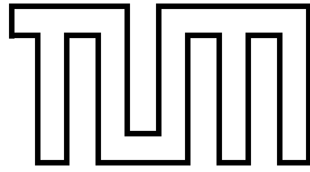


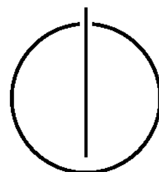
FAKULTÄT FÜR INFORMATIK

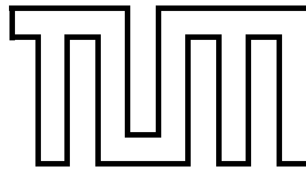
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor Thesis in Computer Science

Multi-Tile Terrain Rendering with OGL/Equalizer

Andreas Kirsch





FAKULTÄT FÜR INFORMATIK

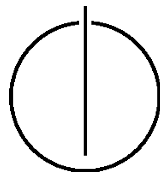
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor Thesis in Computer Science

Multi-Tile Terrain Rendering with OGL/Equalizer

Multi-Tile Terrain Rendering mit OGL/Equalizer

Author:	Andreas Kirsch
Supervisor:	Prof. Dr. Westermann
Advisor:	Christian Dick
Date:	October 15, 2009



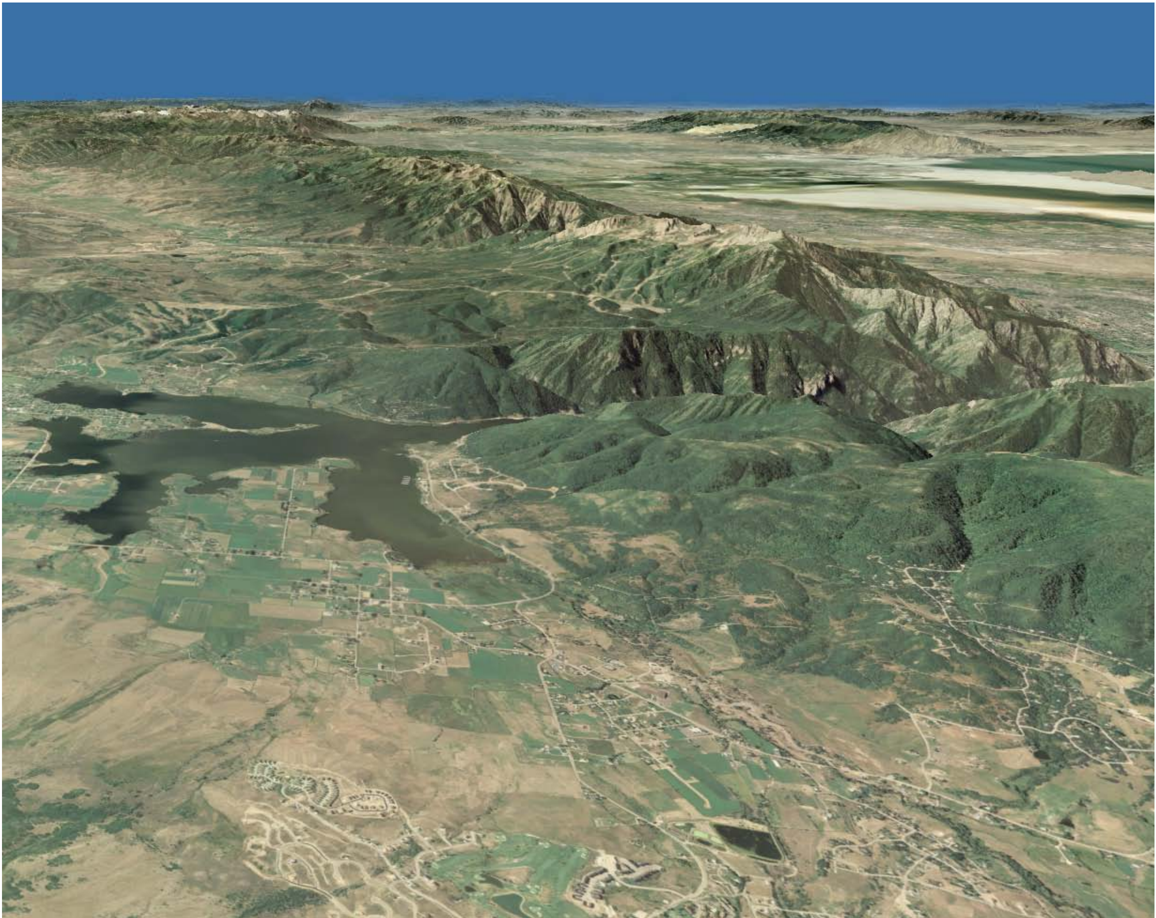
I assure the single handed composition of this bachelor thesis only supported by declared resources.

Munich, October 15, 2009 Andreas Kirsch

Acknowledgments

This bachelor thesis would not have been possible without my supervisor Prof. Dr. Westermann and my advisor Christian Dick. They were always there and ready to lend an ear and answer my questions. I'm also very grateful for the help and plentiful advice of Dr. Jens Schneider, who always found time for me and my rants about the problems I was fighting with, and who contributed many ideas and suggestions. His help in editing and structuring this thesis was invaluable.

I would also like to thank my friends and family for their support and patience. Last but not least I'm also very lucky to have such friends as my fellow students at the TUM who found time to read parts of this thesis and provide valuable feedback.



Abstract

Visualization is a very important area of computer science and it has useful applications in all areas of life. Rendering geographical data in a realistic fashion has been of great interest in the past and is a well-researched topic. One of the main research problems today is dealing with huge datasets effectively and in a way that allows for real-time interaction with them.

Moore's Law has governed the increase of computing power for decades, but hardware manufacturers are reaching the physical limits now and the prevalent solution is to rely on parallelization more heavily instead of hoping for serial performance increases. With parallelization becoming more and more important, Amdahl's Law and Gustafson's Law are the new indicators of achievable performance increases.

This Bachelor Thesis will discuss the steps necessary to port a terrain rendering engine from DirectX/Direct3D to OpenGL and set it up for parallelization on clusters using the Equalizer Parallelization Framework.

I will give an overview of the differences between Direct3D 10 and OpenGL 3.2 and the conceptual steps necessary to port applications from one API to the other. Useful tools and concepts for porting applications in general will be presented. The Equalizer Framework will be introduced briefly and the necessary changes to the application to use it will be explained. The thesis will conclude with a performance comparison between the original terrain rendering engine and the ported version and outline some possible further research topics.

Zusammenfassung

Visualisierung ist ein wichtiges Gebiet der Informatik und hat viele Anwendungen in allen Bereichen des Lebens. Insbesondere das Darstellen geographischer Daten in einer realistischen Art und Weise ist ein wichtiges und gut erforschtes Thema. Eines der Hauptprobleme in der Forschung heutzutage ist das effiziente Verarbeiten von sehr großen Datensätzen um Echtzeit-Interaktion mit jenen zu ermöglichen.

Moores Gesetz hat den Anstieg der Rechenleistung über Jahrzehnte hinweg bestimmt, aber Hardwareentwickler stoßen inzwischen an die von der Physik gesetzten Grenzen. Die vorherrschende Lösung ist sich mehr auf Parallelisierung zu verlegen, anstatt auf serielle Leistungsanstiege zu hoffen. Mit dem Aufstieg der Parallelisierung sind Amdahls Gesetz und Gustafsons Gesetz zu den neuen Indikatoren für erreichbare Leistungszuwächse geworden.

Diese Bachelorarbeit stellt die Schritte vor, die notwendig sind um eine Terrain-Rendering-Engine von DirectX/Direct3D auf OpenGL zu portieren und sie mit Hilfe des Equalizer Frameworks auf die Parallelisierung in Clustern vorzubereiten.

Ich werde einen Überblick über die Unterschiede zwischen Direct3D 10 und OpenGL 3.2 geben, sowie die grundlegenden Schritte vorstellen um eine Anwendung von einer API zur anderen zu portieren. Dabei werde ich nützliche Werkzeuge und Konzepte zum Portieren von Anwendungen im allgemeinen präsentieren. Das Equalizer Framework wird kurz eingeführt und die notwendigen Änderungen an der Anwendung werden erklärt. Die Arbeit schließt mit einem Vergleich der Leistung zwischen der ursprünglichen Terrain-Engine und der portieren Version und zeigt einige weitere, mögliche Forschungsthemen auf.

Contents

Acknowledgements **vii**

Abstract **xi**

Zusammenfassung **xiii**

Table of Contents **xv**

1 Introduction **1**

2 Direct3D 10 and OpenGL 3.2 **3**
2.1 A Short History of OpenGL 3
2.2 A Short History of DirectX and Direct3D 4
2.3 Graphics Pipeline 4
2.4 Direct3D 10 API 6
2.4.1 Device Object 6
2.4.2 State Objects 6
2.4.3 Resource Objects 8
2.4.4 Texture View Objects 9
2.4.5 Shader Objects 9
2.4.6 Effect Files 10
2.5 OpenGL 3.2 API 11
2.5.1 Object Model 11
2.5.2 Direct State Access Extension 15
2.6 Notable Differences between OpenGL and Direct3D 15
2.7 Indexed and Bufferless Drawing 17

3 Terrain3D Overview **19**
3.1 Architecture 19
3.1.1 Resource Pool 19
3.1.2 Data Loader 19
3.1.3 Renderer 21
3.1.4 Application 21
3.2 Texture Compression 21
3.3 Coupling between Terrain3D and DirectX/Direct3D 22

4 OpenGL Port **25**
4.1 Overview 25
4.1.1 Goals 25

4.1.2 Concept 25
4.1.3 Renderer Backend 26
4.1.4 Effect Files 27
4.2 `Effect` Class and Helper Classes 27
4.3 Effect Files 32
4.3.1 GLSL Effect File Format 32
4.3.2 ANTLR Grammar Definition 35
4.3.3 Compiler Code 35
4.3.4 StringTemplate Code 37
4.4 `Device` Class and Helper Classes 40
4.4.1 Class Hierarchy 40
4.4.2 Device Methods 40
4.5 Additional Changes in Terrain3D 46
4.5.1 Coordinate System 46
4.5.2 Indexed and Bufferless Drawing 46
4.5.3 GLUT Library 46

5 Equalizer Port **49**
5.1 The Equalizer Framework 49
5.1.1 Rendering Modes 49
5.1.2 Load Balancers 50
5.1.3 Configuration Files 50
5.1.4 API Overview 52
5.2 Overview of the Porting Process 56
5.3 Equalizer Application Code 57
5.3.1 eqPly's Equalizer Classes 57
5.3.2 Porting the Application Code 59
5.4 Changes to the Renderer Backend 62
5.5 Better Equalizer Support 64

6 Conclusion **67**
6.1 Performance Comparison 67
6.2 Further Reseach and Development 68
6.3 Results 69

Bibliography **71**

1 Introduction

Visualization is a very important area of computer science and it has useful applications in all areas of life. Modern graphics cards achieve a degree of realism previously not thought possible. Especially real-time graphics applications are revolutionizing the way people work in many areas of research and engineering. On the other hand it also revolutionizes the way people live because personal entertainment has been changing at a vast speed, too.

One topic useful for both areas is the rendering of geographical data in a realistic fashion. It has been of great interest in the past and is already a well-researched topic. One of the main research problems nowadays is dealing with huge datasets effectively and in a way that allows for real-time interaction with them. The pure size of the datasets already poses interesting questions about caching strategies and methods to effectively transfer data between the *CPU* (central processing unit) and *GPU* (graphics processing unit).

However, today the question about how to make effective use of parallelization techniques is just as important, and will be of even greater importance tomorrow.

Moore's Law has characterized the increase of computing power for decades, but now it is beginning to fail. It states that the number of transistors, and thus the computing power of a CPU, doubles every two years. But hardware manufacturers are now reaching the physically possible limits, and in recent years the pace of performance increases has slowed down. The only possible solution is to rely on parallelization more heavily instead of hoping for serial performance increases.

Other laws govern the realm of parallel computing: *Amdahl's Law* and *Gustafson's Law* are the new indicators that describe the achievable performance increases. Amdahl's Law states that, for constant data sizes, the maximum possible performance increase is limited by the sequential parts of a program. Gustafson's Law on the other hand argues that with increasing data sizes a linear speed up is asymptotically reached as the number of parallel processors reaches infinity.

The Chair for Computer Graphics and Visualization at the Faculty of Computer Science of the Technische Universität München have been conducting research both in the area of terrain visualization and in the area of parallelization in the past years [SBW06, SW06, BSK+07] and active research is being conducted right now as well [DSW09, DKW09, Kra09].

The Technische Universität München has an academic partnership with the recently-founded King Abdullah University of Science and Technology (KAUST) in Saudi Arabia. KAUST is building a CAVE (Cave Automatic Virtual Environment) system in its research labs. A CAVE is a virtual reality environment that consists of a room whose walls are projection screens. Usually head tracking devices and other means are used to increase the immersion further. The CAVE built by KAUST is driven by a cluster of 24 high-end workstations that each run 2 state-of-the-art NVIDIA Quadroplex units. Each QuadroPlex unit contains 4 Quadro GPUs. This makes it a total of 96 GPUs in a cluster that need to be coordinated to render efficiently.

This Bachelor Thesis discusses the steps necessary to port the terrain rendering engine *Terrain3D* from [DSW09] from DirectX/Direct3D to OpenGL, and how to set it up for parallelization on clusters using the Equalizer Parallelization Framework.

This new version of the Terrain3D terrain rendering engine will then be used to render immersive high-quality terrain landscapes in the CAVE in real-time. The CAVE has six projection walls which can display different images for the left and right eye each frame. This means that twelve different views have to be rendered every frame.

This thesis is structured in six chapters: The next chapter gives a condensed overview of the rendering APIs Direct3D 10 and OpenGL 3.2, and explains the differences between them.

Chapter 3 explains the architecture of Terrain3D and how it makes use of Direct3D 10 and the GPU to render detailed landscapes.

Chapter 4 discusses the conceptual steps necessary for porting applications from one API to the other and examines the OpenGL port of Terrain3D in detail.

Chapter 5 finally shows how to develop applications for the Equalizer Parallelization Framework in general and then specifically how the OpenGL version of Terrain3D has to be changed for Equalizer.

And Chapter 6 concludes this thesis. First it talks about the results and compares the performance of different versions of Terrain3D, then it describes how the work of this thesis can be used as starting point for further research.

2 Direct3D 10 and OpenGL 3.2

To provide a better background for understanding the problem of porting the terrain engine, an overview of the DirectX 10 API, specifically Direct3D 10, and the OpenGL API will be given first. Both APIs are very different in their design choices, which originates in their very different history. For completeness' sake and because it provides a good starting point, the following two sections will contain a brief history of both graphics libraries.

Readers who are already familiar with the APIs can skip these sections or just skim over them and continue with the next chapter. The following text attempts to stay basic and to explain most concepts, but naturally there is not enough space for an extensive explanation of everything. A general introduction to graphics and real-time rendering can be found in a [AMHH08]. The sections about DirectX are based on information available from Microsoft [MSD] and those about OpenGL are based on the official OpenGL 3.2 specification [SA09]. Other helpful resources for OpenGL are [WLH07] and [BSW$^+$07].

2.1 A Short History of OpenGL

OpenGL was created as an open standard by Silicon Graphics, Inc. (SGI) in 1992. The API was heavily influenced by SGI's Iris GL, a proprietary graphics library.

OpenGL is a multi-platform library and supports many different languages. It has a sophisticated extension system, that allows *IHVs* (independent hardware vendors) to implement their own extensions to the specifications. This way they can give developers access to special features of their hardware early on. Because of this, OpenGL developers should theoretically get access to new hardware features earlier[1]. The OpenGL *ARB* (architectural review board) can make an extension semi-official by supporting it and later incorporating it into the specifications.

The OpenGL specification have steadily evolved since the original release while always remaining compatible with older versions. Only with the release of OpenGL 3.0 in 2008 certain features were marked as deprecated (but are still supported by all IHVs).

One thing to note is that the API of OpenGL is quite old and it had to incorporate many transitions in the way computer graphics was dealt with. It has seen the advent of multi-

[1]As a matter of fact the current dominance of DirectX has the IHVs in lock step and extensions are often released to match the functionality of the next DirectX release.

texturing, programmable graphics hardware, buffer objects, off-screen rendering and other techniques that were probably not expected either when the original API was devised.

Although the number of game engines that use OpenGL has declined over time and only a minute number of current AAA game engines use OpenGL exclusively, OpenGL is still very strong in the CAD and design application area, as well as in research. One of the reasons could be that it is a multi-platform API in contrast to DirectX.

2.2 A Short History of DirectX and Direct3D

The first version of DirectX was released in 1996, and back then it did not contain a 3D subsystem. Only with the release of DirectX 3, a subsystem called Direct3D was included for the first time. DirectX was a small player at first, as most companies continued to use OpenGL and 3dfx's Glide. Only with the introduction of DirectX 7, DirectX became more popular. Unlike OpenGL each version of DirectX can offer a totally different interface. Taking advantage of this, Microsoft listened to the users of DirectX and gradually improved the API with each version.

Nevertheless, backwards compatibility is still achieved through the use of Microsoft's *COM* (component object model) and different interface classes. Thus one can still get an interface for Direct3D 3, even though DirectX 10 is installed.

DirectX does not offer an extension system like OpenGL. Instead the different DirectX versions have very different APIs, as mentioned above, that make use of a fixed set of features that graphic drivers have to support. This makes it easier for software developers to write code that runs well on many different system configurations compared to OpenGL, where the develops need to support different extensions or cope with the lack of thereof.

Now that the reader is familiar with the background history of both APIs, it is worth taking a look at the general graphics pipeline used by OpenGL and DirectX before moving to the code-level design of the APIs.

2.3 Graphics Pipeline

Figure 2.1 shows the pipeline that is used by both Direct3D and OpenGL 3.2. They both pretty much use the same pipeline—only some terms are different. The real differences can be found in the actual APIs and in small but important details in the specifications.

As you can see, the pipeline can be divided into seven stages:

Input Assembler The input assembler stage is responsible for reading in data and assembling the primitives[2] needed for rendering. The application feeds the data into the

[2]Primitives can be points, lines, triangles or quadrilaterals.

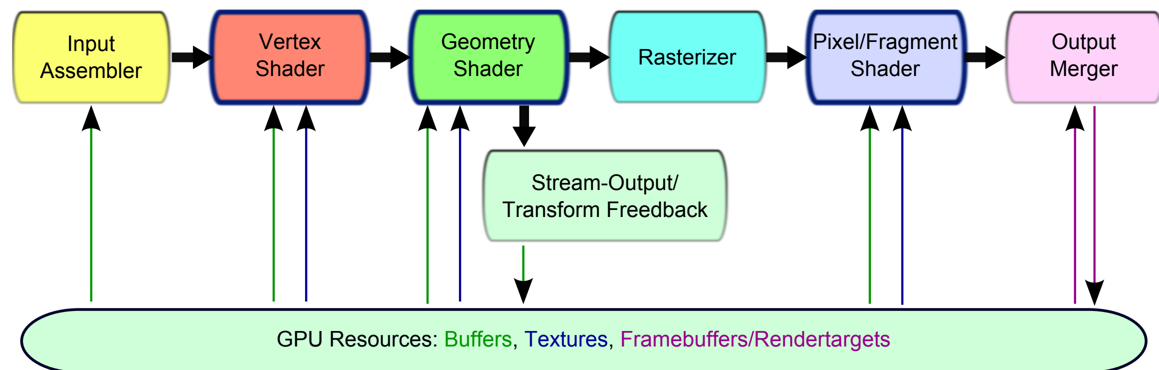


Figure 2.1: Graphics pipeline in Direct3D 10 and OpenGL 3.2

input assembler stage by uploading it into buffers. A buffer is an unstructured GPU resource. Anything can be stored in it and the application must tell the input assembler how to interpret the data inside the buffers.

