FAKULTÄT FÜR !NFORMATIK

Visibility in a Real-World Cross-Platform Game Engine

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Stefan Reinalter
Matrikelnummer 0225790

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Oliver Mattausch

Wien, 28.03.2010

_____
(Unterschrift Verfasser/in)

_____
(Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Stefan Reinalter
Viktor Christ-Gasse 20/20
1050 Wien

"Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Ort, Datum:
Unterschrift:

Abstract

With hardware capabilities and customer expectations rising every new game console generation, efficient visibility algorithms become a more and more crucial part of every modern rendering engine. Although GPUs built into the consoles become better each generation, game developers are always striving to get more performance and better quality out of a game console. Therefore it is a must to employ powerful visibility algorithms which allow the developers to render more complex scenes while maintaining smooth framerates.

This thesis explores whether current state-of-the-art visibility algorithms can be used on game consoles, and describes the layers of abstraction needed in developing a multi-platform rendering engine.

Therefore, the first part of the thesis focuses on the design and implementation of a rendering engine for major current-gen platforms, such as Microsoft Windows, the Microsoft XBox360, Sony's PlayStation 3 and the Nintendo Wii, dealing with the vastly different platform architectures. Furthermore, solutions to engine design problems related to both hardware capabilities as well as software engineering practices are explored. Concluding the first part, prerequisites and building blocks for implementing visibility algorithms are developed.

The second part of the thesis concentrates on developing and integrating visibility algorithms into the aforementioned engine, building upon the components introduced in the first part. The used visibility algorithms are enhanced and tailored to specific hardware needs and capabilities, getting the most performance out of game consoles. Finally, results and possible improvements and future enhancements for current state-of-the-art algorithms conclude this thesis.

Kurzfassung

Aufgrund der immer besseren Hardwarefähigkeiten und steigenden Kundenerwartung, die jede neue Spielkonsolen-Generation mit sich bringt, spielen effiziente Sichtbarkeitsalgorithmen eine immer größere Rolle in der Entwicklung einer modernen Rendering Engine. Obwohl die in die Konsolen integrierten Grafikkarten mit jeder Generation besser werden, streben Spiele-Entwickler immer danach, mehr Performance und bessere Qualität aus der Hardware herauszuholen. Deshalb ist es absolut notwendig, leistungsfähige Sichtbarkeitsalgorithmen zu verwenden, die es erlauben, noch komplexere Szenen bei gleichbleibender, gleichmäßiger Bildwiederholrate zu rendern.

Diese Diplomarbeit erforscht, ob die derzeit gängigsten und modernsten Sichtbarkeitsalgorithmen auch auf Spielkonsolen verwendet werden können, und beschreibt dabei die dafür nötigen Abstraktionen in Bezug auf die Entwicklung einer Multi-Plattform Rendering Engine.

Demzufolge konzentriert sich der erste Teil dieser Diplomarbeit auf das Design und die Implementierung einer Rendering Engine für bedeutende "Current-Gen"-Plattformen, wie Microsoft Windows, die Microsoft XBox360, Sony's PlayStation 3 und die Nintendo Wii, und behandelt die erheblichen Unterschiede in der Architektur dieser Plattformen. Darüber hinaus werden Lösungen zu Design-Problemen untersucht, sowohl in Bezug auf die unterschiedlichen Leistungsmerkmale, als auch in Bezug auf die verwendeten Praktiken der Software-Entwicklung. Abschliessend werden Grundvoraussetzungen und Bausteine für die Implementierung von Sichtbarkeitsalgorithmen entwickelt.

Der zweite Teil der Arbeit konzentriert sich auf die Entwicklung und Integration von Sichtbarkeitsalgorithmen in die zuvor genannte Engine, aufbauend auf den Komponenten des ersten Teils. Die dabei verwendeten Sichtbarkeitsalgorithmen werden erweitert und speziell auf die Hardware der Konsolen zugeschnitten, sodass ein größtmöglicher Performancegewinn erzielt werden kann. Den Abschluss dieser Diplomarbeit bilden die erzielten Resultate, sowie Vorschläge zur Erweiterung und künftigen Weiterentwicklung derzeitiger, hochmoderner Sichtbarkeitsalgorithmen.

Contents

Chapter 1

Introduction

Today, game developers are always looking for solutions which allow them to render more geometry in less time, striving to get more and more performance out of each console generation, satisfying rising customer expectations. Visibility algorithms are one such solution, and this thesis explores whether current state-of-the-art visibility algorithms can be used on game consoles in a feasible manner, by detailing implementation issues faced when developing a multi-platform rendering engine.

The first chapter provides an overview of the thesis' scope and focus, introduces the Daedalus engine developed for this thesis, and shows the main contributions and structure of this thesis.

1.1 Scope of the work

This thesis deals with the design and implementation of a professional, multi-platform rendering engine for Microsoft Windows, Microsoft XBox360, Sony PlayStation 3 and Nintendo Wii, as well as the theory and implementation of state-of-the-art visibility algorithms integrated into this engine.

The first part of the thesis concerns itself with the development of the engine, which was built to be used by Austria's leading game development studio Sproing Interactive Media GmbH [65], serving as a replacement for the company's used engine at the time of writing, allowing Sproing to focus on building new technology, tools and games for today's major platforms. As such, it was extremely crucial that the developed engine fitted the company's requirements and allowed for easy integration into the existing code base and tool-chain.

The second part of this thesis deals with the integration of modern visibility algorithms into the engine and tool-chain, exploring new methods and ways in which the original algorithms can be enhanced, allowing for maximum performance on vastly different hardware.

1.2 Daedalus Engine

The Daedalus engine was developed over a period of two years from August 2007 to July 2009 in close collaboration with Sproing's technical director Gerhard Seiler, building upon the company's core technology and code base from day one. At the time of writing, the Daedalus engine is an integral part of Sproing's own Athena game engine, and is being used in the development of the studio's future multi-platform titles. The necessity for an in-house developed engine arose because Sproing is focusing on using own, custom-built technology in order to become more and more independent from middleware providers.

1.3 Main contributions

The following list provides an overview of the major contributions:

- Chapter 3 provides insights into professional multi-platform engine development, detailing information previously unpublished. Parts of the technology found in Daedalus have already been successfully put to use in shipped games, most notably Cursed Mountain [16], the company's prestige AAA-title that hit the shelves in August 2009.
- Chapter 4 introduces a novel mechanism for making hardware occlusion queries available on platforms which do not natively support them. Our hand-crafted occlusion queries use hardware GPU metrics, interrupts and custom-made CPU-GPU synchronization, resulting in queries that are way faster than query functionality on other platforms.
- Chapter 4 provides a new implementation of render state functionality using advanced C++ features such as template specialization. The implementation is very fast, robust and has an extremely small memory footprint.
- Chapter 4 details a multi-platform interface for immediate mode rendering, providing a device for fast rendering of ad-hoc primitives, optimized using special capabilities of each platform.
- Chapter 6 implements optimizations in parts of the hierarchical view-frustum culling algorithm using branch-free operations on console hardware.
- Chapter 6 establishes an optimized CHC++ algorithm using fine-tuned parameters on console hardware, a branch-free fixed-size stack and a

branch-free fixed-size queue which serve as a replacement for the standard STL containers used in CHC++, proving CHC++ to be feasible on console hardware.

Fig. 1.1: Daedalus technology used in Cursed Mountain. (Image courtesy of GameZONE)

1.4 Thesis structure

The thesis is structured into different chapters as follows:

- Chapter 2 explains the basics of real-time rendering and APIs (*Application Programming Interfaces*) used therein, works on the differences between game and rendering engines, and identifies requirements of a professional engine.
- Chapter 3 deals with the architecture of today's gaming platforms and their differences, general multi-platform code abstraction methodologies, different rendering schemes, and Daedalus specific design decisions.
- Chapter 4 builds on the design patterns introduced in the previous chapter, details some low-level implementations in the Daedalus engine,

which build the foundation for implementing visibility algorithms, and concludes the first part of the thesis with a discussion of how occlusion queries have been implemented in the engine.

- Chapter 5 introduces general visibility terminology and algorithms, and discusses related work on visibility algorithms.
- Chapter 6 shows the basic implementation of online occlusion culling algorithms in the Daedalus engine. Additionally, optimizations to these algorithms are introduced.
- Chapter 7 shows results that were achieved using the visibility algorithms built into the Daedalus engine, and provides a thorough evaluation and discussion of those results.
- Chapter 8 finally summarizes the thesis' contents, and concludes the second part of the thesis with possible future enhancements.

Chapter 2

Real-Time Rendering Engines

2.1 Real-Time Rendering

Rendering is concerned with generating an image from a 3d representation of a virtual world. *Real-Time Rendering* further expresses the fact that these images have to be generated in real-time by an application. Real-time in this context means that 15 or more frames are generated each second, otherwise the images appear to be flickering to the viewer [68]. The speed or performance of such real-time rendering applications is measured in *frames per second* (fps) or *Hertz* (Hz).

Although 15 fps begin to appear interactive, more than 30 or 60 fps are often desirable, especially in applications which have to convey a realistic appearance of the underlying virtual world, which is commonly the case in computer games. At a rate of 60 Hz, this means that a new image has to be generated approximately every 16 milliseconds. For applications which tax the hardware this is quite ambitious.

Compared to *offline* or *photorealistic* rendering, real-time rendering is not so concerned about getting *physically correct* results, but rather *physically plausible* ones in a much shorter time period. Therefore, real-time rendering applications often take shortcuts in rendering physical phenomena. These shortcuts are sometimes barely visible even to a trained eye, and certainly not to the average gamer.

Nonetheless, real-time rendering applications often use offline rendering for certain pre-processing tasks, e.g. baking static, view-independent lighting information into texture maps using state-of-the-art global illumination algorithms. These texture maps are then used at run-time, and can significantly improve the quality of pictures rendered in realtime. One such example of pre-computed lightmaps is shown in Figure 2.1.

However, these texture maps take up memory during rendering – memory which is very limited on consoles. Additionally, because these tasks often take hours or even days to complete, it might be less expensive to invest in a real-

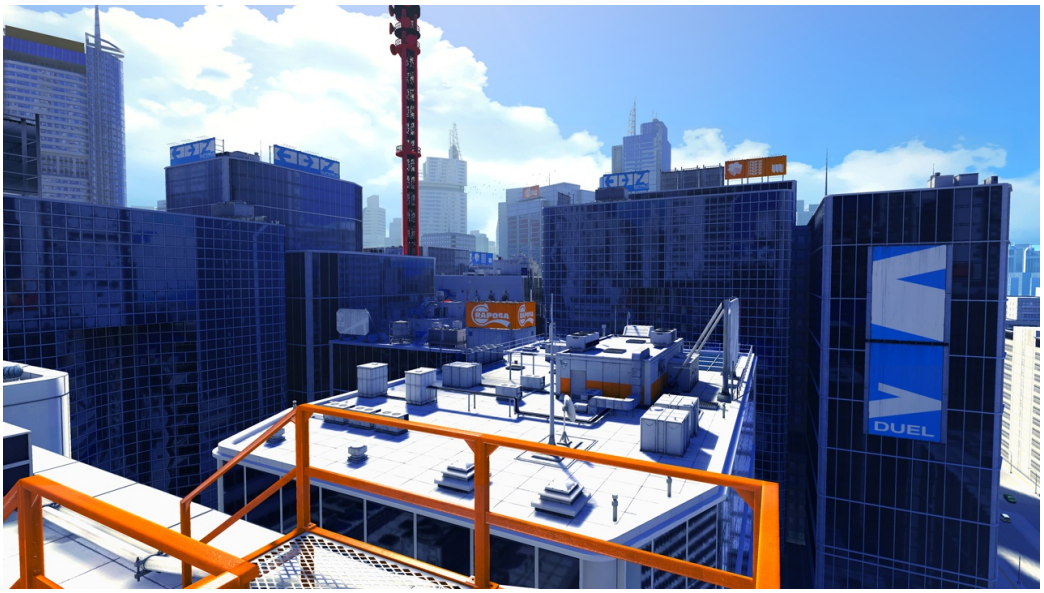


Fig. 2.1: Screenshot from Mirror's Edge, courtesy of EA Digital Illusions Creative Entertainment. Global Illumination baked with Beast [6].

time solution once or buy middleware, and settle with slightly worse quality, shortening turn-around times significantly. This cost-quality tradeoff has to be evaluated on a per-project basis.

Furthermore, as the gap between offline and real-time rendering closes further with every new hardware generation, these tasks will become more seldom. Figures 2.2(a) and 2.2(b) clearly show how much real-time rendering has advanced over the last five years.

Real-time rendering research is also one of the biggest research areas in computer graphics with game developers contributing papers to major conferences like SIGGRAPH [63] or Eurographics [23]. Of course, many of the algorithms seen in computer games today are heavily influenced or inspired by research done in that area.

2.2 Rendering pipeline

According to [68], the real-time rendering pipeline can be categorized into conceptual stages, functional stages and pipeline stages, as depicted in Figure 2.3.

Each stage can either be a pipeline (like the geometry stage), or can be parallelized (like the rasterizer stage). The application stage itself can be pipelined or parallelized, depending on the developer's implementation.

(a) Screenshot from Far Cry, 2004

(b) Screenshot from CryEngine 3, 2009

Fig. 2.2: Real-time rendering comparison

The stages itself exhibit the following properties:

- Application Stage: Completely executed on the CPU, the developer is

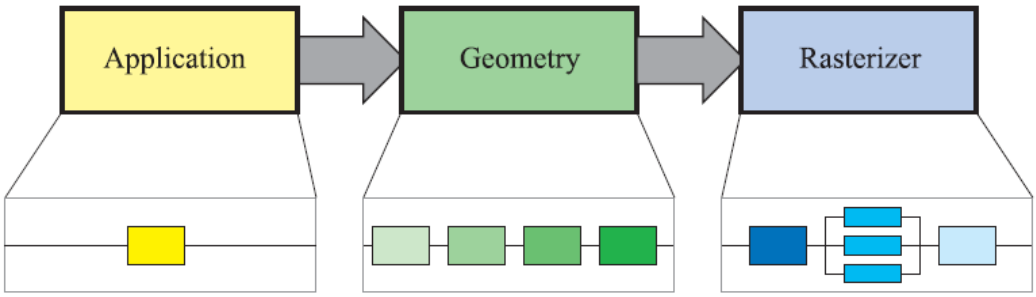


Fig. 2.3: Stages of the rendering pipeline. (Image courtesy of A.K. Peters Ltd., taken from "Real-Time Rendering, Third Edition" [2])

responsible for providing input to the geometry stage.

- Geometry Stage: Typically executed on the GPU, it takes care of most per-vertex operations.
- Rasterizer Stage: Completely executed on the GPU, this stage takes the geometry stage's input and outputs pixels on the screen.

An overview of the different conceptual stages will be given in the following subsections. A more thorough explanation of different stages and pipelines can be found in [68].

2.2.1 Application Stage

The implementation of the application stage is completely determined by the developer. He is responsible for generating rendering primitives (e.g. points, lines, triangles or quads) which are then sent to the geometry stage. Generating these primitives is the last and also the most important task of the application stage.

Usually, the application stage updates entities in a game according to user input, performs tasks such as collision detection, physics update, and visibility culling, and sends the remaining entities to the geometry stage.

2.2.2 Geometry Stage

The geometry stage is responsible for performing transformations from and to several 3d *spaces*. The most commonly used spaces are the *model space, world space* and *camera space*, which is also known as *view space* or *eye space*. Additionally, shading is computed for each input vertex, and projection, clipping and screen-mapping is performed on the resulting coordinates. These

coordinates consist of x- and y-coordinates, as well as a z-coordinate used for depth-testing.

In modern hardware, the geometry stage is completely implemented in the GPU, and parts of it can be programmed via *vertex shaders*.

2.2.3 Rasterizer Stage

The rasterizer stage interpolates various triangle attributes across a primitive using scan conversion, and performs pixel shading on each fragment. Pixel shading uses the interpolated properties as input, and produces one or more colors for each pixel, which are then passed on to the next stage. Most modern effects are implemented using pixel shaders.

Lastly, the *merging stage* of the rasterizer stage defines how the output of the pixel shading stage is combined with the color currently stored in the color buffer. Most parts of this stage are highly configurable, such as: Z-test, alpha-test, stencil-test, and ROPs (*Raster Operations*) or blend operations.

2.3 Graphics APIs

An API (*Application Programming Interface*) defines the software or source-code interface which is to be used by the developer in order to be able to use the underlying hardware, and as such defines a standard for accessing and programming hardware. Contrary to an ABI(*Application Binary Interface*), it does not expose any machine-code or operating-system interfaces, or details thereof.

A *graphics API* or *3d API* enforces a standard to be used by the developer when programming graphics hardware. Low-level functionality supported by the hardware is made available in the API, and the developer doesn't have to worry about hardware details - it is the API's responsibility to yield the same output on vastly different hardware. In addition to low-level functions, a 3d API may provide convenience functions and utilities to shorten development time.

Speaking in terms of the rendering pipeline, a graphics API allows the developer to implement the application stage, exchange programmable parts in the geometry and rasterizer stage such as vertex shaders and pixel shaders, and configure parts of the rasterizer stage, such as z-testing, alpha-testing or stencil-testing.

The following section shows an overview of popular 3d APIs, and additionally gives a brief bit of history.

2.3.1 Direct3D 9

Direct3D 9 is a subset of Microsoft's DirectX API [21], which serves as the standard API for developing multi-media applications for Microsoft Windows. The first version of DirectX 9 was made available in June 2004, and back then consisted of components for 2d and 3d rendering, playing back sounds and music, interfacing with input devices like keyboard, mouse, and joysticks, and networking and streaming video support. Of all components, Direct3D is by far the biggest and most widely used part of the API.

One of the innovations of Direct3D 9 in comparison to its predecessors is HLSL (*High Level Shading Language*), which allows developers to write graphical effects using a high-level language for vertex- and pixel-shaders instead of assembler code. Furthermore, DirectX 9 improved upon D3DX (*Direct3D Extension*), which is a high-level supplemental API for Direct3D 9. It exposes functions for common calculations with vectors, matrices, quaternions, and the like. Because D3DX is contained in DLLs, the APIs functionality can be improved with every new Direct3D update.

Today, DirectX 9 works on almost any hardware and Windows OS up until Windows XP.

2.3.2 Direct3D 10

Direct3D 10 [22] was concurrently introduced with Windows Vista [73], and featured the new Shader Model 4.0. Along other features, Shader Model 4.0 introduced the new geometry shader, which allows geometry to be generated within a shader, introducing completely new possibilities for common algorithms such as generating shadow volumes, generating fur, and rendering to cube maps.

Figure 2.4 shows the rendering pipeline introduced by Direct3D 10.

Additionally, Direct3D 10 defines a set of hardware capabilities which must be available on all Direct3D 10 compatible hardware. This lessens the burden on developers to write different codepaths for some hardware not capable of displaying certain effects and features, like it had to be done in DirectX 9. Of course, this means that games targeting DirectX 10 hardware will only run on Windows Vista and newer operating systems.

Some of the most noteworthy features of Direct3D 10 are the following:

- Full shader-only pipeline, fixed pipeline is no longer supported
- Shader model 4.0, new integer instructions, and geometry shaders
- Texture arrays

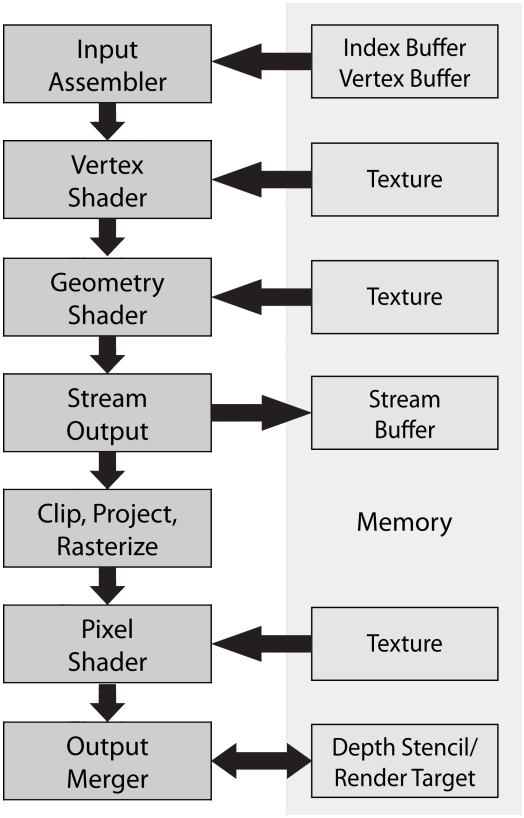


Fig. 2.4: Direct3D 10 rendering pipeline. (Illustration after [18])

- Predicated rendering
- Faster state changes
- Stream Output, which means that the Geometry Shader can output its results to both the rasterizer and the Input Assembler

Direct3D 10.1

Shortly after Direct3D 10 had been introduced, Microsoft announced Direct3D 10.1 which is now included with the first service pack for Windows Vista. Direct3D 10.1 is fully backwards-compatible with Direct3D 10, and adds the following features:

- More developer control over anti-aliasing (multisampling and supersampling)
- Shader model 4.1

- Mandatory 32-bit floating-point filtering

Direct3D 10.1 is only a minor update to Direct3D 10 and mainly adds parts of the specification which were not available at the time Direct3D 10 was introduced.

2.3.3 Direct3D 11

Direct3D 11 was first presented at Microsoft's Gamefest 2008 [27]. As of today, no specification has been made publicly available, but the following is a non-exhaustive list of features which will be introduced in Direct3D 11:

- Hardware supported tessellation
- True multi-threaded rendering: Draw calls can be submitted from several different threads
- Compute shaders: Allow to take advantage of the GPU's resources for computation-expensive tasks such as stream processing and physics calculations
- Shader model 5.0
- New texture compression formats better suited for high-dynamic range imaging

The first release candidate of Windows 7 is the first OS to provide Direct3D 11 support.

2.3.4 OpenGL

OpenGL (*Open Graphics Library*) [50] emerged from IrisGL, a standard developed by SGI (*Silicon Graphics International*) [62]. The OpenGL standard is defined by the OpenGL ARB (*Architecture Review Board*) [51] which consists of several industry big players like AMD/ATI, Dell, IBM, Intel, NVIDIA, and more. In 2006, control over the specification was transferred to the Khronos Group [38], a member-funded industry consortium.

