Technische Universität Wien
Vienna University of Technology

DIPLOMARBEIT

Penta G – A Game Engine for Real-Time Rendering Research

ausgeführt am Institut für

Computergraphik und Algorithmen

der Technischen Universität Wien

unter Anleitung von

Privatdoz. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

durch

Dey-Fuch Chiu

Schneckgasse 12
3100 St. Pölten

1. Mai 2008

Datum

Unterschrift

Abstract

Scientific research often necessitates the usage of middleware for proof-of-concept implementations. In computer graphics, rendering engines are a type of middleware used for such a purpose. For real-time rendering, however, rendering engines often do not provide all functionality that is required, as in real-time, a certain degree of user interactivity aside from graphics is necessary, or can be the center of research. Furthermore, some modern research also explores interactivity with non-visual perceptual channels such as audio. Full game engines provide that functionality, however most state-of-the-art game engines are only available through expensive or exclusive licenses, or are completely unavailable for scientific research.

This thesis presents the design, implementation and adaptation process of a state-of-the-art game engine that was tailored specifically for research purposes. This game engine includes all the necessary components to build real-time-interactive computer graphics applications and has been used in several research projects such as the EU GameTools Project and the EU Crossmod Project.

Kurzfassung

Wissenschaftliche Forschung erfordert häufig Middleware für die Implementation von Machbarkeitsdemonstrationen. In der Computergrafikforschung werden hierfür Rendering Engines hinzugezogen. In der Echtzeitgrafik ist allerdings oft mehr Funktionalität nötig als es Rendering Engines bieten können, da in Echtzeit typischerweise Benutzerinteraktion erforderlich ist, oder dies sogar Teil der Forschungsthematik sein kann. Weiters existieren moderne Forschungszweige, die auch die Interaktivität mit nicht-visuellen Wahrnehmungskanälen wie etwa Audio untersuchen. Vollwertige Game Engines verfügen über diese Funktionalität, allerdings sind die meisten Game Engines entweder nur über teure Lizenzen verfügbar, oder werden für wissenschaftliche Forschungszwecke überhaupt nicht angeboten.

In dieser Diplomarbeit wird der Entwurf, die Implementation sowie die Anpassung einer modernen Game Engine beschrieben, die spezifisch für die Echtzeitgrafikforschung entwickelt wurde. Diese Game Engine enthält alle nötigen Komponenten um echtzeit-interaktive Computergrafikprogramme zu entwickeln und wurde bisher schon in mehreren Forschungsprojekten verwendet wie etwa dem EU-Gametools-Projekt sowie dem EU-Crossmod-Projekt.

Contents

Chapter 1

Introduction

In this chapter the scope and focus of the work described in this thesis will be outlined, and the game engine that is described in this thesis will be introduced. Finally, we will give a short development history of the game that was created using that game engine.

1.1 Scope and Focus of the Work

This thesis describes the design, implementation and application of a game engine as well as a demo game utilizing that engine that was developed for research purposes. Numerous engines – both pure rendering engines as well as full frameworks for games – exist that are currently being used in research for creating demo programs and proof-of-concept implementations of techniques and algorithms.

The scope of the game engine described in this thesis was not to create yet another rendering engine, but to create both a full-featured game engine as well as an actual game, including the asset and workflow pipeline required to create content. While real-time graphics research often has access to a variety of graphics middleware, it is rare that current-generation game engines as well as actual games including their content as well as tools are available to researchers. Typically a commercial game engine and game becomes available to the public – and thus for research – years after that engine has become technically obsolete.

Therefore, the major focus of this work was to create also a full game that utilizes current-generation technologies, in order to provide research with not only a game and rendering engine, but also with all the necessary tools and the asset pipeline. This also enables scientists to work in a real-world-scenario environment when conducting research.

1.2 Introducing GebauzEngine

The underlying low-level framework created in the scope of this thesis was named GebauzEngine, or *GxEngine* for short. It provides access to rendering, filesystem, input and audio functionality. This thesis describes the second iteration of that engine (version 2.0). The first version – which was also already used for some research [26] – only acted as a prototype and had to be majorly refactored into version 2.0.

On top of that engine, a full game was built and various content tools created. The game built on top of that engine was named "Penta G". In the next section, a brief development history will be given.

1.3 Penta G Development History

Penta G and thus the GebauzEngine started as a demo game for the European Union GameTools Project (see Chapter 5.1). The goal was to create a state-of-the-art game and engine that would showcase some of the GameTools Project's technologies.

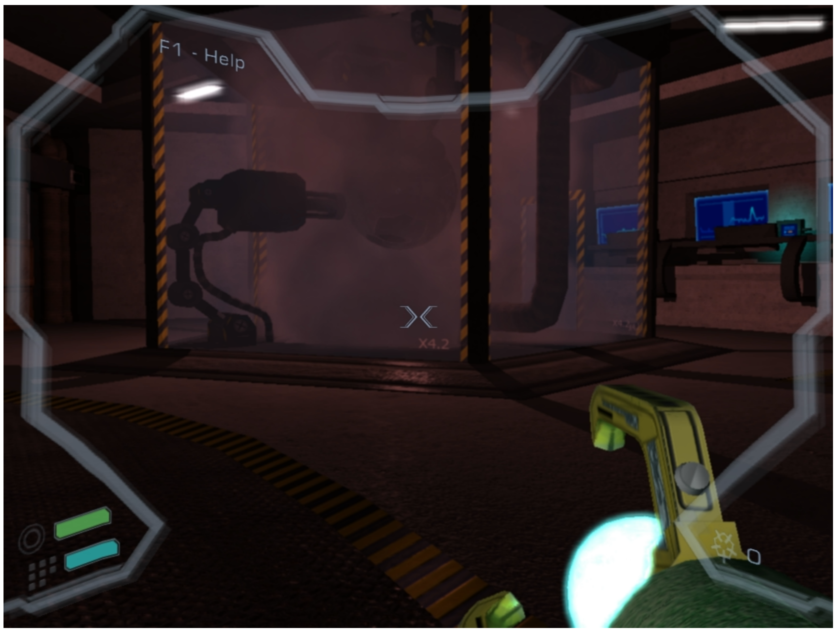


Fig. 1.1: GameTools demo game *Penta G*.

Penta G was completed in Summer 2007 after approximately a year of development with a total of three core team members. The game was showcased to various participating research institutes of the GameTools Project and was met with positive response. It also had exposure to game industry events and exhibitions as a representative demo of the GameTools Project.

When Penta G was nearing completion, researchers of the Crossmod Project (see Chapter 5.2) inquired about utilizing Penta G for their perceptual interaction research. Hitherto only videos of retail games had been used for eyetracking experiments. Having access to Penta G on a source code and source asset level was a benefit as experiments could be custom-tailored for the research focus. Another branch of the Crossmod Project used Penta G for their spatial audio perception research, effectively utilizing and modifying the audio subsystem of the GebauzEngine. Chapter 5 will describe how Penta G was used in research in more detail.

1.4 Structure of the Thesis

The thesis will be structured as follows:

- Chapter 2 introduces the forms of middleware commonly used in real-time computer graphics research as well as outlines the specific problems with each of them. A list of requirements a game engine must fulfill in order to solve these problems is subsequently proposed.
- Chapter 3 describes the design processes that went into the game engine in terms of software design, underlying graphics hardware and APIs, and the asset and workflow pipeline.
- Chapter 4 presents a demo application implemented using the game engine and the technologies used, and finally compares it to the requirements list from Chapter 2.5.
- Chapter 5 is devoted to the two research projects that have used Penta G for research: the GameTools Project and the Crossmod Project. A detailed description of how Penta G was used in each of them will be given.
- Chapter 6 summarizes the efforts made in this thesis and evaluates the result.
- The Appendix provides guides on how to create content for Penta G.

Chapter 2

Engines in Research

This chapter explores the reasons and benefits of using graphics middleware in the context of rapidly advancing graphics technology. Futhermore, an overview is given on the types of graphics and game middleware commonly used in real-time rendering research. After briefly giving exemplary usage scenarios of using game engines in research, we arrive at a requirements list for a game engine intended for research purposes.

2.1 Graphics Hardware and Real-Time Rendering

The following section gives an overview over real-time rendering and graphics application programming interfaces to analyze the necessity for graphics middleware.

2.1.1 What is "Real-Time"?

Rendering describes the process of generating an image of a scene from a different description of that scene. *Photorealistic* rendering typically researches techniques and algorithms that improve the realism of the images generated, often at the cost of rendering speed.

Real-time rendering, on the other hand, focuses on techniques that may also strive for realism and depiction of "physically correct" phenomena, but within real-time constraints.

Real-time systems are defined as systems that contain operations which have a time constraint. In other words, the correctness of the system behaviour does not depend purely on the logical results of the operations defined within the system, but also on the time in which these operations are carried out. These time contraints by which operations must yield results are called *deadlines* [37].

In real-time rendering systems, such a deadline is defined by the maximum time a single frame may take to render an image. The inverse of the frame render time is called the *frame rate* and is measured in *frames per second* (fps). Thus, the maximum frame time also defines a minimum frame rate. In general, a real-time rendering system can be considered *interactive* when it renders output at a minimum at 15 frames per second [62], or approximately 66 milliseconds per frame. Thus, the deadline for each frame of an interactive real-time rendering system is approximately 66 milliseconds measured from the start time of the rendering process of that frame. For truly fluid movement and animation, however, approximately 60 frames per second are necessary as a minimum, as framerates below that threshold result in stuttering movement. Therefore in the scope of this thesis a rendering system will be considered real-time when its output reaches 60 frames per second.

A real-time system may or may not meet its deadlines. In the case of not meeting its operational deadlines, three cases can be classified [37]:

- *Soft deadlines* are time constraints that, when not met, still yield utilizable results.
- *Firm deadlines* on the other hand describe deadlines where the results have no utility when the time constraint is not fulfilled.
- *Hard deadlines* result in a so-called *catastrophic event* in the case of failing to meet the deadline.

According to these definitions, real-time rendering systems can be classified as *soft real-time systems*, as even when the deadline for a single frame is not met, the rendering frame rate will be lower, but the output – the generated image – still has utility as long as it is rendered correctly.

2.1.2 The Rendering Pipeline

The traditional real-time rendering pipeline consists of three major stages [62]:

- The *application stage* holds higher-level information of the scene, and possesses the information necessary to break the scene data down to single render calls – these render calls are called *batches* and pass geometry such as vertex buffers down to the geometry stage.
- The *geometry stage* takes the geometry passed from the application stage, sets up triangles and transforms the vertices. It may also calculate per-vertex lighting information.

- Finally the *rasterizer stage* fills each triangle and calculates the final color value of each pixel by interpolating information from each vertex of a triangle.

Each of these stages can be further broken down into a pipeline of its own. The application stage is processed in software, on the main processor (CPU), while the others may or may not be executed in graphics-specific hardware. Section 2.1.3 will give an overview over such graphics processors and their functionality.

The Application Stage

The application stage contains the entire logic of the program, takes user input and manages the state of the program. It also prepares the state information it wants to visualize in a way that can be processed by the next stage, the geometry stage. How this stage is implemented in detail depends on the task and type of the program. Since this stage runs entirely in software, it is not necessary to structure this stage in the form of a pipeline with substages.

Typically this stage holds scene information, updates it according to the goal of the application (such as performing collision detection between scene objects), and processes geometry primitives such as triangles, points, lines into a form the geometry stage expects. It may also do calculations to reduce the amount of information sent to the next stage, effectively reducing the workload the geometry stage has to process. This can be performed by determining the visibility of geometric components of the scene – a process called *Visibility Culling*.

The Geometry Stage

The geometry stage takes the rendering primitives provided by the application stage and processes them so that the following stage, the rasterizer stage, may compute the final color of each pixel or fragment. In order to do that, it needs to transform the geometric data to view space, (optionally) perform lighting calculations, project the geometry to screen space, perform viewport clipping and finally map the result to individual pixels for the rasterizer stage. Figure 2.1 illustrates the substages of the geometry stage.

In a 3D rendering pipeline, several *coordinate systems* exist, determined by the position and orientation of each point in the system relative to the origin. Geometric data from meshes and models are saved in object-relative space, or *model space*. Such a model or mesh may be positioned and oriented arbitrarily in the world, or several instances of the same model may exist that

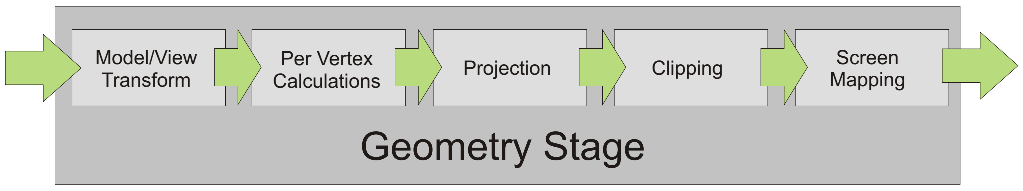


Fig. 2.1: The geometry stage.

all differ in positioning, orientation, or other arbitrary affine transformations. In this *world space*, each vertex of the model or mesh possesses a different position and orientation than in model space. The process of calculating the resulting world space vertex position from the vertex position stored in model space by using a specified transformation is called *transforming*. The transformation itself is typically specified by a 4x4 matrix, as such as matrix can store single transforms such as translation, rotation, scaling, shearing, as well as the result of a series of such transforms. Such a chaining is achieved by multiplying matrices together.

Aside from model and world space, the most commonly used coordinate frames are the camera-relative *view space* (sometimes called *eye space*), and the post-projection *2D screen space*. Other spaces exist and may be used depending on the specific algorithm such as *light space*. The exact orientation of the coordinate axis (x/y/z) of each coordinate system depends on the system that specifies it (such as an *application programming interface*).

In the geometry stage the *model and view transform stage* uses the model-to-world matrix and the world-to-view matrix to transform each vertex of the rendering primitives supplied by the application stage into view space. Typically these two matrices are concatenated by multiplying them together for efficiency reasons, so that transforming with a single matrix directly yields the view space coordinates [30].

Afterwards, per vertex lighting calculatings occur in the *lighting stage*. The application stage may specify various light sources to be active and determine their parameters. Vertex lighting calculations result in lighting values or color values based on lighting terms that are interpolated over a polygon, resulting in a shading scheme named *Gouraud Shading* [27]. Lighting may also be calculated in world space, depending on the application and system.

The *projection stage* transforms the vertices from view space into an unit cube called the canonical view volume. Depending on how the geometry is warped during this transform, we can identify two classes of projection transformations: *perspective projections* and *parallel projections*. Perspective Projections simulate the real-world effect of distant objects appearing smaller than close objects, while parallel projections – also called orthographic projections – preserve the parallel property of edges that were parallel before projections. By dropping the Z coordinate from the resulting vertices – which are stored in *normalized device coordinates* – the transformation from 3D to 2D space is achieved. As result we receive a two-dimensional collection of vertex positions that determine how the previously three-dimensional objects would look like in 2D space. The Z coordinate is not completely dropped, however, since it may play a role in per-pixel visibility calculations such as *Z testing* later on.

Clipping cuts away parts of the polygon objects that are outside the unit cube since those parts will not be visible in the final image. Finally, screen mapping – which can be considered another transform – computes screen device coordinates from the normalized device coordinates by mapping the normalized coordinates onto a viewport with a specific width and height.

The rasterizer stage then processes these polygons to generate the final image.

On modern graphics processing units (GPUs), the entire geometry stage is implemented in hardware, and made programmable through the use of *vertex shaders*. More complexity and flexibility has been added, so the description of this pipeline may not exactly match a specific implementation on a modern GPU.

The Rasterizer Stage

The *rasterizer stage* is responsible for processing data input from the geometry stage in order to output resulting pixel values. Depending on the system, such pre-display pixels may be called *fragments* such as in the *OpenGL* application programming interface (see Section 2.1.4). Figure 2.2 shows a representative structure of the rasterizer stage.

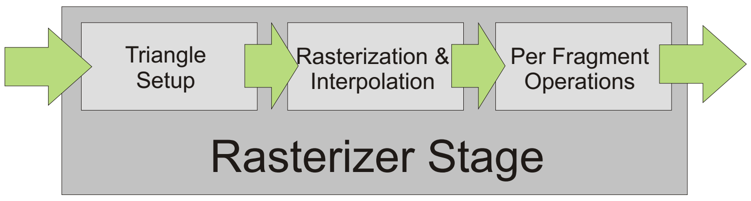


Fig. 2.2: The rasterizer stage.

From the processed primitives of the geometry stage, the rasterizer stage sets up triangles and interpolates per-vertex values across their surface, calculating the fragments that make up the single triangles. For each fragment per-pixel operations are executed, which take the interpolated per-vertex values to calculate the output color.

Aside from information interpolated from vertex data, the rasterizer stage may also use additional data to combine the final pixel from. One type of such input data may be texture data, and the process of reading data from a texture is called *texture sampling* and the resulting read-out data is called a *texel*. Texture images can be used to easily increase the optical fidelity of the resulting image by adding fine details that are not feasible to be added geometrically.

When using *per-pixel lighting*, lighting is not performed in the geometry stage, but in the rasterizer stage. The geometry stage merely provides the necessary interpolated input data to the rasterizer stage, such as the per-pixel normals interpolated from the polygon vertex normals. The lighting term itself is then calculated in the rasterizer stage, resulting in an interpolation or shading scheme called *Phong Shading*. Using additional data such as textures containing normals, it is possible to add fine lighting detail onto a surface. Such textures encode a normal's x, y and z coordinate into the red, green and blue channels of a color texture, called normal maps, and the variation of per-pixel lighting that does not take the interpolated normal from the geometry stage but rather the normal sampled from a normal map is called *Normal Mapping*.

Finally, additional raster operations can be executed on the final color output such as blending. This stage depends on the actual implementation of the rendering pipeline, and may actually happen in the fragment processing stage, or not exist at all on certain systems.

Similar to the geometry stage, the Rasterizer stage is also implemented completely in hardware on modern GPUs. Programmable *pixel shaders* make it possible to develop complex lighting schemes not hardwired to the capabilities of the graphics chip.

The next section explores the gradual development in terms of hardware acceleration of the various stages of the rendering pipeline and gives a brief history on consumer graphics hardware.

2.1.3 Advancements of Graphics Technology

A large role in the progress made in Real-time rendering is played by the hardware technologies developed in the past 10 years. With the introduction of the 3dfx Voodoo 3D accelerator [2], real-time rendering hardware – which previously existed purely in the professional domain – became affordable to consumer audiences.

Early 3D accelerators only realized the rasterization stage in hardware; the application and geometry stages still had to be processed in software. However, with the advent of *Hardware Transform-and-Lighting* in the form of the NVIDIA GeForce 256 graphics card [39], parts of the geometry stage were transferred to hardware as well.

Due to the hardwired implementation of these stages in hardware, the hardware part of the rendering pipeline could only be processed in a very rigid way, with only limited flags or registers that could influence the path through the rendering pipeline. Such a hardware-accelerated pipeline that only had a rigid path through it with very limited control over the process is called *Fixed-Function Pipeline* [62].

Due to the rigid nature of the fixed-function pipeline, hardware designers added various methods of controlling the flow through the rendering pipeline in order to add more flexibility on how vertices and fragments were processed. *Register Combiners* added a form of semi-programmable logic to the rasterizer stage, making it possible to combine samples from multiple texture stages using a range of selectable operators [40].

Graphics accelerators gained full programmable capabilities with the development of *vertex and fragment programs*, which respectively made it possible to programmatically influence the transform-and-lighting stage and the rasterization stage by running small programs called *shaders* directly on the graphics chip. Note that *vertex program* and *fragment program* are OpenGL terminology; in Microsoft Direct3D, these programs are named *vertex shaders* and *pixel shaders*. More details on OpenGL and Direct3D will be given in Chapter 2.1.4.

Early pixel shader implementations on graphics chips hardly had more functionality than the fixed-function register combiners – they could run pixel shader programs containing a maximum of 64 instructions. Also, programming shaders was mostly done in an assembly-like language.

With every new generation of graphics hardware, however, limitations were made less restrictive, and new capabilities were added to the previously very limited shader programs, such as the introduction of Dynamic Branching in Shader Model 3.0 [19], and the introduction of *high-level shader languages* which previously were a feature exclusive to software rendering systems, such as in the form of the *RenderMan Shading Language* [50].

With Shader Model 4.0, which was introduced with the Microsoft Direct3D 10 API, graphics cards also gained the ability to tesselate, alter and generate geometry after the vertex shader stage; for this a new type of shader called the *geometry shader* was added.

Graphics hardware has not only influenced the according graphics application programming interfaces (APIs), but APIs also influenced graphics hardware. Recent consumer hardware is often tailor-designed to Microsoft's DirectX specification. The next section will introduce the most important graphics APIs in this context.

2.1.4 Application Programming Interfaces (APIs) for Graphics

As more graphics accelerator hardware was introduced and made available to consumer audiences, due to the multitude of different graphics processors, it was necessary to define standards for accessing and programming such hardware. With the introduction of affordable consumer 3D accelerators, new such *application programming interfaces* (APIs) were developed or existing ones such as OpenGL were adapted and extended to support such hardware. This section will give an overview on some important graphics APIs that were relevant to such consumer hardware.

OpenGL

OpenGL, or *Open Graphics Library*, is a set of specifications for a 3D application programming interface originally developed by Silicon Graphics (SGI) [56] as a successor to their previous graphics programming interface, *IrisGL* [30]. It is designed to be platform-independent, and several implementations on various operating systems in software as well as hardware-accelerated implementations for graphics hardware exist.

The OpenGL standard was previously maintained by the *OpenGL Architecture Review Board* (ARB), an independent consortium founded in 1992 [45] by SGI. In 2006, the consortium voted to transfer the control over the OpenGL standard to the *Khronos Group* [36].

The core OpenGL API provides programming access to the geometry and rasterizer stages of the rendering pipeline outlined in Section 2.1.2. OpenGL is implemented as a state machine, which means that all rendering parameters such as actively bound textures, vertex buffers, shaders, as well as render states preserve their state until it is changed by the programmer [57]. Depending on the concrete implementation, sometimes called the *OpenGL device driver*, certain features may be hardware-accelerated or software-based, or both.

While the core API is platform-independent, OpenGL possesses a platform-dependent layer that is used to bind a windowing system such as Microsoft Windows or the X Window System to an OpenGL *rendering context*. Depending on the vendor of the OpenGL implementation, different extensions may be supported that give access to extended functionality not available in the OpenGL core API. These extensions are specified in a similar manner to the core API in the SGI extension registry, formulated to be addenda to the OpenGL reference manual [55]. With every version of the OpenGL specification, some extensions become part of the core API. At the time of writing, the current version of the OpenGL standard was 2.1, and drafts for version 3.0 have been introduced by the Khronos Group.

Several different types of extensions are specified, denoted by a certain acronym:

- ARB extensions have been declared standard by the OpenGL architecture review board, and may be part of the core in subsequent versions of the OpenGL standard.
- EXT extensions are supported by at least two vendors, but have not been standardized by the OpenGL architecture review board yet.
- Vendor specific extension types such as NV, ATI, SGIS, APPLE.

The application programmer can query the extensions supported by a specific OpenGL implementation, and can thus determine whether the application will run on the system. In more complex applications, the programmer may implement multiple *render paths* that implement different rendering methods depending on the set of extensions supported.

OpenGL also supports programmable shaders, called *vertex programs* and *fragment programs* respectively. Initially only programmable in shader assembly language, version 2.0 of the OpenGL standard has also introduced a high-level shader language, *GLSL*. GLSL is similar to the syntax of the C programming language [35] and its parser is implemented as part of a specific OpenGL implementation; compatibility with the OpenGL standard or with other OpenGL implementations may therefore vary depending on the quality of the OpenGL implementation.

Glide

Glide is an API introduced with the 3Dfx Voodoo range of graphics cards in 1995 [1]. It provided a simple interface to the rasterizer acceleration features of the Voodoo graphics chips range. It has lost its relevance in real-time graphics by now as it is not maintained any further and its inventing company, 3Dfx Graphics, Inc., no longer exists.

DirectX

Microsoft DirectX is a collection of application programming interfaces intended for game development [19]. The DirectX API was introduced by Microsoft in 1995 under the name “Game SDK”. During its history, several components have been added to the DirectX API, as well as some removed or declared deprecated. The API is based on the Component Object Model (COM) and as such can be accessed via interface classes in an object-oriented manner.

In its history, DirectX has contained the following components:

- *DirectDraw* is an API that provided access to the framebuffer and 2D acceleration features such as hardware blitting and buffer flipping. It has been declared deprecated on Desktop Windows operating systems and is only maintained for the Windows Mobile platform.
- *DirectSound* is an audio rendering API that allowed access to audio acceleration and spatial audio features of soundcards. On newer Windows operating systems such as Windows Vista it is considered deprecated and is only supported via a software layer. The XAudio2 API is intended to replace DirectSound in March 2008.
- *DirectInput* provides access to input controllers consisting of keyboards, mice, and game controllers such as gamepads and joysticks.
- *DirectPlay* is a deprecated networking API that provided a unified layer for accessing networks of various protocols such as TCP/IP and IPX. Due to the ubiquity of TCP/IP and UDP on modern operating systems, DirectPlay is no longer necessary.
- *Direct3D* was developed as an API to access 3D acceleration features of graphics cards.
- *DirectShow* is a video playback API that has been removed from the DirectX SDK and is now part of the Windows Platform SDK.
- *DirectMusic* is an API built on top of DirectSound and provided means to provide interactive dynamic music in interactive applications.
- *XACT* or *Crossplatform Audio Creation Tool* is an audio content authoring toolset and API for both DirectSound/DirectMusic and the XAudio API of the Microsoft Xbox360 game console.

- *XInput* is an input controller API supporting only the Microsoft Xbox360 controller on both Windows and Xbox360. It is intended to supersede the DirectInput component.
- *XAudio2* is the sucessor of both DirectSound and the Xbox360's XAudio API. It is implemented entirely in software and therefore does not utilize any sound acceleration hardware except on the Xbox360 platform where it uses the built-in XMA compressed audio decoding hardware.

Of these components, Direct3D is the API controlling 3D rendering hardware. In contrast to OpenGL, there is only one Direct3D API implementation, which transforms API calls to the underlying device via a driver layer. A standard Direct3D installation also includes a *reference rasterizer*, which implements the rendering device as a software renderer.

The existance of hardware-accelerated features is queried via a capabilities system, where the application can get detailed information about a certain rendering device, and decide which rendering features and algorithms to use depending on these capability flags.

