Conceptual Approach Towards Recursive Hardware Abstraction Layers

Robert Konrad, Polona Caserman, Stefan Göbel, and Ralf Steinmetz

Multimedia Communications Lab - KOM
Technische Universität Darmstadt, Darmstadt, Germany
{robert.konrad,polona.caserman,stefan.goebel,
ralf.steinmetz}@kom.tu-darmstadt.de

Abstract. Cross-platform publishing is a must have in game development. Sophisticated game engines such as Unreal or Unity provide cross-platform publishing capability. Therefore, many developers use these game engines. On the other hand, several game developers also provide their own technology and do not want to become fully dependent on external technology. Based on that situation efficient mechanisms are required to combine both sides: Usage of custom in-house technology enhanced with multi-platform capabilities. This paper introduces a new concept for hardware abstraction layers tackling this issue. Chapter 1 and 2 motivate the use of multiple hardware abstraction layers and provide a brief overview of related work. Chapter 3 describes the Kha and Kore frameworks as basic game technology for custom in-house game engines. In the main part of this paper, a conceptual approach of hardware abstraction layers, is introduced in Chapter 4 and Chapter 5 discusses its practical use for the integration in Unreal and Unity. Finally, Chapter 6 provides an overview and best practice examples of how to use Kha and Kore for Serious Games.

Keywords: Kha, Kore, Hardware Abstraction Layer, OpenGL, Unreal, Unity

1 Introduction

In recent years developing video games became considerably easier than it used to be due to the broad use of off the shelf game engines [1]. The leading game engines provide a big collection of different tools and technical components, from level editors to the low level components which make applications work on a plethora of target platforms. The market is dominated by just two engines - Unreal and Unity - and a difficult hardware situation (large diversity of Android devices, technical challenges in developing for video game consoles,...) makes it hard for new engines to catch on and dangerous to use in-house technology solutions. Like with any kind of monoculture this comes not without problems. It is a hindrance for innovations like new rendering technologies or support for unusual hardware. This situation is especially tragic for Serious Games which can benefit greatly from innovations and often have to be tailored for very specific hardware environments (old school-computers, integration with exercise hardware,...).

Custom game engines could be made viable again if it would be possible to run them inside of one of the market leading engines. Depending on the capabilities and the distribution models of a game engine (open- or closed-source) it can be possible to do exactly that - to avoid all higher level functionality of an engine and target its underlying portability layer more or less directly. This so called hardware abstraction layer (HAL) is the lowest level component in a typical game engine architecture. To abstract the underlying hardware a HAL provides an interface which is functionally equivalent to the common set of features of the targeted hardware devices. As the functionality is dictated by current hardware specifications, different HALs are very similar in the feature sets they provide. Therefore it is feasible to modify a HAL of one game engine to target the HAL of a completely different game engine. If sufficient underlying functionality of a target game engine is accessible, one game engine can consequently be made to run on top of another game engine, inheriting additional cross-platform functionality in the process.

2 Related Work

Targeting a game engine is conceptually similar to the implementation of a multi-platform game engine. Instead of targeting system APIs directly, the HAL of a game engine (if it can be accessed) is targeted and HALs tend to be similar to the APIs they abstract. Functionally complete hardware abstraction layers are relatively rare however. Most openly accessible game engines and game libraries do not implement an abstraction for the graphics APIs and instead solely rely on OpenGL for multi-platform graphics support. Notable exceptions are Oryol[1] and Unreal Engine 4[2]. A second type of applications which implement similar functionality are web browsers. When running on Windows operating systems web browsers do not use OpenGL to provide the closely related WebGL API but instead rely on Microsoft's competing Direct3D APIs for better compatibility. Google's Chrome and Mozilla's Firefox in particular use the ANGLE library[3] for this purpose.

Graphics APIs are by far the most complex system APIs used by game engines and as such multi-platform graphics APIs like OpenGL [2] and Vulkan [3] are relevant as are libraries which purely aim to abstract graphics APIs like bgfx[4] and gfx-rs[5].

For running code written for one game engine on top of another game engine it can be necessary to cross-compile source code from one programming language

[1] https://github.com/floooh/oryol
[2] https://www.unrealengine.com
[3] http://angleproject.org
[4] https://github.com/bkaradzic/bgfx
[5] http://gfx-rs.github.io

to another. The Haxe[6] compiler can compile the Haxe programming language to multiple different target languages, among them are most programming languages which are popular in the games industry like C++, JavaScript, C# and Lua. The emscripten[7] compiler in combination with the LLVM[8] compiler suite translates C and C++ code into an especially efficient subset of JavaScript. SPIRV-Cross[9] compiles SPIR-V bytecode into different GPU programming languages like GLSL, HLSL and the Metal Shading Language.

