API, letting the user select at installation or startup time, maybe even on-the-fly from within the program.

If one decides to support more than one graphics API, maybe in addition to proprietary software rendering, the issue of how to effectively develop in such a scenario becomes critically important. We are currently developing a three-dimensional space-combat computer game [Parsec], that supports OpenGL, Glide, and a proprietary software-renderer transparently. In order to make this possible, we have designed an abstract interface layer comprised of several subsystems. This abstract interface is implemented for all supported APIs. However, all other graphics code resides entirely above this interface layer and is therefore independent from the graphics API actually used at any one time. Supporting additional graphics APIs is possible by simply implementing the functionality required by the interface for the new target. Thus, the vast majority of the code does not need to be changed.

2. High-performance graphics APIs

This section briefly touches on some issues regarding the most important graphics APIs in the current desktop consumer-market. In this paper, we are not going to explicitly discuss aspects of console or coin-op development, although many things may apply equally well to these.

2.1 Glide

In 1996, 3dfx's Voodoo Graphics accelerator made powerful, low-cost 3-D hardware for desktop computers feasible for the very first time. It was able to quickly gain acceptance by developers and game players alike. A crucial reason for this was the free availability of the Glide SDK [Glide], allowing developers easy access to all features natively supported by the chipset. In fact, the Glide API is a very thin layer above the register level of the actual hardware. The core Glide functionality exports exactly the features of the hardware. This implies that the developer can be sure that all features will be hardware-accelerated, obviating the need to query and cater to widely differing capabilities at run-time. The dominating position of the original Voodoo Graphics and its successors during the last few years made it perfectly feasible to support Glide as sole API. If no hardware acceleration was available, most games were able to run entirely in software, most of the time using a proprietary software-renderer.

However, during the last year many new low-cost graphics accelerators offering very high performance have become available, and the driver situation has also improved tremendously. Thus, a migration from using Glide to APIs like OpenGL or Direct3D, that support a multitude of different hardware accelerators, can be observed.

2.2 OpenGL and Direct3D

OpenGL is a very powerful and widely used graphics API that evolved from Silicon Graphics' IrisGL. Originally meant for programming workstation-class 3-D hardware, high-quality OpenGL drivers are available for the most important consumer products today. This is also due to the fact that workstation and consumer hardware are in the process of converging in many respects. A very important property of OpenGL is that it is supported on many platforms. Graphics code written for OpenGL can easily be ported to a PC, a Macintosh, and – of course – an SGI workstation, among others. OpenGL is also an extremely well documented API. For more information about OpenGL see [GLsite].

Direct3D, which is part of [DirectX], also supports a multitude of graphics hardware, although it is almost exclusively restricted to Microsoft Windows platforms. Nevertheless, these comprise the most important segment of the computer games market for the desktop, and Direct3D is widely supported by game developers.

A very important difference between OpenGL and Direct3D is that an OpenGL implementation is required to support the entire feature set as defined by the standard, whereas Direct3D exports the functionality of the hardware in the form of capability bits. This implies that OpenGL must emulate missing hardware functionality in software. With Direct3D, the programmer has to explicitly determine what to do should desired features be found missing.

2.3 Geometry processing support

A crucial issue is whether an API supports geometry processing, or is rasterization-only. Since Glide exports only the features of the actual hardware and all Voodoo Graphics accelerators to date do not support geometry acceleration, the API does not support three-dimensional operations like transformation, clipping, and

projection. All coordinates in Glide are screen-coordinates together with additional attributes that the hardware interpolates over a triangle, e.g., texture coordinates (U,V,W) and color (R,G,B,A). That is, coordinates are already in post-perspective space. As soon as accelerators supporting geometry processing in hardware become widespread, this poses a problem to the API. There are already two major versions of Glide – 2.x and 3.x, the latter already supporting a homogeneous clipping space –, and the API will probably continue to evolve with the capabilities of 3dfx hardware.