Initially, Direct3D offered two modes of operation: *Immediate Mode* and *Retained Mode*. Immediate Mode was the mode that closely resembled other 3D APIs. The programmer had to implement a rendering queue by sending batches of rendering primitives to Direct3D. Retained Mode, on the other hand, provided a higher level of abstraction and offered an interface more closely resembling a *Scenegraph API*.

In DirectX 7, the retained mode was dropped, and support for hardware-accelerated transform and lighting (hardware TnL) was added. The term *"DirectX 7 compatible"* became synonymous to hardware supporting this feature.

DirectX 8 introduced programmable shaders in the form of Shader Model 1.x. Capabilities of this shader model were still limited and offered little more than the hardwired register combiners that were available at that time. With the DirectX 8 SDK, a new toolkit was also added: the *Direct3D Extension Library* (D3DX). D3DX offered a collection of useful utility functions 3D programmers may find useful such as simple ways to create textures from many image formats.

With DirectX 9, Shader Model 2.0 was introduced that increased a lot of the capabilities of shader programming, and the D3DX library vastly extended to feature support for skinned animation, precomputed radiance transfer calculations, mesh optimization, and more. D3DX also introduced a high-level language for shader programming called *HLSL* (for "High-Level

Shader Language"). During updates to the DirectX SDK, which happened on a bimonthly basis, only the D3DX library was updated with added features.

DirectX 9 also saw an incremental update, DirectX 9.0c, which introduced Shader Model 3.0. Shader Model 3.0 allowed for more complex shader programs with features such as dynamic branching and looping, vertex texture fetch, and geometry instancing.

Direct3D 10

Direct3D 10 – sometimes also called *DirectX 10* – has originally been developed separately from the DirectX API collection, under the name *Windows Graphics Foundation 2.0*, or WGF 2.0 [6]. It is currently only available for Microsoft's Windows Vista operating system.

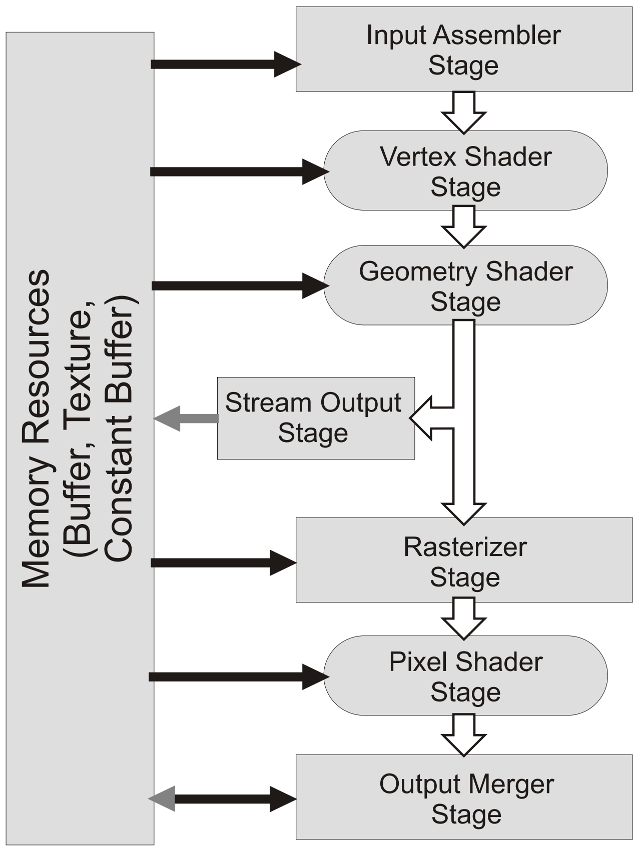


Fig. 2.3: Direct3D 10 Rendering Pipeline.

The API deviates from previous Direct3D incarnations such as Direct3D 8 and 9 by redefining the rendering pipeline in several ways [58] (see Figure 2.3):

- *Unified Shaders* replace the separation between vertex and pixel shaders in a way that, independent of the stage the shader program was used in, the same instruction set would be made available. Every shader stage is based on the *Common Shader Core*.
- *Geometry Shaders* were introduced as a new type of shader stage that has access to entire primitives including adjacency information, and the capability to amplify and de-amplify data.
- *General Purpose Streams* replaced the previous notions of differentiating between textures (and the lookup thereof), vertex attribute streams, and other data. It is now also possible to output pipeline-generated vertex data into memory using the *stream output stage*.

Due to the API changing the traditional rendering pipeline drastically in some ways, the API influenced the hardware design of graphics hardware that were intended to support the Direct3D 10 API, making Direct3D 10 an example of a software API influencing hardware design, whereas in the past the process usually went the opposite way.

2.1.5 Arriving at Middleware – Rendering Engines

Application programming interfaces exposed the functionality of the hardware in a way where programmers would no longer have to worry about specifics of the hardware. Despite that, most APIs provide only basic support for hardware functionality, while application programmers usually also require some higher level facilities such as resource management.

As a measure against reinventing the wheel every time a new application is programmed, packages of commonly used routines and facilities were developed that simplify usage scenarios such as loading geometry and textures, help with managing and cleaning up resources, and provide convenience functions for complex and frequently used processes.

Such packages of useful functionality can be classified as a *rendering engine*. Several different kinds of rendering engines exist, with varying functionality. The next section will attempt to give an overview and try to define and classify types of such middleware.

2.2 Types of Middleware

As mentioned in the previous section, graphics APIs provide access to low level hardware functions, and functionality can be added on top of that layer by utilizing packages or libraries called rendering engines.

This section will outline and try to define several classes of such libraries or frameworks, and give popular examples for each of them. Wherever it is not possible to clearly put a certain framework library into a specific class of engine, it will be noted.

2.2.1 Rendering Engines

Rendering engines provide a means to access graphics rendering capabilities without having to access the underlying graphics API directly. They add another software layer to the application stage of the rendering pipeline. Rendering engines try to simplify programming a graphics API, as well as take over management duties such as resource management – for textures, geometry, scene data [4].

In the context of this thesis a rendering engine will be considered anything that provides a software layer to an underlying graphics API. As such, the types of frameworks and libraries described in the following sections are *subtypes* of rendering engines.

2.2.2 Scenegraph Libraries

Scenegraph libraries are a form of rendering engine that focus on the representation of the scene in the form of graphs – collections of nodes and edges that connect such nodes. A scene is stored as a graph where every node is a certain scene object, and the edges define the relationship between the nodes.

Depending on the scenegraph library, nodes can be anything from geometry to materials to more abstract entities. It is important to note that in most scenegraph libraries, the scenegraph displays a *logical view* of the scene in contrast to the actual rendering logic or rendering order. Scenegraph libraries may have a rendering queue into which *renderable objects* are pushed and possibly resorted to improve the rendering efficiency.

An important feature of scenegraph libraries is the variety of operations that can be defined via a *traversal scheme*. Depending on which task is to be performed on the scenegraph, a different traversal algorithm may be necessary. This also includes the traversal necessary to determine which nodes to render. Typically this is achieved by a form of the *visitor* design pattern.

2.2.3 Game Engines

Game engines can be considered a sub type of rendering engine, but may also be considered of a framework that contains a rendering engine as one of its components. A Game engine extends the functionality of a Rendering Engine by several components that are necessary and useful for interactive applications such as games or virtual world walkthroughs.

Interactivity and User Input

Game engines need facilities to query and interpret user input. Note that while some rendering engines may also exhibit this functionality, many programs created in real-time rendering research simply use a platform API such as the Windows API or a platform-independent API – such as GLUT – for user input query.

In a game engine, user input has to be polled and queried, and mapped accordingly to actions inside the game. Several different input controller types need to be supported.

Interaction Logic

In a rendering engine, the main part of the program is the rendering loop. In contrast, in a game engine, the rendering code has to coexist with the game code, which defines the game or interaction logic. The pseudocode for a main loop in a game engine may look like in Listing 2.1.

```
void Run()
{
    // As long as the application is running...
    while (running)
    {
        // Update and manage game logic
        Update(elapsed_time);

        // Visualize state by rendering it to screen
        Render();
    }
}
```

Listing 2.1: Standard Gameloop

The `Render()` function displays the scene in its current state, while the `Update()` function controls and updates the game state, as well as interprets user input and applies it to the internal state. There is a clear separation between rendering and game logic; the `Render()` function should not change the game's state at all, and therefore make it possible to call `Render()` multiple times without any problem, effectively making it possible to pause any game processing by not calling `Update()`. Therefore, the `Render()` function does not require a parameter specifying the elapsed time since the last frame, in contrast to the `Update()` function, which on the other hand should not utilize rendering or display functionality.

Audio

An important component of many interactive applications such as games is audio processing. Handling and initialization of audio devices should be made simple, and the game engine typically needs facilities to link audio output to the scene that is displayed – that is, positioning audio sources in 3D space.

2.3 Examples of Middleware in Research

This section gives brief overviews on some selected graphics middleware packages.

2.3.1 OGRE 3D

The *Object-Oriented Graphics Rendering Engine* (OGRE) is an open-source rendering engine supporting output using both OpenGL as well as Direct3D [42]. It abstracts these APIs by only providing direct access to higher level scene objects, hiding the details of actual rendering from the developer. OGRE 3D provides its own effect and material framework similar to the HLSL Effect Framework and CgFX, allowing for multi-platform and multi-API shader programming. It does not provide a pre-defined scenegraph structure, but provides a scene node class and a plugin system that allows the programmer to implement an own hierarchical scene or spatial structure.

Ogre 3D is deliberately not a game engine. Several add-on libraries exist that add game engine-like functionality to Ogre. OGRE 3D itself is licensed under the GNU Lesser General Public License (LGPL).

2.3.2 OpenInventor

OpenInventor is an object-oriented scenegraph toolkit by SGI developed in 1988 under the name *IRIS Inventor* [44]. It was developed with the intent to simplify OpenGL programming, and therefore is tailored for and running on OpenGL. The scenegraph library provides a multitude of different node classes and node types for geometry, transform groups, appearances and materials. OpenInventor passes render states on when traversing nodes (*path inheritance*), therefore the rendering of a single scene entity can be spread out over the entire scenegraph, and nodes located deeper in the hierarchy may therefore influence nodes locatged higher in the hierarchy.

OpenInventor and API-compatible OpenSource clones such as *Coin3D* [14] have been widely used in real-time rendering and related research such as virtual and augmented reality [52].

2.3.3 Java3D

Java3D is a scene graph framework for Sun's Java platform [32], enabling it to run across multiple platforms supporting Java. On Windows it supports both OpenGL and Direct3D, and therefore utilizes hardware-accelerated rendering if available. The scenegraph classes offer a variety of node types similar to OpenInventor. In contrast to OpenInventor's path inheritance, Java3D is designed to keep all relevant appearance and geometry information at the respective nodes, leading to a more modern software architecture and design.

2.3.4 NVIDIA SceneGraph

NVIDIA offers a scenegraph library in the form of the *NVIDIA SceneGraph* SDK (NVSG) [41]. It runs on OpenGL and like other scenegraph libraries, offers a multitude of node types that allow structuring the scene. Its main specialty compared to other scenegraph libraries is that NVSG has been designed around shaders in the form of the CgFX framework [12, 64].

2.3.5 OpenSceneGraph

OpenSceneGraph is another C++-based OpenGL-based scenegraph library running on multiple platforms such as Windows, Linux, Mac OS X, FreeBSD, IRIX and Solaris. It is licensed under the OpenSceneGraph Public License (OSGPL), which is based on the GNU Lesser General Public License (LGPL) [46].

2.3.6 OpenSG

OpenSG is also an OpenGL-based C++ framework focusing on multithreaded rendering and clustering. It is licensed under the GNU Lesser General Public

License, and also runs on multiple platforms (Windows, Linux, Mac OS X and Solaris) [47].

2.3.7 Performer

Performer, also known as *OpenGL Performer* or previously *Iris Performer*, is a commercial toolkit based on OpenGL created by SGI as a alternative to OpenInventor focusing on performance [48]. It is available for Windows, Irix and Linux.

2.3.8 YARE and YARE2

YARE and YARE 2 (*Yet Another Rendering Engine*) [4] were custom-built rendering engines developed in the environment of the Vienna University of Technology. The first YARE was developed as a C++ based implementation of the Java3D API for the *Urban Visualization* project [65] at the Institute for Computer Graphics and Algorithms. YARE 2 has been developed by Matthias Bauchinger for his master thesis [3] and focuses on a flexible effect framework built on top of a multi-layered rendering framework. Except for its name, it is otherwise unrelated to YARE.

2.4 Game Engines in Research – Usage Scenarios

This section will outline some common scenarios in which game engines are useful for research purposes.

2.4.1 Research Test Beds

Game engines may be used for proof-of-concept implementations of new real-time rendering algorithms and techniques. They provide a practical test bed for evaluation of these new algorithms which may provide useful insight into the efficiency and scalability of the researched techniques, as well as possible bottlenecks and problems. Results from such testing in an environment close to industry practice can then be further used to modify and tweak the algorithm to better suit and optimize the technique's usefulness.

Previously, most of such work was done in specifically programmed demo applications or utilizing rendering engine packages. The disadvantage of the former lies in the rather limited scope of the demo in terms of both software and content complexity, which may distort important results when evaluating the new algorithm, while the latter approach has the disadvantage that most such packages are pure rendering engines lacking features as described in Section 2.2.3.

Ideally, such evaluations and analysis of algorithms therefore should be done in an environment that is close to the target audience of these real-time rendering algorithms.

2.4.2 Paper Demos

For submissions of research papers, oftentimes a demo, or a video and screenshots of it, may be required. By utilizing a real game engine for such tasks, the authors immediately possess evidence that their researched techniques do work in practice.

2.5 A Requirements List for Research Game Engines

In this section, we derive a requirements and specification list for game engines intended for research by analyzing the conclusions of the previous sections.

2.5.1 Layered Design

A software design that defines components in a layer-like manner enables the developer to choose at which abstraction level to integrate changes or which level to build on when developing a new application running on the engine. This flexibility of choice is necessary in research environments, since research algorithms and techniques may sit on different levels of abstraction; some algorithms may need access to lower-level components, others may require to be implemented on top of a full framework.

As such, the layered design also needs to provide a way to circumvent certain components and replace them with own implementations.

2.5.2 Extensible Rendering System

Most rendering engines provide a pre-defined rendering pipeline, into which the application must fit their needs. Such rendering pipelines may be implemented as a rendering queue, into which the application may push renderable objects, but access to which is restricted most of the time. Details on how the rendering pipeline is processed are often part of the internals of a rendering engine. While a pre-defined rendering scheme simplifies actual application programming, research development often requires heavy modification or access to the rendering pipeline.

2.5.3 Reducing Overengineering

Overengineering can be defined as an attribute of a software design that increases the complexity of the system faster than the benefits of such complexity. Object-oriented design, as well as design patterns, are often tools and paradigms to simplify or reduce redundant client code, make software components more reusable, or reduce code amounts. However, these processes also increase the risk of high inter-dependencies in code. Such dependencies may actually make the code less reusable: an example would be a class that was designed to be slim by deriving it from interface base classes, but therefore introducing a dependency on those interface base classes. Due to that dependency, when reusing the class, all interface classes must be migrated too.

Such high inter-dependencies may also introduce situations where a replacement of a framework component by an application part may prove impossible or difficult, as the application programmer would need to integrate the own component into a large system of inter-dependent classes. Such situations where the application developer may want to not use a framework component at all, and instead replace it with an own implementation, are less prevailent for regular application developers, but may happen frequently in research.

Therefore, when designing a framework for research with a low amount of overengineering, there must be a focus on preventing dependencies.

2.5.4 Comprehensive Toolset

Demo programs and proof-of-concept implementations that accompany real-time rendering research often use standard data sets to create rendering scenes. Such standard data sets may be geometric primitives such as toruses, cornell boxes, or more complex standard data sets such as the Stanford Bunny.

Such scenes typically do not reflect real-world application data set behaviour and performance, such as games. In order to easily bring custom data sets such as geometric models and meshes, materials, effects, textures and full scenes into the engine, a comprehensive toolset is necessary that enables the user to transfer closer-to-real-world assets from common content creation applications such as *Autodesk Maya* or *3D Studio Max* into their own applications.

2.5.5 What is not required

Since computer graphics algorithms, and especially real-time rendering algorithms, may be placed on various different levels of hardware abstraction, access to the rendering API may be necessary, as argued in Section 2.5.1. This also means that fully abstracting and hiding API specifics is not a necessity, and could actually be a hindrance when access to low-level API specifics is required. The requirement outlined in Section 2.5.2 also necessitates the replaceability of lower level components that may use the rendering API directly.

Chapter 3

Designing a Game Engine for Research

In this chapter the design decisions and processes that led to the structure of the game engine discussed in this thesis will be introduced.

3.1 Introduction

First we describe the high-level decisions for the design approach in this introductory section.

3.1.1 Approach

The GebauzEngine was not developed on its own, but as part of the Penta G demo game project intended to showcase GameTools Project research results in a real-world scenario, i.e. games (see Section 5.1).

Therefore, during the entire development of the game engine, both the game and its engine have been developed in parallel, to ensure that features that are being added are close to the requirements of the actual game. A *content-driven approach* to engine development was chosen to ensure that features in the engine would stay close to the needs of the content of the game.

A strong focus was given to separating application-specific code from framework code. Since the domain of the game engine extends to both components, this chapter will be a discussion of both. While the access to API functionality is handled by the GebauzEngine, and the GebauzEngine specifies some higher level facilities such as resource management, content handling is done almost purely in the application domain. This was made to ensure flexibility, as a too rigid form of content resource handling, i.e. one implemented on framework level, could hinder development when deviations of the defined content handling are required.

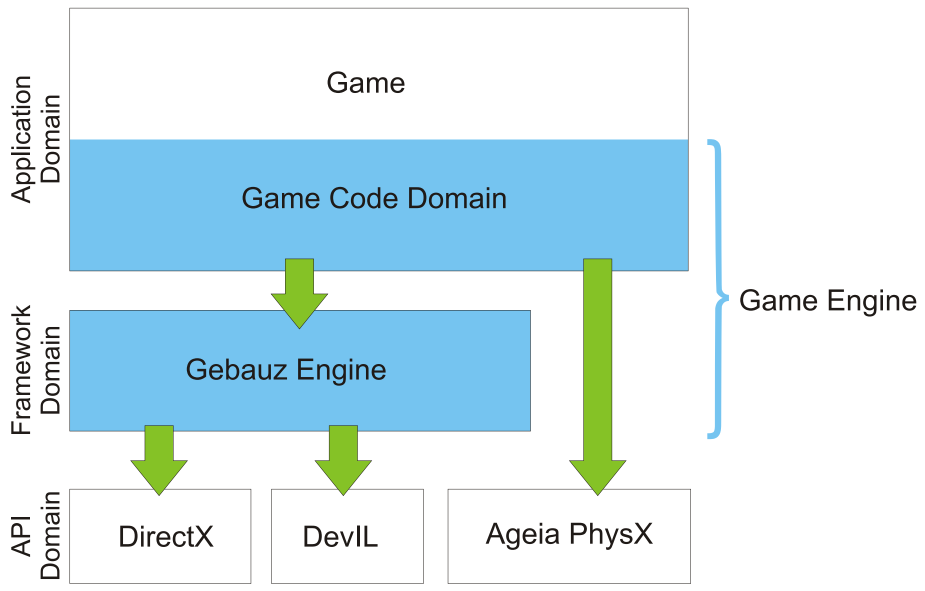


Fig. 3.1: Engine Layer structure.

Figure 3.1 illustrates the layered design of the engine. As illustrated, the term *game engine* refers to the GebauzEngine framework as well as parts of the actual application code.

3.1.2 Content-Driven Development

This content-driven approach resulted in the development workflow we used for the engine and its toolset. Features that the engine required from the tools would be incorporated into the tools, and vice versa. Together with the requirement of avoiding overengineering (see Section 2.5), this resulted in a toolset that is slim and not overly complex.

3.1.3 Asset Tool Chain

To ensure a content-driven approach, tools needed to be developed. The tools aid in importing geometric assets from popular 3D content creation applications, as well as building levels for the game. This is a key point of the engine as due to the comprehensive editor, new content can be added quickly. On the application side, most of the content handling is done on the application level, as it should be possible to flexibly change parts of the content pipeline such as the format of the material script system described in Section 3.5.3. Therefore, when discussing the asset pipeline and tool chain, most of the information relates to application code implemented in Penta G.

3.2 Software Design

This section describes the technical details of the software design.

3.2.1 Overview

The game engine and also the game built on top of it use a layered design. The GebauzEngine provides basic facilities for access to files, rendering, input and audio. The game itself has to implement its own game object hierarchy and game logic. Additionally, an application framework (AppKit) has been developed that simplifies the creation of a Windows-based application, which runs as a separate library and thus can be replaced with other windowing and graphical user interface (GUI) libraries such as wxWidgets or MFC.

This section describes the structure and design of the GebauzEngine. For a detailed description of the Penta G game built on top of the GebauzEngine, see Chapter 4.

3.2.2 Utilized Libraries and APIs

The GebauzEngine and Penta G use several underlying application programming interfaces, libraries and frameworks. Most of them are integrated at GebauzEngine level, whereever it is not, it will be specifically noted. In that case, the integration was on application level, i.e. in Penta G, making it possible to swap the specific library with an alternative one.

DirectX 9.0c

For rendering, the *Direct3D 9* API [19] was chosen, due to its dominance in the game industry. As a demo game intended for industry audiences, it was a logical choice. Due to some of the techniques and algorithms used in Penta G, *Shader Model 3.0* of the Direct3D 9 API was used, enabling shader capabilities such as longer shader programs and dynamic branching.

DirectInput was used to control input devices; however, Penta G only uses keyboard and mouse input. General game controller input such as gamepads and joysticks is available though, and could be easily integrated into the actual game or any other game built upon GebauzEngine.

Audio output is realized via the DirectSound API, but the design of the audio subsystem's interface is general enough to replace DirectSound with a different sound API such as OpenAL or XAudio2. Section 5.2 introduces a branch of the Crossmod project that replaced the GebauzEngine's audio subsystem with a propietary one (XModSoundLib).

Aside from Direct3D, DirectInput and DirectSound, the GebauzEngine also makes heavy use of the *Direct3D Extension library* (D3DX), which is a library containing useful functionality on top of Direct3D. This library also contains the DirectX Effect framework and HLSL compiler, simplifying shader development. However, linking to D3DX introduces a dependency to the D3DX DLL supplied with each DirectX Software Development Kit – typically named d3dx_<*number*>.dll, where <*number*> is the version number that gets incremented with every DirectX SDK iteration. For deployment, this usually means that the D3DX DLL must be present on the target system; since providing the DLL directly in deployments is not legally allowed by Microsoft, this means installation of the (current) DirectX Redistributable [19] which installs all D3DX DLLs in the system.

The most important feature of the D3DX library that is used in Penta G is the Effect Framework, which provides shader management features built on top of the Direct3D High Level Shader Language (HLSL) that simplify development of shaders and interoperability between different applications supporting the Effect Framework [59].

Boost C++ Libraries

Penta G uses a minimal subset of the Boost C++ Libraries [7], specifically the *static assert* feature which allows for compile-time assertions. This is utilized specifically in the vertex buffer templates used by the Geometry part in the Graphics subsystem described in Section 3.4.3.

DevIL

The *Developer's Image Library* is a comprehensive image library that allows reading and writing of most image file formats [31]. The GxImage class uses this library to read in image files that may or may not be used for textures.

Unfortunately, DevIL appears to be out of maintenance, therefore in a future version of the GebauzEngine, the usage of a different image library (such as FreeImage) may be recommended.

Ogg Vorbis

For decoding of Ogg files, the engine uses the *Ogg Vorbis* library. This is used in the Audio subsystem described in 3.4.5. Vorbis is an open source and open standard codec for lossy audio compression similar to *MPEG Layer 3*, or MP3 [22]. Ogg is the accompanying multimedia container format. The Game Engine uses Ogg Vorbis-encoded files for streaming audio, but it is also capable of using standard Windows Wave files.

Ageia PhysX

The physics simulation library *Ageia PhysX SDK* [49] was used for the game. Note that PhysX is not integrated into the engine on the GebauzEngine level, but on application code level (Penta G), making it possible for GebauzEngine to be used with a different physics library such as Bullet [9]

Ageia PhysX supports full software physics simulation, but also enables support for hardware-accelerated physics calculations. Its license is free to use for non-commercial purposes.

For deployment, the PhysX driver has to be installed in the system in order for the application to run.

3.2.3 Collaboration Structure

Figure 3.2 illustrates the collaboration between the various subsystems of the GebauzEngine.

3.3 Development Tools and Standards

In this section, the standards and tools used for development are described.

3.3.1 Development Environment

The engine and game can be compiled in Microsoft Visual Studio 2003, 2005 and 2008. It was initially developed in Visual Studio 2003, and then migrated to Visual Studio 2005. In Visual Studio 2005, deprecation warnings will appear during compilation because 2005 includes an updated C and C++ Standard Library with safe replacements for standard C functions such as `printf()` – which now has been replaced by the safer `printf_s()` function.

Some compiler-specific features such as `#pragma once`, a directive that prevents multiple inclusion of header files, were used when it made sense for

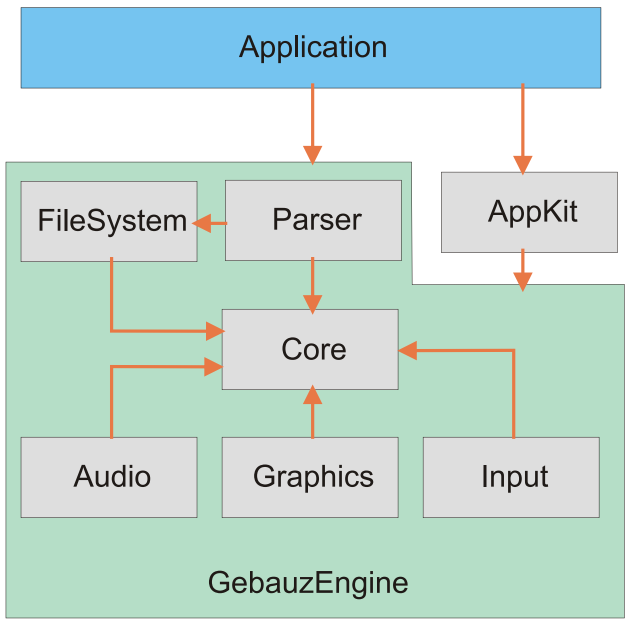


Fig. 3.2: Engine Subsystem Collaboration.

convenience. Since the engine is not designed to be portable, as it already has a Microsoft Windows dependency by utilizing DirectX, this does not pose a big problem.

