Postprint of: Redlarski J., Trzosowski R., Kowalski M., Kowalski B., Lebiedź J.: "Stereoscopy in Graphics APIs for CAVE Applications" Proceedings of the 2018 Federated Conference on Computer Science and Information Systems, M. Ganzha, L. Maciaszek, M. Paprzycki (eds). ACSIS, Vol. 15, pages 893–896 (2018). DOI: 10.15439/2018F223

Stereoscopy in Graphics APIs for CAVE Applications

Jerzy Redlarski
Dept. of Intell. Inter. Systems,
Faculty of Electronics, Telecommunication and Informatics,
Gdańsk University of Technology,
G. Narutowicza 11/12,
80-233 Gdańsk, Poland
Email: jerredla@pg.edu.pl

Robert Trzosowski
Dept. of Intell. Inter. Systems,
Faculty of Electronics, Telecommunication and Informatics,
Gdańsk University of Technology,
G. Narutowicza 11/12,
80-233 Gdańsk, Poland
Email: robtrzos@pg.edu.pl

Mateusz Kowalski
Dept. of Intell. Inter. Systems,
Faculty of Electronics, Telecommunication and Informatics,
Gdańsk University of Technology,
G. Narutowicza 11/12,
80-233 Gdańsk, Poland
Email: matkowa1@pg.edu.pl

Błażej Kowalski
Dept. of Intelligent Interactive Systems,
Faculty of Electronics, Telecomm. and Informatics,
Gdańsk University of Technology,
G. Narutowicza 11/12, 80-233 Gdańsk, Poland
Email: blakowal@pg.edu.pl

Jacek Lebiedź
Dept. of Intelligent Interactive Systems,
Faculty of Electronics, Telecomm. and Informatics,
Gdańsk University of Technology,
G. Narutowicza 11/12, 80-233 Gdańsk, Poland
Email: jacekl@eti.pg.edu.pl

***Abstract*—The paper compares the advantages and disadvantages of a variety of Graphics Application Programming Interfaces (APIs) from the perspective of obtaining stereoscopy in applications written for a CAVE virtual reality environment. A number of problems have been diagnosed and an attempt has been made to solve them using OpenGL, DirectX 11 and 12, Vulkan, as well as the Unity Engine which can internally use DirectX, OpenGL and Vulkan, but has problems and limitations of its own.**

I. Introduction

STEREOSCOPY (stereo imaging) is a technique for producing an illusion of depth by delivering two different 2D images (generated from two different points of view) to each of the viewer's eyes. It is based on predator (and human) way of 3D perception by means of binocular vision (stereopsis). Stereoscopy can rely on direct delivery of two images on separate screens mounted in front of each eye (stereoscope, HMD – Head-Mounted Display) or displaying two images on a common screen visible for both eyes and using special filtering glasses separating images (3D cinema, CAVE – CAVE Automatic Virtual Environment). Filtering glasses can use various technologies: active (separation in time by shutter glasses) and passive (spectrum separation or polarization separation) [1, 2, 4].

The Immersive 3D Visualization Lab (I3DVL) located at the Faculty of Electronics, Telecommunication and Informatics of the Gdańsk University of Technology contains three CAVEs of various sizes: closed BigCAVE with six screen-walls, open MidiCAVE with four screen-walls and MiniCAVE based on four 3D monitors [3, 5, 7]. Immersive 3D visualization in these CAVEs requires stereoscopy and its synchronization on all screen-walls. The popular game engine Unity serves as the basic development tool in I3DVL. Unfortunately, it offers limited support for the creation of software for CAVEs and needs some adaptation.

CAVE systems put a number of unique requirements on applications, such as the need to synchronize a large number of screens or projectors, or the need to support active and passive stereoscopy, preferably at the same time. This in turn may necessitate the use of a quadbuffer (a screen buffer with room for four screen-sized pictures) or similar solution, if graphical artifacts are to be avoided. Rendering has to happen at the correct frequency (120 Hz for our BigCAVE system), or else frame skipping may happen – or worse, stereoscopy is disrupted if frames intended for one eye are displayed to the other eye. Latency should also be minimized, as delays in rendering are known to induce discomfort and dizziness in viewers, sometimes preventing them from extended use of virtual reality.