One major aspect of OpenGL is its extensibility. New functionality can be added by graphics hardware vendors by exposing so-called extensions to the developer. Extensions to OpenGL typically follow the subsequent scheme:

- Vendor extensions: If an extension is exposed by one vendor only, the extension is appended with a particular postfix for this vendor, e.g. NV for NVIDIA[45].

- EXT extensions: If more vendors agree to expose the same functionality, an extension is postfixed with "EXT".
- ARB extensions: If the ARB agrees to standardize this extension, it is postfixed with "ARB".

Most of the standardized extensions by the ARB will sooner or later get integrated into the core of the OpenGL specification. From there on, the extensions' postfix is removed.

While on the one hand this mechanism keeps OpenGL extensible, on the other hand it can lead to what is known as "extension hell", where developers have to write different code paths for e.g. NVIDIA and AMD/ATI[3] cards. This is partly the reason why OpenGL is not so popular among game developers, the second reason being that OpenGL drivers often fall behind DirectX drivers in terms of functionality and stability.

Nevertheless, OpenGL is widely used in professional 3d software, CAD applications, flight simulators, and visualization because it still is the only platform-independent standard. OpenGL is available for a multitude of platforms.

2.4 Rendering engines

A *Rendering engine* is what enables developers to write programs using the graphics hardware, without having to deal with platform or API details. It abstracts those details and allows the developer to run the same rendering code on different platforms. A rendering engine is also commonly being referred to as *3d engine* or *graphics engine*.

Such an engine provides low-level functionality such as rendering primitives, issuing occlusion queries, switching render targets, changing render states and much more. Ideally, it is possible to exchange some of those low-level implementations with own code, or at least enhance the engine's low-level functionality.

Additionally, high-level features allow the developer to use e.g. lighting effects, shadowing algorithms, post-processing effects and many more with only a few lines of code out of the box. New features or rendering effects can be integrated into the engine by the developer using either low-level functionality or building on top of already existing high-level code.

Furthermore, a good rendering engine provides spatial data structures, visibility algorithms and other high-level algorithms and building blocks with which the developer can build games of a vast amount of different genres, be it first-person shooters, racing games, indoors or outdoors. Any good

3d engine should not focus on one specific scenario in games development, although even many commercially available engines pose this restriction. In essence, a well-engineered 3d engine should not dictate the developer how to use it, but rather let him use his own workflow and tools, and allow him to leverage the engine's feature set with ease.

Conceptually, a rendering engine is a subset of a game engine, implementing a game engine's rendering module. A 3d engine usually does not concern itself with core features such as file I/O, debugging functionality, but rather builds upon a common codebase, or lets the developer plug in his own core routines. Furthermore, a strict rendering engine doesn't provide tools such as an editor or world builder. However, many engines don't adhere to such a strict distinction and provide a complete tool-chain, exporters for commercial art packages, and other tools - which is not necessarily a bad thing.

2.4.1 OGRE - Open Source 3D Graphics Engine

Today, OGRE [49] is most probably the best known open source graphics engine. Started in 2000, it is being used by amateur game developers and universities for small projects as well as bigger ones, be it games, scientific visualization, or others. One of the positive aspects of OGRE is that it provides support for many advanced effects, for example different shadowing algorithms, hence the learning curve for starting developers is not very steep. A well-established forum and a big community, as well as exporters for both commercial and open-source art packages like Maya and Blender make it even more appealing for hobby developers.

However, OGRE severely lacks in the extensibility department. Altering low-level rendering features or extending high-level functionality such as the renderer is either impossible or only doable with a huge amount of work. Furthermore, the engine's abstraction facilities rely way to much on inheritance and abstract interfaces, making it a bad choice for commercial multi-platform development because of the negative performance impact this poses on today's game consoles.

OGRE is available for Microsoft Windows, Linux, and Apple Mac OSX.

2.4.2 Irrlicht

Like OGRE, the open-source engine Irrlicht [37] is also very popular among hobby game developers. Irrlicht comes with a lot of nice features, and is extremely easy to use, even for beginners in the 3d graphics area. A simple demo containing animated models, lighting, and a FPS-like camera can be

built with approximately 100 lines of code. Irrlicht claims to be lightning fast, in comparison to commercial engines this statement however is not true.

Almost all of the engine's low-level features are abstracted like in OGRE using abstract interfaces and virtual functions, hence suffering the same shortcomings from a commercial development point-of-view.

Irrlicht is available for Microsoft Windows, Linux, Apple Mac OSX, and Sun Solaris.

2.4.3 Open Scene-Graph

Open Scene-Graph [52] is another open source rendering toolkit mainly used for visual simulation, scientific visualization and modelling. Open Scene-Graph is implemented on top of OpenGL, and therefore supports a multitude of different operating systems.

The engine itself provides a clean and generic abstraction of scene graphs, and supports many features like shadow generation, visibility culling, and particle systems out of the box. Additionally, it supports advanced technology like multi-threaded rendering and support for large terrain data.

2.5 Game Engines

A *Game engine* is the complete driving force behind game technology, and usually consists of several parts or engines itself. These parts can be any of the following:

- Core technology: Memory management, file I/O, debugging, exception handling, threading, serialization, containers such as arrays, vectors, lists, maps, ...
- Rendering engine: Examples would be the **Irrlicht Engine** [37], **OGRE** [49], and **Open Scene-Graph** [52].
- Physics engine: Examples would be **Havok** [34], **PhysX** [55], and **ODE** [47].
- Audio engine: Examples would be **Miles Sound System** [43] and **FMOD** [25].
- Networking engine: Examples would be **RakNet** [60] and **GNE** [29].
- Artificial intelligence technology: An example would be AI Implant [1].
- Scripting language: Examples would be **LUA** [39] or **Squirrel** [66].

- Real-Time editor: Examples would be **CryEngine's Sandbox** [61] or **Valve's Hammer** [32].
- Exporters for commercial art packages.
- Technology for automatically parallelizing tasks on multi-core CPUs or offloading code to the SPUs: Examples would be **Emergent's Floodgate** [24] and **Codeplay's Offload** [48].

Using a game engine allows the developer to focus almost completely on the game and game play itself, saving time and money during the development process. Examples of such game engines which provide almost all of the above would be **Epic's Unreal Engine 3** [69], **Crytek's CryEngine 3** [15], **Valve's Source Engine** [64] and **Emergent's Gamebryo** [26]. A complete list of engines used in commercial game development can be found in [17].

2.5.1 Unreal Engine 3

Being in its third version, this engine is hands down the most widely used in the games industry. Many of today's AAA games leverage Epic's technology to the fullest. One area where the Unreal Engine 3 absolutely shines is in the content creation tools, giving complete control to artists, designers and scripters. This amount of control is essential in professional games development, where many areas of a game sometimes have to be fine-tuned because otherwise the engine would not be able to render the scene in real-time. Additionally this allows the developer to fake certain graphical effects to the maximum possible extent.

Unreal Engine 3 is available for Microsoft Windows, Microsoft XBox360 and Sony PlayStation 3, with older versions of the engine supporting many more platforms.

2.5.2 CryEngine 3

The third version of the CryEngine is the first multi-platform version of this engine, with previous versions being for PC only. Like the CryEngine 2 did on the PC platform, CryEngine 3 defines a new graphical standard for games running on Microsoft Windows, Microsoft XBox360 and Sony PlayStation 3 with unprecedented visual quality. In addition, CryEngine 3 also offers various built-in editing tools for assets, animation, physics, and more. Being available for consoles only recently, it still has to prove itself as true competitor to the Unreal Engine 3.

2.5.3 Gamebryo

Emergent's Gamebryo is probably the second biggest game engine used in the industry. Featuring tools and a complete framework for rapid prototyping, game prototypes can be built in a fraction of the time needed with other engines. Unlike Unreal Engine 3 and CryEngine 3, Gamebryo also supports the Nintendo Wii out-of-the-box, which is the console with the largest installed userbase of any of today's consoles except the PlayStation 2.

Gamebryo is available for Microsoft Windows, Microsoft XBox360, Sony PlayStation 3 and Nintendo Wii .

2.6 Rendering Engine Requirements

Concluding this chapter, a list of requirements of a rendering engine intended to be used for professional game development is gathered.

- Usability: The engine should be easy to use, and hard to misuse – both for junior as well as senior programmers and experts in the field. If possible, misuse should be flagged at compile-time rather than run-time.

- Cross-platform: The engine should provide the same interface for all platforms. Different platform technologies (e.g. SPU stream processing on the PlayStation 3 versus multi-core programming on the Xbox360 versus single-threaded programming on the Wii) should be abstracted into the same interface.

- Performance: The engine should yield the best possible performance by having a small memory footprint, avoiding memory allocations, parallelizing tasks, and making use of dedicated hardware. If parts of the engine are not optimized, it should be easy for the developer to exchange that part with his own implementation, possibly gaining performance.

- Scalability: It should be the developer's choice which parts of an engine he wants to use. The engine should not dictate a certain work-flow to the developer, but rather support him in developing his own. The engine should provide basic building blocks which allow the user to develop his own high-level rendering algorithms.

- Modularity: It should be possible to completely exchange parts of the engine with own implementations, while still keeping the engine's functionality intact.

- Prototyping support: The engine should provide prototyping facilities which enable the developer to prototype features, effects and algorithms quickly and easily.
- Accessibility: The developer should have access to all low-level functionality exposed by the platform. If some part of the platform's API is not abstracted in the engine, it should be possible to work directly with the platform API.
- Comprehensibility: The engine should not require an expert- or guru-level programmer in order to fully understand it. The code should adhere to an easy-to-understand coding style.
- Documentation: Each single part of the engine should be documented in a clear and concise fashion. Furthermore there should be a documentation about how the engine works as a whole, so the developer is able to understand interdependencies between different parts.

Daedalus, the rendering engine developed for this thesis, was built with all these requirements in mind, and Chapter 3 and 4 show how these requirements were fulfilled.

Chapter 3

Introduction to cross-platform design

This section discusses design and code abstraction necessities when working on multiple platforms, deals with general multi-platform development problems, shows differences between the platforms' rendering APIs, and touches upon multi-platform rendering, cache coherency, CPU-GPU synchronization and multi-threaded rendering. Technical implementation details are discussed thoroughly in chapter 4 of this thesis.

3.1 Platform capabilities

First and foremost it is exceptionally important to know each platform's feature set, its API, and its bugs and quirks. It is not unusual that game consoles exhibit hardware bugs, which at best are worked around in the SDK. If not, the developer has to pay attention to not trigger these bugs.

As a bare minimum, each platform's hardware capabilities and architecture should be known before getting started. Although code can be refactored and retrofitted at a later point during development, this often results in suboptimal performance and degraded code quality. To some extent, code refactoring always has to happen sooner or later, but it is best to get intricate details sorted out in the beginning.

3.1.1 PC platform

Of all the platforms described in this section, the PC is clearly the easiest to develop for. At times, developing for a vast amount of different hardware and operating systems can be harder than programming for fixed-hardware platforms, but this flaw is easily outweighed by the fact that there is a wealth of information, a large array of debugging tools and profiling tools available, and turn-around times are the shortest among major platforms.

The following is a collection of the PC platform's characteristics:

- Multi-Core: With multi-core systems gaining popularity even in entry-level desktop systems, the PC can generally be classified as a multi-core platform. Nowadays, the average desktop PC consists of two cores, whereas hardcore gamers own four-core or even eight-core systems.
- GPU memory: The GPU has its own memory dedicated to vertex buffers, index buffers, textures, and rendering resources in general. According to the latest hardware survey by game developer Valve [70], an average PC has at least 256 MB graphics memory for the GPU to work with.
- CPU-GPU synchronization: The graphics API takes care of CPU-GPU synchronization.
- Cache coherency: The operating system and graphics API take care of invalidating instruction and data cache, and vertex buffer and texture cache, respectively.
- Unified shader architecture: Since Direct3D 10, shaders are unified, yielding better performance in vertex- or pixel-shader bound scenes, because more shader computing units can be allocated to either vertex- or pixel-shaders, depending on their workload.
- GPU main memory access: Since Direct3D 10, it is possible to store primitives generated by the geometry shader in a buffer accessible by the GPU, using Stream Out. The GPU accesses graphics memory via PCI-E [53], which is the de-facto standard interface between the GPU and the CPU on the PC.

Figure 3.1 depicts a simplified design for a dual-core PC.

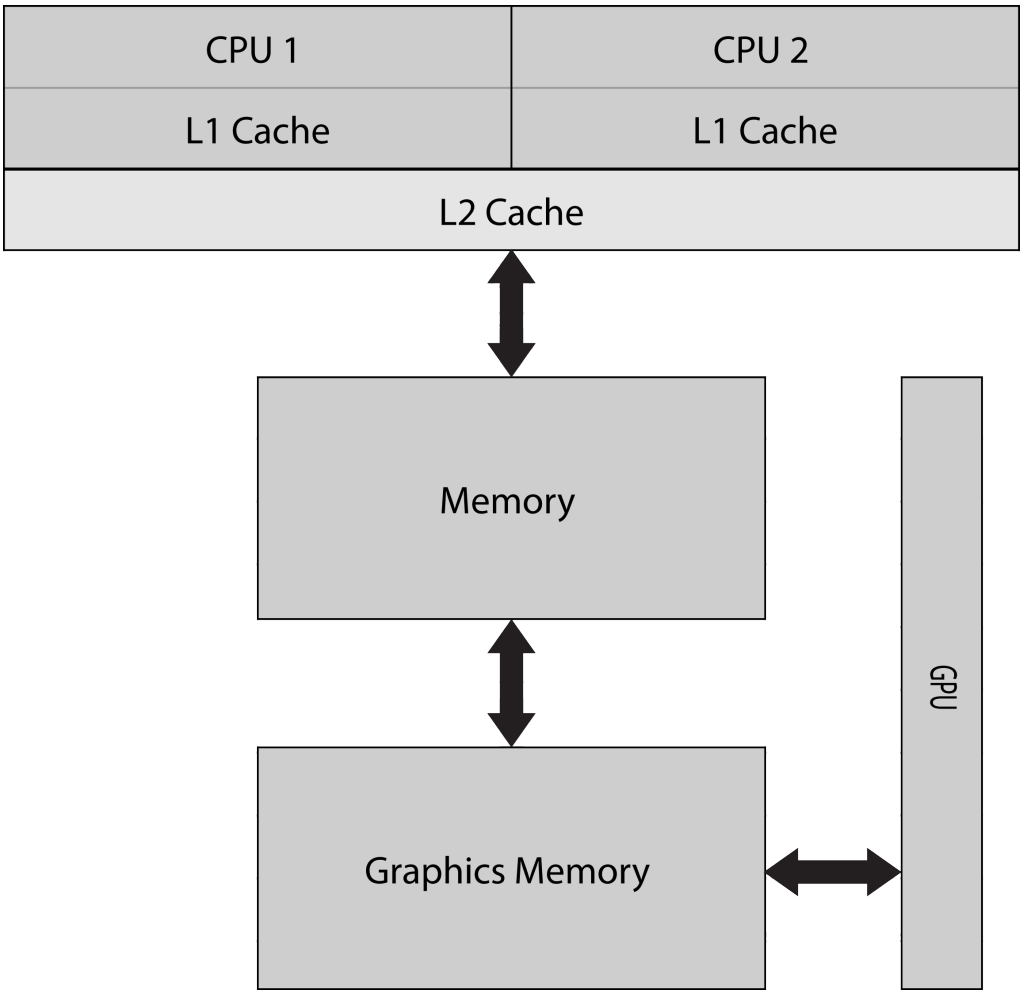


Fig. 3.1: Simplified overview of the PC architecture for a dual-core PC.

3.1.2 Xbox360 platform

On the CPU-side, the Xbox360 platform features the so-called *Xenon CPU* based on IBM's PowerPC, consisting of 3 cores with 2 hardware threads each. Even though all cores can be used by the developer, some hardware threads have to share their resources with API specific tasks like sound decoding and mixing, and OS related activities.

GPU-wise, the Xbox360 features *ATI's Xenos GPU* [20], which is the predecessor of ATI's R600 graphics card series. However, there are many differences compared to a PC graphics card.

The Xenos GPU consists of two separate silicon dies: The main GPU chip and its daughter chip. The main GPU chip also serves as a memory controller, which means that the CPU accesses system memory through the GPU chip, which essentially leads to a *Unified Memory Architecture*. Furthermore, the GPU is connected to the CPU's L2 cache, of which parts can be locked to be accessible by the GPU (shown in Figure 3.2).

The main GPU supports a superset of Shader Model 3.0 with additional features such as MEMEXPORT, which allows the GPU to write directly into main memory by using dedicated assembler instructions in vertex shaders. Additionally, the Xenos GPU is built upon a Unified Shader Architecture supported by an additional load-balancing hardware unit, and also offers a dedicated hardware tesselation unit.

The GPU's daughter chip features 10 Megabytes of *embedded DRAM* (eDRAM) and own dedicated logic for anti-aliasing, alpha- and depth-testing, and other related operations. Because the GPU can only render into the eDRAM, all frame buffers (color buffers, z-buffer and hierarchical z-buffer, render targets) have to reside in the eDRAM. Render targets and the back-buffer have to be resolved from eDRAM into main memory before they can be used as textures or the frontbuffer by the GPU.

The design of the daughter chip brings both advantages and disadvan-

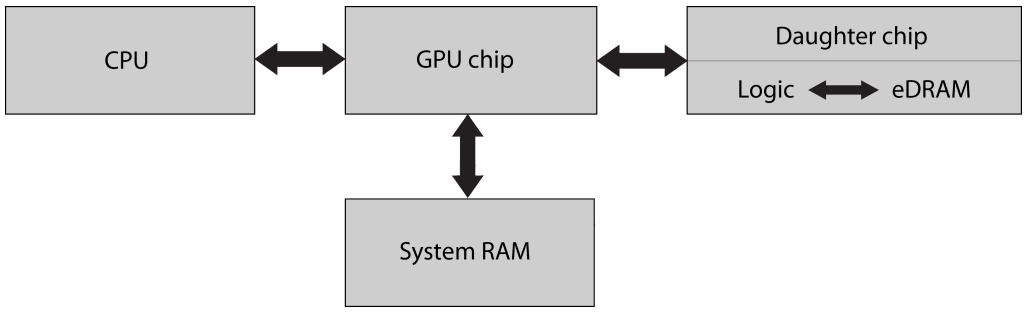


Fig. 3.2: Xbox360 Xenos GPU

tages. Because multi-sampling is handled by dedicated logic in the daughter chip, even 4xMSAA at 720p or 1080i does not cause significant performance drops. Furthermore, rendering to eDRAM and resolving into main memory is extremely fast because of the bandwidth between the main chip and the daughter chip.

However, the eDRAM's limited size of 10 megabytes poses a problem when rendering in HD resolutions. For example, 4xMSAA frame buffers do not fit into eDRAM at once, hence rendering needs to be done on different tiles with intermediate results being resolved to main memory in between. This can cause problems when rendering certain full-screen effects.

Figure 3.3 depicts a simplified design of the Xbox360.

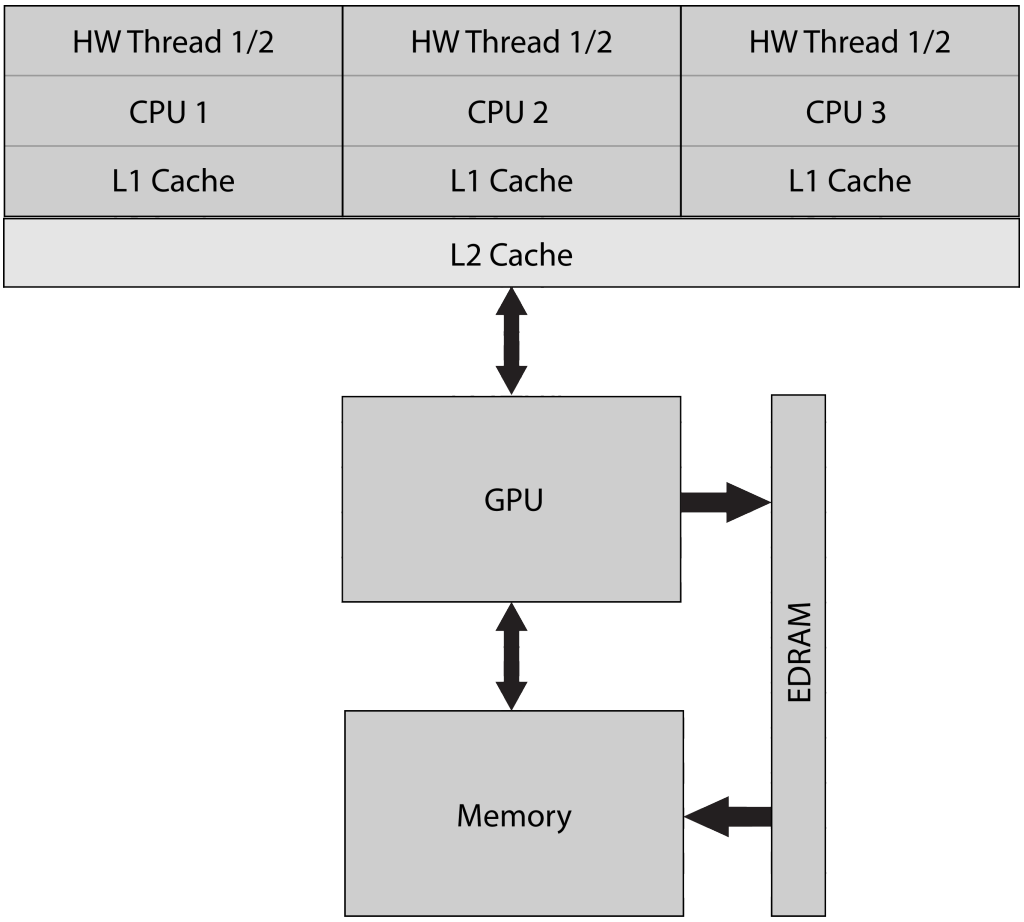


Fig. 3.3: Simplified overview of the Xbox360 architecture.

