Developmental Milestones of Graphics Technologies

Dzmitry Mazouka
Department of Information Management Systems
Belarusian State University
Minsk, Belarus
mazovka@bk.ru

Viktor Krasnoproshin
Department of Information Management Systems
Belarusian State University
Minsk, Belarus
krasnoproshin@bsu.by

Abstract. **The complexity of modern graphics hardware and software has a long history. In this paper we review the historical milestones of computer graphics development. This analysis will help us to understand the common problems and trace a direction for the future improvements.**

Keywords: **computer graphics, graphics pipeline, directx**

I. INTRODUCTION

Computer graphics is a very distinct field in computer science. Unlike other areas, the primary concern of computer graphics lies in the presentation of information, rather than pure computation. This makes computer graphics universally applicable in a vast number of human activities, including science, engineering, manufacturing and entertainment. In a peculiar turn of events, computer graphics development and evolution has been primarily driven by the entertainment industry. Started late 20th century, video games have been at the forefront of computer graphics progress, which was later picked up and expanded by the movie industry. These days it is hard to find an artifact of the popular culture that was not produced using computers in one way or another.

Computer graphics technologies pursue two fundamental goals: photorealism and real-time imagery generation. Despite tremendous progress in the past decades, we are still far from being able to render 3D scenes of arbitrary complexity with perfect picture quality. Perhaps it is not an achievable goal after all, but the development will always be concerned with image quality and the speed of generation.

From general perspective, the technological stack of computer graphics consists of the following layers: graphics hardware, graphics Application Programming Interface (API), visualization system, and application.

Application layer defines a particular visualisation problem that needs to be solved with visualisation tools. It can be a video game, a visual effect in a movie, or a piece of engineering software. The requirements for application layer are defined externally and they often drastically differ from one problem, or product, to another. This is where the need for quality and speed arises, to propagate, eventually, to the lower layers. Due to the nature of application problems, little can be done for formalisation of this layer.

Visualisation system layer can be optional, however complex applications almost never get built from scratch. Many years of experience in various fields of computer graphics provided software engineers with enough information to construct systems that would be applicable to a wide array of visualisation problems. The best example of that would be a graphics engine, as a part of a game engine. A graphics engine (for example: Unreal Engine, Source, Frostbite) is a visualisation system that is developed and tuned to a specific subset of applications – game genres. Common visualisation problems are solved once in an efficient way in a graphics engine and that facilitates higher production speed for subsequent projects that a game development company may take. Another good example of a visualisation system can be Computer-Aided Design (CAD) systems (for example: AutoCAD, SketchUp, Archicad). These also solve common problems, but in the areas of engineering and architecture. The major difference of visualisation system layer with application layer, is that visualisation system plays the role of middleware, and each specific visualisation application problem has to evaluate and choose whether any existing visualisation system can help with the solution or not. None of the visualisation systems are applicable to all possible applications.

Graphics API layer is the most standardised layer in the technological stack. The API is normally represented by operating system drivers and specialised graphics libraries. Of which, the most prominent are: DirectX, OpenGL and Vulkan. These libraries do the mediation work between software and hardware within an operating system, which includes translation of application intent into graphics hardware commands, and control and execution of the rendering process. The development of the graphics libraries is substantially slower than visualisation systems, as they heavily depend on architecture of graphics hardware, and when the hardware updates, the libraries have to change accordingly. Graphics API represents a common

Pattern Recognition and Information Processing (PRIP'2021) : Proceedings of the 15th International Conference, 21–24 Sept. 2021, Minsk, Belarus. – Minsk : UIIP NASB, 2021. – 246 p. – ISBN 978-985-7198-07-8.

language that graphics hardware and software talk to each other, that is why there is so few implementations of it.

Graphics hardware is the base layer in the technological stack of computer graphics. It consists of various specialised hardware components that perform rendering. Most notable form of graphics hardware is graphics cards (for example: NVIDIA GeForce, AMD Radeon). A graphics card contains a dedicated Graphics Processing Unit (GPU), which runs a special algorithm called graphics pipeline. In a similar manner with visualisation systems, graphics hardware accumulates common patterns and solutions to common visualisation tasks coming from applications. And the changes then propagate via graphics API up, towards visualisation systems and applications.