Both OpenGL and Direct3D support geometry processing, allowing to exploit geometry processors if available. Current consumer hardware is still restricted to rasterization, but this will probably change in the not-too-distant future. There is also another issue related to whether an API supports geometry processing; if it does, and there is no geometry processing hardware, the driver has to do all geometry calculations. This has important implications. First, the performance of a program will depend even more on the quality of the driver. However, driver-quality is already quite high and developers are hard-pressed to outperform them with their own code. This is particularly true when special purpose features of the host CPU are leveraged. The MMX extensions to the instruction set of the Pentium processor are not really well suited to geometry code due to their integer nature, but AMD's 3DNow! [AMD] and Intel's Streaming SIMD Extensions [Intel], introduced with the Pentium III, feature SIMD floating point instructions that can be put to good use in 3-D graphics drivers. Supporting all of these extensions directly in a program requires a lot of development effort that might not be justified. The most important reason, however, to use the geometry support of a graphics API is to be prepared for the future widespread availability of hardware with dedicated geometry processing support.

That said, Parsec currently does most geometry calculations before submitting primitives to the driver. OpenGL has to be configured to use an orthogonal projection to only use it for rasterization.

3. Parsec subsystem structure

This section introduces the subsystem structure of Parsec, which we use to ensure modularity, portability, and flexibility. Portability is a high priority design criterion for Parsec in its entirety. This concerns host platforms, as well as graphics APIs. We currently support Win32, MacOS, Linux, and DOS as host platforms. Supported graphics APIs are OpenGL, Glide, and software rendering, although not all are supported on every host platform at the moment. The three graphics-related subsystems are described in detail in section 4.

Figure 1 depicts the Parsec subsystem structure, comprised of seven subsystems altogether.

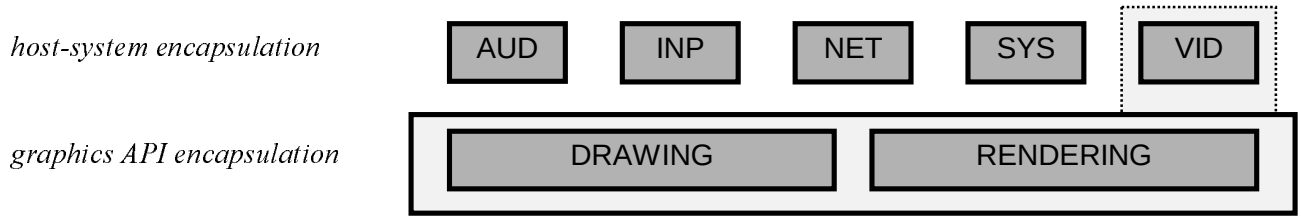

Figure 1: Parsec subsystem structure

3.1 Subsystems overview

There are two distinct types of subsystems. The first type is used to encapsulate general host system functionality. Subsystems of this type are AUDIO (AUD), INPUT (INP), NETWORKING (NET), SYSTEM (SYS), and VIDEO (VID). The second type encapsulates the graphics API and is comprised of the DRAWING (DRAW), and the RENDERING (REND) subsystems. The abbreviations in the parentheses are related to the naming conventions we use, see section 3.2.

Subsystems where this is at all meaningful can be changed on-the-fly from within Parsec's command console. This is possible for NET, VID, DRAW, and REND. For example, using the VID.SUBSYS command the video subsystem can be switched dynamically. In this case, this entails a dynamic change of the VID, DRAW, and REND subsystems, because it makes no sense for the user to change these individually. From a programming perspective, the distinction between these three subsystems is very important and useful, though.

Dynamic subsystems can be considered being drivers, implementing the subsystem functionality for a specific hardware. As with drivers, they are accessed through a uniform interface behind which implementation details are hidden. Although all subsystem implementations currently have to be linked into the executable in order for them to be available for on-the-fly switching, they could easily be extracted into separate shared libraries (e.g., DLLs).