3.3.2 Directory Structure

The directory structure specified in table 3.1 was used throughout the project.

Directory	**Content**
`bin`	Executables, DLLs and configuration files
`doc`	Documentation and doxygen generation batchfile
`include`	Engine include files
`LatestPlayable`	Contains a recent playable executable build
`lib`	Engine library files
`media`	Game content files
`sdk`	Libraries the game engine depends on
`src`	Source files for the engine and game
`tools`	Content creation tools

Tab. 3.1: Directory Structure

There is a separation between include and implementation files for the engine so that the engine may be distributed in binary-only form of only a static library and include files.

3.3.3 Debugging and Memory Leak Tracking

Memory leaks are caused by unfreed memory during the run of a program. They can be an indication of bugs or possible problems in the program, as well as a direct threat for the stability of the program when more memory is frequently consumed without being freed, resulting in less memory being available.

A common way to fight memory leaks is to track allocated memory and check whether each allocated memory block is freed at the end of the program. For such a purpose, the Microsoft Visual C++ Debug Heap offers helpful functions to track and log memory allocation, and to report whether allocated memory has already been freed.

Penta G heavily makes use of the Visual C++ Debug Heap, and also redefines the `new` operator to include source code file and line information, which is reported together with the leak information in the debugger when a memory leak happens. The memory leak tracking is invoked by a single line, and only compiled into the executable in Debug builds, and replaced by empty functions in Release or Master builds.

3.3.4 Performance Testing

During development, several methods of performance testing were used. A direct time-logging feature via performance counters was used to identify bottlenecks or parts with low performance peaks. Intel's *VTune Performance Analyzer* [66] was used for CPU-side performance profiling. And finally, *NVIDIA NVPerfSDK* has been used to analyze GPU performance.

NVIDIA NVPerfSDK includes both an SDK that can be integrated into the application code to identify bottlenecks, as well as a useful overlay display that can be layered over the tested application called *NVPerfHUD*. NVPerfHUD not only displays performance graphs, but also supports active performance testing by freezing a frame, stepping through each render batch call, viewing textures, and analyzing each batch call's performance on its own.

In order to run NVPerfHUD, the application needs to enumerate a special render device during initialization, and ensure that the game logic processing can handle frame times of zero, at which game processing should halt and the game effectively pauses. In Penta G this is achieved by simply adjusting a setting in the *game.cfg* configuration file (`nv_perf_hud_enable = true`), and launching NVPerfHUD by drag-dropping the Penta G executable onto the NVPerfHUD icon.

After launching, NVPerfHUD can be initialized using the shortcut keys defined previously in the NVPerfHUD launcher. After that all facilities of NVPerfHUD such as the frame debugger, graph view, shader debugger and texture viewer can be used to measure and analyze performance.

3.3.5 Code Style Guideline

In larger projects, the code style guide is a tool for communication within the team of programmers. Such style guidelines ensure that the program source code stays consistent and readable, even for developers new to the code.

In general, code style guides should have the following benefits:

- It should ensure a consistent look of the source code.
- It should enable the reader to understand the code easily.
- It should provide information to the reader which would otherwise require extra effort, such as identifying the scope of variables.
- It should enable new developers to understand most of the code immediately after reading the guideline.

In the Penta G engine, the coding style emphasizes on different naming styles for variables depending on their scope, differentiating between class member variables, variables passed via parameters, and local variables. Classes and namespaces are also named in a way so that the reader may immediately understand whether a class is part of the engine framework (where classes are prefixed with *Gx*) or the application domain.

It is also part of the style guideline that each class should be contained in its own header and source file pair, in order not to clutter source files with multiple classes. This simplifies finding the appropiate source and header file for a specific class as well as reduces build times, as a modification of a source file only means a rebuild of that single class' source file and its dependencies, instead of a rebuild of multiple classes.

The code style is extended to the way class declarations are broken into sections such as public methods, member variables, internal types, and member access methods (getters and setters). It also defines how to document the interface part of each class in order to support automatic generation of documentation via the doxygen system, which is explained in the following section.

3.3.6 Documentation

The engine is documented using the *Doxygen system* [18]. Every class includes a class documentation and each method of a class is documented in a standardized way. The documentation is added as comments in the code only in the interface/class declaration, therefore all documentation is fully included in the include files, which also aids an include/library only distribution.

3.4 Structure of the Engine

In this section, the various subsystems of the GebauzEngine will be described in detail.

3.4.1 Core

The core part of the library contains basic utility classes for string handling, debugging, as well as base classes for common design patterns such as the *Singleton* or the *Generic Factory* pattern [24]. It also provides facilities for adding some reflection support to classes and allowing classes to be instantiated via their string name using a generic factory.

Table 3.2 gives an overview over the most important classes of the engine core system.

The `GxEngineCore` class is a singleton that contains references to all subsystems as the application wishes. The application instantiates and initializes the various subsystems, such as Graphics, Audio and Input, and then registers each subsystem in `GxEngineCore` by calling either `SetRenderContext()`, `SetAudio()` or `SetInput()`.

`GxTimer` implements both global timer functions via static functions, and when instantiated, also acts as a local timer. Both the global and local timer can be paused and resumed to control time processing. It builds on the `GxPerfTimer` class which implements querying a hardware performance timer.

There is a debug logging feature built into the engine via the `GxLogger` class. It can output logging messages to either the debugger or a HTML format log file, or both. By using either `gxPrint`, `gxWarning` or `gxError`, this distinction will also be reflected in the color formatting of the HTML log file.

There is basic support for running a thread in the form of the `GxThreadTimer` class. It provides a callback interface which is called periodically and executed in a separate thread.

`GxEngineCore`	The central hub for the entire engine. Contains references to the file system, render context, audio, input.
`GxIRefCountable`	Abstract interface for classes that are reference-counted. The initial reference count after instantiating is 1, so every instantiated object has to be released at least once.
`GxSingleton`	Base class for classes implementing the Singleton pattern. Implements a template-based version of the Singleton pattern.
`GxGenericFactory`	Template for easy implementation of a generic factory pattern.
`GxLogger`	Class that supports writing log output into the debugger and into a HTML file.
`GxSettings`	Implements a registry of settings from a configuration file.
`GxSettingsFile`	Reader/Parser for a configuration file that feeds a `GxSettings` object.
`GxStringList`	Implements a list of C++ standard strings (`std::string`).
`GxStringUtils`	Implements various static convenience functions and operations on `std::string`.
`GxThreadTimer`	Simplifies the task of creating a thread that periodically calls a specified callback function.
`GxPerfTimer`	Encapsulates a timer based on the system's performance counter.
`GxTimer`	Encapsulates a timer that can be paused and used for measuring time. The basic timing facility is based on `GxPerfTimer`. The class also allows global pausing/resuming of all timers in the system.
`GxObject`	Base class for classes that integrate into the class reflection system.
`GxType`	Encapsulates a class type for the reflection system.
`GxTypeInit`	Initializes the reflection-supporting classes of the engine. Application-specific classes that require reflection support must be initialized after calling `GxTypeInit::InitClassDB()`.

Tab. 3.2: Overview over the Core classes.

The reflection system implements a factory which can be used to instantiate objects of reflection-supporting classes by specifying the string name of the class. It also allows for comparing the class types of instances, as well as querying whether a certain class has been derived from another specific class. This is done by employing the `CLASS_HEADER<n>()` macro and the corresponding `CLASS_SOURCE<n>()`, where `<n>` is the number of base classes the class derives from. Classes declared this way expose a `GxType` member accessible via `GetType()` which contains information about the class type and can be used to compare them. A static member function called `GetClassType()` is also available to be called statically from classes without having actual instance of those classes. The `GxType` class itself offers methods for class type comparison and a useful `IsDerivedFrom()` method that traverses the inheritance tree upwards to check whether a class type is derived from another class type.

The `GxGenericFactory` template is a class template that simplifies the process of creating a class hierarchy where each class can be instantiated by just specifying its string name. It also provides a `GxFactoryRegister` template and a `FACTORY_REGISTER` macro which simplifies the registration of classes into the respective Generic Factory. Listing 3.1 illustrates how these templates and macros can be used to set up a system of classes that can be instantiated via a generic factory.

The benefit of such a generic factory system is that the factory that instantiates the objects possesses no dependencies on the actual classes it instantiates. Each class that can be created through the factory simply registers itself into the factory. Furthermore, it is now possible to instantiate a class simply by knowing its string name, which can prove useful for techniques like serialization.

```cpp
////////////////////////////////////////////////////
// Declaration of the Base class

class Base
{
  ...
};

// Define a helper macro for registration
#define REGISTER_CLASS(e) FACTORY_REGISTER(Base, e)

////////////////////////////////////////////////////////
// Declaration of the Derived class

#include "Base.h"

class Derived : public Base
{
  ...
};

////////////////////////////////////////////////////////
// Implementation of the Derived class

#include "Derived.h"

REGISTER_CLASS(Derived);

...

////////////////////////////////////////////////////////
// now, we can instantiate objects of the Derived class

Base* object =
    GxGenericFactory<Base>::Instance().Create("Derived");
```

Listing 3.1: Using the Generic Factory

3.4.2 File System

`GxFileSystem`	The main central hub class for filesystem related tasks. Manages the virtual file system that includes mount points.
`GxStream`	Interface for file I/O streams.
`GxFileStream`	Implementation of GxStream for a disk file.
`GxMemoryStream`	Implementation of GxStream for a stream that reads from a chunk in memory.
`GxFile`	Stub for file operations that uses a `GxStream` interface for file manipulations. It builds several convenience functions on top of GxStream.
`GxArchive`	Processes an archive file which is a collection of multiple files stored in one large file. Supports reading a single contained file into a memory stream.

Tab. 3.3: Overview over the File System classes.

The file system subsystem contains classes for file system handling. It allows the usage of virtual mount points and memory files through a transparent file interface. It also supports archives that package multiple files into a large one. Table 3.3 gives an overview of the subsystem.

The engine supports packed archives which can be generated using a standalone tool. These archives pack multiple files into a single archive file, which subsequently can be treated like a subdirectory that contains these files.

Mount points are starting points at which the filesystem starts searching for files when using GxFile to open a file.

3.4.3 Graphics

The graphics subsystem contains the Direct3D-based rendering classes. The central class here is the GxRenderContext class, which handles initialization and access to the main rendering device.

The Graphics subsystem can be subdivided into several components by category:

- Basic Graphics classes such as GxRenderContext that cannot be put into other categories and upon which other components are built.

- Resource Management classes that provide interfaces for resource management
- Geometry classes that aid in handling geometric data, vertex and index buffers, as well as loading of model and mesh data
- Texturing classes that handle 2-dimensional and cubemap texturing
- Rendertarget classes that build on the texturing classes to provide render-to-texture capabilities
- Material and Effect classes that implement the shader/effect framework of the engine

Core Graphics Classes

`GxRenderContext`	Central graphics hub class.
`GxDXUtils`	DirectX Utility functions.
`GxUtility`	Convenience functions for rendering 2D quads, spheres, etc.
`GxCamera`	Class that helps setting up the view matrix.
`GxFrustum`	Calculates the view frustum and provides simple functions for testing frustum intersections.

Tab. 3.4: Core Graphics Classes.

The `GxRenderContext` class represents the core of the Graphics subsystem. It provides means to initialize and shut down the render device, clearing the render device and buffer swapping. It is completely independent of the used graphical user interface library and simply requires the Window Handle of the viewport to be used for rendering.

Aside from `GxRenderContext`, the other classes listed in Table 3.4 provide convenience functionality simplifying common tasks such as frustum intersection tests, rendering of fullscreen quads, and view matrix set-up.

Resource Management

The engine provides a resource interface class and a resource manager interface class which simplifies adding new types of resources and adding new resource managers that keep track of them (see Table 3.5).

`GxResourceManager`	Resource Manager template
`GxIResource`	Resource base interface

Tab. 3.5: Resource Management Classes.

The `GxResourceManager` template can be used to build a resource manager for arbitrary resources based on the `GxIRefCountable` interface. The template already handles resource list management, resource access and destruction. The purpose of the template is to simplify building resource managers that prevent loading the same resource multiple times.

The `GxIResource` interface introduces two virtual functions that are called on render device loss and render device reset. Such a case can happen when the screen resolution changes during application runtime. In such a case, resources that are not managed by Direct3D (utilizing the `D3DPOOL_MANAGED` pool) must be freed and restored. This interface simplifies that process.

Geometry classes

The geometry classes in Penta G aid in maintaining models, meshes, and the underlying vertex buffers. They also contain classes for loading of geometric assets. Table 3.6 lists all geometry classes and important header files.

The vertex classes enable automatic generation of vertex declarations via nested templates as described in [10]. Vertex attributes can be combined into vertex compounds via the `GxVertexComposer` template via nested templates, and the template automatically generates the appropiate vertex description, which can then be used to create the vertex declaration required for Direct3D. Listing 3.2 shows the declaration of some common vertex types.

`GxVertexBuffer` implements a single vertex buffer, and offers methods to copy vertex data into it. The `GxGeometry` class builds on this and contains an array of such vertex buffers with fixed semantics assigned, such as vertex positions, colors, normals and texture coordinates. The class provides means to populate the vertex buffers with vertex data, to access and modify the vertex data, and to clear them. The `Activate()` method is used to bind vertex buffers to vertex streams, which is how Direct3D renders vertices.

```cpp
// Position only
typedef GxVertexComposer<GxPosition> V3;

// Normal only
typedef GxVertexComposer<GxNormal> N3;

// Color only
typedef GxVertexComposer<GxColor> C4;

// Position + Normal
typedef GxVertexComposer<GxPosition,
  GxVertexComposer<GxNormal> > V3N3;

// Position + TexCoord
typedef GxVertexComposer<GxPosition,
  GxVertexComposer<GxTexCoord2> > V3T2;

// Position + Diffuse Color + Normal
typedef GxVertexComposer<GxPosition,
  GxVertexComposer<GxNormal,
    GxVertexComposer<GxColor> > > V3C3N3;

// Position + Diffuse Color + Normal + 2D TexCoord
typedef GxVertexComposer<GxPosition,
  GxVertexComposer<GxNormal,
    GxVertexComposer<GxColor,
      GxVertexComposer<GxTexCoord2> > > > V3C3N3T2;

// Subsequently, the vertex description which is
// stitched together using meta-programming can be
// accessed by calling V3::GetVertexDescription(),
// V3C3N3::GetVertexDescription(), etc.
```

Listing 3.2: Vertex Types using GxVertexComposer

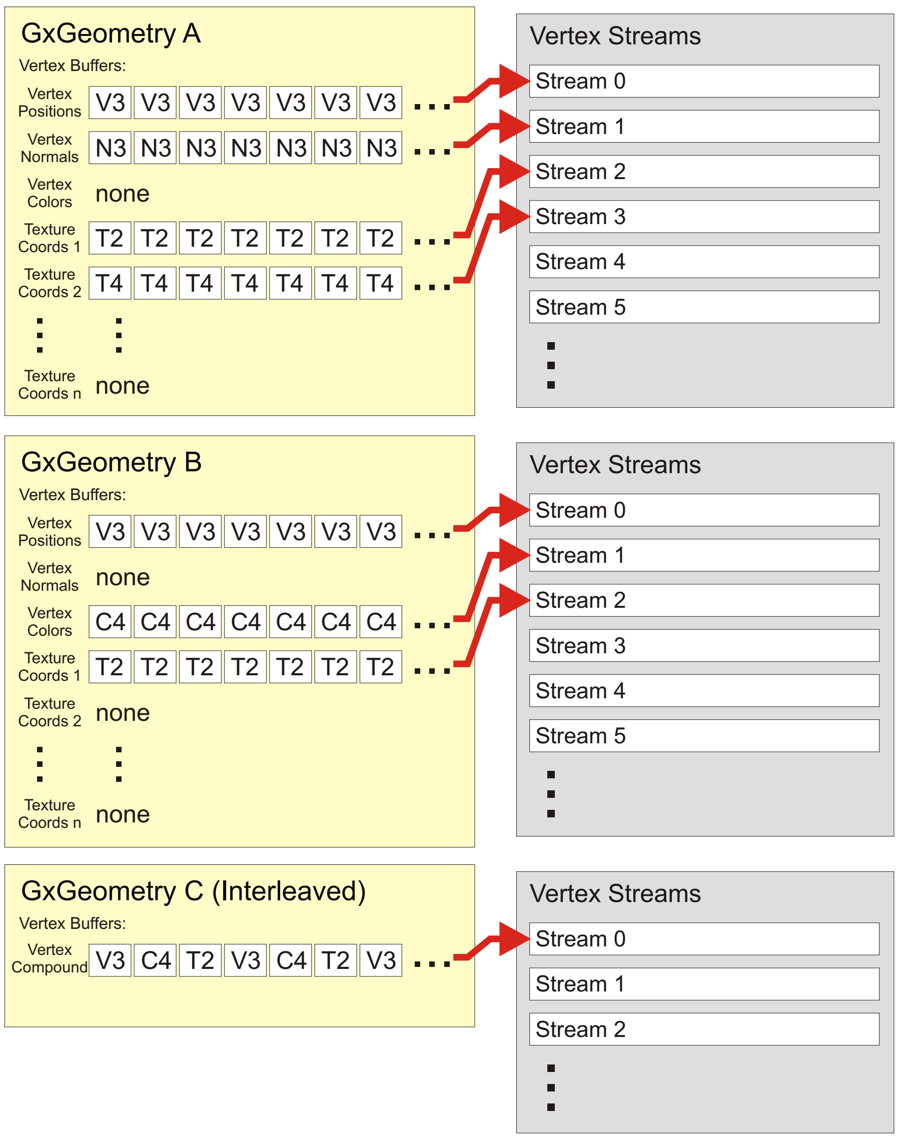


Fig. 3.3: The Geometry class assigns vertex attribute buffers sequentially to streams.

`GxGeometry`	Class that represents geometry including vertex streams for positions, normals, colors, texture coordinates.
`GxIMRenderer`	Convenience class that implements an OpenGL-like immediate mode.
`GxModelLoader`	Loads a model into an instance of `GxModel`.
`GxModel`	Encapsulates a model with multiple meshes that can be used for various purposes.
`GxMesh`	Implements a single mesh as part of `GxModel`.
`GxSubMesh`	Represents a single render batch group of `GxMesh`.
GxVertex.h	Classes that encapsulate a single vertex attribute.
`GxVertexBuffer`	Handles access and data upload to vertex buffers.
`GxVertexComponentDesc`	Helper class that describes a single vertex attribute.
`GxVertexComposer`	Template that links together vertex attributes to larger vertex compounds.
`GxVertexDescription`	Handles management of vertex component descriptions to generate vertex declarations.
GxVertexTypes.h	Contains predefined vertex compound types.

Tab. 3.6: Geometry classes and header files.

Empty vertex arrays that were not populated with data are not assigned to any vertex stream. This is illustrated in Figure 3.3. Note that it is also possible to merge multiple vertex attributes into a single, interleaved vertex buffer, which then gets assigned to just one vertex stream. Listing 3.3 shows some examples on how a `GxGeometry` instance is populated with vertex data. During initialization, `GxGeometry` also handles the creation of a *vertex declaration* which Direct3D uses to match stream indices to the vertex shader varying input by matching the attribute usage type. The created vertex declaration is also bound as the active vertex declaration in the `Activate()` call. The usage types it supports are *position*, *normal*, *color*, and *texture coordinates*. These reflect the HLSL vertex usage types `POSITION`, `NORMAL`, `COLOR` and `TEXCOORDn`, where n is the usage index which is used to be able to specify the same usage type multiple times.

Direct3D defines more usage types such as `TANGENT` or `BINORMAL`, however those only serve as a semantic for human readability and can be replaced by the texture coordinate usage type without loss of functionality. Therefore, in GebauzEngine, texture coordinates should be used for all vertex input types that are not explicitly covered by `GxGeometry`.

`GxIMRenderer` is a useful class that implements an interface similar to the OpenGL Immediate Mode. Instead of just rendering geometry on the fly, it can also be used to bake vertex buffers.

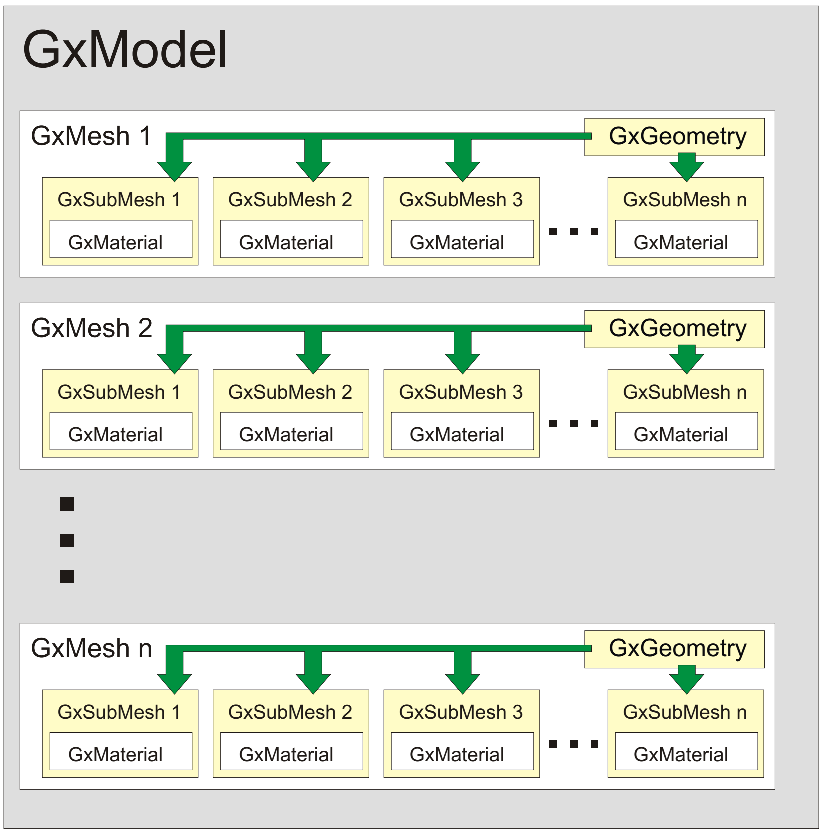


Fig. 3.4: Relationship of the Model Classes.

`GxModel` represents a single GebauzEngine model. This model can contain multiple meshes which can be used for a variety of purposes such as geometric level-of-detail rendering, physics geometry or shadow geometry. Each of these meshes itself is divided into multiple submeshes which represent a subset of the mesh which is rendered in a single draw call, utilizing a single material. Such a submesh may therefore be considered a rendering batch group. This relationship is illustrated in Figure 3.4. The `GxModelLoader` facility enables loading of model files from an engine-propietary format.

```
GxGeometry geometry;

// Populate geometry instance with data.
geometry.Coords().Add(GxVec3f(1.0f, 1.0f, 0.0f));
geometry.Coords().Add(GxVec3f(-2.0f, 0.0f, -3.0f));
// etc...
geometry.Normals().Add(GxVec3f(0.0f, 1.0f, 0.0f));
geometry.Normals().Add(GxVec3f(0.2f, 0.6f, 0.3f));
// etc...

// Texture coordinate sets can be of different dimensions
// This example shows how to add 2d texture coordinates
geometry.TexCoords(texcoord_set).SetDim(2);
// UV coordinate 1
geometry.TexCoords(texcoord_set).AddElement(0.3f);
geometry.TexCoords(texcoord_set).AddElement(0.2f);
// UV coordinate 2
geometry.TexCoords(texcoord_set).AddElement(0.8f);
geometry.TexCoords(texcoord_set).AddElement(0.1f);
// etc...

// GxGeometry can handle both indexed and unindexed
// geometry. This example shows how to add indices.
geometry.Indices().Add(0);
geometry.Indices().Add(3);
geometry.Indices().Add(2);
// etc.

// After populating the GxGeometry with data, it can
// be uploaded to vertex buffers - the bool parameter
// determines whether GxGeometry will collapse the
// attributes into an interleaved vertex buffer.
// This also generates the vertex declaration that
// matches the data.
geometry.Init(interleaved);

// Activate() assigns the created vertex buffers
// to actual vertex streams and sets the vertex
// declaration. This has to be called before
// rendering.
geometry.Activate();
```

Listing 3.3: Examples of using GxGeometry

Texturing and Render Targets

The engine supports 2D and Cubemap textures, which are implemented as a resource and tracked via the texture manager, which itself is derived from the resource manager interface. Table 3.7 lists the texture classes. Section 3.4.3 also introduces the texturing classes used for render-to-texture techniques.

`GxBaseTexture`	Texture base class.
`GxTexture`	2D texture class.
`GxCubeTexture`	Cubemap texture class.
`GxFont`	Loads and renders text using a texture font.

Tab. 3.7: Texture Classes.

Render Targets

In addition to the 2D and Cubemap texture types described in 3.4.3, variants of these texture types exist that are intended to be used as *render targets*. Using these render targets, techniques that require *render-to-texture* functionality can be implemented. Table 3.8 lists the classes used for this purpose.

The render target configuration classes are helper classes that are used to set up a system of render targets and render target textures. The user specifies which render targets are bound to a specific configuration, and subsequently activates the configuration when the application needs to render into these render targets. The render target configuration classes then simplify grabbing a render target texture that contain the rendered information.

Effect framework

The central class of the effect framework is the `GxEffect` class, which is also a resource managed by a resource manager. The class represents a HLSL effect shader contained in an FX file. The Effect Resource Manager provides means of compiling such an FX file into binary code, and writing the binary code to disk, for two reasons: the FX file will subsequently not need to be recompiled, but rather loaded from its binary byte code, and some users of the engine may want to obfuscate their FX shaders.

The Effect framework implements a large part of the *HLSL Semantics and Annotation Standard* (SAS). Semantics are strings assigned to variables in an HLSL shader, which tell the application how that variable is used. The

`GxRenderTexture`	2D texture used as a render target.
`GxCubeRenderTexture`	Cubemap texture used as a render target.
`GxRenderSurface`	Render surface part of a cube map render target.
`GxRenderTargetConfiguration`	Creates a 2D render target set-up.
`GxCubeRenderTargetConfiguration`	Creates a Cubemap render target set-up.

Tab. 3.8: Render Target Classes.

application is then responsible of automatically setting the appropiate values for these variables.

The HLSL Semantics and Annotation Standard defines a large set of commonly used variable types. Table 3.9 lists the Semantics that are supported by the engine and will be automatically set.

Variables assigned to these semantics are set automatically with the `SetUniforms()` method of `GxEffect`.