Despite all the successes that computer graphics has enjoyed so far, there is one problem that is growing with each new feature and every new improvement: the complexity of the technological stack. In this paper we will analyse the developmental milestones of the technological stack, highlight the changes and the reasoning behind them. This will help with understanding of the technology in its current state and possibly give ideas for further development.

II. Method of Analysis

In this paper we will use a model for historical analysis that is based on the technological stack described earlier. We will look at three separate actors: graphics hardware, graphics API, and applications which is a combination of application and visualisation system layers from the technological stack.

Graphics hardware, graphics API and applications relate to each other in the manner depicted in Fig. 1. The relationship between the actors is cyclical and flows in one direction, with arrows pointing at the actors the development of which is informed by the source of the arrow. Graphics hardware development is informed by the nature and requirements of the applications, graphics API reflects the structure and capabilities of the graphics hardware, and applications can only do as much as it is possible with a certain API.

Of course, there is also external input into this system, for instance, graphics hardware is not being developed in a complete isolation and depends on the current state of art in chip manufacturing and related improvements in technologies. Applications are also influenced by the ideas of business products, or advances in a scientific or engineering thought. Graphics API may be influenced by competing technologies or general evolution in software development techniques. But for our purpose, a simpler model will be enough.

With every turn of the relationship cycle, the components change and improve. We identified DirectX API [1] versioning as a representative timeline of the stepping stones that graphics technologies have taken on the way, and in the next section we will begin with the first version.

III. Analysis

A. DirectX 1 (GameSDK) – 1995

First GPUs were pretty simple, and were mostly concerned with 2D image processing and displaying. This was already a good starting point, since it introduced a separation of duties within a single machine: general CPU did not have to bother with graphical tasks consisting mostly in copying large buffers of data from one memory location to another.

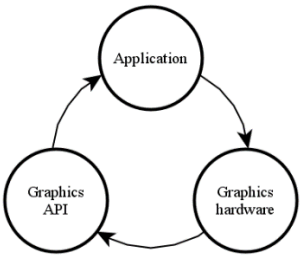

Fig. 1. Actors' relationship cycle

In a similar vein, DirectX 1 only contained a library called DirectDraw, dedicated specifically to the work with 2D graphics. This library unified and abstracted the work with video memory, so that the users would not need to bother about the kind of hardware their applications were running on. This technique was called Hardware Abstraction Layer (HAL).

If application required 3D capabilities, it was mostly on its own, all geometry preparation, including transformation and projection had to be performed by the CPU, and then passed to the graphics hardware via API. But even at this early stage, DirectDraw provided access to double buffering and Z-buffer support, which are essential features to this day.

B. DirectX 2 – 1996

Voodoo 3dfx graphics acceleration card [2] was a major step in development of what has become known as graphics pipeline. The card implemented the rasterisation algorithm and relied on the presence of another video card in the system for 2D output. Rasterisation was another essential step on the way to the true 3D rendering, but transformation and projection still had to be done on the CPU.

DirectX 2 had extensively improved capabilities of DirectDraw, and introduced a library to work with 3D graphics: Direct3D. Direct3D in DirectX 2 could be used in two separate modes with distinct APIs: Immediate

Pattern Recognition and Information Processing (PRIP'2021) : Proceedings of the 15th International Conference, 21–24 Sept. 2021, Minsk, Belarus. – Minsk : UIIP NASB, 2021. – 246 p. – ISBN 978-985-7198-07-8.

Mode (IM) and Retained Mode (RM). Retained Mode was designed for high-level graphics programming, and included functionality and primitives for construction and management of 3D scenes containing hierarchies of objects. RM contained methods for camera manipulation and animations and was built on top of Immediate Mode. Immediate Mode on the other hand, was a low-level programming interface and required good understanding of the intricacies of graphics programming. This demonstrates that graphics development complexity was recognised even at these early stages.