3.1.3 PlayStation 3 platform

The PlayStation 3 features the *Cell Broadband Engine* (CBE) [11] as its CPU, and NVIDIA's *RSX* as its GPU. The RSX is basically a modified Geforce 7800, fully supporting Shader Model 3.0. CPU and GPU are equipped with 256 MB of XDR and GDDR3 memory, respectively, and both of them can access each other's memory via the FlexIO interface which connects the CBE with the RSX. While the RSX reading from main memory is supported at full bandwidth, reading from RSX memory on the CBE is the slowest path in the pipeline, and should be avoided if possible.

The CBE consists of a *PowerPC Processor Element* (PPE) and eight additional *Synergistic Processor Elements* (SPE). The PPE itself offers two hardware threads, and serves as the part of the CPU which runs general-purpose code, and distributes computations to the SPEs. Even though the CBE consists of eight SPEs, only six of them are available for development. One SPE is disabled in order to increase die yields, and one SPE is reserved by the OS. The PPE, SPEs and the FlexIO interface are connected through the *Element Interconnect Bus* (EIB), which is an extremely fast bus with more than 200 GB/s bandwidth [59].

The SPEs itself each consist of a memory controller and a *Synergistic Processor Unit* (SPU). Each SPU is built as a pure SIMD-processor, and can be seen as a heterogeneous processing unit optimized for localized computations, bearing resemblance to the stream processing units in GPU graphics pipelines. As such, an SPU should not be used for general-purpose code, but rather as a processor acting on streams of data, in small pieces of data at a time.

SPUs cannot access main memory or any other memory other than its own *Local Storage*, which is 256 KB in size. Data from the local storage of other SPUs or data from main memory must first be transferred via DMA. Each SPU can handle multiple such DMA transfers in parallel to computation.

In order to leverage the full potential of the PlayStation 3, the SPEs must be used to the fullest extent possible. This, however, brings a paradigm shift for developers because code which was previously executed on GPUs exclusively now has to be written for the SPEs. As an example, developers use the SPEs for skinning, backface culling, and fullscreen effects in order to free some milliseconds off of the RSX. Speaking in terms of the rendering pipeline, the SPEs can be used to implement parts of the geometry stage, essentially making the PlayStation 3 a hybrid CPU-GPU rendering architecture.

More information about the CBE and the PlayStation 3 graphics pipeline can be found at [11], [41] and [54].

Figure 3.4 depicts a simplified design of the PlayStation 3.

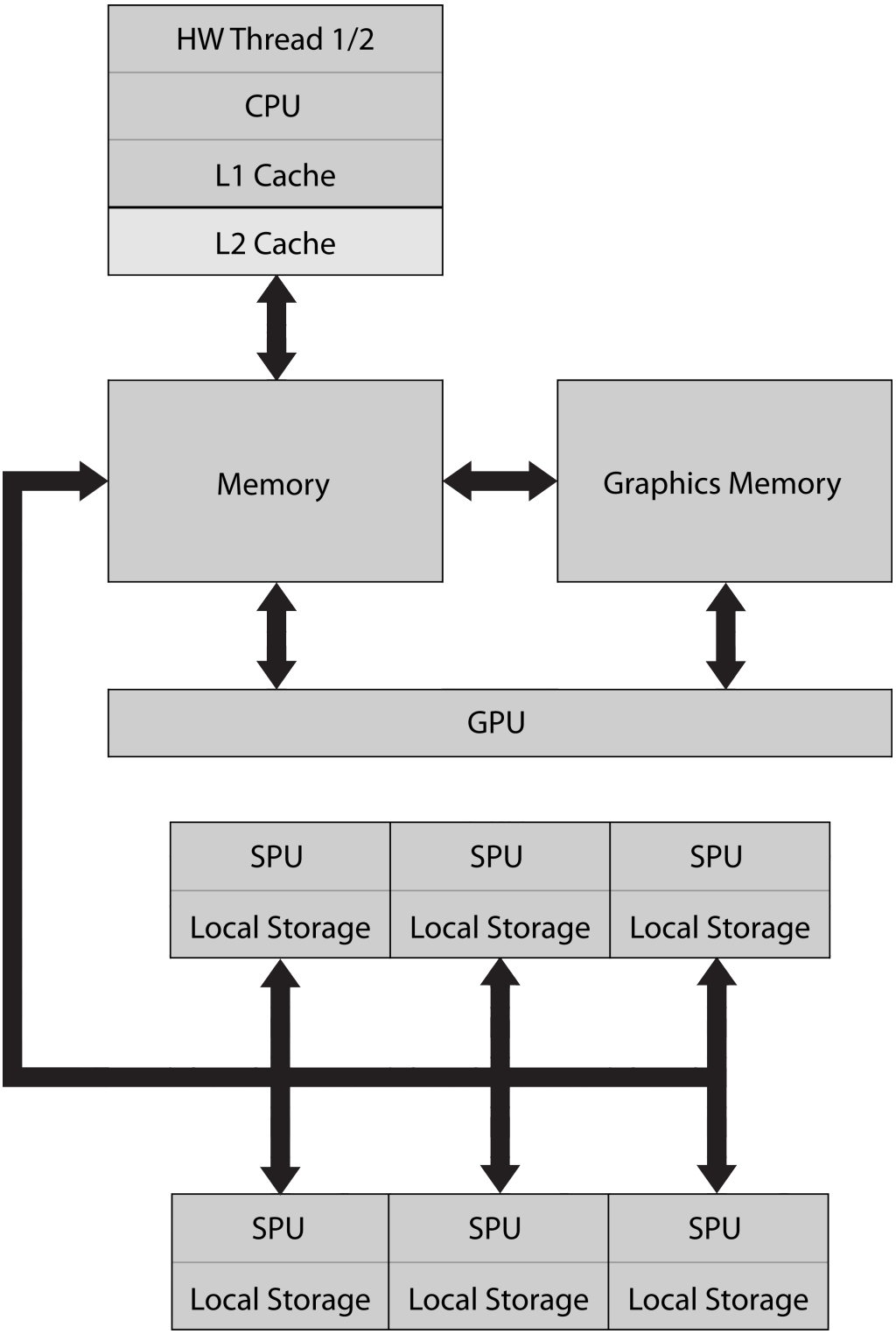


Fig. 3.4: Simplified overview of the PlayStation 3 architecture.

3.1.4 Wii platform

Originally codenamed *Revolution*, the Nintendo Wii features a PowerPC-based *Broadway* processor developed by IBM and Nintendo as CPU, and *ATI's Hollywood* as GPU.

Wii offers 24 MB of internal memory called MEM1, and 64 MB of external GDDR3 memory called MEM2. Both MEM1 and MEM2 can be accessed by the CPU and the GPU, but most of the time MEM1 is used for code and general data structures, while MEM2 is being used for vertex data, index data, and textures.

The CPU is the only CPU of all current-generation consoles which offers neither multiple cores nor multiple threads.

The GPU itself is similar to the GPU found in the Nintendo GameCube, but is approximately 1.5 or 2 times as fast as its precursor. Of all the current-generation GPUs, Hollywood is the only one which doesn't support shaders by using a high-level shading language or something similar, but rather uses a fixed-function pipeline. Nevertheless, developers can program up to 16 Texture Environment (TEV) Stages and an indirect texturing unit, allowing them to simulate nearly a full Shader Model 2.0 feature set. Programming these TEV stages is somewhat similar to programming the early NVIDIA register combiners [46].

Figure 3.5 depicts a simplified design of the Wii.

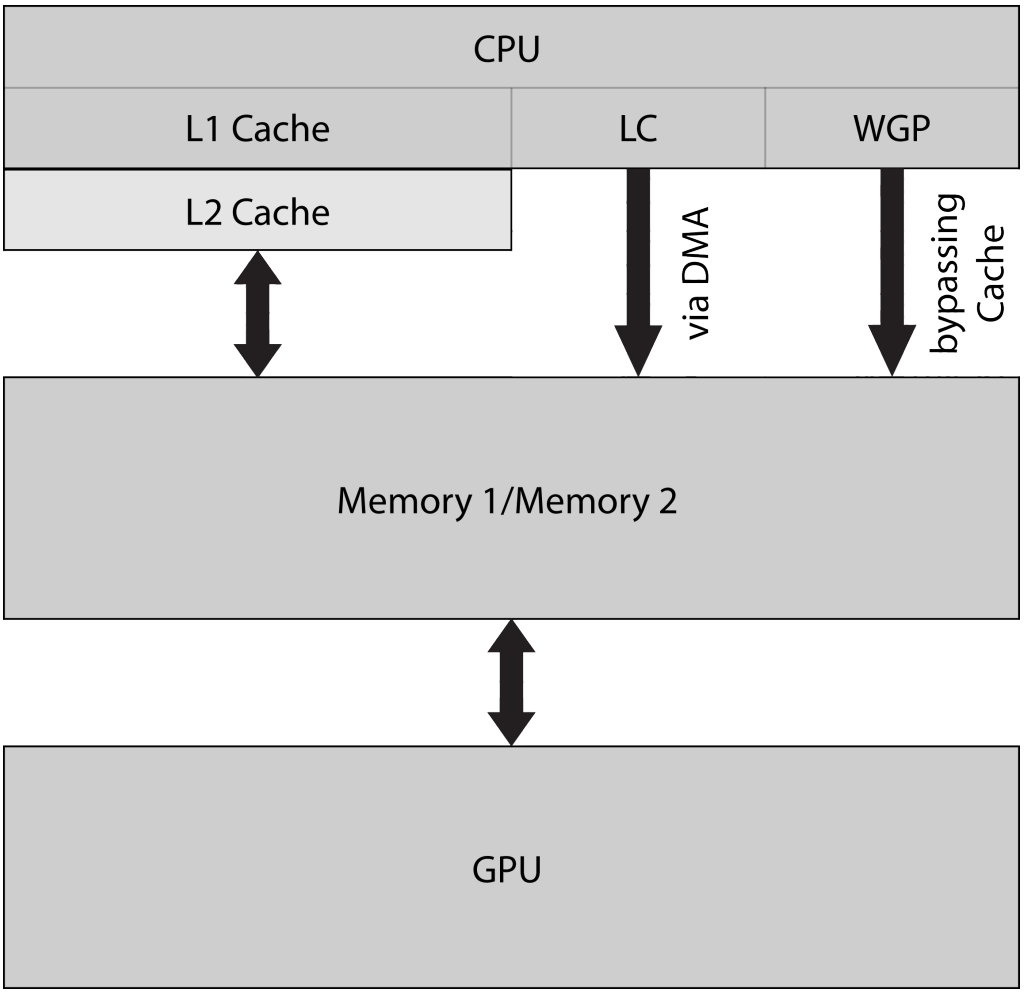


Fig. 3.5: Simplified overview of the Wii architecture.

3.2 Separation of low-level and high-level code

One of the major design aspects when writing multi-platform code is to clearly separate low-level from high-level code.

Low-level code can be thought of as the inner core of an engine which has to abstract the different hardware capabilities and APIs of each platform behind a clear interface, which in turn is used by either the client using the engine or the engine's high-level code itself. Perfect examples of low-level code would be classes responsible for abstracting render states, occlusion queries, vertex and index buffers, and so forth.

High-level code builds upon the engine's low-level code and will almost always share one common implementation on every platform. Some examples of high-level code would be classes implementing shaders, materials, and scene graphs. High-level code should only be written in a platform-dependent manner for optimization purposes, where parts of high-level code are implemented differently in order to gain performance.

3.3 Granting access to platform-dependent functionality

In addition to having a cleanly abstracted low-level interface, it must be possible for clients to make use of platform-specific capabilities, e.g. different render states which are only available on certain hardware. Basically, there are two main methodologies which can be leveraged to accomplish this:

- Add the functionality to the platform-independent interface, and leave the platform-dependent parts empty on platforms where the functionality is not supported.
- Leave it to the user to use preprocessor #ifs in his own code to explicitly state that access to such parts is wanted. An example is given in Listing 3.1.

Daedalus exclusively uses the latter variant because it makes it much more obvious where access to platform-dependent parts is needed. Furthermore, for some platform-exclusive functionality like render states it is simply not possible to implement the first strategy.

Some graphics engines even grant the user access to inner details such as the Direct3D device, but Daedalus restrains itself from doing so. This strategy can completely wreak havoc with the inner workings of an engine, and should therefore be avoided at all costs.

```
{
  RenderState noCull(RenderStates::CULL_MODE_NONE);

#if WINDOWS
  // fog done in Shader on other platforms
  RenderState fog(RenderStates::FOG_LINEAR);
#endif

  // rendering code
  ...
}
```

Listing 3.1: Client code using #if

3.4 Ease of use

Quoting the C++ language's designer Bjarne Stroustrup:

> "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off",

it is essential that an engine (or library, for that matter) makes it hard for the user to use it in a wrong way. There are several techniques for accomplishing this feat, sorted in descending order of their usefulness:

- Disallow misuse by hiding as much code as possible behind protected and private interfaces: The compiler will make sure that neither protected nor private interface parts are used wrongly, and error messages are clean and concise.
- Flag errors at compile-time (e.g. by employing static assertions and templates): Compile-time errors caused by static assertions should be made easy to understand.
- Flag errors at run-time (e.g. by employing assertions and sanity checks): As much detail as possible as to when, how, and what went wrong should be provided.
- Generate run-time warnings and errors: Detection of misuse should go alongside displaying readable warning or error messages.
- Gracefully abort the current operation at critical errors: If something unexpected happens, the engine should gracefully abandon its current operation, inform the user, but just not crash silently.

Keeping the above in mind, sometimes it is simply not possible or undesirable to design code for absolute ease of use, e.g. when dealing with high-performance code. Nevertheless, trading performance for possibly hard-to-find bugs should only seldom be necessary.

3.5 Judicious use of language features

Far too often, language features are used in wrong ways. Such abused C++ features are inheritance and virtual functions, for example. Granted, polymorphism and dynamic binding are very useful language features which have their uses, but they are often utilized for the wrong task. Dynamic polymorphism should be used when dealing with derived classes which can coexist at run-time and one needs to work with base class interfaces, nonetheless this kind of polymorphism is often used for abstracting different platforms behind interfaces. C++ supports advanced language features like templates which are often more suited for this task. Especially static polymorphism and the *Pimpl Idiom* or *Bridge Pattern* prove to be useful in multi-platform development as can be seen in Section 4.

More C++ idioms can be found in [28] and at [14].

3.6 Abstracting low-level interfaces

Undoubtedly the single biggest mistake in multi-platform development is abstracting different platforms by using interfaces with solely pure virtual functions, which are implemented in concrete classes for each platform. From an object-oriented programming perspective this may be arguable, but considering both the performance penalty on today's hardware and the fact that **none** of these interfaces can co-exist at run-time, this is not practical.

There are many hobbyist and amateur engines which suffer heavily from this problem, putting more than 100 virtual functions in an abstract interface.

It should be understood that the penalty of invoking virtual functions stems not only from the fact that an extra level of indirection is added upon each call, but rather introduces several more penalties:

- Virtual function calls cannot be inlined: Even the best compilers and aggressive optimizers cannot inline virtual function calls. This is only possible in very rare circumstances.
- Possible instruction and/or data cache miss: Both types of cache misses are amongst the number one performance hits on modern hardware.

- Possible pipeline hazard because of mispredicted branches: Due to the fact that virtual functions cannot be inlined, mispredicted branches are more likely to happen. Branch prediction on a PowerPC based architecture is different to the x86 architecture, hence the performance impact on console hardware is higher because every platform is PowerPC based. The PlayStation 3 SPUs do not have any branch prediction at all.

- Virtual functions cannot be run on an SPU without requiring an awful lot of work.

That being said, it should be noted that virtual function calls should **not** be avoided at all costs – interfaces and virtual functions can increase productivity and make code more reusable, they should just not be used carelessly. Emulating virtual functions almost always results in a slower version of what the compiler can do, and hence should be avoided.

Fortunately, there are several solutions to the problem of having to provide one interface to the user, but having different implementations for each platform. None of these solutions require virtual functions.

3.6.1 Preprocessor directives

Perhaps the most simple solution is to implement all interfaces in the same file, with each implementation being wrapped with preprocessor directives for inclusion/exclusion of the respective code for each platform.

Although this solution solves the original problem of not relying on abstract interfaces, it clearly introduces some new, serious problems:

- Unreadable, unmaintainable code: Every method is cluttered with ugly preprocessor directives.

- Possible code duplication for some platforms.

- Tedious to add new platforms: New platforms must be added to every method and file.

- Missing implementations are tedious to flag at compile-time: Respective preprocessor commands would have to be put into every method and file.

```
class Interface
{
  ...

  void Method(void)
  {
#if _WINDOWS
  // Windows code here
#elif _XBOX360
  // XBox360 code here
#elif _PS3
  // PlayStation 3 code here
#elif _WII
  // Wii code here
#endif
  }
};
```

Listing 3.2: Using preprocessor directives in a single file

3.6.2 Splitting interfaces across files

A better solution is to split the different implementations, put them into separate files and keep them in distinct folders. Preprocessor directives are only used in the header file which is being used in client code:

```
#if _WINDOWS
        #include "InterfaceWindows.h"
#elif _XBOX360
        #include "InterfaceXBox360.h"
#elif _PS3
        #include "InterfacePS3.h"
#elif _WII
        #include "InterfaceWii.h"
#else
        #error "Unknown platform!"
#endif
```

Listing 3.3: Header file used by client code

Each one of the *Interface*.h* files contains the implementation of the respective interface for **one** platform. This solution offers several major advantages over the first one:

- Clear code.

- Clear separation of platforms.
- New platforms can be added easily, missing platforms can be flagged at compile-time.

This solution is perfectly valid when an interface's implementation proves to be completely different for each and every platform. However, most of the time interfaces or classes need a more sophisticated solution because they consist of both a platform-independent and a platform-dependent part. Using the above solution in such circumstances would result in unnecessary code duplication distributed over several implementations. This can be tedious to find and refactor when used in huge amounts of code.

There is always the idea to put platform-independent parts into a base class, and let platform-dependent parts derive from that class. However, this solution is sometimes not feasible because of the following reasons:

- If the base class needs a virtual destructor, a virtual function table is added for every instance of the class. This can be undesired for serialization, in-place loading, and so forth.
- Adding overloads in the derived class can lead to unwanted function hiding.
- If other classes such as utility classes need to work with base-class pointers, they can never access the platform-dependent parts, unless ugly casts are used.
- As a general rule, inheritance should model an "is-a" relationship. Inheritance should not be abused to inherit common functionality.

3.6.3 Modified PIMPL idiom

A more elegant solution is to utilize the PIMPL *(Private Implementation)* idiom [56], and modify it by passing the implementation as a template parameter to the interface, which in turn now becomes a class template. This is illustrated with the simplified Texture class shown in Listing 3.4.

As can be seen, platform-dependent methods simply delegate their work to the given implementation. By using a template parameter, the implementation is subject to static binding, hence the compiler will inline all these calls. The implementation files for each platform additionally contain a typedef which makes the template-class completely transparent to the user, as shown in Listing 3.5.

```
template <class Impl>
class TextureBase
{
public:
        ...

        // Platform-independent interface
        unsigned int GetWidth(void) const { return width; }
        unsigned int GetHeight(void) const { return height; }

        // Platform-dependent interface
        void Lock(void) { impl->Lock(); }
        void Unlock(void) { impl->Unlock(); }

        ...

private:
        ...
        unsigned int width;
        unsigned int height;
        ...
        Impl* impl;
};
```

Listing 3.4: Modified PIMPL

```
// In TextureWindows.h:
typedef TextureBase<TextureImplWindows> Texture;

...

// In TexturePS3.h:
typedef TextureBase<TextureImplPs3> Texture;
```

Listing 3.5: Template typedefs

Using this pattern and a corresponding header file to be included by the client as suggested in Section 3.6.2, this approach solves all the original problems and provides the following characteristics:

- Best performance, no virtual function calls.
- Clear separation of platforms.
- Clear separation and abstraction of platform-independent and platform-dependent code.
- Absolutely no redundant code.
- Platform-dependent code can be changed to platform-independent code without hassles (and vice versa).

If desired, forward declarations of the template classes can be added to a separate header file, similiar to the **iosfwd.h** [36] of the standard template library. This ensures that the client code does not need to include both the platform-independent and platform-dependent parts in every header, but rather uses the correct forward declarations instead.

Additionally, it should be noted that the implementation class does not need a pointer to the platform-independent base class for accessing common functionality. If designed accordingly, all common functionality can be placed in utility classes or free functions, which can be used by both base and implementation classes.

3.7 Multi-platform rendering

Although the platform's graphics hardware is somewhat similar to a certain degree, the underlying APIs provide different levels of hardware access, which lead to several possible strategies for writing rendering code on multiple platforms. These strategies will be discussed in the following sections.

3.7.1 Submission-based vs. buffer-based rendering

These two rendering schemes describe **how** rendering commands are issued to the graphics hardware. They do not deal with the fact **when** these commands are issued.

Submission-based rendering

In principle, rendering nowadays can be done in two different ways, with submission-based rendering being the more common one. Submission-based rendering describes the way most rendering APIs work, e.g. Direct3D 9 and OpenGL. Rendering is done by submitting lists, strips or fans of primitives (triangles, quads, or lines) to the graphics hardware using the respective API calls, which are known as drawcalls.

Because each such drawcall has to go through the DirectX DLL and the graphics driver before finally arriving at the hardware, drawcalls still take up a significant amount of CPU time in rendering code on the PC platform. Newer versions of DirectX (DirectX 10 and 11) somewhat alleviate this problem, but it still is a concern to game developers.

The advantage of submission-based rendering is that dealing with primitives is still done in a fairly high-level way, and both the graphics API and the graphics driver are responsible for building commands for the GPU to execute, which essentially puts less burden on the developer and makes the rendering process less error-prone.

The disadvantage of this kind of rendering scheme is that every drawcall has to be re-submitted every frame, hence rendering instructions can't be cached between frames and can take up a considerable amount of CPU time.

APIs mainly provide submission-based rendering when they have to work on a multitude of different hardware, keeping intricate hardware details away from the developer.

Buffer-based rendering

Buffer-based rendering is mostly present in rendering APIs for consoles, and allows developers to take full control over the hardware and leverage the GPUs capabilities to its fullest extent possible. In buffer-based rendering, so-called command buffers consisting of plain GPU commands are submitted directly to the graphics hardware, which of course offers a great deal of performance optimizations like pre-compiled command buffers, exploiting temporal coherency, and so forth.