3 Kha and Kore

Kha[10] and Kore[11] are frameworks for cross-platform multimedia application development. They are especially well suited for games as they tend to be the most complex type of multimedia applications but instead of providing a complete game engine they concentrate purely on providing very complete hardware abstraction layers, including a cross-platform build system, asset management and shader cross-compilation.

Kha and Kore are functionally very similar, the primary difference being that the former is implemented in the aforementioned Haxe programming language, which can be cross-compiled to other programming languages, therefore boosting its cross-platform capabilities and the latter being implemented in C++, providing lower level access and a potential for higher performance.

Compared to conventional game engine packages the usage of Kha and Kore requires a much deeper understanding of the technological foundations of video games. This is an explicit design goal for both of these frameworks. Kha was originally created in an educational context and Kore is currently used to teach a Game Technology course at the Technische Universität Darmstadt.

4 Conceptual Hardware Abstraction Requirements

In the following a minimal viable feature set for a hardware abstraction layer is defined based on theoretical observations as well as practical experience based on the implementation of the Kha and Kore frameworks.

On the most basic level a computer consists of different input and output devices as well as internal units for computation. The computation devices in most modern computers are CPUs and GPUs (often residing on the same chip). Output is restricted to visuals and audio on most systems and common input devices are keyboards, mice, gamepads, touch surfaces and accelerometer data. Additionally computers include networking hardware and storage devices.

[6] http://haxe.org
[7] https://github.com/kripken/emscripten
[8] http://llvm.org
[9] https://github.com/KhronosGroup/SPIRV-Cross
[10] https://github.com/Kode/Kha
[11] https://github.com/Kode/Kore

4.1 Computation on CPUs and GPUs

In theory access to a turing-complete programming language is sufficient to provide all necessary functionality [4] but especially in the context of video games execution speed has to be considered. Game engines are often split in part in an engine-implementation language (typically C++) and a game-logic language (often Lua). When only the latter is accessible to a developer, performance might be unacceptable for running an additional game engine inside of it.

GPUs are programmed using so-called Shaders which are programs which execute in parallel on the GPU's many execution units. Shader programming is generally not turing-complete, disallowing recursive function calls, but provides access to many special graphics hardware features. Proving the equivalence of different shading languages is therefore difficult but the actual current situation is less complex as game engines tend to use just one of two established shader programming languages: GLSL or HLSL. Cross-compilation from GLSL to HLSL and from HLSL to GLSL is widely used in practice, for example in Unreal Engine and Unity [5] as well as in any modern web browser running on the Windows operating system. Therefore support for either GLSL or HLSL shaders by a game engine makes it a viable target regarding computations on the GPU.

4.2 Graphics Output

Visual output is internally represented by a dedicated memory area for which the content is replicated on a monitor - the so called framebuffer [6][7]. On modern systems however the framebuffer can not be directly accessed by an application. Instead GPU APIs are used to write to the framebuffer indirectly using shader programs in combination with several blocks of configurable graphics functionality.

Although GPU feature sets are constantly advancing a reasonable minimum configuration can be defined based on OpenGL ES 2 which is designed to run on the majority of today's hardware and as of now nearly 40% of all Android devices still support only OpenGL ES 2 [8].

Apart from the shading language GLSL the OpenGL ES 2.0 specification [9] contains the following functional blocks:

Draw calls Fundamentally GPUs rasterize triangles and the OpenGL API resolves around this. So called draw calls are the actual commands which initialize geometry drawing processes.
OpenGL ES 2's draw calls support rendering of points, lines and triangles based on vertex buffers and optional index buffers, the former defining the points of the geometry and the latter defining which points make up each geometrical primitive. Points and lines however are not mathematical lines of zero size or thickness - actual mathematical points and lines are invisible and therefore not useful for displaying graphics. A line with a thickness is in mathematical terms a rectangle as is a point of a certain size and rectangles can be triangulated trivially and therefore it is sufficient to only support triangles.

When no index buffer is provided, OpenGL works as if it uses an implicit index buffer containing the numbers from 0 up to the size of the index buffer minus one. This index buffer could also be provided explicitly thus eliminating the need to make index buffers optional. Considering these simplifications only a single kind of draw call is necessary, supporting indexed vertices to draw triangles.