There are two types of buffers: vertex buffers and index buffers. Vertex buffers contain vertex data. Index buffers store indices into the vertex buffers that determine in which order the input assembler reads data out of the vertex buffers. If no index buffer is used, the input assembler reads data out of the vertex buffers sequentially.

It is possible to render without suppling vertex data by only using index buffers, or even to render without supplying any data at all to the input assembler. See Section 2.7 on page 17 for more information about this advanced feature.

Vertex Shader The vertex shader stage is a programmable stage. The application provides a so-called vertex shader which is used to transform the vertices that enter the stage. Usually coordinate transformations are applied to orient and move the geometry, and lighting is simulated to increase realism.

A vertex shader can always only access one vertex and operate on its data. This allows the stage to be heavily parallelized. In addition to the vertex's data, a vertex shader can also access textures and buffers. Textures are containers for structured data like images. It is possible to create one dimensional, two dimensional or three dimensional textures[3]. There also exist more advanced texture types.

Geometry Shader The geometry shader stage is also a programmable stage. Here a geometry shader is used to operate on the input vertex stream. In contrast to the vertex shader the geometry shader is operating on a primitive as a whole. Thus it has access to the transformed data of all vertices that make up one primitive. It can create

[3]Texture are in a way like arrays: they have a fixed dimension and the base type has to be set up-front.

additional primitives or discard primitives. Even the primitve type can be changed between input and output stream.

Geometry shaders can be used to tessellate geometry or perform other advanced operations that cannot be executed in vertex shaders because of their limitations.

The geometry shading stage can additionally access texture and buffer resources like the vertex shading stage.

Stream-Output/Transform Feedback Direct3D calls this the stream-output stage. OpenGL calls it the transform feedback stage. It does the same in both though: it is an optional stage, that allows writing out the primitives, respectively vertices, to a buffer after they have been transformed by the vertex and geometry shader.

Using a geometry shader and stream-output, it is possible to create new geometry on the fly and store it for later rendering.

Rasterizer The rasterizer stage takes the primitives and rasterizes them. This means that the area of each primitive is converted into fragments. Fragments are precursors of the pixels that will later appear on the screen. To do this the rasterizer takes the vertex data of the the different vertices and interpolates the properties that are needed later across the primitives, and for each pixel that could be written, a fragment is generated with a snapshot of the interpolated values.

Pixel/Fragment Shader This stage is called the pixel shader stage in Direct3D and the fragment shader stage in OpenGL. It is again a programmable stage and the pixel shader, respectively fragment shader, which is supplied by the application, can operate on the fragment data and set the output values of the fragment (usually the color and depth of the fragment).

Like other shaders it can access buffer and texture resources.

Output Merger The output merger stage takes the shaded fragments and decides whether they should be turned into pixels or not. They have to pass multiple tests before they are written to the framebuffer.

An example for such a test is the *depth test*: usually to pass the depth test, the depth value of a fragment has to be smaller than the one of the pixel currently residing in the framebuffer (that is the fragment is in front of the current pixel).

The framebuffer can either be the screen or a system buffer that is swapped with the screen buffer every frame to avoid tearing, or it can be a framebuffer object in OpenGL or a rendertarget in Direct3D. Framebuffer objects and rendertargets both reference a GPU resource like a buffer or texture for writing. They wrap the type of

the resource for the output merger. In the case of OpenGL, framebuffer objects do not only identify one output resource but all of them, while in Direct3D multiple rendertargets can be set at the same time and each only identifies one output resource.

This was a condensed overview of the graphics pipeline. A good and more in-depth introduction can be found in [AMHH08]. For details about Direct3D 10's pipeline [MSD] is the reference source and [SA09] for OpenGL.

2.4 Direct3D 10 API

DirectX overall uses COM and a typical object-oriented API. Direct3D 10 has a straightforward class hierarchy:

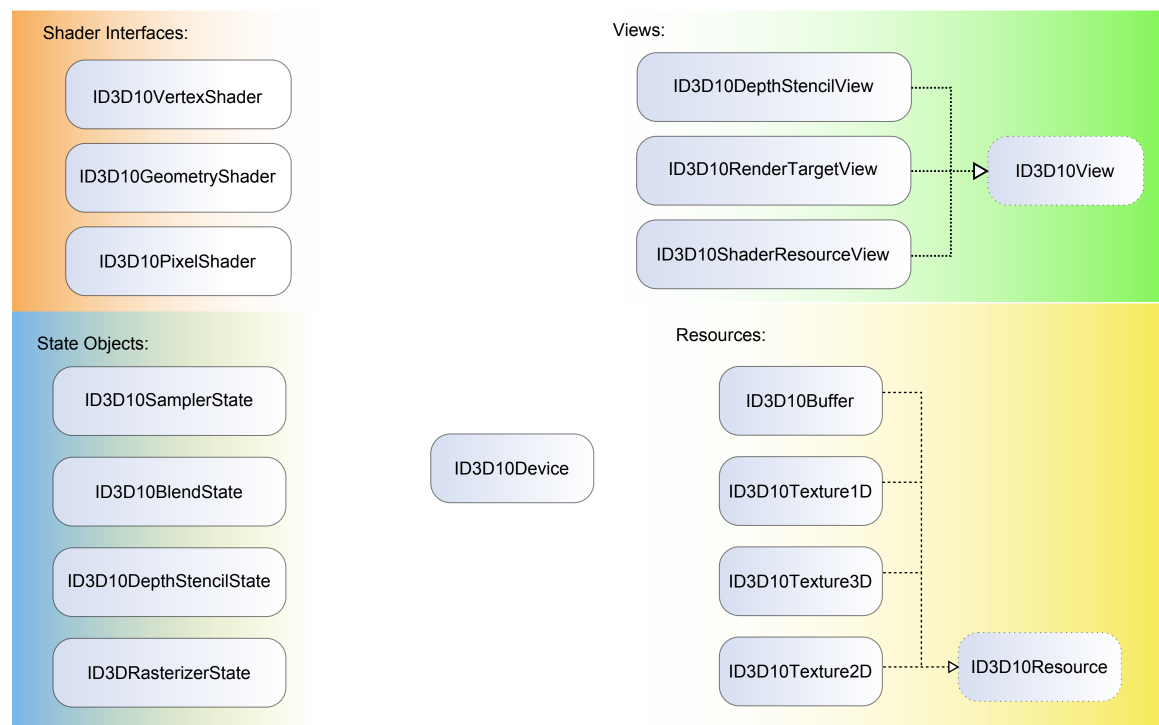


Figure 2.2: Direct3D 10 core and resource classes

2.4.1 Device Object

`ID3D10Device` is the main interface you use to change the graphics device's state (using state objects), create new objects on the device (textures, buffers, shaders, etc.) and render primitives.

Table 2.3: Method prefixes in ID3D10Device

Prefix	Stage	Example Method Name
GS	Geometry Shader	GSSetShader
IA	Input Assembler	IASetPrimitiveTopology
OM	Output Merger	OMSetRenderTargets
PS	Pixel Shader	PSSetShader
RS	Rasterizer Stage	RSSetViewports
SO	Stream Output	SOSetTargets
VS	Vertex Shader	VSSetShader

It has 95 methods and many are used to change the state of the pipeline and its different stages. To make it easier for the programmer to see which pipeline stage a method affects prefixes are used. See Table 2.3 on the facing page for an overview of all used prefixes.

2.4.2 State Objects

Most of the state changes in Direct3D are wrapped using state objects. State objects cannot be changed after creation. This allows for more efficient state management in the driver. There are four state interfaces in Direct3D 10: **`ID3D10BlendState`**, **`ID3D10DepthStencilState`**, **`ID3D10RasterizerState`**, **`ID3D10SamplerState`**.

One usually creates several different state objects at program initialization and later sets the state of a pipeline stage with one call to the corresponding **`*SetState`** method of **`ID3D10Device`**. This reduces the function call overhead for state changes, which has been an issue with previous versions of DirectX and still is an issue in OpenGL today. See Section 2.5.2 on page 15 for an extension that mitigates the problem in OpenGL.

The example code in Listing 2.4 on the following page creates a rasterizer state object to render primitives in wire-frame mode and makes it active.

2.4.3 Resource Objects

Resource objects all inherit from ID3D10Resource and are created using one of the **`Create*`** methods of **`ID3D10Device`**. There are four resource interfaces: **`ID3D10Texture1D`**, **`ID3D10Texture2D`**, **`ID3D10Texture3D`**, **`ID3D10Buffer`**.

When a resource is created, so-called bind flags and a usage setting have to be specified for it. The bind flags specify which pipeline stages the resource can be used with. They also specify whether it can be used as vertex or index buffer (or both) for rendering. See Table 2.5 for a list of all flags. The usage setting specifies how it is used: whether it is immutable, dynamic, or allows read and write operations from the CPU. See Table 2.6 on the facing page for a list of all possible settings.

Listing 2.4: State object example

```
ID3D10RasterizerState*  pRasterizerStateWireframe = NULL;

D3D10_RASTERIZER_DESC RSDesc;
ZeroMemory( &RSDesc, sizeof(RSDesc) );
RSDesc.FillMode = D3D10_FILL_WIREFRAME;
RSDesc.CullMode = D3D10_CULL_BACK;
RSDesc.FrontCounterClockwise = false;
RSDesc.DepthClipEnable = true; // Clipping
RSDesc.MultisampleEnable = true;
pd3dDevice->CreateRasterizerState( &RSDesc,
   &pRasterizerStateWireframe );

pd3dDevice->RSSetState( pRasterizerStateWireframe );
```

Table 2.5: Bind flags for resources in Direct3D 10 (members of D3D10_BIND_FLAG). Source: MSDN

D3D10_BIND_VERTEX_BUFFER
D3D10_BIND_INDEX_BUFFER
D3D10_BIND_CONSTANT_BUFFER
D3D10_BIND_SHADER_RESOURCE
D3D10_BIND_STREAM_OUTPUT
D3D10_BIND_RENDER_TARGET
D3D10_BIND_DEPTH_STENCIL

Table 2.6: Usage flags for resources in Direct3D 10 (members of D3D10_USAGE). Source: MSDN

Setting	Description
D3D10_USAGE_DEFAULT	GPU has read and write access
D3D10_USAGE_IMMUTABLE	Read-only after initialization
D3D10_USAGE_DYNAMIC	GPU has read and write access, CPU is write-only
D3D10_USAGE_STAGING	Only used for moving data between the CPU and GPU

Listing 2.7: D3D10_TEX2D_SRV definition from MSDN

```
typedef struct D3D10_TEX2D_SRV {
    UINT MostDetailedMip;
    UINT MipLevels;
} D3D10_TEX2D_SRV;
```

ID3D10Device contains a few methods that deal with resources (other than the respective **Create*** methods). The most important ones are: **CopyResource** and **UpdateSubresource**. (They are self-explaining.)

Buffer objects are containers for data that is stored in GPU memory. DirectX doesn't care about their internal structure. They are often used to store vertex and index data. Texture objects on the other hand are structured and the programmer has to tell Direct3D what color format is used. [AMHH08] contains a very good introduction to these topics.

A texture object cannot be bound to a stage directly, instead a texture view object has to be created for the texture.

2.4.4 Texture View Objects

Texture view objects can be bound to a stage using methods of **ID3D10Device** such as **OMSetRenderTargets** and **PSSetShaderResources**.

There are three texture view interfaces: **ID3D10DepthStencilView**, **ID3D10RenderTargetView** and **ID3D10ShaderResourceView** . The first two views are used to bind a texture as output to the output merger stage. The last one is used to bind it to shaders. What is the advantage of having additional objects for this?

For example, when a shader resource view is created, you specify two mipmap parameters (see Listing 2.7): the maximum mipmap level and the base mipmap level. As this is a per view setting, one texture can have multiple pre-created views which can be used in different shaders without any state changes in-between.

This sounds trivial, but in OpenGL, for example, this is state that is stored per texture, because OpenGL does not have views. Thus you would have to change the texture state multiple times when switching between shaders, which is more expensive.

2.4.5 Shader Objects

There are 3 shader interfaces: **ID3D10GeometryShader**, **ID3D10PixelShader** and **ID3D10VertexShader**. **ID3DDevice** offers the **CreatePixelShader**,

CreateVertexShader and **CreateGeometryShader** methods to create shader objects.

They take the compiled shader bytecode as parameter, which can be generated from **HLSL** shader code by using the **D3D10CompileShader** global function or a variant of it.

High Level Shading Language

HLSL stands for High Level Shading Language and is the C-like shader language used by DirectX. In Direct3D 10 HLSL borrows some concepts from C++. While it does not support the definition of new classes or templates, most datatypes can be accessed using an object-oriented and template-like syntax:

Listing 2.8: Examples of HLSL's C++-like syntax

```
// templated matrix type
matrix<float, 4, 4> mWorldView;

// texture access example
Texture2D<float4> txTerrain;
[...]
float4 result = txTerrain.Sample(samplerState, input.vTexCoord);

// geometry shader
[maxvertexcount(3)]
void GSDecompressStripP1(point VSDecompressStripP1Out input[1],
   inout PointStream<GSDecompressStripP1Out> pointStream) {
    GSDecompressStripP1Out result;

    result.vertices[0].r = input[0].vertices[0].g; // v2
    [...]
    pointStream.Append(result);
```

2.4.6 Effect Files

Direct3D offers a powerful file format to store shaders, and organize them in a natural way: the effect file format. Effect files commonly use the extension *.fx*.

The philosophy behind it is that shaders are used to render effects. Usually there are different ways to render an effect depending on the capabilities of the hardware or the

desired quality. Thus an effect can comprise several different techniques to render the effect. Each technique can require multiple rendering passes with different shaders.

.fx File Format

Employing this idea, an effect file can contain several different techniques and each technique is made up of one or more passes. A pass is described by the shader functions that should be used for pixel, vertex and geometry shading, and the state the device should be set to before rendering. Therefore an effect file contains HLSL code, state block definitions and a number of technique and pass declarations. See Listing 2.9 on the following page for a simple effect file.

API

There are four main interfaces that deal with effect files in Direct3D 10: **`ID3D10Effect`**, **`ID3D10EffectTechnique`**, **`ID3D10EffectPass`**, **`ID3D10EffectVariable`**. See Figure 2.10 on page 13 for a class diagram of the most important effect file classes. Effect files can be compiled into a binary format to speed up loading using functions like **`D3DX10CompileFromFile`**.

An effect is loaded by calling **`D3DX10CreateEffectFromFile`** or a similar function. The functions return an **`ID3D10Effect`** object. The **`ID3D10Effect`** objects offer methods that query the number of available techniques, return a certain technique as **`ID3D10EffectTechnique`** object or a certain global shader variable as **`ID3D10EffectVariable`**.

`ID3D10EffectTechnique` itself allows one to query the passes the technique contains and retrieve an **`ID3D10EffectPass`** object for a pass. **`ID3D10EffectPass`** has only one important method: **`Apply`**. It applies all settings for the pass from the effect file to the device: it sets the state as defined in the state block used by the pass, and activates the shaders. After calling **`Apply`**, primitives are rendered as specified in the pass declaration.

`ID3D10EffectVariable` can be used to obtain interfaces for shader variables such as **`ID3D10EffectMatrixVariable`** or **`ID3D10EffectVectorVariable`** which have *set* methods to change the value of shader variables.

2.5 OpenGL 3.2 API

2.5.1 Object Model

OpenGL was devised in an era when C was commonly used, and it only supports a C interface. It still has the notion of different objects but it uses a context-based design to

Listing 2.9: *tileRenderingBB.fx* (adapted)

```
DepthStencilState EnableDepth {
    DepthEnable = FALSE;
};

cbuffer cbEnvironment {
    matrix mWorldView;
    matrix mProjection;
    float4 vColorBB;
}

struct VSInBB {
    float4 vPos : POSITION;
};

struct PSInBB {
    float4 vPos : SV_Position;
};

PSInBB VSBB(VSInBB input) {
    PSInBB output = (PSInBB)0;
    output.vPos = mul(input.vPos, mWorldView);
    output.vPos = mul(output.vPos, mProjection);
    return output;
}

float4 PSBB() : SV_Target {
    return vColorBB;
}

technique10 RenderBB {
    pass P0 {
        SetVertexShader(CompileShader(vs_4_0, VSBB()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, PSBB()));

        SetDepthStencilState( EnableDepth, 0 );
    }
}
```

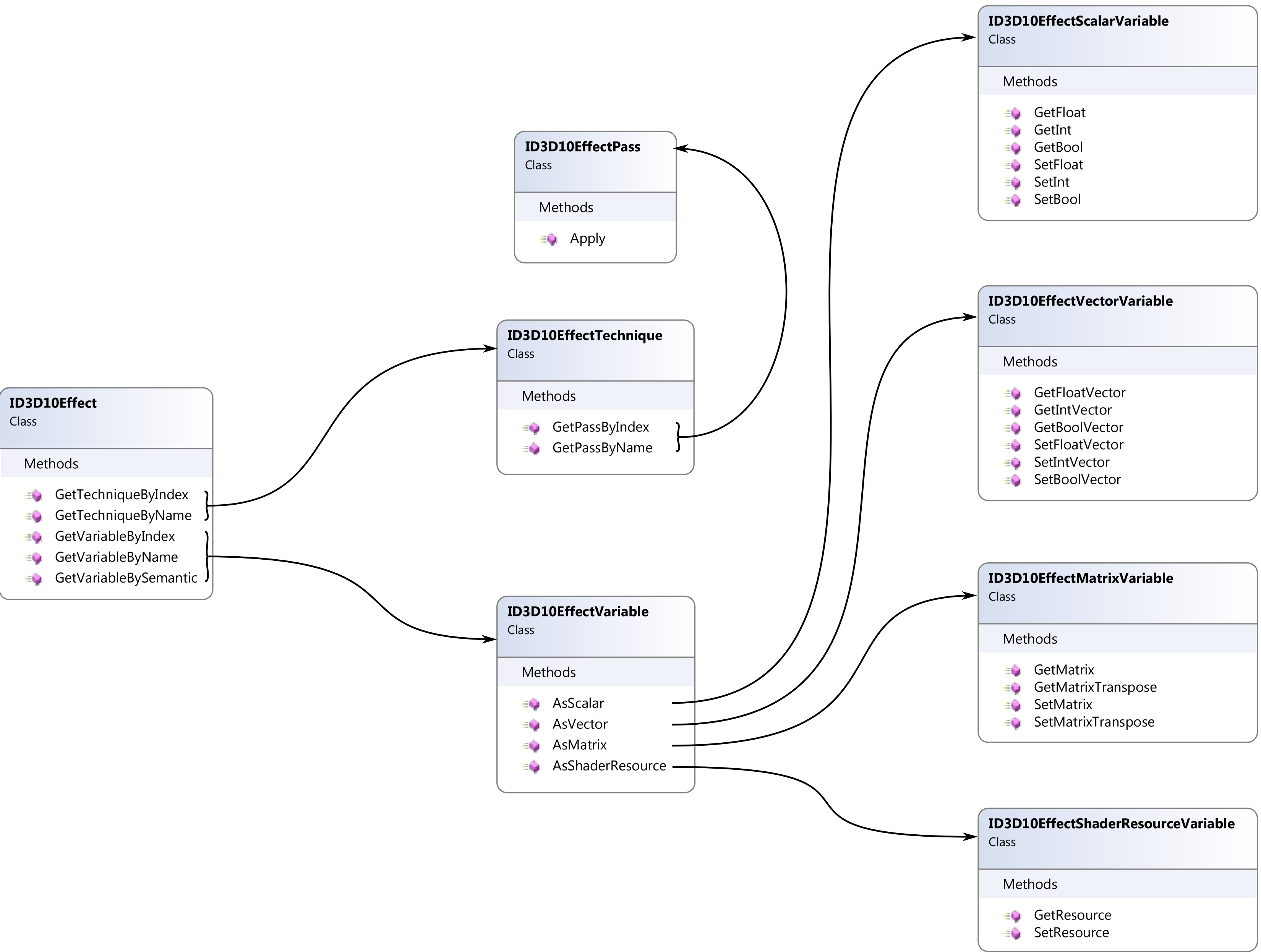


Figure 2.10: Direct3D 10's effect class hierarchy (showing selected methods)

access them: objects (e.g. textures, vertex buffers and framebuffers) are identified using unsigned integer handles. You can picture the identifier as being a key into a hash map containing the pointers to the actual objects—this is probably how the drivers implement it internally, too. Each object type gets its own map. Thus the same identifier can be used eg for a texture and a vertex buffer at the same time. To change the state of an object, it has to bound, that is, made active. For every object type there are different binding points it can be bound to. Only one object can be bound to one binding point at a time. Many OpenGL functions have a parameter that specifies the binding point they affect. This adds a level of indirection to the API, because an object has to be bound to a binding point and then the binding point is used as target parameter in OpenGL functions that modify the object's state.

