Beholder Framework

A Unified Real-Time Graphics API

Daniil Rodin
Institute of Mathematics and Computer Science
Ural Federal University
Yekaterinburg, Russia

***Abstract*—This paper describes Beholder Framework, which is a set of libraries designed to provide a single low-level API for modern real-time graphics that combines clarity of Direct3D 11 and portability of OpenGL. The first part of the paper describes the architecture of the framework and its reasoning from the point of view of developing a cross-platform graphics application. The second part describes how the framework overcomes some most notable pitfalls of supporting both Direct3D and OpenGL that are caused by differences in design and object models of the two APIs.**

Keywords—real-time graphics; API; cross-platform; shaders; Direct3D; OpenGL;

I. Introduction

Real-time graphics performance is achieved by utilizing hardware capabilities of a GPU, and to access those capabilities there exist two “competing” API families, namely Direct3D and OpenGL. While OpenGL is the only option available outside Windows platform, it has some significant drawbacks when compared to Direct3D including overcomplicated API [1] and worse driver performance [2]. For this reason, developers, who are working on a cross-platform graphics application that must also be competitive on Windows, have to support both Direct3D and OpenGL, which is a tedious and time-consuming work with many pitfalls that arise from the design differences of Direct3D and OpenGL.

This paper describes a framework that solves those issues by providing an API that is similar to Direct3D 11, while being able to use both Direct3D and OpenGL as back-ends. The former allows developers to use well-known and well-designed programming interfaces without the need to learn completely new ones. The later allows applications developed using the framework to be portable across many platforms, while maintaining the Direct3D level of driver support on Windows.

II. Why OpenGL is Not Sufficient

Since OpenGL provides a real-time graphics API that is capable of running on different platforms, including Windows, it might look like an obvious API of choice for cross-platform development. But if you look at the list of best-selling games [3] of the last ten years (2003 – 2012), you will notice that almost every game that has a PC version [3] uses Direct3D as main graphics API for Windows and most of them are Direct3D (and thus, Windows) – only.

If OpenGL was at least near to being as powerful, stable, and easy to use as Direct3D, it would be irrational for developers to use Direct3D at all. Especially so for products that are developed for multiple platforms, and thus, already have OpenGL implementations.

These two facts bring us to a conclusion that, in comparison to Direct3D, OpenGL has some significant drawbacks.

In summary, those drawbacks can be divided into three groups.

The first reason of Direct3D dominance is that from the Direct3D version 7 and up, OpenGL was behind in terms of major features. For example, GPU-memory vertex buffers, that are critical for hardware T&L (transform and lighting), appeared in OpenGL after almost four years of being a part of Direct3D, and it took the same amount of time to introduce programmable shaders as a part of the standard after their appearance in Direct3D 8. [4]

And even today, when the difference between Direct3D 11 and OpenGL 4.3 features is not that noticeable, some widely used platforms and hardware do not support many important of them for OpenGL. For example, OS X still only supports OpenGL up to version 3.2. Another example is Intel graphics hardware that is also limited to OpenGL 3.x, and even OpenGL 3.x implementation has some major unfixed bugs. For instance, Intel HD3000 with current drivers does not correctly support updating a uniform buffer more than once a frame, which is important for efficient use of uniform buffers (a core OpenGL feature since version 3.1).

The third OpenGL drawback is very subjective, but still important. While trying to achieve backwards-compatibility, Khronos Group (an organization behind OpenGL) was developing OpenGL API by reusing old functions when possible, at the cost of intelligibility (e.g. `glBindBuffer`, `glTexImage3D`). This resulted in an overcomplicated API that does not correspond well to even the terms that the documentation is written in and still suffers from things like bind-to-edit principle [1]. On the other hand, Direct3D is being redesigned every major release to exactly match its capabilities, which makes it significantly easier to use.

III. Alternative Solutions

Beholder Framework is not the first solution for the problem of combining Direct3D and OpenGL. In this section we will discuss some notable existing tools that provide such abstraction, and what are their differences from the Beholder Framework.

A. OGRE

OGRE (Open Graphics Rendering Engine) [5] is a set of C++ libraries that allow real-time rendering by describing a visual scene graph that consists of camera, lights, and entities with materials to draw, which are much higher-level terms than what APIs like Direct3D and OpenGL provide.

