Parallel Crowd Rendering

Master Thesis

Johannes Spohr

March 26, 2008

Research Group Programming Languages / Methodologies
Dept. of Computer Science and Electrical Engineering
Universität Kassel

Supervisors:
Dipl.-Inform. Björn Knafla
Prof. Dr. Claudia Leopold
Prof. Dr. Ing. Dieter Wloka, M. Eng.

Abstract

Parallel rendering in real-time applications is a challenge that many in the field of computer graphics are currently facing. Especially with large scenes containing tens of thousands of objects - *crowds* - scalability is important, which makes parallelism a definite requirement.

This thesis discusses several real-time rendering approaches and subsequently lays out the architecture of a lightweight parallel rendering toolkit called *Pace*. The Pace renderer uses CPU-level and CPU/GPU concurrency to exploit the parallel nature of rendering hardware with an emphasis on flexibility and scalability. Pace also includes techniques to accelerate OpenGL applications for real-time crowd rendering.

Declaration of Origin

I hereby declare that current Master of Science Thesis *Parallel Crowd Rendering* has been created by myself without the help of anybody and/or anything else, except the references that I have explicitly mentioned in this thesis.

Johannes Spohr
Kassel, March 2008

Contents

List of Figures

1 Introduction

In the development of video games and similar real-time graphics applications, the rapid display of densely populated scenery is a common requirement. Examples include urban environments, forests, or complex, artificially created compounds of objects for simulation purposes. The term *crowd rendering* describes the process of creating computer-generated images of scenes composed of large numbers of spatially close objects. While crowds are usually thought of as assemblies of people, crowd rendering is content agnostic. In many applications, human characters are the main digital content, but there are also cases where models of flora or more abstract entities are subject to crowd rendering. Figure 1.1 shows two scenes from a video game, which contains large fantasy landscapes, abundantly covered with vegetation, and populated by crowds of enemy creatures. The user can freely move his character and the virtual camera around in the game, while the rendered image is updated in real time.

In this context, the term *real-time* refers to an application which displays interactive graphics. The interactivity eliminates some possibilities for optimization which can be exploited in offline rendering, like high frame to frame coherency and precomputed lighting and visibility information. The images shown in figure 1.1 were generated by a video game system in less than $\frac{1}{60}$th of a second to give the player the smooth impression of real motion.

Massive real-time crowd rendering is a very demanding discipline of computer graphics. It puts an enormous strain on the rendering hardware, and also provides a challenge for digital content artists, as they are required to put out an immensely huge and varied collection of detailed and possibly animated models which satisfy the increasing demand for visual realism.

A typical 3D video game made for current PCs and game consoles spends most of the

Figure 1.1: Screenshots from the game "Kameo: Elements of Power"

available processing power on rendering. Graphics rendering hardware is usually a hybrid system consisting of a central processing unit (CPU) and a graphics processing unit (GPU), which communicate via a system bus. To be able to give graphics in games more and more detail, the performance of these components has been and will be steadily improved by hardware vendors. Judging from graphics hardware currently available to consumers, it is apparent that all vendors share the insight that building faster processors is less cost-efficient than employing parallel architectures with multiple processing cores. Although these platforms offer immense processing power, a disadvantage of parallelism as of now is that many developers have not yet grasped the implications of these architectures. Most of the libraries and tools developed for single-threaded graphics programming are not capable of exploiting the full potential of parallel hardware. Furthermore, it is hard to write applications which run equally well on different kinds of parallel platforms.

Thus, graphics application developers are looking for a rendering solution which hides the complexity of parallel hardware while providing scalability. This means that by adding additional processing cores, an application should be able to render proportionally more 3D content at the same speed. Unfortunately, it is currently impossible to achieve high scalability without optimizing a specific application to its target hardware architecture. A generic rendering solution might be able to scale well in many cases, but not for all applications on each platform. This problem is caused by the fact that a renderer is a pipelined system whose performance is limited by its bottleneck, which is the slowest part - or stage - of the pipeline. The location of the bottleneck depends both on the application and on the rendering hardware. Thus, careful balancing is required between the pipeline stages to efficiently utilize the hardware. For example, to reduce the load on the geometry processing stage, the application could choose to employ a more sophisticated culling method, which is responsible for rejecting invisible geometry, so it does not have to be processed by the remaining pipeline stages. However, this would probably increase the load on the culling stage.

While GPUs have received a tremendous performance increase over the last decade, their CPU counterparts have not been able to keep up [Owe05]. The GPU is usually used for rendering only, but the CPU is additionally tasked with updating animation state, capturing user input, driving potentially artificially intelligent agents and a plethora of other functions inherent in games or simulation applications. Shifting the load from the CPU to the GPU is not trivial, as the GPU is not a general purpose processor. GPUs

also vary in their computational abilities and rendering features, which makes it difficult to implement the same techniques for different models of graphics cards.

This thesis addresses these problems by analyzing popular approaches to real-time rendering and their feasibility for displaying crowd-level scenes on modern graphics hardware. The results are devised into a concept of a parallel renderer, *Pace* (Parallel Architecture for rendering Crowded Environments). Its focus lies on flexibility to account for the wide range of graphics applications and hardware architectures. Pace does not force a certain rendering strategy upon the application, but is designed to be extensible and customizable, and very lightweight. Its purpose is to serve as an architecture for research on parallel rendering and for applications requiring a very customizable rendering solution.

Chapter 2 starts with an overview of rendering concepts, introduces rendering hardware and presents a selection of approaches to real-time rendering and their inherent limitations. This leads to the description of the design of *Pace* in chapter 3. It is supplemented by chapter 4, which addresses notable implementation details. Chapter 5 shows the results obtained and compares them to the initial expectations. The conclusion of the thesis follows in chapter 6, which also mentions possibilities of further work and research on the subject.

2 State of the Art

This chapter presents currently used methods for rendering large, detailed scenes in real time. It contains an overview of rendering hardware and software, but tries to place more emphasis on the general concepts than describing specific techniques. Its purpose is to show why real-time computer graphics is such a complex, yet interesting field of software development. It also motivates some of the design decisions which are the topic of the next chapter.

After an introduction to graphics hardware and the rendering pipeline, different programming interfaces are presented and compared. At the end of the chapter, two existing crowd rendering solutions are briefly introduced.

2.1 Rendering Hardware

It is obvious that when discussing rendering hardware, the current state of the art will already be obsolete in a matter of months. Thus, the following sections will not mention any details of specific hardware. Instead, they provide an overview of the currently common rendering hardware architecture in consumer PCs.

When comparing graphics hardware of the last generation with the current one, there is always a steady increase in rendering performance. These improvements are currently accompanied by a shift in design principles. As power consumption and chip complexity were reaching a critical mass, hardware vendors looked for alternatives to higher clock speeds and transistor counts. Splitting the workload into chunks to be processed in parallel by multiple processors effectively increases the overall performance. It also increases efficiency in terms of cost and energy usage, which is important to the hardware manufacturer.

These changes occurred in the designs of both CPU (central processing unit, the main processor) and GPU (graphics processing unit, the processor on the graphics card). These are detailed in the following two sections.

2.1.1 CPU

The common CPU is a general purpose microprocessor, which is used to execute all kinds of program code, which makes its design very complex. Up to today, the majority of software which runs on it is still inherently sequential, and makes extensive use of branching. This means it does not employ data parallelism and contains highly non-linear and unpredictable execution paths.

To be used to full capacity, multicore CPUs require explicit parallel design during initial development, or redesign of existing applications. A graphics application which fully utilizes a single core has the potential to reach a multiple of its rendering speed by splitting the computation into more than one thread of execution. Such an application is said to *scale* well, or possess high *scalability*, if it can make use of all available cores on a system to increase performance proportionally.

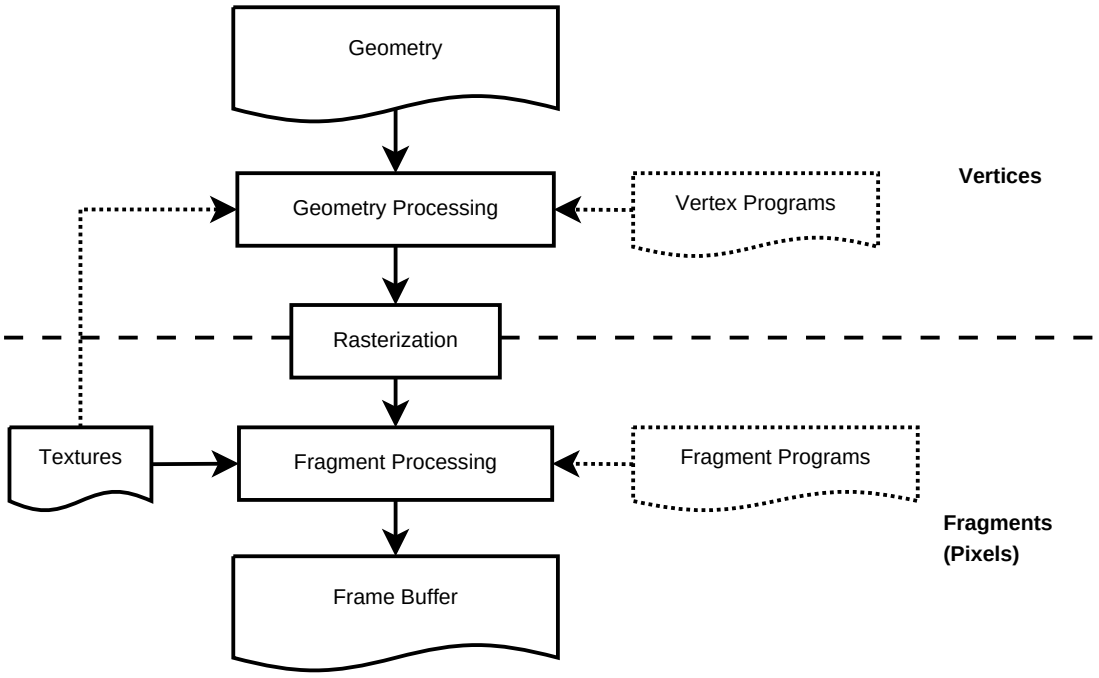

Figure 2.1: Simplified overview of the GPU pipeline. Dotted lines indicate features introduced by the programmable pipeline.

What is still seen as potential today, may become a requirement soon, as the roadmaps of CPU manufacturers predict [Int, HKO07]. The amount of cores per CPU is expected to increase, which will lead to a transition from multicore to manycore. Using only one of those many cores is simply a waste of resources. Worse yet, a single-threaded application might actually run slower on future CPUs, as manufacturers might reduce the clock speed of the cores to improve efficiency and reduce costs. In contrast, multithreading can improve performance even on single core computers.

2.1.2 GPU

Figure 2.1 shows the general structure of the pipeline contained in a typical GPU. Its operation can be divided into *vertex*- and *fragment*-processing.

A vertex is the basic primitive of geometry in computer graphics. It represents a point in

space and forms primitives such as lines or polygons when connected to other vertices[1]. A collection of vertices and primitives is called a *mesh*.

The term *fragment* is used to describe the single elements of images generated on the graphics card. The more general term *pixel* (short for *picture element*) can also be used, but while pixels only describe color, a fragment may have further attributes such as depth.

Geometry processing includes the transformation of vertices according to the viewer's position and the direction she is facing, which when combined are referred to as camera space (see [AMH02], section 2.3.1). It is followed by *culling* and *clipping* of the triangles formed by the vertices, and their projection into two dimensional screen coordinates for *rasterization*.

Culling is the process of rejecting parts of the geometry which are not visible to the viewer. Clipping checks visible triangles for intersections with the screen boundaries. The parts outside these boundaries are cut off before the triangles are sent further down the pipeline. Rasterization transforms vertex data into fragments, which are stored in a fragment buffer. The frame buffer is the main fragment buffer which will be presented to the user at the end of the rendering period, which is generally referred to as a *frame*. The faster a frame can be rendered, the more often the image of the scene is updated per second. This is important for real-time graphics to give an impression of fluid animation and to reduce the time - or *latency* - from receiving user input until the results are displayed on screen. The terms *frame rate* or *frames per second* (FPS) are often used to describe the performance of a graphics application.

Compared to a CPU, the GPU is a very specialized processor [Owe05]. It is not designed to run general purpose programs, instead it offers a *stream programming model* [KRD+03], which is shown in figure 2.2. It consists of streams of data (shown as arrows) and kernels which process the data (shown as boxes). A stream is defined as an ordered set of data of the same type, e.g. vertex data or fragment data. Kernels perform relatively simple operations on the data elements of the stream, but carry them out very fast. They are also usually massively parallel, which means that they can process

[1]The preferred primitive type is the triangle, which is the simplest form of a polygon. Thus, most GPUs only handle triangles and their hardware drivers have to subdivide polygons into triangles on the fly before they enter the geometry stage. Therefore, it is recommended for graphics applications to exclusively work with triangles.

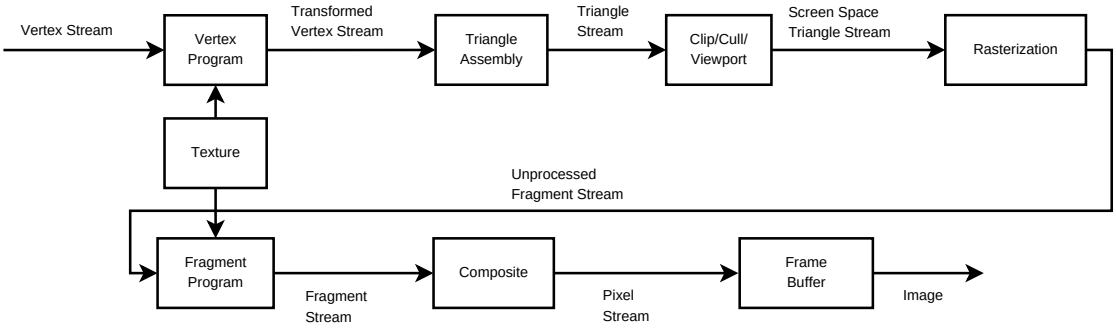

Figure 2.2: The GPU pipeline represented as a stream model [Owe05]

many elements of a stream simultaneously. They cannot, however, just operate on single elements, but only on entire streams. Thus, stream processing provides very high throughput for large data sets, but also has very high latency, which is the time taken by a single stream element from the beginning of the GPU pipeline to the end.

The first hardware iterations had hardwired geometry and fragment processors, which were later replaced by programmable units. These two different architectures are called the *fixed pipeline* and the *programmable pipeline*, respectively. A fixed pipeline has a configurable state, which means that parts of it can be (de-)activated or slightly altered. The possibilities are very limited, though, and changing the pipeline's state may cause it to be flushed, which means that no further rendering is done until the state change is applied. The programmable pipeline, while inheriting the structure of the fixed pipeline, introduced programmable *shader units* (or shaders), to give more flexibility to certain rendering stages. These units are able to execute *shader programs*, which are short, special purpose programs for either vertex-, fragment-, or unified shading [Ros06]. Shading is a general term for the modifications to graphics primitives, which usually includes computing color and position, as well as user defined properties (see [AMH02], sections 2.3.2 and 4.3).

Shader programs are still quite restricted compared to general purpose programs running on a CPU. The number of instructions per program is limited, and control structures such as branches and loops are expensive. Memory access is also restricted, and using data structures like lists which require pointers are not supported directly. However, due to these restrictions, shader programs lend themselves very well to parallel execution. As each program works on separate vertices or fragments, there are no synchronization

Figure 2.3: Comparison of different texture mapping methods to improve the apparent detail of a flat polygon [MM05]

issues and a very high scalability can be achieved.

Additional input to the GPU pipeline are *texture maps* (or textures), which are applied to polygons to give the impression of more detailed geometry (figure 2.3). There are different kinds of texture maps which contain colors, normal vectors, or height values. Combined with shader programs, the visual appearance of almost any material can be simulated in a graphics application.

The combination of texture maps, shader programs, and shading parameters is called a *material*. This term is often used in digital content creation tools, shading frameworks such as RenderMonkey, as well as in the specification of COLLADAFX [Col].

2.1.3 The Graphics Rendering Pipeline

Möller and Haines describe the graphics rendering pipeline in [AMH02], chapter 2. It is divided into 3 stages:

1. The application stage
2. The geometry stage
3. The rasterizer stage

Each stage can be a pipeline in itself. In the majority of cases, the application stage is executed on the CPU, and contains coarse culling and sorting, as well as other object level updates. The geometry and rasterizer stages are usually implemented on the GPU, as illustrated in figure 2.1. The rasterizer stage includes rasterization and fragment processing as substages.

On the transition from the application to the geometry stage, the CPU and the GPU need to communicate. This is made possible by an interface called the graphics application programming interface (API), which is the topic of the next section.