Most of the applications run on our CAVE system have been developed in the Unity engine, using a third party, proprietary library to achieve synchronization and stereoscopy. However the library had major limitations, including unsatisfactory performance and inability to work with Unity versions past 5.0.2. We also wanted a solution where we'd have access to the source code, so that students and researchers could implement not only applications, but also make changes to or expand the library itself if necessary. This forced us to start working on a new library, since the existing ones were either outdated or had proprietary licenses.

For a long time, the Unity Engine itself had no native support for stereoscopy, while also limiting direct access to the underlying graphics APIs – particularly during the initialization stage. Thus, the only way to achieve synchronization and stereoscopy was to render the scene to a texture, then use a separate rendering context – created outside of Unity – to display the resulting image. Since the contexts

This work was supported in parts by Entity Grant to Finance the Maintenance of a Special Research Device (SPUB) from the Ministry of Science and Higher Education (Poland) and DS Funds of the Faculty of ETI at the Gdańsk University of Technology

were separate, it was possible to use different graphics APIs for each task, so our testing included using DirectX 11 and OpenGL in Unity to render images for both eyes, while using OpenGL, DirectX 12 or Vulkan in another window to display those images at the correct frequency and synchronized between projectors and with active glasses.

II. Single camera stereoscopy

One of our attempts involved the simplest way to achieve stereoscopy in Unity – by alternating the placement of the camera every other frame. To achieve fluid stereoscopic animation, the camera is synchronously (on all client computers) moved back and forth between points A and B, representing the position of left and right eye respectively. This needs to happen exactly 120 times per second (for our setup), each frame shown for the exact same time. It also needs to be synchronized with the active 3D glasses, which alternate between darkening left and right LCD shutters at the same rate.

While alternating the camera's position between frames is trivial, implementing the synchronization with glasses in Unity is problematic. Even when configured to keep a fixed frame rate of 120 frames per second, it's common for frames to vary in length, depending on unpredictable changes in rendering times. This results in the viewer often seeing either two overlapping pictures, or reversed pictures, where the left eye sees the picture intended for the right eye and vice versa – a (dizzying for the viewer) occurrence referred to as a desynchronization.

Attempts to reduce the issue by increasing the frame rate to 240 Hz didn't solve the problem, even when the number of frames between switching camera positions was adaptively adjusted based on the amount of time a frame actually took to render. Increasing the frame rate further (360 Hz) was barely possible, since even an empty scene in Unity, with no running scripts, physics or camera movement, typically oscillates around 300 to 330 frames per second.

In best cases, using this method resulted in stereoscopy that would switch between correct and reversed pictures every few seconds, and after less than a minute resulted in complete desynchronization. Most likely the problem lies in the fact that the Unity Engine is not designed with constant frame rates in mind. Additionally, this method is useless in applications that – due to their complexity – suffer from occasional or frequent drops in frame rates. Such applications would need to be better optimized and tested – which most applications don't do sufficiently for this method to work. This is especially true if they don't normally (for use without stereoscopy) need a constant frame rate of 120 frames per second – such as when the movement is very slow.

III. Stereoscopy support in Unity

To properly display stereoscopic graphics in a CAVE installation, a number of tasks needs to be done. Each screen requires two images rendered from two points – representing the left and right eyes. Each of the viewer's eyes can view any part of any screen at any time, in contrast to HMD (head mounted display) devices which split the display into parts visible by one eye only. In a CAVE, the left and right eye images need to use the same area of the display screens. To obtain correct perspective, the position of the eyes in regard to the display surface is crucial – it is calculated using the head tracking system. The interpupillary distance is then used to position both cameras correctly in eye positions. The images on screens that share an edge need to be consistent. This is achieved by correctly setting the projection matrices and view matrices, achieving an asymmetric frustum with edges passing through the corners of the display. The display surface is where the images for both eyes converge. Unlike in HMD devices, the frustum does not follow the rotation of the viewer's head, but rather is always facing the same direction. If the technology used (such as active shutter glasses) requires the left and right eye images to alternate, it is crucial to synchronize this between all screens, so that they all display images intended for the same eye at the same time – especially if each screen is governed by a separate instance of the application. To achieve this, the usual solution is to prepare two images every application update, then pass the task of displaying the correct one to synchronized graphics cards such as NVIDIA Quadro Sync (used in I3DVL).