Even though graphics hardware did not support lighting, transformation and projection at the time, DirectX implemented the missing parts of the rasterisation algorithm in software, and provided specialised components in the form of Transform, Lighting and Raster modules.

C. DirectX 3 – 1996

DirectX 3 did not have any improvements to DirectDraw or Direct3D, and only updated other components of the SDK, such as DirectSound, DirectInput and DirectSetup.

It should also be noted that DirectX version 4 was skipped and the next released version became DirectX 5.

D. DirectX 5 – 1997

In 1997 graphics cards manufacturers introduced a new data transfer interface in their products: Accelerated Graphics Port (AGP) [3]. This data transfer bus significantly increased the rate at which graphics data could be passed to graphics hardware. That, in turn, helped to increase the size of 3D scenes and quality of textures. In addition to this, graphics hardware added support of multitexturing – an ability to use multiple texture maps during single surface rasterisation. This feature had opened a way to many visualisation techniques, such as bump mapping, specular mapping, prebaked lighting and so on.

When it was introduced, Direct3D Immediate Mode required programmers to record the instructions that they wanted to pass to graphics hardware in a special data structure called Execute Buffer. Execute Buffers were pretty low level and required a fair amount of boilerplate code when operated. For this reason, DirectX 5 added a more convenient set of instructions in addition to Execute Buffers: Draw Primitive commands. Direct3D Retained Mode was expanded with a number of interfaces for animation and managing geometry with variable levels of details.

With the new improvements of the hardware and API, applications could render scenes with textured materials more efficiently and the selection of available visual effects have increased.

E. DirectX 6 – 1998

In DirectX 6, DirectDraw did not have any significant changes, but it increased the number of methods that simplified working with graphics hardware. Direct3D Immediate Mode improved its performance and added support for new hardware features: single-pass multiple texture blending, texture cache management, vertex buffers, and many others. Direct3D Retained Mode was incrementally improved without any noteworthy changes.

Graphics application were provided with a better selection of tools as the result of these changes; however, the more basic capabilities were still quite primitive, for instance, lighting and transformation was still performed by CPU, and this prompted the next big challenge for the hardware.

F. DirectX 7 – 1999

In 1999 graphics cards got a new module that extended the capabilities of the hardware graphics pipeline: a Transformation and Lighting module (T&L) [4]. This change allowed to remove from the CPU the need of performing geometry transformation (projection from 3D coordinate system onto 2D screen coordinates), and per-vertex lighting calculations. Now, graphics hardware started processing the true 3D data.

The new features of Direct3D included: T&L support, environment mapping with cubic textures, geometry blending, device state blocks. Additionally, Execute Buffers support was ceased, and draw primitive methods have become the only way of pushing the work to graphics hardware.

Direct3D Retained Mode was completely removed from DirectX SDK. The reason why it was done can be speculated that a decision was made to concentrate efforts on a single component of the library – Immediate Mode, – which would be better suited as a low-level, high-performance interface to graphics hardware. Retained Mode could not be made generic enough, and could not play the role of a general graphics engine. In lieu of removed Direct3D RM, DirectX introduced a special Direct3DX Utility Library (D3DX) that contained a wide selection of functions that helped with management of Direct3D interface objects, provided functions for loading graphical assets from files, and a range of 3D math functions. This made Direct3D a more complete package for graphics programming.

These improvements to hardware and API had increased performance of rendering applications. But there was another problem looming in the background: graphics pipeline as it was, was implemented in a particular manner, called Fixed Function Pipeline (FFP) [5]. Which meant, that the implementation of the pipeline algorithm was static, and the data passing

Pattern Recognition and Information Processing (PRIP'2021) : Proceedings of the 15th International Conference, 21–24 Sept. 2021, Minsk, Belarus. – Minsk : UIIP NASB, 2021. – 246 p. – ISBN 978-985-7198-07-8.

through was transformed on the way using fixed functions that could only be changed by special state values. FFP was a significant limiting factor in the generality of the pipeline: application developers were coming up with all sorts of possible visual effects that simply could not be implemented with the fixed functions.