2.2 Graphics API

There are several approaches of interfacing CPU and GPU, which differ in their level of abstraction from the hardware and the functionality they offer. They also focus on different types of applications (also called the *clients* of the API).

The suffixes *library*, *API*, and *engine* are often used when describing a rendering system. They are added to label it as a piece of reusable software and hint at its purpose, scope, and complexity. As these are very general terms, they are not to be taken literally, and often they are used to describe the same thing. A software developer most probably has a good grasp of the terms *library* (a reusable implementation of a certain functionality) and *API* (an interface to a library). A generally accepted definition of the term *engine* is a functionally exhaustive framework, which attempts to take away most of the implementation complexity from an application and put the functionality into a library to be reused. Generally, a library is usually much smaller in scope as well as in functionality compared to an engine. Engines are very large software packages which can be composed of multiple libraries which depend on each other. A game engine most probably has libraries for graphics, audio, input devices, management of digital content files (*assets*) and more.

Porcino [Por] distinguishes between four approaches to real-time rendering:

- State management libraries
- Scene management libraries

- Shader systems
- Submission engines

The following sections explain the differences of these systems and which application types they support best.

2.2.1 State Management Libraries

In real-time rendering, state management libraries (or APIs) are usually the basis on which a rendering system is built, since they provide the lowest interface level to the hardware driver, which in turn is the interface to the hardware itself. The state management API and the hardware driver are closely coupled, and the implementation of the API is usually provided by the driver vendor. The API can be used directly by a graphics application, but this is usually only done for prototyping scenarios. Properly engineered applications benefit from additional levels of abstraction from the rendering hardware, which are offered by the systems described in the next 3 sections. They provide platform independence, robustness and usually an API which is easier to use.

There are two common APIs in the consumer PC graphics market which can be used as interfaces to the GPU: Direct3D and OpenGL. Direct3D was designed to be used in games and is a component of the game programming API DirectX. It is not officially supported on any other platform than Microsoft Windows and Microsoft's game consoles Xbox and Xbox 360. OpenGL, however, represents a mature industry standard and is available on all common PC operating systems, as well as embedded devices in the form of OpenGL ES. There are ways to emulate the Direct3D API on non-Microsoft platforms by layering it on top of OpenGL, but they are not yet ready for use in mainstream applications. Thus, Direct3D is only cited here for reference, and the focus of this thesis and the accompanying implementation lies solely on OpenGL.

Both of these interfaces represent the underlying hardware as a state machine. A set of states, called the *render states*, determine how the GPU will transform and rasterize geometry. Render states are addressed using identifiers which are defined by the API. They contain a value which can be of various data types, such as:

- Flags: either true or false, toggling certain functionality of the rendering pipeline, such as lighting or texture mapping
- Enumerations: similar to flags, but with more alternative values, such as culling front-facing primitives, or back-facing ones, or none at all
- Numerical values: usually to be specified in a given range, like translucency, which ranges from 0 to 1

Usually multiple render states control different aspects of the same GPU functionality. Translucent rendering (also called *alpha blending*, see [AMH02], section 4.5) is configured by a flag to enable or disable it, enumeration states which control the type of blending between two pixels, and the numerical alpha value, which determines the amount of translucency.

An application usually associates a mesh with a set of states to be set before it is rendered, which define its material and other rendering attributes. However, changing states is prone to being expensive, although redundant changes to the same state may be caught and ignored by the API. There are states which are cheaper to modify than others, however, this depends both on the hardware and the optimization strategies employed inside the API. Usually, states which are represented by numerical values, such as the colors used for shading, inflict less of a performance penalty than switching a flag- or enumeration state [For]. Such states toggle the operation of fundamental parts of the hardware. Changing them means bringing the graphics processor to a full stop, flushing its pipelines and reconfiguring the internal data flow.

The reason for this difference in cost of state change is that numerical values are more like factors in an equation, while flags are similar to branches in a program. Altering a factor in an equation does not change the way the result is computed, unlike following a different branch in a program. However, setting a numerical state to zero can have the same effect as clearing a flag.

This leads to the conclusion that avoiding costly state changes is a way to improve application performance. In newer iterations of the mentioned APIs, render states can be grouped to *state blocks* to reduce the number of function calls to the API by setting multiple states at once. This reduces function call overhead and allows the API to internally optimize the blocks for the layout of the GPU pipeline.

After setting the appropriate render states, the application issues a *draw call*, which is a call to an API function which takes a list of geometric primitives to render. The API sends them to the GPU, which then transforms, shades and rasterizes them according to the render states. Similar to state changes, it is advisable to keep the amount of draw calls to a minimum to avoid overhead caused by communication between CPU and GPU. However, the separation of CPU and GPU can be an advantage here, as they are able to run concurrently most of the time. This means that the state management library may return from the draw call as soon as possible, so the CPU can continue to run asynchronously. To avoid synchronization to the GPU, the application should not query render states between draw calls.

To reduce the number of draw calls, clients should attempt to send as much geometry for each call as possible. A technique called *instancing* also helps keeping the overhead of draw calls low by passing transformation data for meshes as parameters to the shading units. Instancing can also be performed completely on the GPU (hardware instancing, see section 4.2.4). A separate list of vertices needs to be specified for this, in which each vertex contains the position of a mesh instance. The draw call then instructs the GPU to draw one instance per vertex in this auxiliary list. Hardware instancing is a relatively new technique, though, and is not widely supported yet.

2.2.2 Scene Management Libraries

Scene management libraries such as OpenSceneGraph [OSG] or Ogre 3D [Ogrb] provide another layer of abstraction on top of the state management API. They implement a tree structure where objects can be stored and grouped to express hierarchical relationships. This structure is called a *scene graph*, and is one of the most popular data structures in computer graphics ([AMH02], section 9.1.4).

Besides geometrical objects, scene graphs can store transforms, render states, levels-of-detail, bounding volumes, light sources, audio sources, and other data which is relevant to a scene. For example, a racing game would use a scene management library to manage the models of all cars participating in a race. Each car would be represented by a node just below the root of the tree structure. A car's geometry, shaders, engine sounds, and particle effects, as well as the model of the car's driver would be attached as children of

the car's node. The hierarchy becomes deeper as the node of the driver model in turn contains child nodes for shaders and geometry.

To render the scene, the scene graph is traversed in depth-first order, while handling each visited node depending on its run-time type. Geometry nodes are tested for visibility and eventually rendered, and render states and light sources are applied using the state management API before rendering the geometry contained in their child nodes.

Scene graphs are a very flexible concept which can be used to model the contents of a scene at a very abstract level. However, it is easy to abuse the tree structure to store application data which completely unrelated to rendering. Scene management libraries often encourage this by design, as they contain large hierarchies of node classes, advocating the mental theme that everything should inherit the node interface, so it can be managed by the library. The graph slowly converges into a catch-all data structure, with no clear separation of model and view representations [GHJV94]. Thus, encapsulation and modularity of the system suffer and the maintenance of an application becomes harder. To prevent this from happening, an application developer must take care that a separate model data structure is created for the application to work with, while the scene graph is built and updated to reflect it in a one-way relationship. Ideally, the application would not even know that there is a scene graph at all, it should only be aware of a view component it has to inform of changes to the data model.

While scene graphs are a great tool for expressing hierarchical relationships, they are not optimal for storing large amounts of varied kinds of objects, which might have different relationships depending on context. It is impossible to design a graph which expresses relationship of spatial location, visibility, render state modifications and lighting equally well. This leads to duplicated information or dependencies which are not modelled by the scene graph, making it a lot more complex to manage and traverse than initially intended. This can result in decreased performance and maintainability. Forsyth [For] advocates the use of specialized, explicitly synchronized trees to express different concepts and relationships.

2.2.3 Shader systems

Shader based systems are common in the domain of offline, non-real-time rendering. Their strengths lie in sophisticated lighting systems and superior levels of visual quality for non-interactive scenes. Their usage of shaders must not be confused with the shader programs executed in the real-time shading units of graphics hardware. They provide more control and are less restricted in terms of shading instructions, while trading speed for quality of the rendering output.

Applications which use shader systems include modelling tools and renderers for computer generated images or movies which require photo-realistic results. Thus, shader systems are only mentioned here for reference, and have no relevance for the purpose of this thesis.

2.2.4 Submission Engines

Porcino [Por] describes submission engines as the best way to achieve high performance rendering on current graphics hardware. Unfortunately, at the time of writing no publicly available submission engines exist, and there is no circumstantial literature available on the subject. Information on submission-based rendering was collected from internet resources such as blog entries and mailing lists.

A submission-based renderer is a wrapper around the state management API. Objects are rendered by passing (*submitting*) them to the engine each frame. The application is responsible for the way it organizes these objects, and may choose a structure as complex as a scene graph or as simple as a list. Additionally, it specifies *composite operations*, which trigger modifications of the frame buffer as a whole during the rendering process. Such modifications include activating, clearing, or compositing of frame buffers. As an example, an application could separate rendering into two layers: the first contains a box with a sky texture as the background, and the second represents the 3D environment surrounding the camera as the foreground. The depth buffer, which is associated with the frame buffer and stores the depth value of each fragment, needs to be cleared after the background was rendered. Otherwise, back- and foreground layers might conflict as they are rendered at different distances from the viewer, resulting in different ranges

of depth values. To achieve this, the application submits a composite operation which clears the depth buffer in between the two layers.

Internally, the renderer will sort submitted objects to minimize state changes, especially expensive ones. Certain state changes will cause the GPU pipeline to be flushed, which has to be avoided as often as possible. Furthermore, to keep the pipeline filled, the GPU has to be continuously fed with draw calls. This ensures the highest possible utilization of the graphics hardware.

As mentioned in section 2.2.1, the cost of state changes is hardware-dependent. However, the client can give hints in form of sorting preferences to the renderer. If the client specifies a composite operation after submitting a sequence of objects, the renderer can only sort objects within the boundaries between this composite operation and the previous one.

Besides these constraints, the submission engine has full control over the render states and the order of draw calls and can give an application a great deal of improved performance over managing state by itself. It also gives the client the most freedom in choosing its scene representation.

2.3 Existing Crowd Rendering Engines

The following selection of existing engines with support for crowd rendering is not meant to be exhaustive. There are many proprietary implementations in use by video game developers which the general public has no access to. Besides, every real-time rendering engine can be used to draw crowds, but many of them are not optimized to do so and will usually not scale very well with large numbers of objects.

2.3.1 Ogre 3D Extensions

The *Ogre 3D* [Ogrb] rendering engine is a very popular open source engine. It covers a wide range of functionality and is by no means a dedicated crowd renderer. During

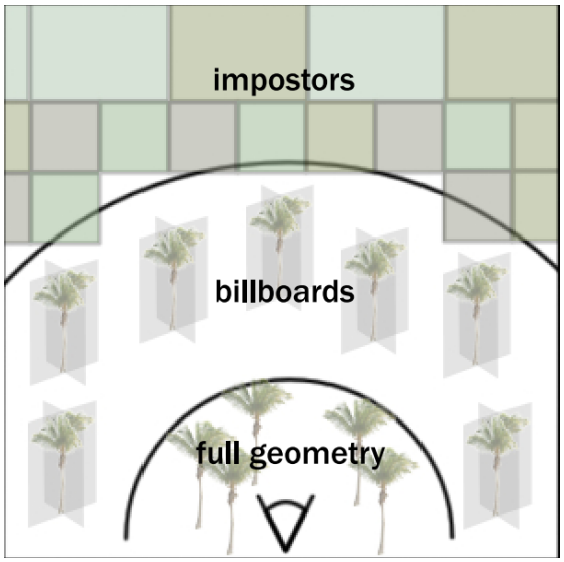

Figure 2.4: Rendering different levels-of-detail using billboards and impostors

Google's Summer of Code 2006, an extension was developed which implements so called *shader-based instancing* (see section 4.2.4). It was integrated into the Ogre distribution as of version 1.4. Judging from the technical details discussed by the developers [Ogra], the design of the extension is quite involved. It supports calculation of character animations on the GPU, including independent animations for each character. Shadows are also mentioned, which gives the impression that the extension is not really encapsulated, but has some dependency on the shader system used in Ogre. Instancing may thus not be easily integrated into existing applications which feature complex shaders.

Guerrero [Gue06] approaches the problem of rendering forests by using impostors and billboards and implements them using the Ogre framework. The impostor technique works by rendering a mesh into a texture, which is mapped on a rectangle and drawn instead of the actual mesh. If the orientation of the mesh towards the viewer changes more than a given threshold, the impostor is regenerated. This reduces the load of the GPU's geometry stage, but at the same time often reduces visual quality. Guerrero's billboarding technique works in a similar way, but use different views of the same object and puts them on rectangles which intersect in their vertical center. Blending these rectangles over each other gives more depth to the impostor. Figure 2.4 shows how the approach selects different rendering techniques at varying distances. This solution can be made to scale for larger scenes by trading in worse quality of rendering, but it does

not make use of parallel hardware.

2.3.2 Horde3D

Horde3D [Hor] is an open source rendering system developed at the University of Augsburg. It is titled a "next-generation graphics engine". Figure 2.5 shows some visually impressive screenshots of the engine in action, using crowd rendering to display particle effects and to populate a scene with animated characters.

The engine is purely shader based and does not contain support for the legacy fixed pipeline. It uses a technique called deferred lighting, which performs lighting as a separate render pass to simplify shader development and to reduce the number of draw calls per light source. To speed up rendering, dynamic level-of-detail and vertex data optimization are employed.

A scene graph API is used to build applications. This may impose unnecessary restrictions upon the user, who has to adopt this kind of scene representation. While it has advantages in rapid prototyping scenarios, it might require some effort to wrap the data structures to combine the engine with other middleware, or when trying to use it with an existing code base which has its own methods of scene management. Section 2.2.2 contains a short discussion of scene graphs.

However, Horde3D's implementation is very lightweight and seems to offer good performance. It is not able to make use of multiple processing cores or GPUs, though.

Figure 2.5: Horde3D screenshots

3 Concept of a Parallel Crowd Renderer

This chapter contains the design of Pace, a lightweight, flexible, parallel rendering architecture targeting crowd rendering applications. From the discussion of several rendering interfaces in chapter 2, the submission-based concept was chosen as the initial design, because of its flexibility and its focus on performance.

The opening section of this chapter analyzes which demands a hypothetical application might make on a renderer. The results are used to motivate the requirements and goals of Pace, which are followed by a section discussing its design. It specifies and motivates the most important classes and data structures.

3.1 Case Study

Video games are a common field of application for real-time rendering. Consider a car racing game as an example, which faces very high demand for visual realism from the player. There are highly detailed car models to be rendered, alongside rich scenery which surrounds the racing tracks. These are supplemented by graphics effects such as reflections on cars, shadows cast from cars onto the scenery and vice versa, and particle effects for car exhaust smoke and dust stirred up from the road.

While all of this is being rendered, the player is moving through the track at high speed, which means that the virtual camera is always rapidly changing position. This makes it difficult to adjust the level of detail without the player noticing it. It also means that a very high frame rate is required to ensure that the player feels immediate response to the car's controls, and that he can react quickly to upcoming obstacles or road bends, which requires minimum input latency.

Besides rendering, the game has various tasks to carry out during frame time to update the game's state according to user input, simulated physics, opponent's artificial intelligence (AI) and collision response.

To summarize, the tasks of a renderer in this kind of application are as follows:

- Manage a scene database containing models of cars, tracks and environment objects
- Render highly detailed, dynamic, close-up meshes of cars
- Render large amounts of low to medium detail, static roadside objects such as signs, buildings, and trees
- Render special effects, like shadows and reflections
- Distribute CPU load between rendering and updating game state

These requirements are generalized in the following sections.

3.2 Requirements and Goals

The main design requirements were already mentioned in the introduction in chapter 1. This section will recapitulate and refine these requirements.

3.2.1 Performance

The performance of a real-time graphics renderer is not easy to measure objectively, as it depends very much on the content and purpose of the application. There are two values, however, which provide an estimate of a renderer's performance: throughput and scalability.

The throughput (or capacity) of a pipelined system is the amount of data it can process in a given finite time unit. The scalability of such a system is determined by the increase in throughput it can achieve when more processing cores are added to stages of the graphics pipeline. Cores can be added in form of CPUs or GPUs. Although the maximum number of additional processors and graphics cards on current PCs is limited, these limitations will be lifted more and more in the future. Throughput is always limited by the pipeline's slowest stage, the *bottleneck*. Thus, it is important that the bottleneck benefits most from added processing power, because it defines the scalability of the whole system. Unfortunately, the location of the bottleneck is application-dependent.

Good performance is obviously a goal of a real-time renderer, but it is impossible to specify it exactly as a requirement. In terms of scalability, however, it is expected from a parallel renderer to improve performance when adding more cores. This still depends on the application's content, because if the added hardware is not working on the bottleneck of the rendering pipeline, there is nothing the renderer can do to improve performance. In this case, the application must balance the pipeline to decrease the load of the bottleneck.