When OpenGL is used to write software, one can use quadbuffers to achieve stereoscopy [9]. Firstly, one needs to set the `PFD_STEREO` flag in the `PIXELFORMATDESCRIPTOR` structure using the `SetPixelFormat` function [11]. This can only be done during context initialization. This in turn makes it possible to render to the left and right backbuffers, switching between them using `glDrawBuffer(GL_BACK_LEFT)` and `glDrawBuffer(GL_BACK_RIGHT)` calls.

In the Unity3D engine, rendering is done with the `Camera` component. To configure it for each CAVE screen the `projectionMatrix` and `worldToCameraMatrix` fields have to be set with appropriate matrices. We tried to use quadbuffering in Unity, but it proved impossible due to the lack of low-level access to the graphics device during the initialization step. It was not possible to set the aforementioned flag. We found two workarounds to this problem.

The first one was to inject a modified DLL library during application launch. The library would intercept the `SetPixelFormat` function call and set the `PFD_STEREO` flag, resuming with normal initialization afterwards. A program launched this way could use two cameras, switching between the target back-buffers in the `OnPreCull` camera event from the native plugin level. A native plugin in Unity is one that is compiled from non-virtual machine languages such as C and C++. The disadvantage of this solution is that it's hard to implement and requires a separate piece of software for injecting the DLL at application launch.

The other workaround is to create a separate OpenGL window (from a native plugin) which is only tasked with displaying the images, while the Unity application renders those images to a virtual texture and passes them to the OpenGL window. When launched, the Unity application would create the window (which needs to belong to the same process) with a new graphics context initialized with

the PFD_STEREO flag. Using the wglShareLists function we can enable the Unity context and the OpenGL window context to share a single display-list space. On the Unity side the two cameras create RenderTexture objects, the virtual textures, which are set as targetTexture of the cameras. This results in Unity rendering the image for left and right eye to those textures. The GetNativeTexturePtr function gives us pointers to these textures which can be passed to the OpenGL window. Thanks to resource sharing, the window can then bind those textures with the glBindTexture function. All that is left to do is for the window to draw those textures to the correct backbuffers.

This workaround has a number of disadvantages. One is the slightly lowered performance because the main unity application is now considered by the operating system to be running in the background, while the OpenGL window which does almost no work is in the foreground. This can result in suboptimal assignment of processing power to the threads, since modern operating systems tend to prioritize foreground windows. Another problem is that input from devices such as keyboard and mouse will be directed to the foreground window, which may require redirecting the input events to the Unity application.

Unity3D has been offering support for VR (virtual reality) applications for a while, but their focus is on HMD devices [8]. Since 5.4 version, they extended support to include stereoscopy-capable flat panel displays. The newly added "Stereo Display (non head-mounted)" option makes it possible to obtain stereoscopic images which can be rendered using a variety of graphic APIs – OpenGL, Vulkan, Direct3D 11 and Metal. At the time of this writing, it doesn't work with Direct3D 12 yet. The Camera object can be set to render images for both eyes or we can use two Camera objects each responsible for one eye. The matrices for these cameras are set using the SetStereoProjectionMatrix and SetStereoViewMatrix functions. We can also set in the project options whether rendering should alternate between left and right images or if we want to render both images simultaneously (known as the Single-Pass Stereo rendering optimization [10]).

Using Unity's engine for stereoscopy has a number of advantages. The engine unifies the implementation of stereoscopy for the different graphics APIs and operating systems. The engine itself also performs a multitude of optimizations (such as the aforementioned Single-Pass Stereo rendering) [8]. Creating and launching the application is easier – no need for injecting libraries or intercepting control devices input from the second window. However, there are also drawbacks. The main one is that (non-HMD) stereoscopy support in Unity is fairly new and a niche need, and thus software bugs and instability are common. Common problems include the random freezing of rendering for one of the eyes, especially with HDR (High Dynamic Range) or MSAA (Multi-Sample Anti-Aliasing) on. Other problems were encountered with Reflection Probes and Terrain objects (such as trees) improperly displaying shadows, which were rendered to the wrong eye or not at all, as shown in (Fig. 1).