The effect is applied by rendering batches within a `Begin()`/`End()` block. The `ForEachPass()` determines if there are render passes left. Listing 3.4 illustrates this mechanic.

```
// effect is a pointer to a GxEffect instance
effect->SetUniforms();
effect->Begin();
while (effect->ForEachPass())
{
  // Render batches here
}
effect->End();
```

Listing 3.4: Effect Rendering Passes

Parameters not handled by semantics can be set via the methods listed in Table 3.10.

Such parameters can be set either via a `D3DXHANDLE` to a parameter, which can be retrieved via the `Handle()` method, or via a Semantic Type. The first method retrieves the `D3DXHANDLE` from the effect if that parameter

`WORLD`	Object-to-World matrix (World matrix)
`WORLDINVERSE`	Inverse World matrix
`WORLDINVERSETRANSPOSE`	Transposed inverse World matrix
`WORLDVIEW`	Combined World and View matrix
`WORLDVIEWINVERSE`	Inverse WorldView matrix
`WORLDVIEWINVERSETRANSPOSE`	Transposed inverse World-View matrix
`WORLDVIEWPROJECTION`	Combined World, View and Projection matrix
`WORLDVIEWPROJECTIONINVERSE`	Inverse WorldViewProjection matrix
`WORLDVIEWPROJECTIONINVERSETRANSPOSE`	Transposed inverse World-ViewProjection matrix
`VIEW`	View matrix
`VIEWINVERSE`	Inverse View matrix
`VIEWINVERSETRANSPOSE`	Transposed inverse View matrix
`VIEWPROJECTION`	Combined View and Projection matrix
`VIEWPROJECTIONINVERSE`	Inverse ViewProjection matrix
`VIEWPROJECTIONINVERSETRANSPOSE`	Transposed inverse View-Projection matrix
`PROJECTION`	Projection Matrix
`PROJECTIONINVERSE`	Inverse Projection matrix
`PROJECTIONINVERSETRANSPOSE`	Transposed inverse Projection matrix
`VIEWPORTPIXELSIZE`	Size of the viewport in pixels as float2
`VIEWPORTCLIPPING`	Near distance, far distance, width angle(radians), height angle(radians) as float4
`TIME`	Current engine system time
`LASTTIME`	Last frame time
`ELAPSEDTIME`	Time between adjacent frames

Tab. 3.9: HLSL Semantics supported by the engine.

`SetFloat`	Set a float shader parameter.
`SetFloat2`	Set a float2 shader parameter.
`SetFloat3`	Set a float3 shader parameter.
`SetFloat4`	Set a float4 shader parameter.
`SetTexture`	Set a texture parameter. The supplied texture can be a 2D Texture or a Cubemap.
`SetBool`	Set a bool shader parameter.
`SetInt`	Set an int shader parameter.
`SetTechnique`	Sets the active technique.

Tab. 3.10: Methods for setting effect parameters

has been queried for the first time, stores that handle in a map, and subsequently returns that cached handle when asked for that parameter again. The latter method requires a Semantic, and sets the values of the variables that are bound to that semantic.

The vertex streams from a `GxGeometry` object are bound sequentially (if they exist) to the vertex shader input structure in the following order:

- Vertex positions
- Vertex colors
- Normal vectors
- Texture coordinate layers which are also used for other purposes than texture UV mapping (for instance, tangent space binormals)

Thus the vertex shader input layout in a specific effect file must match both what is in the model file as well as this order.

3.4.4 Input

The input subsystem handles input controllers via DirectInput. The central hub class is GxInput, which handles all initialization tasks. Through it the developer can also access the keyboard, mouse and optional game controllers plugged into the system. Table 3.11 illustrates the classes present in this subsystem.

The input device implementations generally support querying of the current state of an input device, as well as query whether during the last polling,

`GxInput`	Handles initialization and destruction of input devices, and enumeration of game controllers. It is also responsible for polling all input devices it supports.
`GxInputDevice`	Base class for all input devices.
`GxKeyboard`	Implements keyboard access routines.
`GxMouse`	Implements mouse access routines.
`GxController`	Implements routines to access game controllers in a general fashion.

Tab. 3.11: Overview over the Input classes.

an input state has changed such as a keyboard key transitioning from neutral to pushed state or vice versa.

3.4.5 Audio

The audio subsystem enables the usage of sound devices. Similar to the graphics and input subsystems, there is a central hub class called `GxAudio`, which – similar to `GxRenderContext` and `GxInput` – handles initialization tasks and acts as a hub to the audio subsystem functionality. Table 3.12 lists the relevant classes of this subsystem.

When a multiple instances of a sound are played, multiple audio buffers are instantiated, as each sound instance – also called a *voice* – requires an own audio buffer. However, when playing back the same sound multiple times, generating multiple audio buffers with full copies of that sound sample would unncessarily waste memory. Therefore, a DirectSound feature called *buffer cloning* is used, which intantiates a copy of an audio buffer that points to the same sample memory, thus effectively sharing the same sample memory, allowing it to be played back multiple times with own parameters such as panning and volume.

3.4.6 Parser

The parser subsystem is a collection of classes aiding in parsing text files. It accepts a definition of tokens and keywords and can subsequently check an input string for lexical correctness, and output a token stream which then can be grammatically parsed. Table 3.13 gives a brief explanation of the system's components.

`GxAudio`	Handles initialization and destruction tasks, and provides simple interfaces for audio playback.
`GxAudioBuffer`	Handles a single sound sample that is kept in memory as a whole. Such sounds are called *resident samples*.
`GxWaveFile`	Handles the loading of an entire wave file into memory. Feeds `GxAudioBuffer` with a resident sample from a wave file.
`GxAudioStreamBuffer`	Implements a buffer the sample data of which is frequently updated. Such samples are called streaming samples, as they are commonly used to continually read from an audio file too large to be loaded as a resident sample such as music.
`GxIStreamSource`	Abstract interface implementing a data source that feeds the `GxAudioStreamBuffer`.
`GxWaveStream`	Implements the `GxIStreamSource` interface to feed a streaming sample with data from a wave file.
`GxOggStream`	Implements the `GxIStreamSource` interface to feed a streaming sample with data from an Ogg file.
`GxAudioSource`	A separate audio voice that plays back a sample (resident or streaming). It may have positional parameters for 3D sounds.

Tab. 3.12: Overview over the Audio classes.

`GxTokenizer`	Tokenizes a string into a token stream by lexical analysis.
`GxXMLParser`	Uses `GxTokenizer` to parse an XML string and convert it into an XML document tree.

Tab. 3.13: Overview over the Parser classes.

The tokenizer can be fed with certain keywords and tokens and its scanning behaviour configured such as which characters to use for commenting. The result token stream can then be easily parsed and processed in whichever way the developer wishes to.

3.4.7 Application Framework

The application framework (AppKit) is a small library that enables quick development of new applications based on the GebauzEngine. It is built directly on the Win32 API and supports simple creation of an application main window in fullscreen or windowed, as well as a simple event routing system.

Section 3.6 shows how to build a minimal application using the application framework.

3.5 Asset Pipeline

This section describes the workflow of importing geometry into a format the engine can use, and the design of the content pipeline. It will give a mostly structural view; for a guide on creating Penta G content, refer to the Appendix.

3.5.1 Toolchain Structure

The asset tools of Penta G consist of two primary tools: the *SUX Model Editor*, and the *GTP Editor*. The SUX Model Editor is an asset tweaking and import/export tool not specific to GebauzEngine but used for other engines as well. The GTP Editor is a separate level editor for Penta G that contains Penta G-specific functionality, but is general enough that it has been used for non-GebauzEngine applications in the past [28].

Both tools have been written in Turbo Delphi, which is freely available by CodeGear, the developer tools subsidiary of Borland. The reason for this is that Delphi features a comfortable designer for graphical user interfaces. The rendering engine that runs both tools is the SUX Engine, another game engine that is based on OpenGL and Delphi. The main reason for using this different engine for tools was that the SUX Engine and the SUX Model Editor both already existed and was therefore synergetic to use instead of reinventing the wheel.

No changes had to be made in the SUX Model Editor specifically for Penta G, as the conversion between right-handed and left-handed coordinates, caused by the different behaviour between DirectX and OpenGL, was made during loading of the model format.

Figure 3.5 roughly illustrates the asset pipeline workflow, with the components and tools necessary to create content.

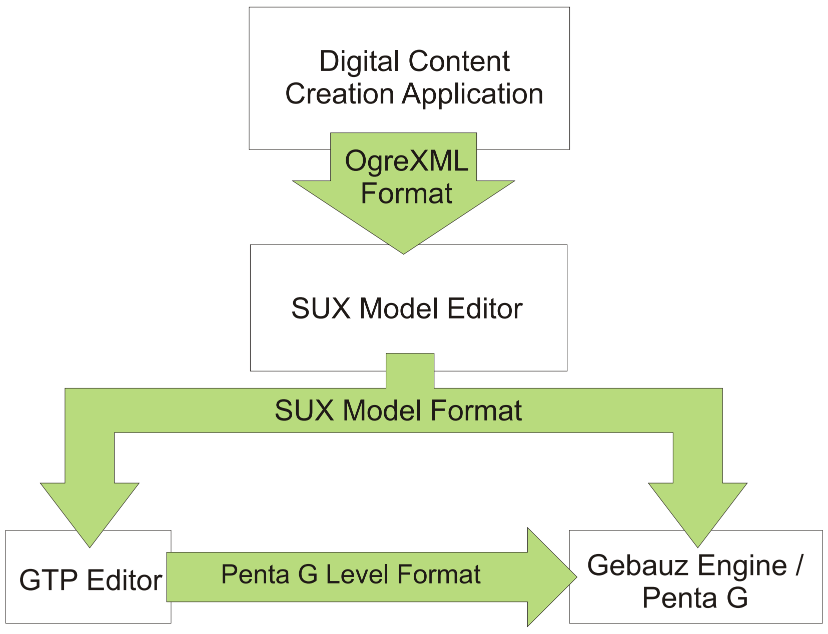


Fig. 3.5: Structure of the Asset Pipeline.

3.5.2 Collaboration with the Engine

SUX models can be loaded via the `GxModelLoader` class. Appropiate materials applied to that model are loaded via the system described in Section 3.5.3. The model loader reads the model file and generates the instance of the `GxModel` class that represents that model. It also provides conversion from OpenGL's right-handed coordinate system to DirectX's left-handed one. Section 3.4.3 gives a detailed description of the geometry data structures in GebauzEngine.

Level files, which accumulate several SUX models in a level, are handled on the application side, as level files created with the GTP Editor contain application-specific data. This process is described in Section 4.3.

3.5.3 Material/Effect System

The material system uses so-called *material instance scripts*, which define the shader and parameters for a single material instance of a batchgroup inside a mesh. This system is fully implemented on application level and is therefore replaceable. The GebauzEngine only provides an abstract material-loader interface so that custom scripts can be created and loaded.

Material scripts reference an effect that they use, and subsequently set parameters those effects expose. Hence, custom effect shaders can be integrated into the system. The material instance scripts are edited and entered in the SUX Model Editor application.

Material Instance Script Syntax

In the material instance scripts, each semicolon-separated line defines a statement. Table 3.14 gives a brief overview over the available instructions within each statement.

`uses [fx_filename]`	Links the material to a specific effect file.
`set [type] [var] [params]`	Set the value of a variable.
`transparent`	Marks the material as one that uses alpha-blending in order to perform depth-sorting.
`alphatest`	Marks the material as one that uses alpha-testing.

Tab. 3.14: Material instance script instructions.

The `set` instruction sets the value of a certain variable. The first parameter it expects is the type of the variable, followed by the variable name, and the actual value. If the variable type requires more than one value, such as for 4-dimensional floating point variables, the values are specified using a comma-separated list. Table 3.15 lists the available variable types.

Listing 3.5 shows an example material script.

The developer of the shader needs to set the used effect file and specify the technique that is used. Other parameters such as textures or variables correspond to uniform variables in the effect file, and therefore depend on the actual effect file.

The material instance script is parsed by an implementation of the `GxIMaterialLoader` interface, which is then passed to the

`technique`	Specifies a technique to be used. The parameter is a string denoting the technique.
`texture2d`	Sets a 2d texture. Followed by a string specifying the name of the texture variable and the texture file to be used.
`textureCube`	Sets a cubemap texture. Followed by a string specifying the name of the texture variable and the texture file to be used.
`float`	Sets a float variable. Followed by a string specifying the name of the float variable and the actual float value.
`float3`	Sets a float variable. Followed by a string specifying the name of the float3 variable and three float values separated by commas.
`float4`	Sets a float variable. Followed by a string specifying the name of the float4 variable and four float values separated by commas.
`int`	Sets an integer variable. Followed by a string specifying the name of the integer variable and the actual integer value.
`bool`	Sets a boolean variable. Followed by a string specifying the name of the boolean variable and the actual boolean value.

Tab. 3.15: Possible variable types for material instance scripts.

```
uses "normalmapping.fx";
set technique "parallax_glow_to_ambient";
set texture2d "colormap" "rustybox_cm.png";
set texture2d "normalmap" "rustybox_nm.png";
set texture2d "surfacemap" "rustybox_sm.png";
set float "ambientfactor" 15;
set float4 "lightdir" 0.6, 0.6, 0.2;
transparent;
```

Listing 3.5: Example Material Script

SetMaterialLoader() method of GxModelLoader. When a model is loaded and a material loader implementation specified, the model loader will use the specified material loader implementation and call its CreateMaterial() method which takes a string containing the material instance script and returns an instance of GxBaseMaterial or a derived class.

Thus it is possible to completely replace the material instance script system by implementing a different parser. The Penta G engine's material script parser is implemented in the MaterialLoader class. The existing system already allows for high flexibility to use any effect and shader with any model, though, so an actual custom implementation of a material loader may not be necessary.

It is also important to note that these effect files are only read and handled by Penta G itself, and the level editor (see Appendix B) contains its own predefined materials for solid and transparent objects. This is because for simplicity, the level editor does not feature all effects that the regular game offers, as this was found to not be an important necessity. In future versions of the editor, this restriction could be lifted in order to make the editor utilize the same material system and effect shader files as the actual game.

3.6 Anatomy of a minimal GxEngine Application

This section will give a code example of a minimal application built on the engine. This minimal application will do the following tasks:

1. Read settings such as fullscreen/windowed mode and resolution from a configuration file
2. Create a window for rendering, respecting the previously read settings
3. Initialize the Direct3D render device by initializing the Graphics subsystem
4. Initialize the Input subsystem
5. Initialize the Audio subsystem
6. Start the main game and render loop where the render device is cleared, and its backbuffer swapped

Items 1 to 2 are handled by the Application Kit. It suffices simply deriving an own application class from the GxApplication class. In order to listen to application events, it is recommended to also derive from the `GxIEventHandler` interface, which registers the class as an event listener. The `GxApplication` base class sends events to all registered event listeners by invoking their implementation of the `OnEvent()` method.

Another useful class to derive from is the `GxDefaultEventRouter` class. It defines several pure abstract event handler methods which are invoked by its `OnDefaultEvent()` method. An application utilizing this base class may just call `OnDefaultEvent()` for every event it doesn't want to handle by itself, and the `GxDefaultEventRouter` class will re-route the events to the appropiate virtual methods.

Listing 3.6 illustrates a basic application built in such a way that satisfies points 1 to 2 mentioned above.

```
class MinimalApp :
  public GxApplication,
  public GxAppKit::Abstract::GxIEventHandler,
  public GxAppKit::GxDefaultEventRouter
{
public:
  MinimalApp();
  virtual ~MinimalApp();

  virtual GxEvent::ResultType OnEvent(const GxEvent& e)
  {
    return OnDefaultEvent(e);
  }

  virtual bool OnInit();
  virtual bool OnShow();
  virtual void OnDestroy();

  virtual void OnFrame(float frameTime);

  void Update(float frameTime);
  void Render();

protected:
  GxRenderContext mRenderContext;
  GxInput mInput;
  GxAudio mAudio;
};
```

Listing 3.6: Minimal Application Class Declaration

The class also already includes member variables for the main graphics, input and audio classes. It also implements the `OnInit()`, `OnShow()`, `OnDestroy()` and `OnFrame()` event handler methods.

```
GxEvent::ResultType MinimalApp::OnEvent(const GxEvent& e)
{
  return OnDefaultEvent(e);
}

bool MinimalApp::OnInit()
{
  if (!mRenderContext.Init(GetMainWndHandle(), mWidth,
     mHeight, mStartFullscreen))
    return false;
  gxEngineCore().SetRenderContext(&mRenderContext);

  if (!mInput.Init(GetInstance(), GetMainWndHandle()))
    return false;
  gxEngineCore().SetInput(&mInput);

  if (!mAudio.Init(GetMainWndHandle()))
    return false;
  gxEngineCore().SetAudio(&mAudio);

  return true;
}

bool MinimalApp::OnShow()
{
  // initialize game-relevant things here
}

void MinimalApp::OnDestroy()
{
  // graphics, input, and audio get uninitialized in
     their destructor
}
```

Listing 3.7: Minimal Application Initialization and Destruction

The `OnInit()` method is called right after initialization of the application, but before displaying the main window and before starting the main loop. `OnShow()` is invoked right after the main window is activated and made visible, but also before the main loop. `OnDestroy()` is called after exiting the main loop. These three methods are the main location where further initialization and destruction of the application should be done, such as for the graphics, input and audio subsystem. Their implementation thus handles points 3 to 5 in the list mentioned above, as shown in listing 3.7.

Lastly, the `OnFrame()` method is called frequently when the application is active and idle and no system messages are processed. This is the main point where game logic and rendering should be implemented. The method gets the elapsed time since the last call of that method, and is responsible for handling that frame time appropiately.

Listing 3.8 shows a sample implementation that splits the method into an own *Game Update* and *Game Rendering* part.

```
void MinimalApp::OnFrame(float frameTime)
{
   if (frameTime > 0)
   {
     Update(frameTime);
   }

   Render();
}

void MinimalApp::Update(float frameTime)
{
  // update game logic dependent on frame time
}

void MinimalApp::Render()
{
  if (mRenderContext.CheckDevice())
  {
    mRenderContext.Clear();
    gxRenderContext().BeginScene();

    // do actual rendering

    gxRenderContext().EndScene();
    gxRenderContext().Swap();
  }
}
```

Listing 3.8: Minimal Application Update and Rendering

In the rendering part, the screen is cleared and the backbuffer is swapped. The `CheckDevice()` method handles render device loss and returns true whenever rendering is possible.

This minimal application can be used as the basis for a custom application built on the engine.

Penta G itself is also built this way, and adds a lot of additional functionality in its application code domain. These parts together with the GebauzEngine framework constitute what is called a *Game Engine*. The following chapter will detail these Game Engine parts of the application domain.

Chapter 4

Building a Game for Research – Penta G

This chapter will describe the development of the demo game based on the GebauzEngine called *Penta G*. We will present the software components of the game that are not part of GebauzEngine but rather built on top of it. Furthermore, we will outline the more advanced features and techniques utilized in Penta G, and finally we will compare the game to the requirements list we developed in Section 2.5.

4.1 Introduction

The initial purpose of the demo game and thus the reason for the development of both the GebauzEngine and the Penta G game was to showcase technologies of the GameTools project (see Section 5.1). It has been afterwards extended to be used in several other research areas, as the interest in Penta G being a state-of-the-art game and game engine fully available to research grew.

One of the goals during the demo game development was to create a game as professionally as possible, and as such, Penta G can be used as a test bed for research that is very close to real-world application scenarios.

4.2 Game Object Hierarchy

Every in-game object in Penta G is derived from the *Entity* base class. This base class provides virtual methods for various tasks such as updating, rendering, and reading data from a level file. Table 4.1 gives an overview over the game entity hierarchy in Penta G.

Each of these classes is also registered in the engine's reflection system, and thus it is possible to compare the types of entity instances, check whether an entity instance is derived from a certain entity class, and to generate an entity from its string name. The latter feature is used when reading a level file: the level loading mechanism reads the string name of an entity class and thus can instantiate a class object via the reflection/factory system. Subsequently, the virtual *Load()* method of that class instance is called. Every entity serializes itself from the level file by implementing this method.

Game entities are well defined for a specific game, therefore a lot of the entities presented here are specific to Penta G gameplay, which essentially implements a standard *first person shooter*. A full conversion of Penta G into a completely different game with different gameplay and game mechanics would require the implementation of new entities, as well as discarding some Penta G entities that have no use in that different game. However, some more generally usable game entities also exist in Penta G, such as the particle system classes.

Entities are managed in the *EntityList* class, which contains a list of all existing entities. This list may be larger than the initial list after reading a level file, because newly spawned entities which are generated later during the game are also added to this list. Deleted entities are removed from this list as well. This class also simplifies iterating through the list.

Each entity's visibility is marked by setting a visibility flag as well as the frame integer ID where that flag was last updated. This mechanic is necessary for *Coherent Hierarchical Culling* as explained in Section 4.6.4, but is used for generic frustum culling too.

Every entity can be named in the editor, but since that is a feature used purely for usability, no mechanic is in place to ensure unique names.

In the following sections, some pivotal game object types will be described in more detail.

4.2.1 Start Position Object

The start position object simply marks the point in space where the Player Object will spawn.

4.2.2 Environment Models

Environment Models are static level geometry that themselves do not react to physics, but are still collidable geometry. Other dynamic physical entities can collide with environment models and will react to them. This type of entity is the main type for building a level.

4.2.3 Actors

Actors are actively moved entities in the level, that offer more dynamic behaviour than environment models. They have a health property and are

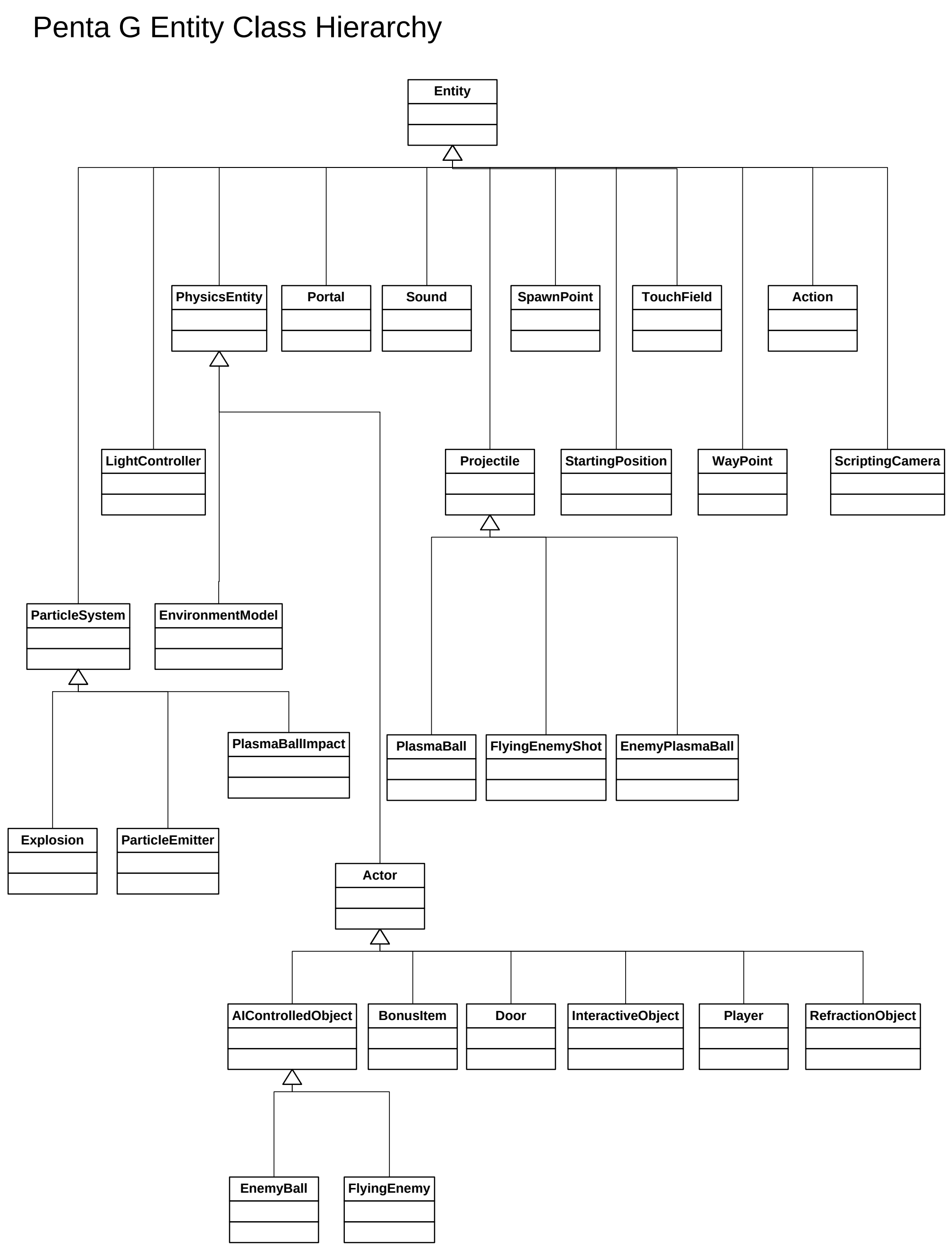


Fig. 4.1: Entity Hierarchy

destructable. Several subtypes of actors exist and are described in the following sections.

The Player Object

The Player Object is a special entity that can only exist once in the entire level. It encapsulates the player as a scene entity, so that other entities can react to it, or influence it in other ways. It itself is a physics entity that implements a capsule-based character controller.

User input influences movement of the player object. The player object also updates the camera depending on its position and viewing direction, since in a first person shooter type of game, the camera eyepoint is always identical to the player object eyepoint.

The player object also possesses instances of the Weapon class, that implement the player weapon in the game.

Interactive Objects

Interactive Objects are dynamic level geometry, that are affected by physics, and react to forces. These are typically used for level props that can be moved around.

AI Controlled Objects and Spawn Points

AI Controlled Objects are entities that utilize their own artificial intelligence to move in the scene. The `AIControlledObject` base class provides waypoint utility functions so that classes derived from it can follow paths defined by waypoints. It also provides means to set and follow target entities. AI controlled objects are not placed directly, but emitted from the `SpawnPoint` entity. This entity spawns AI controlled object in a user-specified frequency. The spawn point does not start emission automatically, but only when triggered via an action entity.

4.2.4 Waypoints

Waypoints are points defined in the scene by their position and radius. AI Controlled Objects can use them to define paths to follow. A radius can be defined so that objects using the waypoint do not move exactly to the central position, but can also choose a random location within the radius to make AI movement appear more natural.