3.2.2 Portability

Pace is implemented in C++ and is required to compile and run on the PC operating systems Windows, Linux, and MacOS (Intel). An implementation of the OpenGL API must be available on the target system to be able to run it.

Other software libraries which Pace depends on are listed in section 4.1. They are written in C or C++ and are tested to work on all platforms mentioned above. In part, they enable Pace to run on multiple platforms as they provide abstractions from hardware and operating-system-specific APIs.

3.2.3 Functionality

The design presented in this chapter will have to account for a broad range of popular techniques which exist in real-time rendering today.

The renderer must be able to manage a scene database composed of triangular geometry, which is the standard form of geometrical primitives that all modern rendering hardware is able to process. To build a scene, it must be possible to instantiate the models stored in the scene database with a minimum memory footprint, so that high numbers of instances of the same model are possible. These instances must be able to be moved independently and use different materials than the original model. It should also be possible to give the instances different sequences of animation, although this may require separate models for each sequence.

Special effects such as shadows and reflections, as well as post-processing effects must be possible, which require multiple frame buffers and composite operations.

3.3 Pace

Pace was developed to create a prototyping environment for parallel rendering. As graphics applications use different scene representations with varying degrees of size

and complexity, it was designed to be easily adaptable. Unlike a large, monolithic graphics library, which tries to be as feature-complete as possible, Pace was planned to be extended to accommodate to the application's needs, not the other way around. Thus, it was not of high importance to provide an API which hides the complexity of the rendering process from the application. It was of high importance, though, that all major parts of the library are loosely coupled, so that the application can pick certain functionality it needs, but discard the parts it does not. Being a research project, its goal is to be a rendering toolkit, not a fully functional engine. Nevertheless, throughout this chapter, Pace will often be referred to as a library instead of a toolkit, simply because it comes as a library in the software development sense of the term.

3.3.1 Overview

An application using Pace, like any real-time application, can be separated into two phases: initialization and rendering loop. Figure 3.1 shows these phases and the sub-phases they contain.

In the initialization phase, the scene to be rendered is put together. This phase occurs each time the application fills a scene with content, and before rendering the first frame. In the racing game example, this phase is entered once the player has selected a racing track, and it includes loading the models of all cars, the track, and the environment, as well as audio clips and other necessary data.

The rendering loop is the real-time phase of the application. Each iteration of the loop displays a complete frame and computes the state of the scene at the current time. Time is measured using a real-time clock provided by the system. This means that the longer a frame takes to render, the further the game state will progress during one iteration of the loop. The application has to take care of parametrizing the movement of objects with the amount of time passed since the last iteration.

The reason for separating the two main phases is the performance-critical nature of the rendering loop. Since one iteration of the loop has to complete in a time span of 20 milliseconds or less, many applications try to do as much processing as possible before entering the loop. However, initialization and rendering loop might overlap in certain

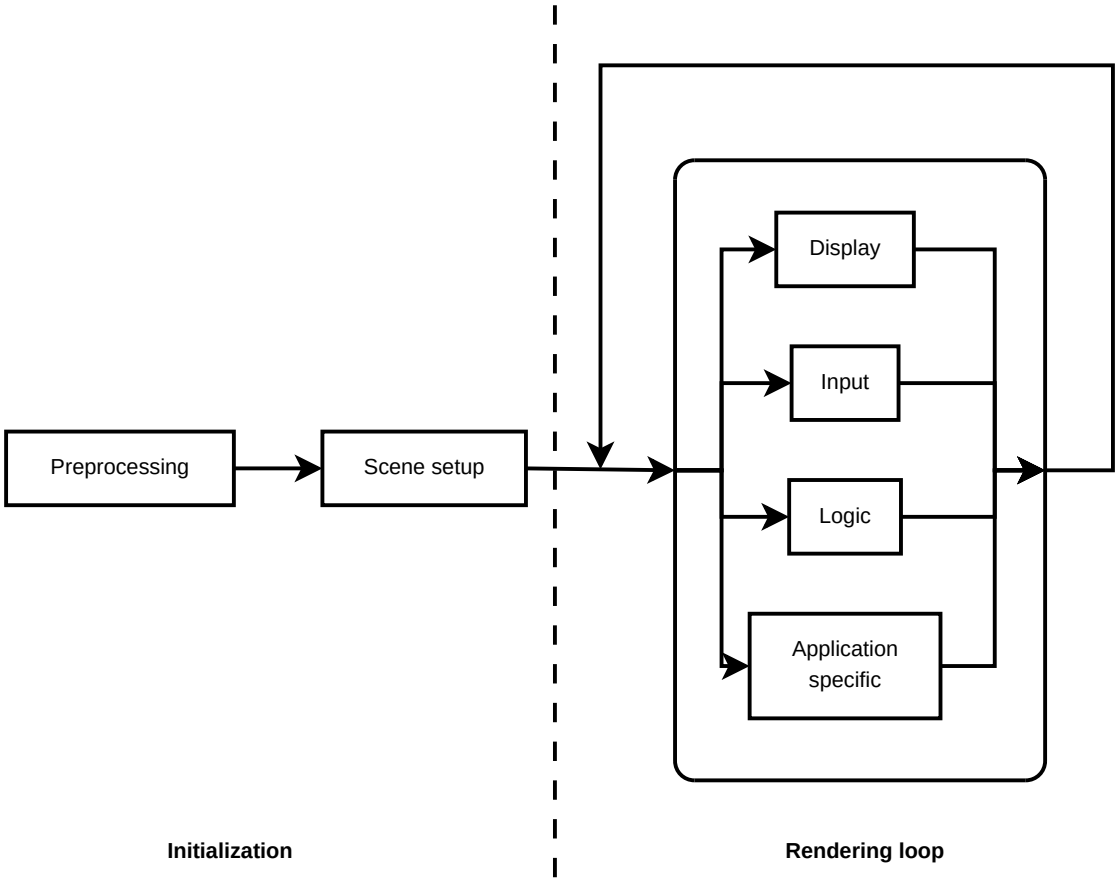

Figure 3.1: The work phases of a rendering application

cases. If the game is about to load a racing track which does not fit into available memory, it may choose to dynamically load content into the scene simultaneously to the rendering loop, a process called *streaming*. There are many games which deliberately choose to stream all data to be able to create large, seamless environments. This avoids the interruptions which players have to wait through in conventional games, which contain multiple levels with separate loading and gameplay phases. Although streaming is possible using Pace, the examples in this chapter will focus on the simpler scenario of separate initialization.

The following two sections describe the two phases in more detail and show how applications implement them using Pace.

3.3.2 Initialization

Crowd rendering is a challenge to the graphics hardware, because it requires large, partially dynamic data sets to be processed by the GPU which is connected to the main processor through a bus with relatively high latency. When designing a 3D application, this leads to the motivation of preparing all static data at loading time. Static data is not modified after initialization, while dynamic data is the remaining set of values which change frequently. Dynamic data is submitted to the renderer during the real-time rendering loop.

This separation is achieved by partitioning source data into a *scene database*, which contains static data, and a dynamic data structure which references the database and is traversed each frame, referred to as the *scene* (figure 3.2). The scene database is built from the content the application wants to display, which are meshes, materials and other assets.

In this context, an *instance* is a lightweight object which contains a reference to a mesh in the database, and represents something that is going to be rendered. The additional data an instance contains is dynamic and may change each frame, such as location and orientation in the scene, relative to the origin. Instances can also override the material a mesh uses. This is depicted in figure 3.2, where a material is referenced both by a mesh stored in the database and an instance referring to another mesh. When this instance is rendered, the geometry of the mesh will be drawn using the material referenced by the instance instead of the one referenced by the mesh.

The initialization phase contains two subphases:

1. Preprocessing: source data (geometry, textures, shader programs) is generated or loaded from disk to build the scene database. Appropriate data is uploaded to the graphics hardware's memory, while optionally being compressed to reduce memory and bandwidth consumption. If source assets are not conforming to the graphics hardware's criteria, they have to be converted. For applications requiring a database which is too large to fit into available memory, the engine can delay loading parts of the data until they are referenced by an instance.
2. Scene setup: the application builds the scene it is about to draw by placing in-

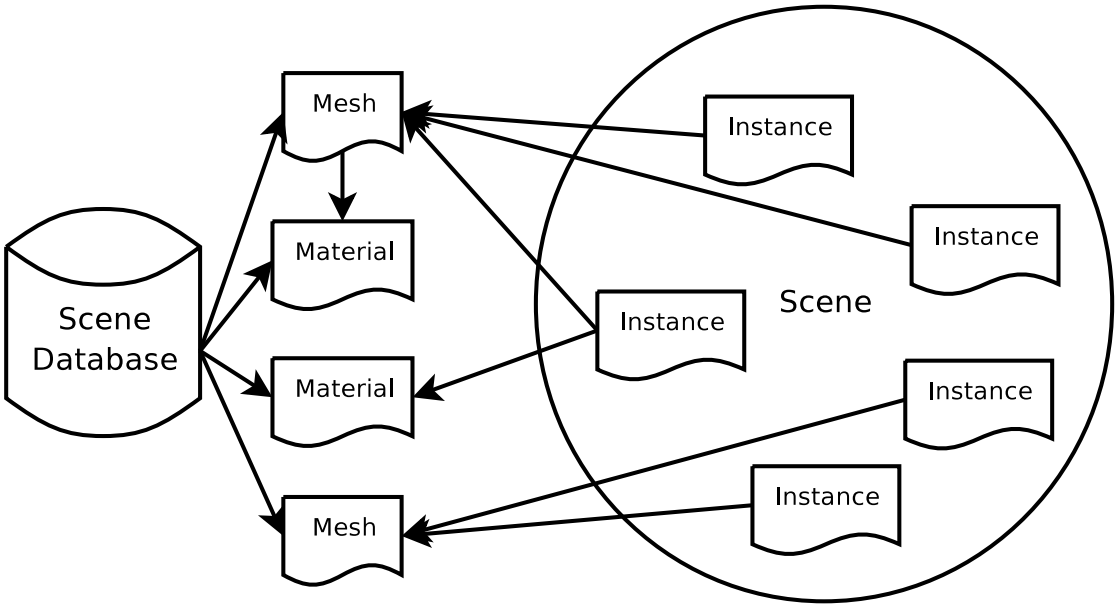

Figure 3.2: The scene database and the concept of instances. Arrows indicate references between objects.

stances in the virtual world. Placement of light sources is also part of this phase. This phase is often hardcoded in smaller applications, but usually a level editor or a similar tool is used to create a scene description which is read by the application.

To illustrate the preprocessing phase, the following example describes a simple, but applicable workflow for transporting content from a modelling tool into an application. Note that actual professional content workflows are more complex than this, and involve additional tools between the modelling tool and the target application.

After the 3D model of an object is finished, its data must be stored in a file format which is understood by the application. This process is called *export* and is separated from the save function in the 3D modelling tool, which most often results in a proprietary, undocumented file format.

During preprocessing, the exported file is parsed by the application, which is a process called *import*. The result is a data structure which consists of various buffers for geometry and textures, which are referenced by meshes and materials. Shader programs which are used to implement materials are also parsed and compiled in the process.

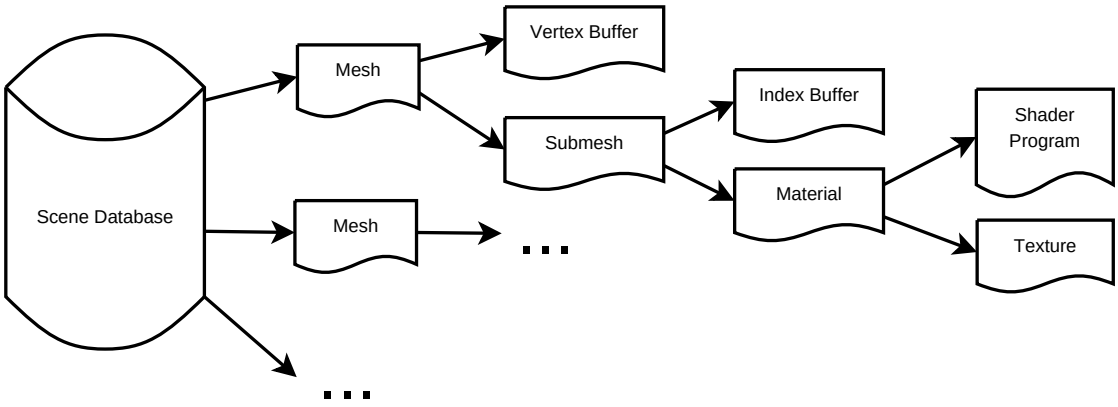

Figure 3.3: An example of a scene database

The Pace library provides an importer for Wavefront OBJ files [Bur], which can be exported by most modelling tools available. The importer class `obj_importer` parses an input file and returns the extracted data in a `mesh` object. An OBJ file can contain material definitions, which are also parsed, and the resulting materials are added to the mesh. The application can freely choose where to store the `mesh` object, and will probably use a standard container to do so. In fact, it could be as simple as a `std::vector<pace::mesh>`, which creates an array of mesh instances. This container is the scene database, while the mesh is one of its entries which will later be referenced by the scene to be rendered. Figure 3.3 attempts to convey the links of this data structure visually.

The way the scene elements are stored during scene setup depends on the application. Simulations usually operate on a set of entities and their state, which form the simulated world. Each entity has a visual representation which is parametrized to convey the entity's state. An instance of it is placed in the scene according to the entity's position. This is a model-view-controller system, where the set of entities are the model, the scene and the renderer displaying it are the view, and the simulation combined with user input represents the controller. Smaller applications often simplify this approach by combining model and view into a single data structure, usually a scene graph. By decoupling model and view from each other, maintainability and extensibility are improved, which usually makes up for the additional effort required to create a clean separation of the data structures involved.

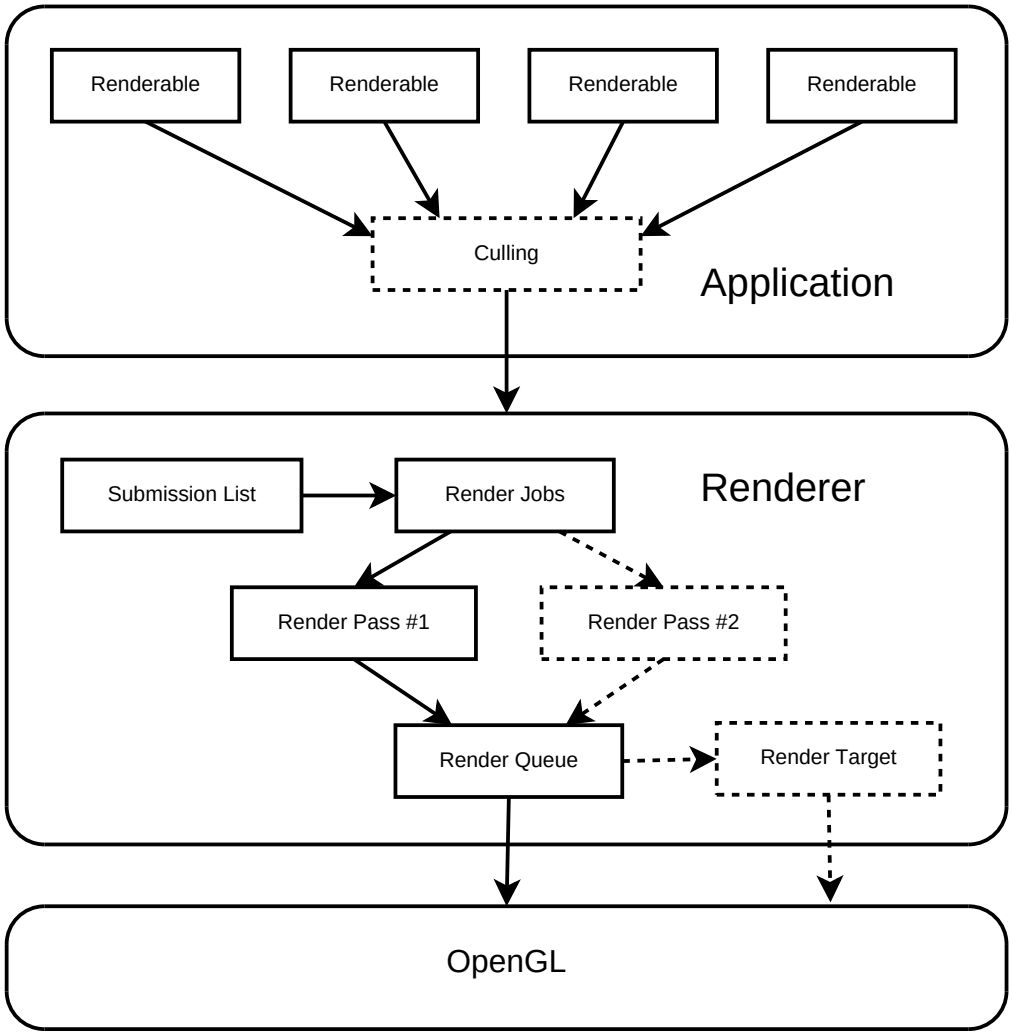

Figure 3.4: Example configuration of the Pace renderer. Optional components are indicated by dashed lines.