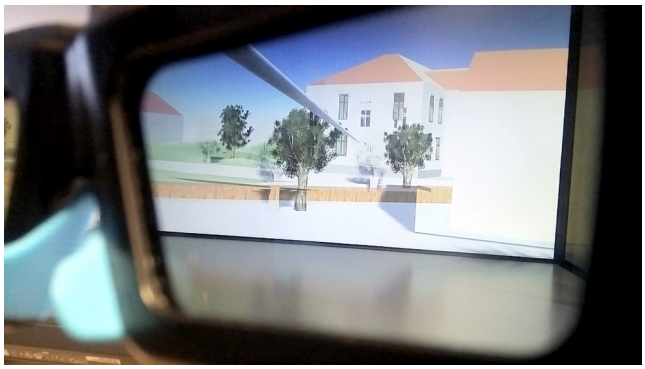

Fig 1. Improperly displayed tree shadows in Unity, as seen through active glasses.

IV. DirectX 12 and Vulkan in separate context

Vulkan is a modern 3D graphics API, released in 2016. Its main goal is to provide a low-level alternative to OpenGL as a multiplatform API, and thus offer better efficiency and more direct control over computations [12]. Compared to older APIs it excels at using multiple CPU cores in parallel (through support for multithreading), and the CPU load at runtime is reduced greatly by precompiling shaders to SPIR-V, an intermediate form that allows drivers to be much simpler [6].

Despite its advantages, Vulkan has a few drawbacks too. It is more verbose, requiring more code, and thus work, to get it to do things that are easier done in higher-level languages. This also means more room for bugs and harder maintenance. Another problem is the novelty, which means the API might see frequent changes as it matures. People looking to learn the API may have fewer options compared to older APIs that have hundreds of books and tutorials available. People who run into problems are less likely to find solutions on the internet. Few development tools, third-party libraries and game engines have support for Vulkan.

For our purposes, Vulkan would offer a few significant advantages – the ability to run linux-based applications on the CAVE being the primary one. This would be a benefit for applications that intend to use the Triton supercomputer which is connected to our CAVE and runs Linux (having both the front-end and back-end run on Linux isn't strictly necessary, but might be beneficial for some applications, especially if using the high-speed Infiniband connection and/or distributing computations between Triton and the client computers in our CAVE). However, at the moment most applications developed for our CAVE use the Unity engine and thus run on both Windows and Linux, with Windows having better support. Additionally, not all of our utility software (for managing projectors, tracking, and management of virtual reality applications themselves) are available for Linux. Other advantages of Vulkan, such as lower latency and better performance, are shared with DirectX 12. Thus, for applications that want to push graphics

quality as far as possible, or use very complex scenes, while still maintaining the high frame rates necessary for comfortable VR experience and stereoscopy, either of those two APIs can be used.

Microsoft DirectX is a collection of APIs for multimedia functionality in applications, such as games and video, on Microsoft platforms: the Windows operating system and Xbox gaming console. Direct3D – the component dealing specifically with the rendering of 3D scenes, is the flagship part of DirectX and thus names DirectX and Direct3D are often used interchangeably. The DirectX software development kit (SDK), which is now a part of the Windows SDK, contains binary libraries, header source files and documentation. The latest edition of Direct3D is version 12, sharing many similarities with Vulkan – such as the low level approach, lowered CPU utilization and better support of multithreading [13].

Unity Engine allows rendering to texture, a mechanism which can be used to display a stereoscopic image with a separate application – e.g. using DirectX 12 or Vulkan. It is crucial to keep the latency low, because otherwise virtual reality can cause discomfort for users. Thus, to speed up communication between the Unity application and the window used for display, shared memory can be used. The Unity Engine would write rendered images to the shared memory, while the window would read and display them on the screens, properly synchronized with the active glasses. The engine offers the `Camera` object, which gives us control over rendered images. From the `Camera`, we can retrieve a `TargetTexture` and assign it to a `RenderTexture` object. This creates a new `Texture2D` object with the `Camera`'s texture. Calling the `GetNativeTexturePtr` method results in a resource address, which we can use to retrieve all the necessary data that needs to be shared with the DirectX 12 or Vulkan process. The process takes the raw data and needs to recreate the resource objects. Once the resource objects have been created, they need to be updated every frame. The resource objects need to be linked with the previously written `Shader`, which is tasked with separating the left and right eye pictures into appropriate 'backbuffers'. Using a `Quadbuffer` allows for fluent rendering of stereoscopic images, since the engine can render two images (back left and back right) at the same time, while two other (front left and front right) are beings displayed – either one after the other (in case of active stereoscopy) or simultaneously if passive stereoscopy is used.