The distinction between VID on the one hand, and DRAW and REND on the other, is very important to separate video/graphics functionality that depends on the host system (as well as the graphics API) from functionality that depends on the graphics API alone. To make this more clear with an example, there is a different VID for OpenGL under Windows (WGL), and OpenGL under the X Window System (GLX). However, there is only one implementation for OpenGL DRAW and REND functionality for both of these platforms, since they contain system-independent OpenGL code.

3.2 Naming conventions

We use a rather strict naming convention to distinguish functions of the various subsystems. This pertains to the names of the interface functions themselves, as well as the module names containing their implementations. The Parsec codebase mostly consists of code written in a C-like subset of C++, which necessitates name prefixes for easy distinction between different classes of functions, since we do not use qualifiers of the form VIDEO:: and so on.

The implementation of VID under Linux is contained in modules all starting with the VL_ prefix, VW_ modules contain Win32 implementations of the exact same functionality. All Vx_ modules together comprise the entire subsystem implementation for platform x. Similarly, rendering functionality for Glide is contained in RG_ modules, whereas RO_ modules contain OpenGL code.

The exported function names themselves contain only subsystem-specific prefixes, since the names have to be the same for all implementations. For example, rendering functions all start with R_, drawing functions with D_.

3.3 Dynamic vs. static subsystem binding

It is possible to choose at compile-time whether to compile an executable supporting dynamic subsystem binding, to allow on-the-fly switching of subsystems at run-time. If this is not desired, all functions will be bound statically, that is, the compiler inserts actual function addresses into the code, as usual. If dynamic binding is specified for a specific subsystem, however, all calls to its exported functions will be routed through a global jump-table, containing the addresses of the functions implementing the subsystem's abstract interface. Subsystem switching may then be performed by deinitializing the current subsystem, setting all addresses in the jump-table to their new values, and calling the init function of the then newly activated subsystem.

The naming convention is also very important for dynamic subsystem binding and switching. We also want to transparently use the same function names, regardless of whether they are bound dynamically or statically, of course.

Note that all this should not be confused with the optional capability of the host system to load shared libraries. Subsystem binding happens internally in a portable way and does not depend on host system support. Support of some sort of DLLs on the host system may be exploited to load graphics API libraries, however (e.g., opengl32.dll or glide2x.dll). This happens on a lower level and does not influence the subsystem interface in any way. It is a means of ensuring that there need not be driver support on the host system for every graphics API supported by the executable.

We have to distinguish between the caller of a function, and the callee (the function itself).
The caller has to include the subsystem header file if it wants to use a subsystem's functionality. For example, VID_DEFS.H to gain access to all exported VID functions. Figure 2 shows what the part of VID_DEFS.H responsible for declaring a single example function, VIDs_InitDisplay(), looks like and how it can be called.
The preprocessor constant `DBIND_VID` has to be defined if dynamic binding is desired. Otherwise, static binding will be used.

```
VID_DEFS.H:

#ifdef DBIND_VID
     // make use of jump-table transparent
     #define VIDs_InitDisplay     (*vid_subsys_jtab.InitDisplay)
#else
     // ordinary prototype
     void    VIDs_InitDisplay();
#endif


VID_CALLER.C:

#include ″vid_defs.h″

// use VID function
VIDs_InitDisplay();
```

Figure 2: Example caller (use) of VID function (VID_CALLER.C)

Thus, the caller simply uses the function name without considering the type of binding at all. The next figure shows how the callee declares a function it wants to implement. Note that in the case of dynamic binding the actual function name that will go into the object module has to be unique among subsystems – hence the redefinition of the function name. The only module knowing about these names is the one setting the correct addresses in the jump-table.

```
VW_INITG.H:

#ifdef DBIND_VID
     // undefine because we are callee (implementor)
     #undef  VIDs_InitDisplay
     // redefine name to avoid nameclash with other implementations
     #define VIDs_InitDisplay      VIDs_GLIDE_InitDisplay
#endif


VW_INITG.C:

// subsystem header
#include ″vid_defs.h″

// local module header
#include ″vw_initg.h″


// actual name may have been redefined transparently
void VIDs_InitDisplay()
{
  // implementation for this subsystem
}
```