Screen clearing OpenGL provides an explicit screen clearing call which can be simulated by drawing two triangles. The clear call can be optimized in hardware and therefore be faster but it is not strictly necessary.

Backface culling The front and back-face of a triangle are defined by its winding order [10]. Culling of back facing triangles is an optimization technique but is also important for rendering semi-transparent objects - without backface culling the innards of a semi-transparent object would be visible which is generally not intended. Backface culling is therefore a necessary feature.

Textures Texture mapping is the process of mapping image data to geometry [11] and it is fundamental to modern realtime 3D graphics. OpenGL ES 2 supports texturing including mip mapping and cube maps.

Mip maps improve scaling quality. Optimal scaling quality requires reading all pixels of an image to produce what can be a single pixel on screen which is highly inefficient and not supported by GPUs. Mip maps are arrays of pre-scaled images and depending on the necessary scaling the most fitting mip layer is used to read actual pixel data [12].

Cube-maps are arrays of six images which represent the inner six sides of a cube [13]. Cube-maps are used to pre-calculate lighting environments. As cube maps consist of six regular textures sampling a cube map can be simulated by implementing the texture coord calculations in a shader but care has to be taken when sampling at the edges of a single texture. Emulating cube-maps degrades performance and, depending on the implementation, image quality.

Texturing support itself is essential while mip maps and cube maps are not strictly necessary.

Frame- and Renderbuffers More complex rendering techniques and any kind of post-processing require access to data from previously executed render passes. OpenGL supports this by provide functionality for rendering into a texture instead of the framebuffer. These textures can then be used as normal. This functionality can not be achieved otherwise and is therefore necessary.

Write Masks Depth and stencil channels as well as individual color channels can be masked. This is useful for some rendering tricks like writing special data to the alpha channel of an image. This functionality can not be trivially replicated by a shader because it has no read access to its own render target and can not write to only a component of a color by itself. Multiple render targets can be used alternatively at great performance costs but as write masks are not widely used this compromise would be acceptable for most applications.

Stencil operations The stencil buffer is a special drawing buffer which can be used in combination with predefined comparison functions. Stencil buffers are best known for the stencil shadows algorithm [14]. Stencil operations are useful in special cases but not used much if at all in modern engines. Stencil shadows in particular have been replaced by shadow mapping and therefore do not seem to be absolutely necessary.

Blending On most hardware shader programs do not have read access to the current render target to increase parallelization efficiency. Blending colors is therefore not programmable and instead achieved using predefined blending modes.

Blending modes have historically become more and more complex but most commonly colors are either mixed directly based on the alpha values ($newcolor * alpha + oldcolor * (1 - alpha)$) or use additive blending which is typically used in combination with premultiplied alpha images [15][16].

Blending could be emulated using render targets but performance would likely be unacceptable, making support for at least the most basic blending modes necessary.

Scissoring Scissor support can be used to mask rectangular screen regions which is especially important for components of graphical user interfaces e.g. for scroll-views. Scissoring can trivially be emulated using render-targets but with severe performance implications. Alternatively triangles can be clipped beforehand, resulting in potentially large CPU overhead. Due to performance considerations and the omnipresent usage of graphical user interfaces scissoring is highly important.

Viewports Viewports define rectangular regions to which a scene is mapped. This is important for rendering different views of a scene at the same time. This feature can be emulated using render targets with reasonable performance.

Reading back pixels OpenGL allows reading pixel data from the framebuffer but this is a very slow operation because the GPU has to finish drawing and then transfer data back to CPU memory and is therefore avoided in modern game engines. Apart from features like screenshots or image analysis on the CPU reading back pixels is not necessary.

Monitor Synchronization For fluid animations applications have to be synced with the monitor refresh rate. Many game engines provide callback mechanisms which are triggered when a new frame can be rendered. This is not strictly an OpenGL feature as it is commonly provided by the operating system in a platform-specific way - nonetheless it is a necessary feature.

Graphics Intricacies Apart from the listed feature sets graphics APIs include several definitions about how they work and how data is structured which eventually have to be adapted: Images data can start at the top or bottom of an image. Matrices can be row or column major. Clip space - the final rendering coordinate system - can be defined differently. All of these situations can be handled in the shader cross-compilation step when the definitions are clear.

4.3 Remaining Hardware

Audio Current operating systems represent audio output by a small ring buffer which is written to by software and read by the audio hardware. As with the visual output modern systems do not allow direct access to the global audio ring buffer but the same concept is in use - applications are provided with their own audio buffers which are then mixed into the global buffer by the underlying system software. A hardware abstraction layer can use this concept of an audio ring buffer directly to provide all audio features easily and every more advanced audio can be built on top.