G. DirectX 8 – 2000

The problem of FFP was solved with the introduction of programmable pipeline technology in graphics hardware. Programmable pipeline introduced two new stages into the process as a replacement for Transformation and Lighting and multitexturing modules – Vertex and Pixel Shaders [6]. Shader is a special small program that is executed by GPU at certain stages of data processing, and is applied to specific types of primitives: vertices for vertex shaders and fragments (or pixels, loosely) for pixel shaders. Shader programs were written at first in a version of assembly language that could be compiled in the runtime by application and uploaded to graphics hardware. Later, however, high level languages were developed to simplify shader programming, such as High-Level Shader Language (HLSL). HLSL later developed in a number of stages defined by Shader Model version, thus the first implementation was using Shader Model 1.

In DirectX 8, DirectDraw was completely merged into Direct3D since there was a little need in maintaining a separate library for display device management. The new combined component was named DirectX Graphics, and it included support of programmable pipeline and implemented a number of features, including: multisampling, point sprites, 3D volumetric textures, higher-order surface primitives, multiresolution geometry, vertex blending. Another useful addition was introduction of resources and the ability to manage where graphics data should be located in the memory, which gave applications better control over the data flow.

Direct3DX library was significantly expanded with functions that support working with meshes, geometry skinning (vertex blending), functions to assemble shaders, and a specialised Effect interface that encapsulated some of the common work of defining graphics pipeline using declarative syntax.

H. DirectX 9 – 2002

DirectX 9 had become a standard for Windows graphics development for many years to come. Even after the following versions were released, DirectX 9 was still in use. It can be said that this version encapsulated most of the requirements posed by applications, and some significant changes were needed on the application side in order to facilitate further development.

There were no radical changes in DirectX 9 compared to DirectX 8. All of the previous capabilities of the API were enhanced and improved. The API model underwent minor iterative version releases, which supported further extensive development in graphics hardware. HLSL was updated to Shader Model 2 and 3.

For the following years, graphics hardware, graphics API and applications were developing in an extensive manner, improving performance and increasing the number of supported resources.

I. DirectX 10 – 2006

In DirectX 10, graphics pipeline model was changed fundamentally. Legacy features of DirectX 9, like fixed function pipeline, were stripped off. And in general, the API had been upgraded and made cleaner. The following functional improvements were made: added a new programmable shader stage – geometry shaders (Shader Model 4); ability to output vertex data from the pipeline; pipeline state was organised into 5 immutable objects that significantly reduced loss of performance due to state switching; improved resource access; changed API architecture to have a layered runtime; and many others.

A drastic change like that meant that a lot of applications created using older versions of API could not be ported easily to use the new API. That throttled a widespread adoption of DirectX 10 for some time.

J. DirectX 11 – 2009

It had become clear at the time that programmable pipeline was the appropriate technology of choice for graphics hardware, since it combined great performance with a lot of flexibility necessary for applications. Thus, the main ways of graphics hardware development were to improve on the capabilities of shader stages and introduction of new ones.

DirectX 11 kept the architecture model of its predecessor. It expanded shaders to Shader Model 5 with addressable resources and resource types, subroutines, new types of shaders: compute, hull and domain. Two important improvements were introduced in DirectX 11: already mentioned compute shaders and multithreading. Compute shaders were a big change in graphics hardware world, they made it possible to execute general parallel algorithms very efficiently. Afterall, that was the whole purpose of graphics hardware from the beginning – to process large amounts of data in the most efficient manner. Now, graphics hardware had discovered a new use, and no longer was locked just to rendering. Multithreading support, on the other hand, was a big win for the rendering itself. Up until that moment, rendering processes were structured as a single conveyor belt with a single global state, which limited any attempts at parallel execution. And with GPUs hitting the limits of

Pattern Recognition and Information Processing (PRIP'2021) : Proceedings of the 15th International Conference, 21–24 Sept. 2021, Minsk, Belarus. – Minsk : UIIP NASB, 2021. – 246 p. – ISBN 978-985-7198-07-8.

single core improvements, it meant that some serious change was needed in order to unlock the next performance boost, and that change was multithreading.