Example:

```
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );

glBufferData( GL_ARRAY_BUFFER, sizeof( data ), data,
    GL_STATIC_DRAW );

glBindBuffer( GL_ARRAY_BUFFER, 0 );
glDeleteBuffers( 1, &buffer );
```

The example creates a new buffer object, binds it to the **`GL_ARRAY_BUFFER`** binding point and uploads data to the buffer object through the binding point. Afterwards it unbinds the object by binding the default 0 object, and deletes it.

The paradigm described above is used for creating and managing vertex buffer objects, framebuffers and textures. The general management functions are (whereas **`Type`** can be either Buffer, Framebuffer or Texture):

```
void glGenTypes( GLuint count, GLuint *handles );
void glDeleteTypes( GLuint count, GLuint *handles );

GLbool glIsType( GLuint handle );

void glBindType( GLenum target, GLuint handle );
```

target is the parameter that specifies the binding point. For textures there are several binding points: one for each possible texture type (1D, 2D, 3D, etc). Buffer objects also have different binding points: **GL_ARRAY_BUFFER** is used for vertex buffers, **GL_ELEMENT_ARRAY_BUFFER** is used for index buffers, and **GL_TRANSFORM_FEEDBACK_BUFFER** is used as target buffer for the transform feedback stage. See [SA09] for more binding points. In subsequent rendering commands the currently bound objects are used.

A notable exception from this paradigm is the creation of shader and shader program objects, which uses a different style:

```
GLuint glCreateProgram( void );
void glDeleteProgram( GLuint programHandle );

GLuint glCreateShader( GLenum type );
void glDeleteShader( GLuint shaderHandle );
```

A Shader is a specific shader: either a vertex, fragment or geometry shader. A fragment shader is the equivalent of a pixel shader in Direct3D[4]. A Program links several of these shaders together. It can be bound to the programmable pipeline, while a single shader cannot be activated separately from a program[5].

2.5.2 Direct State Access Extension

As mentioned above, if you want to modify an object, you have to bind it to a binding point. This, of course, changes a binding point which might have been used for a different object and might be required for subsequent rendering calls. This places an additional burden on programmers, because they have to keep track of bound objects and binding points, and must ensure that the old state is restored, if necessary, after modifying an binding point.

Especially for third-party libraries, this can cause a performance penalty. They cannot cache state locally and instead need to use OpenGL's state query functions.

To address this problem and to allow for a tighter coupling between the object to be modified and the calls that modify it, an OpenGL extension has been introduced in late 2007. It adds an additional parameter to many OpenGL functions to specify the target

[4]Fragment shader is actually a better name, because these shaders work with fragments. Pixels are usually elements of the framebuffer, whereas the output of the shaders may not pass the depth test or one of the other tests and thus not become a pixel after all.

[5]The recent **EXT_separate_shader_objects** extension adds support for binding shaders directly.

object directly. The extension is called **EXT_DIRECT_STATE_ACCESS** [DSA]. The following two listings show how this extension reduces code bloat:

```
/* code that uploads data to a buffer (without using the
   extension) */

// backup the state of the binding point
GLuint oldBuffer;
glGetIntegerv( GL_ARRAY_BUFFER_BINDING, &oldBuffer );

// upload data
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferSubData( GL_ARRAY_BUFFER, offset, size, data );

// restore the old state
glBindBuffer( GL_ARRAY_BUFFER, oldBuffer );
```

```
/* version that uses the extension */
glNamedBufferSubDataEXT( buffer, offset, size, data );
```

This was a very brief overview of OpenGL. In contrast to DirectX, which does not have real specifications, specifications for OpenGL are available und very comprehensive.

2.6 Notable Differences between OpenGL and Direct3D

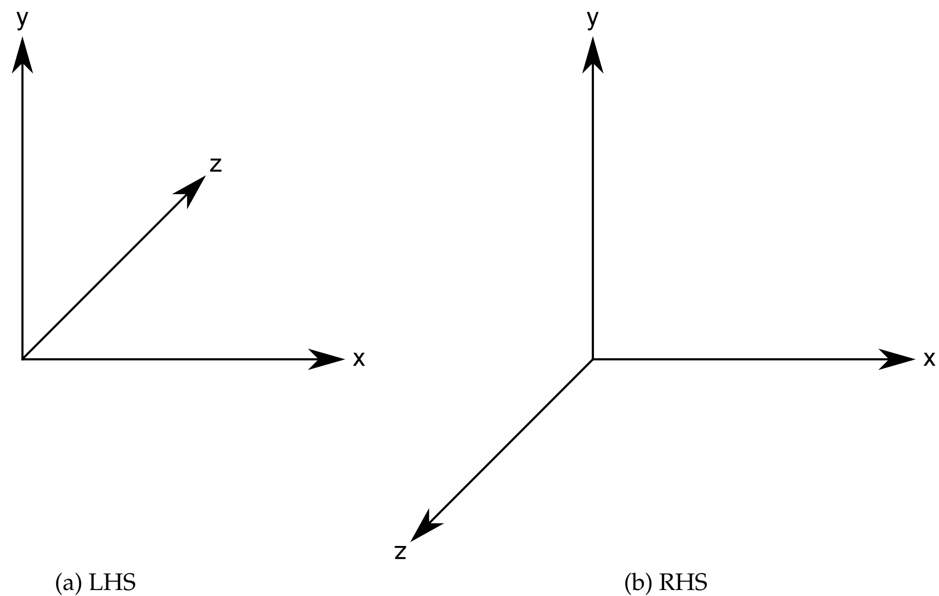
There are a few fine differences between the two APIs that are worth remembering:

Coordinate System Direct3D uses a *left-handed coordinate system*, while OpenGL uses a *right-handed coordinate system*. A left-handed coordinate system can be transformed into a right-handed one and vice-versa by flipping the z-axis. See Figure 2.11.

Origin of Window Coordinates OpenGL assumes that $(0, 0)$ is the lower-left corner of the window, while Direct3D assumes it is the upper-left corner.

Provoking Vertex Direct3D assumes that the first vertex of a primitive is the *provoking vertex*, while in OpenGL it is by default[6] the last vertex of a primitive. The provoking vertex is the vertex that supplies the values for the *flat-shaded* attributes of a primitive.

[6] OpenGL 3.2 introduces the function **ProvokingVertex**, which allows you to change this setting



(a) LHS

(b) RHS

Figure 2.11: Left-handed vs right-handed coordinate system

Flat-shaded attributes are attributes that are not interpolated between the different vertices.

Bind Flags Objects in OpenGL do not have any bind flags.

Usage Flags Textures in OpenGL do not have a usage setting.

Matrix Storage Direct3D and HLSL use *row-major order* for matrices. OpenGL and GLSL use *column-major order* for matrices. In row-major order a matrix is stored column by column in memory, thus if it were accessed as two-dimensional array in C, the first, major, dimension would specifiy the row. In column-major order a matrix is stored row by row in memory. The same order is used when initializing matrices in HLSL and GLSL, so you have to pay attention to switch between the two when porting code. For example, this matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

would be specified in HLSL as follows:

```
int3x3 matrix = {
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
```

```
};
```

while in GLSL the code would look like this:

```
mat3 matrix = {
    1, 4, 7,
    2, 5, 8,
    3, 6, 9
};
```

2.7 Indexed and Bufferless Drawing

In Direct3D 10 it is possible to render geometry without specifying a vertex buffer. This is is possible because Direct3D always supplies the vertex and geometry shader with three implicit inputs: **SV_VertexID**, **SV_PrimitiveID** and **SV_InstanceID**. **SV_VertexID** is the index of the current vertex in the vertex buffer. If no index buffer is used, this value is incremented with each processed vertex. On the other hand if an index buffer is used, the value corresponds to the index of the vertex that is currently processed.

SV_PrimitiveID is incremented with each primitive that enters the vertex shader stage. For example for first three vertices of a triangle list, **SV_PrimitiveID** is 0, for the next three it is 1, and so on.

SV_InstanceID is used for instanced drawing: the same geometry is drawn multiple times and **SV_InstanceID** is incremented each time.

By using these input values it is possible to create vertices and geometry without vertex buffers. *Indexed drawing* refers to storing all the needed in the index buffer and reading **SV_VertexID** to access it. Indices can be 8-bit, 16-bit or 32-bit integer values. Bit-shifting and -masking operators are available in HLSL, too. One way to use indexed drawing for rendering is to pack vertex data into the indices and unpack it in the shader.

Bufferless drawing refers to rendering with neither index nor vertex buffers. This can be useful for rendering data that can be generated from the consecutively increasing **SV_VertexID** and **SV_PrimitiveID** values. Bufferless drawing can be used to render procedural primitives, or to store the vertex data directly in the shader instead of in a vertex buffer and index it manually. The latter is often used to draw a single triangle that covers the entire screen during post-processing passes.

3 Terrain3D Overview

Since this thesis revolves around Terrain3D (previously described in [DSW09]), you should become familiar with the architecture of the project and its different subsystems, and get to know how the different subsystems use the GPU. The following sections describe Terrain3D before the port to OpenGL and Equalizer.

3.1 Architecture

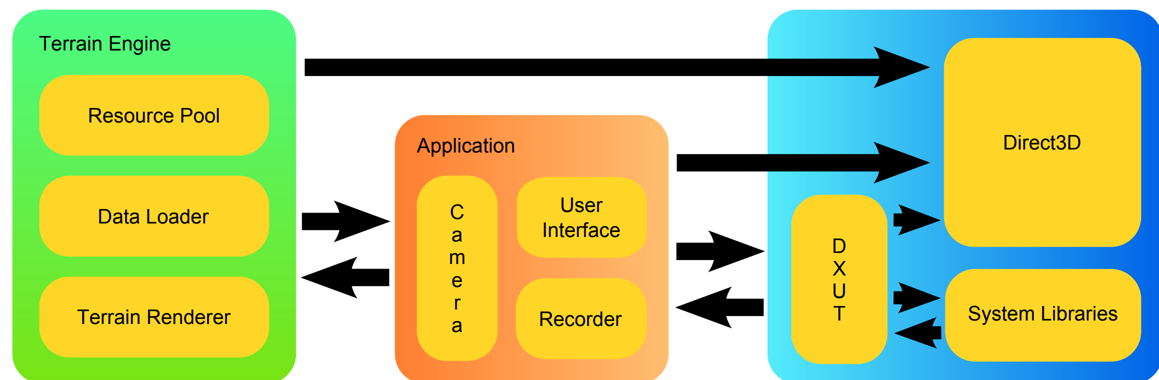


Figure 3.1: Overview of Terrain3D's different modules and interaction with libraries

Terrain3D can roughly be divided into two parts: terrain-specific code like the resource pool, the data loader and the terrain renderer on the one hand and on the other hand the application code which consists of the user interface and windowing system code. The windowing system code uses DXUT[1] (DirectX Utility Library). See Figure 3.1.

3.1.1 Resource Pool

The resource pool manages the texture and vertex buffer resources of the terrain. It tries to reuse device resources in order to avoid creating new resources after initialization. For this it uses reference counting to keep track of resource usage, and treats the resources like a cache. Loaded resources are evicted using the least-recently-used principle.

It allocates and manages GPU resources such as textures and vertex buffers.

[1]DirectX's analog to GLUT

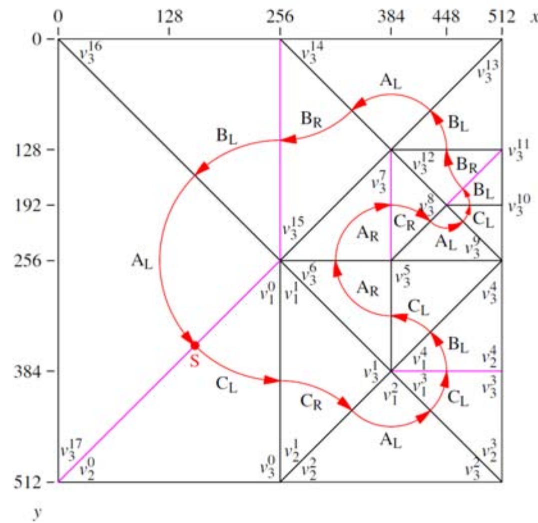


Figure 3.2: Example for a closed triangle path of a terrain tile (taken from [DSW09])

3.1.2 Data Loader

The data loader loads geometry and texture data from the hard disk on demand.

The terrain is stored in a tiled quadtree. To increase data throughput 4×4 tiles are bundled in one page.

New pages are read from disk when needed. More specifically the data loader prefetches the data in a mostly view-direction agnostic way[2]: it computes a prefetch radius for each LOD (level-of-detail) level depending on the current screen resolution and field of view, and loads the relevant pages inside these radii.

Geometry data and texture data is stored in compressed form. The geometry data of each tile is stored as a compressed closed-path triangle strip, which allows for very space-efficient encoding. See Figure 3.2 for an example. There are three different ways in Terrain3D to decompress the data:

- decompressing the data on the CPU;
- decompressing the data on the GPU with a single combined vertex and geometry shader pass, which Direct3D's stream-out stage to store the result in a vertex buffer;
- decompressing the data on the GPU using multiple vertex and fragment shader passes, which uses texture ping-ponging[3], and a final geometry shader pass with Direct3D's stream-out stage to store the result in a vertex buffer.

[2]Tiles in the center of the screen have priority though

[3]Assuming two textures A and B are used, then in the first pass A is the render target. In the second pass B becomes the render target and A is the input texture. In the third pass A becomes the render target and B is the input texture, and so on.

See [DSW09] for a detailed explanation and analysis.

The texture data can be compressed in three different ways:

- using DXT1/BC1 compression,
- using DXT1/BC1 compression and a clever mipmap extraction scheme to reduce redundant data, or
- using Vector Quantizer compression.

This will be discussed in more detail in Section 3.2.

3.1.3 Renderer

The renderer is responsible for rendering the terrain. It uses frustum culling to determine the set of visible tiles and renders them using a single pass.

To increase vertex data throughput, one vertex is encoded in 32 bits: 10 bits each for the x and y component and 12 bits for the z component . Compared to supplying three floats per vertex, this saves 8 bytes per vertex – that is, three times as many vertices can be sent to the GPU. This is an important gain in a bandwidth limited application like Terrain3D.

The renderer in the Direct3D 10 version also uses indexed drawing to speed up renderin (see Section 2.7 on page 17).

3.1.4 Application

The Application loads the configuration files, initializes the specialized modules described above and processes user events. It supports keyboard as well as mouse and joystick input and provides a `Camera` class that controls the viewer position and direction. The `Recorder` class adds the ability to record flights over the terrain to a file and play them back later. This feature is useful for demos and benchmarks.

3.2 Texture Compression

In Section 3.1 on page 19 three different texture decompression modes were presented. In this section we will take a closer look at them:

DXT1/BC1 textures DXT1/BC1 texture compression is used which has a compression ratio of 50%. Each tile contains a full mipmap chain.

Shared DXT1/BC1 textures As can been seen in ..., tile 0 covers the area of tiles 1, 2, 3 and 4. Likewise tile 1 covers the area of tiles 5, 6, 7, 8. Every tile uses a texture with the

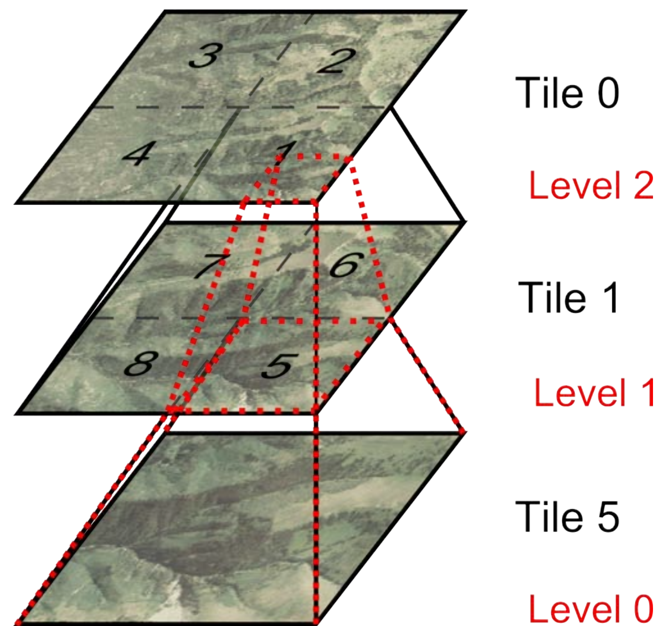


Figure 3.3: Tile tree of depth 3, shown along the path to tile 5 (the shared mipmap chain is marked in red)

same resolution. Only tile 0 stores an explicit mipmap chain. The other tiles extract their mipmap levels from their parent tiles.

For example, tile 5 uses the lower-right quadrant of tile 1's texture as mipmap level 1 and the lower-right octant of tile 0 as mipmap level 2. The remaining mipmap levels are extracted from the mipmap levels of tile 0. Figure 3.3 on the following page visualizes this example.

Using this technique only about three quarters of the texture data of the non-shared DXT1/BC1 texture method need be stored on disk [4].