While this was the most natural way of rendering in the times of fixed-function pipeline, and thus, providing this functionality in the engine was only a plus, nowadays rendering systems that are not based on scene graph are becoming more widespread because the approach to performance optimizations has changed much since then [6, 7]. The other aspect of OGRE API being higher-level than Direct3D and OpenGL is that it takes noticeably longer to integrate new GPU features into it, since they must be well integrated into a higher-level object model that was not designed with those capabilities in mind.

Therefore, even though OGRE is providing rendering support for both Direct3D and OpenGL, it is not suited for those applications that require different object model or newer GPU features.

B. Unity

Unity [8] is a cross-platform IDE and a game engine that allows very fast creation of simple games that expose some predefined advanced graphics techniques like lightmaps, complex particle systems, and dynamic shadows.

Unity provides excellent tools for what it is designed for, but has even less flexibility than OGRE in allowing implementation of non-predefined techniques. It also forces an IDE on the developer, which, while being superb for small projects, is in many cases unacceptable for larger ones.

C. Unigine

Unigine [8] is a commercial game engine that supports many platforms, including Windows, OS X, Linux, PlayStation 3, and others while using many advanced technologies and utilizing low-level API to its limits. Having said that, Unigine is still an engine which forces the developer to utilize the graphics the specific way instead of providing a freedom like low-level APIs do.

In comparison to all the discussed solutions, Beholder Framework aims to provide the freedom of a low-level API (namely, Direct3D 11) while maintaining portability of supporting both Direct3D and OpenGL.

IV. Beholder Framework Architecture

Beholder Framework is designed as a set of interfaces that resemble Direct3D 11 API, and an extensible list of implementations of those interfaces. The framework is being developed as a set of .NET libraries using the C# language, but it is designed in such a way that porting it to C/C++ will not pose any significant problems if there will be a demand for that.

All the interfaces and helper classes are stored in the Beholder.dll .NET assembly including the main interface – `Beholder.IEye` that is used to access all the implementation-specific framework capabilities. In order to get an implementation of this interface, one can, for example, load it dynamically from another assembly. This is a preferred way since it allows using any framework implementation without recompiling an application. At the time of writing, there are three implementations of the framework – for Direct3D 9, Direct3D 11, and OpenGL 3.x/4.x.

When the instance of the `Beholder.IEye` is acquired, one can use if to perform four kinds of tasks that are important for initializing a graphics application. The first one is enumerating graphics adapters available for the given system along with their capabilities. By doing this, one can decide what pixel formats to use, what display modes to ask the user to choose from, and whether some specific features are available or not. The second task is creating windows or preparing existing ones for drawing and capturing user input. The third one is initializing a graphics device, which is a main graphics object that holds all the graphics resources and contexts (corresponds to the `ID3D11Device` interface of Direct3D 11). Finally, the fourth task that `Beholder.IEye` can be used for is initializing a "game loop" – a specific kind of an infinite loop that allows the application to interact with the OS normally.

Another useful feature that the framework provides at the `Beholder.IEye` level is a validation layer. It is an implementation of the interfaces that works like a proxy to a real implementation while running a heavy validation on the interface usage. This is useful for debugging purposes, and since it is optional, it will not affect performance of a release build.

When the device is initialized and the game loop is running, an application can use Beholder Framework in almost the same way it could use Direct3D with only minor differences. The only exception to this is a shader language.

V. Unifying Shaders

Even though both Direct3D 11 and OpenGL 4.3 have similar graphics pipelines, and thus, same types of shaders, they provide different languages to write them, namely HLSL and GLSL respectively. Compare, for example, these versions of a simple vertex shader in two languages:

A. HLSL

```
cbuffer Transform : register(b0)
{
    float4x4 World;
    float4x4 WorldInverseTranspose;
};
```

```
cbuffer CameraVertex : register(b1)
{
    float4x4 ViewProjection;
};

struct VS_Input
{
    float3 Position : Position;
    float3 Normal : Normal;
    float2 TexCoord : TexCoord;
};

struct VS_Output
{
    float4 Position : SV_Position;
    float3 WorldPosition : WorldPosition;
    float3 WorldNormal : WorldNormal;
    float2 TexCoord : TexCoord;
};

VS_Output main(VS_Input input)
{
    VS_Output output;
    float4 worldPosition4 = mul(float4(input.Position, 1.0), World);
    output.Position = mul(worldPosition4, ViewProjection);
    output.WorldPosition = worldPosition4.xyz;
    output.WorldNormal = normalize(
        mul(float4(input.Normal, 0.0), WorldInverseTranspose).xyz);
    output.TexCoord = input.TexCoord;
    return bs_output;
}
```