Figure 3: Example callee (implementation) of VID function (VW_INITG.C)

In this way, there will be functions containing _GLIDE_, _OPENGL_, etc. in their function names. However, these are never used directly, apart from the jump-table management module. This system is very simple and transparent to both caller and callee – and therefore very easy to use. The only overhead involved is that all calls to exported subsystem functions will be indirect instead of direct. This has proved to impact performance nearly not at all, also due to the fact that all subsystem functions are designed in a way that their execution takes significantly longer than the time the call and return themselves take. Still, dynamic binding can be disabled by changing a single `#define` if the overhead is felt to be unnecessary and there is only one implementation for each subsystem.

4. Parsec graphics subsystems

This section describes in detail the three graphics subsystems of Parsec – VIDEO, DRAWING, and RENDERING. From a user perspective, these three subsystems together can be viewed as comprising a single graphics subsystem. On-the-fly switching makes only sense for all three subsystems at once, for instance.

4.1 VIDEO subsystem (VID)

As already mentioned in section 3.1 the implementation of the video subsystem depends on the host system, as well as on the graphics API. It is responsible for establishing the connection between the video system (say, the Win32 API and WGL) and the rendering system (say, OpenGL).

This includes creating and destroying windows, binding rendering contexts, managing full-screen and windowed modes, setting the gamma correction if available, etc. Like all other subsystems, VID has its own global jump-table, a portion of which can be seen in Figure 4. In total, there are about 30 different functions in VID, not counting system-independent helper functions.

The names of the corresponding functions as actually used would be VIDs_ClearRenderBuffer(), VIDs_CommitRenderBuffer(), and so on. The functions are divided into logical groups like BUFF (frame buffer management), and INIT (mode initialization, restoration). Although this is no requirement, normally all functions of a group are implemented in a single module, for example, VW_BUFFG.C for the implementation of BUFF functionality under Win32/Glide.

The interface has to be able to transparently handle vastly differing video properties. In the software-renderer, for instance, VIDs_CommitRenderBuffer() would actually blit the entire off-screen render buffer to the visible page in video memory, or just switch pages, depending on the graphics mode (host memory rendering or video memory rendering, with and without double buffering, ...). Under Glide it would just call grBufferSwap(), under OpenGL/Win32 SwapBuffers(), etc. So, the implementation is only required to ensure correct, conforming semantics.

```
struct vid_subsys_jtab_s {

  // BUFF GROUP
  void  (* ClearRenderBuffer )();
  void  (* CommitRenderBuffer )();

  // INIT GROUP
  void  (* InitDisplay )();
  void  (* RestoreDisplay )();
  void  (* InitFrameBufferAPI )();

  // SUPP GROUP
  int   (* SetGammaCorrection )( float gamma );

};
```

Figure 4: VIDEO subsystem jump table (excerpt) for dynamic function binding

4.2 DRAWING subsystem (DRAW)

The drawing subsystem is responsible for two-dimensional graphics code, e.g., drawing of bitmaps, font drawing, etc. There are about 40 different functions in DRAW. It solely depends on the graphics API and has to be portable to all supported host systems providing the respective API. Host system-specific code has to be put into the video subsystem.

The focus on 2-D code is not the only and most important difference from the rendering subsystem described in the next section, however. The drawing subsystem is inherently low-level and graphics primitive-oriented. Originally, it only existed to be able to draw various kinds of bitmaps. Over time it has evolved to the first half of a two-tier rendering functionality encapsulation. The level of abstraction of the functions it is comprised of is very much comparable to what the Glide API offers with respect to rendering. That is, it is primitive-oriented (triangles, rectangles, triangle strips, etc.) and provides mostly rasterization facilities.

These primitives are specified either in eye-space, or in already post-perspective screen-space. Although, say, a triangle, may have a Z coordinate specified, this could already be the value that will actually be stored into the depth buffer without any further conversion or mapping (screen-space Z). It could also be eye-space Z for which the corresponding function automatically performs the projection. Most of these primitive-functions are subsumed in the ITER interface, whose name stems from the property that basically 2-D primitives are specified, together with attributes that will be iterated over the area of the primitive, which may be Z, texture coordinates, color, and the like.