In exchange, buffer-based rendering exhibits much bigger sources of error than submission-based rendering, mainly because now the developer has to deal with hardware bugs, elaborate hardware details and CPU-GPU synchronization himself.

Despite being primarily submission-based rendering APIs, Direct3D 10/11 and OpenGL offer a very limited form of buffer-based rendering via contexts and display lists, respectively. It should be noted that rendering in Direct3D

3 was based on execute-buffers, which were later abandoned in favor of the DrawPrimitive API.

3.7.2 Immediate vs. delayed rendering

Contrary to the aforementioned schemes, these two rendering schemes describe **when** rendering commands are issued to the graphics hardware.

Immediate rendering

In immediate rendering, every rendering command is immediately submitted to the API during a frame's rendering. These commands can be cached and buffered transparently, and the API or driver will decide when these commands are to be processed by the GPU. Both submission-based and buffer-based rendering schemes can easily be used to implement immediate rendering.

A perfect example of immediate rendering would be writing commands directly into the GPU's primary ring-buffer, with these commands being fetched by the GPU, immediately performing the respective tasks.

Delayed rendering

In delayed rendering, a whole frame's worth of commands is buffered in memory before being submitted to the GPU at once. The well-established standard is to submit commands for frame n to the GPU, while building commands for frame $n+1$ on the CPU. In essence, this enables the CPU and GPU to run almost completely in parallel. Naturally, a buffer-based rendering scheme is more suitable for delayed rendering than submission-based rendering, but makes CPU-GPU synchronization harder.

3.8 Multi-threaded rendering

With multi-core CPUs becoming more and more mainstream, it is essential that a rendering engine supports efficient multi-threaded rendering. Efficient in this context means that it's not an option to wrap low-level engine calls with mutexes or critical sections, but rather use lock-free algorithms and more elegant solutions.

With the Nintendo Wii being a single-core platform, it is furthermore essential that the engine's design does not hurt the rendering performance on single-core platforms, but scales well on multi-core platforms. This is further complicated by the fact that Direct3D 9 only allows drawcalls to

be submitted from one thread, while newer APIs support multi-threaded rendering.

By the nature of its design, submission-based rendering lends itself harder to multi-threaded rendering, because CPU-GPU synchronization is implicitly done upon submitting primitives to the API. This makes it hard to use hardware resources with the maximum amount of parallelization across different threads, because each thread can possibly stall upon submission to the API.

In contrast, buffer-based rendering is harder to get bug-free when dealing with multiple threads and delayed rendering, because the additional one frame of lag makes it hard to track which resources are still in use and which can be accessed safely across different threads.

3.9 Comparison between rendering schemes

	Submission-based	**Buffer-based**
Implementation	Easy	Hard
Frame coherency	No	Yes
Multi-threaded rendering	Hard	Easier

	Immediate	**Delayed**
CPU-GPU synchronization	Easier	Harder
Performance	Average	Best
CPU-GPU parallelism	Average	Best

Tab. 3.1: Comparison of different rendering schemes.

3.10 CPU-GPU synchronization

Synchronization between the CPU and the GPU needs to be done to prevent the GPU from using resources while they are being altered on the CPU at the same time. When working with APIs such as Direct3D or OpenGL, this is automatically done by the API. On consoles, this has to be taken care of by the developer because the used APIs are much more low-level.

Basically, synchronization has to be done when dynamically updating resources like vertex buffers, index buffers or textures, but also when dealing

with render targets and occlusion queries, where subsequent rendering operations rely on previous ones. Implementation details for each platform are discussed in Section 4.1.

3.11 Cache coherency

Another issue which can be completely ignored in PC development is keeping memory and cache in a coherent state, which is usually handled by the operating system. Some APIs provide fine-grained control over zeroing, prefetching and flushing ranges of memory, and have to be used accordingly by the developer. Manually zeroing or prefetching cache lines can greatly improve performance in critical code, and flushing the data cache has to be done whenever the CPU alters GPU resources because these are directly fetched from main memory by the GPU.

Additionally, invalidating the GPU's vertex and textures caches has to be done by the developer. Cache coherency can be tricky to get right because invalidating caches too often can lead to performance losses, whereas doing it only rarely leads to rendering artifacts at best – most of the time, it will cause subtle and hard-to-find bugs.

3.12 Platform APIs

3.12.1 Microsoft Windows Direct3D 9

Direct3D is the de facto standard in rendering APIs used by games today. With Windows Vista and Windows 7 not being mainstream at the time of writing, using Direct3D 9 instead of version 10 or 11 for the PC version of the proposed engine was a natural choice. This also offers a greater amount of backwards compatibility with older hardware. Section 2.3.1 deals with Direct3D 9 in more detail.

3.12.2 Microsoft XBox360 Direct3D

The XBox360 API basically is a mix between the Direct3D 9 and Direct3D 10 API, and offers some Xbox360-only features such as memory export, which can enable DX10-class functionality such as stream-out. However, the Xbox360 does not offer a full DirectX 10 feature set.

3.12.3 Sony PlayStation 3 LibGCM

The PlayStation 3 offers two different graphics APIs: *PSGL* and *LibGCM*.

PSGL is a graphics API based on OpenGL ES 1.0 with extensions for modern GPUs, and is primarily a high-level API. As such, it can be used for prototyping or porting applications from OpenGL to the PlayStation 3, but almost all games use LibGCM because it allows the developer to exploit the hardware's full capabilities. On the other hand, PSGL is available even to hobby developers working with Linux on the PlayStation 3.

LibGCM provides low-level access to the graphics hardware, and is used by professional game developers.

3.12.4 Nintendo Wii GX

The GX family of functions builds the graphics API for the Nintendo Wii, which bears resemblance to the OpenGL API.

3.12.5 Comparison

Of all platforms, the PC and Xbox360 are the best choice for rapid development and getting started with 3d rendering, mainly because the Direct3D API is relatively easy to understand, both platforms integrate well into the Visual Studio IDE, and there is a vast amount of documentation available. However, mastering the Xbox360 is significantly more complex than mastering the PC.

In contrast, developing for the PlayStation 3 needs thorough understanding of the hardware and API, even when just getting started. Mastering the platform is notoriously hard, and requires elaborate software engineering skills. However, the PlayStation 3 provides the most horsepower of all consoles.

Chapter 4

Implementing the occlusion query functionality on multiple platforms

This section discusses parts of the Daedalus engine which are crucial for enabling the occlusion query functionality on all platforms. Almost all visibility algorithms described in the later parts of this thesis make extensive use of this functionality. All provided source code listings are taken from the Daedalus engine, which has been developed for this thesis.

Main contributions in this chapter are the following:

- A new implementation of render states using advanced C++ features, which is very fast, robust and has an extremely small memory footprint.
- A multi-platform interface for immediate mode rendering, providing a device for fast rendering of ad-hoc primitives, optimized using special capabilities of each platform.
- A novel low-level mechanism for enabling occlusion query functionality on platforms without hardware-support was invented.

4.1 CPU-GPU synchronization

As discussed in Section 3.10, CPU-GPU synchronization is crucial for many low-level parts of a graphics engine. However, the user should not have to deal with synchronization himself, but rather use different means for accessing shared data between the CPU and GPU.

Low-level resources like vertex buffers, index buffers, and textures all offer two methods for granting access to their data in Daedalus: **Lock** and **Unlock**, as shown in Listing 4.1. As soon as a resource is locked, it can be accessed by the engine or user, and must be freed again by a call to Unlock, otherwise the GPU might stall forever.

Different kinds of locks allow the user to specify the desired level of access - either read-only, write-only, or read-and-write. This allows for certain

```
class VertexBuffer
{
  ...

  /// Locks the vertex buffer
  void Lock(unsigned int vertexOffset, unsigned int
     vertexCount, VertexBufferLock lock);
  void Lock(unsigned int vertexOffset, unsigned int
     vertexCount, VertexBufferLock lock) const;

  /// Unlocks the vertex buffer
  void Unlock(void);
  void Unlock(void) const;

  ...
};

class IndexBuffer
{
  ...

  /// Locks the index buffer
  void Lock(unsigned int indexOffset, unsigned int indexCount,
       IndexBufferLock lock);
  void Lock(unsigned int indexOffset, unsigned int indexCount,
       IndexBufferLock lock) const;

  /// Unlocks the index buffer
  void Unlock(void);
  void Unlock(void) const;

  ...
};

class Texture
{
  ...

  /// Locks the texture object
  void Lock(const IRect& region, TextureLock lock, unsigned
     int mipLevel);
  void Lock(const IRect& region, TextureLock lock, unsigned
     int mipLevel) const;

  /// Unlocks the texture object
  void Unlock(unsigned int mipLevel);
  void Unlock(unsigned int mipLevel) const;

  ...
};
```

Listing 4.1: Accessing resources on the CPU

optimizations on some platforms, e.g. as long as the user only reads from a resource the GPU is allowed to access its contents as long as they are not altered by the GPU itself.

The following subsections describe the implementations for all platforms.

4.1.1 PC platform

Fortunately, Direct3D 9 provides easy access to resources via Lock and Unlock methods. CPU-GPU synchronization is handled inside the API and driver, hence there's no need to implement any additional functionality.

4.1.2 Other platforms

On the other platforms, CPU-GPU synchronization must be implemented manually. The process is somewhat similar to issuing **events** in Direct3D 9. Basically, specific GPU commands are written to the command-buffer, which can be used to let the GPU tell the CPU about certain graphical states or events. As an example, events such as occlusion queries report the number of rendered pixels, which can be read on the CPU.

Similarly, by issuing events using a monotonically increasing value, the CPU can find out which values have already been encountered by the GPU, and this value in turn can be used for synchronizing the CPU and GPU. Implementations on the driver-level of such functionality (such as in Direct3D on the PC platform) are probably similar to a custom low-level implementation, but details about the underlying strategy are not available for the Direct3D API.

Details and source code for our implementation are available on request.

4.2 Render states

Changing render states like alpha-blending, z-testing, backface-culling and others is one of the most common operations when rendering a frame. Furthermore, render states are also among the number one source of errors when using graphics engines, mainly because they can be in a wrong state for certain operations like drawing transparent geometry if not properly reset by the developers. Experience shows that this can lead to obscure bugs, and even more obscure bug-fixes. Additionally, most visibility algorithms using occlusion queries rely on render states being implemented in a fast way.

Hence, Daedalus makes the following demands on render states:

- They must be cached internally, so only changed states get propagated to the hardware before rendering.
- They must be impossible to break, e.g. rendering should not be faulty because a certain state was not properly reset by the developer.
- They must be extremely fast because they are used so often, especially in visibility algorithms using hardware occlusion queries.
- They must have an extremely small memory footprint.

The following subsections explain how these demands are satisfied.

4.2.1 Caching

Internal caching of render states is easy - rather than setting a render state directly on the hardware, we store its current value (the one set by the user) and the current value set on the hardware. Upon submitting the next drawcall, we flush the render states cache and set only those states whose value differs from the value set on the hardware.

Internally, a render state is defined as in Listing 4.2.

```
struct InternalRenderState
{
  RenderStates::Type type;
  RenderStates::Value hardwareValue;
  RenderStates::Value value;
  u8 padding[4];
};
```

Listing 4.2: Internal render state

In a global cache, we store one *InternalRenderState* instance for each render state exposed by the hardware. Whenever the engine submits a drawcall, the cache gets flushed, and changed render states are set on the hardware, as seen in Listing 4.3. Note that the listing leaves out some important optimizations that will be explained in one of the following sections.

4.2.2 Impossible to break

Caching makes setting redundant render states faster, but does not solve the problem that the user might change render states when using a certain

```
void RenderStatesCache::Flush(void)
{
  ...
  unsigned int i;
  for (i=0; i<RenderStates::COUNT; ++i)
  {
    // check if the hardware value actually differs
    if (states[i].hardwareValue != states[i].value)
    {
      // yes, so set the new value
      device->SetRenderState(states[i].type, states[i].value);
      states[i].hardwareValue = states[i].value;
    }
  }
  ...
}
```

Listing 4.3: Flushing the render states cache

technique or shader, and not restore them to their original state, potentially breaking some other rendering code. The solution to this problem is to make all render state operations completely stack-based.

Basically, we use one global stack for pushing and popping render states. Each stack entry stores the type and value of the render state, and the index into our cache, as seen in Listing 4.4.

```
struct StackState
{
  int index;
  RenderStates::Type type;
  RenderStates::Value value;
  u8 padding[4];
};
```

Listing 4.4: Render states stack entry

Using stack-based render states makes the user's life easier, but he still might forget to pop certain states from the stack. Therefore, neither the *RenderStatesCache* nor the *RenderStatesStack* can be accessed directly by the user. Instead, a single *RenderState* must be used, which automatically pushes the new state onto the stack in its constructor, and pops the value from the stack in the destructor.

Although stack-based render states help a lot, there is still room for breaking them. A naive implementation would let the user specify both type and value of a render state, opening the door for mismatch errors between types and values which can only be flagged at run-time. However, we can do much better than this and let the compiler work for our benefit.

In C++, each **enum** acts as a separate type, hence we can define an enum for each render state type, and add all valid values for this state to the respective enum, as shown in Listing 4.5.

By making the aforementioned *RenderState* class take a template parameter, the compiler will automatically deduce the enum type for us, so render state type and value mismatches can no longer occur. The *RenderState* class is shown in Listing 4.6.

There are still two issues left to solve:

- So far, the compiler knows the type of the enum. But how do we get the respective type needed for the low-level API call?
- Which index of the render states cache does each type occupy?

Both of these issues can be solved by using another advanced C++ feature, namely template specialization.

4.2.3 Speed and memory footprint

By using template specialization on each enum type, we can basically assign an index and type to each render state at compile-time. Listing 4.7 shows how this is done on the PC platform. Furthermore, this gives room for another kind of optimization.

Each time a render state is set in our cache, we can keep track of the minimum and maximum index of all render states touched since the last flush. This way, we need to only check a small range of render states upon each cache flush, instead of all states supported by the engine. This drastically reduces the cost of each render states cache flush.

Furthermore, by assigning proper indices in the template specializations, we can group more commonly used render states together, further reducing the cost of each cache flush. This could even be implemented by the user to fine-tune the engine to his needs, if desired.

Memory-wise, each *RenderState* takes up one stack variable, which amounts to 32 bytes on each platform. The stack and cache are global for the whole program and take up about 4 kB as a whole by default, which allows for more than 200 render states to be pushed onto the stack. The stack size

```
enum CullMode
{
  CULLMODE_NONE = D3DCULL_NONE,
  CULLMODE_CW = D3DCULL_CW,
  CULLMODE_CCW = D3DCULL_CCW
};

enum ZTest
{
  ZTEST_FALSE = D3DZB_FALSE,
  ZTEST_TRUE = D3DZB_TRUE,
  ZTEST_USE_W = D3DZB_USEW
};

enum ZWrite
{
  ZWRITE_FALSE = FALSE,
  ZWRITE_TRUE = TRUE
};

enum ZFunction
{
  ZFUNCTION_NEVER = D3DCMP_NEVER,
  ZFUNCTION_LESS = D3DCMP_LESS,
  ZFUNCTION_EQUAL = D3DCMP_EQUAL,
  ZFUNCTION_LESSEQUAL = D3DCMP_LESSEQUAL,
  ZFUNCTION_GREATER = D3DCMP_GREATER,
  ZFUNCTION_NOTEQUAL = D3DCMP_NOTEQUAL,
  ZFUNCTION_GREATEREQUAL = D3DCMP_GREATEREQUAL,
  ZFUNCTION_ALWAYS = D3DCMP_ALWAYS
};

enum AlphaTest
{
  ALPHATEST_FALSE = FALSE,
  ALPHATEST_TRUE = TRUE
};

enum AlphaFunction
{
  ALPHAFUNCTION_NEVER = D3DCMP_NEVER,
  ALPHAFUNCTION_LESS = D3DCMP_LESS,
  ALPHAFUNCTION_EQUAL = D3DCMP_EQUAL,
  ALPHAFUNCTION_LESSEQUAL = D3DCMP_LESSEQUAL,
  ALPHAFUNCTION_GREATER = D3DCMP_GREATER,
  ALPHAFUNCTION_NOTEQUAL = D3DCMP_NOTEQUAL,
  ALPHAFUNCTION_GREATEREQUAL = D3DCMP_GREATEREQUAL,
  ALPHAFUNCTION_ALWAYS = D3DCMP_ALWAYS
};
```

Listing 4.5: An example of render state enums

```
class RenderState : private NonCopyable
{
public:
  template <typename T>
  RenderState(T value)
  {
    RenderStatesStack::Push(RenderStatesCache::Get<T>());
    RenderStatesCache::Set(value);
  }

  ~RenderState(void)
  {
    RenderStatesCache::Set(RenderStatesStack::GetTopIndex(),
       RenderStatesStack::GetTopState(), RenderStatesStack::
       GetTopValue());
    RenderStatesStack::Pop();
  }
};
```

Listing 4.6: *RenderState* class

```
namespace RenderStateTypes
{
  template <typename T> struct Extract {};
  ...
  template <> struct Extract<RenderStates::CullMode> { enum {
      TYPE = D3DRS_CULLMODE, INDEX = 0 }; };
  template <> struct Extract<RenderStates::ZTest> { enum {
      TYPE = D3DRS_ZENABLE, INDEX = 2 }; };
  template <> struct Extract<RenderStates::ZWrite> { enum {
      TYPE = D3DRS_ZWRITEENABLE, INDEX = 3 }; };
  template <> struct Extract<RenderStates::AlphaBlending> {
      enum { TYPE = D3DRS_ALPHABLENDENABLE, INDEX = 8 }; };
  ...
}
```

Listing 4.7: Render state template specialization for Direct3D 9

can furthermore be customized by the user, leaving a very small memory footprint.

Listing 4.8 shows how the final render states are used in client code.

```
{
  RenderStates noCull(RenderStates::CULLMODE_NONE);
  RenderStates noZTest(RenderStates::ZTEST_FALSE);
  RenderStates zWrite(RenderStates::ZWRITE_TRUE);

  // rendering code
  ...
}
```

Listing 4.8: Client code using the RenderState class

4.3 Immediate mode rendering

Most online visibility culling algorithms require the engine to render bounding boxes of certain objects or nodes of a hierarchy in order to determine their visibility. Using unique vertex- and index-buffers for each bounding box is certainly too slow on most platforms, and additionally each submitted draw call incurs a slight performance overhead. Therefore, it is crucial that a small amount of simple ad-hoc primitives can be rendered as fast as possible, using the *RenderImmediate* interface.

In games, such immediate mode primitives must be rendered e.g. when rendering completely dynamic geometry such as in-game texts, some HUD elements, or particles. As described in Section 3.7.2, immediate mode means that every rendered primitive is sent to the hardware as soon as possible.

In principle, the *RenderImmediate* class wraps the low-level access to the GPU's command buffer on each platform. The *RenderImmediate* interface offers methods for putting vertices, texture coordinates, normals, colors, and other data into the underlying buffer. A typical usage for immediate mode rendering is shown in Listing 4.9, which shows how debug text is rendered using the interface.

The following subsections describe how immediate mode rendering is implemented on different platforms.

```
RenderImmediate::Begin(shader, material, vertexDeclaration,
   PRIMITIVE_TYPE_QUAD_LIST, numCharacters*4, numCharacters);
{
  ...
  for (i=0; i<numCharacters; ++i)
  {
    ...
    RenderImmediate::Vertex(x, y, 0);
    RenderImmediate::Color(nativeColor);
    RenderImmediate::TexCoord(u, v);
    ...
  }
  ...
}
RenderImmediate::End();
```

Listing 4.9: Immediate mode rendering

4.3.1 PC platform

Unfortunately, Direct3D 9 on the PC platform doesn't grant access to such low-level details as command buffers, therefore an abstraction layer must be built. In this case, *RenderImmediate* manages its own dynamic vertex buffer which is used as a ring-buffer so as to not overwrite previously rendered data, in order to avoid possible GPU stalls. However, as soon as a wrap-around in the ring-buffer occurs, the old data is discarded and the buffer is filled from the beginning again. This all happens completely transparent to the user. Internally, the different *RenderImmediate* methods fill this ring-buffer, as seen in Listing 4.10.

4.3.2 Other platforms

RenderImmediate on the console platforms makes use of special PowerPC instructions in order to improve performance. Additionally, console-specific features are used, resulting in code that is more than 10 times faster than simply writing the values into memory via straightforward C++ code. Details and source code are available on request.

4.4 Occlusion queries

Typically, an API offers facilities to query the GPU about different kinds of data, such as cache utilization, bandwidth timings, pipeline timings, and the

```
class RenderImmediate : private MonoState
{
public:
  ...
  static inline void Vertex(float x, float y, float z)
  {
   register u8* __restrict ptr = (u8* __restrict)iterationPtr;
   *((f32*)(ptr)) = x;
   *((f32*)(ptr + 4)) = y;
   *((f32*)(ptr + 8)) = z;
   iterationPtr = ptr+12;
  }

  static inline void TexCoord(float x, float y)
  {
   register u8* __restrict ptr = (u8* __restrict)iterationPtr;
   *((f32*)(ptr)) = x;
   *((f32*)(ptr + 4)) = y;
   iterationPtr = ptr + 8;
  }
  ...
};
```

Listing 4.10: Parts of the *RenderImmediate* interface on the PC

number of rendered pixels. The latter is usually called an **occlusion query**, and is used to determine how many pixels survived the z-test when issuing drawcalls inside a query.

Occlusion queries in Daedalus provide the interface shown in Listing 4.11. Typically, an *OcclusionQuery* object is used the following way:

- Begin the query by calling *Begin()*.
- Issue drawcalls to the hardware.
- End the query and flush the command buffer by calling *End()*.
- Either poll the query status by calling *IsFinished()* while working on other data, or stall the CPU and wait until data is available by continuously calling *IsFinished()* in a loop.
- Retrieve the data by calling *GetNumVisiblePixels()*.