Vector Quantizer The Vector Quantizer has a bitrate between 1.5 and 3 bits per texel. An uncompressed texture uses 24 bits per texel, thus the Vector Quantizer reaches compression rates between 16:1 and 8:1, with typical ratios being around 12:1 (at fixed a bitrate per tile). Although it is superior to DXT1/BC1 texture compression (both in compression efficacy and compression speed), it has to be fully decoded on the GPU to ensure correct texture filtering during rendering. It results in a textures working set that is about *six times* larger (for any given view) than the one achieved with DXT1/BC1 compression. Since this is unacceptable for many applications, it has been deprecated.

[4]This can be easily deduced by the fact that a texture mipmap chain for a square texture of resolution $2^w \times 2^w$ requires $4^w + 4^{w-1} + 4^{w-2} + \cdots + 4^0 = \frac{4^{w+1}-1}{4} \approx \frac{4}{3} \cdot 4^w$ texels. If all tiles except for the coarsest one only store the full resolution texture, only about 75% of the texels are needed.

Table 3.4: Techniques in terrain3d.fx

Technique	Number of Passes
RenderBitBlt	4
Sprite	2
RenderFrustum	1

Table 3.5: Techniques in terrain.fx

Technique	Number of Passes
Render	2
VQDecode	3
RenderBB	1
MultiPassDecompressGeometry	17
SinglePassDecompressGeometry	1

3.3 Coupling between Terrain3D and DirectX/Direct3D

Terrain3D is a complex project that uses many different features of Direct3D 10. As mentioned above every subsystem is coupled to Direct3D 10. D3DX, a Direct3D helper library, is also used extensively: Terrain3D uses its matrix and vector classes. The application code uses the DXUT framework to keep the window management code small.

Additionally the project uses two effect files: *terrain.fx* for the terrain subsystems and *terrain3d.fx* for the application code itself. The former contains techniques that the data loader and renderer use and the latter contains some helper techniques that are used by the application to display a debug frustum and perform post-processing on the output frame. See Table 3.4 on the next page and Table 3.5 on the facing page for a list of all techniques and the number of passes they contain.

Effect file loading and access is wrapped in a generic **`Effect`** class and each effect file has its own specialized class that inherits from this **`Effect`** class and sets up member variables for all uniforms in the effect file. Figure 3.6 on the next page shows the class diagrams.

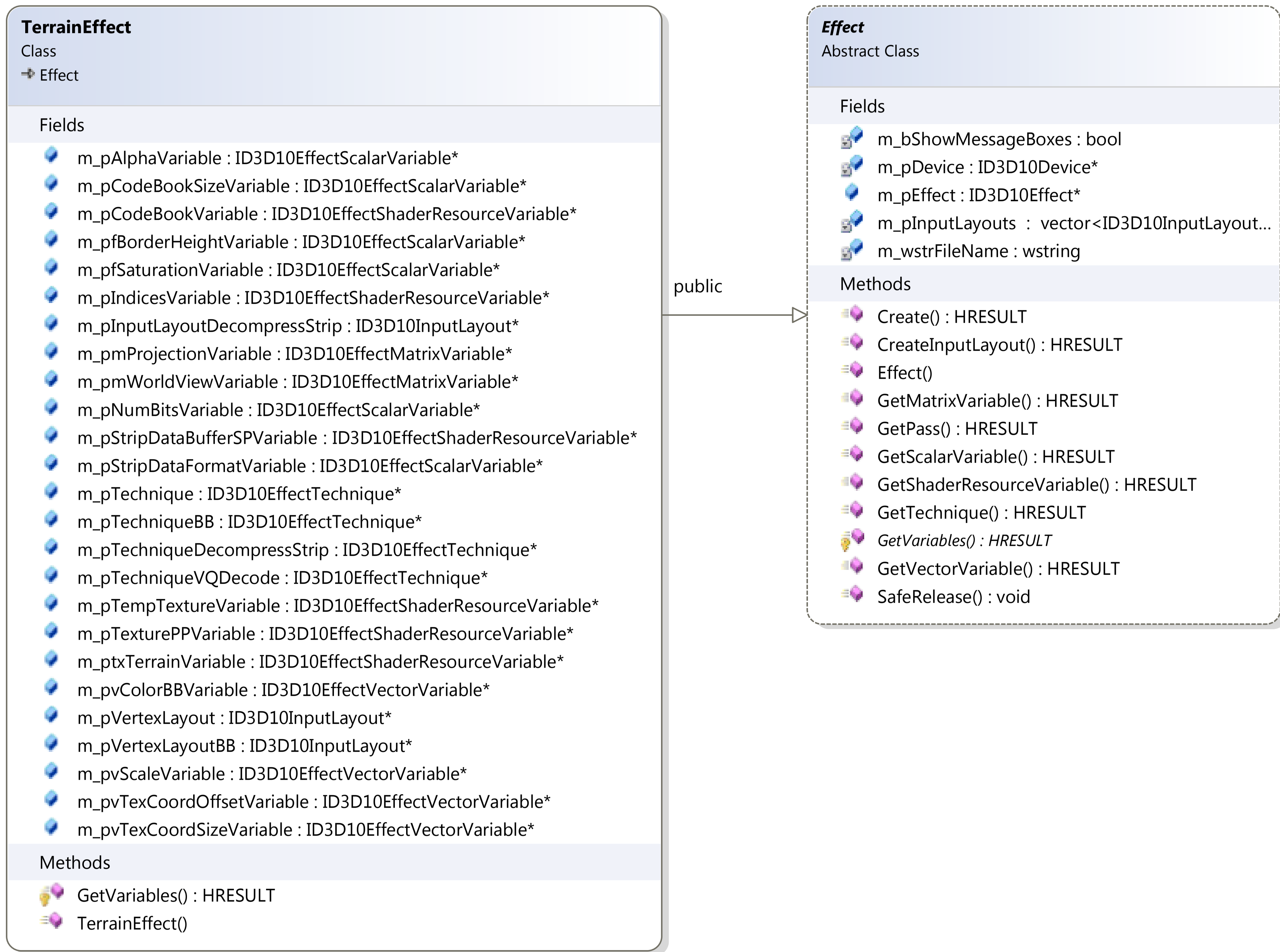


Figure 3.6: Class diagram of the original effect classes

4 OpenGL Port

Now that the reader is familiar with the general concepts and specific differences between OpenGL and Direct3D 10, and also knows the general architecture of Terrain3D, the porting to OpenGL can be discussed.

4.1 Overview

As explained in Section 3.3 on page 22 Terrain3D has some very tight coupling with DirectX. Because of this, porting becomes more difficult, as many areas need be changed and adapted. Moreover, the various differences, big and small, between Direct3D and OpenGL make careful planning a necessity.

Simply starting to replace all calls to Direct3D with calls to OpenGL is impossible and keeping the project in an uncompilable state for a long period of time is not an option. Debugging would take an unacceptable amount of time as well, because there would be no way to verify that the changes were correct.

4.1.1 Goals

The following goals were set for the porting phase:

- the code should be compilable and running correctly most of the time,
- the Direct3D version and the OpenGL version should compile from the same codebase,
- changes to the graphics code should become isolated to a few files instead of being spread over the whole project,
- the porting phase should be dividable into many small verifiable steps.

All these goals aim to simplify debugging and testing. They make it easier to spot bugs early on, when they are still identifiable with little effort.

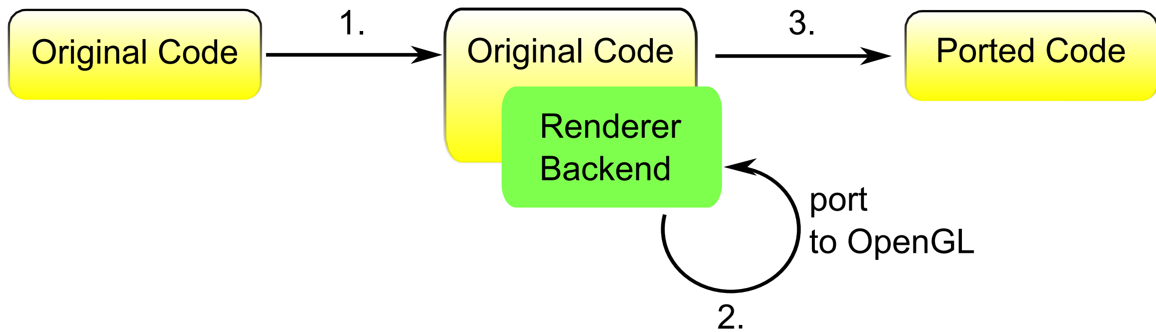


Figure 4.1: The three steps for porting Terrain3D

4.1.2 Concept

The goals above point to a straightforward concept (see also Figure 4.1 on the next page):

1. all Direct3D calls and object pointers are wrapped in a new renderer backend
2. the new renderer backend is ported to OpenGL and tested
3. Terrain3D is linked against the new backend and debugged

The first step can be implemented by gradually moving Direct3D calls to the wrapper with small code transformations. This means that incremental changes can be used, and these can be verified easily. This also keeps the project compilable while simplifying the port to OpenGL in the second step. All in all this makes porting easier and lowers the chance for bugs.

4.1.3 Renderer Backend

The renderer backend is a set of classes which encapsulates Direct3D objects and calls. It used to centralize the areas of change for the port to OpenGL later on. There are three important design requirements:

1. KISS[1],
2. renderer backend code and unchanged Direct3D code can coexist in the codebase, and
3. compiler errors are preferred over runtime errors.

These goals deserve more explanation: it's important not to write a generic Direct3D/OpenGL wrapper, because that would be a behemoth of a task. Rather only the

[1] Keep It Simple, Stupid

subset of Direct3D that is used by Terrain3D is wrapped. This reduces the amount of debugging required and also minimizes the performance penalty as the code can be more specific and to the point.

During the port it is important that incremental changes are possible. Thus the renderer backend cannot be a black box, but needs to allow access to the Direct3D objects it is supposed to hide. When everything is ported, these access methods won't be needed anymore[2].

An advantage of compiler errors versus runtime errors is that they appear earlier and more reliably, thus reducing the testing required (see [McC04] for an exhaustive treatment of the benefits). So, for example, instead of using flags to specify the stages a texture can be bound to, multiple classes can be used. Nonetheless duplicate code can be avoided using inheritance. This moves some of the usual flag error checking to the compiler and helps readability, because now it is easy to determine exactly what a texture is used for.

4.1.4 Effect Files

Porting the effect files is a problem. They cannot be wrapped incrementally and all need to be ported together in the second step. To ease that change and honor the mentioned goals, I decided to implement a simple effect file format for GLSL programs and the corresponding compiler. Because this is mostly independent of the other code, it will be examined in detail first.

As mentioned in Section 3.3 on page 22 Terrain3D originally used two effect files that contain many different techniques. The main difficulties with this are: the lack of support for effect files or a similar concept in OpenGL, and the huge amount of HLSL code that needs to be ported to GLSL.

The latter can be mitigated by separating the *terrain.fx* effect file into one file per technique (see Table 3.5 on page 23 for an overview over all techniques). Among the techniques only few share variables are shared, so only little code has to be duplicated during this refactoring. However, porting becomes easier, since one effect file at a time can be ported and tested separately. *terrain3d.fx* does not have to be ported, because it is only referenced in the DXUT-specific application code, which is discarded anyway.

Some of the techniques have multiple passes and for each pass one shader program has to be created and set up. Furthermore, for every shader program all uniform locations have to be queried and stored, and when a shader variable of an effect is changed, the uniforms of each program have to be updated[3]. This can be automated easily while it is tedious to write all the code above by hand.

[2] It turns out that it is not worth porting all the code, because some is not shared between the different versions, so it's beneficial that that code does not have to be fully wrapped.

[3] The uniform buffers introduced in OpenGL 3.1 are a major improvement in this area.

The effect compiler is a code generator that transforms a GLSL effect file into a C++ source/header file pair which can be compiled and linked with the project code. The ANTLR parser generator and the StringTemplate library from Terence Parr were used to the reduce development time of the compiler.

Before examining the implementation further it makes sense to take a look at the wrapper code in Terrain3D and the GLSL effect file format.

4.2 `Effect` Class and Helper Classes

The **`Effect`** class, which resides in the **`Renderer`** namespace to avoid collision with unwrapped code, wraps access to a few Direct3D effect classes mentioned in Section 2.4.6 on page 11: **`ID3D10Effect`**, **`ID3D10EffectTechnique`** and **`ID3D10EffectPass`**. Only one technique per effect is supported, which is not a problem after the refactoring described above (where every effect file was split into one file per technique). Each effect file has one specialized effect class that represents the effect file in the code.

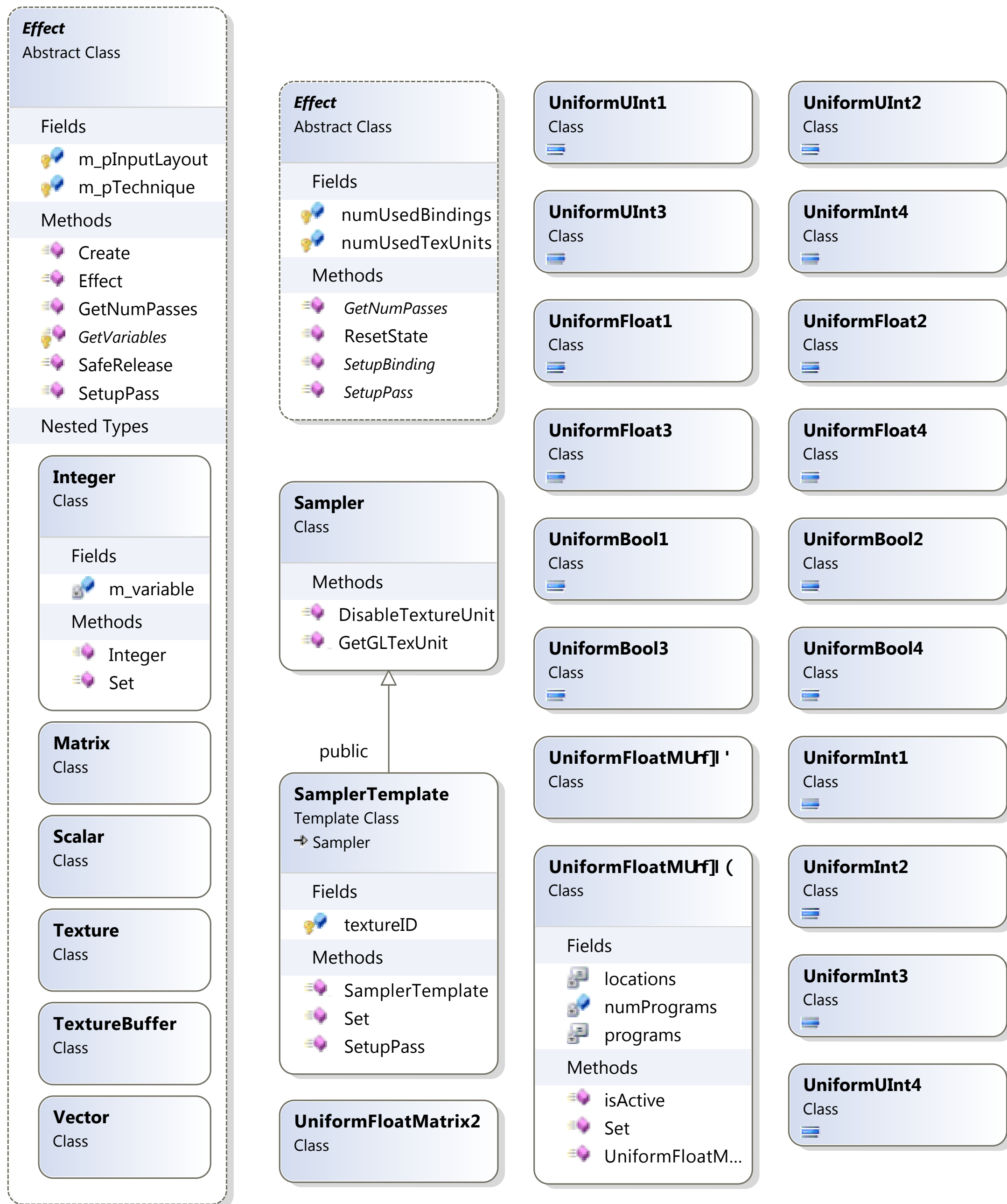
The Direct3D and the OpenGL implementation are quite different internally. The differences are hidden in **`Device`**[4] and specialized effect classes however. In both versions **`Effect`** has the public methods **`GetNumPasses`** and **`SetupPass`**. The former returns the number of passes the effect file contains—remember that it always only contains one technique—and the latter sets up the device for a specific pass. The specialized effect classes contain member variables that correspond to the shader variables, respectively uniforms, of the effect files. Each such member variable has a **`Set`** method to change its value. Figure 4.2 on the facing page shows the class diagrams of both implementations. They will be explained in the next sections.

Direct3D implementation

In the Direct3D implementation **`Effect`** has got an abstract **`GetVariables`** method, which the specialized effect classes implement to initialize the member variables and to create the input layout used by the effect (if there is one).

The **`Effect`** class also defines helper classes for the different shader variable types which wrap the **`ID3D10Effect*Variable`** classes. To reduce the amount of duplicate code, a multi-line macro is employed. See Listing 4.3 on the next page for its code.

[4]**`Device`** wraps **`ID3D10Device`** in the renderer backend, and will be discussed later in Section 4.4 on page 40.



(a) Direct3D effect classes (b) OpenGL effect classes

Figure 4.2: Class diagram of **Effect** and its helper classes (without the device classes)

Listing 4.3: Shader variable helper macro definition (for the Direct3D version)

```
#define ShaderVariableClass( name, variableType, valueType,
   valueName, setExpression ) \
    class name {\
    public:\
        void Set( const valueType const valueName ) {\
            setExpression ;\
        }\
\
        name( variableType *&variable ) : m_variable( variable )
           {}\
        \
    private:\
        variableType *&m_variable;\
    }

    ShaderVariableClass( Texture,
       ID3D10EffectShaderResourceVariable,
       RenderTargetShaderTexture2D *, texture,
       m_variable->SetResource( texture->GetShaderResourceView()
       ) );
    ShaderVariableClass( TextureBuffer,
       ID3D10EffectShaderResourceVariable, ShaderBuffer *,
       buffer, m_variable->SetResource(
       buffer->GetShaderResourceView() ) );
    ShaderVariableClass( Scalar, ID3D10EffectScalarVariable,
       float, value, m_variable->SetFloat( value ) );
    ShaderVariableClass( Integer, ID3D10EffectScalarVariable,
       int, value, m_variable->SetInt( value ) );
    ShaderVariableClass( Matrix, ID3D10EffectMatrixVariable,
       float *, matrix, m_variable->SetMatrix( (float*) matrix )
       );
    ShaderVariableClass( Vector, ID3D10EffectVectorVariable,
       float *, vector, m_variable->SetFloatVector( (float*)
       vector ) );
#undef ShaderVariableClass
```

OpenGL implementation

In the OpenGL implementation **Effect** has two additional public methods: **SetupBinding** and static **ResetState**. **SetupBinding** sets up the vertex array pointers for the input layout used by the effect, and **ResetState** resets the active shader program and disables all used vertex pointers and texture units.