At some point during initialization, the application has to create a Pace `renderer` object. This is the main interface through which the contents of the scene are submitted. An advantage of the renderer architecture in Pace is that a renderer object can be configured to adapt to the scene the application wants to draw. This happens completely dynamically, so the renderer can be reconfigured or replaced at run-time to be used for a scene with different requirements, for example a special post-processing effect. See figure 3.4 for a simple example of such a renderer configuration. It contains the following components, which are explained in detail in the sections 3.3.6 ff.:

- Renderable: an abstract interface to a renderable object, for example a mesh instance. A scene is built from objects which implement this interface.
- Renderer: the origin of the rendering process, and the main interface where renderable objects are submitted.
- Submission List: part of the renderer which collects renderables and transforms them into render jobs. These are usually cached so the transformation can be done during initialization.
- Render Job: a passive data object which contains all information required to execute a draw call (see section 2.2.1). There can be multiple render jobs per renderable object. Render jobs are stored independently of the renderable objects in a read-only container inside the submission list, to be able to provide efficient, parallel access to them.
- Render Pass: parts of the renderer which represent its configuration. A render pass usually iterates over a list of render jobs and inserts them into a render queue. However, render passes can also implement composite operations (see section 2.2.4).
- Render Target: optional component which represents a frame buffer other than the default one. Using a render target, the output of a render pass can be directed into a texture, which can then be used by another render pass.
- Render Queue: as the final part of the rendering process, the render queue is responsible for collecting render jobs from one or more render passes, sorting and actually rendering them using the state management API.

The following section goes into more detail on how the renderer works.

3.3.3 Rendering Loop

During initialization, an application creates a scene which is now about to be rendered. The rendering loop is the real-time phase of an application. It contains the following subphases:

- Display: the contents of the scene are displayed.
- Input: user input is queried and processed.
- Logic: the application logic is executed.
- Application-specific: tasks which fit none of the other categories (see below)

The example renderer in figure 3.4 leaves the task of visibility culling to the application. During the display subphase, the exemplary racing game iterates over its scene representation and determines which cars, which parts of the track and which environment objects are visible. To do this, it could use different data structures for each of these categories, as they have unique characteristics in terms of mobility, size and geometric complexity. Choosing appropriate data structures will accelerate the culling process, especially with the large amounts of objects expected. Each visible scene element is submitted to the renderer as an object which implements the renderable interface.

As part of the input phase, the game queries the state of input devices and optionally returns collision feedback effects to them if they support it. In the logic phase, opponent AI and car physics are updated, operating on a model of the game world which is separate from the visual scene representation. At the end of the logic phase, the updates to the model need to be reflected in the location or visual appearance of objects in the scene. Application-specific tasks might include audio output and network communication, if multiple players compete in the game via remote connections.

Listing 3.1 gives an overview how the rendering loop of the game might look like[1]. For simplicity, this loop is completely sequential, and does not make use of the parallel architecture of Pace. It also omits the application-specific phase and the culling implementation. A parallel version is discussed in section 3.3.4.

[1]The `operating_system` namespace contains wrappers for functions which are platform-dependent.

Listing 3.1: Rendering loop example

```
bool done = false;
do {
  // Logic phase: get wall clock time and update game state
  pace::timer::time_type time = pace::get_time();
  update_game_state(time);

  // Display phase: submit all instances to the renderer and draw them
  std::for_each(scene.begin(), scene.end(), submit_if_visible(renderer));
  renderer->render();
  renderer->clear_submission_list();

  // Input phase: check if the player quit the game
  if (operating_system::poll_event() == quit_event) done = true;
} while (!done);
```

Inside the loop, the `std::for_each` function [SGI] iterates over a sequence of elements from the container `scene`, which stores the application's scene as a list of renderable objects. For each element, the `submit_if_visible` functor is called which submits the element to the given renderer after determining that it is visible. The renderer is subsequently instructed to draw the submitted objects and remove them after this is done, to be ready for the next frame.

Figure 3.4 shows two sections of the display phase which potentially lend themselves to parallelization: the culling operations in the application part, and the render passes processing the render jobs. Culling is a prime candidate for parallel treatment, as the visibility determination for the renderable objects is completely independent of each other. If the renderer uses multiple render passes, only those working independently can run in parallel. This is discussed in more detail in the following section.

3.3.4 Parallelization

The layout of the subphases of the rendering loop in figure 3.1 indicates that they might run in parallel. This section gives a few examples of how this can be achieved using Pace.

Due to the separation of CPU and GPU, there are three kinds of concurrency a graphics

application can benefit from[2]:

- CPU-level concurrency
- Multi-GPU concurrency
- CPU/GPU concurrency

CPU-level concurrency strives to exploit the processing power of multicore CPUs or even multiple CPUs containing one or more cores each by using multithreaded execution. When writing applications that use OpenGL, it is important to only call OpenGL functions from the thread which owns the corresponding OpenGL *context*. Such a context wraps the complete set of render states and platform-specific information. Accessing a context from other threads than the one which made it the current context is unsafe and deemed undefined behavior. A thread may give up ownership of the context so it can be made the current context of another thread (*context switch*). This of course has to happen in a synchronized way and will not leverage parallelism. A context switch will also cause the GPU pipelines to be flushed. An application can create multiple contexts, which may share resources such as textures and shader programs. However, this is only useful when rendering to separate render targets and it is unclear as to how much of a performance gain it has to offer for crowd rendering. It is possible to split the viewport up into smaller rectangular sections, which are rendered in parallel on multiple contexts, and combined to form the final image. If only a single GPU is available, the OpenGL driver has to schedule the thread's contexts to a single hardware resource, causing permanent context switching. Thus, it depends on the implementation of the driver and its interface to the hardware how well this technique performs.

A simpler, more lightweight approach is to create a dedicated OpenGL drawing thread, and several worker threads which execute the high level rendering code of Pace (figure 3.5). As already mentioned, the context of the OpenGL thread must not be accessed from the worker threads. To notify the drawing thread of new render jobs, these threads either perform explicit synchronization through an implementation of locks provided by the operating system (mutexes, semaphores), or use lock free containers where render jobs are stored and retrieved in a producer-consumer fashion. The latter option is usually faster, since locks cause a more significant overhead.

[2]Concurrency of the processors inside a single GPU is controlled mainly by the hardware and its driver and lies beyond the influence of a normal application.

Listing 3.2: Excerpt of the main function of a game using Pace with multiple threads

```
// Initialization code which creates the OpenGL context
...

// Create a multithreaded render queue
pace::render_queue * render_queue = new pace::multithreaded_render_queue();

// Create the renderer thread
operating_system::create_thread(&render_thread);

// Rendering loop of the OpenGL thread
bool done = false;
while (!done) {
  // Clear frame buffer
  glClear(clear_bits);

  // Draw render jobs
  render_queue->draw_jobs();

  // Present the frame buffer to the player
  operating_system::swap_buffers();

  // Synchronize with rendering thread using a semaphore
  operating_system::incremenent_semaphore(next_frame);

  // Check if the player quit the game
  if (operating_system::poll_event() == quit_event) done = true;
}
```

Listing 3.3: Rendering thread example

```
void render_thread() {
  // Loop forever
  for (;;) {
    // Get wall clock time and update game state
    pace::timer::time_type time = pace::get_time();
    update_game_state(time);

    // Submit all instances to the renderer and draw them
    std::for_each(scene.begin(), scene.end(), submit_if_visible(renderer));
    renderer->render();
    renderer->clear_submission_list();

    // Wait until the OpenGL thread is done with the current frame
    operating_system::wait_semaphore(next_frame);
  }
}
```

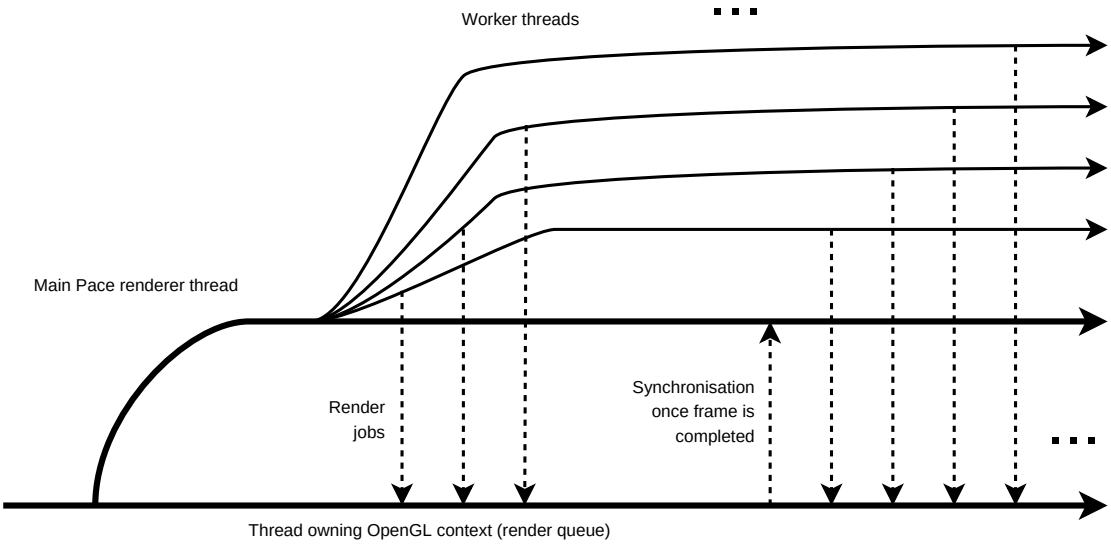

Figure 3.5: A threading model for CPU-level concurrency

To give an example, listings 3.2 and 3.3 illustrate how an application can benefit from CPU-level concurrency. The rendering thread in listing 3.3 bears obvious resemblance to the rendering loop in listing 3.1. For simplicity, unlike suggested in figure 3.5, there are no worker threads used in this example. The `std::for_each` could be replaced by a concurrent algorithm like one of those provided by the Threading Building Blocks library [TBB] (see section 4.1.1). This example only contains two threads, which form a producer-consumer queue (see section 4.3.2). Thus, scalability is very limited, and an application whose bottleneck is the CPU should make sure to parallelize the logic phase implemented in `update_game_state`. As the display phase depends on the results of the logic phase, they cannot run in parallel on the CPU-level. However, there are ways to exploit parallelism through CPU/GPU concurrency.

As already described in section 2.2.1, CPU and GPU are able to work concurrently, if there are no events which cause synchronization between them, such as context switches or state queries. The above scenario is not able to make good use of this potential, as figure 3.6 outlines. Most of the time, either the GPU or the CPU is idle waiting for the other to complete its tasks. To increase the length of the concurrent period, Pace will attempt to issue draw calls to OpenGL as soon into the frame as possible. While the application is submitting objects, it periodically flushes the render queue when the size

of the submission list reaches a certain threshold.

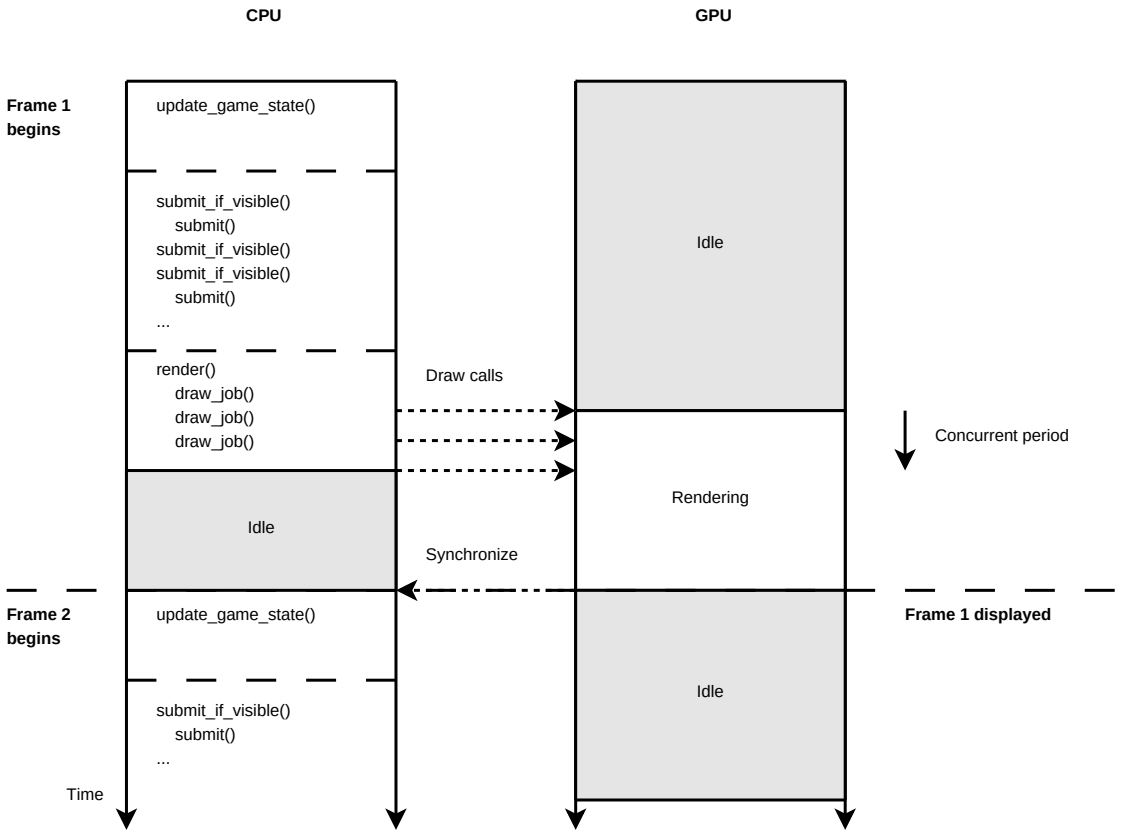

Figure 3.6: A rendering loop with minimal CPU/GPU concurrency (simplified sketch)

By restructuring the rendering loop, it is possible to leverage CPU/GPU concurrency at the expense of increased latency. Figure 3.7 shows the modified rendering loop, in which the logic phase of the application is executed while the GPU is busy drawing the *previous* frame. This means that the moment of time represented by the rendered image that the player sees, has already passed. Especially for games where reaction time is crucial, this could be a severe disadvantage. However, if the performance gain is big enough and the frame rate improves considerably, the difference in latency might be unnoticeable. The application should make sure to run the logic phase and other CPU-intensive code while the GPU is busy, which happens just after the submission of the scene to the renderer is completed.