The following subsections explain how occlusion queries are implemented on different platforms.

```
class OcclusionQuery
{
public:
  ...
  /// Begin the query
  void Begin(void);

  /// End the query
  void End(void);

  /// Returns whether the query has finished and data from the
      GPU is available
  bool IsFinished(void) const;

  /// Returns the number of visible pixels inside the query
  unsigned int GetNumVisiblePixels(void) const;
  ...
};
```

Listing 4.11: Occlusion queries in Daedalus

4.4.1 PC platform

The Direct3D API provides several kinds of queries which can be used to retrieve different performance metrics. Conveniently, one such query type is an occlusion query, and commands issuing these queries can be put into the rendering pipeline with ease, as shown in Listing 4.12.

On the PC, issuing queries can take up a huge amount of clock cycles whenever the command buffer is to be flushed, because a mode transition from user mode to kernel mode takes place, as described in [58].

4.4.2 Other platforms

For platforms on which occlusion queries are not offered by the API, we devised a novel mechanism for enabling this functionality using GPU metrics and hardware interrupts in conjunction with a special kind of CPU-GPU synchronization.

Normally, each GPU (and therefore probably each API) exposes functions for retrieving metrics such as the number of rendered pixels. Using such an API, the number of pixels rendered by the GPU up to a certain point in time can be gathered by simply subtracting such values, which are retrieved in *OcclusionQuery::Begin()* and *OcclusionQuery::End()*.

However, that solves only part of the problem, because reading metrics

```
OcclusionQuery::OcclusionQuery(void) :
  query(NULL),
  numPixels(0)
{
  RenderDevice::Instance()->GetDevice()->CreateQuery(
      D3DQUERYTYPE_OCCLUSION, &query);
}


void OcclusionQuery::Begin(void)
{
  query->Issue(D3DISSUE_BEGIN);
}


void OcclusionQuery::End(void)
{
  query->Issue(D3DISSUE_END);

  // flush the command buffer to give the GPU a chance to
      actually finish
  query->GetData(NULL, 0, D3DGETDATA_FLUSH);
}


bool OcclusionQuery::IsFinished(void) const
{
  return (query->GetData(&numPixels, sizeof(DWORD),
      D3DGETDATA_FLUSH) == S_OK);
}
```

Listing 4.12: Implementation of *OcclusionQuery* on the PC platform

usually does not insert a command into the command buffer, but rather extracts the GPU's metrics at the time of calling. Depending on the GPU's workload and the filling level of the command buffer, the GPU might be rendering primitives completely unrelated to the occlusion query, which would ultimately lead to wrong and fluctuating results.

This can be worked around by using events as described in Section 4.1.2. If available, an event triggers an interrupt during which the GPU metrics can be read. This makes sure that even though rendering is done in parallel, the metrics are still read at the correct time. Unfortunately, while the CPU is handling those interrupts, the GPU might already be consuming commands from the buffer again, resulting in slightly wrong occlusion query results.

This race condition can be avoided by making the GPU stop consuming further commands whenever an event used for occlusion queries is encountered. As soon as the corresponding interrupt for an event is handled, the GPU can start fetching commands again. Even though this stalls the GPU for a very short amount of time, the achieved occlusion query functionality makes up for this shortcoming.

Figure 4.1 shows a complete timeline for a successful occlusion query, using the aforementioned algorithm. Details and source code of our implementation are available on request.

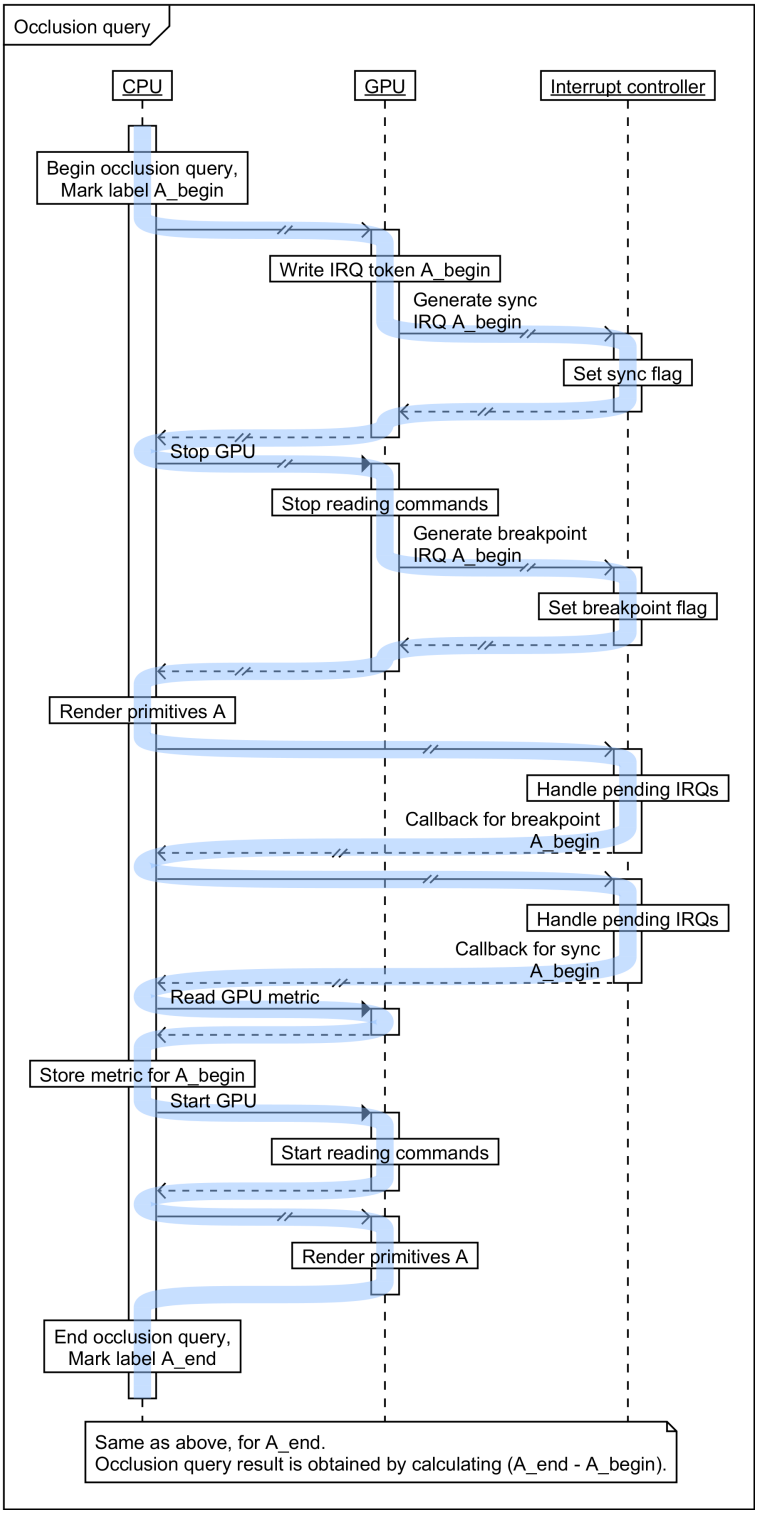


Fig. 4.1: Timeline for a successful occlusion query

Chapter 5

Related work on visibility algorithms

This chapter discusses visibility culling in general as well as current state-of-the-art visibility algorithms, and describes the theory behind the algorithms implemented in the Daedalus engine.

5.1 Visibility Culling

Visibility culling describes the process of removing parts of the scene which upon rendering will not contribute to the rendered image. This means that e.g. objects which cannot be seen from the current camera's point-of-view are not sent down the rendering pipeline. Visibility culling can improve an application's performance greatly because of the following reasons:

- If culled, some objects like particle systems and skinned meshes can abandon their simulation part. This can save great amounts of CPU time.
- Culled objects are not sent to the graphics driver, saving CPU cycles.
- Culled objects are not transformed, saving vertex shader cycles.
- Culled objects are not shaded, saving pixel shader cycles.
- Culled objects are not rasterized, saving raster operations.

With multi-core CPUs more and more becoming the trend nowadays, visibility culling is crucial for every real-time rendering application because the GPU will almost always be the bottleneck, especially on console hardware. Thus, improving the GPU's performance by limiting the number of rendered primitives is an essential prerequisite for real-time rendering.

The next sections will explain some of the employed algorithms.

5.1.1 Backface Culling

Backface culling discards all primitives of closed, opaque objects which do not face the camera, and hence do not contribute to the final image. Determining whether a triangle faces the camera can be done by computing the normal of the triangle in screen-space, and discarding it if the normal turns out to be pointing away from the camera.

Backface culling can be configured easily by setting appropriate render states in the graphics API, enabling either backface or frontface culling, or disabling culling completely. The latter is useful when rendering transparent primitives or double-sided materials, for example.

Although backface culling reduces the number of rendered triangles and therefore increases rasterization performance, it puts more load on the geometry stage, where the computations are usually done. Because of that, it proves to be beneficial to carry out backface culling on the CPU/SPU if there is enough computational power left to spare, and the GPU has slow triangle and vertex setup, as it is the case on the PlayStation 3. Typically, one of the SPUs performs culling of back faces and micro-triangles on an input stream of primitives, and generates a new index buffer for each draw call [4].

5.1.2 View-Frustum Culling

As can be seen in Figure 5.1, every perspective camera describes a frustum, capped at the camera's near and far plane. Because every primitive outside the camera's frustum ends up being outside the unit cube in clipping space, it need not be sent down the rendering pipeline. The process of determining and discarding such primitives which lie outside the view frustum is called view-frustum culling.

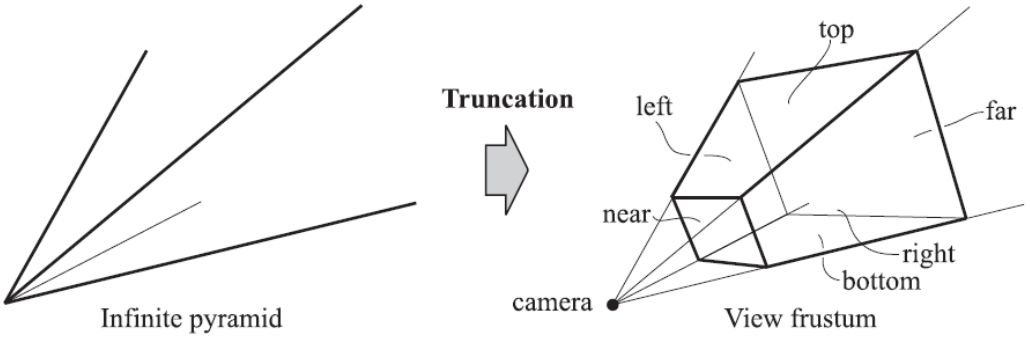


Fig. 5.1: View frustum described by a perspective camera. (Image courtesy of A.K. Peters Ltd., taken from "Real-Time Rendering, Third Edition" [2])

Typically, bounding primitives such as *axis-aligned bounding boxes* (AABBs) or *oriented bounding boxes* (OBBs) (see Section 5.2.1) instead of single triangles are tested against the view frustum. Therefore, even for a large amount of rendered triangles, the number of performed frustum tests is quite small, normally less than a thousand tests per frame. As described in Section 6.1, these tests can be further optimized, and can benefit from hierarchical data structures as well as temporal coherence. In typical game scenarios, view-frustum culling can already tremendously increase rendering performance.

Because of its simplicity, generality and good performance characteristics, view frustum culling is still one of the most common culling algorithms implemented in games nowadays, even though the idea is more than 30 years old [12]. Section 6.1 deals with this algorithm in more detail.

5.1.3 Occlusion Culling

Occlusion culling makes use of the fact that objects which are completely occluded by other objects in the scene do not need to be rendered. Determining the set of occluded primitives is easily understood in theory, but can be hard to implement in a real-time rendering context. As such, occlusion culling algorithms are among the most complex culling algorithms to implement.

Naturally, the performance improvement obtained by employing occlusion culling depends very much on the scene itself. If the rendered scene exhibits almost no occlusion to take benefit of, the algorithm itself will only add overhead, but no performance gain. Nevertheless, a major advantage of online occlusion culling is that it can be used for both static and dynamic geometry, and there is no need to differentiate between these two kinds of geometry.

Many different occlusion culling algorithms have been developed over the last years, some of them using available hardware functionality [35], others being based on software solutions [13] [31] [74].

Sections 6.2, 6.3, 6.4 and 6.5 of this thesis describe the theory and implementation behind a naive implementation as well as two of the most common occlusion culling algorithms used today, which can be completely implemented using consumer graphics hardware. Game development studios such as *Insomniac Games* successfully use occlusion culling algorithms in their games [33].

5.1.4 Online vs. offline culling

Online visibility culling algorithms determine the set of visible objects every frame, without any precomputation. This does not necessarily mean that

these algorithms have to do redundant work each frame, quite the contrary – a good online culling algorithm makes use of temporal coherence. Furthermore, a good culling algorithm makes use of spatial data structures in order to work in a hierarchical fashion.

Offline visibility culling algorithms accomplish most of their work using a precomputed set of data, which is calculated in an offline process. At run-time, usually not much is left to be done, hence offline culling algorithms tend to not have such a performance impact as other online algorithms do. Usually, using offline culling does not add significant overhead to the rendering phase, even when almost no objects can be culled. However, depending on the algorithm and the data set, the amount of precalculated data can be quite substantial, therefore increasing an application's memory footprint. Additionally, offline culling algorithms can only be used for static geometry.

These kinds of algorithm determine a PVS (*Potentially Visible Set*) [40] of objects for predefined viewpoints in an offline process. These viewpoints do not need to follow any spatial alignment, and can be grid-based, sector-based or completely arbitrary. Each such region of viewpoints is typically called a view cell.

The PVS of each view cell describes a set of other objects which can possibly be seen from this cell in any direction. Depending on the type of application and the number of view cells, this amount of data can be quite large, requiring efficient packing to be employed. Most of the time, however, the number of view cells is in the range of a few hundred cells, and only one bit needs to be stored to signal whether another view cell is visible or not. Even for 1000 view cells, the visibility data needs approximately 120KB storage.

Because all the visibility information is precalculated in an offline process, the geometry of those view cells needs to stay static, which is one of the major drawbacks of PVS culling algorithms. Still, PVS culling is an extremely popular culling mechanism because it can be used in conjunction with other culling algorithms for dynamic geometry, and its only run-time cost boils down to a few simple comparisons of single bits.

5.2 Spatial Data Structures

In culling algorithms, spatial data structures serve as a means to drastically lessen the number of performed queries such as intersection or visibility tests. Such data structures are very useful for other types of queries such as collision detection as well, and can generally be applied to n-dimensional space, not only two-dimensional and three-dimensional problems.

A spatial data structure is usually organized in a hierarchical fashion, such that an object in the hierarchy fully encloses all its children. This way, the complexity of queries can typically be reduced from *O(n)* to *O(log n)*, introducing a significant speed up when n is large. There are many different types of spatial data structures, such as *bounding volume hierarchies* (BVHs), *binary space partitioning* (BSP) *trees* and *octrees*. Because all of the implemented algorithms use BVHs, we limit the remaining discussion to this kind of spatial data structure.

5.2.1 Bounding Volume Hierarchies

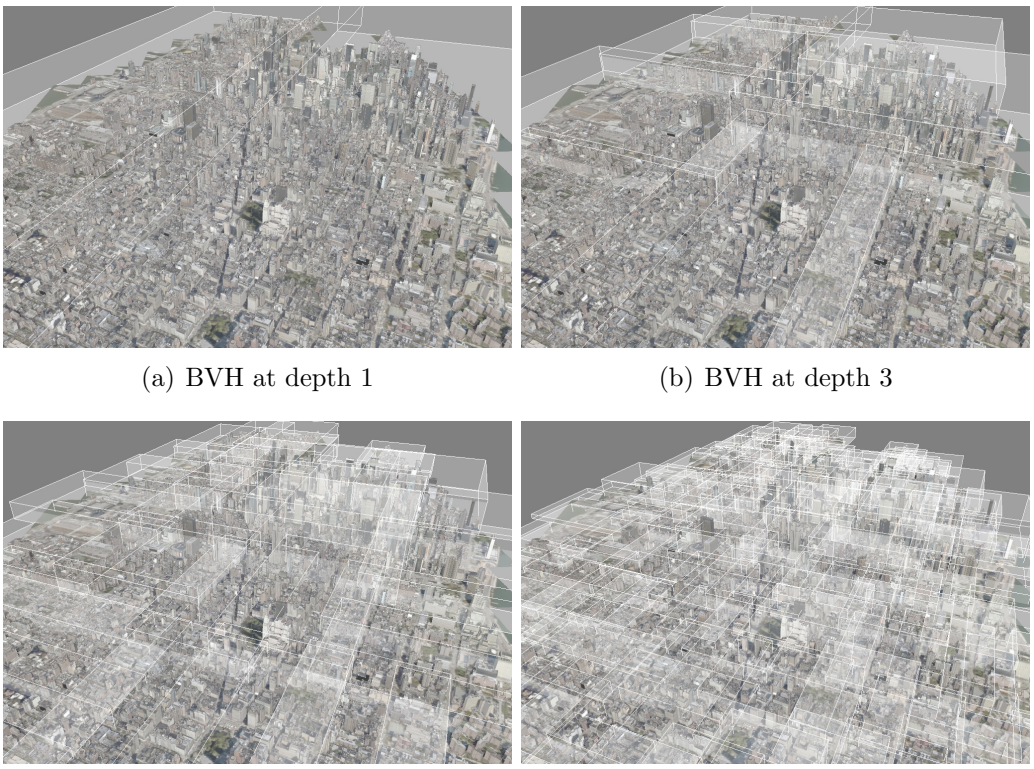
As the name implies, a bounding volume hierarchy is built using a set of bounding volumes, organized in a hierarchical fashion. Such a bounding volume itself encloses a set of objects, and is typically a simple primitive such as a bounding sphere, *axis-aligned bounding box* (AABB) or *oriented bounding box* (OBB). A common trait shared by all bounding volumes is that queries on them are relatively simple compared to the objects they contain. As an example, testing two AABBs for intersection is much easier and faster than to test all individual objects inside each AABB for intersection.

Generally speaking, a bounding sphere is the fastest volume to query but encloses the biggest volume, whereas OBBs provide the tightest fit around objects, but are the computationally most expensive bounding volumes to query. Of course, actual bounding volume metrics vary depending on the nature of the application where they are used. Therefore, all implemented algorithms use hierarchies consisting only of AABBs in order to provide a good compromise for game applications. Figure 5.2 depicts such a BVH made of AABBs.

An AABB hierarchy is organized in a tree structure, and distinguishes between three different kinds of nodes:

- Root node: The root node is the topmost node in the hierarchy, and does not have any parent nodes.
- Leaf nodes: Leaf nodes do not have any child nodes, and contain the actual geometry to be rendered.
- Internal nodes: Internal nodes store pointers to its child nodes.

Each and every node in the tree is assigned a bounding volume which completely encloses the objects in its entire subtree. Thus, depending on the type of query, a decision can be made on a whole subtree at once, without



(a) BVH at depth 1

(b) BVH at depth 3

(c) BVH at depth 5

(d) BVH at depth 7

Fig. 5.2: BVH consisting of AABBs

having to test subtrees further down the hierarchy. This can tremendously reduce the number of needed computations.

Based on the fact that any kind of bounding volume is normally stored at the mesh level in games, a BVH can be built by starting from a root node fully enclosing all bounding volumes, and subdividing nodes using a top-down approach. Many different strategies can be used to build such a hierarchy [57] [67], such as the *surface area heuristic* (SAH) [71]. All AABB hierarchies used in this thesis have been built using this heuristic.

It should be noted that – depending on the data set and heuristic – BVHs can take some time to build, and are therefore generated in an offline process for static geometry. For dynamic geometry, a separate BVH hierarchy can be rebuilt from scratch each frame using the *spatial median split heuristic*. After rebuilding the hierarchy, the visibility classification of each object can be used to reconstruct the visibility classification of the hierarchy nodes. Using this method, some hundred dynamic objects can easily be handled.

Chapter 6

Implementing online occlusion culling on multiple platforms

This chapter describes the implementation of the occlusion culling algorithms implemented in the Daedalus engine. Additionally, optimizations to some algorithms are introduced, resulting in better performance on all platforms.

Main contributions in this chapter are:

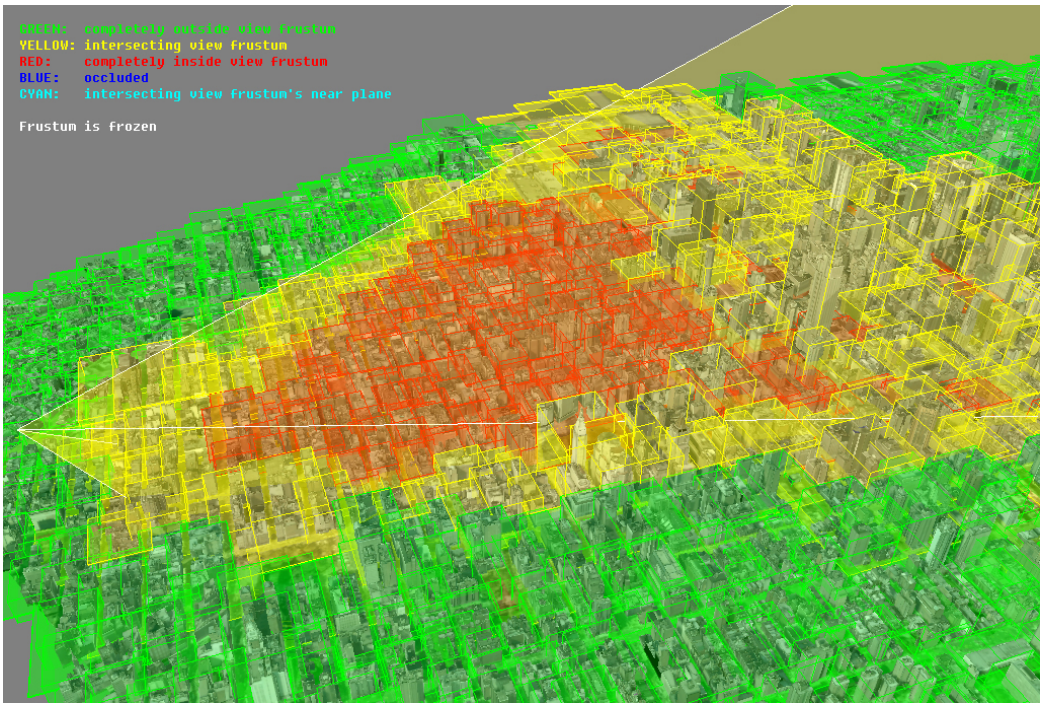
- Optimization of parts of the hierarchical view-frustum culling using branch-free operations on console hardware.
- A custom tool used for evaluating optimized parameters for CHC++ on fixed-hardware platforms.
- An optimized CHC++ algorithm using fine-tuned parameters on console hardware, a branch-free fixed-size stack and a branch-free fixed-size queue which serve as a replacement for the standard STL containers used in CHC++.