The helper classes which wrap access to the uniform variables of an effect are a bit more complex as OpenGL does not have a class that can simply be wrapped and thus more state has to be stored in the objects themselves. Each pass in an effect corresponds to one distinct shader program and every shader program can have a different uniform location for a specific uniform variable. This information has to be stored in the wrapper objects: a uniform class keeps an array with the handles of all shader programs it is used in, and an array with the locations of the uniform variable it represents. To avoid duplicate code and keep it simple[5], several preprocessor macros are used again to declare the various uniform classes. See Listing 4.4 on page 31 for the main macro and how it is used to define uniform classes (except for the Sampler ones).

Sampler uniforms are wrapped using two classes: **Sampler2D** and **SamplerBuffer**. They are actually typedefs of the **SamplerTemplate** template class, which in turn is derived from **Sampler**.

Sampler provides one public static method: **DisableTextureUnit**. It is used to disable a texture unit after it has been used by a sampler.

SamplerTemplate implements the **Set** method and a **SetupPass** method. The former is required for the code to be compatible with the Direct3D version. The latter is used by the GLSL effect file code to bind a sampler in OpenGL.

The classes described in this section are used by the GLSL effect compiler extensively. The next section looks at the effect file format and introduces the concepts based on a simple example.

4.3 Effect Files

4.3.1 GLSL Effect File Format

The effect file format only contains the features necessary to port Terrain3D's effect files. While it is minimalistic, the grammar itself can easily be extended to support more features of the HLSL effect file format.

Listing 4.5 shows a very simple GLSL effect. The shader is written against the GLSL 1.20 specs - hence the version 120 line. It defines several uniform variables, and unlike the

[5]Before, template classes were used which resulted in quite complicated and messy code.

Listing 4.4: Uniform helper macro definition (for the OpenGL version)

```
#define UniformClass( className, valueType, uniformSetter ) \
    class className { \
    public: \
        className( unsigned numPrograms, const GLuint *programs,
            const GLint *locations ) \
            : numPrograms( numPrograms ), programs( programs ),
              locations( locations ) { \
        } \
 \
        void Set( const valueType value ) { \
            for( unsigned i = 0 ; i < numPrograms ; i++ ) { \
                if( isActive( i ) ) { \
                    GLuint program = programs[ i ]; \
                    GLuint location = locations[ i ]; \
                    uniformSetter; \
                } \
            } \
        } \
 \
        bool isActive( unsigned pass ) { \
            return locations[ pass ] != -1; \
        } \
 \
    private: \
        unsigned numPrograms; \
 \
        const GLint *locations; \
        const GLuint *programs; \
    }

#define UniformSingleClass( className, valueType, shortie) \
    UniformClass( className, valueType,
       glProgramUniform1##shortie##EXT( program, location, value
       ) )
#define UniformVectorClass( className, valueType, shortie,
   compCount ) \
    UniformClass( className, valueType *,
       glProgramUniform##compCount##shortie##vEXT( program,
       location, 1, value ) )

UniformClass( UniformFloatMatrix2, float *,
   glProgramUniformMatrix2fvEXT( program, location, 1, false,
   value ) );
// other classes ...
```

Listing 4.5: *tileRenderingBB.gfx* (adapted)

```
version 120

uniform {
    // vertex shader uniforms
    mat4x4 mWorldView;
    mat4x4 mProjection;

    // fragment shader uniforms
    vec4 vColorBB;
}

inputlayout { position: FLOAT[3] }

pass {
    vertex {
        in vec4 position;

        void main() {
            gl_Position = mProjection * mWorldView * position;
        }
    }
    fragment {
        void main() {
            gl_FragColor = vColorBB;
        }
    }
}
```

HLSL effect file format, the input layout is declared inside the effect file, too. In this case the input layout maps three floats to the **position** attribute of the vertex shader. Usually the input layout is kept separate from the shader programs, because different input layouts might be used with the same shader, but Terrain3D does not use this. Thus the input layout specification can be moved into the effect file. One GLSL effect file contains exactly one technique, which simplifies the format and the compiler. Multiple passes can be defined using pass { ... } blocks. In the shader above only one pass is defined and the vertex and fragment shader code is specified in the respective vertex { ... } and fragment { ... } blocks. To declare a geometry shader a slightly more advanced syntax is used:

geometry(**input**, **output**, **maxEmitVertices**){ **glslCode** }

input and **output** specify the input and output primitive type. The accepted values are simply the OpenGL primitive type constants without the initial **GL_** prefix: **POINTS**, **TRIANLGES**, **TRIANGLE_STRIP**, etc.

To facilitate code reuse between different passes, vertex { ... }, fragment { ... } and geometry { ... } blocks can be defined outside of pass { ... } blocks. They are included in-order before the code specified in the pass { ... } blocks. Additionally a shared {...} block is supported, whose code is included in every shader type.

This can be used together with the C preprocessor to write effect files that are very similar in structure to the Direct3D ones (compare this to HLSL version in Listing 2.9 on page 12):

Listing 4.6: tileRenderingBB.gfx with global code blocks and preprocessor macros

```
#define SetVertexShader( callExpr ) vertex { void main() {
   callExpr; } }
#define SetFragmentShader( callExpr ) fragment { void main() {
   callExpr; } }

version 120

uniform {
    // vertex shader uniforms
    mat4x4 mWorldView;
    mat4x4 mProjection;

    // fragment shader uniforms
    vec4 vColorBB;
}

inputlayout { position: FLOAT[3] }

vertex {
    in vec4 position;

    void VSBB() {
        gl_Position = mProjection * mWorldView * position;
    }
}
```

```
fragment {
    void PSBB() {
        gl_FragColor = vColorBB;
    }
}

pass {
    SetVertexShader( VSBB );
    SetFragmentShader( PSBB );
}
```

While this seems pointless for such short shaders, it makes code reuse easier for bigger ones. Furthermore the structure of the original HLSL shaders can be preserved this way, which makes comparing GLSL and HLSL code in effect files easier while debugging the different versions of Terrain3D.

A fragDataName **variableName** statement can be used to specify the output variable of the fragment shader in a pass and a feedback **variableA**, **variableB**, ... statement can be used to declare the output variables of a shader that are used in transform feedback mode.

A pass definition can also include state blocks, which specify whether depth testing and/or stencil testing need to be enabled or disabled for the pass. Multiple state blocks can be specified, too. Blocks specified later have priority over ones specified earlier. By default depth testing is enabled and stencil testing is disabled (see also Listing 4.12 on page 37).

Listing 4.7: Feedback and state block example

```
[...]

pass {
    SetVertexShader( VSDecompressStrip() );
    SetGeometryShader( POINTS, POINTS, STRIP_LENGTH * 3,
       GSDecompressStrip() );

    feedback { vertices }
    state {
        depthTest: false
    }
}
```

Because the normal C preprocessor is used, the usual **#** character to introduce a GLSL preprocessor command cannot be utilized. Instead **$** is used for GLSL preprocessor commands. This can result in confusing code if both preprocessors are used for macros, but it is needed for GLSL **#pragma**s and **#extension**s.

Listing 4.8: $ preprocessor example

```
shared {
    $extension GL_EXT_gpu_shader4: require
}
```

See Figure 4.9 on the following page for the full grammar used by the parser.

4.3.2 ANTLR Grammar Definition

The compiler uses the ANTLR parser generator to parse effect files. ANTLR is a powerful tool that creates recursive-descent parsers. The rules are specified using a BNF-like syntax. ANTLR can be used to create both parsers and lexers. In the examples that follow, it is sufficient to know, that lexer rules have all-capitals names, while parser rules do not. For a good introduction and reference for ANTLR, see [Par07]; [Para] contains lots of good tutorials, too.

The only real issue that arose during the creation of the effect file grammar was figuring out how to parse GLSL code blocks. Code blocks should be read in as-is and the grammar should only care about nested braces, ie the grammar should contain GLSL-specific rules. [Par07] contains an example of a lexer rule that reads in nested code blocks. The rule also takes a parameter to decide whether to strip the braces from the text or not. It is shown in Listing 4.10 on page 37.

The problem with this solution is that the actual grammar also needs to parse single curly braces for pass, state and uniform definitions (see Listing 4.9 on the following page). The lexer ANTLR generates, however, always chooses the token type, ie lexer rule, that matches most text and, if there is a tie, then the one that is defined first wins—see [ANT] for a detailed description of the lexer's behavior. Because of this the **CODE** lexer rule always wins compared to the single curly brace, even if a single curly brace is expected. The only way to fix this is to make the **CODE** lexer rule a parser rule instead. A new problem arises then: since `~('{' |'}'))*` is now part of a parser rule, it matches any lexer rule, ie token type, that is not a curly brace, instead of any character except for curly braces. This means that any character used in GLSL code and not in the grammar will cause it to fail.

The solution is to add a new lexer rule after all other lexer rules that matches any character. Because it comes last and only matches one character, all real lexer rules are matched

Listing 4.9: Simplified GLSL effect file grammar (exported from ANTLRWorks)

```
effectFile
    :   ('version' versionNumber=NUMBER)?
        (
              'shared' commonCode=codeBlock
            | 'vertex' vertexCode=codeBlock
            | 'fragment' fragmentCode=codeBlock
            | 'geometry' geometryCode=codeBlock
            | 'uniform' '{' uniformDeclaration* '}'
            | /*max one per file*/ 'inputlayout' '{'
                    (inputDeclaration (',' inputDeclaration)*)?
                '}'
        )*
        passDefinition+
    ;

uniformDeclaration
    : type=ID name=ID ';'
    ;

inputDeclaration
    :   name=ID ':' type=ID ('[' size=NUMBER ']')?
    ;

passDefinition
    :   'pass' '{'
        (
              /*max one per pass*/ 'vertex' vertexShaderCode=codeBlock
            | /*max one per pass*/  'fragment' fragmentShaderCode=codeBlock
            | /*max one per pass*/  'geometry' '('
                    inputType=ID ',' outputType=ID ',' maxEmitVertices=parameter
                ')' geometryShaderCode=codeBlock
            | /*max one per pass*/ 'feedback' '{'
                    ( transformFeedbackVarying=ID (',' transformFeedbackVarying=ID)* )?
                '}'
            | /*max one per pass*/ 'fragDataName' fragDataNameID=ID
            | stateDefinition
            | ';' /* eat ;s */
        )*
        '}'
    ;

stateDefinition
    : 'state' '{'
        (
          ('depthTest' ':' toggle=bool)
        | ('stencilTest' ':' toggle=bool)
        )*
       '}'
    ;

bool
    : 'true' | 'false'
    ;

parameter
    : '(' .* ')'
    ;

codeBlock
    : '{' .* '}'
    ;
```

Listing 4.10: Lexer rule example that reads in code blocks (Source: [Par07, 108])

```
fragment
CODE[boolean stripCurlies]
    :   '{' ( CODE[stripCurlies] | ~('{' | '}') )* '}'
        {
            if ( stripCurlies ) {
                setText(getText().substring(1, getText().length()));
            }
        }
    ;
```

first and thus win, and it is only matched for characters that are not used elsewhere in the grammar.

4.3.3 Compiler Code

The compiler is written in Java. It uses the ANTLR parser generator[6] to parse effect files, and the StringTemplate library[7] to generate C++ code.

The ANTLR grammar parses the effect file and stores its content in several Java classes (see Figure 4.11 on page 38 for a class diagram):

Binding is the container structure of an input binding specified in inputlayout. It contains fields for the name, size and type of a binding.

Pass is the container for a pass {...} block. It contains fields to store all data specified in the grammar.

Pass.State is a static subclass of `Pass` and contains the state variables. See Listing 4.12 for the default values.

Pass.GeometryShader contains fields for the geometry shader code and shader parameters in a geometry(**input**, **output**, **maxEmitVertices**){ ... } statement.

SharedCode stores a global code block or uniform declarations.

Uniform stores the type and name of a uniform variable declared in a uniform { ... } block.

The classes `SimpleGLSLEffectFileLexer` and `SimpleGLSLEffectFileParser` are generated by ANTLR. The class `SimpleGLSLEffectCompiler` contains the 'glue' code

[6] ANTLR is available for download at http://www.antlr.org
[7] StringTemplate is available for download at http://www.stringtemplate.org/

SharedCode
Class

Fields
code : String
type : Type
uniforms : List<Uniform>

Methods
isForAllShaders() : boolean
isForFragmentShaders() : boolean
isForGeometryShaders() : boolean
isForVertexShaders() : boolean
isUniformBlock() : boolean
SharedCode() (+ 1 overload)

Nested Types

Type
Enum

COMMON
VERTEX
FRAGMENT
GEOMETRY
UNIFORM_BLOCK

Pass
Class

Fields
fragDataName : String
fragmentShader : String
geometryShader : GeometryShader
stateBlock : State
transformFeedbackVaryings : List<String>
vertexShader : String

Methods
Pass()

Nested Types

GeometryShader
Class

Fields
code : String
inputType : String
maxEmitVertices : String
outputType : String

Methods
GeometryShader()

State
Class

Fields
depthTest : boolean
stencilTest : boolean

SimpleGLSLEffectFileParser
Class
→ Parser

Methods
effectFile() : effectFile_return
SimpleGLSLEffectFileParser() (+ 1 overload)

Nested Types

effectFile_return
Class
→ ParserRuleReturnScope

Fields
bindings : List<Binding>
passes : List<Pass>
sharedCodes : List<SharedCode>
uniforms : List<Uniform>
version : Integer

Uniform
Class

Fields
name : String
type : String

Methods
isSampler() : boolean
Uniform()

SimpleGLSLEffectCompiler
Class

Methods
main() : void
printHelp() : void

Nested Types

StringFormatRenderer
Class
→ AttributeRenderer

Methods
toCString() : String
toString() : String (+ 1 overload)

SimpleGLSLEffectFileLexer
Class
→ Lexer

Binding
Class

Fields
name : String
size : int
type : String

Methods
Binding()
isInteger() : boolean

Figure 4.11: Class diagram of the GLSL effect compiler

Listing 4.12: Pass.State definition

```java
public static class State {
    public boolean depthTest = true;
    public boolean stencilTest = false;
};
```

that parses the effect file and uses the StringTemplate library to generate the output C++ code.

4.3.4 StringTemplate Code

StringTemplate is another Java library created by Terence Parr, which provides a flexible way to create text output from input data using string templates. The only documentation available at the moment can be found at the library's homepage [Parb]. A string template is a text string that contains additional tags which are replaced at runtime with data from various sources. Additionally conditional expressions, templates that take parameters, and expansion of lists and arrays are supported. The GLSL effect compiler uses all these features to create the output C++ code from the data classes filled in by the parser.

The string template creates an effect class with the name of the GLSL effect file and the suffix **Gfx**. The class inherits from the abstract **Effect** class, see Figure 4.2 on page 28, and implements the **SetupBinding** and **SetupPass** methods as described in Section 4.2 on page 27. It stores handles to all shader programs (one per pass) and the uniform locations for each uniform variable and for each shader program. For each uniform variable a public member is declared using the wrapper classes described in Section 4.2 on page 27.

SetupBinding sets the vertex attribute pointers for the effect as specified by the inputlayout { ... } block in the effect file and enables the required vertex attribute arrays. It also disables vertex attribute arrays that are no longer needed on the other hand.

SetupPass sets the state for the pass as specified in the state {...} blocks (or the default values) and calls **SetupPass** for all used samplers. It also disables texture units that aren't used anymore.

Both methods use static member variables in **Effect** to keep track of the number of used texture units and vertex attribute arrays.

They use a trick to overcome a limitation of the StringTemplate library: although it has a tag **<i0>** which returns the index of the current element when looping over a list during expansion, there is no way to filter the results beforehand. Only conditionals can be used to prevent output for unused samplers in **SetupPass**. If **<i0>** was used for selecting

Listing 4.13: Code excerpt from the decompression effect showing the index workaround

```
// set the texture bindings
GLuint texUnit = 0;

if( uniformLocations[ 0 ][ 7 ] != -1 ) {
    stripHeadersBuffer.SetupPass( 7, texUnit ); texUnit++;
}
if( uniformLocations[ 1 ][ 7 ] != -1 ) {
    stripDataBuffer.SetupPass( 7, texUnit ); texUnit++;
}
if( uniformLocations[ 2 ][ 7 ] != -1 ) {
    texPingPong.SetupPass( 7, texUnit ); texUnit++;
}
```

the texture unit, the used texture unit indices would not be tightly packed, leaving gaps for uniform variables that are not samplers. This is a problem: if there are more uniform variables than texture units and the last declared uniform variable is a sampler, the code would fail to set the texture unit. The only workaround around for this is to define a variable in the code and use it as an index that is incremented each time a sampler has been set up. See Listing 4.13 for a small code excerpt showing this trick. It is also used when the sampler uniforms are initialized.

Create method creates the shader and program objects for each pass, compiles the shader sources and links them. It also queries all uniform locations and informs the uniform and sampler objects.

The fragment output variable is set, and if transform feedback is used, the output varying variables are also set.

4.4 **Device** Class and Helper Classes

4.4.1 Class Hierarchy

The renderer backend wraps all calls to Direct3D in terrain-specific code (that is, application code, which is not reused in the ported version, is ignored) and is the main area of change between the Direct3D and OpenGL version. The porting of the effect files and the respective classes is described in Section 4.1.4 on page 27. Here it suffices to know that all effects inherit from an abstract **Effect** class. All classes presented here are part of a separate **Renderer** namespace in Terrain3D to avoid collisions with existing code.