B. *GLSL*

```
#version 150

layout(binding = 0, std140) uniform Transform
{
    mat4x4 World;
    mat4x4 WorldInverseTranspose;
};

layout(binding = 1, std140) uniform CameraVertex
{
    mat4x4 ViewProjection;
};

in vec3 inPosition;
in vec3 inNormal;
in vec2 inTexCoord;

out vec3 outWorldPosition;
out vec3 outWorldNormal;
out vec2 outTexCoord;

void main()
{
    vec4 worldPosition4 = vec4(inPosition, 1.0) * World;
    gl_Position = worldPosition4 * ViewProjection;
    outWorldPosition = worldPosition4.xyz;
    outWorldNormal = normalize(
        (vec4(inNormal, 0.0) * WorldInverseTranspose).xyz);
    outTexCoord = inTexCoord;
}
```

As you can see, even though the shader is the same, the syntax is very different. Some notable differences are: many cases of different naming of same keywords (e.g. types), different operator and intrinsic function sets (e.g. while GLSL uses '*' operator for matrix multiplication, in HLSL '*' means per-component multiplication, and for matrix multiplication `mul` function is used instead), different input/output declaration approaches, and many others. Also notice how in HLSL output position is a regular output variable with a special HLSL semantic `SV_Position` ('SV' stands for 'Special Value'), while in GLSL a built-in `gl_Position` variable is used instead.

To enable writing shaders for both APIs simultaneously, one would naturally want to introduce a language (maybe similar to one of the existing ones) that will be parsed and then translated to the API-specific language. And as you will see, Beholder Framework does that for uniform buffers, input/output, and special parameters (e.g. tessellation type). But because fully parsing and analyzing C-like code requires too much time-commitment, the author decided to take a slightly easier approach for the current version of the framework.

Here is the same shader written in the 'Beholder Shader Language':

```
%meta
Name = DiffuseSpecularVS
ProfileDX9 = vs_2_0
ProfileDX10 = vs_4_0
ProfileGL3 = 150

%ubuffers
ubuffer Transform : slot = 0, slotGL3 = 0, slotDX9 = c0
    float4x4 World
    float4x4 WorldInverseTranspose
ubuffer CameraVertex : slot = 1, slotGL3 = 1, slotDX9 = c8
    float4x4 ViewProjection

%input
float3 Position : SDX9 = POSITION, SDX10 = %name, SGL3 = %name
float3 Normal   : SDX9 = NORMAL,   SDX10 = %name, SGL3 = %name
float2 TexCoord : SDX9 = TEXCOORD, SDX10 = %name, SGL3 = %name

%output
float4 Position: SDX9=POSITION0, SDX10=SV_Position, SGL3=gl_Position
float3 WorldNormal   : SDX9 = TEXCOORD0, SDX10 = %name, SGL3 = %name
float3 WorldPosition : SDX9 = TEXCOORD1, SDX10 = %name, SGL3 = %name
float2 TexCoord      : SDX9 = TEXCOORD2, SDX10 = %name, SGL3 = %name

%code_main
    float4 worldPosition4 = mul(float4(INPUT(Position), 1.0), World);
    OUTPUT(Position) = mul(worldPosition4, ViewProjection);
    OUTPUT(WorldPosition) = worldPosition4.xyz;
    OUTPUT(WorldNormal) = normalize(
        mul(float4(INPUT(Normal), 0.0), WorldInverseTranspose).xyz);
    OUTPUT(TexCoord) = INPUT(TexCoord);
```

As you can see, `%meta`, `%ubuffers`, `%input`, and `%output` blocks can be easily parsed using a finite-state automaton and translated into either HLSL or GLSL in an obvious way (`slotDX9` and `SDX9` are needed for `vs_2_0` HLSL profile used by Direct3D 9). But to translate the code inside the main function, the author had to use a more 'exotic' tool – C macros, which, fortunately, are supported by both HLSL and GLSL.