4.2.5 Particle Systems

Particle systems are systems that emit objects according to a certain emission scheme. These objects are called particles, and they also move according to well-defined schemes, including a per-particle life-time. Such particle systems can be used to simulate atmospheric effects like fog, clouds or explosions. The particle system classes are responsible for updating particles and rendering them efficiently, which usually means batching particle sprites into larger batch groups before sending them to the render device.

4.2.6 Actions

Actions are an entity type that implement a similar functionality to the *delegate* design pattern [24]. They define certain predefined actions that can be executed on either light sources or entities. Entity types handle actions by implementing the `Trigger()` method that is called when an Action is executed. A delay for waiting until the execution is initiated can also be specified.

This class is implemented as an entity so that it can be placed and positioned in the level editor and visually edited instead of separately edited. This is also the central class for Penta G scripting (see Section 4.4.3).

4.2.7 Touchfields

Touchfields are geometric areas in the scene that execute a `TRIGGER` action when other entities enter or leave them. They can be used to trigger events depending on the movement and placement of scene entities.

4.2.8 Scripting Cameras

Scripting Cameras provide alternative camera views for sequences and cutscenes and can be invoked via the Action entity.

4.2.9 Portals

A portal is a simple trigger that loads a new level. This is mainly used to load the next section when arriving at a certain point in a level.

4.2.10 Sound Entities

A sound entity can be an ambient sound, which loops infinitely, or a triggered sound, which is activated on a trigger event. It has a position which is also applied to the sound played.

4.2.11 Light Controllers

Light Controller entities can control light sources in a variety of ways, such as making light sources flicker or pulsate, turning them on and off, as well as animate them. These actions are defined in the editor.

4.3 The Asset Pipeline on Application Level

The GebauzEngine only provides a basic model loading functionality. Since a game typically requires extended functionality such as game-relevant information, the application implements an own Model class that encapsulates the engine's model class (`GxModel`) via aggregation. The application domain also maintains a simple list of all the models in a level, implemented in the `ModelList` class.

The engine provides a material loading interface where applications can plug in their own material script parser. Since the engine does not provide an own material script parser, it is implemented in the application code. The Penta G material instance script parser is executed by the model loader. The syntax of the material instance script is explained in detail in Section 3.5.3.

The GTP Editor (see Appendix B) and the level format are also completely on application level – the engine does not have a notion of levels. The level is basically a scene populated with game objects the implementations of which are also contained in the application domain.

4.4 Game Logic and Scripting

This section will describe the game state logic and how the update flow from the application main loop down to each game object's update is structured.

4.4.1 Main Loop

Programs typically have a main loop where the application's event processing is handled. The application runs until it receives an event that designates a *quit* command, and during that time, processes a queue of event messages if that queue is non-empty. These events are operating system specific, for example in Microsoft Windows they are named *Windows Messages*.

In times where the application is idle, i.e. there are no pending operating system events to be processed, it can process the game itself. In the case of

Penta G's AppKit, an own event framework is provided that handles engine events by invoking callback functions implemented as virtual functions of an `IEventReceiver`-derived class. During the application main loop's idle time, an `ON_FRAME` event is triggered that calls the appropiate handler function.

That handler function subsequently has to split engine processing into a separate `Update()` and `Render()` call. The `Update()` call receives the elapsed time span since the last time it was invoked. It also has to prevent processing when the elapsed time is zero, because delta times of zero may influence the game logic negatively.

Ideally, the handler function also splits updating into maximum time chunks when the elapsed time is higher than a specific threshold to ensure that `Update()` always receives a small enough delta time so that processing that may require smaller time steps is ensured to work correctly. A typical type of processing that requires small time steps is collision detection – large time steps may cause objects to pass through walls that should be unpenetrable.

At the same time, the logic update needs to make sure that it works robustly when times between frames are zero. It must both prevent division-by-zero bugs which may happen if parts of the code divide by the frame time, as well as make sure that when the frame time is zero, that the game effectively pauses. This is especially important when using performance analyzation tools such as *NVIDIA NVPerfHUD*.

4.4.2 Game States

Game States implement the game's application flow as a state machine. A Game State is defined as a part of the game that implements its own game logic and that is mutually exclusive to other game states, meaning that the game may only be in one single game state at a time. The game state manager handles the transition between game states and ensures that the currently active game state is properly updated and rendered.

In Penta G, the game state manager also implements a generic factory that can create instances of game states via its string name. Each game state must thus be registered in the generic factory. The *GxGenericFactory* template included in the engine core classes provides easy to use macros to simplify this.

4.4.3 Scripting System

Scripting is the process of automating and controlling a software application without modifying the software's own source code. In the context of games, scripting refers to automated processes and events that can be defined by a level designer for purposes such as automatic sequences, cinematic cutscenes, or event processing. In larger and general purpose game engines, scripting is often achieved by integrating a separate *scripting language* on top of the actual engine source code.

During the development of Penta G, scripting language systems such as *Lua*, *Boost.Python* and *Stackless Python* were evaluated, but a more straightforward and restrictive scripting system was implemented instead, since such scripting language systems added too much maintenance effort compared to the time constraints in which Penta G had to be developed.

Action Type	**Result**
Trigger Entity	Trigger an entity.
Set Active	Activate an entity.
Set Inactive	Deactivate an entity.
Lock	Lock an entity.
Unlock	Unlock an entity.
Delete Entity	Delete an entity.
Kill Entity	Set an actor's health to zero.
Inflict Damage (P1) on Entity	Add damage to an actor's health.
Set Entity Model (P1)	Change an entity's assigned model.
Set Entity Position to Waypoint (P1)	Put an entity to a waypoint.
Move Entity To Waypoint (P1) with Speed (P2)	Causes an entity to move towards a waypoint.
Turn On Light	Switches on a light source.
Turn Off Light	Switches off a light source.
Set Light Position (P1)	Puts a light source to a specified position.
Set Light Color (P1)	Changes the light source color.
Set Light Radius (P1)	Changes the light source radius.
Set Actor targetable	Makes an actor targetable by AI.
Set Actor not targetable	Makes an actor untargetable by AI.

Tab. 4.1: Action Types. *The result of most actions depends on the entity the action is applied to.*

The Scripting System of Penta G uses predefined action types and relies on the implementation of each Entity to handle the action correctly in the context of the entity type. The central entity type for scripting is the `Action` class, which connects and sends action types to actual entities. An Action defines the actual action that has to be executed and the entities on which the action operates, plus additional parameters such as a waiting delay after which execution starts and some conditions. The action types available are predefined in the engine. Table 4.1 lists the action types available in the engine and gives a short description for each. Note that the actual result of the action depends on how the specific target entity implements it.

Actions themselves can also be triggered by various entities that define certain events, such as the *Starting Position entity*'s "On Start Level" event which is triggered when the player entity is spawned. Due to the generality of the action types, most in-game scripting needs should be satisfied, and more complex sequences can be created by chaining actions. For instance, one could start an action when the level loads, which changes the camera view, triggers an action that fades in from black, which in turn triggers an action that starts playing a sound.

4.5 The Rendering Pipeline

This section describes the rendering pipeline of Penta G. Penta G applies multiple passes to achieve its rendering output, as shown in Figure 4.2. The render passes applied are:

1. **Depth Only Pass**. Establishes the Depth Buffer and renders the linear depth into the color channel for later usage. Visibility information is established and stored.

2. **Shadow Map Passes**. Shadow-casting light sources render their shadow maps into cubemap render targets.

3. **Environment Map Passes**. Reflecting or refracting entities require a cubemap rendering of their surroundings.

4. **Color/Glow Pass**. Renders the color output and the glow buffer into two targets in a Multiple Render Target (MRT) setup.

5. **Horizontal Glow Blur Pass**. Blurs the glow buffer horizontally.

6. **Vertical Glow Blur Pass, Image composition**. Uses 1 pass to vertically blur the glow buffer and combine the color render output and the blurred glow.

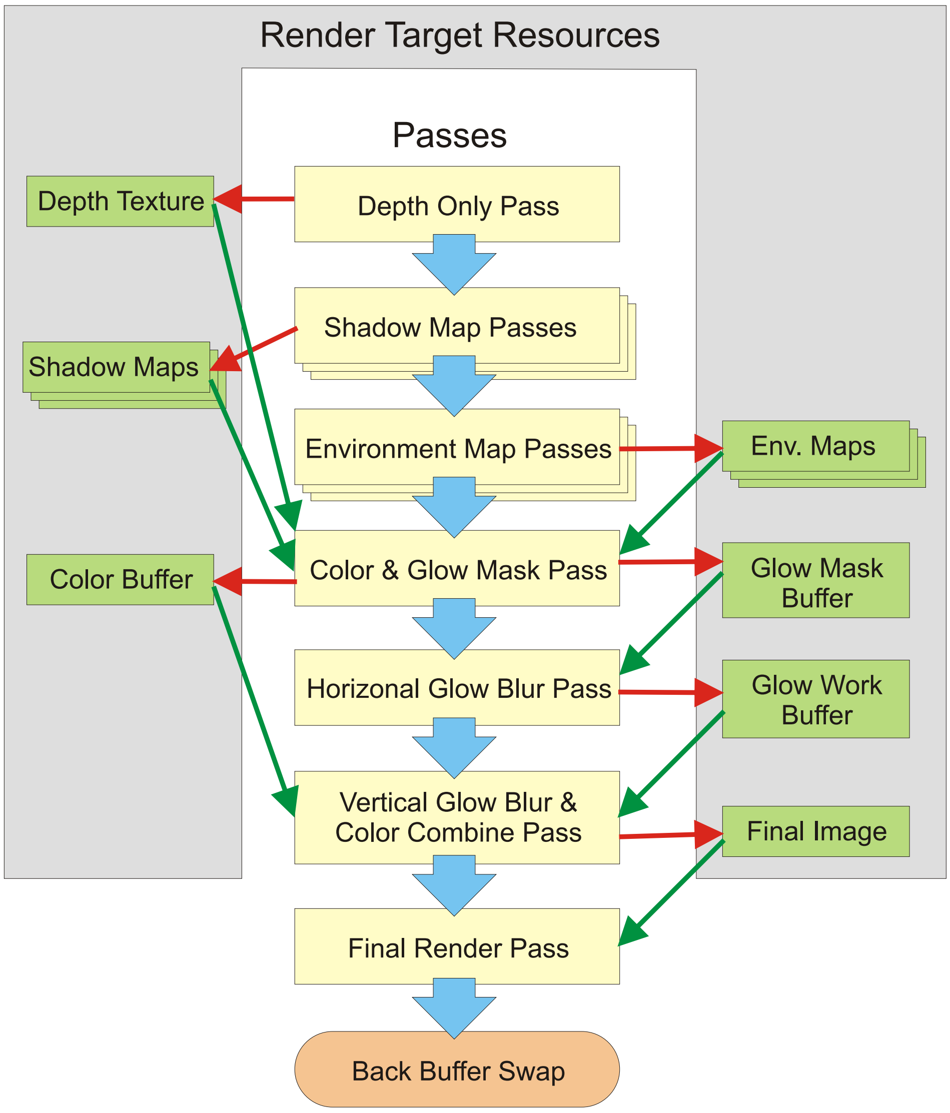


Fig. 4.2: The Penta G Rendering Pipeline.

7. **Postprocess Effects**. Damage and distortion effects are applied to the result of pass 6, and the On-Screen Heads-Up Display (HUD) is rendered

Pass 1 uses either traditional per-entity frustum culling or Coherent Hierarchical Culling (see Section 4.6.4) to determine visibility for each scene entity. This information is subsequently stored in each entity so that subsequent passes do not need to newly determine visibility.

Pass 2 is actually is made up of multiple passes, 6 for each shadow-casting light source as Penta G uses omni-directional light sources emitting light into every direction and thus also casting shadows in all directions. Despite utilizing light source culling, effectively only rendering shadow maps for light sources the influence radius of which are intersecting the view frustum, this can be a performance bottleneck when a lot of light sources are visible in a scene. This performance can be finetuned and scaled by tweaking the shadow map sizes though.

Pass 3 is optional and is only executed when there are any visible reflecting or refracting objects in the scene. Like the shadow map passes in 2, each single reflecting or refracting object requires 6 passes to generate its cube environment map.

Pass 4 is the actual scene render pass which renders color output into a render target, but also renders the glow portions of the scene into a separate render target. This separate render target is subsequently blurred vertically and horizontally and finally composed together with the original color rendering in passes 5 and 6.

Finally, pass 7 applies some additional postprocess effects such as a distortion effect to depict shield damage.

The post process passes (5-7) have been removed in the version for the Crossmod Project (see Section 5.2 because for perceptual experiments, such postprocess effects disturb results too much. For that modification, the multiple render target rendering has also been removed, so that pass 4 only renders the color output into one render target.

4.6 Advanced Features and Technologies

This section describes some of the more advanced features some of which were integrated for the purpose of showcasing GameTools technologies (see Chapter 5.1).

4.6.1 Multiple Render Targets and Postprocess Effects

Several *Postprocess Effects* have been added to the Penta G rendering pipeline. Postprocess effects are techniques that modify final render outputs, and operate on them in two-dimensional image space.

In the case of Penta G, the actual scene rendering outputs into more than one render target in a single pass. This is a feature of modern graphics processing unit that is called *Multiple Render Target Rendering* (MRT rendering). The pixel shader can specify which color to write to which specific render target in the MRT chain. The disadvantage is that on DirectX 9 class hardware, MRT rendering cannot be used in conjunction with *multisampling anti-aliasing* (MSAA). The advantage is that one MRT pass can prove to be more efficient than several single passes – this is especially the case for higher geometric complexity, as in an MRT setup, geometry only needs to be transformed once. Penta G uses an MRT setup with two render targets:

- A *color render target* which contains the color rendering of the actual scene.
- A *glow buffer render target* which contains the glow intensity per pixel.

The Penta G postprocess effects modify the render target and also use the glow buffer render target (as well as possible additional textures) as parameter source for the effect.

The effects used in Penta G include the following:

Glow Effect

This effect uses a separable gaussian blur to amplify glowing parts of the scene, similar to the glow technique by Greg James and John O'Rorke [33]. These glowing parts are specified via the surface map texture each object in Penta G possesses, and are rendered into the glow buffer render target. This glow buffer is then blurred once horizontally, and once vertically, to achieve a blur using a separable convolution Gauss filter.

In the final scene composition, this blurred glow image is superimposed and blended onto the color render target and results in an intensified glow. Figure 4.3 illustrates this blurring and image composition process to create a convincing realtime glow effect.

Distortion Effect

The distortion effect used for shield damage feedback in the game is also implemented as a postprocess effect. The goal of the effect was to achieve a degradation of the image that is visually similar to artefacts in digital transmissions, such as MPEG data corruption artefacts [38]. Figure 4.4 graphically shows what this effect looks like.

4.6.2 Parallax Mapping

Normal Mapping is a way to encode surface information into texture maps to enable highly detailed per-pixel lighting. Instead of RGB color values, each

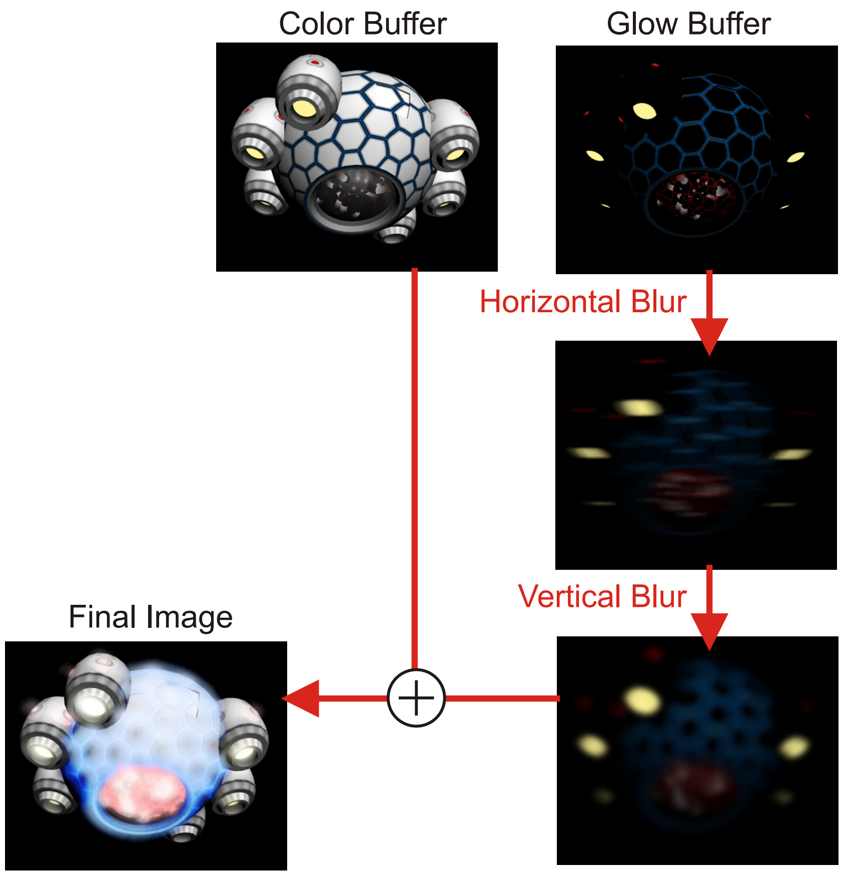


Fig. 4.3: Glow Effect Image Composition.

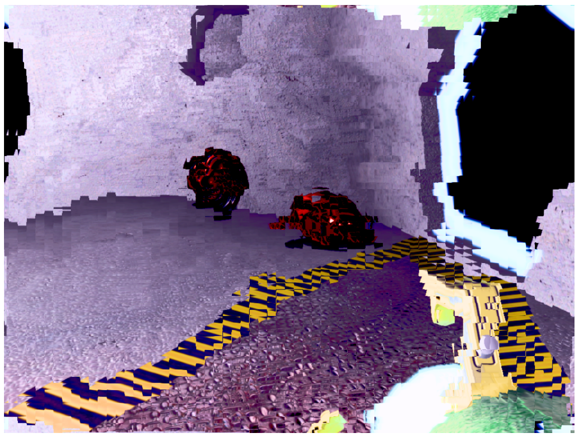
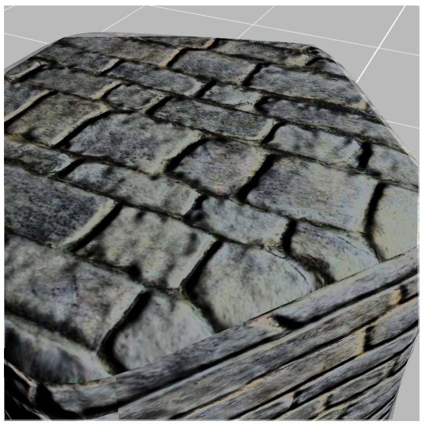
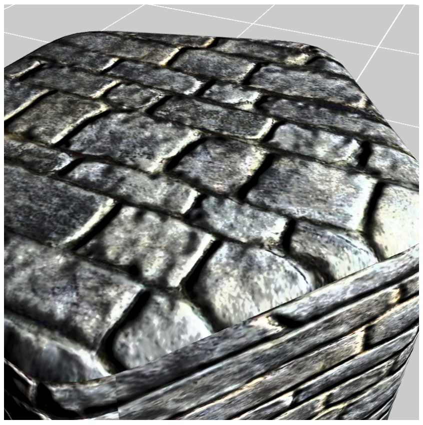


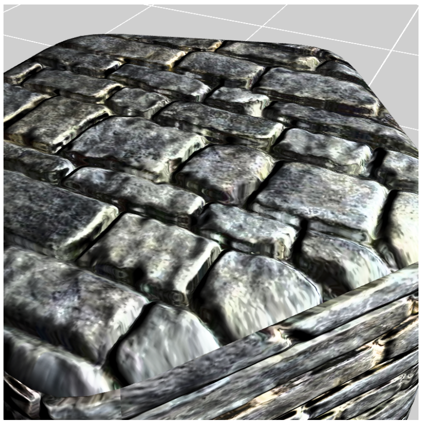
Fig. 4.4: Distortion Effect.



(a) Regular Rendering



(b) Normal Mapping



(c) Parallax Mapping

Fig. 4.5: The Effect of Normal and Parallax Mapping.

texel stores a normal vector at that point in tangent space. During lighting calculations in the pixel shader, the normal is sampled from the normal map and transformed into view space where lighting calculations take place.

Such normal maps can be generated from higher resolution meshes and UV-mapping them onto lower resolution versions of those meshes [13]. In Penta G we used traditional manual texture UV mapping inside the 3D content creation package, and generating normal maps via additional tools, such as the NVIDIA Normal Map Filter [51] and CrazyBump [15].

An extension to normal mapping is *Parallax Mapping* [34]. Additionally to the way lighting is calculated in the normal mapping technique, a height value is added which allows to calculate the offset by which the texel should be moved to give high frequency surfaces a parallax effect mostly visible when changing the view direction onto the surface. In such a way, depth differences on a surface can be rendered without increasing the geometric complexity. Figure 4.5 illustrates the effect of this technique.

4.6.3 Omnidirectional Shadow Mapping

During development of Penta G, several shadowing techniques were evaluated. Two major shadowing techniques exist: *shadow mapping* [67] and *shadow volumes* [17].

Shadow volumes use stencil buffering to track a rasterized pixel's entering and exiting a shadow umbra and deciding whether the shaded pixel lies within a shadow. This shadow umbra is formed by a shadow volume by extruding the silhouette of a shadow caster from the shadow casting light source (see Figure 4.6). The standard algorithm first renders the scene unshadowed to establish the depth buffer, then renders the shadow volume front and back-faces with different stencil modes so that each pixel receives a classification of whether it is shadowed or not. This second step nowadays can be done in one pass; older hardware required two passes for this step. Finally, the result of the previous steps are used to apply a shadow mask to the scene.

Shadow mapping is a class of techniques that render the scene's depth buffer or eye-view distances from the view of the light source, and using that information during normal scene rendering to decide whether a rendered pixel is inside the shadow or light by comparing its distance to the light source to the corresponding pixel in the previously rendered light-view rendering (see Figure 4.7).

Since shadow mapping is a texture-based technique, precision and aliasing problems exist which several techniques try to solve. *Perspective Shadow Mapping* [60] is a technique warping the light frustum, which is essentially the view frustum in the light-view render pass, and thus increasing effective depth resolution. An extended version, *Light-Space Perspective Shadow Mapping* [70], or *LiSPM* in short, has been developed that calculates a perspective transform in light space that effectively turns all lights into directional lights without changing the light direction, preventing some artefacts and singularities of regular Perspective Shadow Mapping. These techniques can be combined with calculating the minimum visible area the shadow map covers from the light and view frustum, and focusing the shadow map on this visible portion of the scene as suggested by Stefan Brabec [8].

Both shadow volumes and shadow mapping, including LiSPSM, have been implemented and evaluated in the pre-production prototype of Penta G. We decided against shadow volumes due to the softer shadow borders that shadow maps are capable of producing through techniques such as *Percent-*

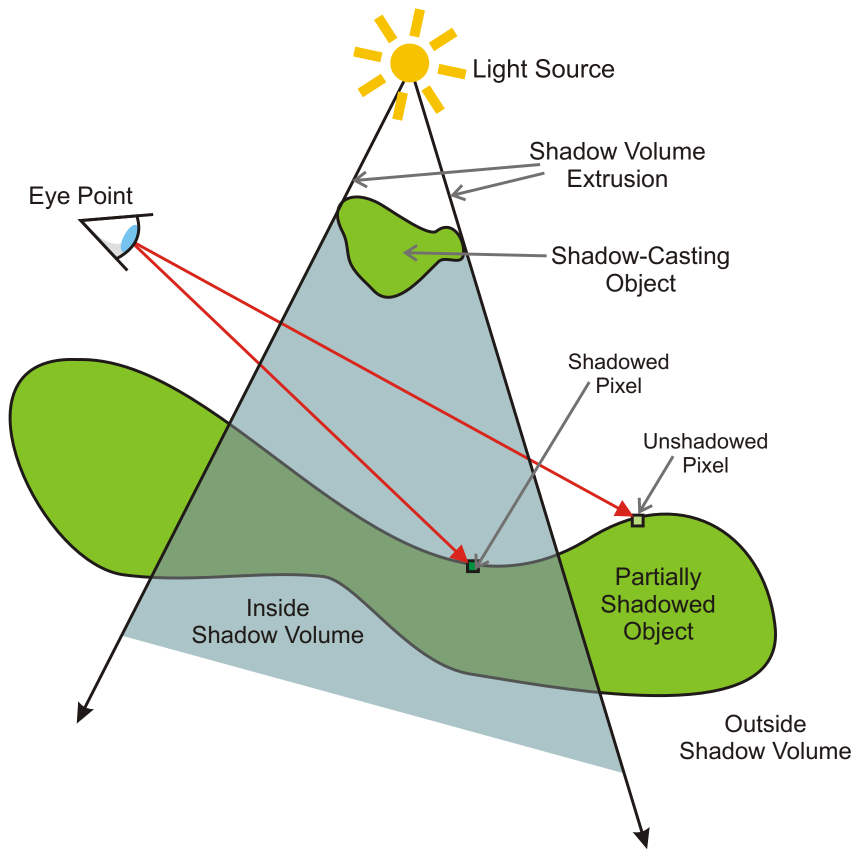


Fig. 4.6: Shadow Volume Principle.

age Closer Filtering [21]. Also shadow volumes would not have been able to achieve correct shadowing of finely detailed scene parts realized through alpha-testing such as grids or fences (Figure 4.8).