See Figure 4.14 on the facing page for an overview of the class hierarchy in the renderer

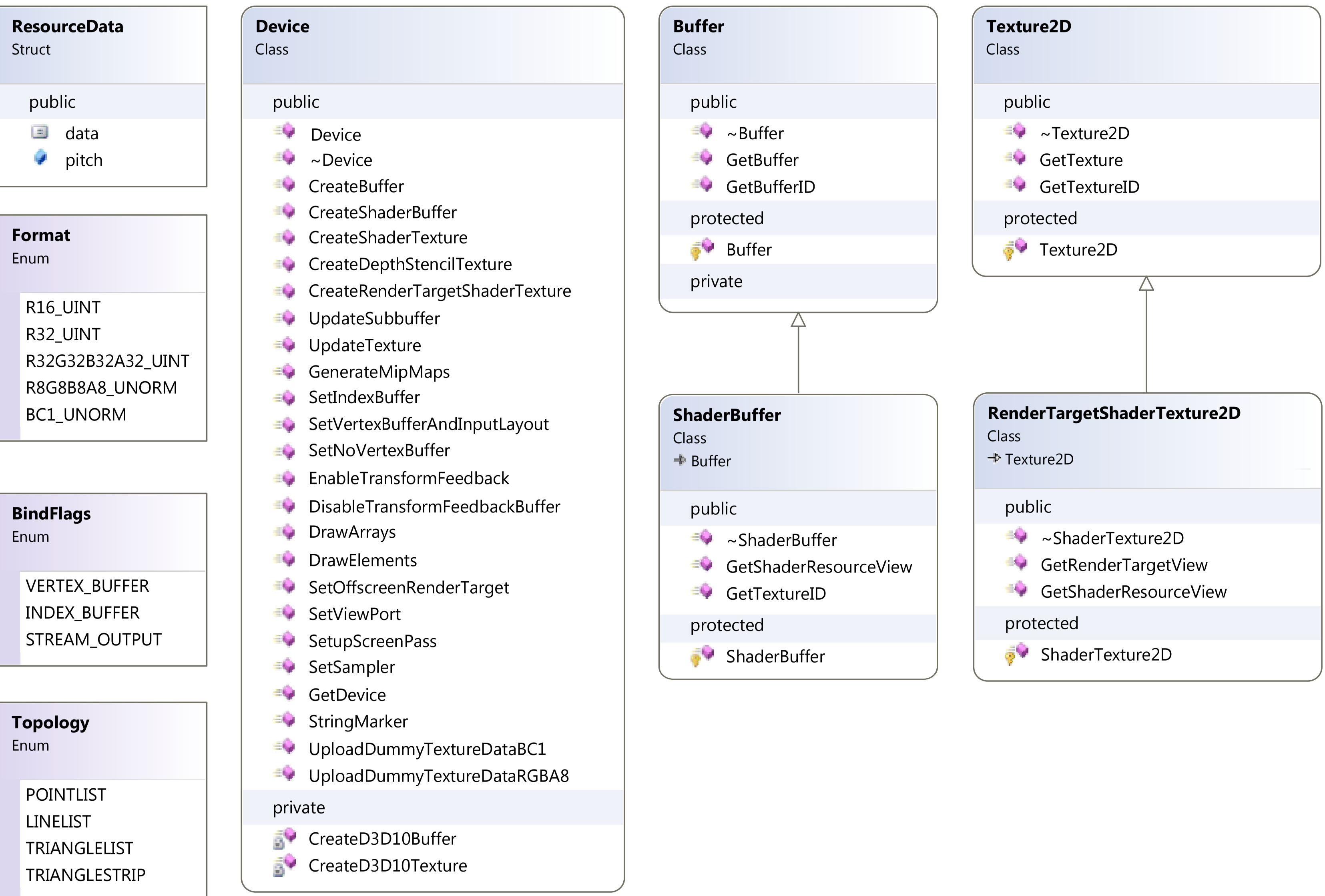


Figure 4.14: Class diagram of **`Device`** and its helper classes (without the effect classes)

Listing 4.15: **Format** definition

```
#ifdef USEDX10
#   define DXGL( a, b ) a
#else
#   define DXGL( a, b ) b
#endif
enum Format {
    R16_UINT = DXGL( DXGI_FORMAT_R16_UINT, GL_R16UI ),
    R32_UINT = DXGL( DXGI_FORMAT_R32_UINT, GL_R32UI ),
    R32G32B32A32_UINT = DXGL( DXGI_FORMAT_R32G32B32A32_UINT,
        GL_RGBA32UI ),
    R8G8B8A8_UNORM = DXGL( DXGI_FORMAT_R8G8B8A8_UNORM, GL_RGBA8
        ),
    BC1_UNORM = DXGL( DXGI_FORMAT_BC1_UNORM,
        GL_COMPRESSED_RGBA_S3TC_DXT1_EXT )
};
```

backend. **Device** wraps all calls to **ID3D10Device**. **Buffer** and **ShaderBuffer** wrap **ID3D10Buffer** pointers. The **Shader** prefix of **ShaderBuffer** refers to the bind flags of the object: objects of this type are always created with **D3D10_BIND_SHADER_RESOURCE** bind flag and have a valid pointer to an **ID3D10ShaderResourceView** object. **Buffer** objects on the other hand do not have such view object and are always created without the bind flag.

Texture2D and **RenderTargetShaderTexture2D** wrap **ID3D10Texture2D** pointers and and similarly **RenderTargetShaderTexture2D** is created with the **D3D10_BIND_SHADER_RESOURCE** and **D3D10_BIND_RENDER_TARGET** flags.

In the OpenGL version **Buffer** and **ShaderBuffer** wrap the unsigned integer handles to buffer and texture objects and cache the size of the buffer. Likewise **Texture2D** and **RenderTargetShaderTexture2D** wrap the unsigned integer handles of texture and framebuffer objects and cache properties like the format of the texture, and its width and height.

Format, **BindFlags** and **Topology** wrap enumerations that the methods of **Device** use. As it is determined at compile time which graphics library is used, the values of the enumerations map to the values of the respective enumerations in either Direct3D or OpenGL. See Listing 4.15 on page 42 for the definition of **Format**. **Format** also only enumerates the formats actually used by Terrain3D.

4.4.2 Device Methods

There are two versions of the methods in `Device`: a Direct3D one, which was written during the first step of the port (see Figure 4.1 on page 26), and an OpenGL one which was written afterwards. Actually there are two OpenGL versions: one uses the direct state access extension described in Section 2.5.2 on page 15 and the other one uses normal OpenGL functions. The latter is mainly used for debugging Terrain3D with gDEBbuger[8].

Preprocessor `#ifdef`s are used to keep the different versions in the same file. The preprocessor constant `USEDX10` is used to switch between the Direct3D and OpenGL version, and the preprocessor constant `GDEBUGGER` is used to disable or enable support for the direct state access extension.

The method names of `Device` are inspired by both the Direct3D and OpenGL API. The methods `SetIndexBuffer` and `SetVertexBufferAndInputLayout` are an example of the former and the draw methods `DrawArray` and `DrawElements` of the latter.

`SetVertexBufferAndInputLayout` is interesting in another aspect, too. It performs two operations, which are independent in Direct3D: setting the active vertex buffers[9] and setting the input layout. In OpenGL, however, the vertex buffer has to be bound before the vertex pointers can be set, so the two calls have been merged into one method to ensure that they are always called in the correct order. See Listing 4.16 on the facing page for the code.

`DrawArrays` and `DrawElements` include the topology as parameter. Direct3D has a separate `IASetPrimitiveTopology` method, but OpenGL does not, and again the operations are merged to keep the OpenGL version simple. See Listing 4.17 on the next page for the implementation.

Finally there are two methods which are very Terrain3D-specific: `SetSampler` and `SetupScreenPass`.

SetSampler is used to set the minimum LOD level for transitions between different tile levels. In Direct3D a continuous fade is not possible, because discrete, immutable sampler state objects have to be used. In OpenGL on the other hand the texture parameter can simply be changed.

SetupScreenPass sets up the rendering for a screen pass, that is drawing a single triangle that covers the whole screen/framebuffer. In Direct3D it disables the vertex buffer, because the shader can use the vertex id to choose the coordinates. But in OpenGL this is not possible, so a special vertex buffer has to be used. See Listing 4.18 on the facing page for the implementation of both versions.

[8] gDEBUGGER is a debugger specifically targeted for debugging OpenGL applications. More information can be found on http://www.gremedy.com

[9] Terrain3D always only uses one vertex buffer though, so the function also only sets on vertex buffer

Listing 4.16: **SetVertexBufferAndInputLayout** implementation

```
void Device::SetVertexBufferAndInputLayout( Buffer *buffer,
   unsigned stride, const Effect &effect, unsigned offset /*=
   0*/ ) {
    assert( buffer != NULL );

#ifdef USEDX10
    m_pDevice->IASetVertexBuffers( 0, 1, &buffer->GetBuffer(),
       &stride, &offset );
    m_pDevice->IASetInputLayout( effect.m_pInputLayout );
#else
    glBindBuffer( GL_ARRAY_BUFFER, buffer->m_bufferID );
    checkGLError();

    effect.SetupBinding( stride, offset );
#endif
}
```

Listing 4.17: **DrawArrays** implementation

```
void Device::DrawArrays( Topology topology, unsigned count ) {
#ifdef USEDX10
    m_pDevice->IASetPrimitiveTopology(
       (D3D10_PRIMITIVE_TOPOLOGY) topology );
    m_pDevice->Draw( count, 0 );
#else
    glDrawArrays( topology, 0, count );
    checkGLError();
#endif
}
```

Listing 4.18: **SetupScreenPass** implementation

```
#ifndef USEDX10
void Device::InitScreenPassVertexBuffer() {
    static float positions[3][4] = {
        {3.0f, -1.0f, -0.5f, 1.0f},
        {-1.0f, -1.0f, -0.5f, 1.0f},
        {-1.0f, 3.0f, -0.5f, 1.0f}
    };
#   ifdef GDEBUGGER
    {
        ArrayBufferScope scope( screenPassVertexBufferID );
        glBufferData( GL_ARRAY_BUFFER, sizeof( positions ),
            &positions, GL_STATIC_DRAW );
    }
#   else
    glNamedBufferDataEXT( screenPassVertexBufferID, sizeof(
        positions ), &positions, GL_STATIC_DRAW );
#   endif
}
#endif

// either use vertex id in the shader in d3d or a simple vertex
   array in ogl (vertex shader is just pass-through in that case)
void Device::SetupScreenPass() {
#ifdef USEDX10
    SetNoVertexBuffer();
#else
    glBindBuffer( GL_ARRAY_BUFFER, screenPassVertexBufferID );
    glVertexAttribPointer( 0, 4, GL_FLOAT, false, sizeof(float)
        * 4, NULL );
#endif
}
```

The non-direct-state-access version is only used for debugging purposes. It does not have to be optimized, and because of this the code can simply backup binding points by querying OpenGL directly, execute the code, and then restore the old state. For this a few helper classes like **ArrayBufferScope** are used, which save the OpenGL state that is going to be changed, set the new state and restore it, when the instance is destructed. There are four such helper classes: **FrameBufferScope**, **TextureScope**, **TextureBufferScope** and **ArrayBufferScope**.

Three methods are simplified by specifically targeting Terrain3D with the wrapper code: **SetIndexBuffer**, **DrawElements** and **EnableTransformFeedback**.

SetIndexBuffer and **DrawElements**[10] always assume that the index buffer uses 32-bit indices. This reduces the complexity because this information is used in the Direct3D version of **SetIndexBuffer** and in the OpenGL version of **DrawElements**. Otherwise **Device** would have to store this information in a member variable to use it at the right time.

EnableTransformFeedback makes use of the fact that Terrain3D only streams out point primitives. Again an additional member variable would be required otherwise.

There is also one method whose OpenGL version is a lot more complex than the Direct3D version: **UpdateTexture**. It uploads a texture subresource to a texture resource, ie a mipmap of a texture's mipmap chain. The texture data can be uploaded with a certain pitch value. The pitch value specifies how many bytes lie between two consecutive texture rows (including the actual data).

However, OpenGL only supports uploading compressed texture data with a pitch that matches the canonical length of a texture row for the respective format[11]. Coincidentally Terrain3D only uses compressed texture data with gaps when using the *Shared DXT1/BC1 textures* texture decompression mode, which described in Section 3.2 on page 21.

Therefore the OpenGL version of **UploadTexture** has to check for this case and handle it separately: it manually copies the texture data row by row into a temporary memory buffer, which is packed tightly. This tightly packed data can then be uploaded to OpenGL.

Another subtlety arises when uploading compressed textures in the DX1/BC1 format (which compresses the texture in blocks of 4x4 texels) and the texture width or height is smaller than 4. In that case a full 4x4 texture block has to be uploaded. The unneeded texels are ignored by OpenGL.

This concludes the discussion of **Device** and its helper classes. Even though the major

[10] **DrawElements** renders indexed primitives

[11] that is, there are no gaps between two consecutive rows or put differently: the texture data is packed tightly between rows

differences between the Direct3D and OpenGL version are concentrated in the renderer backend, some additional changes are necessary throughout the codebase.

4.5 Additional Changes in Terrain3D

4.5.1 Coordinate System

Direct3D uses a left-handed coordinate system, while OpenGL uses a right-handed coordinate system. Only two things have to be changed to accommodate this: the projection matrix has to be calculated for a right-handed coordinate system and the z-axis of the view matrix has to be inverted.

4.5.2 Indexed and Bufferless Drawing

It is generally not possible in OpenGL to render anything without supplying vertex data, because OpenGL only supplies a meaningful vertex id, when at least one vertex buffer is bound. There are two places in the Direct3D version of Terrain3D where indexed and bufferless drawing are used.

The first one is in the tile rendering code for the terrain. It uses indexed drawing to render the tile more efficiently as described in Chapter 3 on page 19. The OpenGL version of the code simply binds the buffer as vertex buffer and renders it non-indexed. This reduces the performance because the post-transform vertex cache cannot be used anymore, but I have not found a specification conform way to avoid this.

The second one is in the geometry decompressing code that uses multiple shader passes. It uses bufferless drawing and **`SV_VertexId`** to process data from two input textures. In the OpenGL version a vertex buffer that contains the indices, that Direct3D would create implicitly, is used for rendering.

In current NVIDIA drivers[12] it is possible to render outside the valid range of a bound vertex buffer. This can be used to implement indexed and bufferless drawing: you create a small dummy vertex buffer and bind it (this enables **`gl_VertexID`**), then you use either **`glDrawArrays`** for bufferless drawing or **`glDrawElements`** for indexed drawing.

According to the specifications the behavior of OpenGL is undefined in this case, but current NVIDIA drivers accept it and a noticeable performance increase is measured (see Section 6.1 on page 67). Terrain3D supports a third **`INDEXED_DRAWING`** preprocessor constant which switches between OpenGL-specification conform rendering and the method described above.

[12] October 2009

4.5.3 GLUT Library

The Direct3D version uses the DXUT library to deal with operating system calls, window management and event handling. DXUT only supports DirectX. For the OpenGL version the GLUT library is being used.

DXUT and GLUT have a very similar API[13]: both are frameworks, and the application implements and registers a number of callbacks with either of the libraries. The libraries take control over the application, process events in their own main loop and only pass control back to the application code through the callbacks.

[13]GLUT was created a decade before DXUT was written and the similarity of the names suggests that DXUT was inspired by GLUT.

5 Equalizer Port

5.1 The Equalizer Framework

Equalizer is an advanced parallelization framework for graphics applications that use OpenGL. It is scalable from a single desktop computer with one graphics card to a cluster with multiple connected workstations and multiple graphics cards in each. The source code of an application has to be adapted to work with it, but in return Equalizer manages distributing rendering tasks, assembling the resulting frames and balancing the load automatically. It is highly configurable, and different setups can be implemented by only changing a configuration file. No source code changes are necessary.

Documentation can be found on the library's homepage [EQh] and in its programming guide [Gmb].

The following sections are a compact introduction and present the major features of Equalizer without going too far into the details. First a few terms that Equalizer uses need to be introduced:

Server Equalizer uses a client-server approach. The server handles a cluster of render clients and distributes the work among them. It is also responsible for loading the configuration file.

Application The application connects to a server and is forwarded the configuration loaded by the server. It reacts to events and controls how the server works.

Node The different render clients are also called *nodes* in Equalizer.

Pipe GPUs are called *pipes*. A node can contain several pipes.

Window An operating system window or an off-screen rendering target is called a window.

Channel A window can be divided into several *channels* which can each render into a different part of it.

Compound A compound is a logical grouping of channels that is used to distribute rendering tasks among them.

Equalizer's class hierarchy also reflects this structure and uses the same names for these entities.

5.1.1 Rendering Modes

Equalizer supports different compound types. They determine the way the rendering load is distributed on the different nodes.

2D/Sort-First Compounds The view is split into several rectangular tiles. Each channel assigned to the compound renders one tile and the results are assembled to produce the final image.

DB/Sort-Last Compounds The scene is split into several parts, eg based on distance to the viewer. Each channel of the compound renders a different part of scene. The parts are then merged back using their depth data to determine the final image. Because this is done after rendering, overdraw and network load are higher compared to sort-first decomposition.

Stereo[scopic] Compounds The scene is rendered once for the left and the right eye. The result is then combined in anaglyphic mode, quad-buffered stereo or auto-stereo mode.

DPlex Compounds Each channel of the compound renders full frames in a round-robin scheme, ie time-multiplexed. This adds a latency as high as the number of channels used, but it is very easy to support in the application code.

Pixel Compounds Instead of separating the view into different rectangular tiles as done for 2D compounds, the view is split into parts that are interleaved to produce the final image. Each channal renders a subset of the pixels that form the full view.

5.1.2 Load Balancers

Usually the configuration file determines how the load is distributed on the different channels, but Equalizer also provides run-time load balancers[1]:

Load Equalizer The load equalizer changes the width and height of channels to balance the load of different channels.

[1]The Equalizer Programming Guide also mentions a Monitor Equalizer, but it is only used to display a miniature view of multiple channels and has nothing to do with actual load balancing

View Equalizer The view equalizer dynamically assigns channels to rendering tasks. Multiple channels from different nodes can be assigned to the compound. It dynamically distributes the work on the nodes. If a node is idle, it will help other nodes to render their view.

Framerate Equalizer The frame rate equalizer tries to smoothen the frame rate over time to prevent it from oscillating. To accomplish this, it dynamically caps the maximum frame rate to the average framerate of the channels. The latter can be calculated by monitoring the actual time spent on rendering in each channel.

DFR Equalizer DFR stands for *Dynamic Frame Resolution*. The DFR equalizer automatically lowers the resolution of channels to keep the framerate steady.

5.1.3 Configuration Files

Equalizer is configurable through an extensive configuration file format. It uses a configuration file to create its whole runtime structure: the nodes and their IP addresses are specified in it, as well as the pipes, windows, channels and compounds.

Listing 5.1 on the preceding page shows a simple configuration, which has two nodes. The first node is an appNode, which is a node that resides in the application process. If no appNode is specified, the application only handles events and runs the main loop and does not render anything on its own.

The second node is hosted on a different computer. The connection {...} block specifies how the server can connect to a node. The pipe {...} blocks tell the server that each node will access one graphics card. By default Equalizer uses the first installed GPU. This can be overridden by specifying device #index in the pipe {...} block. This is not shown in this example. Multiple windows can be declared inside a pipe {...} declaration. In Equalizer all windows of a pipe share their GL contexts to use the resources of the GPU optimally. In a window declaration viewport [**x y width height**] specifies the position and extent of the window on the screen. A window does not have to be a visible GUI window but can also be an off-screen buffer (a pbuffer or a framebuffer object).

Inside the windows the channels are declared. Each window has one channel in this example. As with the pipes above, a window can have several channels. In that case the channels would partition the window into disjunct areas by specifying viewports for the windows that do not overlap. This is also not shown in this example.

This was first half of the configuration file. The second half describes the compound and how the channels are used for rendering. To do this the wall {...} block specifies the frustum of projection device . A wall projection is a projection against a planar surface, so three points suffice to establish the projection matrix. The three points are defined using

Listing 5.1: An Equalizer configuration file

```
# stereo rendering on two nodes
server {
    config {
        appNode {
            pipe {
                window {
                    viewport [ 0.25 0.25 .5 .5 ]
                    channel  {
                        name "channel-left"
                    }
                }
            }
        }
        node {
            connection {
                type TCIP
                port 123
                hostname "192.168.0.35"
            }
            pipe {
                window  {
                    viewport [ 0.25 0.25 .5 .5 ]
                    channel {
                        name "channel-right"
                    }
                }
            }
        }
        compound {
            wall {
                bottom_left  [ -.4 -.5 -1 ]
                bottom_right [  .4 -.5 -1 ]
                top_left     [ -.4  .5 -1 ]
            }
            compound {
                channel "channel-left"
                eye [ LEFT ]
            }
            compound {
                channel "channel-right"
                eye [ RIGHT ]
            }
        }
    }
}
```

bottom_left, bottom_right and top_left. Their [x y z] values describe the position of the points in a coordinate system—usually the OpenGL coordinate system (see Figure 2.12b on page 16), but the application code is free to redefine this.