Furthermore, events causing synchronization of CPU and GPU are an issue which has

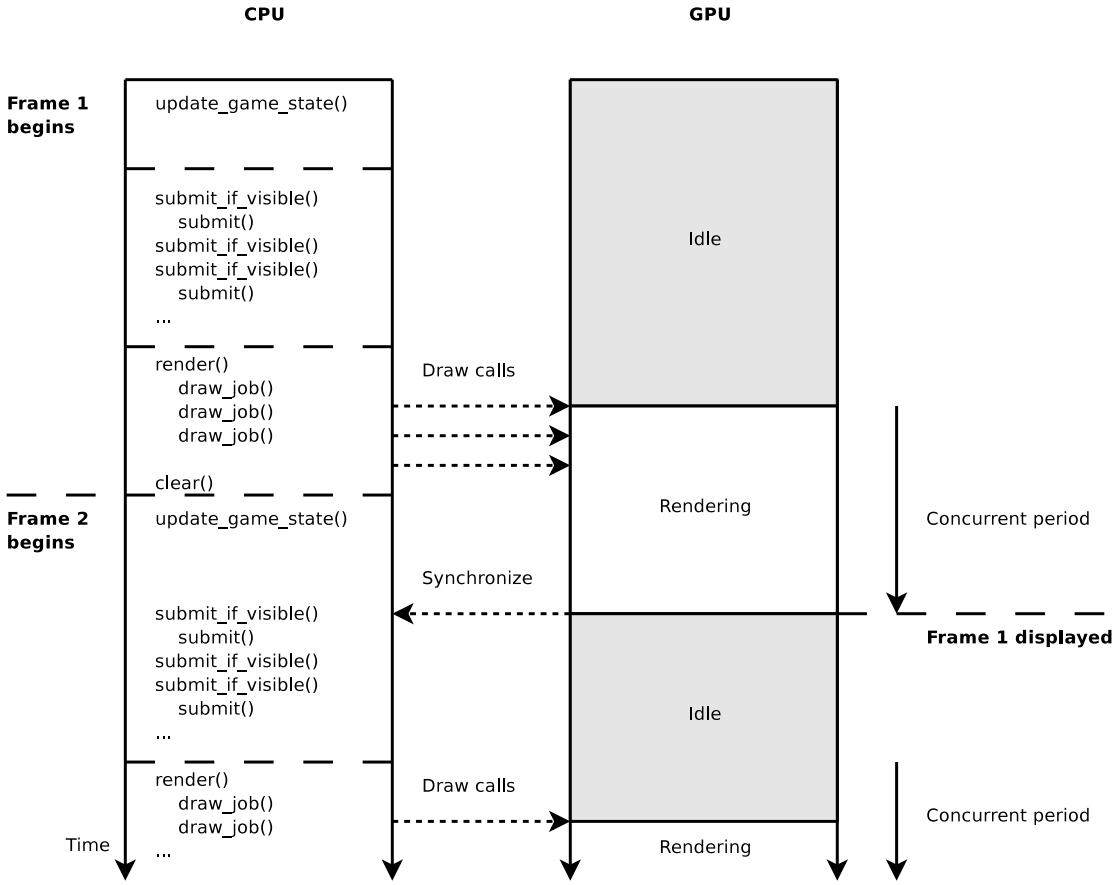

Figure 3.7: A rendering loop with good CPU/GPU concurrency, but increased latency (simplified sketch)

an influence on performance. To achieve high throughput, synchronization should only occur when necessary. This means that during the processing of a scene, only state changes and draw calls should be sent to the GPU, but state queries must be avoided. In particular, calls to OpenGL's `glGet*` family of functions during rendering will stall the GPU pipelines and cause synchronization.

On a system with more than one GPU available, one context per GPU could be used without the need to switch contexts, achieving concurrency between the different GPUs. By dedicating a CPU-level worker thread to each GPU, and using the synchronization model outlined above, this could improve scalability on a system whose bottleneck is the GPU. This would require one render queue per GPU, though.

3.3.5 Class Structure

As Pace is written in C++, its design follows object-oriented principles. All classes belong to one of the following categories:

- Interface: abstract classes which provide the high-level interface to the rendering pipeline
- Implementation: classes implementing actual functionality by inheriting one or more interfaces
- Data: classes which carry data, have no functionality, and are not meant to be subclassed

This separation cannot always be strict, there will be implementation classes which carry data, and there might be interfaces which provide functionality. It was introduced because it helps in the design of the class structure during the modeling process. It also gives the client a better grasp of which classes might be necessary to extend when adapting Pace to an application.

The following sections will describe the interfaces and data structures of the Pace library. Chapter 4 focuses on the implementation of Pace and contains more detail on some of the classes described here.

3.3.6 Interfaces

The interfaces listed in this section provide the user of the library, referred to as the client, with access to the rendering functionality. Figure 3.4 shows an example of how some if these interfaces collaborate from a high level point of view. At the highest level, the application communicates with the renderer, which operates on a list of render passes. These are pushing render jobs into a queue, which sorts them and in the end calls OpenGL functions to render them.

For rapid prototyping needs, there are default implementations for all abstract interfaces. For example, the `render_pass` interface (detailed in section 3.3.6) is accompanied by the

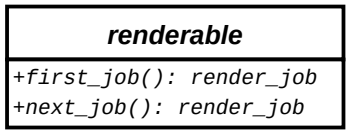

Figure 3.8: The Pace `renderable` interface class

`default_render_pass` implementation. It performs drawing of all geometry by using the render queue interface. Special rendering techniques like the ones suggested in the car racing game example require more sophisticated render pass implementations, but for simple rendering, this default implementation is fully sufficient and eliminates the need to bother with the lower-level interfaces involved.

Renderable

Figure 3.8 shows the `renderable` class. It represents the concept of an object which can be rendered. Being rendered is equivalent to being transformed into a list of *render jobs*, which are subsequently executed. A render job is a data structure which contains links to all the data required to render a piece of geometry (see section 3.3.7). It has no functionality and is designed to be as lightweight as possible. A renderable object may end up as multiple render jobs, because it may contain surfaces of different materials.

Imagine a usual 6-sided dice, which is be created by the 3D artist using separate textures for each side. Its geometry is stored as a single mesh, which contains 8 vertices, and 6 polygons, which come in pairs of two triangles each. Each polygon is associated with a different texture and thus a different material. This means the dice mesh produces 6 render jobs, containing two triangles each[3].

The interface through which the transformation into render jobs is exposed, consists of the abstract methods `first_job` and `next_job`. `first_job` returns the first render job of the renderable or `NULL` if there are no jobs, which is an error state and is not guaranteed

[3] Although just a theoretical example, this would be an inefficient way to render a dice, as there is at least one draw call per render job during rendering. A way to reduce the number of render jobs is to cut down on the number of distinct materials. This can be done by putting multiple images into one big conglomerate texture, while adapting the texture coordinates of the mesh's vertices. In a similar way, lighting properties such as surface detail, reflectiveness and transparency can be stored in separate maps which current hardware can apply in one drawing pass to an entire mesh.

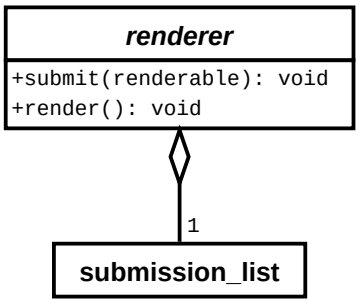

Figure 3.9: The Pace `renderer` interface class

to be expected and handled correctly. The `next_job` method returns the next render job of the renderable, or `NULL` if there are no more jobs, at which point the code using the interface must not call `next_job` again. The `next_job` method must also not be called before `first_job`.

The implementation of the `renderable` interface is free to choose if it holds the render jobs permanently, or if it generates them on demand. It is intentional that the interface specifies no container class where the render jobs are stored. This gives both the implementation and the client the choice of which container to use. There is also no indexed access which would allow the interface to consist of just one method. The reason for this is that there will be implementations of `renderable` which are itself composed of `renderable`s, and thus will have problems performing direct indexed access into their render jobs.

Renderer

The `renderer` class (figure 3.9) represents the interface to the rendering strategy, which describes the way in which a scene is drawn to the frame buffer at the highest level. It contains a submission list which stores incoming renderable objects and their render jobs (see section 3.3.7). A client can submit objects to be rendered by using the `submit` method.

Once the client is finished submitting `renderable`s, it calls the `render` method, which is the abstract part of the interface. Its implementation decides how the contents of

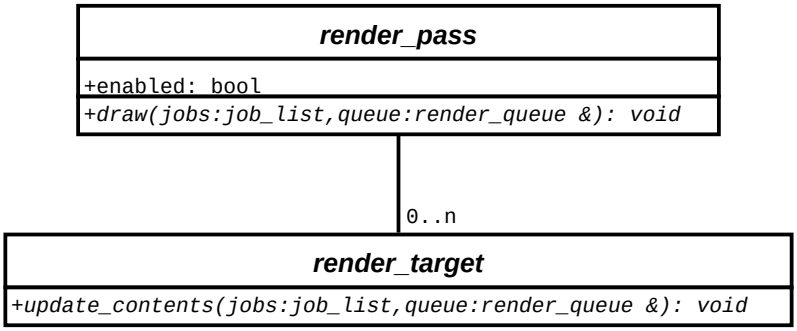

Figure 3.10: The Pace `render_pass` interface class

the submission list are processed. A default implementation is provided, which uses the concept of render passes to describe the rendering process in a dynamic way. Its `render` method simply delegates the rendering to a list of these render passes. During the following sections, the usage of the term *renderer* usually refers to this default implementation of the interface. The render pass interface is the topic of the next section.

Render Pass

Through the `render_pass` interface, a `renderer` implementation can perform multi-pass rendering. This is a process during which a scene may be drawn multiple times, either completely or partially, to one or more render targets. Render targets are objects which direct the results of rendering to a buffer which is not the frame buffer, like a texture. The frame buffer is always the default render target, so if a render pass is not given a render target, its jobs will be drawn to the frame buffer. As shown in figure 3.10, these render targets are associated with the render pass during setup time. The render passes are then added to a multi-pass implementation of the `renderer` interface, such as the one shown in figure 3.4 on page 37.

During rendering, the render passes are executed in the order in which they were added to the renderer, by calling the `draw` method. It receives the render jobs stored in the renderer's submission list. A render pass can choose to use this list or manage its own, so it can be used to render only parts of the scene. The jobs a pass wants to render are pushed into a render queue, which is also passed to the `draw` method and usually is the

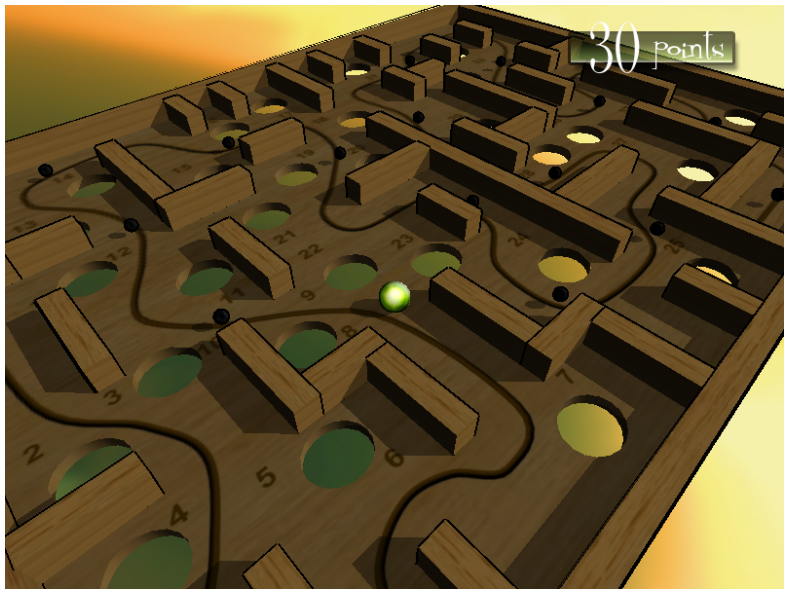

Figure 3.11: Screenshot from the game Powder, which uses one pass for the background, one for the playfield and the ball, and one to display the score in the upper right corner.

same object for all passes. Note that this is the way the default renderer implementation uses the render passes. This process is entirely customizable, which means that render passes could be executed in parallel, and they could use dedicated render queues per pass which work on different GPUs.

A small application which just needs to draw a scene probably needs only a single render pass. A typical game scene however, as the one shown in figure 3.11, usually requires at least 3 passes, which draw different parts:

1. A backdrop image, such as a sky or landscape, usually called sky dome or sky box
2. The geometry related to gameplay, the player model, and the local environment
3. A 2D overlay image displaying game status and parts of the user interface, usually called the head-up display (HUD)

These passes require different drawing strategies, and have to be rendered in the given order. It would be possible to push all these elements through a single render pass, and have the render queue figure out the correct order in which to draw them. However,

the separation leads to less cost for sorting, and individual passes can be toggled during run time by changing the `enabled` flag of the `render_pass` class. This is useful during development, and in case of the HUD, even to the end user.

The car racing game is another example for an application which uses such a render pass setup. Its scene contains only the cars and environment geometry, not the sky box or the HUD. To configure the renderer, it creates two implementations of the `render_pass` interface. For pass 1, a `sky_box_pass` is implemented, which is given a submission list containing the instance of the sky box mesh. Instead of processing the list of render jobs it gets from the default renderer, it only puts the contents of its own submission list into the render queue and flushes it. Furthermore, the pass uses only the orientation of the view, not the location, so the sky box seems to always have the same distance from the viewer. Pass 2 is rendered using a `default_render_pass`, which simply draws the contents of the scene. In pass 3, a `hud_render_pass` is used, which draws its own submission list, similar to the first pass. The list contains HUD elements like a speedometer, and the place of the player as well as his lap time. It uses an orthogonal projection which ignores depth. This means that no perspective view is used and all objects appear flat and at the same distance to the viewer. The game must make sure that the passes are added to the renderer in the correct order.

This is just a simple example, and current games use much more complex rendering phases. The application programmer has to decide how he creates a set of render passes which are capable of rendering the scene the way he wishes. This can be seen as the renderer's configuration. It can be modified or completely exchanged at run time, allowing various different rendering scenarios in a single application. It can also be implemented in a fully data driven way, which means that a content creator can write a file which contains the configuration of the renderer, which is loaded and processed together with the rest of the scene data.

The `draw` method of the `render_pass` class is responsible for drawing a list of render jobs. This list is treated as read-only, so there is no synchronization required if it is accessed from multiple threads. How the jobs are drawn is completely up the implementation, and can be done in manifold ways. To name a few:

- Direct rendering: the simplest way, where the list of jobs is drawn directly as they come using the state management API. This bypasses the render queue and should

only be used in exceptional cases.

- Queue insertion: a reasonable default implementation, where jobs are inserted into the render queue, which can sort and send them in batches to the state management API.
- Debug drawing: instead of drawing the render jobs, extract their bounding volumes and draw a visual representation of them to aid in the debugging of culling or collision detection. By using wireframe rendering, which only draws the outlines of polygons, this kind of debug information can be drawn over the existing scene as the final render pass, to make sure that bounding objects are placed correctly.
- Post-processing: draw a subset of the jobs with special shaders to achieve some kind of post-processing effect by blending it over the rendered frame, like glowing or blurring of objects. The jobs might also be drawn to a texture render target, to be used in a fragment shader which uses that texture to achieve the effect in a succeeding pass.
- Shadowing: project the shadows of the rendered geometry into the scene and darken the parts which receive no light from a given light source position
- Reflections: draw a reflected image of the scene, or multiple images in the case of cube mapping ([AMH02], pp. 156-158)

This list is an indication of what can be achieved using the `render_pass` interface. Some of the graphics effects mentioned above are very hard, if not impossible to implement in a single pass, due to limitations of the common GPU pipeline. Rendering shadows with the shadow mapping method (see [AMH02], section 6.12.4) requires one additional pass per light source: first, the scene is drawn from the view point of the light source into a depth buffer, which stores the distance from the light source to each pixel. Second, while rendering the actual scene, this depth buffer is projected as a texture into the scene in such a way that it matches the direction of the light. Each pixel is tested against the stored depth value and determined if it is visible from the light source. If not, the pixel receives no light from it. For GPUs which do not support shadowing in hardware, this obviously cannot be done in a single pass.

Similar restrictions apply to the rendering of reflections, where there is an extra pass required to draw the scene from the view point of the reflecting surface into a texture render target. This texture is then applied to the surface in the primary render pass.

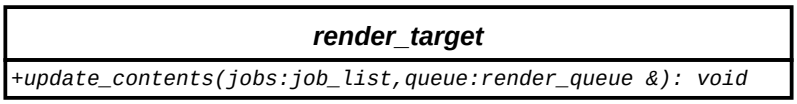

Figure 3.12: The Pace `render_target` interface class

With the explanations of these techniques, it becomes evident that there are often dependencies between the passes. To keep the implementation simple, this is not explicitly modeled in the current design. Instead, the client is expected to add the render pass instances to the renderer in the correct order.

An additional usage for render passes are what Porcino [Por] describes as the composite operations of a submission engine (see section 2.2.4). A render pass implementation may not render anything at all, but instead combine the results of passes which are already completed into one render target. This, again, can happen in different ways:

- Combine the colors of the pixels using a compositing operator, such as addition or modulo
- Combine separate tiles of the final image which have been rendered in parallel on multiple GPUs
- Combine separately rendered sets of objects, using the depth buffer to select the visible pixels

The way the render passes interact with the render targets is detailed in the next section.

Render Target

An implementation of the `render_target` interface (figure 3.12) directs a list of render jobs to a render buffer which differs from the default render target (most probably the frame buffer of the OpenGL context). Such a buffer might be a texture, which can be used in other passes to implement reflections, or show a different view of the scene on a virtual monitor screen.

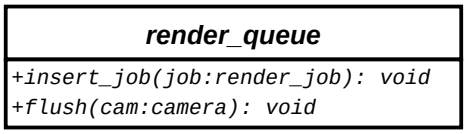

Figure 3.13: The Pace `render_queue` interface class

The racing game example could use a render pass with a texture render target to add a rear view mirror. The render pass is associated with a different view, and draws it into a texture, which has a lower resolution than the frame buffer. In the main render pass which draws the game view, the texture is mapped onto the frontal geometry of the mirror.

As mentioned in the previous section, render targets collaborate with render passes during the drawing of a pass. The method `update_contents` is called from the render pass, receiving the same arguments (render jobs and a target render queue) as the `draw` method of the pass. The task of the implementation is to bind an appropriate target render buffer if necessary, and then to proceed putting render jobs into the supplied queue, either by itself, or by using the render pass implementation.