6.1 Hierarchical View-Frustum Culling

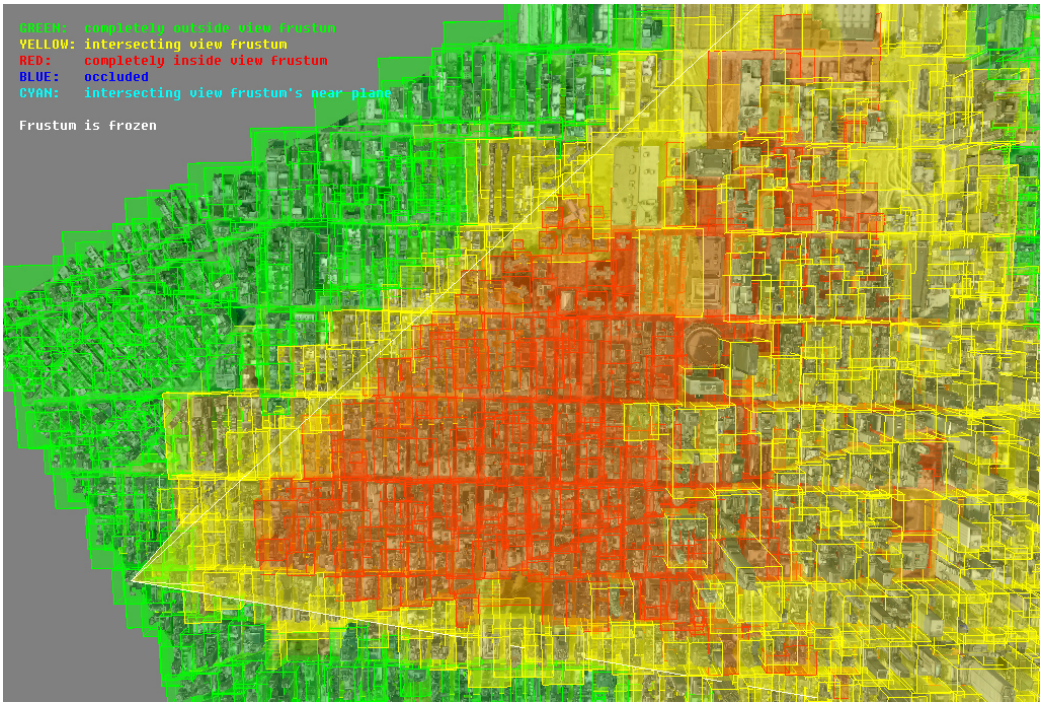
Basic view-frustum culling determines whether a given bounding volume such as an AABB is either completely *outside* the frustum, *intersects* the frustum, or is completely *inside* the frustum. Figure 6.1 shows a scenario displaying each of these three cases.

Each frustum is defined by six planes which are used to perform the inside/outside test. A basic, naive intersection test will perform this test against all frustum planes for each of the eight corners.

Due to the frustum's nature, some AABBs might falsely be classified as being intersecting the frustum while they are outside. Such bounding volumes and their primitives must be treated as being visible, otherwise rendering artifacts will occur.



(a) Frustum culling using AABBs



(b) Frustum culling using AABBs, seen from above

Fig. 6.1: Non-hierarchical view frustum culling. Green AABBs are completely outside the frustum, yellow AABBs are intersecting the frustum, and red AABBs are completely inside the frustum.

By utilizing a BVH, the basic view-frustum culling algorithm can be extended to work in a hierarchical manner, cutting down on the number of performed culling tests. This is based on the two following observations:

- If a node is classified as being *outside* the frustum, all of its children must be outside the frustum as well. Hence, the node and its whole subtree can be discarded from rendering.
- If a node is classified as being *inside* the frustum, all of its children must be inside the frustum as well. Hence, the node and its whole subtree can be rendered immediately. Child nodes then need not be tested further.

Following these two simple rules can already cut the number of performed frustum tests from *O(n)* down to *O(log n)*, where n is the number of single bounding volumes. The next sections introduce further optimizations for the test itself, as well as strategies how to further exploit the BVH and take advantage of temporal coherence.

6.1.1 N- and P-Test

Instead of testing each of the eight corners against all planes, only two points need to be tested [30] in order to classify the AABB, namely the two points called the *n-* and *p-vertices*, which form a diagonal that is aligned with the plane's normal, passing through the box center [5]. The *p-vertex* has a greater signed distance from the plane than the *n-vertex*. Listing 6.1 shows the view-frustum culling implementation taking advantage of this optimization.

The comparisons inside the *IsOutsidePlane()* and *IsIntersectingPlane()* functions are perfect candidates to take advantage of the **fsel** instruction *(Floating-Point Select)* which produces a branchless selection based on floating-point values. By using these instructions, possible floating-point pipeline stalls triggered by branches do not occur. Unfortunately, the **fsel** instruction is only available on PowerPC-based platforms, and not on the x86 architecture – but the latter possesses good branch prediction facilities anyway, hence this is not really a penalty.

6.1.2 Exploiting Temporal Coherence

Assuming that the frustum planes do not change significantly each frame, the plane which culled a bounding volume last frame can be stored for each volume. When performing the test against this plane first, it is likely that the bounding volume can be culled immediately, without performing tests for the other planes afterwards.

```
Frustum::IntersectionResult Frustum::Intersects(const AABB&
    box) const
{
  // test if the box is fully outside
  unsigned int i;
  for (i=0; i<NUM_PLANES; ++i)
  {
    if (IsOutsidePlane(box, planes[i]))
      return IR_OUTSIDE;
  }

  // test if the box intersects at least one plane
  for (i=0; i<NUM_PLANES; ++i)
  {
    if (IsIntersectingPlane(box, planes[i]))
      return IR_INTERSECTING;
  }

  return IR_INSIDE;
}

bool IsOutsidePlane(const AABB& box, const Vec4& plane)
{
  const Vec3& boxMin = box.GetMin();
  const Vec3& boxMax = box.GetMax();

  // p-vertex
  const float x = Math::FSel(plane.x, boxMax.x, boxMin.x);
  const float y = Math::FSel(plane.y, boxMax.y, boxMin.y);
  const float z = Math::FSel(plane.z, boxMax.z, boxMin.z);

  return (x*plane.x + y*plane.y + z*plane.z + plane.w) < 0;
}

bool IsIntersectingPlane(const AABB& box, const Vec4& plane)
{
  const Vec3& boxMin = box.GetMin();
  const Vec3& boxMax = box.GetMax();

  // n-vertex
  const float x = Math::FSel(plane.x, boxMin.x, boxMax.x);
  const float y = Math::FSel(plane.y, boxMin.y, boxMax.y);
  const float z = Math::FSel(plane.z, boxMin.z, boxMax.z);

  return (x*plane.x + y*plane.y + z*plane.z + plane.w) < 0;
}
```

Listing 6.1: View Frustum Culling

6.1.3 BVH Plane Masking

If a node is completely inside a certain plane of the view-frustum, then the bounding volumes of the node's children also lie completely inside the same plane [7]. When traversing the BVH, a simple bit-mask can then be used to identify which planes need not be tested against. The deeper we are in the hierarchy, the more likely it is that fewer planes need to be tested.

There are more optimizations, but some of them very specific, pertaining only to certain situations [5]. Therefore, some of them were not implemented in the Daedalus engine.

6.2 Stop-and-Wait Occlusion Culling

This algorithm demonstrates occlusion culling in its simplest and purest form. As such, it is easy to implement and aids in debugging more advanced occlusion culling algorithms, but is likely to be slower than any other algorithm.

Basically, the algorithm first sorts all objects to be rendered based on their distance to the camera. The objects are then rendered in front-to-back order, while performing the following steps for each object:

1. Disable writes to the depth and color buffer.
2. Start an occlusion query, and draw the object's bounding volume.
3. Stop the occlusion query, and wait until the bounding volume has been drawn.
4. Retrieve the number of pixels rendered in Step 2 using the occlusion query.
5. If the number of visible pixels is greater than a certain threshold, the object is considered to be visible, and the algorithm continues with Step 6. Otherwise, the object is invisible, and is not rendered at all.
6. Enable writes to the depth and color buffer again.
7. Render the object.

Steps 1-4 are commonly referred to as **issuing an occlusion query**. As can easily be seen, this will give perfect occlusion culling as long as the threshold in Step 5 is set to be zero pixels. Any other threshold will produce small rendering artifacts, which however might be unnoticed, as long as the threshold is reasonably small (approx. 10 pixels).

One thing to keep in mind is that objects whose bounding volume intersects the frustum's near plane must be rendered immediately. Otherwise, the occlusion query might report zero pixels being rendered due to backfaces being culled, while the object might very well be visible. Figure 6.3 shows a scene rendered using occlusion culling.

The way occlusion queries are used in this algorithm severely hinders the performance gained from occlusion culling. By constantly waiting for the GPU to report its results to the CPU, they both are effectively no longer working in parallel, but rather in an alternating fashion. The CPU is stalled while waiting for the result of the occlusion query, and the GPU is starving for new commands after the CPU has received a result. This procedure is depicted in Figure 6.2.

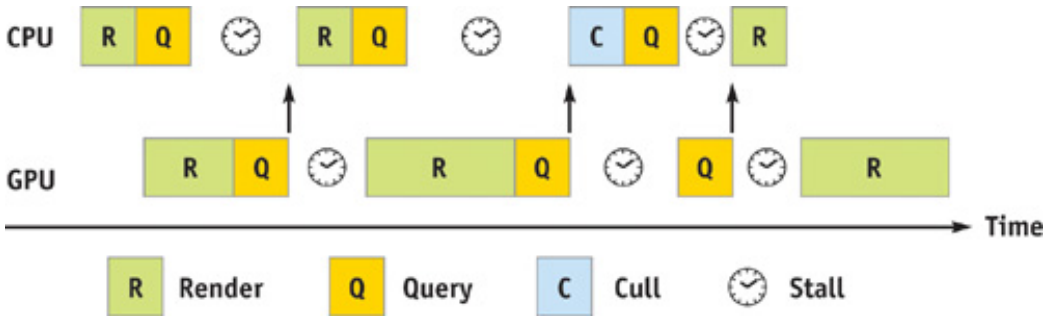


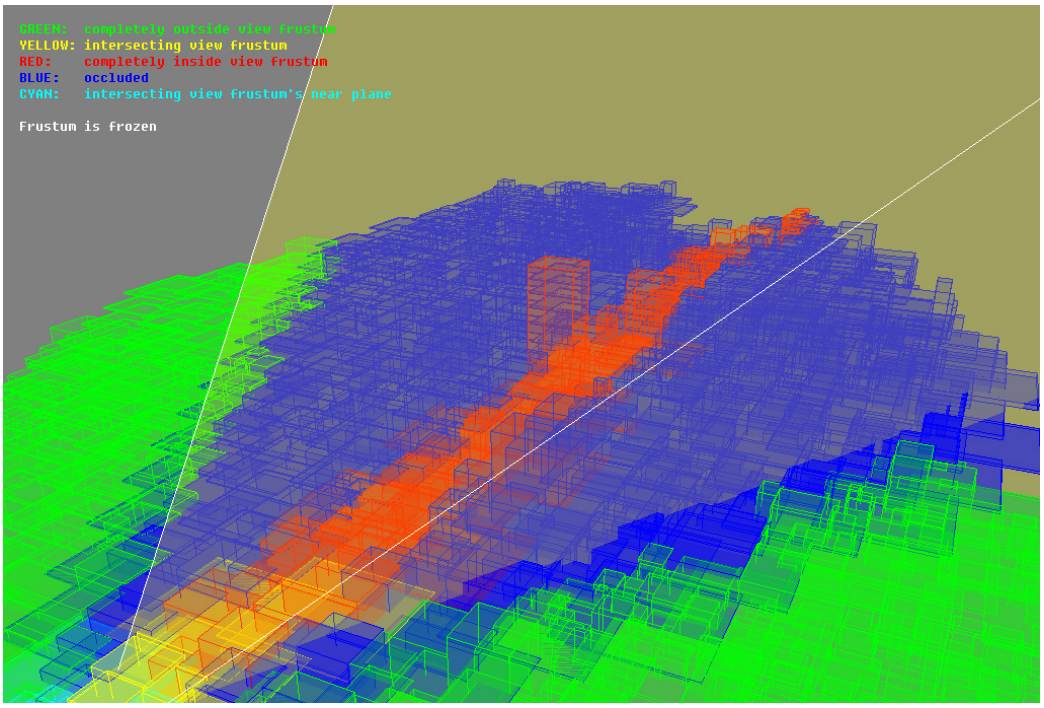
Fig. 6.2: CPU stalls and GPU starvation. (Image courtesy of Michael Wimmer)

Furthermore, the algorithm introduces even more overhead because of the following two reasons:

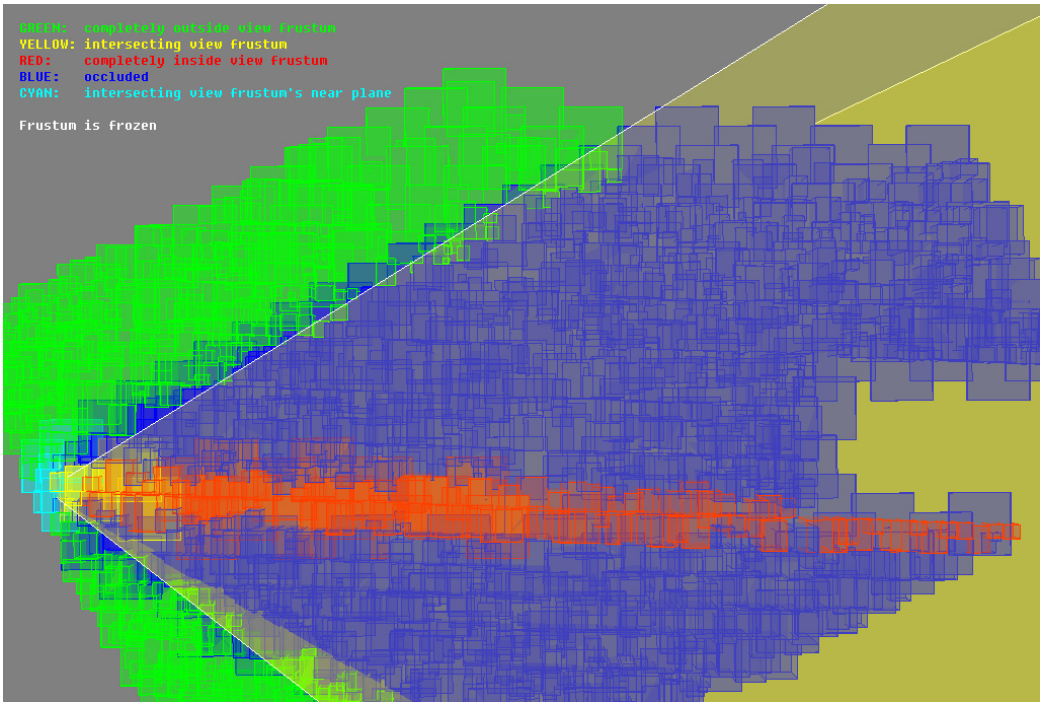
- An occlusion query must be issued for each rendered object. Depending on the number of objects, the number of queries can skyrocket.
- Additional geometry must be rendered for each object, which adds further draw calls and possible render state changes.

However, depending on the platform, even this naive algorithm still yields better performance than just frustum culling for occlusion-heavy scenes, as can be seen in the next chapter.

Listing 6.2 gives an implementation of the Stop-and-Wait algorithm.



(a) Occlusion culling using AABBs



(b) Occlusion culling using AABBs, seen from above

Fig. 6.3: Scene rendered using occlusion culling. Blue AABBs are completely occluded, cyan AABBs intersect the frustum's near plane.

```
void VisibilitySystemSAW::Render(void)
{
  sortedObjs.clear();

  for (size_t i=0; i<objects.size(); ++i)
  {
    // calculate view-space Z approximation using
    // center of AABB, and add to list
    AddObject(objects[i]);
  }

  SortObjects<FrontToBack>();

  SortedObjects::iterator it;
  for (it=sortedObjs.begin(); it!=sortedObjs.end(); ++it)
  {
    RenderObject* object = it->first;

    // if the object intersects the near plane,
    // it must be assumed that it is visible
    if (frustum.IsIntersectingPlane(object->GetAABB(),
                                    Frustum::PLANE_NEAR))
    {
      object->Render();
      continue;
    }

    {
      RenderState colorWrite(RenderStates::COLORWRITE_FALSE);
      RenderState zWrite(RenderStates::ZWRITE_FALSE);

      query->Begin();
      RenderSolidAxisAlignedBox(object->GetAABB());
      query->End();
    }

    while (!query->IsFinished())
    {
      // busy-wait for the query to finish
    }

    if (query->GetNumVisiblePixels() > 0)
    {
      // object is not occluded -> render it
      object->Render();
    }
  }
}
```

Listing 6.2: Stop-and-Wait Occlusion Culling

6.3 CHC – Coherent Hierarchical Culling

CHC [9] tries to remedy the shortcomings of naive occlusion culling algorithms such as Stop-and-Wait by taking advantage of spatial and temporal coherence. The latter can be taken advantage of by using the visibility information from the previous frame when rendering the current frame. Assuming moderate camera movement, it is highly likely that most BVH nodes stay visible after they have been visible the last frame, and occluded nodes are similarly going to stay occluded in the current frame. Hence, as an initial guess, the algorithm assumes a node's visibility to stay the same from the previous to the current frame.

However, because the visibility classification of nodes will sooner or later change, the algorithm has to verify whether those estimates have been correct or not. If not, it needs to rectify its choices.

Depending on how a node's visibility has changed during two consecutive frames, one of the following two scenarios can occur:

- The node has been visible the previous frame, but is actually occluded in the current frame. Although the node has been processed unnecessarily, all that needs to be done is to update the node's classification.
- The node has been occluded the previous frame, but became visible in the current frame. This situation needs to be remedied by rendering the node, otherwise objects will be missing in the current frame.

Classification of nodes is again done by issuing occlusion queries, like in the Stop-and-Wait algorithm. By cleverly interleaving the traversal of the spatial hierarchy and handling the results of issued queries, we arrive at the original CHC algorithm:

- Traverse the BVH in a front-to-back order.
- If the node is an interior node and was visible in the previous frame, process it immediately.
- For all other nodes, issue an occlusion query, and store this query in a queue called the *query queue*. Additionally, if the node is a leaf node which has been visible the previous frame, render it immediately.
- Check the results of the query queue after each visited node. If a node was visible in the previous frame, and became occluded, simply update its classification. If a node was occluded in the previous frame, and became visible, process it immediately.

One crucial observation is that no query needs to be issued for interior nodes that have been visible in the previous frame, because their classification can be deduced from its children at the end of each frame. This decreases the number of issued queries, and potentially saves a huge amount of fill rate.

Another useful observation is that previously visible leaf nodes will always be rendered, hence it does not make sense to use a bounding volume when issuing the occlusion query, but rather the actual geometry. This saves additional fill rate, draw calls and state changes. More in-depth algorithmic details can be found in [9] and [72].

Like other algorithms, CHC also has its disadvantages:

- In scenes with average occlusion, CHC becomes even slower than view frustum culling because of the additional overhead caused by issuing many occlusion queries and changing render states. This is especially true on platforms which exhibit a high query overhead, such as the PC.
- CHC is hard to integrate into rendering engines having optimized rendering loops, because the issuing of queries, state changes and draw calls is arbitrarily interleaved. This does not allow for highly optimized sorting of shaders, materials, textures or state changes, as is often the case in rendering engines. Although this can be somewhat alleviated by drawing a depth-only pass first, this is not possible in vertex-bound scenes.

6.4 CHC++ – Coherent Hierarchical Culling Revisited

CHC++ [42] [8] describes an improved version of the CHC algorithm, trying to eradicate its shortcomings by introducing further optimizations and batching strategies.

6.4.1 Reducing state changes

One of the biggest problems in the original CHC algorithm is the fact that state changes are required for every issued occlusion query. States to be changed include depth and color writes, as well as a change of shaders. Depending on the platform, its API and driver implementation, such state changes can be costly, and should be kept to a minimum during rendering.

CHC++ proposes several strategies for batching occlusion queries and performing other operations, which are described in the following sections.

6.4.2 Batching previously invisible nodes

By introducing a new queue for previously invisible nodes (called the *i-queue*), queries to be issued can be added to this queue until a user-defined batch size b is reached. Only then are the rendering states changed and the queries issued for each node contained in the i-queue.

Essentially, the i-queue constitutes a trade-off between performing less state changes, and introducing additional query latency, because visibility changes need longer to be detected. Therefore, the optimal value of the parameter b depends on the scene's complexity, and the engine's capabilities to handle state changes and caching thereof.

6.4.3 Batching previously visible nodes

Comparable to the i-queue for previously invisible nodes, CHC++ introduces another queue called the *v-queue* for previously visible nodes. Because the results of the queries for these nodes need to be available only at the end of each frame, those nodes can be batched into the v-queue. Nodes are added to the v-queue as long as either nodes can be traversed or results from previously issued queries are available.

As soon as the CHC algorithm is idle waiting for query results to arrive, nodes from the v-queue are processed. This results in batching of state changes for previously visible nodes.

6.4.4 Batching draw calls

By introducing a third queue called the *render queue*, draw calls can be batched and state sorting can be performed by the rendering engine. Whenever CHC++ needs to render a node, it gets added to the render queue instead. All nodes from this queue can then be rendered by a single engine API call, and the engine can perform whatever sorting it wants to.