Two child compounds are declared for the channels. The child compounds automatically use the frustum of their parent. The eye [LEFT] and eye [RIGHT] lines tell Equalizer that "channel−left" should render the left eye and "channel−right" the right eye. This is sufficient to make Equalizer render stereoscopic images on two nodes.

There are many more options and the example does not even contain canvases and layouts with offer more ways to manage output devices. [Gmb] contains a useful introduction to these topics and a detailed explanation of the configuration file format. Equalizer's home page also contains a good write-up of the latter, see [Eil].

5.1.4 API Overview

Now that the reader is familiar with the concepts of Equalizer, the code architecture can be discussed. The primary question is: how do I write an application that uses Equalizer?

Like DXUT and GLUT, Equalizer is a framework. Inside the main event loop the application hands over control to the library and only reacts to callbacks. Equalizer is a bit more flexible though. Instead of implementing a few callbacks, in Equalizer the application extends classes and implements their virtual methods. The *Factory* design pattern (see [GHJV95]) is used to proffer the adapted classes to the framework by extending the class **`NodeFactory`** to create user-defined objects instead of instances of the default Equalizer classes.

Equalizer offers many classes which can be extended. Figure 5.2 on page 54 shows the most important ones. **`Config`** represents the loaded configuration file. The only instance of this class is created in the application process.

By convention the **`Config`** instance initializes shared data and registers it with the network layer. It then gives over control to Equalizer and passes the network identifier of the shared data to it. The different nodes can then query this identifier and map the data over the network.

Pretty much every object that can be defined in the configuration file has a class that can be extended by the application programmer. Possible classes thus include: **`Node`**, **`Pipe`**, **`Window`** and **`Channel`**.

`Config` has callbacks to initialize and finalize itself (**`init`**, **`exit`**), to start and finish the current frame (**`startFrame`**, **`finishFrame`**), and to handle events such as mouse or keyboard events (**`handleEvent`**).

`Node` and the other classes have similar callbacks with the addition of methods to handle clearing the screen, rendering the scene and assembling the final image

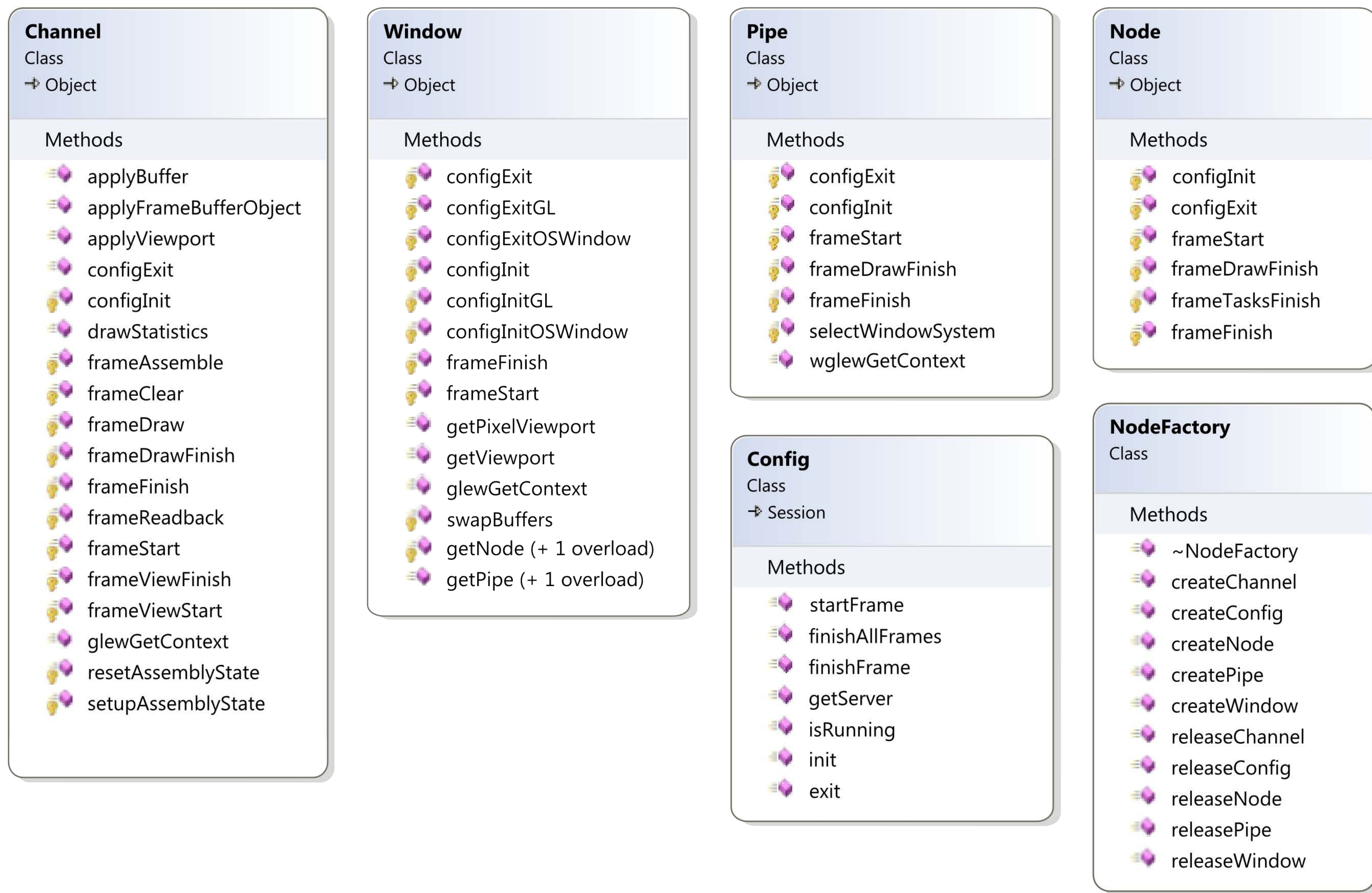


Figure 5.2: Overview of the most important Equalizer classes (with selected methods)

(**frameClear**,**frameDraw**, **frameAssemble**). With the exception of the **Config** class, callbacks have the form *nounVerb*. Thus the callbacks to initialize and finalize an object are called **configInit** and **configFinish** for all classes (except **Config**).

All callbacks provide a meaningful default implementation. This is also true for **NodeFactory**. If the default implementation of **NodeFactory** is used, Equalizer simply renders a black screen.

Listing 5.3 shows the main function of a simple Equalizer application:

Listing 5.3: Main function and main-loop of a simple Equalizer application (adapted from Equalizer's eqPly example)

```
class NodeFactory : public eq::NodeFactory {
public:
    virtual eq::Config*  createConfig( eq::ServerPtr parent )
        { return new eqTerrain3D::Config( parent ); }
    virtual eq::Node*    createNode( eq::Config* parent )
        { return new eqTerrain3D::Node( parent ); }
    virtual eq::Pipe*    createPipe( eq::Node* parent )
        { return new eqTerrain3D::Pipe( parent ); }
    virtual eq::Channel* createChannel( eq::Window* parent )
        { return new eqTerrain3D::Channel( parent ); }
    virtual eq::Window*  createWindow( eq::Pipe* parent )
        { return new eqTerrain3D::Window( parent ); }
};

int main( const int argc, char** argv ) {
    // 1. parse arguments
    eqTerrain3D::LocalInitData initData;
    initData.parseArguments( argc, argv );

    // 2. Equalizer initialization
    NodeFactory nodeFactory;
    if( !eq::init( argc, argv, &nodeFactory ) )
    {
        return EXIT_FAILURE;
    }

    // 3. initialization of local client node
    RefPtr< eq::Client > client = new eq::Client;
```

```
    if( !client->initLocal( argc, argv ) )
    {
        eq::exit();
        return EXIT_FAILURE;
    }

    // 4. run main-loop
    int ret = run( client, initData );

    // 5. cleanup and exit
    client->exitLocal();
    client = 0;

    eq::exit();

    return ret;
}

int run( RefPtr< eq::Client > &client,
   eqTerrain3D::LocalInitData &initData ) {
    // 1. connect to server
    eq::ServerPtr server = new eq::Server;
    if( !client->connectServer( server )) {
        return EXIT_FAILURE;
    }

    // 2. choose config
    eq::ConfigParams configParams;
    eqTerrain3D::Config* config =
       static_cast<eqTerrain3D::Config*>(server->chooseConfig(
       configParams ));

    if( !config ) {
        client->disconnectServer( server );
        return EXIT_FAILURE;
    }

    // 3. init config
```

```
    config->setInitData( initData );
    if( !config->init() ) {
        server->releaseConfig( config );
        client->disconnectServer( server );
        return EXIT_FAILURE;
    }

    // 4. run main loop
    while( config->isRunning() ) {
        config->startFrame();
        config->finishFrame();
    }
    config->finishAllFrames();

    // 5. exit config
    config->exit();

    // 6. cleanup and exit
    server->releaseConfig( config );
    client->disconnectServer( server );
    server = 0;

    return EXIT_SUCCESS;
}
```

NodeFactory is the *Factory* class used to connect the application code to Equalizer.

initData is the shared data mentioned above. It is initialized from command-line arguments or by reading in a configuration file (not shown in this example).

If the process is started as a client node that is controlled by the server, Equalizer takes over control during the call to **initLocal** in step 3 and does not return. In the application process, it connects to a server or creates a local one and then loads the configuration from it.

Finally the **run** function is called and the main loop is entered, in which frames are processed. The loop in this example is very minimalistic but performs all necessary tasks. **finishAllFrames** is called after the loop to finish outstanding frames because Equalizer supports a frame latency on slower nodes if desired and they might need a bit to catch up.

This suffices as short introduction to Equalizer's API. Interested readers can consult

[Gmb] for more information. Section 5.3 on the facing page also contains additional information.

5.2 Overview of the Porting Process

The goal of the Equalizer port was to take the OpenGL port and replace the GLUT application code with code that would run within the Equalizer framework. This port requires some changes to the renderer backend, too, because the rendering code has to be embedded in a channel and has to support some of Equalizer's features like using frustums from the configuration file or rendering into off-screen buffers instead of the screen. Many of Equalizer's features should be supported, like multiple channels, windows and pipes, and it should run as efficiently as possible.

The porting can be divided into three steps:

1. port the application code,
2. adapt the renderer code to support rendering to the default Equalizer configuration (one window with one channel),
3. add support for multiple channels, multiple windows and multiple pipes per node.

The following sections describes each of the three steps in detail.

5.3 Equalizer Application Code

Generally Equalizer applications have similar architectures as described in Section 5.1.4 on page 52. To simplify the porting and to have a running Equalizer version of Terrain3D quickly, the *eqPly* example from Equalizer's source was used as a basis for the port of the application code.

Using eqPly has one consequence: its source is licensed under the LGPL. Because of this, Terrain3D's actual code has to be moved into a static or dynamic library to adhere to the license, which would otherwise requires the release of the source code on demand.

5.3.1 eqPly's Equalizer Classes

Everything that does not belong to Equalizer was removed, leaving a skeleton of Equalizer code that could be filled with the application logic of Terrain3D. Figure 5.4 on the following page shows a class diagram of the Equalizer-specific classes in eqPly.

`View` is not needed, so it was removed. The only classes that haven't been described yet are **`EqPly`**, **`FrameData`**, and **`InitData`** and **`LocalInitData`**. **`EqPly`** extends

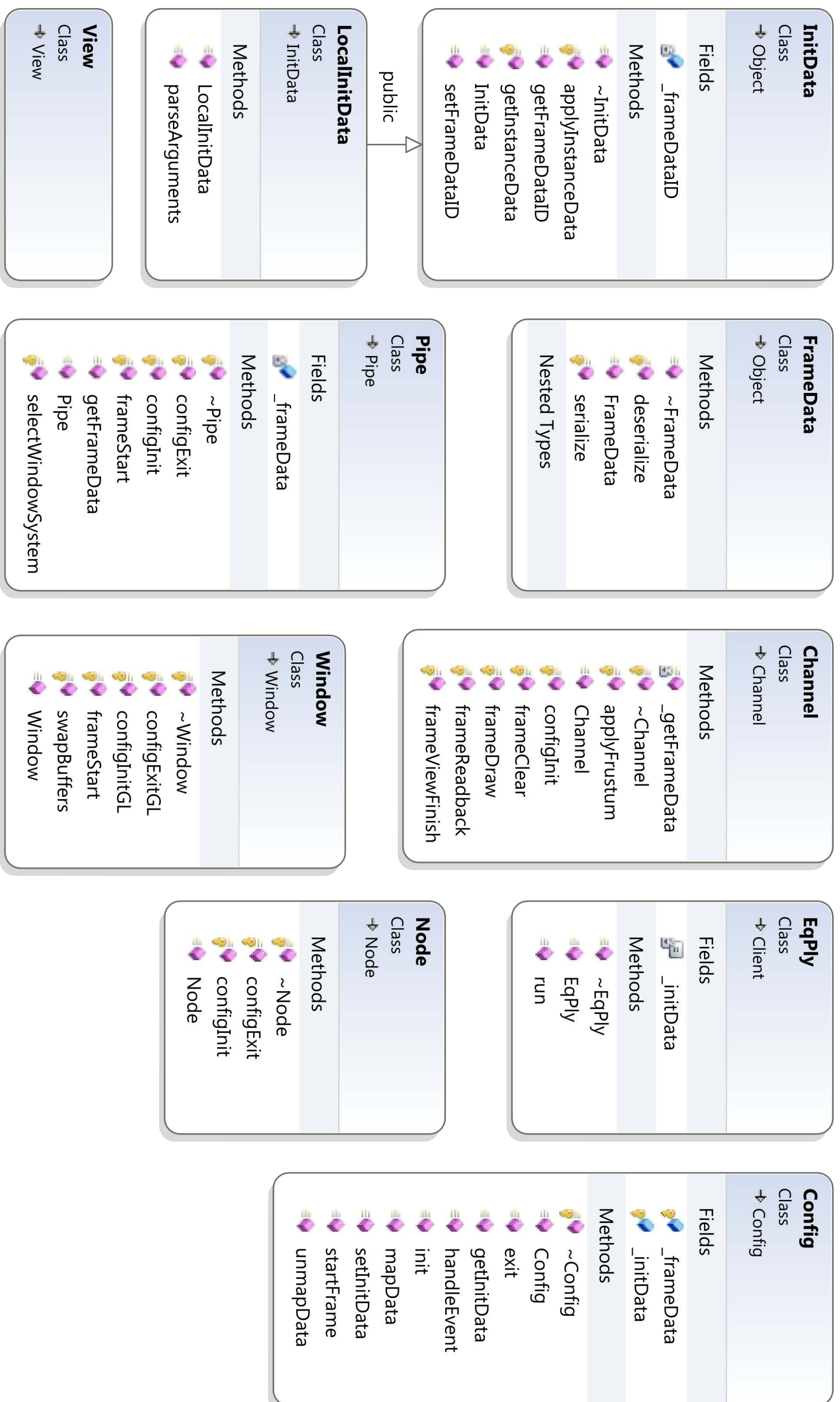


Figure 5.4: Equalizer classes in eqPly (with selected methods and members)

eq::Client[2]. It really only adds a static **run** function, which can be moved to the **main** function. The class can then be removed and instead **eq::Client** can be used directly (see Listing 5.3 on page 53).

InitData is a class that contains static data that every node needs access to. **LocalInitData** extends **InitData** and additionally contains data that is only needed in the application.

The application initializes an instance of **LocalInitData** and sends the **InitData** part to all nodes. This is done using Equalizer's network layer, which allows you to register objects with the server and get a unique identifier for them. On initialization the server sends an application-specified value to all nodes. Usually the application sends the identifier of the **LocalInitData** object. The nodes then map the object to a local copy using the identifier and thus get access to the shared static data of the application.

FrameData holds dynamic data that changes between frames. It is registered at initialization as network object and its identifier is stored in **InitData**. The nodes map it using the identifier to local copies, which are updated every frame.

To send an object to another node, it is serialized and deserialized by the network layer. For this an object has to provide **serialize** and **deserialize** methods[3], which are used by the network layer to send and receive the object's data.

5.3.2 Porting the Application Code

Terrain3D usually loads its configuration from two files: a system-specific file and a terrain-specific one. The former specifies the terrain to load, the allowed screen-space error and other general settings. The latter contains information about the terrain data like the path to the actual data, or the depth of the tile tree.

SystemConfiguration and **TerrainConfiguration** encapsulate this data. Both become members of **InitData** in the Equalizer version. **SystemConfiguration** is sent over the network using the serialization method described above, but **TerrainConfiguration** is not. Instead, after receiving the system configuration, each node loads the terrain configuration from disk itself. The idea behind this is that different nodes can use different datasets if necessary. In heterogeneous environments this can be an easy way to statically balance the load on the nodes.

Porting the code for **FrameData** is straight-forward, too. Terrain3D has quite a few settings that can change every frame like, for example, whether it should render the terrain in wireframe mode or using occlusion culling, or whether it should render frame statistics. This can be simply moved into **FrameData** and serialized over the network. Every node

[2] Equalizer's API resides inside the **eq::** namespace. Because other classes like **Node** or **Pipe** share their name with their base classes, the namespace will always be specified to distinguish between them.

[3] Again this is a simplification, see chapter 6.3 in the programming guide [Gmb] for more details.

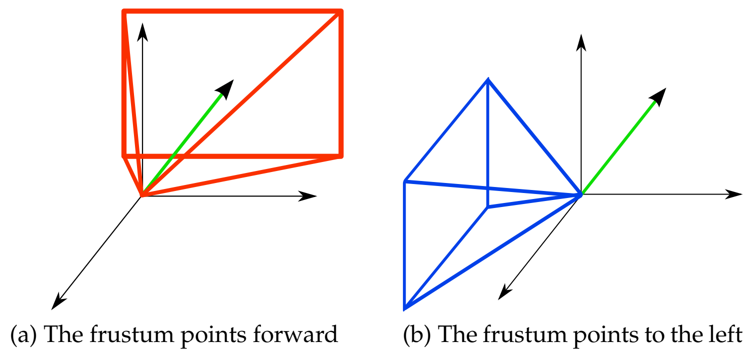
Listing 5.5: Excerpt of Config::handleEvent

```
bool Config::handleEvent( const eq::ConfigEvent* event )
{
    switch( event->data.type )
    {
        case eq::Event::KEY_PRESS:
        {
            const eq::KeyEvent& eventData = event->data.keyPress;
            _camera->KeyboardEvent( eqKeyToCamKey( eventData.key
                ), true );
            _recorder->KeyboardEvent( (unsigned char)
                eventData.key );

            _frameDataKeyEvent( (unsigned char) eventData.key );

            _redraw = true;
            break;
        }
        [...]
```

also needs to access the view matrix and position of the camera. Consequently these values are also kept in **FrameData**.

The application code has to be split into code that can be run in the application process, and code that has to run on each node. The **Camera** and the **Recorder** objects can obviously reside in the application process. For this **_camera** and **_recorder** member variables are added to **Config**. The changed view position and matrix are sent to all nodes, when **FrameData** is updated. **Config::handleEvent** handles mouse and keyboard events. See Listing 5.5 on the following page for an excerpt. **Config::startFrame** updates the camera and recorder every frame.