Render Queue

The `render_queue` interface (figure 3.13) buffers render jobs which are ready to be drawn by the underlying state management API. In most cases, its implementation is the only part of the renderer which communicates directly with the API[4]. Therefore, it is the place where most API-specific optimizations will be applied.

The render queue sorts the incoming render jobs in a way which makes best use of the graphics pipeline by minimizing the amount of expensive state changes. The differences in cost of various state changes have been mentioned in section 2.2.1, while section 4.3.1 delves into the methods which can be used to implement state sorting. Although sorting is inevitable to efficiently utilize the GPU, the prioritization of state changes depends very much on the actual hardware and may change from one generation of graphics

[4]Implementations of the `material` interface, which is covered in the next section, are also managing render states, but their interface is used only by the render queue, as it binds materials before executing render jobs.

accelerators to the next. Therefore, the render queue is designed as an interface, whose implementation can be exchanged at run-time. An application can supply different render queue implementations for various system configurations and select the most appropriate one during initialization.

Additionally, the sorting scheme has to consider the coarse distance of objects from the viewer. This is motivated by two distinct problems: translucent rendering and minimizing overdraw. Both problems are connected to depth buffering, which is the process of performing depth comparisons for each fragment before it is written to the frame buffer (see [AMH02], section 15.1.3).

If translucent geometry is rendered in an arbitrary order, depth buffering may lead to artifacts in the final image. Assume two distinct, translucent objects at different distance from the viewer, but on the same line of sight, so they overlap when drawn to the frame buffer. If the object further away is rendered after the nearer one, it will not appear in the image at all, as it is cancelled out by the depth comparison, although it should be partially visible through the other object. The most obvious solution to this problem is to render in two passes:

1. Draw opaque objects
2. Draw translucent objects, in back-to-front order as seen from the camera[5]

By using a rendering technique called depth peeling, the sorting can be avoided (see [AMH02], section 4.5). It is implemented using fragment shaders which operate on several layers of depth, using multi-pass rendering. Each pass peels away the next depth layer and tests fragments against the new depth, which results in a rough back-to-front order. The drawback of this method is that it offers reduced depth sorting precision, which depends on the number of passes. Using more passes increases the amount of fragment shader overhead.

[5] This doesn't solve the problem at sub-object level. The renderer cannot easily influence the order of trianglesdrawn during a draw call. This means that during the rendering of an object, its triangles may exhibit the same problem and cause visibility artifacts if they are not drawn in back-to-front order. For convex, single color meshes, this can be solved by culling triangles which face away from the viewer (back face culling [AMH02], section 9.3.1), so that no overdraw occurs during the draw call. Meshes that do not fall in this category, and exhibit highly visible artifacts, have to be rendered twice. Once with front facing triangles culled, and a second time with back facing triangles culled.

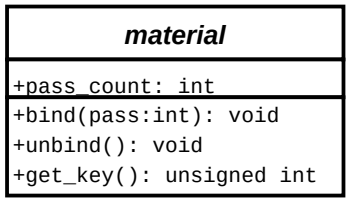

Figure 3.14: The Pace `material` interface class

The second problem which sorting is able to resolve, is the reduction of overdraw. On current GPUs, depth tests are very fast, so using them to effectively cull hidden pixels reduces the load on the fragment shading unit. In the first drawing step sketched above, the opaque objects may be sorted in front-to-back order. Thus, when objects overlap, only the objects in front will be actually rendered, and those visited afterwards are rejected by depth tests.

Besides sorting, the render queue has the task to synchronize parallel rendering with the OpenGL API. Potentially, there are many threads which produce render jobs, which the render queue has to store and pass on to OpenGL sequentially. Thus, a parallel implementation of the `render_queue` interface has to use thread-safe containers or locks to provide synchronization.

A render passes uses the `insert_job` method to put render jobs into the queue. When finished, it calls the `flush` method, which sorts the jobs and executes a draw call for each one. Parallel implementations might use a concurrent sort algorithm to improve scalability of this process. The draw calls must be executed sequentially, though. See section 4.3.2 for more details on the parallel render queue implementation.

Material

As mentioned in section 2.1.2, a material is an abstract definition of the visual properties of a surface. The most important part of the `material` interface (figure 3.14) is the abstract `bind` method. Its implementation is responsible for setting up render states and binding textures and shader programs to the corresponding shading units. The `unbind` method will undo any changes made to render states during the bind process.

Shader based applications require only one material implementation for the shading language in use, e.g. GLSL [Ros06], to represent almost any material by employing the programmable pipeline.

Section 3.3.2 stated that materials are created during the preprocessing stage and are stored as a part of the scene database. Most often, they are referenced by the meshes which make up the database. Once an instance which references such a mesh enters the renderer, the reference to the materials of the mesh are stored in its render jobs.

To support the render queue when sorting jobs to reduce state changes, materials generate sort keys. The method `get_key` returns an integer value which is similar to a hash value for the material and can be used to sort render jobs by material (see section 4.3.1). A render job caches this value so it does not need to be calculated each frame.

The `pass_count` attribute can be set to a value greater than 1 to express that the material has to be rendered in multiple passes. This is required in the rare case of a material which the GPU is incapable of shading pixels with in a single pass. Most of the time, this happens if there are not enough texture units available. However, this case has to be accounted for by the material implementation. Generic implementations like `glsl_material` will deliberately ignore this value, and binding the material will fail if the hardware is ill-equipped. In case of a `pass_count` value greater than one, the default implementation of the render queue will call the `bind` method repeatedly, passing the number of the current pass, followed by a draw call, which renders the geometry.

3.3.7 Data Structures

This section discusses classes which manage the data passed between the interfaces presented previously.

Buffer

Buffers are container classes which encapsulate the storage of data which is potentially located in the memory of the graphics card, also called video memory or VRAM. As

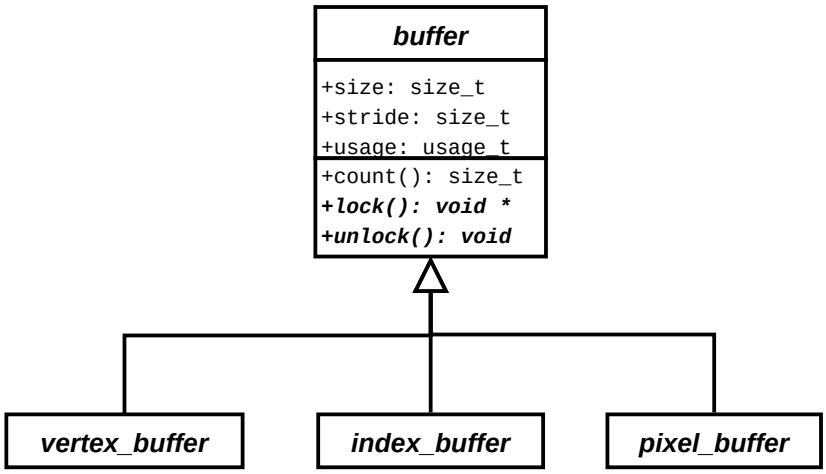

Figure 3.15: The Pace `buffer` class

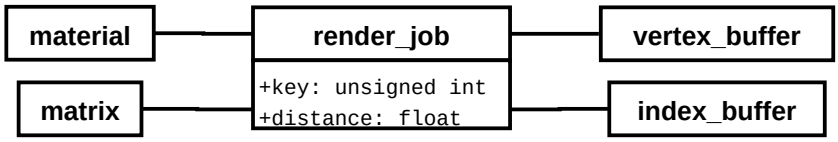

Figure 3.16: The Pace `render_job` class

there are several kinds of data types eligible for VRAM storage, as well as different methods for VRAM allocation and access, the `buffer` class (figure 3.15) also contains an abstract interface part.

The data types stored in buffers are vertices, indices, and texture map contents (mostly pixels). These are handled by the `vertex_buffer`, `index_buffer`, and `pixel_buffer` classes, respectively. Each of these adds information about the specific format of the stored data to the buffer interface. See section 4.2.2 for a more detailed discussion of buffer formats.

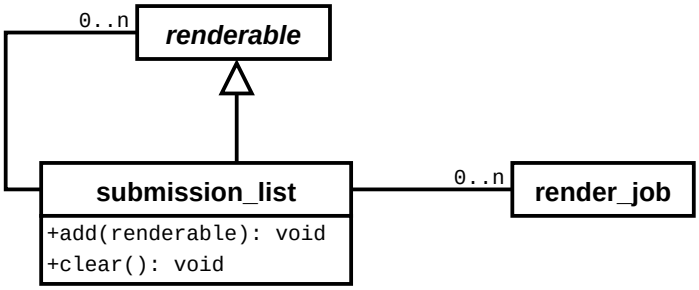

Figure 3.17: The Pace `submission_list` class

Render Job

As mentioned previously, a render job is usually part of a renderable object. The `render_job` class (figure 3.16) references all objects involved in a draw call:

- Vertex buffer
- Index buffer
- Material
- Transformation matrix

The steps involved in executing a render job are presented in section 4.3.

Introducing the `render_job` class instead of only working with the `renderable` interface throughout the rendering pipeline is motivated by two related arguments: being a very lightweight class, render jobs can be stored as an array of structures, which can be processed in a linear, cache-friendly fashion. Furthermore, because the submission list manages the render jobs, they are guaranteed to be read-only during the rendering process, which makes them suitable for fast, unsynchronized parallel access.

Submission List

The `submission_list` class (figure 3.17) is a container for instances of `renderable` objects, as well as their render jobs. As displayed in the figure, it is a `renderable` itself.

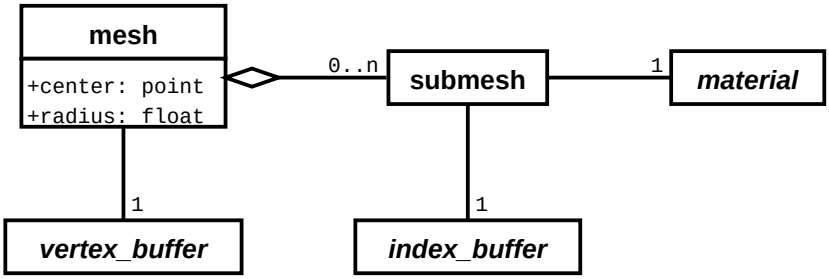

Figure 3.18: The Pace `mesh` and `submesh` classes

This means that submission lists may contain other submission lists. This can be used to build rendering sequences which are optimized and sorted during the preprocessing phase. If a given set of renderable objects is known not to change state during rendering, the application can put them into a precompiled submission list. It will generate sort keys and store them in the render jobs, so this does not need to be done during rendering. The jobs are also sorted prior to rendering, which reduces the effort to sort during the real-time phase. The list can be submitted to the renderer like any other renderable object.

The render jobs cached in the submission list can be directly read by the renderer. However, if one of the contained renderables changes its contents and this results in a different set of render jobs, the submission list has to be cleared and all renderables must be resubmitted.

Mesh and Submesh

The `mesh` class (figure 3.18) is a data structure which is part of the scene database (see section 2.2.4). It is composed of a vertex buffer and a list of submeshes. Additionally, it carries information about its bounding volume, in form of the attributes `center` and `radius`, which describe the smallest enclosing sphere of the mesh's geometry. This information is used for visibility culling and collision detection. See the next section on how culling can be implemented.

Splitting up the mesh into submeshes is motivated by the dice example given in section 3.3.6. The dice mesh has 6 unique materials assigned to its sides. To represent this, the

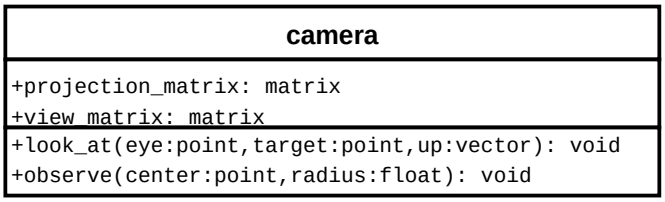

Figure 3.19: The Pace `camera` class

`mesh` object has 6 `submesh` objects, each with its own material and index buffer. Each index buffer is used to index into the mesh's vertex buffer to build the triangles of the respective side of the dice. For example, the index list (0, 1, 2) describes a triangle which consists of the first 3 vertices in the vertex buffer. There are several different types of index lists, which are explained in [AMH02], section 11.4.5. An index buffer is simply an array containing a sequence of indices, and has a primitive type, which tells how these indices have to be interpreted to form triangles or lines.

It is important to note that the submeshes are fixed and can only index the vertex buffer of the mesh. This means that a single mesh cannot represent a 3D model of an object whose parts can move relative to each other, like a car and its wheels. Though they are connected, separate meshes have to be used to render the car's body and each wheel. To model such a relationship of connected meshes, an application usually uses some form of parent/child relationship. A scene graph (see section 2.2) is an example for such a structure. It ensures that when the body of the car is moved, the wheels are moved along with it.

Character animation may state an exception to the above rule, as there are techniques like skinning and morphing, which allow to animate the parts of a single mesh independent of each other. These are special cases however, and the degrees of freedom for movement are limited. Also note that breaking down an object into multiple meshes may increase visibility culling accuracy, while the amount of draw calls necessary to render the object is usually similar.

Camera

Cameras in computer graphics are a representation of the user's viewpoint and the direction at which he is looking into the scene. The `camera` class (figure 3.19) features the minimum requirements to represent such a model. By specifying two transformation matrices, OpenGL calculates the resulting view direction and projection of 3D coordinates on the screen. Thus, those matrices are called the *view matrix* and the *projection matrix*, respectively.

The default render pass implementation has an associated `camera` instance, which is used to set the view- and projection matrices before flushing the render queue. Alternatively, if only a single camera is used, these matrices can be set by the application prior to rendering.

The camera can also be represented as a capped pyramid, called *frustum*. It has 6 planar sides, which can be used to test if an object is visible by the camera (see [AMH02], section 13.13). The frustum can be extracted from the view- and projection matrices, which is done at the beginning of a new frame, after the position and direction of the viewer are known. Then, for each object in the scene, the bounding volume of its geometry is tested for visibility against the frustum planes. Invisible objects do not need to be submitted to the renderer for the current frame. Depending on the camera position, its field-of-view, and the distribution of objects, this can reject more than half of the contents of a scene.

4 Implementation

The rendering architecture outlined in the last chapter already hinted at how most of the functionality would be implemented. This chapter supplements the concept by giving details on solutions to problems which were encountered during the implementation phase.

It opens with the description of the libraries which were used to support the parallel renderer in areas which are outside the scope of this thesis.

The remaining sections cover various parts of the rendering library which were non-trivial to develop, motivate implementation-specific decisions, and explain some technical details which were only briefly mentioned in the previous chapters.

4.1 Software Libraries

Pace uses some excellent open source libraries to support it in areas which are either platform-specific, or too broad to be covered by a lightweight framework.

These libraries are, in no particular order:

- Intel Threading Building Blocks: task-based threading interface, parallel algorithms and thread-safe containers. The next section explains the motivation for choosing TBB for parallelizing Pace.
- VectorMathLibrary: part of the open source Bullet physics engine [Bul]. It provides vector arithmetic functions for C and C++ applications. It contains optimizations for several hardware platforms, including PowerPC, PlayStation 3, and x86 PC with SSE.
- Simple Direct-Media Layer (SDL): used for portable graphics and input management, as well as portable access to threading functionality.

4.1.1 Intel Threading Building Blocks

The Intel Threading Building Blocks (TBB) library [Rei07, TBB] provides a C++ interface to task-based parallelism. Tasks in TBB are objects which carry out small pieces of work independently of each other. A scheduler distributes tasks among worker threads provided by the operating system. By working with tasks instead of threads, parallel programming becomes more intuitive. It is easier to divide an algorithm into separate tasks than to attempt to create a threaded model of execution.

Because task-based parallelism works at a higher level than threads, it is easier to make it scalable. If enough tasks are available, TBB can map them onto as many worker threads as possible, usually equal to the number of cores in the system. The number of worker threads is decided at run-time, so the application does not have to be recompiled to run faster on a different system with more cores available.

Pace uses TBB's lock-free containers to create a producer-consumer queue for the parallel

implementation of the render queue (see section 4.3.2). It also uses its parallel algorithm `parallel_for` to parallelize visibility culling. It is a parallel version of `std::for_each` which creates tasks for ranges of elements within the iterated sequence and executes them concurrently. The size of these ranges can be adjusted to balance parallelism and cache usage, as it is very efficient to work on elements of a range that fit into the cache at once.

4.2 OpenGL Crowd Rendering

To get the best performance out of the graphics hardware, one has to look behind the abstract view of the rendering pipeline provided by the state management API. Figure 4.1 shows how an application communicates with the underlying hardware. There are several points which are critical to rendering performance.