The notion if the ITER interface is so important within Parsec, that it is described in more detail in section 5.2. It is the most important part of the drawing subsystem and is used for direct rendering of primitives

specified in an abstract format that will be automatically converted to the format of the underlying graphics API.

Figure 5 shows a portion of the global jump-table for the drawing subsystem.

```
struct d_subsys_jtab_s {

  // BLIT GROUP
  int   (* ReadBuffRegion )( int buffid, ... );

  // BMAP GROUP
  void  (* PutBitmap )( char *bitmap, ... );
  void  (* PutTrBitmap )( char *bitmap, ... );

  // ITER GROUP
  void  (* DrawIterTriangle3 )( IterTriangle3 *ittri );
  void  (* DrawIterTriStrip3 )( IterTriStrip3 *itstrip );
};
```

Figure 5: DRAWING subsystem jump table (excerpt) for dynamic function binding

4.3 RENDERING subsystem (REND)

In contrast to the low-level functionality provided by the drawing subsystem, the rendering subsystem is inherently high-level. Its level of abstraction is the level of entire objects. That is, one provides a function with an entire object, maybe the entire world, and wants to have that rendered. Such an object might also be a particle object; an object solely comprised of a set of particles. Due to the high level of this subsystem there are not as many exported functions as in the other subsystems. In total, it is comprised of only about 15 different interface functions.

At this level of abstraction there never are single primitives involved. Therefore, the rendering subsystem is responsible for drawing everything that has a huge impact on overall program performance. Because its implementation is already at the level of the underlying graphics API, there is a lot more code duplication than in the drawing subsystem. There is OpenGL code to render an entire object of a specific type, Glide code for practically doing the same thing, and so on.

Well, why is that? The answer is simple – performance. The interface between portable code and graphics API-dependent code is at a very high level. This means that the rendering code can also be optimized at a more global scope than is possible at the primitive level. For example, if a particle object consists of particles all having the same attributes, it may be possible to render the entire object with basically a single API call like glDrawArrays(). This helps tremendously in the minimization of state switches which has a direct and often tremendous impact on performance.

If the ITER interface (part of the drawing subsystem) were used to render each particle independently, the number of API calls would be proportional to the number of particles, not the number of particle objects. Naturally, how much an object is likely to profit from being rendered at this level depends on its type. Independent, dynamically generated polygons (e.g., for shockwaves) are better suited to being rendered at the primitive-level, for instance.

Figure 6 shows a portion of the global jump-table for the rendering subsystem.

```
struct r_subsys_jtab_s {

  // OBJ GROUP
  void  (* RenderObject )( GenObject *objectp );
  void  (* DrawWorld )( const Camera camera );

  // PART GROUP
  int   (* DrawParticleCluster )( pcluster_s *cluster );
  void  (* DrawParticles )();

  // SFX GROUP
  void  (* DrawLensFlare )();
  void  (* DrawPanorama )();
};
```

Figure 6: RENDERING subsystem jump table (excerpt) for dynamic function binding

5. Rendering with an abstract API

Developing for multiple graphics APIs simultaneously most likely means developing for an internal API, instead of directly using a system-provided graphics API. Calls to this API are then translated to the underlying API by the respective subsystem implementation. The design of such an abstraction of the functionality of all supported actual implementations is crucial and has a tremendous impact on many important aspects, from performance to ease of use for the developer.

5.1 Graphical primitives

An important issue is the types of primitives that are supported. What low-level primitives are there, what high-level primitives? This will directly influence what can be done with the API and how well it is able to fit together with, say, OpenGL, or Glide.

High-level rendering in Parsec operates at the level of entire objects, be they polygonal or particle. These objects may contain auxiliary data that has been created or loaded, depending on the actual graphics API currently in use. For instance, they might contain a BSP tree for visibility determination in software, or their polygons might be sorted on textures to minimize hardware state switches.