Input Common input devices are conceptually very simple, only consisting of buttons and two-dimensional movements. Touch input can be represented as an array of 2D positions. All of this can typically be mapped trivially between different game engines.

Networking UDP is the most basic networking protocol supported in the world's networking infrastructure, adding only a target port to an IP package. All other networking functionality can be built on top and UDP support is therefore sufficient.

Storage Devices Some method to read and write data, preferably based on files and directories, is of course required.

5 Feasibility analysis for targeting game engines using a hardware abstraction layer

5.1 Unreal Engine 4

Unreal Engine 4 is distributed including the full C++ source code and it is possible to directly access Unreal's own hardware abstraction layer inside of a regular Unreal project. Consequently apart from some small complications all required functionality can be accessed:

Computation on CPUs and GPUs Unreal projects are implemented in C++ making it an ideal target for other game engines in this regard.
Shaders for Unreal are written in HLSL but Unreal has no direct support for per project shader files and all new shader files have to be copied into Unreal's own directory tree, which is usually global per system. This is an unfortunate situation but can be handled satisfactorily using name mangling. Shaders also can only be loaded during Unreal's PostConfigInit phase which itself only works inside of a plugin. Unreal plugins can be components of regular Unreal projects but the project structure becomes more complicated.

Graphics Output Unreal's RHI package represents the graphics hardware abstraction layer. It is used similarly to a graphics API like OpenGL and actually provides more advanced features than Open GL ES 2 with an API that more closely resembles newer APIs like Metal using concepts like pipeline states and command buffers but retaining a relatively simple interface for setting shader parameters unlike Vulkan and Direct3D 12. This is to be expected

as the current iteration of Unreal Engine (version 4.x) is relatively new, being first released to the public in 2014 and requires relatively recent graphics hardware. Monitor synchronization can be achieved by overriding the Tick method of the AActor class. To then run the code on the render thread one of the
ENQUEUE_UNIQUE_RENDER_COMMAND macros can be used.

Remaining Hardware Unreal's USoundWaveProcedural API provides access to an audio ring buffer which is all that is required for full audio support. The UInputComponent class can be used for all input functionality, which is provided in the SetupPlayerInputComponent method of the APawn class which is a subclass of AActor. Unreal provides network access via a UDP socket API using the FSocket, FUdpSocketBuilder and FUdpSocketReceiver classes. For further functionality Unreal contains a source package which is called HAL and does indeed contain the hardware abstraction layer - excluding however any graphics or audio functionality but including file access.

5.2 Unity

Unity is fundamentally a closed source platform and source licenses are expensive (actual prices are not disclosed publicly) making it a potentially problematic target.

Computation on CPUs and GPUs Unity applications are developed on top of a .NET runtime, the actual C++ engine is inaccessible. The .NET code is either executed using the Mono runtime or a custom .NET to C++ cross-compilation based solution. Performance is not as fast as running compiled C++ code but fast enough to run at least lower end game engines on top of it.
Shaders for Unity are written in HLSL but fragment and vertex shaders reside in a single file next to some additional data and use special HLSL input semantics. A shader cross-compiler requires small adjustments to target HLSL for Unity.

Graphics Output Unity supports draw calls via the Graphics.DrawMeshNow method which draws triangles based on vertices and indices but does not support subranges of indices which is a surprising omission which can result in performance problems.
All other required functionality is supported but some of it like blending and culling is not controlled by a programming API but instead defined in the shader files.
Whether image data starts at the top or bottom of an image is not abstracted and applications have to read the UNITY_UV_STARTS_AT_TOP define and handle the situation themselves.
Monitor synchronization is achieved using the OnPostRender callback.

Remaining Hardware Generating audio programatically is supported using OnAudioFilterRead. The input API is called Input and provides methods to access keyboards, mice, gamepads, touch and accelerometer state. Being based on .NET Unity supports the System.Net.Sockets API which includes support for UDP sockets. Files have to be added to the Unity project structure. Directly supported file types can be loaded using Unity's Resources.Load calls. For binary access filenames have to end with a ".bytes" extension. Bytes files can then be cast to the TextAsset class which provides a bytes member.

5.3 Prototypical implementations in Kha and Kore

Two prototypical implementations have been implemented to verify the presented concepts.