Using macros helps to level out many of the language differences. Type names are translated easily, so are many intrinsic functions. Input and output macros for GLSL while being not so obvious are, nevertheless, absolutely possible. For example, input/output declaration that is generated by the framework for OpenGL looks simply like this.

```
#define INPUT(x) bs_to_vertex_##x
in float3 bs_to_vertex_Position;
in float3 bs_to_vertex_Normal;
in float2 bs_to_vertex_TexCoord;

#define OUTPUT(x) bs_to_pixel_##x
#define bs_to_pixel_Position gl_Position
out float3 bs_to_pixel_WorldPosition;
out float3 bs_to_pixel_WorldNormal;
out float2 bs_to_pixel_TexCoord;
```

While using macros does not make the unified shader language as beautiful and concise as it could be if it was being parsed and analyzed completely, it still makes writing a shader for all APIs at once not much harder than writing a single shader for a specific API, which is the main goal of a unified shader language.

VI. PITFALLS OF USING OPENGL AS DIRECT3D

Since Direct3D and OpenGL are being developed independently and only the fact that they must work with the same hardware makes them be based on similar concepts, it comes with no surprise that the APIs have many subtle differences that complicate the process of making one API work like another. In this section we will discuss the most notable of such differences and ways to overcome them.

A. Rendering to a Swap Chain

While Direct3D, being tightly integrated into Windows infrastructure, applies the same restrictions for both on-screen (swap chain) render targets and off-screen ones, in OpenGL the restriction can be unexpectedly different. For example, at the time of writing, Intel HD3000 on Windows does not support multisampling and several depth-stencil formats for on-screen rendering that it supports for off-screen rendering using OpenGL.

To counter this, Beholder Framework uses a special off-screen render target and an off-screen depth-stencil surface when a developer wants to render to a swap chain, and then copies render target contents to the screen when `Present` method of a swap chain is called. This may seem like overkill, but as you will see in the next section, it has more benefits to it than just being an easy way to overcome OpenGL limitations.

B. Coordinate Systems

Despite the common statement that "Direct3D uses row vectors with left hand world coordinates while OpenGL uses column vectors with right hand world coordinates", it is simply not true. When using shaders, an API itself does not even use the concept of world coordinates, and, as demonstrated in the previous section, GLSL has the same capabilities of working with row vectors (which means doing vector-matrix multiplication instead of a matrix-vector one) as HLSL. Nevertheless, there are still two notable differences between OpenGL and Direct3D pipelines that are related to coordinate systems.

The first difference is Z range of homogeneous clip space. While Direct3D rasterizer clips vertex with position *p* when *p.z / p.w* is outside of [0,1] range, for OpenGL this range is [-1,1]. Usually, for cross-platform applications it is recommended to use different projection matrices for Direct3D and OpenGL to overcome this issue [10]. But since we are controlling the shader code, this problem can be solved in a much more elegant way by simply appending the following code to the last OpenGL shader before rasterization:

```
gl_Position.z = 2.0 * gl_Position.z - gl_Position.w;
```

This way, Z coordinate of a vertex will be in the correct range, and since the Z coordinate is not used for anything else other than clipping at the rasterization stage, this will make Direct3D and OpenGL behave the same way.

The second coordinate-related difference is texture coordinate orientation. Direct3D considers the Y coordinate of a texture to be directed top-down, while OpenGL considers it to be directed bottom-up.

While the natural workaround for this difference would seem to be modifying all the texture-access code in GLSL shaders, such modification will significantly affect performance of shaders that do many texture-related operations. But since the problem lies in texture coordinates, which are used to access the texture data, it can be also solved by inverting the data itself.

For texture data that comes from CPU side this is actually as easy as feeding OpenGL the same data that is being fed to Direct3D. Since OpenGL expects the data in bottom-up order, it can be inverted by feeding in in top-down order, in which it is expected by Direct3D.