Now only the rendering code is left. Since only one channel and one window have to be supported at the beginning, all the rendering code can be put into **Channel**. **Channel::configInit** and **Config::configExit** initialize and release the data loader and terrain renderer and **Channel::frameDraw** updates them and renders the scene. For this it needs the current view direction, view matrix and position. However, the channel can use a different view direction than the camera because the projection it was defined with might not point forward. See Figure 5.6 on the next page for an example. **eq::Channel** provides two methods that help retrieve the relevant information: **getHeadTransform** and **getFrustum**. The head transform orients the coor-



(a) The frustum points forward (b) The frustum points to the left

Figure 5.6: Differently oriented frustums

dinate system in such way as to point in the direction of the wall. **getFrustum** simply returns the frustum that can be used to compute a normal non-oriented projection matrix. Listing 5.7 on page 61 shows how the final view matrix and direction are computed. **viewToGLTransform** is needed because Terrain3D uses a different, left-handed coordinate system internally: z points upwards, y to the right and x forward. OpenGL uses a normal right-handed coordinate system: z points backwards, y upwards and x to the right. The step from left-handedness to right-handedness is the reason why the z-axis is inverted in the code: **viewToGLTransform.scale**(1.0, 1.0, −1.0);.

headTransform is the transformation needed to orient the frustum of the channel. It is applied after **viewToGLTransform**, so the OpenGL coordinate system can be used when specifying walls and projections[4]. After the transformation the negative z axis of **glViewMatrix** points into the view direction of the channel. Consequently **eq**::**Vector4f viewDirection** = −**glViewMatrix.get_column**(2); retrieves it.

This concludes the discussion of the application code in Equalizer. The next section explains the changed in the renderer backend code.

5.4 Changes to the Renderer Backend

Only few changes are required in the renderer backend. For one Equalizer always sets the scissor size, too, when the viewport changes, so **Device::setViewport** has to be adapted.

The render backend uses *glew* (GL Extension Wrangler Library) to access OpenGL extensions and Equalizer does so as well. It uses different compiler options for it though: it links against it statically and it uses multiple contexts. The latter deserves more explanation: extensions in OGL have to be loaded manually. The function pointers for each new function that an extension introduces have to be queried at runtime.

[4]This has the benefit that all example configuration from Equalizer continue to work correctly, too

Listing 5.7: View matrix and direction computation from `Channel::frameDraw`

```
const eq::Matrix4f &viewMatrix = frameData.getViewMatrix();

// the terrain code uses a different coordinate system internally
eq::Matrix4f viewToGLTransform( eq::Matrix4f::IDENTITY );
viewToGLTransform.rotate_y( DEG2RAD( 90 ) );
viewToGLTransform.rotate_x( DEG2RAD( 90 ) );
viewToGLTransform.scale( 1.0, 1.0, -1.0 );

const eq::Matrix4f &headTransform = getHeadTransform();
eq::Matrix4f glViewMatrix = headTransform * viewToGLTransform *
    viewMatrix;

if (!frameData.getStaticMesh())
{
    eq::Vector4f position = frameData.getPosition();
    eq::Vector4f viewDirection = -glViewMatrix.get_column( 2 );

    getPipe()->updateDataLoader(position.array,
        viewDirection.array);
    terrain->Update(glViewMatrix.array, position.array,
        viewDirection.array, frameData.getOcclusionCulling());
}
```

Libraries like glew automate this. If multiple GL rendering contexts are used, the specifications say that the function pointers have to be queried for each one because they could be different depending on the context.

Usually glew ignores this and only queries all function pointers once. Equalizer uses a special compiler option to make glew support loading the function pointers for each context. Normally the function pointers are stored as global variables; in this mode, though, a container structure called **GLEWContext** is used to hold the function pointers. These two ways of storing the function pointers are incompatible and consequently the renderer backend has to be changed to support multiple contexts, too.

When glew is compiled to use multiple contexts, it expects a function or macro called **glewGetContext** to exist, that returns a pointer to the **GLEWContext** that should be used. To avoid keeping such a pointer in every object that uses extension functions, the renderer backend implements a global function **glewGetContext**, which in turn references a global thread-local-storage variable that stores the context pointer. A thread-local storage variable exists separately for every thread in the application.

Equalizer uses one thread per pipe and every window in a pipe shares its rendering context with each other, so one **GLEWContext** per pipe is sufficient. Listing 5.8 shows the code that adds support for multiple **GLEWContext**s.

Listing 5.8: Code to add support for multiple **GLEWContext**s in the Equalizer version

```
#ifdef GLEW_MX
extern __declspec( thread ) GLEWContext *__glewContext;
inline GLEWContext * glewGetContext() { return __glewContext; }
inline void glewSetContext( GLEWContext *context ) {
  __glewContext = context; }
#endif
```

When rendering into a channel, only Equalizer knows whether OpenGL should render into the backbuffer or into a framebuffer, so **Device::SetBackBufferRenderTarget** has to be changed to let Equalizer decide. Equalizer provides many helper methods in **Channel** to accommodate setting the OpenGL state in the right way. Among them are **applyFrameBufferObject**, **applyBuffer** and **applyViewport**, which can be used to set up OpenGL for rendering into a channel. Listing 5.9 on the facing page shows Equalizer version of the code.

Finally after rendering everything, **Effect::ResetState** needs to be called to reset the OpenGL state that has been modified by using effect files. Equalizer can then correctly continue and use OpenGL itself to perform the tasks of reading back image data and/or reassembling it. See Listing 5.10 for a shortened version of **Channel::frameDraw**.

Listing 5.9: **Device::SetBackBufferRenderTarget**

```
void Device::SetBackBufferRenderTarget() {
#ifndef EQUALIZER
    glBindFramebuffer( GL_FRAMEBUFFER, 0 );
    glDisable( GL_RASTERIZER_DISCARD );
    checkGLError();
#else
    channel->applyFrameBufferObject();
    channel->applyBuffer();
    channel->applyViewport();
    glDisable( GL_RASTERIZER_DISCARD );
    checkGLError();
#endif
}
```

Listing 5.10: Shortened version of **Channel::frameDraw**

```
void Channel::frameDraw( const uint32_t frameID ) {
    // [...] variable initializations

    glCullFace( GL_BACK );
    glEnable( GL_CULL_FACE );
    glFrontFace( GL_CW );

    // [...] update data loader and terrain, see Listing??

    device->SetBackBufferRenderTarget();

    if (frameData.getSatModify()) {
        terrain->SetSaturation(frameData.getSaturation());
    }

    if (frameData.getWireframe()) {
        _rasterizerStateWireframe();
    }
    else {
        _rasterizerStateSolid();
    }
```

```
    if (frameData.getPageBoundingBoxes()) {
        dataLoader->Render(terrain);
    }

    terrain->Render(frameData.getBoundingBoxes(),
        frameData.getSatModify());

    Renderer::Effect::ResetState();

    glDisable( GL_CULL_FACE );
}
```

5.5 Better Equalizer Support

To support multiple windows and pipes a few changes are necessary. All windows of a pipe share their GL contexts. The data loader and terrain renderer are always bound to one GPU. This means that all windows of a pipe can share the data loader and terrain renderer.

Moving the initialization and destruction code from **`Channel`** to **`Pipe`** is a simple refactoring step. However there is a problem: when **`Pipe::configInit`** is called, the windows of the pipe haven't been initialized yet and no GL rendering context is available. Only when **`Windows::configInitGL`** and **`Channel::configInit`** are called, a context is available. Similarly when **`Pipe::configExit`** is called, the rendering contexts have already been released. **`Channel::configExit`** and **`Pipe::configExitGL`** are called before this happens.

The problem is solved by taking advantage of this: initialization is delayed until **`configInitGL`** is called for the first window, and destruction is called when **`configExitGL`** is called for the last window. A counter is used to keep track of the windows that are using the pipe. The counter is incremented in **`Pipe::delayedInit`** and decremented in **`Pipe::earlyExit`**. When **`Pipe::delayedInit`** is called and the counter is 0, the initialization code is executed. Likewise when it reaches 0 again in **`Pipe::earlyExit`**, the destruction code is executed. See Listing 5.11 on the facing page.

Another issue is that both the data loader and the terrain update methods are called in **`Channel::frameDraw`** and if there are multiple channels using the same pipe, these update methods are also called multiple times. **`Terrain::Update`** must be called for each channel, because each channel can have a different different view direction

Listing 5.11: Delayed init and early exit code

```
bool Pipe::delayedInit() {
    if( _refCounter++ != 0 ) {
        return true;
    }

    return _initDevice() && _initTerrain();
}

void Pipe::earlyExit() {
    if( --_refCounter == 0 ) {
        _exitTerrain();
        _exitDevice();
    }
}
```

and this method selects the tiles that are rendered depending on it. But the calls to **`DataLoader::Update`** can be reduced to one per pipe instead of one per channel.

For this a **`dataLoaderUpdated`** flag is added to **`Pipe`** and reset each frame in **`Pipe::frameStart`**. Instead of calling **`DataLoader::Update`**, **`Pipe::updateDataLoader`** is called, which checks **`dataLoaderUpdated`** to make sure it only updates data loader once per frame.

When multiple channels are used, the data loader chooses the finest level of detail necessary to meet all channel's requirements for the screen-space error limit. Each channel passes the relevant parameters to the data loader once per frame. This works well as long as there is enough memory available to keep all needed pages in memory. Otherwise the data loader will start to page tiles in and out every frame if different view directions and level-of-details are used, which kills performance.

The data loader is parameterized by the view direction, the frustum and maximum screen-space error. Only the view direction determines how pages are prioritized during load. The other two parameters are needed to calculate the required level of detail. The **`DataLoader::ComputeLoadingRadiuses`** method computes the different loading radii for each level-of-detail. These loading radii specify inside which circular area tiles of a certain level-of-detail should be loaded from disk.

To support multiple channels, each channel gets its own set of loading radii, and every time after one is updated, the one with the highest values, that is, the one that will cache most tiles, is chosen as the new active set.

With these changes efficient rendering into multiple channels and windows on a single pipe is possible. Further improvements can be achieved, but this is beyond the scope of this thesis.

6 Conclusion

6.1 Performance Comparison

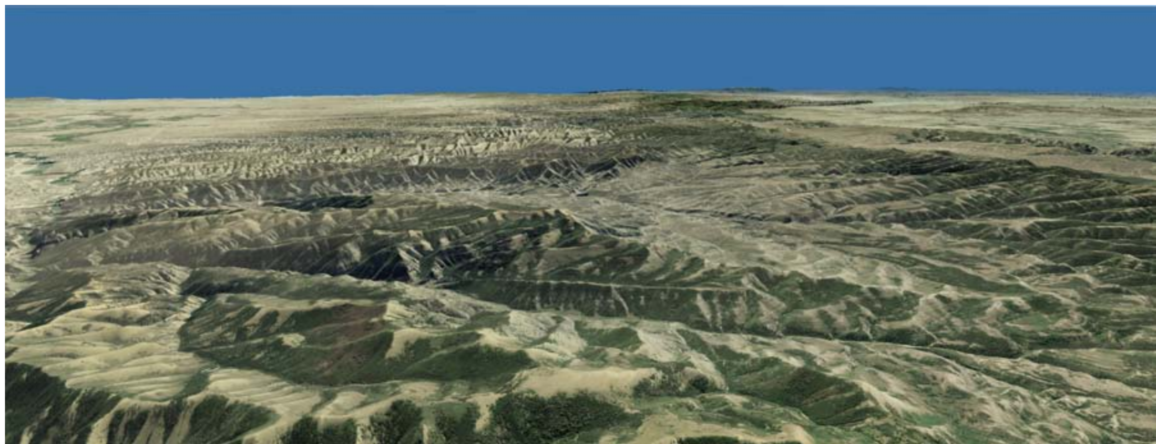


Figure 6.1: A panorama screenshot from **Benchmark B**

Table 6.2: Performance Comparison

	Benchmark A				Benchmark B	
	D3D 10					
	(Original)	(Wrapped)	OGL	Equalizer	OGL	Equalizer
Normal	—	—	51 fps	36 fps	88 fps	81 fps
Indexed	69 fps	66 fps	70 fps	59 fps	136 fps	118 fps

Table 6.2 shows how the different versions of Terrain3D perform on a map of Utah. A prerecorded flight is played back in each version and the average fps count is recorded.

Benchmark A uses a map of Utah at a resolution of 8 m for both geometry and textures and total compressed size of 2.88 GB. The system used is a Core2 Duo 2.0 GHz with 2 GB RAM, a Geforce 8600M GT and a standard 160GB IDE hard disk. The terrain was rendered at a resolution of 2560x1024.

Benchmark B uses a map of Utah at a resolution of 5 m for geometry and 1 m for textures. The total compressed size of the terrain data is 171.5 GB. The system used is a Core 2 Quad

Q9650 3.00 GHz with 8 GB RAM and two NVIDIA QuadroPlex 2200 D2. It has a 1 TB SATA hard disk. The terrain was rendered at a resolution of 1280x800.

As you can see, there is small fps drop between the original Direct3D version and the one that uses the renderer backend. The difference between Direct3D 10 and OpenGL 3.2 is also rather subtle. OpenGL's API seems to have less call overhead than Direct3D. On the other hand it is very noticeable that both the pure OpenGL and the Equalizer version benefit a lot from indexed drawing. Equalizer adds a visible overhead, too. But it is still a lot better than non-indexed drawing.

6.2 Further Reseach and Development

This thesis can serve as starting point for further research and development. For one, it only looks at parallelization in Equalizer in a very basic form.

Support for sort-last compounds could be added, and the terrain and data loader could be adapted to honor this. Another optimization could consist of rewriting the data loader and terrain code to support multiple GPUs directly. At the moment one data loader has to be created per pipe, because the data loader does not support uploading data to multiple GPUs at the same time.

Likewise the terrain renderer could be decomposed to store one page tree per pipe and keep separate renderer trees for each channel. This would improve support for differently sized channels, respectively frustums. The current design works best with channels and frustums that are same size or, to be more precise, need the same level-of-detail because the maximum needed level-of-detail is used for all tiles when loading them.

It would be interesting to research how to distribute rendering tasks best to let multiple pipes or nodes render images cooperatively in a way that allows each data loader instance to load as little unnecessary data as possible from hard-disk. In the best case this would result in storing different datasets on the different nodes, which would increase the theoretically possible dataset size by an order of magnitude.

Equalizer also supports dynamic load-balancing across multiple nodes through view compounds. It would be interesting to experiment with this and different strategies of prefetching and decompressing data to see how small the response times during load balancing can become.

Another topic for further research and development could be the GLSL effect compiler. It could be extended to support more features of OpenGL, eg bigger state blocks or automatic configuration of texture parameters for passes—similar to sampler blocks in Direct3D's effect file format. It is also worth evaluating whether a code generator is the best solution in general or if a binary format and a static library should be preferred. The code generator approach certainly provides most comfort for the programmer when it comes to

code maintenance, but on the other hand even minor changes to a shader require rebuilding the project.

The development of the GLSL effect compiler has also raised another interesting, even though not graphics-related, question. StringTemplate is a powerful library and the templates can be used in many ways, but the fact is, that it is not a real programming language, and you reach its limitations quickly. It might be interesting to examine, for example, how PHP could be used for code generation or the tasks StringTemplate is used for in general, or how regular programming languages could be extended to facilitate string output by adding additional text operators.

6.3 Results

This thesis describes the differences between Direct3D and OpenGL in detail. It explains all the steps necessary to port a complex application from Direct3D to OpenGL and points out where the common pitfalls lie.

The wrapper it presents is well tested and mature. It supports most of the common features of the two APIs and some special ones as well. Moreover, it can easily be extended to support more features and formats, and can be used to port other applications, too.

A simple yet powerful GLSL effect file format has been designed from the ground up, along with a compiler that converts it into C++ code. The effect file format is easily extensible and the compiler makes use of high-level libraries for parsing and code generation, which guarantees maintainability. Considering that there is no other simple GLSL effect file format available at the moment, this file format could well be developed further and released to the public as helper library similar to glew or GLUT.

This work also examines how to develop applications for Equalizer and presents the steps necessary from a result-oriented point of view. It uses this approach to port Terrain3D to Equalizer and explains how to embed the application logic in Equalizer's structures. The last part of the thesis talks about Equalizer-specific optimizations to increase efficiency and to support more of Equalizer's features.

Most importantly, this thesis has resulted in a version of Terrain3D that can run on multiple GPUs in parallel and on multiple clients in a cluster. It is capable to drive a CAVE system like the one the research labs at KAUST have got.

Bibliography

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[ANT] ANTLR - What is the intended behavior of the lexer? Available from: http://www.antlr.org/wiki/pages/viewpage.action?pageId=4882470 [cited October 8, 2009].

[BSK+07] Kai Bürger, Jens Schneider, Polina Kondratieva, Jens Krüger, and Rüdiger Westermann. Interactive visual exploration of instationary 3D-flows. In *Eurographics/IEEE VGTC Symposium on Visualization (EuroVis), to appear*, 2007.

[BSW+07] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1, 6th Edition*. Addison-Wesley Professional, 2007.

[DKW09] Christian Dick, Jens Krüger, and Rüdiger Westermann. GPU ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, pages 43–50, 2009.

[DSA] Ext_direct_state_access. Technical report. Available from: http://www.opengl.org/registry/specs/EXT/direct_state_access.txt [cited October 15, 2009].

[DSW09] Christian Dick, Jens Schneider, and Rüdiger Westermann. Efficient geometry compression for GPU-based decoding in realtime terrain rendering. *Computer Graphics Forum*, 28(1):67–83, 2009.

[Eil] Stefan Eilemann. Configuration file format [online]. Available from: http://www.equalizergraphics.com/documents/design/fileFormat.html [cited October 9, 2009].

[EQh] Equalizer: Parallel rendering [online]. Available from: http://www.equalizergraphics.com/.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gmb] Eyescale Software GmbH. *Equalizer Programming and User Guide*. Available from: http://www.equalizergraphics.com/downloads/DBCAAF49A0C0/ProgrammingGuide.pdf [cited October 9, 2009].

[Kra09] Martin Kraus. The pull-push algorithm revisited. In *Proceedings GRAPP 2009*, 2009.

[McC04] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[MSD] *Microsoft Developer Network Online Documentation*. Available from: http://msdn.microsoft.com.

[Para] Terence Parr. Antlr parser generator [online]. Available from: http://www.antlr.org.

[Parb] Terence Parr. Stringtemplate template engine [online]. Available from: http://www.stringtemplate.org/.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[SA09] Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification (version 3.2 (core profile) - july 24, 2009). Technical report, Khronos Group Inc., July 2009. Available from: http://www.opengl.org/registry/doc/glspec32.core.20090803.pdf.

[SBW06] Jens Schneider, Tobias Boldte, and Ruediger Westermann. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In *Vision, Modeling and Visualization 2006*, 2006.

[SW06] Jens Schneider and Rüdiger Westermann. GPU-friendly high-quality terrain rendering. *Journal of WSCG*, 14(1-3):49–56, 2006.

[WLH07] Richard Wright, Benjamin Lipchak, and Nicholas Haemel. *OpenGL® SuperBible: Comprehensive Tutorial and Reference, Fourth Edition*. Addison-Wesley Professional, 2007.