4.2.1 Efficient Drawing

The number and cost of draw calls depend on the OpenGL commands used. There are several ways of issuing draw calls:

- Immediate mode: The least efficient way to draw is immediate mode. It uses one or more calls per vertex and requires the driver to create a copy of the data each time. While it is the easiest method to use, immediate mode does not scale well for large data sets and is not recommended for crowd rendering applications.
- Display lists: For static geometry with vertex data that does not change frequently, display lists can be used to accelerate drawing. A display list records immediate mode draw calls and compiles them into an atomic unit, which can be issued to OpenGL for rendering in a single call. The driver is allowed to reorder the calls inside the list to achieve better performance. It will also most probably transfer the vertex data it has collected to the local memory of the GPU (video memory, or VRAM), which it can access much faster than standard system memory.
- Vertex buffer objects (VBO): To give the application more control over the storage of vertex data, the vertex buffer objects extension was added to OpenGL. They

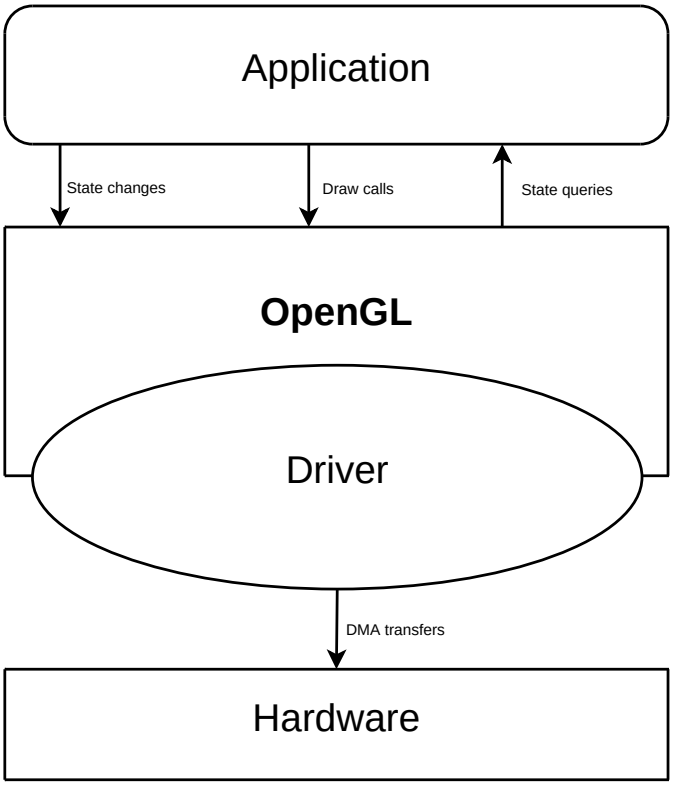

Figure 4.1: Model of communication between an application and the graphics hardware through OpenGL

combine the speed improvements of display lists with an efficient way to manage dynamic geometry data. Section 4.2.3 contains a detailed discussion of VBOs.

The large number of draw calls required to draw a crowd scene can be countered by the usage of hardware instancing techniques (see section 4.2.4), which accelerate the drawing of large numbers of instances using vertex programs on the GPU. Hardware instancing can be combined with each of the options above.

4.2.2 Vertex Buffers

The usage of vertex- and index buffers follows the paradigm of *retained mode* rendering, as opposed to *immediate mode*[1]. In this context, retained mode means that vertices passed to OpenGL are not drawn immediately. Instead, after submitting vertex data in a buffer, the application enters its rendering loop and sends a request for parts of the buffer to be drawn. The time span in between can be used by the graphics driver to optimize the data in any ways it sees fit. Additionally, it can store it in VRAM, which the GPU has fast access to. Both techniques should generally increase rendering speed. However, retained mode is significantly more involved from an application programmer's point of view, when compared to the simple immediate mode API. Furthermore, it is difficult to provide methods to render dynamic vertex data. This is a disadvantage for applications which modify their meshes on the vertex or triangle level. The data in the buffers has to be updated each frame, which gets in conflict with the optimizations just mentioned. Buffer contents which are regularly changing should not be optimized between rendering calls, as this would lead to more computation cost than the results are actually worth. There is also a cost for transferring the data to VRAM which happens over a comparatively slow system bus.

Vertices can contain several types of information about the geometry, called *attributes*. Attributes are attached to vertices to influence the way the geometry is to be rendered. Some attributes are very common, such as position, normal vector, and texture coordinates (see [SWND05], chapters 2 and 9), so they are part of the OpenGL standard. There are other standard attributes, but they have little significance when using the programmable pipeline. Shader programs have lead to the inclusion of generic vertex attributes. They can be used to store arbitrary data accessible by a vertex shader program to compute lighting or influence rendering in user defined ways.

To account for these possibilities, the Pace `vertex_buffer` interface lets the client specify a `vertex_format` (figure 4.2) object, which contains the description of the attributes used by the stored vertices. The vertex buffer is just a continuous block of memory, in which vertices are stored one after the other. Each vertex is subdivided into one or more attributes, which most often contain a set of float values. Figure 4.3 shows the contents of an exemplary vertex buffer.

[1] Scene management libraries are often called retained mode APIs, which makes it an ambiguous term. It is used here strictly in relation to buffer objects.

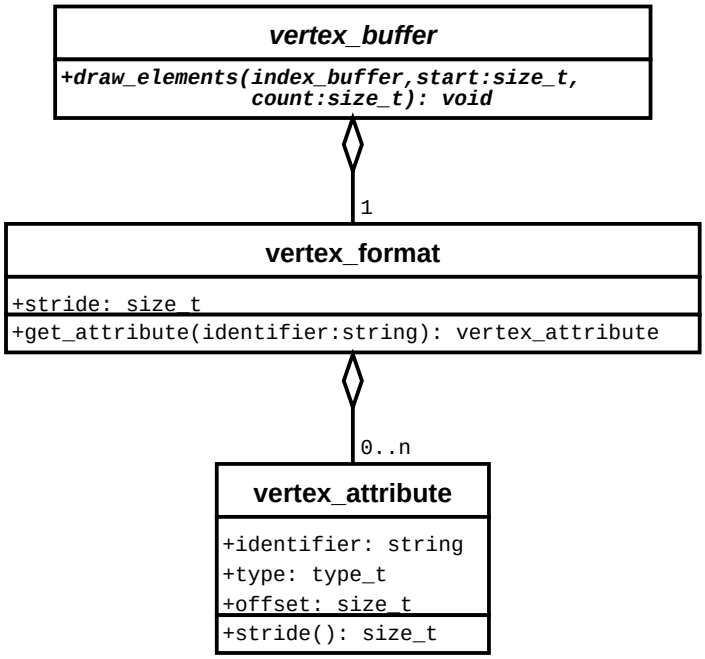

Figure 4.2: The Pace `vertex_buffer`, `vertex_format` and `vertex_attribute` classes

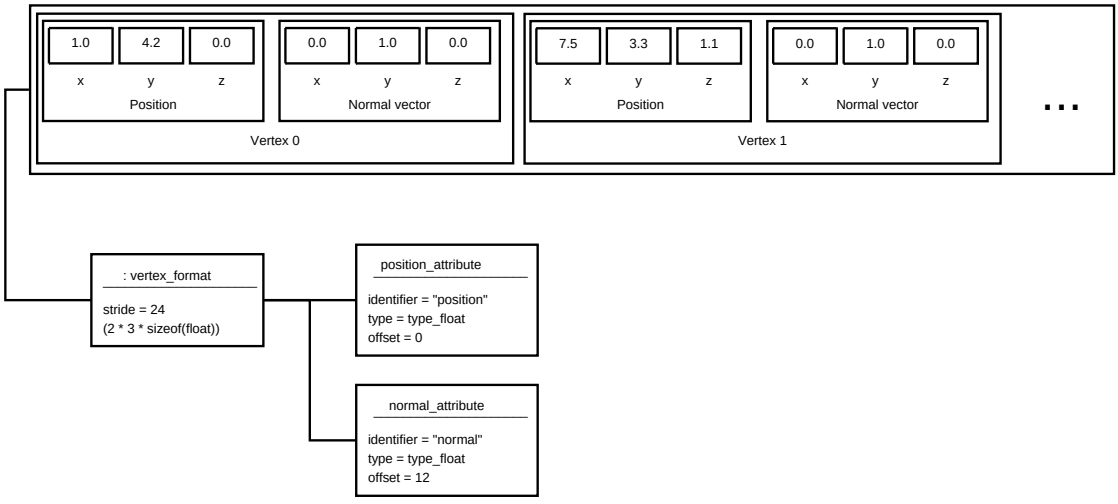

Figure 4.3: An example of a vertex buffer and its vertex format

To store and retrieve values from a vertex buffer during preprocessing, an instance of the `vertex_attribute` class provides an attribute's offset, which is its location relative to the start of the vertex in bytes, and its stride, which stores the number of bytes the attribute occupies. The location of an attribute of the vertex n, given the vertex stride s and the attribute offset o, can thus be calculated as $n \times s + o$.

4.2.3 Vertex Buffer Objects (VBO)

Vertex buffer objects [VBO03] are a variety of OpenGL's *buffer objects*. The purpose of buffer objects is to encapsulate data to increase transfer speed and give the hardware more control over it. A well known kind of buffer object is the texture. After the creation of a buffer object, the client receives its identifier, which is an integer value. Using this identifier, the client can bind the buffer to the OpenGL context, and modify the data stored inside it. During rendering, the buffer must also be bound when used, so the GPU can use the identifier to access the buffer data.

To access a VBO, its contents can be mapped into system memory. The client receives a pointer to the memory location, and can then write to it. This happens as described in the previous section. Afterwards, it is unmapped and the pointer to the memory becomes invalid. The driver transfers the data to VRAM and the vertex buffer is ready to be used for rendering. If the buffer contents are not modified after the initial transfer, this results in a much better rendering speed than client side vertex buffers in system memory allow. However, if the contents are dynamic, and need to be updated at least every few frames, there is less performance to be gained. Furthermore, VRAM is a limited resource, so if all of it is already allocated for texture and vertex data, the VBO will remain in system memory. The OpenGL driver has control over the allocation, so it will try to keep frequently used VBOs in VRAM, while swapping out data which was not referenced for a longer period of time.

The `usage` attribute of the `buffer` class (see section 3.3.7) is an enumeration type which defines how the buffer will be used, which is important when creating a VBO. There are several usage scenarios represented by the enumeration, which differ in the way the data is accessed by the CPU:

Enumeration value	Written	Read	Storage
`static`	Once	Frequently	System memory or AGP
`dynamic`	Frequently	Frequently	AGP
`static_write_only`	Once	Never	VRAM
`dynamic_write_only`	Frequently	Never	AGP
`dynamic_discardable`	Each frame	Never	VRAM

AGP (Accelerated Graphics Port [Mac01]) is a high speed bus which connects the graphics card to the PC motherboard. It offers higher bandwidth than older busses, but has been surpassed by the PCI Express bus. The term *AGP memory* for 3D graphics still remains in use, though. AGP memory is sort of a compromise between system memory and VRAM storage, and is especially useful for data which has to be read by the CPU during rendering. Using AGP means that while the data remains in system memory, it is stored in a way that allows the graphics card faster access to it. Note that if a buffer is stored in VRAM, it can still be read by the CPU, but this will trigger a very slow bus transfer. Thus, it is important to specify the appropriate usage option when creating a buffer.

Usually, `static_write_only` will be the most frequently used option, as it guarantees fast rendering, and most vertex data in a game will not be modified after it was loaded. For dynamic vertex buffers, `dynamic_write_only` is the best option regarding performance. If the client needs to read back vertex data from a buffer, it might consider to keep an extra copy of it in system memory for this purpose.

Despite their name, VBOs can also be used to store index data. To access the contents of an index buffer, it is mapped into system memory exactly like a vertex buffer. The application now only needs to know the size of an index (usually either 2 or 4 bytes), which it can retrieve from the `stride` attribute of the buffer.

Binding a VBO for rendering is an expensive state change. To reduce the number of times the VBO needs to be switched, multiple vertex buffers can be stored in a single VBO, which requires adding an offset value to the buffer address during rendering so it points to the start of the actual buffer. During the initialization phase, an application may not know the size of all vertex buffers combined when allocating memory for a VBO. To address this problem, Pace offers the `opengl_memory_manager` class, which manages

VRAM using a first-fit memory allocation scheme.

4.2.4 Hardware Instancing

Hardware instancing (also called *shader-based instancing*) is used to decrease the number of draw calls required to render large crowds. As stated in section 2.2.1, draw calls cause overhead as they require the CPU and the GPU to communicate. The overhead can be reduced by ensuring that as little data as possible has to be transferred from system memory to the GPU. Still, when drawing large sets of instances with no state changes in between, a draw call per object wastes precious CPU time.

Usually, there is little data which differs from one draw call to the next. This data usually contains the position and orientation of the instances. By submitting this data up-front, which is similar to buffering vertices, hardware instancing makes draw calls very cheap. Pure hardware instancing uses a buffer object to store these values, which is then transferred to VRAM. To draw the instances, a single draw call is sufficient. Instead of the function `glDrawElements`, which is used to draw regular vertex buffers, the extension function `glDrawElementsInstancedEXT` must be used. The function receives the number of instances to be drawn. Internally, the GPU then executes the draw calls while incrementing an instance identifier value, which starts at 0. This value can be read by the vertex program from a special parameter. Using this value as an index, the program can read the per-instance values from the corresponding buffer object. Using the extension, the maximum number of instances in one draw call is 1024. The application should make sure that most of these instances are actually visible from the current view.

The extension is only available on graphics cards supporting OpenGL 2.0, which is currently not yet wide-spread. A hardware instancing technique which works on older graphics cards uses shader parameters to store per-instance information. This technique is also called *pseudo-instancing* [Zel04]. Figure 4.4 shows the difference in performance between the techniques, which is impressive, especially when compared to using no instancing at all.

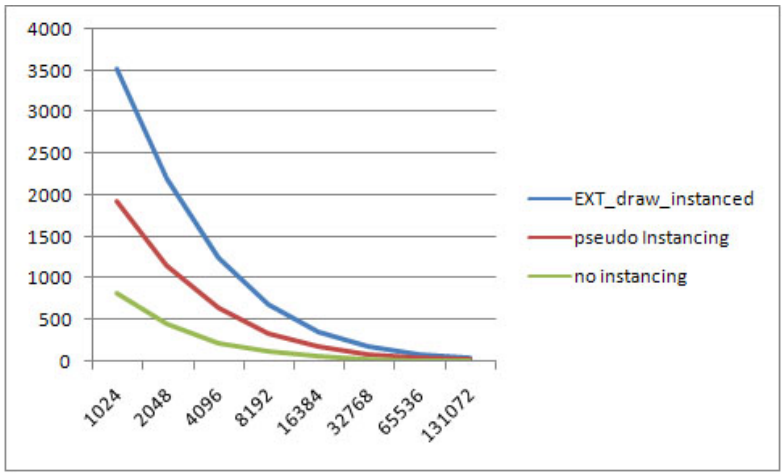

Figure 4.4: Performance comparison of hardware instancing and pseudo instancing [Tha]. The x-axis displays the number of instances, while the y-axis shows the frame rate in Hz. Measured on an Nvidia Geforce 8800 GTX.

4.3 Render Queue

The render queue is the most low-level part of the Pace rendering pipeline. As discussed in section 3.3.6, it directly works with the state management API and is thus responsible for utilizing the GPU the best way possible. The following two sections explain concepts which are used to achieve this, which are render state sorting and parallelization.

4.3.1 Sorting by Render State

Section 2.2.1 already explained that it is advisable to minimize the amount of render state changes when using a state management API. In Pace, render states are hidden behind the abstract `material` interface. To know the best order of execution for the incoming render jobs, the render queue can use their *sort key*, which is stored as a 32 bit integer value. This sort key is generated by the material's implementation of the `get_key` method.

As an example, the GLSL material implementation uses the identifier of its shader program as the lower 8 bit part of the sort key. This way, all render jobs with the same

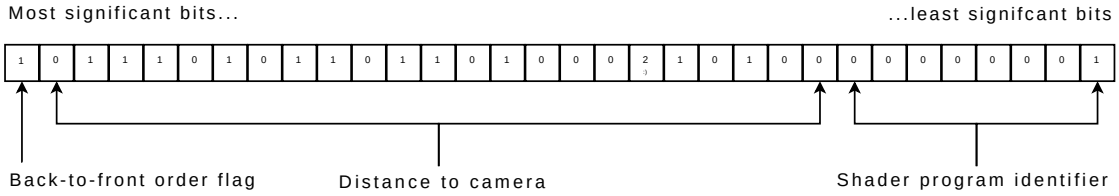

Figure 4.5: Structure of a material sort key

shader program are rendered in succession. The other 24 bits of the key are used for depth sorting. If a client of the GLSL material uses translucent rendering, he must set the `requires_back_to_front_order` attribute of the material to `true`. This will raise the most significant bit of the sort key, and cause the translucent render jobs to be drawn last[2]. The remaining bits in between are used to store the most significant bits of the render job's `distance` attribute, which contains the distance of the instance to the camera for the current frame. A combined sort key is illustrated in figure 4.5.