The final version of Penta G used *uniform shadow mapping*, as the quality benefit of Light-Space Perspective Shadow Maps was found to be lost again through the temporal aliasing that happens due to the recalculation of the shadow frustum in every frame. Also we did not focus the shadow map because it also leads to visible temporal aliasing artefacts where shadow borders change whenever the camera view varies. Other factors for the decision against any form of perspective shadow mapping were the relatively small benefit in indoor scenes with small local light sources, where the area covered by shadowmaps would in most cases be small anyway and thus the resolution benefit of such techniques smaller than in outdoor scenarios. Furthermore, shadow maps in Penta G are only updated when there is a significant change

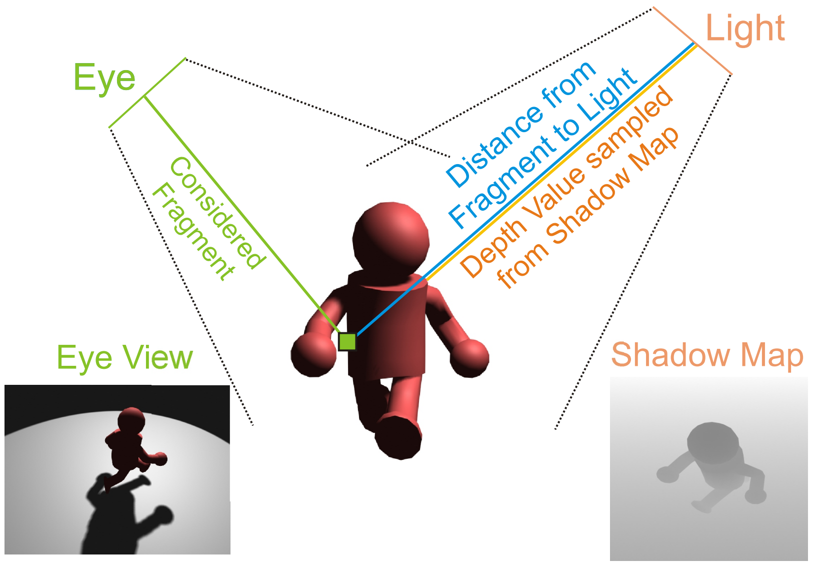

Eye
Light
Considered Fragment
Distance from Fragment to Light
Depth Value sampled from Shadow Map
Eye View
Shadow Map

Fig. 4.7: Shadow Mapping Principle. *The considered fragment is transformed from eye space into light space, where the distance comparison with the sampled shadowmap depth is made.*

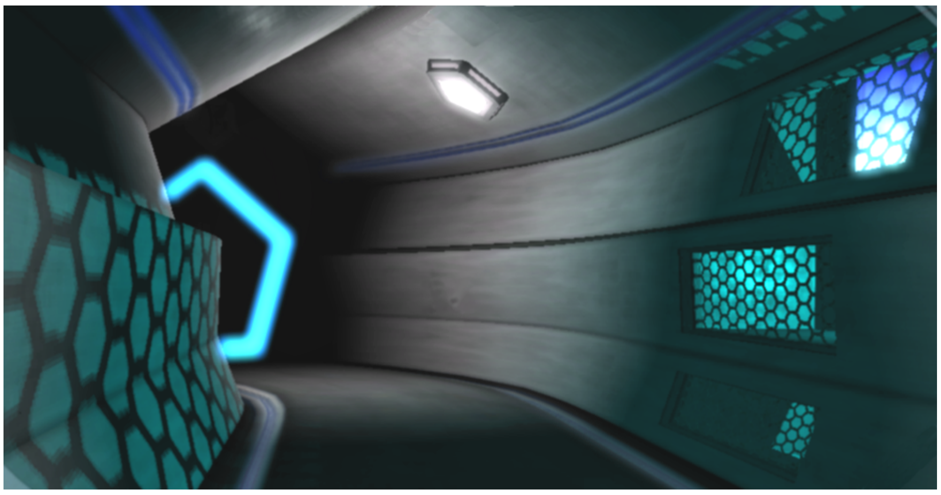


Fig. 4.8: Shadows cast through Alpha-Testing.

within a light source's radius, whereas focused shadow maps would require updates with every change of the camera view.

We implemented omni-directional shadow mapping by using a cubemap to cover the 6 directions from such a light source. The cubemap side frustums were adjusted to use a 90° field of view angle so that each cubemap side render seamlessly connects to the others. Due to the 6 directions, a light-view pass now takes 6 actual passes, one for each cubemap side. We also did not render the depth, but the linear fragment distance into these cubemap sides [25], utilizing floating point render target types for the cubemap.

Due to the cubemap render targets containing the linear distance, the comparison in the scene shadowing pass is trivial, as it's simply a comparison of the distance of the scene fragment to the light source compared to the distance stored in the shadowmap for that fragment – if the one in the shadowmap is less than the scene fragment distance, it means that there is a closer object to the light source obstructing the view from the light source to the scene fragment and thus casting a shadow on the scene fragment.

Our approach was optimized by not doing the cubemap render passes for light sources that have no influence on the visible portion of the scene. In order to save video memory, we also allocate only the shadow map render targets needed for the current view, which costs a bit of performance. Previously, we implemented a caching scheme which allocated fixed statically allocated render targets to the visible light sources, but this approach used more video memory than the dynamic allocation.

4.6.4 Coherent Hierarchical Culling

The research field of *visibility* intends to develop and analyze methods and algorithms to do more efficient or more accurate detection of visible portions of a scene in order to minimize the rendering work by avoiding rendering of invisible parts of the scene.

Occlusion queries have long been a feature of modern graphics processors. They provide a method of retrieving the number of visible portions of a series of render calls by counting the number of fragments that pass the Z test and Alpha test and are therefore visible. Due to the asynchronous nature of GPU-CPU intercommunication, the result of such queries cannot be retrieved immediately after issuing those render calls. Stalling all further rendering work on the CPU until the query result arrives, however, causes a performance loss in most cases, as no useful work is done while the CPU is waiting, effectively reducing or reversing the benefits of a more accurate occlusion detection. For example, issuing a query for each object in the scene, waiting for the result, and then render them depending on whether the result indicates if the object is visible, will in most cases be slower than simply rendering all objects without any queries. Utilizing spatial scene hierarchies is therefore necessary in order to cull large parts of the scene as early as possible. Unfortunately, this hierarchical approach adds additional necessary queries for interior non-leaf nodes, and thus can be even slower in the worst case. This naive hierarchical approach is called the *Hierarchical Stop-and-Wait* method [69].

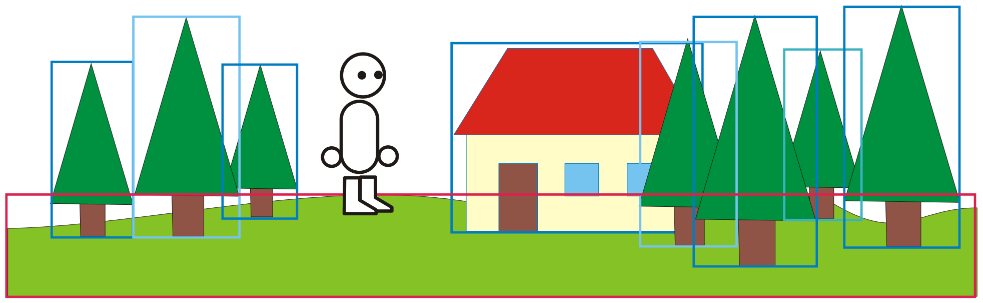
Coherent Hierarchical Culling is a technique that tries to countermeasure the CPU performance loss introduced by the Hierarchical Stop-and-Wait approach by performing useful calculations on the CPU during the time it waits for the result of an occlusion query, effectively interleaving multiple queries by queuing them and retrieving results while rendering actual geometry. This approach has the implication that it is difficult to integrate Coherent Hierarchical Culling into existing engines with well-defined rendering pipelines, such as Ogre3D [42]. Penta G, however, was developed with CHC in mind from the beginning.

The effect of frame coherence – which is the frame-by-frame similarity in a rendering system – is exploited in order to make assumptions about nodes in a scene hierarchy, and rendering their geometry while waiting for queries. For instance, if a node has been found visible in the previous frame, there is a high probability of the same node being visible in the current frame as well. These assumptions are verified during the render of a single frame, and can subsequently be used in the following frame.

In order to prevent occlusion query overhead introduced by having to issue queries for interior nodes in the scene hierarchy, query results are propagated upwards in a scene hierarchy, utilizing the fact that if a single child node in a spatial hierarchy is visible, its parent node is also visible. Thus, interior nodes get their visibility information indirectly by deducing visibility information from child nodes.

Coherent Hierarchical Culling necessitates the existance of a spatial scene bounding volume hierarchy. In Penta G, we opted for a semi-automatic approach where the top-level bounding volumes are defined by the level designer – these consist of axis-aligned bounding boxes called *view cells*. The actual entity bounding volumes are then automatically assigned to the view cells that contain them, including updating the hierarchy for moving objects.

Further optimization that is orthogonal to Coherent Hierarchical Culling was added by adding the possibility to link view cells together that are visible from each other, similar to the Visibility Portal Culling technique and similar to the mixed static/dynamic systems implemented in some state-of-the-art game engines for next-generation consoles such as the ones developed by Insomniac Games [29]. For the demonstration of the benefits of Coherent


(a) In *outdoor scenes*, the viewer's eye point typically is outside scene object bounding volumes.

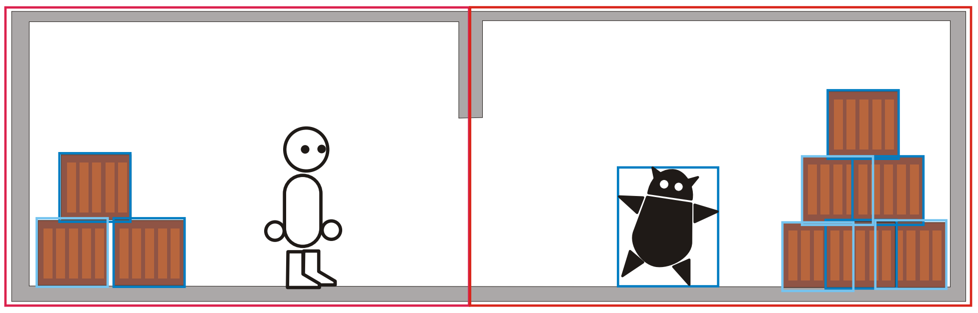

(b) In *indoor scenes*, rooms are part of level geometry and as such possess bounding volumes; the viewer is always inside at least one of them.

Fig. 4.9: Indoor/Outdoor Comparison. *Character Designs © by Ben Croshaw.*

Hierarchical Culling, this feature was deactivated though.

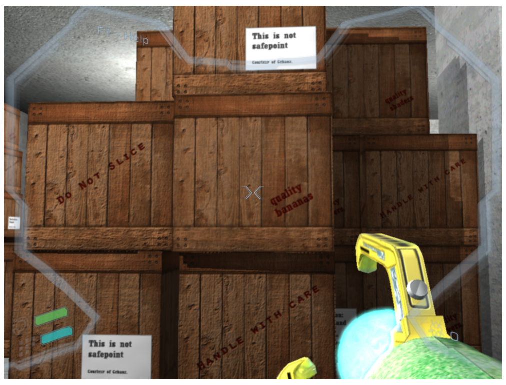
Further notice must be made to the fact that Penta G is an *indoor game*. The implication of this is that in contrast to the standard implementation and also in contrast to the pseudocode that is presented by Michael Wimmer and Jiří Bittner [69, 5], it is always the case that the eyepoint is inside a bounding volume (see Figure 4.9). This is an important difference because the bounding boxes of such room geometry would not be visible in a query because their polygon faces show outwards and thus would not generate fragments during the query. This is a generalized case of the near-clipped bounding volume case mentioned by Michael Wimmer and Jiří Bittner [69]. The authors suggest to set such nodes to visible without issuing an occlusion query. This leads to a modification of the standard algorithm for Coherent Hierarchical Culling. Listing 4.1 shows the modified algorithm used in Penta G.

A special CHC demonstration room has been included in the Penta G

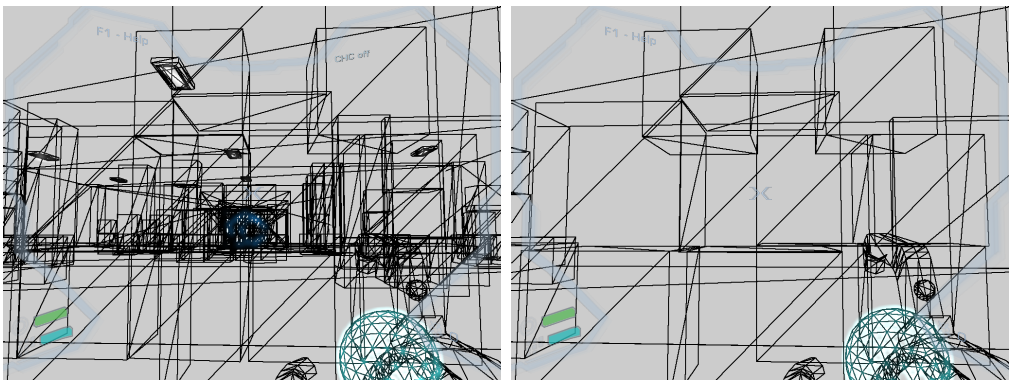
```
TraversalStack.Push(hierarchiy.Root);
while (!TraversalStack.Empty() ||
       !QueryQueue.Empty())
{
  //- PART 1: process finished occlusion queries
  while (not QueryQueue.Empty() &&
         (ResultAvailable(QueryQueue.Font()) || TraversalStack.Empty()))
  {
    node = QueryQueue.Dequeue();
    visiblePixels = GetOcclusionQueryResult(node);
    if (visiblePixels > VISIBILITY_THESHOLD)
    {
       PullUpVisibility(node);
       TraverseNode(node);
    }
  }

  //- PART 2: hierarchical traversal
  if (not TraversalStack.Empty())
  {
    node = TraversalStack.Pop();
    if (InsideViewFrustum(node))
    {
      wasVisible = node.visible && (node.lastVisited == frameID - 1);

      //- NOTE: modification of the original algorithm:
      if (IsEyePointInsideNode(node))
      {
        node.visible = true;
        node.lastVisited = frameID;

        PullUpVisibility(node);
        TraverseNode(node);
        continue;
      }

      leafOrWasInvisible = !wasVisible || IsLeaf(node);
      node.visible = false;
      node.lastVisited = frameID;
      if (leafOrWasVisible)
      {
        IssueOcclusionQuery(node);
        QueryQueue.Enqueue(node);
      }

      if (WasVisible) TraverseNode(node);
    }
  }
}
```

Listing 4.1: Coherent Hierarchical Culling as utilized in Penta G

demo mode, which illustrates the benefits of CHC. Figure 4.10 shows a wireframe comparison between Coherent Hierarchical Culling and regular frustum culling.



(a) The Scene

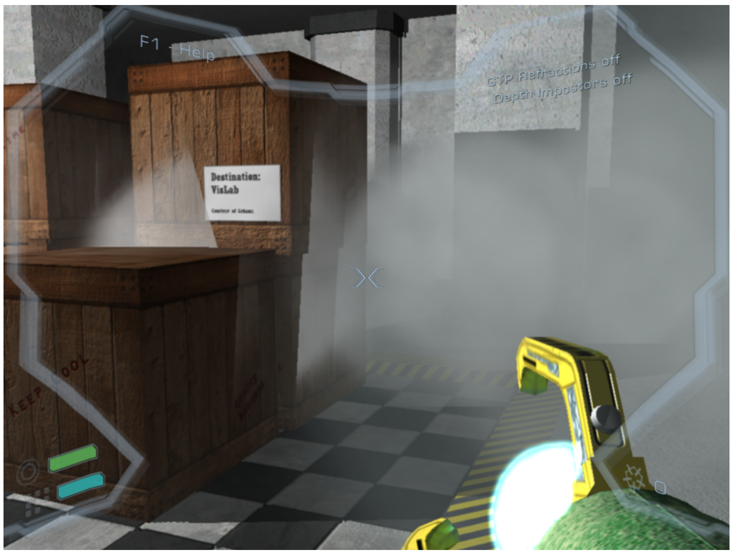


(b) Frustum Culling (c) Coherent Hierarchical Culling

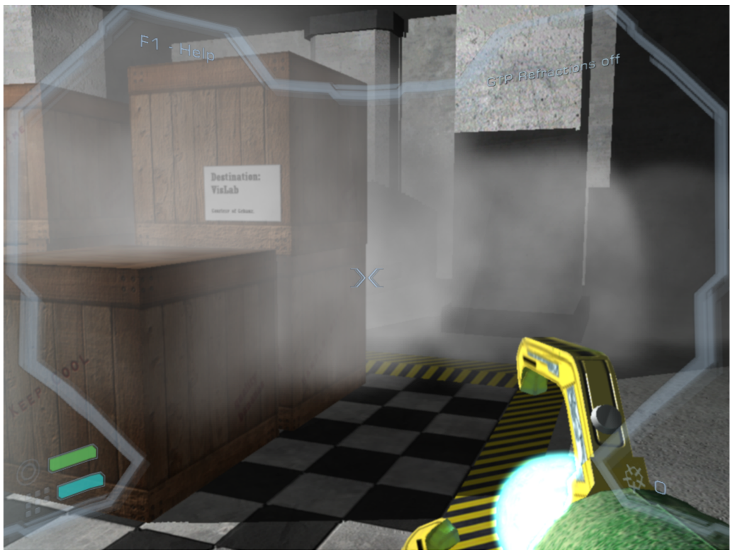
Fig. 4.10: CHC compared to traditional frustum culling.

4.6.5 Depth Impostors

Depth Impostors are another GameTools effect introduced by Tamás Ummenhoffer and László Szirmay-Kalos [63]. They extend the *billboard sprites* technique that is popular in realtime graphics and games. This technique utilizes a single quad polygon face that is aligned to the view transform in a way so that it always faces the camera. The term *impostors* refers to billboard sprites used to replace three-dimensional objects in the scene. Such impostors are frequently used for level-of-detail rendering schemes, or as components in *particle systems* to simulate atmospheric phenomena such as cloud or fog.



(a) Regular Billboard Sprites. *Note the clipping artefacts on the boxes.*

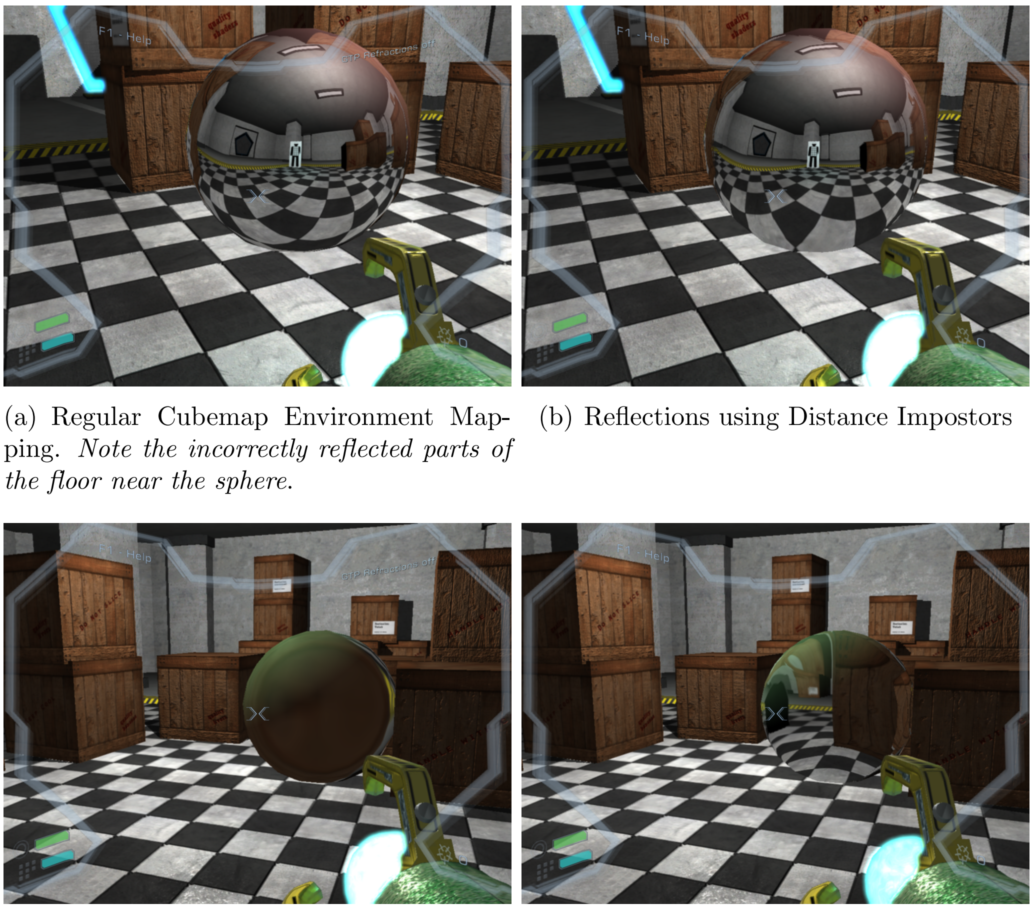


(b) Depth Impostors

Fig. 4.11: Depth Impostors compared to traditional billboards.

The rendering artefacts that occur with traditional billboards used in particle systems are caused by their two-dimensional nature. As a textured flat surface polygon, such billboards possess no depth. Intersections with other scene geometry produces highly visible clipping artefacts (See Figure 4.11(a)), and there is no lighting applied to the billboard. By storing depth information within the billboard texture, such impostors can correctly interact with the rest of the scene, as well as calculate lighting (see Figure 4.11(b)).

4.6.6 Reflections and Refractions with approximate Raytracing



(a) Regular Cubemap Environment Mapping. *Note the incorrectly reflected parts of the floor near the sphere.*

(b) Reflections using Distance Impostors

(c) Regular Refraction using an Environment Map.

(d) Refraction using Distance Impostors

Fig. 4.12: Comparison of Distance Impostors to Environment Mapping.

This method introduced by Szirmay-Kalos et al. [61] utilizes the distance rendering of a scene into an cube environment map to find a point hit by a ray fast, resulting in an approximate raytracing method that can be used for reflections and refractions. The benefit of this technique lies in the more physically exact way reflections and refractions appear especially for environments close to the refracting or reflecting object. This is shown in Figure 4.12.

The method can be scaled by specifiying the number of iterations for the calculation. We found that the best compromise between performance impact and exactness benefit was achieved with around two iterations. Higher iteration numbers had a very high performance hit which made the method still realtime-interactive for small-scale demo programs, but unsuitable for games. The benefits are mostly visible for scene parts close to the reflective or refractive object, while scene parts farther away have almost no visible difference to traditional cube environment mapping.

4.7 Comparison with requirements list

In Section 2.5, we developed a requirements list for the game engine. This section will evaluate and discuss to what extent each requirement has been reached.

4.7.1 Layered Design

Penta G offers an application and an engine layer on top of direct access to Direct3D 9, and functionality can be built on top of all of these levels. The developer has the opportunity to build upon each of these layers, depending on the technique to implement.

4.7.2 Extensible Rendering System

The GebauzEngine does not pre-define a rendering pipeline. The rendering pipeline is part of the application domain, in this case the Penta G application code, thus providing the application programmer with full access to how objects are rendered.

As discussed in Section 4.6.4, certain algorithms assume direct access to rendering, and Penta G has been developed with this requirement in mind.

4.7.3 Reducing Overengineering

The class structure of both the GebauzEngine and Penta G contain classes with a low level of inter-dependencies. Most of these classes can be replaced with own implementations. *Design Patterns* are used to simplify code and increase code reuse, and straightforward implementations were favoured over large complex implementations that have a higher risk of becoming unmaintainable.

4.7.4 Comprehensive Toolset

The core of Penta G development was the asset pipeline. Two main standalone tools – the SUX Model Editor and the GTP Level Editor – ensure a stable and comfortable asset workflow from any 3D content creation package to actual Penta G levels. A set of useful game object classes help in flexibly creating game scenes for Penta G.

4.8 Performance

Due to the long and complex rendering pipeline, Penta G performance testing proved rather complex. Some stages of the rendering pipeline proved to have different bottlenecks than others, and there was an occasionally happening frame-time peak due to some driver implementations with regards to shader-recompilation. We tested Penta G on the following configurations:

1. Intel Core 2 Duo 2.4 GHz with 2 GB DDR2-RAM and NVIDIA GeForce 7950 GT 512 MB
2. AMD Athlon 64 2.5 GHz with 1 GB DDR-RAM and NVIDIA GeForce 7900 GTX 512 MB
3. AMD Athlon 64 2.5 GHz with 1 GB DDR-RAM and ATI Radeon X1900XT 512 MB
4. Intel Core 2 Duo 2.67 GHz with 2 GB DDR2-RAM and NVIDIA GeForce 8800 GTX 640 MB
5. Intel Pentium-M 2 GHz laptop with 1 GB DDR-RAM and NVIDIA GeForce Go 6600 128 MB
6. Intel Core 2 Duo 2 GHz laptop with 2 GB DDR2-RAM and NVIDIA GeForce Go 7700 512 MB
7. AMD Athlon 64 2.0 GHz with 2 GB DDR-RAM and GeForce 7800GT 256 MB

On the higher specification machine configurations, Penta G ran on average at around 80 to 100 frames per second. The values shown in Figure 4.13 were measured running at a 1024x768 resolution using the *FRAPS* utility for framerate measurement.

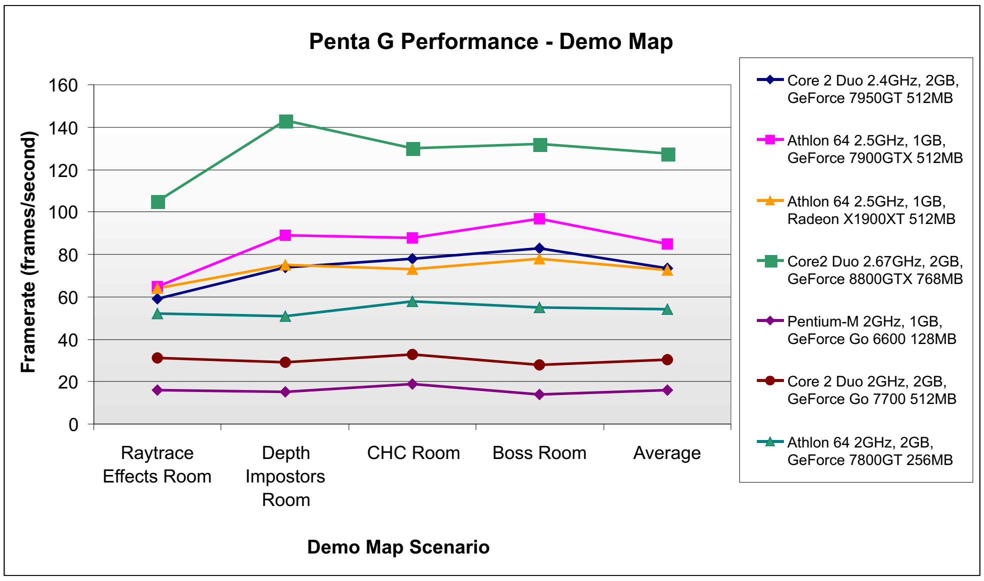


Fig. 4.13: Performance graph in the demo map.