6.4.5 Visible node randomization

One optimization to reduce the number of occlusion queries is to make the assumption that a visible leaf will stay visible for at least n frames. Although this will reduce the number of issued queries by a factor of n, it introduces frame rate spikes whenever queries get issued in the same frame, as is the case when different nodes become visible in the same frame.

This problem of temporally aligned nodes can be solved by randomizing the amount of assumed visible frames only the first time a query is issued.

As long as the node stays visible, a query will then be issued only every n-th frame. The optimal choice for the parameter n depends on the scene complexity, rendering engine and platform capabilities.

6.4.6 Multiqueries

Multiqueries provide another strategy to cut down on the number of used occlusion queries. By grouping occlusion queries for invisible nodes which are likely to stay invisible, a single query can be used to determine the new visibility classification for each node belonging to such a group. At worst, one occlusion query is wasted for each group if one of the nodes becomes visible.

Determining which invisible nodes are grouped together is done by evaluating a cost-benefit heuristic. The exact details can be found in the original paper [42] and a follow-up article [8].

6.5 Optimized CHC++

While CHC++ provides a significant enhancement over CHC, it can still be optimized to each platform's peculiarities by tweaking the parameters b and n. The original CHC++ algorithm was developed with common PC hardware and drivers/APIs in mind, and proposes values for b to be between 20-80, and n to be between 5-10.

Although CHC++ works excellently on the PC platform, first results on other platforms show that CHC++ sometimes performs worse than hierarchical view frustum culling for certain viewpoints, although not much. The reason for this is the fact that API overhead is much less on console platforms compared to the PC, hence the algorithmic requirements and optimizations must be slightly shifted.

According to the DirectX documentation [19], flushing the command buffer on the PC causes a kernel mode transition, which incurs a large penalty when issuing occlusion queries. Measurements on the console platforms show that occlusion queries are more than twice as fast on both the Xbox360 and the PlayStation 3 compared to the PC, and have even less overhead when using our custom-made occlusion queries described in Section 4.4.2.

6.5.1 Finding optimal parameters

Because it is desirable to have one algorithm which outperforms all others in almost any situation, a tool was developed in order to collect the optimal

CHC++ parameters for a scene. The tool itself is a command-line tool which utilizes the Daedalus engine, and renders the scene from a large amount of different viewpoints, using different values for both b and n according to minimum and maximum values for both parameters given as command-line arguments.

The viewpoints can be predefined or follow along a camera-path, e.g. depicting the player's progression through a level in a game scenario. Additionally, replays can be used as well, making it possible to use automatically recorded walkthroughs rather than manually built camera-paths for viewpoints.

Values for b and n are determined by starting at average values for both, and searching the whole range of possible values for values which will yield better performance, using binary search on the range of values.

For each combination of parameters b and n, the tool renders the scene from all viewpoints, and logs the time needed for rendering into a CSV (comma-separated values) file. The values contained in the CSV can then be evaluated offline, which allows for using optimized values for different scenes on each platform.

The optimal values for each of the CHC++ parameters are presented in the next chapter.

6.5.2 Fixed-size stack

The CHC++ algorithm makes heavy use of common data structures such as stacks and queues. Even though each standard STL implementation provides both containers, it has proven to be beneficial to implement a custom fixed-size stack and queue. Using containers with a fixed size makes sure that absolutely no memory allocations happen during the CHC++ algorithm, and that all data is stored contiguously in memory. Both traits are crucial for performance.

Listing 6.3 shows the implementation of the fixed-size stack. The macro **INLINE** ensures that the compiler inlines a certain method, on each platform.

6.5.3 Fixed-size queue

The fixed-size queue is internally implemented as a ring-buffer of elements, using separate indices for read and write operations. Additionally, wrapping at the end of the ring-buffer is implemented using branch-free operations instead of a simple if-statement, making this implementation especially fast

```
template <typename T>
class fixed_size_stack
{
public:
  fixed_size_stack(void) : data(NULL), dataSize(0), index(0)
      {}
  ~fixed_size_stack(void) { delete[] data; }

  void reserve(size_t numElements)
  {
    delete[] data;
    dataSize = numElements;
    data = new T [dataSize];
  }

  INLINE void clear(void) { index = 0; }

  INLINE void push(const T& value) { data[index++] = value; }

  INLINE void pop(void) { --index; }

  INLINE const T& top(void) const { return data[index-1]; }

  INLINE bool empty(void) const { return (index == 0); }

  INLINE size_t size(void) const { return dataSize; }

private:
  T* data;
  size_t dataSize;
  size_t index;
};
```

Listing 6.3: Fixed-size stack

on platforms with long pipelines and almost no branch prediction, such as some console platforms.

Listing 6.5 shows the implementation of the fixed-size queue. Listing 6.4 shows the branch-free implementation of wrapping values, taken from an article from Mike Acton [10].

```
INLINE static uint32_t WrappingInc(const uint32_t value, const
    uint32_t minValue, const uint32_t maxValue)
{
  const uint32_t result_inc = value + 1;
  const uint32_t max_diff = maxValue - value;
  const uint32_t max_diff_nz = (uint32_t)((int32_t)(max_diff |
      -max_diff ) >> 31);
  const uint32_t max_diff_eqz = ~max_diff_nz;
  const uint32_t result = (result_inc & max_diff_nz) | (
     minValue & max_diff_eqz);
  return (result);
}
```

Listing 6.4: Branch-free wrapping increment

```
template <typename T>
class fixed_size_queue
{
public:
  fixed_size_queue(void) : data(NULL), dataSize(0), readIndex
      (0), writeIndex(0) {}
  ~fixed_size_queue(void) { delete[] data; }

  void reserve(uint32_t numElements)
  {
    delete[] data;
    dataSize = numElements;
    data = new T [dataSize];
  }

  INLINE void clear(void) { readIndex = 0; writeIndex = 0; }

  INLINE void push(const T& value) { data[writeIndex] = value;
       writeIndex = WrappingInc(writeIndex, 0, dataSize); }

  INLINE void pop(void) { readIndex = WrappingInc(readIndex,
      0, dataSize); }

  INLINE const T& front(void) const { return data[readIndex];
      }

  INLINE bool empty(void) const { return (readIndex ==
      writeIndex); }

  INLINE uint32_t size(void) const { return dataSize; }

private:
  T* data;
  uint32_t dataSize;
  uint32_t readIndex;
  uint32_t writeIndex;
};
```

Listing 6.5: Fixed-size queue

Chapter 7

Results

This chapter provides results gathered for all implemented algorithms on each platform. Additionally, the results are discussed and compared between different platforms.

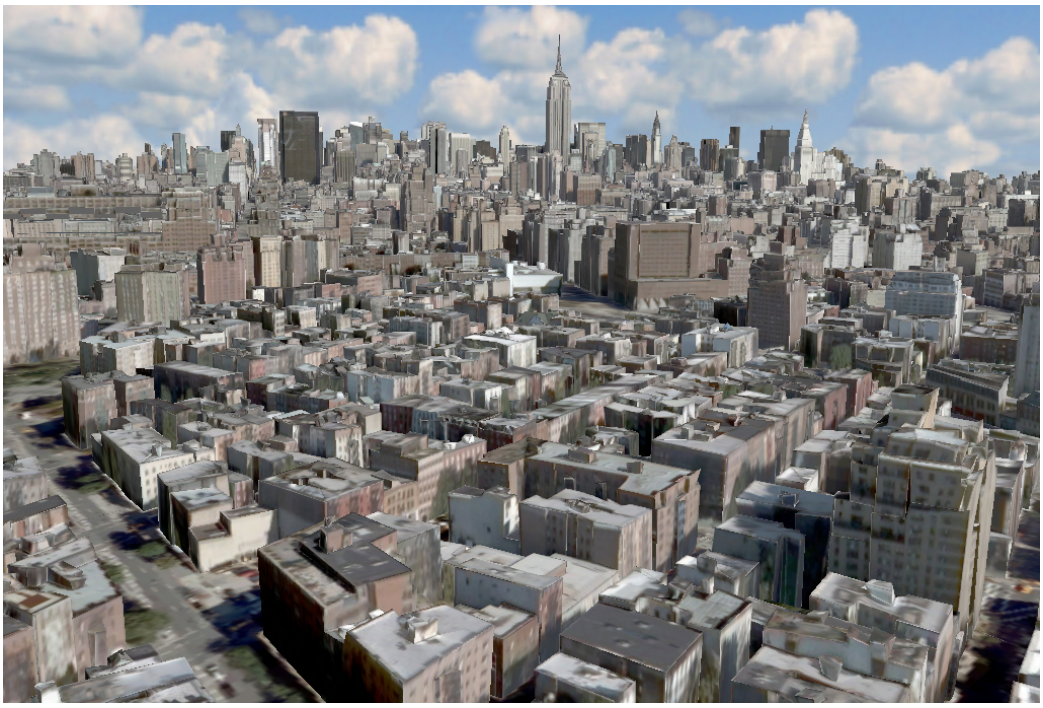
7.1 Test Environment

The test scene is a model of New York City which consists of roughly one million triangles, and offers a large amount of occlusion between the buildings. Parts of the scene are shown in Figure 7.1.

This scene is used to gather the following data during rendering:

- Frametime: The time it takes in milliseconds to render the scene from a certain viewpoint.
- Draw calls: The number of draw calls issued for rendering the scene itself, and additional bounding geometry (if any).
- Issued queries: The number of queries issued (if any).
- Triangles: The number of triangles rendered for the scene itself, and additional bounding geometry (if any).
- State changes (SW): The number of render state changes in software. This equals the number of local RenderState variables constructed.
- State changes (HW): The number of render state changes in hardware. This equals the number of API calls to set a new state after caching of changes has been performed by the engine.

In order to provoke both a best-case and worst-case scenario for the algorithms, all data is gathered from ground-level as well as sky-level viewpoints, respectively. Viewpoints above the ground exhibit the largest amount of



(a) Part of NYC

(b) Part of NYC

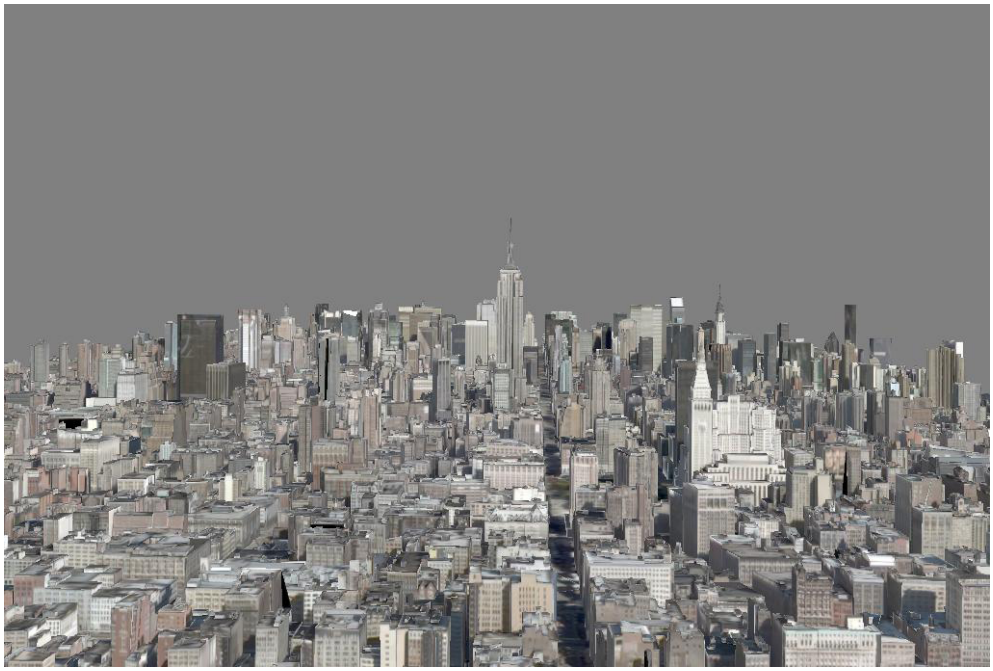
Fig. 7.1: New York City Scene.

occlusion in the test scene, while viewpoints in the sky convey the most overhead for some algorithms.

All presented results are gathered from multiple viewpoints following a camera path, in order to be able to exploit temporal coherence. All following data is gathered by averaging the measured data of all viewpoints. Two representative viewpoints for both ground-level and sky-level camera paths are shown in Figure 7.2.

Both used camera paths moved the camera from one end of the model of New York City to the other, taking 30 seconds at a frame rate of 60 Hz.

(a) Ground-level viewpoint



(b) Sky-level viewpoint

Fig. 7.2: Sample viewpoints.

7.2 PC platform

The following results were gathered on an Intel Core 2 Duo CPU at 3 GHz with 2 GB RAM, and a Geforce 9500 GT GPU with 512 MB RAM. On the PC, no results for the optimized CHC++ algorithm are provided because its parameters are heavily influenced by the underlying hardware. The CHC++ algorithm uses values of n=20 and b=50 on all platforms.

Ground level						
	Time	*Triangles*	*Draw calls*	*Issued queries*	*SW state changes*	*HW state changes*
No culling	8.25	1,059,654	1,898			
HVFC	4.90	747,936	1,235			
Stop-and-Wait	18.40	81,083	1,294	1,227	7,362	361
CHC	4.75	260,000	510	300	600	180
CHC++	2.30	103,600	195	45	40	10

Sky level						
	Time	*Triangles*	*Draw calls*	*Issued queries*	*SW state changes*	*HW state changes*
No culling	8.20	1,059,654	1,898			
HVFC	4.95	775,077	1,295			
Stop-and-Wait	32.20	563,496	2,125	1,295	7,770	4,985
CHC	10.90	736,000	1,434	500	400	300
CHC++	5.10	670,000	1,180	60	200	14

Tab. 7.1: Results on the PC platform.

As can be seen in Table 7.1, hierarchical view-frustum culling almost doubles the performance compared to performing no culling at all. As expected, stop-and-wait occlusion culling is unusable on the PC platform because of the high cost of occlusion queries and state changes, even though the least amount of triangles is rendered using this algorithm. For worst-case viewpoints with only few occlusions, almost 5000 state changes are performed at the driver level, respectively on the hardware.

While CHC performs well in situations offering a lot of occlusion, it performs poor for worst-case scenarios, as stated in the original paper. Still, even

in moderate scenarios CHC offers only the same performance as hierarchical view-frustum culling, even though only a third of all triangles is actually rendered. On the other hand, CHC++ performs really well in all situations, even worst-case. Therefore, it can be used as a drop-in replacement for every other culling algorithm on the PC platform.

Figure 7.3 shows a comparison of the different algorithms.

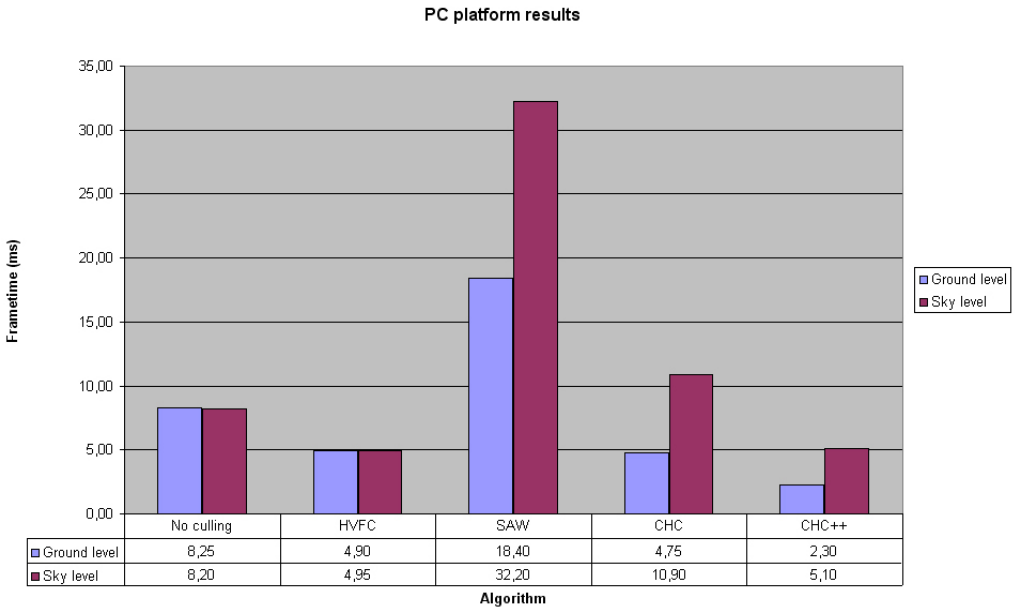


Fig. 7.3: Results on the PC platform.

7.3 Xbox360 platform

The optimized CHC++ algorithm uses values of n=12 and b=20 on the XBox360 platform for the NYC scene.

Ground level						
	Time	*Triangles*	*Draw calls*	*Issued queries*	*SW state changes*	*HW state changes*
No culling	14.25	1,059,654	1,898			
HVFC	10.90	747,936	1,235			
Stop-and-Wait	16.25	68,056	1,297	1,227	7,362	377
CHC	4.85	230,000	470	300	600	150
CHC++	3.15	98,000	180	45	34	8
Optimized CHC++	3.00	99,000	190	50	40	16

Sky level						
	Time	*Triangles*	*Draw calls*	*Issued queries*	*SW state changes*	*HW state changes*
No culling	14.25	1,059,654	1,898			
HVFC	11.40	775,077	1,295			
Stop-and-Wait	29.95	549,923	2,125	1,295	7,770	4,983
CHC	13.20	722,000	1,380	500	400	300
CHC++	11.40	665,000	1,160	60	100	12
Optimized CHC++	11.25	665,000	1200	90	120	24

Tab. 7.2: Results on the XBox360 platform.

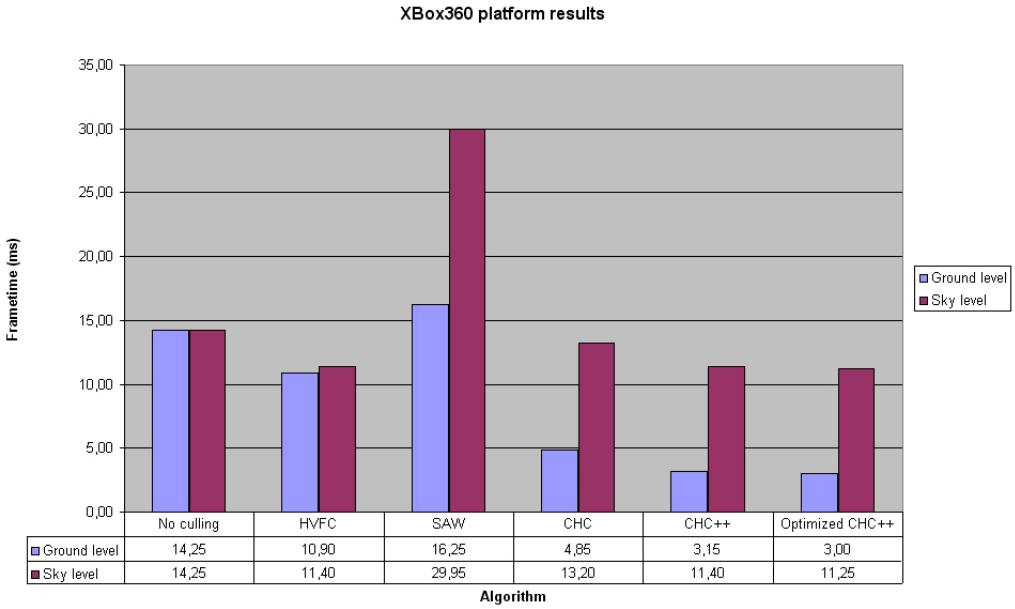
The results on the XBox360 are similar to those on the PC, with a few minor changes. Generally speaking, the performance gap between hierarchical view-frustum culling and CHC is smaller than on the PC, for both ground and sky level viewpoints.

Similarly, CHC++ provides a significant enhancement over CHC, albeit not as large as on the PC. This stems from the fact that occlusion queries and state changes are way cheaper on the XBox360, hence the number of rendered triangles has more impact on the final performance, also due to the

fact that the XBox360 hardware is less powerful compared to the PC.

Optimized CHC++ manages to gain an extra 1-5% of performance, although more queries are issued and more state changes have to be performed on average. In all the viewpoints used during testing, there was never a situation where optimized CHC++ performed worse than hierarchical view-frustum culling.

Figure 7.4 shows a comparison of the different algorithms.



	No culling	HVFC	SAW	CHC	CHC++	Optimized CHC++
Ground level	14,25	10,90	16,25	4,85	3,15	3,00
Sky level	14,25	11,40	29,95	13,20	11,40	11,25

Fig. 7.4: Results on the XBox360 platform.

7.4 PlayStation 3 platform

On the PlayStation 3, values of n=8 and b=30 were used for the optimized CHC++ algorithm.

Ground level						
	Time	*Triangles*	*Draw calls*	*Issued queries*	*SW state changes*	*HW state changes*
No culling	43.00	1,059,654	1,898			
HVFC	30.80	747,936	1,235			
Stop-and-Wait	38.40	80,460	1,293	1,227	7,400	370
CHC	14.60	230,000	500	300	620	210
CHC++	7.85	102,000	195	45	110	25
Optimized CHC++	7.75	102,000	190	50	120	29

Sky level						
	Time	*Triangles*	*Draw calls*	*Issued queries*	*SW state changes*	*HW state changes*
No culling	41.00	1,059,654	1,898			
HVFC	28.50	775,077	1,295			
Stop-and-Wait	110.00	563,496	2,130	1,295	7,808	5,038
CHC	35.00	736,000	1,300	500	450	280
CHC++	28.50	672,000	1,180	60	210	29
Optimized CHC++	27.70	675,000	1200	110	350	33

Tab. 7.3: Results on the PlayStation 3 platform.

Similar to both the PC and XBox360 platform, it is quite clear that the CHC++ algorithm manages to dramatically reduce both the number of issued occlusion queries as well as the number of performed state changes, compared to all other occlusion culling algorithms.

Because the PlayStation 3 is the platform offering the weakest GPU compared to its CPU (PPU), the naive stop-and-wait algorithm performs exceptionally bad. Furthermore, the architecture of the PlayStation 3 benefits the most from rendering large command-buffers at once, and as such its render-

ing performance is weakened by submitting hundreds of small draw calls, respectively command-buffers.