Using a sort algorithm such as `std::sort`, the job queue can be sorted like this:

```
std::sort(jobs.begin(), jobs.end(), &compare_job_by_key);
```

The `compare_job_by_key` function is defined by the header file of the `render_job` class. TBB provides a parallel version which is called `parallel_sort` and has the same signature as above. This algorithm is used in the TBB implementation of render queue, which is detailed in the next section.

4.3.2 Parallelization

The parallel implementation of the render queue uses TBB and its lock-free containers (see section 4.1.1). Their usage was decided based on the argument that locking techniques such as mutexes or semaphores are slower than lock-free methods ([Rei07], chapter 5). The `tbb_render_queue` uses a `concurrent_queue`, which is a lock-free container offering a first-in-first-out queue interface.

[2]Note that this does not work correctly with intermediary submission (see section 3.3.4). If the client requires this feature, it needs to use a separate render pass for translucent objects.

The queue contains *job buckets*, which are little else than ordinary lists of jobs. The reason to avoid storing render jobs directly in a `concurrent_queue` is that the render queue can operate on the buckets' lists without synchronization. The only time synchronized access is required is when pushing/popping buckets from the queue[3].

During rendering, the render passes send jobs to the render queue, which stores them in a bucket. When the render pass flushes the render queue, it pushes the bucket into the bucket queue. Meanwhile, the OpenGL thread has called the `draw_jobs` method of the render queue, as shown in listing 3.2 on page 42. The task of this method is to sleep until buckets arrive. On such an event it pops the bucket from the bucket queue to process it. This is a standard producer-consumer queue, with the render passes representing the producers and the render queue being the consumer.

[3]Note that this optimization is only feasible if render passes are not executed in parallel. Future versions of Pace might thus have to use synchronization on each queue access or several queues which feed a central queue (divide-and-conquer).

5 Results

In this chapter, the goals of Pace are examined in terms of performance and flexibility, while it is discussed how close Pace has come to achieve them.

Regarding the performance of Pace, an artificial test application called *Pacemark* was constructed and measured. For each test, the raw throughput was determined in instances per second as well as triangles per second. Using a run-time profiler implemented for use in Pace applications, an application profile was generated and analyzed. The profiler is used to measure the time spent in certain areas of an application, and outputs how much these areas consume of the frame time. This results in a clear indication of the location of an application's bottleneck, and was used to give an estimation of the scalability of Pace.

Pacemark loads a mesh from a file and generates a regular, three-dimensional grid of instances from this mesh. The default mesh, which was used to compare the application on different machines, is a box which contains 8 vertices and 6 submeshes. This mesh requires 6 draw calls to render, of which each one draws only 2 triangles. The intention of using this particularly simple kind of mesh was to shift the bottleneck of the application to the CPU, to be able to measure scalability when using a multicore CPU.

The following table shows the throughput of the default scene of Pacemark on different hardware and operating systems. The default scene consists of 8,000 instances of the box mesh. Instances per second means actually rendered instances, about 78% of the scene is removed by visibility culling. Note that the values for frames per second (FPS) and instances per second are not comparable to a real application, because geometry and scene structure are constructed in a very unusual way. The comparison was performed to see which factors influence the frame rate the most.

#	CPU	GPU	OS	FPS	Instances/sec.
1	2x 2.2 GHz	Geforce 8600 GT	Windows XP	67	118,000
2	2x 2.2 GHz	Geforce 8600 GT	Ubuntu Linux	43	75,000
3	2x 2.0 GHz	Mobility FireGL 5200	Windows XP	42	74,000
4	1x 3.0 GHz	Geforce FX 5500	Ubuntu Linux	26	46,000
5	4x 2.0 GHz	Geforce 7800 GTX	Ubuntu Linux	56	99,000

Performance in this test seems to primarily depend on the GPU and its driver, and the profiler confirms this. On system 1, the largest part of the frame time is consumed in the SDL function `SDL_GL_SwapBuffers`. This call is responsible for displaying the contents of the frame buffer on screen. During most of the time spent in the function, the CPU is waiting for the GPU to finish rendering.

With the graphics card in this system, it is not possible to process enough instances to make the CPU the bottleneck. Even with a scene consisting of 1,000,000 instances, one third of the frame time is spent waiting for the GPU. Obviously, the GPU is overwhelmed by such a large scene, and it needs about 2 seconds to render a single frame. Yet, the CPU has lots of time to spare, and the operating system shows that both cores are at about 70% of their capacity. This makes it hard to determine scalability on the CPU level.

By replacing the VBO implementation of the `vertex_buffer`, which contains the OpenGL draw call, with a dummy implementation which doesn't draw anything, the bottleneck is shifted to the render queue. The most time is spent sorting render jobs and binding materials before drawing a job. The latter task must be executed serially, but the sorting is parallelized by using TBB's `parallel_sort` algorithm, which uses a parallel quicksort. This improves CPU-level scalability and leads to a performance increase of 15% for a scene of 64,000 instances on a machine with 4 cores. Besides the remaining serial part of the render queue, the rendering loop now scales quite well and keeps 4 cores running at almost 90% capacity, which leads to an overall 200% frame rate increase compared to single-threaded execution. However, consider that this is a very contrived example and as soon as the GPU bottleneck factors into the equation, the gains of multithreading are diminished.

Nevertheless, this demonstrates that once GPUs are fast enough to shift the bottleneck to the CPU part of the graphics pipeline, Pace will be able to accelerate applications on

multicore CPUs by using task-based parallelism for collecting and sorting render jobs, as well as visibility culling. These parallelization efforts are also of use for current real-world applications which use the CPU extensively. Such applications gain more CPU time to spend by using multiple cores in certain periods of the display phase. For CPU-limited applications, it is also important to exploit CPU/GPU concurrency by performing their calculations when the GPU is busy (see section 3.3.4). If the GPU starves because the renderer is not able to execute draw calls at the right time, the overall frame rate will suffer.

However, profiling of Pacemark uncovers that not all OpenGL drivers actually give the application the opportunity to benefit from CPU/GPU concurrency. In fact, only system 1 actually executes draw calls asynchronously on Windows XP. The same system running Ubuntu Linux using recent drivers does not exhibit this behavior, and this might be a reason why the frame rate is lower. As mentioned in section 3.3.4, Pace is able to issue draw calls from a separate thread, so the other parts of the display phase as well as the update phase can run concurrently, regardless of this shortcoming. It is recommended to install the latest version of the OpenGL drivers available for the target machine, as driver-level concurrency is still able to improve performance by a small margin.

Figure 5.1 shows that the throughput of instances per second is only marginally influenced by the actual number of instances rendered. This proves that Pace is able to cope with very high numbers of instances internally. Again, this is very application-specific, and it is impossible to make an absolute statement whether Pace is fast or slow compared to existing rendering engines. It depends on many aspects of the application, e.g. how it provides its geometry, the data structures used for culling, as well as the amount of GPU load caused by shader programs. But as Pace is very flexible and configurable, it can be used to balance the graphics rendering pipeline. Hardware instancing could be used by implementing a render queue which supports it, but the application still has to adapt its vertex shaders to use it. Thus, hardware instancing is not a feature of Pace, but it can be easily implemented and used for a given application. This is an advantage over the "black box" principle of other renderers, which try to get good performance out of a generic scene description, without the option to benefit from the knowledge of the application designer.

Creating applications using Pace is certainly more involved than using a high level scene management library or game engine, but they benefit from the lightweight structure as

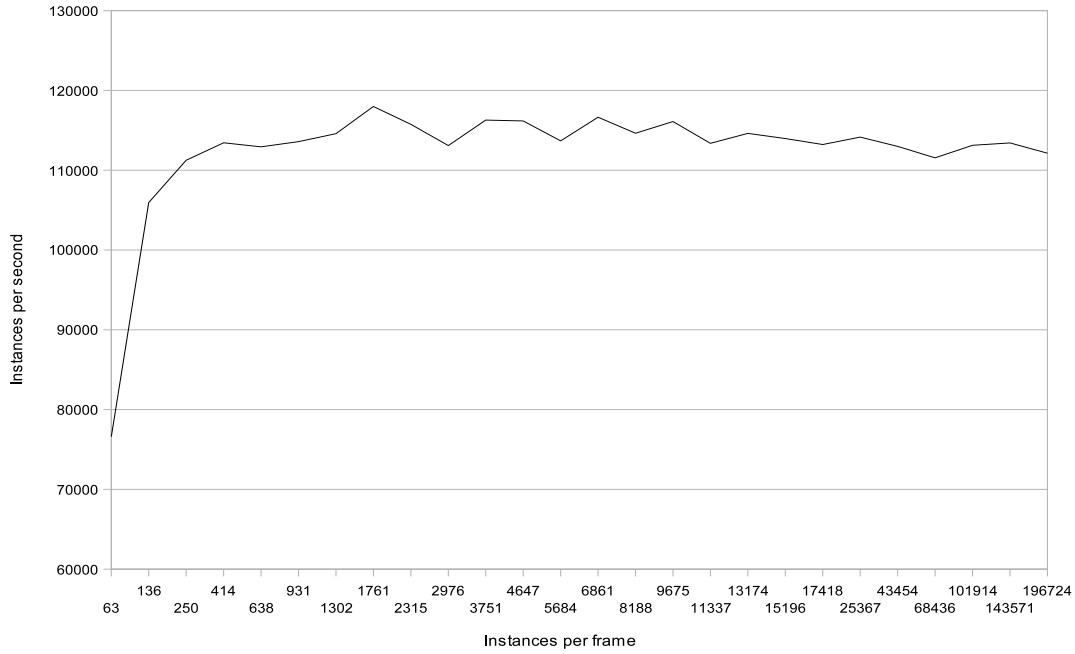

Figure 5.1: Pacemark scales well with the number of instances, as the throughput of instances per second stays almost constant.

well as the customizable implementation and the possibility to make use of almost any rendering concept possible.

Additionally, Pace offers a few interfaces which abstract OpenGL features such as VBOs, including video memory management, and GLSL shader programs, and handles OpenGL error states transparently. It also simplifies the usage of vertex and index buffers and contains all necessary data structures for scene database management. Furthermore, it contains utility classes to provide debug logging and code profiling, which are required for most graphics applications.

Pace compiles on Windows using the MSVC and MinGW compilers, and on Linux and MacOS X using the gcc compiler. On all compilers, the code is completely free of compiler warnings, even on the highest possible warning levels.

6 Conclusion

In this thesis, a flexible parallel rendering toolkit, *Pace*, was introduced to approach the problem of rendering scenes containing large amounts of objects (*crowds*). Motivated by a description of the current state of the art in graphics rendering hardware and software, a concept was developed for a submission-based renderer. Such a renderer is able to drive an underlying state management API, such as OpenGL, to utilize the available graphics hardware to maximum capacity.

To feed the renderer, the usage of a scene database was advocated, which contains the applications 3D content, including meshes and materials. It was also stated that it is important to manage separate data structures for the scene database, the scene itself, and the model of the application. The scene contains instances of the contents of the scene database and is submitted to the renderer. The model is used to represent the internal state of the application, and is synchronized with the scene to be displayed by the renderer.

On the basis of a case study, the integration of the rendering loop into the application was demonstrated. In this context, several kinds of concurrency for parallel rendering were discussed. It was shown how to leverage CPU-level concurrency and CPU/GPU concurrency using Pace. CPU-level concurrency turned out to have no effect on the rendering itself, as most of the work is currently done on the GPU. Most CPU-limited applications use the CPU for many things besides rendering, so separate parallelization effort is required to make them scalable. Applications which are GPU-limited, however, will not benefit from additional CPU cores at all, although they can still use Pace to achieve CPU/GPU concurrency and possibly more potential for GPU pipeline balancing using its flexible renderer.

The design of Pace can be considered stable. Parts of the implementation are still lacking,

but will most probably be improved when being used more extensively. The interface is expected to be flexible enough even for very demanding applications, and the runtime configurable renderer allows very fast testing turnaround times and provides useful debugging features.

It is important to note that most of the flexibility of Pace lies in its design, not its implementation. Most of the code which implements a certain functionality is very encapsulated, simple, and often even trivial. There are few dependencies between these implementations, and on their own, they provide almost no options for customization. But through the combination of these small blocks of code, a very flexible system is possible. During the development of Pace, modularity was always preferred over versatility. Sections 3.3.6 and 3.3.7 show that most classes have a very simple interface and few dependencies on other classes.

Future improvements could include spatial data structures to accelerate visibility culling. Flat, linear data structures like grids can be traversed in parallel very well and should also be able to speed up collision detection and other spatial queries required for games.

Multi-GPU concurrency is considered possible using Pace, but was not yet implemented. For applications which make extensive use of independent render passes, this should improve scalability considerably. In combination with spatial data structures, it could be possible to subdivide a scene to be rendered simultaneously on multiple cores using separate GPUs, while compositing the results into a single frame.

Pace was not yet used in a real-world application. It will become the replacement for an existing, serial rendering backend of the game engine which was used to create the game Powder (see figure 3.11). It is also planned to be used as a testbed for research on parallel rendering.

Bibliography

[AMH02] Tomas Akenine-Möller and Eric Haines. *Real-time Rendering*. AK Peters, second edition, 2002.

[Bul] Bullet physics library. Website. `http://www.bulletphysics.com`, last visited March 23, 2008.

[Bur] John Burkardt. Wavefront object files (.obj) reference. `http://people.scs.fsu.edu/~burkardt/txt/obj_format.txt`.

[Col] COLLADA. Website. `http://www.khronos.org/collada`, last visited March 14, 2008.

[For] Tom Forsyth. Blog of Tom Forsyth. Website. `http://www.eelpi.gotdns.org/blog.wiki.html`, last visited March 19, 2008.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[Gue06] Paul Guerrero. Rendering of forest scenes. Report, 2006. `http://www.cg.tuwien.ac.at/research/publications/2006/G_P_06_RFS`.

[HKO07] Mark Hummel, Mike Krause, and Douglas O'Flaherty. Protocol enhancements for tightly coupled accelerators. Technical report, AMD, Inc., 2007.

[Hor] Horde3D. Website. `http://www.nextgen-engine.net`, last visited Feb. 23, 2008.

[Int] Intel Tera-scale computing research program. Website. `http://www.intel.com/go/terascale`, last visited March 17, 2008.

[KRD+03] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *IEEE Computer*, pages 54–62, 2003.

[Mac01] Dean Macri. Fast agp writes for dynamic vertex data. *Game Developer*, pages 36–42, May 2001.

[MM05] Morgan McGuire and Max McGuire. Steep parallax mapping. *I3D 2005 Poster*, April 2005. `http://www.cs.brown.edu/research/graphics/games/SteepParallax/index.html`.

[Ogra] Ogre 3D Instancing, Crowd Rendering. Website. `http://www.ogre3d.org/wiki/index.php/SoC2006_Instancing`, last visited Feb. 26, 2008.

[Ogrb] Ogre 3D. Website. `http://www.ogre3d.org`, last visited Feb. 14, 2008.

[OSG] OpenSceneGraph. Website. `http://www.openscenegraph.org`, last visited Feb. 14, 2008.

[Owe05] John Owens. Streaming architectures and technology trends. In *GPU Gems 2*. Addison-Wesley Professional, 2005.

[Por] Nick Porcino. The Four S's of Realtime Rendering. Website. `http://meshula.net/wordpress/?p=42`, last visited Feb. 14, 2008.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly Media, 2007.

[Ros06] R. J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, 2nd edition, 2006.

[SGI] std::for_each reference. Website. `http://www.sgi.com/tech/stl/for_each.html`, last visited March 23, 2008.

[SWND05] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison-Wesley Professional, 5th edition, 2005.

[TBB] Intel Threading Building Blocks (Intel TBB). Website. `http://www.threadingbuildingblocks.org`, last visited March 23, 2008.

[Tha] Benjamin Thaut. OpenGL instancing (english translation). Website. `http://blog.benjamin-thaut.de/?p=29#more-29`, last visited March 23, 2008.

[VBO03] Using vertex buffer objects. Whitepaper, 2003. `http://developer.nvidia.com/object/using_VBOs.html`.

[Zel04] Jeremy Zelsnack. GLSL pseudo-instancing. Technical Report, 2004. `http://developer.download.nvidia.com/SDK/9.5/Samples/samples.html`.