Low-level rendering operates at the graphical primitives level. Triangles, rectangles, polygons, triangle fans and strips, and so on. This level normally is a very thin layer above the actual API, but exactly for this reason it definitely has the most overhead, because each primitive has to be converted and rendered separately, without easily being able to take global information into account.

5.2 The ITER interface

In addition to what has already been said about this interface in section 4.2, it not only renders primitives, but it also specifies how to describe them independently of any underlying graphics API. Most importantly, it specifies how to shade a primitive. This shading specification is not as complicated and powerful as, for example, specifying shaders in RenderMan, but it encapsulates what current hardware is able to perform in real-time in order to shade primitives. This mostly means iterating some specified attributes over the area of the primitive, that is, linearly interpolating them. The ITER interface is inherently hardware-oriented. That is, it currently knows nothing about lighting, or surface properties like ambient and diffuse color. It simply takes attribute values like RGBA for each vertex of a primitive and, well, iterates them.

Figure 7 shows the type that is used to specify the mode of iteration, as well as composition. Composition specifies how, for instance, iterated color should be combined with a color sample from a texture and how the finished fragment should be combined with the previous contents of the frame buffer.

```
enum itertype_t {

  iter_constrgb,      // constant (rgb) attributes
  iter_constrgba,     // constant (rgba) attributes
  iter_rgb,           // iterate (rgb) attributes
  iter_rgba,          // iterate (rgba) attributes
  iter_texonly,       // apply texture without shading
  iter_texconsta,     // modulate texture with constant (a)
  iter_texconstrgb,   // modulate texture with constant (rgb)
  iter_texconstrgba,  // modulate texture with constant (rgba)
  iter_texa,          // modulate texture with iterated (a)
  iter_texrgb,        // modulate texture with iterated (rgb)
  iter_texrgba,       // modulate texture with iterated (rgba)
  iter_constrgbtexa,  // constant (rgb) with alpha-only texture
  iter_constrgbatexa, // constant (rgba) with alpha-only texture
  iter_rgbtexa,       // iterated (rgb) with alpha-only texture
  iter_rgbatexa,      // iterated (rgba) with alpha-only texture

  iter_overwrite,     // overwrite destination
  iter_alphablend,    // alpha blend with destination
  iter_modulate,      // modulate destination (multiply)
  iter_specularadd,   // add to destination (emissive/specular)
  iter_premulblend,   // blend with premultiplied alpha
};
```

Figure 7: Specification of vertex attributes iteration and fragment combination

6. Performance considerations

In this section we are going to look at some very important performance considerations when using current rendering hardware.

6.1 State-switch minimization

It is very important to minimize hardware and driver state switches as much as possible. Their impact may range from the execution of unnecessary code to stalling the entire pipeline on the graphics accelerator. One of the most costly state switches on probably nearly all accelerators is the texture state. A texture switch might flush the entire pipeline and the cache, maybe even necessitate downloading the texture into on-board texture memory all over again. Therefore, if the currently active graphics subsystem is susceptible to texture state hiccups, polygons should be rendered sorted according to their texture. After that, sorting on other attributes like color might also be helpful.

One problem with all this is that if one uses a series of API calls in the same code segment avoiding redundant calls may be easy. It gets worse, though, when the actual API calls are encapsulated behind an abstract interface which is used per primitive. In Parsec, we log the current state of the graphics API and use it to automatically filter redundant API calls. The `iter_` constants of figure 7 help a lot with this, because each one of them has a given associated set of state variables and they need not be checked individually.