Kore provides a prototypical implementation of an Unreal backend which is not yet fully automated but all basic functionality could be verified to work. C++ code can be directly linked into an Unreal project and direct access to the HAL - which is provided trivially for all regular Unreal projects - provides all necessary features in a direct way. Kore applications can optionally run inside of any Unreal texture which can be used freely within a 3D scene. Not being able to add shader files directly to a project sadly complicates project export and in combination with Unreal's shader caching and loading behavior proved to be the most challenging aspect of targeting Unreal.

Kha provides a fully working Unity backend. Kha's Haxe code is cross-compiled to C# with no special adjustments. Kha's GLSL based shaders are cross-compiled to Unity's special form of HLSL shaders but the differing concepts of how shaders are used in Kha and Unity proved problematic - Kha uses separate vertex and fragment shader files and sets all rendering state via APIs while Unity combines vertex and fragment shaders together with some rendering state in single files. To fully support Khas feature set all possible shader and render-state combinations have to be create beforehand resulting in a large number of shader files and currently the functionality is compromised to reduce that number. API mapping showed no further problems apart from the slightly restricted draw call API.

6 Development of Serious Games with Kha and Kore

A focus on understandable low level technology sports a number of additional advantages beyond educational aspects. Common game engines suffer from technological drawbacks because they tend to be optimized for the mass market hardware situation only although it can make a lot of sense for Serious Games to target hardware which does not fulfill these criteria. Unity recently stopped supporting Direct3D 9 [17] which is required for proper Windows XP support. Nonetheless many schools around the world continue to use very old hardware. The Raspberry Pi is a very popular platform for building custom hardware solutions (for example an ergometer containing integrated exercise games) but is neither supported by Unreal nor Unity. Arguably the most successful serious game - Dr. Kawashima's Brain Training - was only ever released on Nintendo

DS. The most recent sequel - Dr. Kawashima's Devilish Brain Training - was released exclusively for Nintendo 3DS which is not supported by Unreal or Unity. Even a popular software platform proves problematic: Web browser support is not ideal due to the typical RAM requirements of large, cross-compiled C++ game engines (a non-standardized http extension is supported by Firefox for that reason [18]).

Kha and Kore - in contrast to Unreal and Unity - do not dictate a specific workflow and embedding into other applications is easy - Armory3D and StoryTec are two examples which demonstrate this approach.

6.1 Armory3D

Armory3D embeds a Kha-based realtime 3D graphics engine in Blender. With Blender being a full 3D modeling suite such a combination offers features beyond what Unreal and Unity can provide and demonstrates the feasibility of implementing high end realtime rendering pipelines (see figure 1) in Kha.

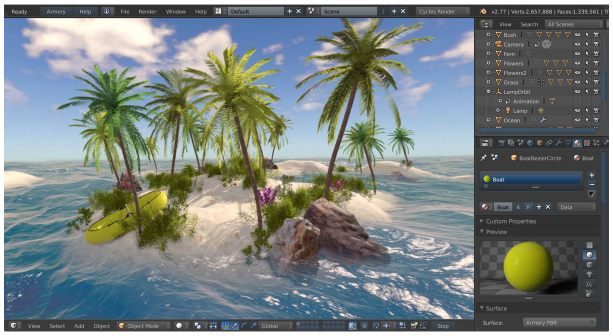


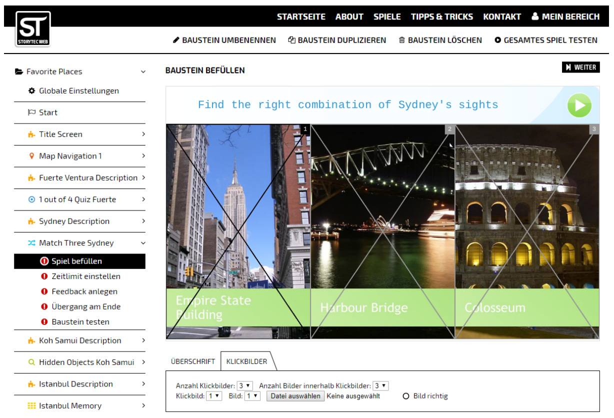
Fig. 1. Armory3D

It is aimed specifically at artists, creating a more efficient workflow for 3D asset creation by utilizing technology and algorithms originating from the game development community. It also provides options to add interactivity using a graphical programming system, empowering artists to target new application domains.