Using NVPerfHUD for measurement on the NVIDIA-based configurations, we derived the following performance conclusions:

- The game is fillrate limited when a lot of large, almost screen-filling particles are visible, such as in the depth impostors room in the demo map.
- The game utilizes more than 400 MB of video RAM, thus making the 512 MB configurations much better performing than the configurations with less than 512 MB of video RAM.
- Certain effects introduce different limitations; for instance, the approximate raytracing effects will create a higher pixel shader load and make the game more pixel-shader limited.
- Due to the effective culling, occasionally there will be frame-time peaks (peaks of low framerate for short durations) due to a part of the scene instantly being visible, and coherent hierarchical culling taking a few frames until it can make use of frame coherence.

- On GeForce 6 class hardware (which includes also the GeForce 7 range of hardware), the driver tries to recompile shaders when it evaluates that such an action will result in higher performance, especially with shaders that utilize dynamic branching. This also leads to occasional peaks.
- On GeForce 8 class hardware and ATI Radeon X1000 class hardware, no such peaks were noticed.
- The GeForce 8 configuration was also tested under Windows Vista, and we also experienced smoother framerates with no or few peaks there.

Chapter 5

Penta G in Research

This section will discuss how Penta G was used for research purposes, as well as for research-related purposes such as promotion and public relations for the attraction of industry interest in the respective projects.

5.1 The EU Gametools Project

5.1.1 Introduction

The GameTools Project is a research project realized by the 6th Framework Programme of the European Union that brought together universities throughout Europe in an international collaborative effort to network leading computer graphics researchers with companies from the game and related industries [54].

The project lasted from 2004 to 2007 and incorporated the research work of several European universities and industrial partners [23]. It focused on algorithms in the fields visibility, global illumination, and geometry. A *Special Interest Group* was also maintained where interested companies could register and benefit from the research results by getting first-hand access to GameTools technology.

The game engine and demo game implemented in this thesis was used both as a demonstration application as well as in several research papers that were created as part of the GameTools project.

5.1.2 Penta G as a Demo Game

Penta G was originally tailored to be a demo game for the GameTools project with the goal of showing that the algorithms created in the GameTools project work in an actual game environment. The GameTools Project focused on offering its results to companies in the games industry, and therefore required a proof-of-concept that its technologies can work outside of a typical paper demo, which normally has a far smaller scale.

Three demo games have been created for this purpose: *Jungle Rumble*, *Have U Seen My Shadow*, and *Penta G*. The latter two have been completely developed from scratch for the GameTools Project. In total, Penta G development took 11 months with 3 full-time team members.

Penta G was designed to showcase, among other non-GTP realtime graphics techniques, three major GameTools effects and algorithms: Depth Imposters, Reflections and Refractions through approximate Raytracing, and Coherent Hierarchical Culling. A detailed description of these effects is given in Section 4.6.

Besides as a tech demo, Penta G has also been used for promotion and public relations for the GameTools Project. At the time of writing, there had been several opportunities where GameTools presented itself and its technologies. In all of these occasions, Penta G had been used to promote the project and to gain public interest.

Eurographics 2006

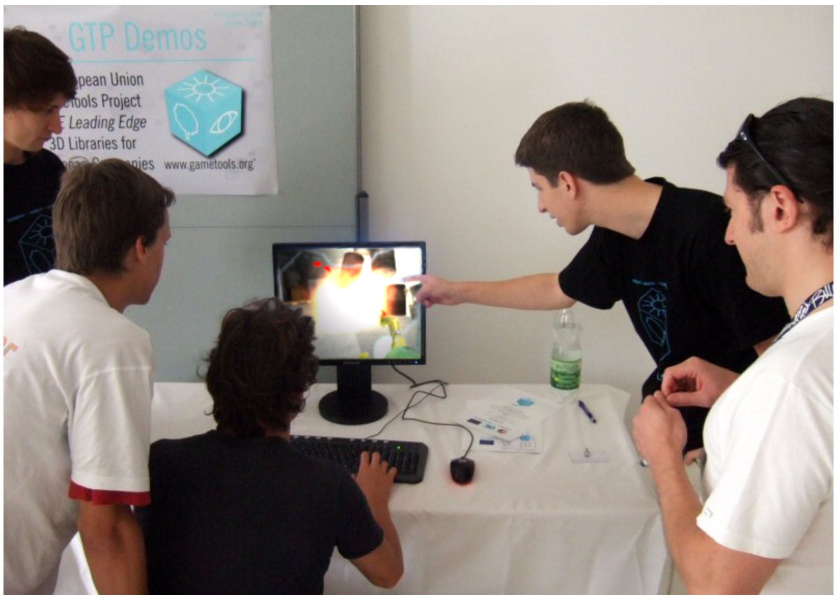


Fig. 5.1: The GameTools Demo Booth at EuroGraphics 2006.

At the Eurographics 2006 conference, the GameTools Project presented research results and demo games on the *Graphics meets Games* exhibition floor [20]. Some of the effects utilized were explained to interested visitors and computer graphics researchers from around the world using footage from Penta G.

RESFEST 2006

RESFEST is a festival for digitally produced short films, documentaries and animations, held in multiple cities world-wide each year [53]. In 2006, RESFEST visited 45 international cities, including Vienna, where the GameTools Project displayed its demo games including Penta G for the purpose of demoing future interactive graphics technologies.

GamesConvention Developer's Conference 2007

The GamesConvention Developer's Conference (GCDC) is held every year together with the GamesConvention expo. In 2007, the GameTools Project had the opportunity to present its results in the Science track. The presentation was held by Michael Wimmer [68], and Penta G was shown as a demo for the effects.

CeBIT 2008

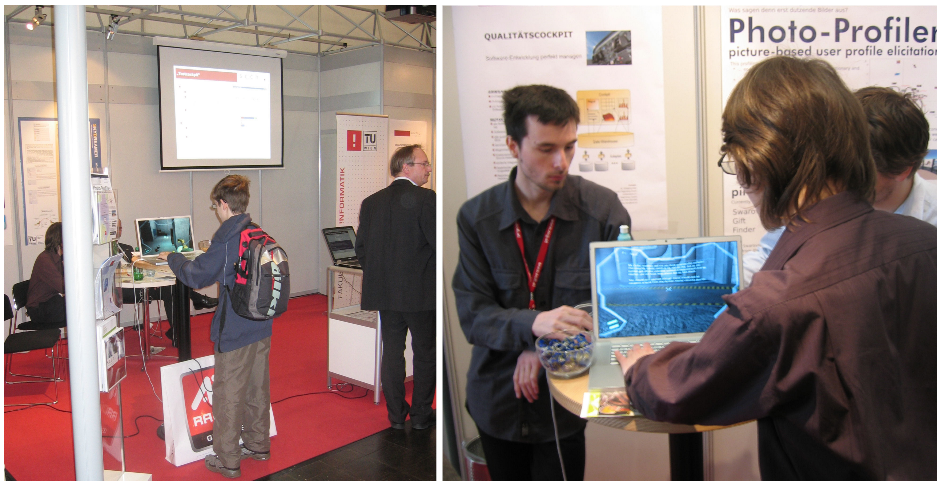


Fig. 5.2: Penta G presented at the CeBIT 2008.

The CeBIT is the world's largest computer expo, held yearly in Hannover Germany [11]. In 2008 the CeBIT was held from March 4th to March 9th and Penta G got a chance to be showcased for the GameTools Project at a booth sponsored by the Austrian Computer Society (OCG) [43].

5.1.3 Penta G in GTP Research

The scope of Penta G grew when in the second half of the project, the potential of using the Penta G engine for research was discovered. An early pre-production prototype of the engine had already been successfully used in some research papers and book articles [26], and the final version of the engine had been completely refactored into the form discussed in this thesis. It should be noted that the final version of both the game engine and the game itself have changed drastically compared to the version used in these papers.

5.2 The EU Crossmod Project

5.2.1 Introduction

The European Union Crossmod Project is a research project under the "Future Emerging Technologies Call" of the 6th framework programme exploring perceptual phenomena between visual and auditory senses (*crossmodal* or *bi-modal* phenomena) and how they can be used to create more convincing virtual environments such as games. The project goals are examining auditory, visual and audio-visual perception, develop new algorithms based on this newly gained knowledge, and evaluate them in target applications [16].

5.2.2 Eyetracking for Perception Experiments

One track of Crossmod research is measuring, quantifying and predicting human perception. For the measurement of human perception, eyetracker hardware can be used. Such hardware detects *gaze points*, which are the points that the user is looking at on a monitor screen. The specific hardware used at the Institute of Interactive Media Systems at the Vienna University of Technology uses an infrared camera to detect the eye pupils and calculates the gaze points according to metrics configured by calibration. It is connected to the computer via an USB interface, and communicates with software in a server/client structure via a local TCP/IP server. To simplify getting useful data from the eyetracking server software, a library called *eyeLib* has been developed that records gaze point data such as time stamp, gaze point coordinates and error metrics directly in a human-readable text file.

Penta G is currently being used for research in this area [16]. For this purpose, several modifications and features had to be added:

- Connecting to the eyetracker via the eyeLib.

- Recording a gameplay session, and replaying it at a later time.
- During replay, writing out framebuffer images and item buffer images to disk.
- Writing out a control file which *lyzer*, a tool developed in the project, can read to analyze gaze data and images side-by-side.

Item buffers are renderings of the scene where each relevant discrete scene object is rendered in an own distinct color, which serves as a unique ID for that object in the scene. In Penta G, the scene part granularity required was a per-entity one (although higher levels of granularity are still possible). Each entity receives a unique integer ID which identifies it, and that integer ID is also rendered into the item buffer by converting it into a 32-bit RGBA color. Subsequent queries of the item buffer translate back the RGBA color into the previous integer object ID.

Penta G is interesting for eyetracking experiments because it provides a game environment that is close to professionally produced commercial games, which means that research can be conducted in a practical environment close to reality. The asset pipeline and especially the scripting support means that people can easily create new perceptual experiment environments. Before Penta G, only commercial games have been used, so access to source code and assets were not available and the breadth of data that could be extracted was limited. With Penta G, data extraction can be done either via the framebuffer or from the internal data of the engine.

5.2.3 Spatial Audio Rendering

Another important part of the Crossmod project is the study of the relationships between auditory and visual senses, and how they influence each other [16]. Penta G was provided as a software toolkit to conduct this research. A *Crossmod Sound Engine* (XModSoundLib) has been developed and successfully integrated into Penta G by researchers of the Crossmod project, replacing Penta G's own DirectSound-based sound engine.

Chapter 6

Summary and Future Work

This chapter will summarize the work presented in this thesis and give an outlook on future work related to it.

6.1 Conclusion

In this thesis we presented a game and the underlying engine that was created for research purposes. The main goal was incorporating state-of-the-art technology in a framework that has a minimum of overengineering to prevent a loss of flexibility in order for researchers to be able to realize and experiment with ideas quickly. Another point of focus was the intention to keep close to real-world conditions; the game engine should perform and act like an engine created in industrial environments.

Furthermore, we explored formal requirements for game engines intended for research use. Providing a flexible and easy-to-use asset tool chain ensures that experimental and demonstrational applications can be created quickly without the hassle of integrating complicated asset transfer libraries, while still providing the developer with the liberty to implement own rendering methods and schemes. A content-based approach in developing the engine – that is, developing a game built on top of the engine in parallel to the engine itself – made it possible to perceive and analyze the important parts of the engine and helped getting a stronger focus on features with higher impact.

In the main chapters of the thesis we provided a complete reference of the software system which aids the reader in utilizing the engine in other applications.

Finally, a report was given on how the engine was already successfully used in research tasks as well as for purposes that aid research indirectly such as promotion and public relations.

6.2 Future Work

While Penta G served its purpose well and outgrew its original intent of just being a demo game, there are several points that could be improved in a future version of the engine.

During the development, the decision of what components would become part of the base engine framework, and which parts would stay on the application code domain was approached in a conservative way; due to the experience gained through the pre-production prototype engine, which had the entire scene management on engine framework level, we realized that carelessly putting software components into the engine domain could result in unmaintainable code.

On the other hand, this led to many software parts integrated into the application domain which could, at least partially, be implemented on engine framework level. Such systems would include the basic scene object base interface classes, the serialization system that reads scene object properties and gamestate management. Such systems would have reduced the amount of code necessary on application side, and thus made the game engine easier to maintain.

The game application side also grew evolutionary and as such, has some unnecessarily complicated constructs such as the multiple levels of game code spread over the Application class, Main GameState class, and actual Main Game Logic class. The rendering pipeline as implemented in the game rendering class could have been made more flexible via parameters so that there would be more graphics options such as alternatively switching between One-Pass Rendering using Multiple Render Targets and Multi-Pass Rendering without MRTs, but with the possibility of utilizing Multisampled Anti-Aliasing.

The engine framework's subsystems each have a central hub class that could be implemented as a singleton, instead of being forced to being members of a monolithic singleton. As the subsystems are free of inter-dependencies, the singleton creation order can be arbitrary.

With stronger in-engine serialization and factory pattern facilities, the game entity code on application domain could have been drastically simplified. Stronger utilization of object-oriented design patterns could also have reduced the amount of virtual methods implemented in each entity class in the entity hierarchy.

Finally, while the engine framework is fully documented via the Doxygen system [18] and its coding style well-defined, the application code does not follow the same style and documentation guidelines, and as such, is harder to read and understand. In a future game engine, the application code would require to follow either the game framework's Coding Style or define a specific one on its own.

At the time of writing, the game engine framework, called GebauzEngine, has already been refactored into a new version that incorporates many of the points outlined in this Future Work section. It remains to be seen if that new version also gains a root in research as the version described in this thesis did.

Appendix A

Importing Model Data to Penta G

This guide will explain how to get model data into the engine.

A.1 Prerequisites

Model Data import requires the following software:

- A 3D content creation package such as *Autodesk Maya* or *3ds Max*.
- The *OGRE XML exporter* plugin for that package.
- The *SUX Model Editor* tool.

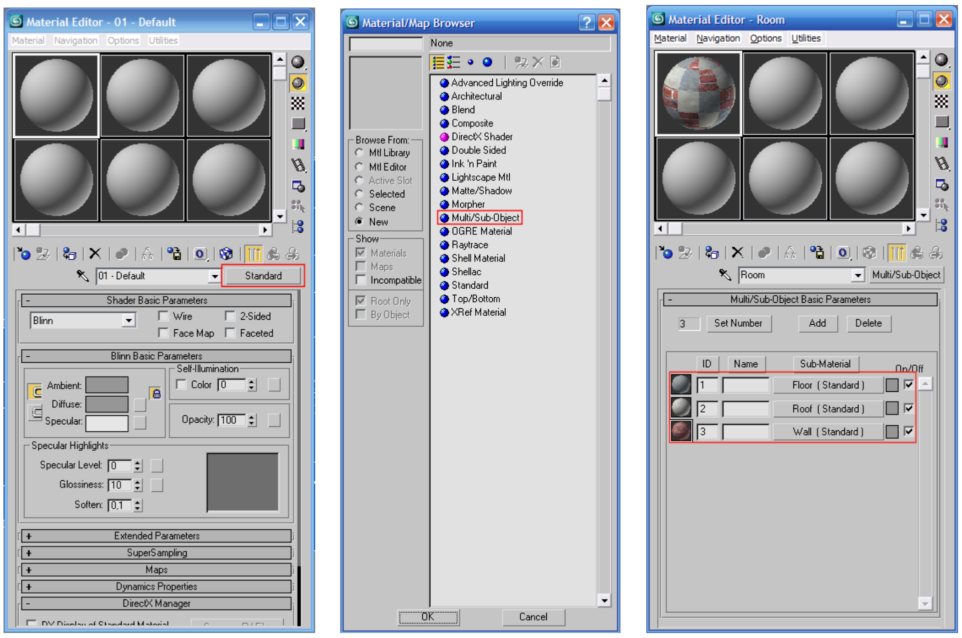
A.2 Creating the Model

The Penta G and GebauzEngine toolchain is independent of a specific 3D content creation package. The *OGRE XML* format is used for data exchange due to the number of export plugins available for various content creation applications, but it would be easy to implement support for a different model exchange format such as FBX or Collada. This guide uses *Autodesk 3ds Max* to illustrate the model import process.

When creating content in 3ds Max, it is important to note that only single meshes can be exported with the OGRE XML Exporter, so if a model contains several components, it is necessary to attach them together into a single mesh. This can be done by converting the mesh to an *Editable Poly*, and choosing *Attach* in the according rollout panel.

To create a single mesh that contains multiple materials, a *Multi/Sub-Object* type material must be created in the Material Editor (see Figure A.1). This is a type of material that contains multiple sub-materials, to which integer IDs are assigned. This *material ID* feature can be used to assign parts of a mesh to different sub-materials by selecting the polygons using the

Face Sub-Object selection mode, and assigning them specific material IDs in the rollout panel (see Figure A.2).



(a) Changing the Material Type (b) The *Multi-Sub-Object* Type (c) Sub-Materials

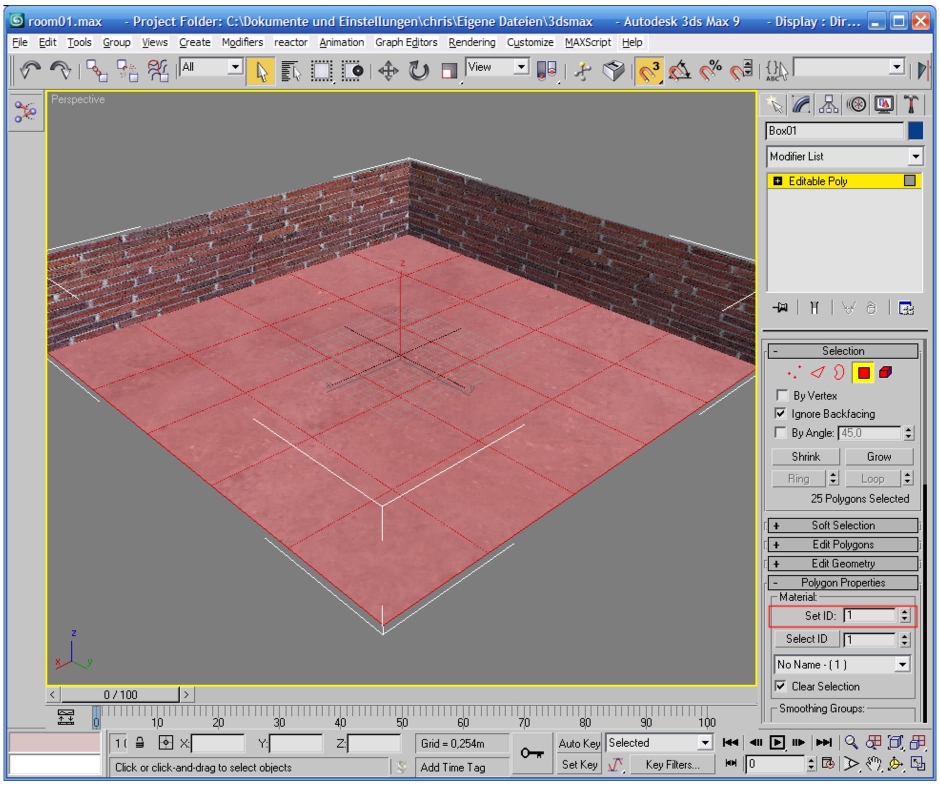
Fig. A.1: Creating a Multi/Sub-Object Material.

A.3 Exporting the Model

When the mesh is ready for export, the OGRE Exporter (Figure A.3) is used to export the model into the OGRE XML format, which is a simple and human-readable mesh format. Various Exporter versions may have different settings for export such as correcting the coordinate systems or rescaling. When the export process is done, a `.mesh.xml` file has been generated.

A.4 Tweaking and converting the Model

Using the SUX Model Editor, the OGRE XML file can now be imported. Inside the SUX Model Editor, various utility functions can be invoked, but the most necessary step is to assign correct materials to the submeshes, which are called *Material Groups* in the SUX Model Editor. Each Material


room01.max - Project Folder: C:\Dokumente und Einstellungen\chris\Eigene Dateien\3dsmax - Autodesk 3ds Max 9 - Display : Dir...

Box01

Editable Poly

By Vertex: ☐
Ignore Backfacing: ✓
By Angle: ☐ 45,0

25 Polygons Selected

Set ID: 1
Select ID: 1
No Name - (1)
Clear Selection: ✓

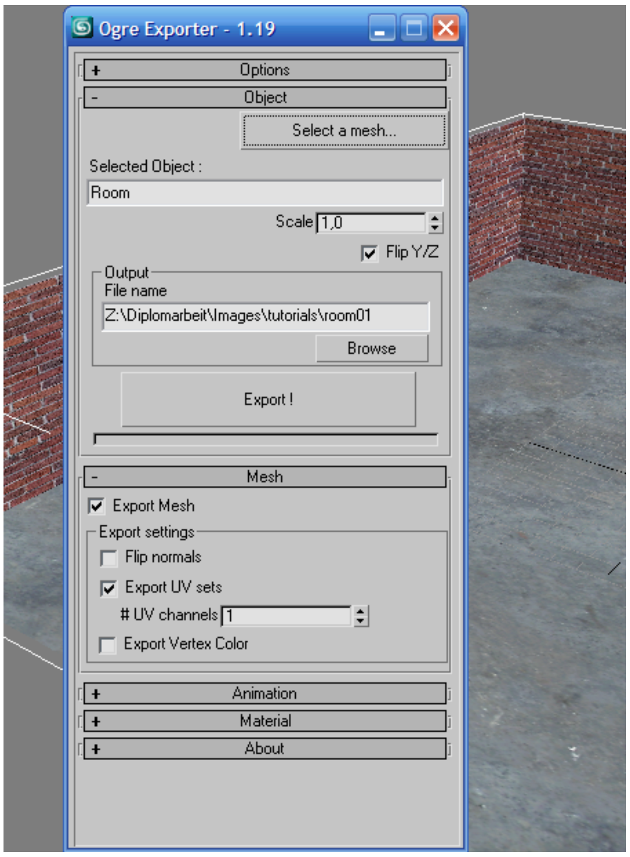
Grid = 0,254m

Click or click-and-drag to select objects

Fig. A.2: Setting Material IDs.

Group has a material, and the only necessary parameter for GebauzEngine is the material instance script, which contains a script adhering to the syntax described in Section 3.5.3. Since the SUX Model Editor runs on a different engine, it is also necessary to set the appropiate material for the display in the SUX Model Editor by setting a Material File (in this case we set it to "materials\solidobject.material") – this file has nothing to do with Penta G and is simply there for the SUX Model Editor to display the model correctly. Figure A.4 shows how to this process works. For tutorial purposes, we use textures and shaders already included in Penta G.

Note that for Penta G purposes, tangent space data needs to be generated, if any of the normal mapping or parallax mapping shaders are to be used. This can also be done by choosing "Generate Tangent Space Information" in the SUX Model Editor under "Operations". Furthermore, it is recommended especially for dynamic objects, but also for static level geometry, to add a physics mesh to be used as the geometry used for physics calculations. If none is set, the Ageia PhysX SDK tries to calculate a convex hull of the regular mesh, but this does not always yield desirable results. The most



Selected Object : Room
Scale 1,0
[✓] Flip Y/Z
Output
File name Z:\Diplomarbeit\Images\tutorials\room01
[✓] Export Mesh
Export settings
[ ] Flip normals
[✓] Export UV sets
UV channels 1
[ ] Export Vertex Color

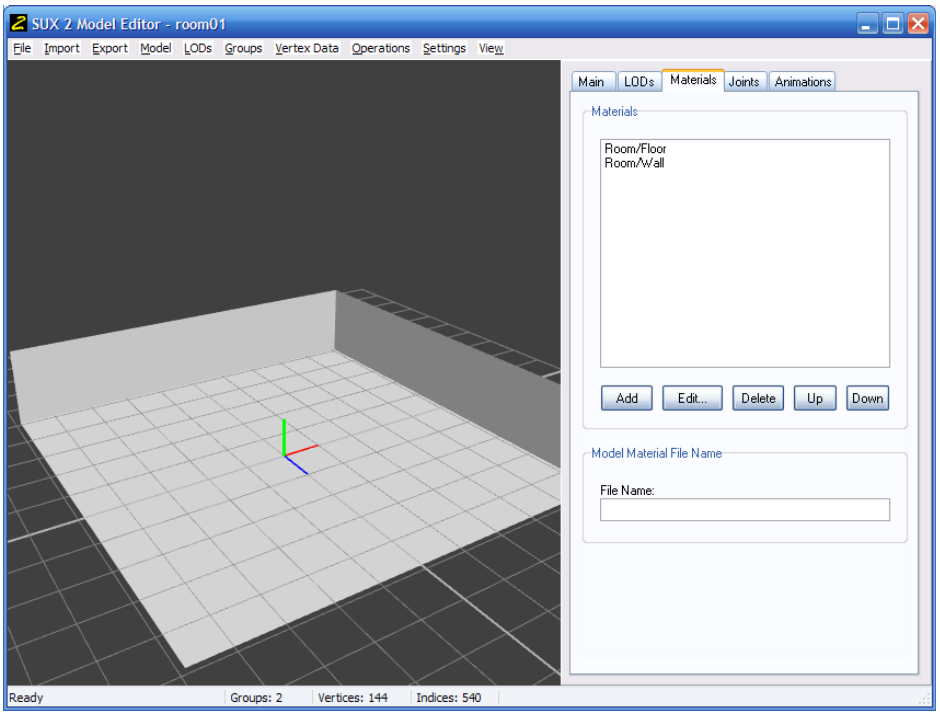
Fig. A.3: Using the OGRE Exporter.

control an artist has over how the model will act physically in the scene is by modelling a simple convex mesh to be used as a physics mesh. This is set as an additional "LOD" in the SUX Model Editor that is named "physics".

When all materials have been set and look correct in the SUX Model Editor, the model is ready for export into GebauzEngine's own propietary format. This is achieved by simply saving the model in the SUX Model Editor.