6.2 Performance-enhancing data preprocessing

Sorting on attributes brings us to the issue of data preprocessing to ensure the data is in the best possible format for the currently active graphics API and hardware. This might involve converting texture formats, like pre-expanding palettes to RGB, using RGB565 instead of RGB888, the other way around, etc. Color format conversion might be especially useful when switching frame buffer color depth. Some of these conversions may be mandatory, say, if the API cannot take some data in its original format, or simply performance-enhancing. A BSP tree may enhance performance, e.g., in the software-renderer, or prevent drawing in texture- instead of visibility-order and better be dropped, or only used for collision detection. It might also be useful to triangulate all models at load-time if they are not already triangle-meshes, maybe also automatically extracting triangle strips. In order to avoid having to retriangulate due to clipping, guard band clipping may be used if supported by the hardware. This basically means clipping at the fragment-level in a certain region just outside the valid viewport.

6.3 Texture memory management

A huge issue might be whether the application has to do its own texture memory management. Glide, for example, provides direct, low-level access to texture memory, but the developer is responsible for memory management and caching. The performance of the texture cache may have – and usually has – a tremendous impact on the overall application performance. This approach provides maximum flexibility put also requires a large amount of work.

OpenGL, on the other hand, provides texture management by the driver. Thus, the developer has to consider the actual caching scheme used. Rendering sorted on textures might defeat a LRU policy, for instance, if no precautions are taken. It is also not easy to avoid having to hold all MIP-map levels of a texture in texture memory at the same time, even if only two or three might be in use for a large number of consecutive frames.

Considering MIP-mapping; it is very important to provide MIP-maps if a texture might be downsampled. If consecutive texel lookups have to be done at widely separated memory locations the hardware texture-cache will practically be useless.

7. Conclusions

We have described our approach to simultaneously developing for multiple high-performance graphics APIs, like OpenGL and Glide, using the computer game Parsec as a case study. We encapsulate all graphics functionality using three different subsystems. The VIDEO subsystem is responsible for managing video modes and establishing the connection between the operating system and the graphics API. The DRAWING subsystem is a low-level graphics-functionality abstraction used to draw individual primitives. It is very flexible and can easily be ported to additional graphics APIs, but involves a lot of run-time conversions which impact performance. The RENDERING subsystem is a high-level abstraction used to render entire objects. It is very fast, but requires a lot of effort for porting and is not as flexible as the primitive-level abstraction. One could compare the drawing subsystem to an immediate-mode approach, and the rendering subsystem to a retained-mode approach to rendering.

It is important to note that low-level is not synonymous with high-performance in our design. If the interface between portable and non-portable code is at a very low level, global optimizations are not easily possible. In contrast, if the interface is at a very high level, global optimizations are very easy to do. Furthermore, the implementation of a high-level subsystem can be extremely tailored to the actual underlying graphics API, whereas its low-level counterpart has the problem that it is only a thin layer above the API. Therefore, its structure is much more tightly bound to the abstract interface specification itself.

We think that it is crucial to provide different levels of abstraction when designing an abstract interface for the encapsulation of different graphics APIs. This makes it possible to combine the advantages of all of these different levels. Most importantly, flexibility and ease of use versus performance.

One approach we have adopted for implementing new types of objects and special effects, is to start out with a portable implementation using the drawing subsystem. If the new code proves to be performance-critical, it can be moved into the rendering subsystem at any time later on. For this reason, we sometimes have two implementations of object and special effects rendering code we can use, the slower but more portable one, and the faster one that requires more porting effort. For new target APIs the easily portable code can always be used first.

8. Acknowledgments

I would like to thank the entire Parsec team (Andreas Varga, Clemens Beer, Michael Wögerbauer, Alex Mastny, and Stefan Poiss) for their great work, and Michael Wimmer for proof-reading this paper and many useful suggestions.

9. References

[AMD] AMD web-site. See *http://www.amd.com*.

[DirectX] DirectX and Direct3D. See *http://www.microsoft.com/directx*.

[Glide] 3dfx Interactive Glide Rasterization Library. See *http://www.3dfx.com*.

[GLsite] OpenGL web-site. See *http://www.opengl.org*.

[Intel] Intel developer web-site. See *http://developer.intel.com*.

[Parsec] Parsec – Fast-paced multiplayer cross-platform 3D Internet space combat. See *http://www.parsec.org*.