6.2 StoryTec

StoryTec is an authoring environment for Serious Games with a focus on narrative design. It provides an integrated environment including editors for the overall structure of a game, the creations of game scenes and for graphical interaction programming. It is aimed at users with very little or no programming experience, making it possible for a teacher to implement game-based e-learning courses or for a curator in a museum to create a virtual tour guide.

StoryTec exists in two different versions. One is a complex application developed for offline usage in C# and WPF. The other is a simplified online version running entirely in HTML5. Both versions of StoryTec make use of Kha. StoryTec uses it for broad multiplatform export support and to create a runtime component which can be used independently (on all target platforms) as well as integrated in a scientific analysis application. The integration in the web version of StoryTec goes even deeper. The editor itself uses an integrated Kha runtime, executing the same code which is also used to run the stories (see figure 2).



Find the right combination of Sydney's sights

Anzahl Klickbilder: 3 ▾ Anzahl Bilder innerhalb Klickbilder: 3 ▾
Klickbild: 1 ▾ Bild: 1 ▾ Keine ausgewählt ○ Bild richtig

Fig. 2. StoryTec

Many serious games have already been developed and released using StoryTec, some examples are NeuroCare[12], Der Chaos-Flush[13] and the IUNO-Serious Game[14].

7 Conclusion

This paper provides a conceptual approach for Hardware Abstraction Layers as basic game technology to support cross-platform publishing of games and serious games using in-house game technology combined with sophisticated game engines. Running two game engines on top of one another is a development strategy which works in theory and practice but the benefit of better hardware compatibility has to be weighted against the disadvantages of additional complexity. Nonetheless a game engine HAL backend is a useful fall-back plan for compatibility challenges like the very diverse market of Android devices or for

[12] https://neurocare-aal.de
[13] http://darmstadt-marketing.de/fileadmin/spiel/
[14] https://www.iuno-projekt.de/veranstaltungen/termine/eventdetail/44/-/hessentag.html

hardware which is hard to access for smaller development teams - for example Nintendo for a time only supported Unity based development for small teams on its Wii U games console. The initial implementation of a HAL backend for a game engine can take a lot of effort because this use-case is typically not considered in the accompanying documentation, which proved to be true for Unreal and Unity, but the current implementations in Kha and Kore are useful starting points which can speed up this process immensely.

References

1. Statista (2014). https://www.statista.com/statistics/321059/game-engines-used-by-video-game-developers-uk/. Accessed April 2017
2. Frazier, C., Leech, J., Brown, P.: The OpenGL Graphics System: A Specification (2016)
3. The Khronos Vulkan Working Group: Vulkan 1.0.48 - A Specification (2017)
4. Turing, A. M.: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London mathematical society, 2(1), 230-265. (1937)
5. Pranckevicius, A.: Cross Platform Shaders in 2014 (2014). http://aras-p.info/blog/2014/03/28/cross-platform-shaders-in-2014/. Accessed April 2017
6. Newman, W. M., Sproull, R. F.: Principles of interactive computer graphics. McGraw-Hill, Inc.. (1979)
7. Foley, J. D., Van Dam, A.: Fundamentals of interactive computer graphics (Vol. 2). Reading, MA: Addison-Wesley. (1982)
8. Android (2017). https://developer.android.com/about/dashboards/index.html. Accessed April 2017
9. Munshi, A., Leech, J.: OpenGL ES Common Profile Specification Version 2.0.25 (2010)
10. Hultquist, J.: Backface culling. In Graphics gems (pp. 346-347). Academic Press Professional, Inc.. (1990)
11. Heckbert, P. S.: Survey of texture mapping. IEEE computer graphics and applications, 6(11), 56-67. (1986)
12. Williams, L.: Pyramidal parametrics. In Acm siggraph computer graphics (Vol. 17, No. 3, pp. 1-11). ACM. (1983)
13. Voorhies, D., Foran, J.: Reflection vector shading hardware. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques (pp. 163-166). ACM. (1994)
14. Bilodeau, W., Songy, M.: U.S. Patent No. 6,384,822. Washington, DC: U.S. Patent and Trademark Office. (2002)
15. Porter, T., Duff, T.: Compositing digital images. In ACM Siggraph Computer Graphics (Vol. 18, No. 3, pp. 253-259). ACM. (1984)
16. Smith, A. R.: Image compositing fundamentals. Microsoft Corporation. (1995)
17. Gram, M.: Deprecating DirectX 9 (2017). https://blogs.unity3d.com/2017/07/10/deprecating-directx-9/. Accessed August 2017
18. Mozilla Developer Network - Large-Allocation (2017). https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Large-Allocation. Accessed April 2017