V. Conclusions

We found out that the Vulkan and DirectX 12 APIs offered enough low-level control to serve for our purposes, while also offering possible performance benefits. Vulkan is also supported on more platforms, which might make it the best choice for applications that require the computing power of a supercomputer – such as the Tryton cluster connected to our CAVE, which runs on Linux. However, these APIs have disadvantages as well – due to their low level design and novelty, maintaining a library based on them would require extra work.

The Unity Engine, over time, began to support different stereoscopy technologies and other virtual reality technologies. Their main focus were HMD (Head-Mounted Display) devices, but it was also possible to use these new features for CAVE systems. However, unlike HMDs which are now mass-produced to a few specifications, CAVE systems vary a lot, each being a unique installation, with different resolutions, sizes, projector positioning, tracking systems, etc. which means many parameters have to be set by hand (such as projection matrices, viewports). Nevertheless, using Unity's native support mechanisms has many benefits – the engine unifies the implementation of stereoscopy for the different graphics APIs (DirectX 11, DirectX 12, Vulkan, OpenGL Core, Metal) and operating systems. There's also no need to share textures between two rendering contexts – thus speeding up and simplifying the application. The engine itself also performs a multitude of optimizations (e.g. Single-Pass Stereo rendering). Support of control devices is also easier – especially now that there was no need for a second rendering window.

In conclusion, using the Unity Engine as basis for our library proved to be the easiest solution, offering many advantages in terms of simplicity and ease of use, both from library and application developer point of view. The main drawback is that it forces applications to be developed in Unity, which may not be the preferred development environment for all developers. It also has other limitations, and we may have to eventually expand our library to directly use graphics APIs, likely Vulkan or Direct3D 12, when the need to develop applications using other engines (such as the Unreal Engine or VBS engine) arises in our CAVE.

References

[1] S. Gateau, D. Filion, "Stereoscopic 3D Demystified: From Theory to Implementation in Starcraft 2," Game Developers Conference GDC 2011, http://www.nvidia.com/content/PDF/GDC2011/Stereoscopy.pdf .

[2] S. Gateau, S. Nash, "Implementing Stereoscopic 3D in Your Applications," GPU Technology Conference GTC 2010, https://www.nvidia.com/content/GTC-2010/pdfs/2010_GTC2010.pdf .

[3] I3DVL, "Immersive 3D Visualization Lab," https://eti.pg.edu.pl/i3dvl .

[4] J. Lebiedź, "3D visualization," Proceedings of the Polish Conference on Computer Games Development WGK 2013 (in Polish), vol. 3, Gdańsk 2013, pp. 105-115.

[5] J. Lebiedź, J. Redlarski, "Applications of Immersive 3D Visualization Lab," 24th International Conference on Computer Graphics, Visualization and Computer Vision WSCG 2016 – Poster Papers Proceedings, Plzeň 2016, pp. 69-74.

[6] P. Łapiński, Vulkan Cookbook, Packt Publishing 2017.

[7] A. Mazikowski, J. Lebiedź, "Image projection in Immersive 3D Visualization Laboratory,".18th International Conference in Knowledge Based and Intelligent Information and Engineering Systems KES 2014, Procedia Computer Science 35, 2014, pp. 842-850, http://dx.doi.org/10.1016/j.procs.2014.08.251

[8] Unity Documentation, "How to do Stereoscopic Rendering," 2018, https://docs.unity3d.com/Manual/StereoscopicRendering.html .

[9] "NVIDIA 3D Vision Pro And Stereoscopic 3D," 2010, http://www.nvidia.com/docs/IO/40505/WP-05482-001_v01-final.pdf

[10] Unity Documentation, "Single-Pass Stereo rendering," 2018, https://docs.unity3d.com/Manual/SinglePassStereoRendering.html

[11] Microsoft Developer Network, "OpenGL on Windows", 2018 https://msdn.microsoft.com/en-us/library/dd374293(v=vs.85).aspx

[12] The Khronos Group, Inc. 2018, https://www.khronos.org/vulkan/

[13] Microsoft Developer Network, "Direct3D 12 Programming Guide", 2018, https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121(v=vs.85).aspx