Cutting down the number of draw calls is also what makes CHC and CHC++ perform so good on this platform, at least for ground level viewpoints. There, CHC and CHC++ are two times and four times as fast as ordinary view-frustum culling.

Figure 7.5 shows a comparison of the different algorithms.

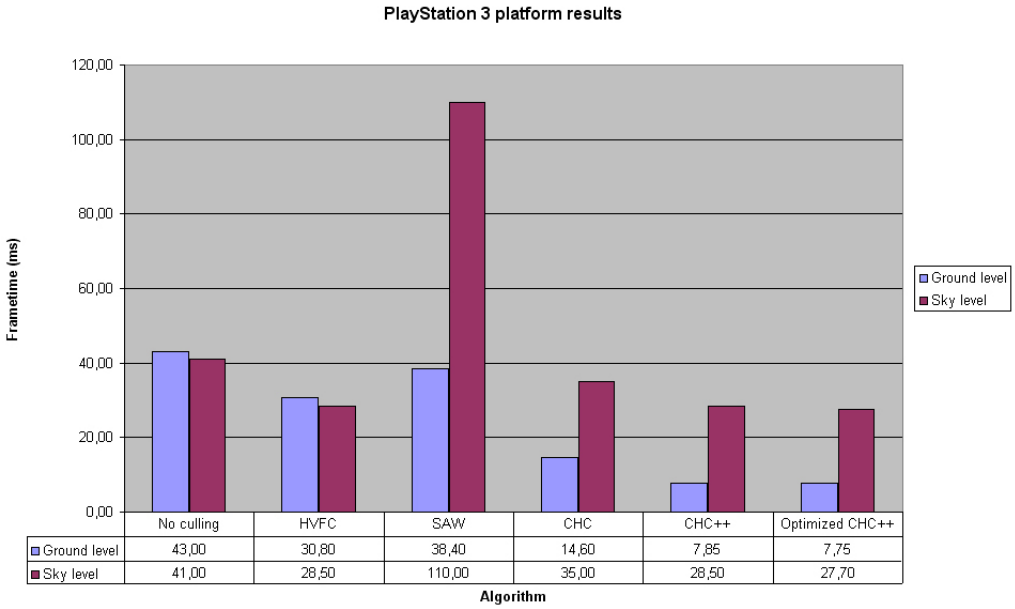


Fig. 7.5: Results on the PlayStation 3 platform.

7.5 Wii platform

Values of n=7 and b=25 have been used for the optimized CHC++ algorithm on the Wii platform.

Ground level						
	Time	*Triangles*	*Draw calls*	*Issued queries*	*SW state changes*	*HW state changes*
No culling	113.55	1,059,654	1,898			
HVFC	81.60	747,936	1,235			
Stop-and-Wait	32.25	64,700	1,293	1,227	7,362	353
CHC	28.85	212,000	180	300	650	190
CHC++	14.80	102,000	200	45	80	8
Optimized CHC++	14.45	105,000	350	45	150	16

Sky level						
	Time	*Triangles*	*Draw calls*	*Issued queries*	*SW state changes*	*HW state changes*
No culling	113.55	1,059,654	1,898			
HVFC	84.60	775,077	1,295			
Stop-and-Wait	82.60	536,000	2,100	1,295	7,770	4,839
CHC	87.00	736,000	1,320	550	450	250
CHC++	75.80	658,000	1,150	65	140	12
Optimized CHC++	73.90	658,000	1,150	130	160	20

Tab. 7.4: Results on the Wii platform.

The Wii platform offers one peculiarity, and that is the stop-and-wait algorithm performing so well. This is due to the following reasons:

- Extremely low overhead for occlusion queries: Even issuing 1000 occlusion queries takes up only 1 ms, which is marginal compared to the whole frametime.
- Weakest GPU of all platforms: Because the GPU is relatively weak compared to the other platforms, it is beneficial to cut down the number

of rendered triangles, even if a lot of state changes have to be performed.

While even the naive implementation performs well, CHC++ still provides a performance gain of up to 200% compared to this algorithm, for viewpoints exhibiting a lot of occlusion. At ground-level viewpoints, this enables the Daedalus engine to render the NYC scene with a steady 60 frames per second, which is quite an impressive feat considering the fact that the scene consists of more than a million triangles. Even AAA-games on the Wii usually render no more than 250,000-300,000 triangles at 30 frames per second.

Figure 7.6 shows a comparison of the different algorithms.

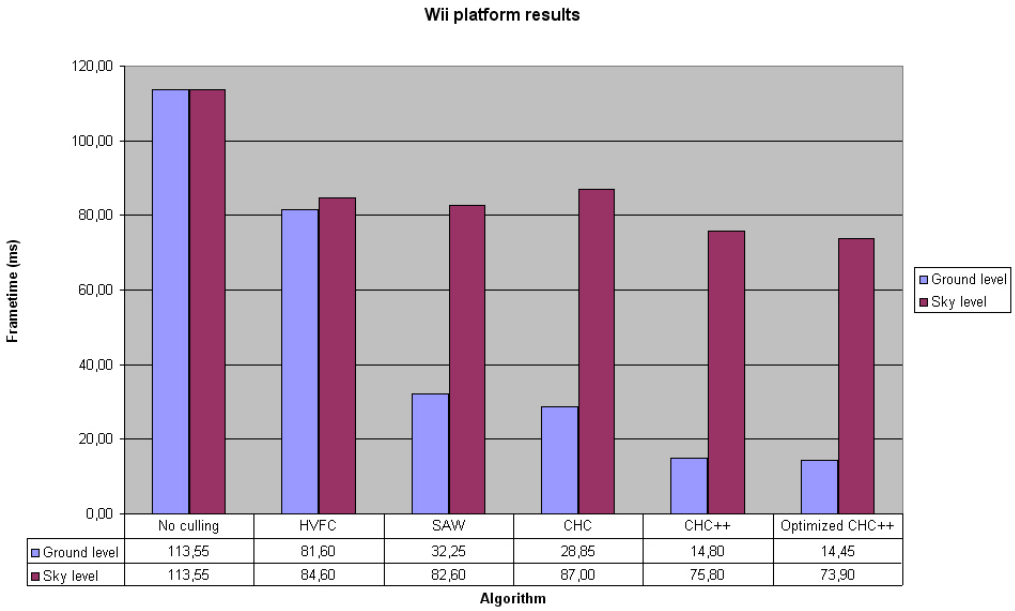


Fig. 7.6: Results on the Wii platform.

7.6 Comparison

Considering the fact that hierarchical view-frustum culling is still the most employed culling algorithm in games today, and CHC++ performed the best on all platforms, those two algorithms are compared in Figures 7.7 and 7.8.

Evidently, CHC++ offers tremendous performance gains compared to traditional culling algorithms. Even in worst-case scenarios, CHC++ manages to perform at least equal to or even slightly better than other algorithms.

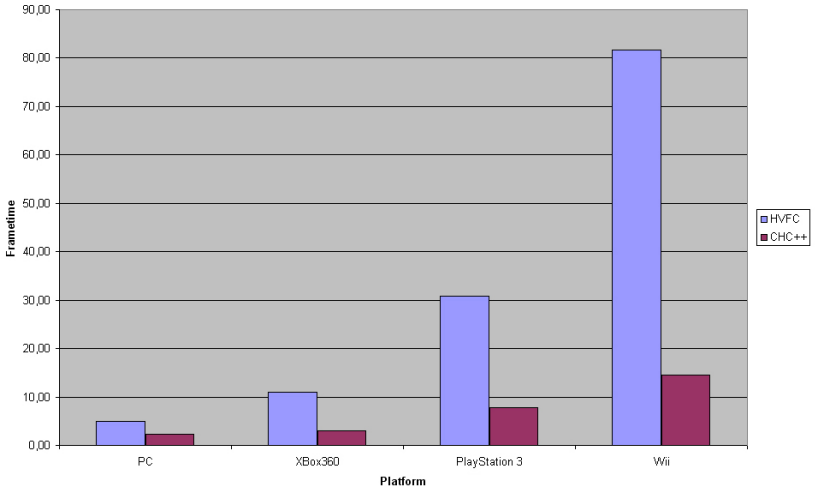


Fig. 7.7: Comparison of results for ground-level viewpoints.

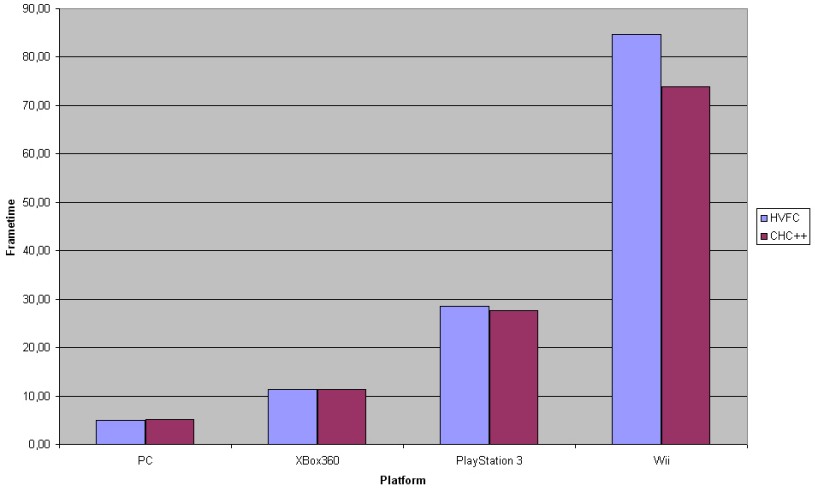


Fig. 7.8: Comparison of results for sky-level viewpoints.

Tailoring CHC++ to specific scenes and platforms can gain another few percent performance, although this is only possible on fixed hardware.

Chapter 8

Summary and future work

In this last chapter, a summary of the thesis is given, and ideas for future algorithms are explored.

8.1 Conclusion

In this thesis, the design and implementation of parts of a real-world multi-platform rendering engine as well as visibility algorithms and their implementation was presented.

The first chapters of the thesis covered general aspects of rendering engines, and provided an overview of other available engines, both free and commercial ones. The majority of the first part of the thesis concerned itself with vastly differing hardware capabilities, low-level rendering engine design, multi-platform rendering issues, CPU-GPU synchronization and platform API differences. This part showed that it is no easy challenge to design a rendering engine for four different platforms, three of them being exceedingly different to each other. Thus, possible solutions for solving several design problems were given.

Building upon the design foundation laid out in the aforementioned chapters, the final chapter of the first part detailed the low-level implementation of the Daedalus engine, describing the building blocks needed for implementing the visibility algorithms described in the second part of the thesis.

Consecutively, strategies for solving the problem of CPU-GPU synchronization were deployed, and implementation details for render states, immediate mode rendering and occlusion queries were given. This part of the thesis concluded with devising a new algorithm for enabling occlusion queries on the Nintendo Wii, showing a complete implementation thereof in the Daedalus engine.

The second part of the thesis introduced current state-of-the-art visibility algorithms, and provided parts of their implementation in the Daedalus engine, along with possible optimizations on console hardware. The achieved

results prove that integrating newer visibility algorithms into a real-world multi-platform engine is easily possible, whether the engine has been designed for this kind of algorithm in the first place or not, provided the engine is sufficiently structured.

Finally, the results speak for themselves and evidence that algorithms like CHC and CHC++ are worthy replacements for visibility algorithms employed in today's rendering engines, on all currently available major platforms.

8.2 Future work

Even though state-of-the-art visibility algorithms can be used on current-gen platforms, there is still a lot of room for improvement, especially concerning the occlusion culling algorithms described in this thesis.

Both CHC and CHC++ could be slightly improved by using the conditional rendering facilities available on the Xbox 360 and PlayStation 3 platforms. Conditional rendering allows primitives to be rendered based on the outcome of a previously issued query, completely without any CPU intervention. This could be used to reduce CPU stalls and idle-times.

Software rasterization

Another possibility would be to resort to software rasterization for occlusion culling. At the time of writing, almost every console game is GPU-limited, the power of the PlayStation 3 SPUs still needs to be completely harnessed, and multi-core CPUs are more and more becoming mainstream. Therefore, there is a lot of CPU/SPU power available for completing other tasks, such as rasterizing depth for occlusion culling algorithms. Especially SPUs are notorious for performing such tasks.

One of the reasons for CPU idle times in CHC and CHC++ is the fact that the result of an occlusion query is not available until all operations inside a query have been completed. When rendering primitives using software rasterization, many of those idle-times could be avoided by making use of early-out optimizations. After rasterizing a scanline, a simple check could test whether any pixel was written to the framebuffer, which would allow the algorithm to continue without having to wait for the primitive to be rendered completely.

Furthermore, software rasterization allows for other optimizations to be performed when rasterizing depth values only, such as rendering at lower resolutions, because the depth-buffer is not needed for subsequent GPU rendering. Thus, the rasterizing process can be carried out at e.g. only 1/4 of

the original resolution without introducing noticeable artifacts. In this case, the gathered visibility information is no longer exact, but provides a good approximation.

As an additional optimization, depth rasterization could be performed in 16-bit or fixed-point, depending on the platform's capabilities.

Finally, software rasterization does not need any render state changes at all, and puts less burden on the GPU's vertex unit. Using the GPU for color rasterization and the CPU for depth rasterization in the occlusion culling algorithms would therefore reduce CPU-GPU stalls, and completely eliminate some of the needed render state changes.

Commercial engines such as CryEngine 3 from Crytek [44] and Frostbite from DICE [4] already use software rasterization for the purpose of occlusion culling.

8.3 Main contributions

Chapter 3 provided an introduction to intricacies of console hardware programming, and pointed out difficulties in cross-platform engine development along with appropriate solutions. It showed how to solve reoccurring problems in professional engine development, as well as console-specific programming problems such as CPU-GPU synchronization.

Chapter 4 introduced a novel mechanism for making hardware occlusion queries available on platforms which do not natively support them by using hardware GPU metrics, interrupts and custom-made CPU-GPU synchronization, resulting in queries that are way faster than query functionality on other platforms. Furthermore, it provided an implementation of render state functionality using advanced C++ features, resulting in very fast and robust render states. Additionally, it introduced a multi-platform interface for immediate mode rendering, optimized using special capabilities of each platform.

Chapter 6 showed optimizations to current visibility culling algorithms using branch-free operations on console hardware, and additionally detailed a tool for evaluating optimal CHC++ parameters on fixed-hardware platforms such as consoles.

Chapter 8 presented possible future enhancements to the algorithms introduced in Chapter 6 by making use of specific hardware features and capabilities, potentially increasing the algorithms' performance even more.

List of Figures

List of Tables

List of Listings

Bibliography

[1] AI Implant. http://www.ai-implant.com/.

[2] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[3] Advanced Micro Devices. http://www.amd.com/uk/Pages/AMDHomePage.aspx.

[4] Johan Andersson. The Intersection of Game Engines and GPUs – Current and Future. Technical report, Graphics Hardware, 2008. http://www.graphicshardware.org/presentations/andersson-game_engines_and_GPUs.pptx.

[5] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools*, 5(1):9–22, 2000.

[6] Illuminate Labs Beast. http://www.illuminatelabs.com/.

[7] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, and Michael Shantz. Designing a pc game engine. *IEEE Comput. Graph. Appl.*, 18(1):46–53, 1998.

[8] Jiří Bittner, Oliver Mattausch, and Michael Wimmer. Game-engine-friendly occlusion culling. In Wolfang Engel, editor, *SHADERX7: Advanced Rendering Techniques*, volume 7, chapter 8.3. Charles River Media, March 2009.

[9] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum*, 23(3):615–624, September 2004. Proceedings EUROGRAPHICS 2004.

[10] Increment and decrement wrapping values. http://cellperformance.beyond3d.com/articles/2006/07/increment-and-decrement-wrapping-values.html.

[11] Cell Broadband Engine Resource Center, IBM. http://www.ibm.com/developerworks/power/cell/index.html.

[12] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[13] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 83–ff., New York, NY, USA, 1997. ACM.

[14] More C++ Idioms. http://en.wikibooks.org/wiki/More_C++_Idioms.

[15] CryEngine 3. http://www.crytek.com/technology/cryengine-3/specifications/.

[16] Cursed Mountain. http://cursedmountain.deepsilver.com.

[17] Mark DeLoura. Game engine showdown. *Game Developer*, 16(5):7–12, May 2009.

[18] Direct3D 10 Pipeline. http://msdn.microsoft.com/en-us/library/bb205123(VS.85).aspx.

[19] DirectX Query Documentation. http://msdn.microsoft.com/en-us/library/ee422167(VS.85).aspx.

[20] Michael Doggett. *Xenos: Xbox 360 GPU*. GDC Europe 2005, 2005.

[21] Microsoft DirectX. http://msdn.microsoft.com/directX.

[22] Microsoft DirectX 10. http://global.gamesforwindows.com/de-DE/aboutGFW/directX10.aspx.

[23] Eurographics - European Association for Computer Graphics. http://www.eg.org/.

[24] Emergent's Floodgate. http://www.emergent.net/Products/Gamebryo/Technical-Details/Floodgate/.

[25] FMOD. http://www.fmod.org/.

[26] Gamebryo. http://www.emergent.net/.

[27] Microsoft Gamefest 2008. http://www.xnagamefest.com/presentations08.htm.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994–1995.

[29] Game Networking Engine. http://www.gillius.org/gne/.

[30] Ned Greene. Detecting intersection of a rectangular solid and a convex polyhedron. pages 74–82, 1994.

[31] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238, New York, NY, USA, 1993. ACM.

[32] Valve Hammer. http://developer.valvesoftware.com/wiki/Category:Hammer:de.

[33] Al Hastings. Occlusion – Visibility determination for static and dynamic objects. Technical report, Insomniac Games R&D, 2007. http://www.insomniacgames.com/tech/articles/1107/occlusion.php.

[34] Havok. http://www.havok.com/.

[35] K. Hillesland, B. Salomon, A. Lastra, and D. Manocha. Fast and simple occlusion culling using hardware-based depth queries. Technical report, 2002.

[36] Forward declarations to template classes in the STL. http://msdn.microsoft.com/en-us/library/1af12yty(VS.80).aspx.

[37] Irrlicht Engine. http://irrlicht.sourceforge.net/.

[38] The Khronos Group. http://www.khronos.org/.

[39] LUA. http://www.lua.org/.

[40] David Luebke and Chris Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 105–ff., New York, NY, USA, 1995. ACM.

[41] Dominic Mallison and Mark DeLoura. *CELL: A New Platform for Digital Entertainment*. Game Developers Conference, March 2005, 2005.

[42] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Proceedings Eurographics 2008)*, 27(2):221–230, April 2008.

[43] Miles Sound System. http://www.radgametools.com/miles.htm.

[44] Martin Mittring. The Importance of Multi-Core for Game Development at Crytek. Technical report, Go parallel – Technical University Munich, 2008. http://www.crytek.com/fileadmin/user_upload/inside/presentations/2008/MunichIntel_eng.ppt.

[45] NVIDIA Corporation. http://www.nvidia.com.

[46] NVIDIA Register Combiners. http://developer.nvidia.com/object/registercombiners.html.

[47] Open Dynamics Engine. http://www.ode.org/.

[48] Codeplay's Offload. http://offload.codeplay.com/.

[49] Ogre. http://www.ogre3d.org/.

[50] OpenGL. http://www.opengl.org.

[51] OpenGL Architecture Review Board. http://www.opengl.org/about/arb/.

[52] Open Scene-Graph. http://www.openscenegraph.org.

[53] PCI-E Specification. http://www.pcisig.com/specifications/pciexpress/.

[54] Cedric Perthuis. *Introduction to the graphics pipeline of the PS3*. Eurographics 2006, Graphics Meets Games Talks, 2006.

[55] PhysX. http://www.nvidia.com/object/physx_new.html.

[56] PIMPL Idiom. http://www.gotw.ca/publications/mill04.htm.

[57] Stefan Popov, Iliyan Georgiev, Rossen Dimov, and Philipp Slusallek. Object partitioning considered harmful: space subdivision for bvhs. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 15–22, New York, NY, USA, 2009. ACM.

[58] Accurately Profiling Direct3D API Calls. http://msdn.microsoft.com/en-us/library/bb172234(VS.85).aspx.

[59] Sony Playstation 3 Element Interconnect Bus. http://www.sony.net/SonyInfo/technology/technology/theme/cell_01.htm.

[60] RakNet. http://www.jenkinssoftware.com/.

[61] CryEngine Sandbox. http://doc.crymod.com/SandboxManual/.

[62] Silicon Graphics International. http://www.sgi.com/.

[63] ACM SIGGRAPH - Special Interest Group on Graphics and Interactive Techniques. http://www.siggraph.org/.

[64] Source Engine. http://source.valvesoftware.com/.

[65] Sproing Interactive Media GmbH. http://www.sproing.com.

[66] Squirrel. http://squirrel-lang.org/.

[67] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13, New York, NY, USA, 2009. ACM.

[68] Eric Haines Tomas Akenine-Möller. *Real-Time Rendering, Third Edition*. A.K. Peters, Ltd., Wellesley, Massachusetts, USA, 2008.

[69] Unreal Engine 3. http://www.unrealtechnology.com/.

[70] Valve Hardware Survey. http://store.steampowered.com/hwsurvey.

[71] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society.

[72] Michael Wimmer and Jiří Bittner. Hardware Occlusion Queries Made Useful. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, March 2005.

[73] Microsoft Windows Vista. http://www.microsoft.com/germany/windows/windows-vista/.

[74] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff, III. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

Acknowledgements

> "The way your heart sounds makes all the difference, it's what decides if you'll endure the pain that we all feel. The way your heart beats makes all the difference in learning to live."
>
> – Dream Theater, "Learning To Live"

First, I want to thank Gerhard Seiler and Harald Riegler for making this diploma thesis possible by letting me use Sproing's development equipment.

Secondly, thanks to Michael Wimmer for giving me the opportunity to work on this topic the way I wanted it.

Sincere thanks to Oliver Mattausch for all the fruitful discussions, and for his help whenever I needed it.

Thanks to all my co-workers at Sproing for making everyday's life so much more enjoyable, with extra credit to "The guys from the Mario Room" – you guys rock!

Special thanks to all my friends who still bear with me, given the fact that I have spent so many evenings, nights, and weekends at the office.

Last but not least, I'd like to express my gratitude to my family for all their support over the past years.