For texture data that is generated on the GPU using render-to-texture mechanisms the easiest way to invert the resulting texture is just to invert the scene before rasterization by appending the following code to the last shader before the rasterization stage (the same place where we appended the Z-adjusting code):

```
gl_Position.y = -gl_Position.y;
```

This will make off-screen rendering work properly, but when rendering a final image to the swap chain, it will appear upside-down. But, as you can remember, we are actually using a special off-screen render target for swap-chain drawing. And thus, to solve this problem, we only need to invert the image when copying it to the screen.

C. Vertex Array Objects and Framebuffer Objects

Starting from version 3.0, OpenGL uses what is called "Vertex Array Objects" (usually called VAOs) to store vertex attribute mappings. This makes them seem to be equivalent to Direct3D Input Layout objects and makes one want to use VAOs the same way. Unfortunately, VAOs do not only contain vertex attribute mappings, but also the exact vertex buffers that will be used. That means that for them to be used as encapsulated vertex attribute mapping, there must be a separate VAO for each combination of vertex layout, vertex buffers, and vertex shader. Since, compared to just layout-shader combinations, such combination will most likely be different for almost every draw call in a frame, there will be no benefit from using different VAOs at all. Therefore, Beholder Framework uses a single VAO that is being partially modified on every draw call where necessary.

Unlike Direct3D 11 that uses "Render Target Views" and "Depth-Stencil Views" upon usual textures to enable render-to-texture functionality, OpenGL uses a special type of objects called "Framebuffer Objects" (usually called FBOs). When actually doing rendering to a texture, FBO can simply be used like a part of the device context that contains current render target and depth-stencil surface. But clearing render targets and depth-stencil surfaces, which in Direct3D is done using a simple functions `ClearRenderTargetView` and `ClearDepthStencilView`, in OpenGL also requires an FBO. Furthermore, this FBO must be "complete", which means that render target and depth-stencil surface currently attached to it must be compatible.

When clearing a render target, this compatibility can be easily achieved by simply detaching depth-stencil from the FBO. But when clearing a depth-stencil surface, there must be a render target attached with dimensions not less than ones of the depth-stencil surface.

Therefore, to implement Direct3D 11 – like interface for render-to-texture functionality on OpenGL while minimizing the number of OpenGL API calls, Beholder Framework uses three separate FBOs for drawing, clearing render targets, and clearing depth-stencil surfaces. Render target FBO has depth-stencil always detached, and depth-stencil FBO uses a dummy renderbuffer object that is large enough for the depth-stencil surface being cleared.

VII. Conclusion and Future Work

Supporting both Direct3D and OpenGL at the lowest level possible is not an easy task, but, as described in this paper, a plausible one. At the moment of writing a large part of Direct3D 11 API is implemented for Direct3D 9, Direct3D 11, and OpenGL 3.x/4.x and the project's source code is available on GitHub [11].

After collecting public opinion on the project, the author plans to implement the missing parts that include staging resources, stream output (transform feedback), compute shaders, and queries. After that the priorities will be a better shader language and more out-of-the-box utility like text rendering using sprite fonts.

References

[1] About 'bind-to-edit' issues of OpenGL API. http://www.g-truc.net/post-0279.html#menu

[2] Performance comparison of Direct3D and OpenGL using Unigine benchmarks. http://www.g-truc.net/post-0547.html

[3] List of best-selling PC video games. http://en.wikipedia.org/wiki/List_of_best-selling_PC_video_games

[4] History of competition between OpenGL and Direct3D. http://programmers.stackexchange.com/questions/60544/why-do-game-developers-prefer-windows/88055#88055

[5] Official site of the OGRE project. http://www.ogre3d.org/

[6] "Scenegraphs: Past, Present, and Future". http://www.realityprime.com/blog/2007/06/scenegraphs-past-present-and-future/

[7] Noel Llopis. "High-Performance Programming with Data-Oriented Design" Game Engine Gems 2. Edited by Eric Lengyel. A K Peters Ltd. Natick, Massachusetts 2011.

[8] Official site of Unity project. http://unity3d.com/

[9] Official site of Unigine project. http://unigine.com/

[10] Wojciech Sterna. "Porting Code between Direct3D9 and OpenGL 2.0" GPU Pro. Edited by Wolfgang Engel. A K Peters Ltd. Natic, Massachusetts 2010.

[11] Beholder Framework repository on GitHub. https://github.com/Zulkir/Beholder