A.5 Loading the Model in Penta G

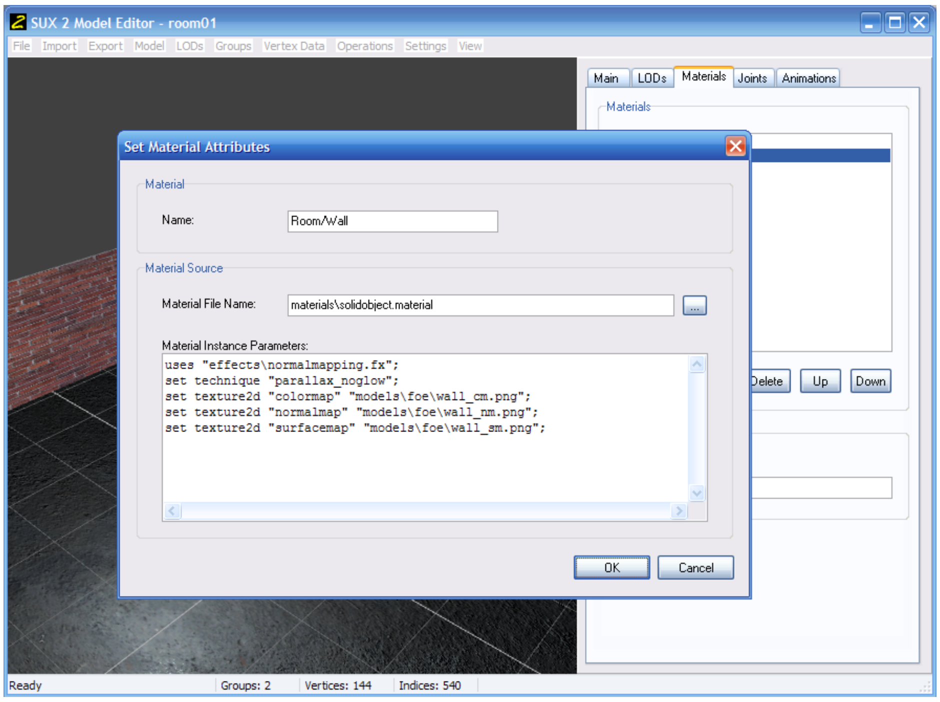
Now, several methods exist to use the model in Penta G. One could simply load the model via program code in Penta G, which means invoking the `LoadModel()` method of the `Model` class. This loads the model on application level, and handles material instance scripts via the `MaterialLoader` class. On GebauzEngine level, the `GxModelManager` class derived from


SUX 2 Model Editor - room01
File Import Export Model LODs Groups Vertex Data Operations Settings View
Main LODs Materials Joints Animations
Materials
Add Edit... Delete Up Down
Model Material File Name
File Name:
Ready

Room/Floor
Room/Wall

Groups: 2 Vertices: 144 Indices: 540

(a) Model after importing


SUX 2 Model Editor - room01
File Import Export Model LODs Groups Vertex Data Operations Settings View
Main LODs Materials Joints Animations
Materials
Set Material Attributes
Material
Material Source
...
Delete Up Down
OK Cancel
Ready

Name: Room/Wall

Material File Name: materials\solidobject.material

Material Instance Parameters:

```
uses "effects\normalmapping.fx";
set technique "parallax_noglow";
set texture2d "colormap" "models\foe\wall_cm.png";
set texture2d "normalmap" "models\foe\wall_nm.png";
set texture2d "surfacemap" "models\foe\wall_sm.png";
```

Groups: 2 Vertices: 144 Indices: 540

(b) Setting materials

Fig. A.4: Importing and setting materials.

the resource manager template class and can also load the model file, but since the material instance script loader is implemented on application level, the programmer would have to provide their own material loader.

Finally, the model can simply be used by dropping it into the *Penta G Level Editor* and use them in various ways.

Appendix B

Building a Level for Penta G

The following guide describes how to create a simple level for Penta G and demonstrates all the necessary features.

B.1 Prerequisites

This guide assumes the following prerequisites and requirements:

- Level model data, converted into the engine's own format using the *SUX Model Editor*.
- The *GTP Level Editor*.
- *Penta G* for testing the level.

B.2 Introduction to the Level Editor

The GTP Level Editor is a standalone application that is designed to create and edit maps for Penta G. It has been programmed in Delphi, which is a programming environment that significantly simplifies Windows GUI creation. Due to the different programming environment, the GTP Editor runs on a different engine called the SUX Engine, which itself builds on OpenGL. The SUX Engine was chosen as the basis for the editor because several other tools – some of which are not Penta G-related – have already been written in it. The Model Editor also uses the same framework.

This choice has some implications such as that the GTP Editor does not support the Penta G material system, but implements its own and provides some own default materials for map previewing. This might cause the level to look different in the editor, therefore testing the level in the actual game is important to achieve the desired visual look.

B.3 Creating Static Level Geometry and Lights

It is a good idea to start with the static level geometry, which makes up the visible portions of the level that cannot be moved around, but still are passively affected by physics in that dynamic objects collide and interact with them. The entity type for this task is the `EnvironmentModel`, which takes a model as its main parameter. Using the level editor, we can insert environment models into the scene and place them where we wish to.

In order to have correct lighting, we need to add light sources to the scene. Light sources can either be plain lights or shadow casters, and a variety of parameters can be adjusted.

B.4 Making the Level work in Penta G

After creating the static level geometry, the level cannot be viewed in Penta G yet. The level designer still needs to create view cells in the editor, which will define the spatial partitioning of the scene. Every entity in the level, even initially invisible ones, must be enclosed by at least one viewcell.

And finally, a starting position for the player must be set. This can be done by creating a `StartingPosition` entity in the level. Multiple such entities can be created, but the one that is used to spawn the player must be set in the editor in the “Edit Attributes” menu command in the “Level” menu. Figure B.1 shows the result, which can be seen as a minimal working level.

B.5 Creating Dynamic Level Geometry

Dynamic level geometry are props and objects that can be moved around and are dynamically affected by physics. They are implemented using the `InteractiveObject` entity type and are otherwise similar to Environment Models in that they also require a model as their main parameter. Like Environment Models, they are simply placed in the scene using the level editor (see Figure B.2).

B.6 Placing AI Controlled Objects and Pickups

AI Controlled Objects such as enemy entities are not placed via the editor, but they are spawned by the `SpawnPoint` entity. These spawn points can be set to emit AI controlled objects in a certain frequency. They do not spawn enemies automatically, however, but need to be triggered.

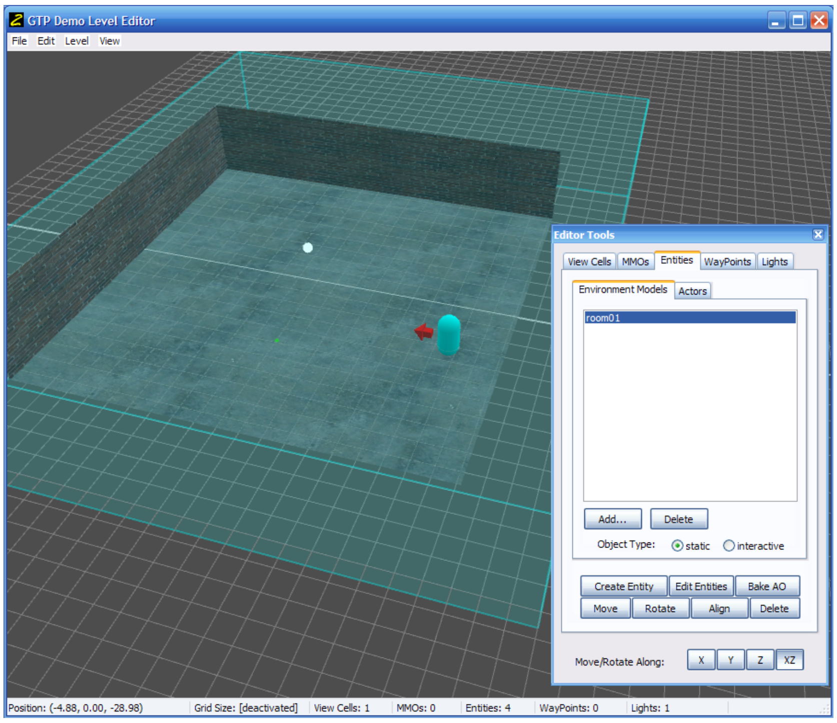

GTP Demo Level Editor
File Edit Level View
Editor Tools
View Cells MMOs Entities WayPoints Lights
Environment Models Actors
room01
Add... Delete
Object Type: static interactive
Create Entity Edit Entities Bake AO
Move Rotate Align Delete
Move/Rotate Along: X Y Z XZ
Position: (-4.88, 0.00, -28.98) Grid Size: [deactivated] View Cells: 1 MMOs: 0 Entities: 4 WayPoints: 0 Lights: 1

Fig. B.1: A minimal working level.

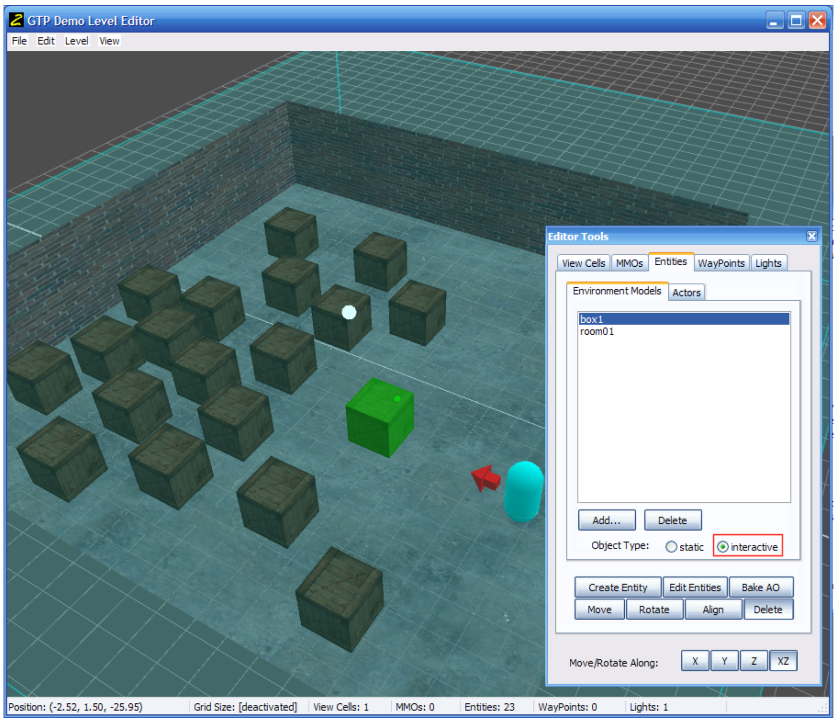
In order to trigger spawning of enemies, we can place an Action entity in the level. This action entity should be set to the type "Trigger Entity", with the Target entity being the spawn point created. Now when the action gets executed, it will send the "Trigger Entity" command to the spawn point, which in turn will start emitting enemies. In order to execute the action, we need to bind it to a certain event, which can be a `TouchField` entity's "OnEnter" event, or the Starting Point entity's "On Start Level" event. The latter will cause the spawn point to emit enemy entities as soon as the level starts.

Pickups are placed as `BonusItem` entities. Before it is usable, the type of pickup (such as a weapon or a health pickup) must be set. When these are added as in Figure B.3, the level becomes actually playable.

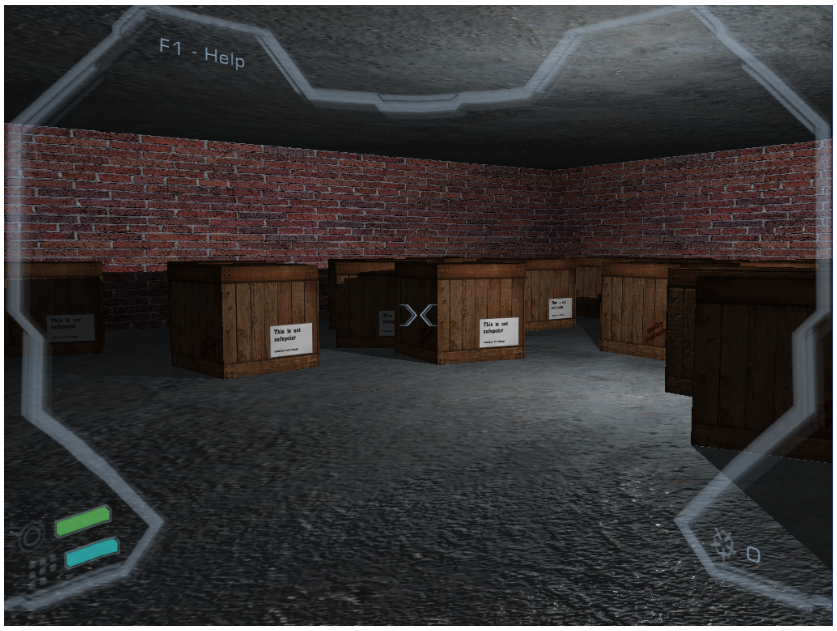
B.7 Adding a simple Trigger Script

Scripting is achieved by setting *Action* entities, which provide predefined action types that can be applied to entities. A more detailed description of actions and scripting is given in Section 4.4.3

The following steps will demonstrate how to script a trigger field that will

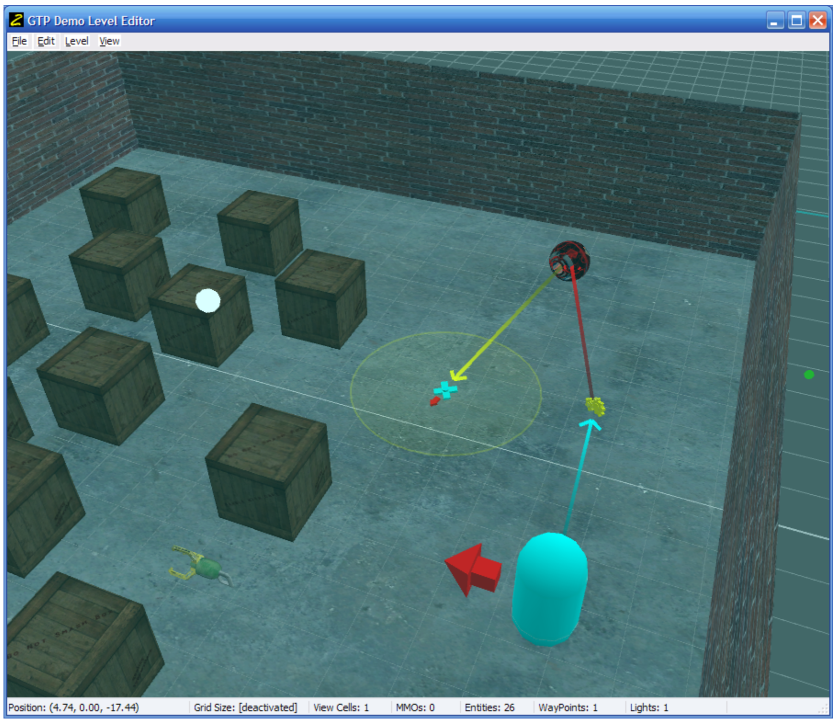


(a) Scene in the Editor



(b) Scene in Penta G

Fig. B.2: Adding interactive objects.



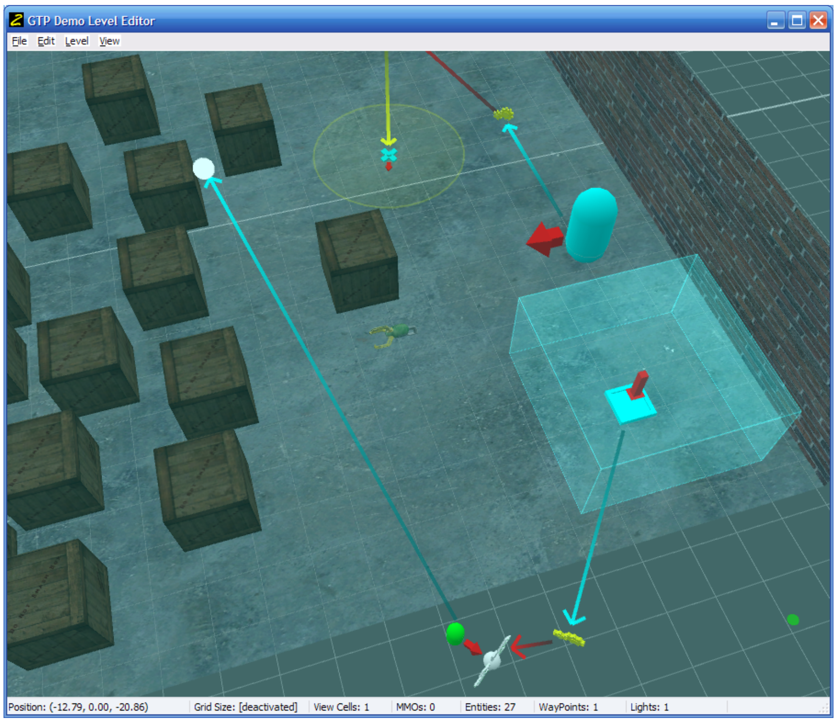
Position: (4.74, 0.00, -17.44) Grid Size: [deactivated] View Cells: 1 MMOs: 0 Entities: 26 WayPoints: 1 Lights: 1

Fig. B.3: Spawnpoint, Bonus Pickup and Trigger Actions.

start making the only light in the scene flicker as if it were defective:

- First, a `TouchField`, an `Action` and a `LightController` entity must be placed in the level.
- The light controller is set to inactive, and its *light animation* type is set to “Defective Flicker (even on-off time)”
- Then the touch field’s “On Enter” event is connected to the action.
- The action’s type is set to “Set Active” and the light controller is set to be one of its targets.

The result will look like in Figure B.4. Note how the arrows visualize the scripted connections between entities. Now when the level is executed in Penta G, when the player enters the touch field area, the light in the room will start flickering.



Position: (-12.79, 0.00, -20.86) | Grid Size: [deactivated] | View Cells: 1 | MMOs: 0 | Entities: 27 | WayPoints: 1 | Lights: 1

Fig. B.4: A Simple Script.

B.8 Testing in Penta G

Finally, we can test the level in Penta G by setting it as the start level in the configuration file. The level should start, and the player should be able to pick up a weapon. When the player enters the trigger box we set, the light should start flickering, confirming that the functionality we created is working. For debugging purposes, everytime an action is triggered, it is also output in the log, so the level designer can see if the actions are actually executed.

List of Figures

List of Tables

List of Listings

Bibliography

[1] 3Dfx Interactive, Inc., San Jose, CA, USA. *Glide 2.2 Programming Guide. Programming the 3Dfx Interactive Glide^{TM}Rasterization Library 2.2.* `http://www.gamers.org/dEngine/xf3D/glide/glidepgm.htm`.

[2] 3Dfx Interactive Inc. Defunct company, 1994-2002.

[3] Matthias Bauchinger. YARE2. `http://www.yare.at`.

[4] Matthias Bauchinger. *Designing a Modern Rendering Engine. Design Decisions and Implementation Details.* Vdm Verlag Dr. Müller, 2008.

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

[6] David Blythe. Windows Graphics Overview. In *WinHEC 2005*, 2005.

[7] Boost C++ Libraries. `http://www.boost.org`.

[8] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. Practical shadow mapping. *Journal of Graphics Tools: JGT*, 7(4):9–18, 2002.

[9] Bullet physics library. `http://www.bulletphysics.com`.

[10] Iain Cantlay. Taming Vertex Data Using C++ Templates. *Game Developer Magazine*, pages 34–41, October 2003.

[11] CeBIT – Centrum der Büro– und Informationstechnik (Centre of Office and Information Technology). `http://www.cebit.de`.

[12] NVIDIA Cg Toolkit. `http://developer.nvidia.com/object/cg_toolkit.html`.

[13] Paolo Cignoni, Claudio Montani, Claudio Rocchini, and Roberto Scopigno. A General Method for Recovering Attribute Values on Simplified Meshes. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 59–66. IEEE, 1998.

[14] Coin3D. `http://www.coin3d.org/`.

[15] Crazy Bump. `http://www.crazybump.com`.

[16] CROSSMOD – Cross–modal perceptual Interaction and Rendering. `http://www.crossmod.org`.

[17] Franklin C. Crow. Shadow Algorithms for Computer Graphics. *Computer Graphics (SIGGRAPH '77 Proceedings)*, 11(2), Summer 1977.

[18] Doxygen. `http://www.doxygen.org`.

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

[20] Eurographics 2006 Graphics meets Games Exhibition. `http://www.cg.tuwien.ac.at/events/EG06/program-graphicsmeetsgames.php`.

[21] Randima Fernando. Percentage-closer soft shadows. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 35, New York, NY, USA, 2005. ACM.

[22] Fraunhofer Society. `http://www.fraunhofer.de`.

[23] EU GameTools Project. `http://www.gametools.org`.

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

[25] Philipp Gerasimov. Omnidirectional Shadow Mapping. In *GPU Gems – Programming Techniques, Tips and Tricks for Real-Time Graphics*, pages 193–203. Charles River Media, 2004.

[26] Markus Giegl and Michael Wimmer. Queried Virtual Shadow Maps. In Wolfgang Engel, editor, *ShaderX 5 – Advanced Rendering Techniques*, volume 5 of *ShaderX*. Charles River Media, December 2006.

[27] Henri Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6):623–629, June 1971.

[28] Herbert Grasberger. Introduction to stereo rendering. Technical report, Vienna University of Technology, Institute of Computer Graphics, 2008.

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

[30] Donald Hearn and M. Pauline Baker. *Computer Graphics with OpenGL, 3rd Edition.* Prentice Hall, 2003.

[31] DevIL – Developer's Image Library. http://openil.sourceforge.net.

[32] Java3D. http://java3d.dev.java.net/.

[33] Greg James and John O'Rorke. Real-time glow. In *GPU Gems – Programming Techniques, Tips and Tricks for Real-Time Graphics*, pages 343–362. Charles River Media, 2004.

[34] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed Shape Representation with Parallax Mapping. In *International Conference on Artifical Reality and Telexistance 2001*, 2001.

[35] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, 1st Edition.* Prentice Hall, Englewood Cliffs, NJ, USA, February 1978.

[36] Khronos Group. http://www.khronos.org.

[37] Hermann Kopetz. *Real-Time Systems.* Kluwer Academic, Boston, MA, USA, 1997.

[38] Moving Picture Experts Group. http://www.chiariglione.org/mpeg/.

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

[40] NVIDIA Corporation, Santa Clara, CA, USA. *GPU Programming Guide.* http://developer.nvidia.com/object/gpu_programming_guide.html.

[41] NVIDIA SceneGraph. http://developer.nvidia.com/object/nvsg_home.html.

[42] Ogre3D. http://www.ogre3d.org.

[43] Austrian Computer Society (OCG). http://www.ocg.at.

[44] SGI Open Inventor. http://oss.sgi.com/projects/inventor/.

[45] OpenGL Architecture Review Board. http://www.opengl.org.

[46] OpenSceneGraph. http://www.openscenegraph.org.

[47] OpenSG. http://opensg.vrsource.org.

[48] Performer. http://www.sgi.com/products/software/performer.

[49] Ageia PhysX SDK. http://www.ageia.com.

[50] Pixar Renderman. https://renderman.pixar.com/.

[51] NVIDIA Normal Map Filter. http://developer.nvidia.com/object/photoshop_dds_plugins.html.

[52] Gerhard Reitmayr, Chris Chiu, Alexander Kusternig, Michael Kusternig, and Hannes Witzmann. iOrb – Unifying Command and 3D Input for Mobile Augmented Reality. In *IEEE VR 2005 Workshop on New Directions in 3D User Interfaces*, 2005.

[53] RESFEST 10. http://www.resfest.com, 2006.

[54] Mateu Sbert and Jordi Palau. GameTools: Advanced Tools for Developing Highly Realistic Computer Games. In *IVth ITRA World Conference, Alicante*, 2005.

[55] D. Schreiner, editor. *OpenGL® Reference Manual: The Official Reference Document to OpenGL, Version 1.4*. Addison Wesley, 2004.

[56] Silicon Graphics, Inc. http://www.sgi.com.

[57] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL® Programming Guide: The Official Guide to Learning OpenGL, Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[58] Peter-Pike Sloan. Using Direct3D10. *Eurographics 2006 Graphics meets Games Track*, 2006.

[59] Sebastien St-Laurent. *The Complete Effect and HLSL Guide*. Paradoxal Press, Redmond, WA, USA, 2005.

[60] Marc Stamminger and George Drettakis. Perspective Shadow Maps. In John Hughes, editor, *SIGGRAPH 2002 Conference Proceedings*, Annual Conference Series, pages 557–562. ACM Press/ ACM SIGGRAPH, 2002.

[61] Laszlo Szirmay-Kalos, Barnabas Aszodi, Istvan Lazanyi, and Matyas Premecz. Approximate Ray-Tracing on the GPU with Distance Impostors. *Computer Graphics Forum*, 24(3):695–704, 2005.

[62] Eric Haines Tomas Akenine-Möller. *Real-Time Rendering, Second Edition.* A.K. Peters, Ltd., Natick, MA, USA, 2002.

[63] Tamás Ummenhoffer and László Szirmay-Kalos. Real-Time Rendering of Cloudy Natural Phenomena with Hierarchical Depth Impostors. In *Eurographics 2005 Short Papers*, pages 65–68, 2005.

[64] Randima Fernando und Mark J. Kilgard. *The Cg Tutorial. The Definite Guide to Programmable Real-time Graphics.* Addison-Wesley Longman, 2003.

[65] UrbanViz – Real-time Rendering of Urban Environments. `http://www.cg.tuwien.ac.at/research/vr/urbanviz/`.

[66] Intel VTune Performance Analyzer. `http://www.intel.com`.

[67] Lance Williams. Casting Curved Shadows on Curved Surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 270–274, August 1978.

[68] Michael Wimmer. GameTools: Advanced Rendering Effects for Next-Gen Engines. *GamesConvention Developer's Conference*, 2007.

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

[70] Michael Wimmer and Daniel Scherzer. Robust Shadow Mapping with Light Space Perspective Shadow Maps. In Wolfgang Engel, editor, *ShaderX 4 – Advanced Rendering Techniques*, volume 4 of *ShaderX*. Charles River Media, March 2006.

Acknowledgements

"There has never been a cat
Who couldn't calm me down
By walking slowly
Past my chair."
– Rod McKuen

I would like to send my most sincere thanks to the following people:

The *Computer Graphics Club* at the Institute of Computer Graphics for moral support and motivation and especially *Commander Kustl* for letting me use his Toolchain and for doing his part on Penta G.

Georg Semanek for providing ideas, designs and assets to Penta G.

The *Institute for Computer Graphics* for providing the advisory support for my thesis and using Penta G for further research.

The *Faculty of Computer Science* for granting me a generous scholarship for this thesis, and *Michael Wimmer* and *Werner Purgathofer* for supporting my application for the scholarship.

And finally, my family:

My mother *Shiow-Yuh Chiu* who always supported me in times of stress and provided me with the freedom to choose and to take on things at my own pace.

My late father *Shang-Yi Chiu* (1942-1997) who – with his ETH Zürich engineering background and videogame enthusiasm – has influenced my way to computer science, digital games and the games industry in many ways.

My cat *Niaomi* who passed away shortly before the completion of this thesis and has always been just there at my side to keep me company.