From the applications perspective, there was a split: simpler applications that did not need cutting edge features were still using DirectX 9, but large players recognised and promoted DirectX 11 further. And the next version of DirectX was a conclusion of this endeavour, so far.

K. DirectX 12 – 2015

DirectX 12 had made the next large update to the architecture of the API. Compared with DirectX 11, the new version sported [7]: vastly reduced CPU overhead, up to 20% improvement in GPU efficiency and cross-platform development across Windows 10 devices. This came at a cost of the API being made lower-level, without attempting to abstract hardware capabilities any longer, it instead gave control over the hardware to graphics programmers.

The API had become thinner, and a lot of opportunities were opened for performance optimisation in relation to concrete visualisation applications. This, however, had significantly increased the entry level, and API documentation [7] explicitly said that DirectX 12 was designed for advanced graphics programmers. In a way, the API had become closer to the first version – DirectX 1. The API removed multiple ways it previously used to synchronise data and state between CPU and GPU processes, now all the work for resource management and command execution had to be performed by the application. Work submission was made truly parallel with introduction of a new model based on command lists. Those command lists may be reminiscent of Execute Buffers in the early versions, but in DirectX 12 they represented a completely independent executable workload for hardware that did not share a state with the rest of the computation, so it could be constructed and submitted in parallel in a stateless manner.

IV. Conclusion

The development arc of the graphics hardware, graphics API and applications has been dramatic in the past 25 years. The API started from low-level, attempted to add high-level abstractions, but eventually gave up and now it mirrors the hardware architecture. The hardware tried implementing rasterisation algorithm in a static manner, but it was proven not flexible enough to support the variety of applications, so the programmable pipeline was developed and enhanced. Applications started small in numbers, but then the number of them exploded, and after introduction of newer architectural models, a split appeared between simpler and more complex systems. Nowadays, DirectX 9 has become outdated, and it is harder for graphics programmers to base new applications purely on graphics APIs. In game development, for instance graphics engines have become very prominent since their development teams had enough expertise to utilise the power of the new API fully.

In our previous works we discussed the problem of complexity of the graphics pipeline and API [8, 9, 10]. Our approach to extension of the pipeline with higher-level primitive processing can make graphics API better suited for use in complex graphics application going forward.

References

[1] Microsoft, DirectX SDK Documentation for versions 1–12, 1995–2015.

[2] “Rise of 3dfx”, https://vintage3d.org/3dfx1.php

[3] Intel Corporation, “Accelarated Graphics Port Interface Specification”, 1996.

[4] NVIDIA Corporation, “Transform and Lighting”, Technical Brief, http://developer.download.nvidia.com/assets/gamedev/docs/TransformAndLighting.pdf

[5] David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors, 2nd ed., Morgan Kaufmann, 2013, pp. 23–39.

[6] Ron Fosner, Real-Time Shader Programming, Morgan Kaufmann, 2003, pp. 88–111.

[7] Microsoft, DirectX 12 Programming Guide, https://docs.microsoft.com/en-us/windows/win32/direct3d12/what-is-directx-12.

[8] V. Krasnoproshin, D. Mazouka, “Frame Manipulation Techniques in Object-Based Rendering” Communications in Computer and Information Science, vol. 673: “Pattern Recognition and Information Processing”, Springer, 2017, pp. 97–105.

[9] V. Krasnoproshin and D. Mazouka, “Graphics Pipeline Evolution Based on Object Shaders” Pattern Recognit. Image Anal. 30, 2020, pp. 192–202, https://doi.org/10.1134/S105466182002008X

[10] V. Krasnoproshin, D. Mazouka, “Data-Driven Method for High Level Rendering Pipeline Construction”, Neural Networks and Artificial Intelligence. Communications in Computer and Information Science, vol. 440, 2014, pp. 191–200.

Pattern Recognition and Information Processing (PRIP'2021) : Proceedings of the 15th International Conference, 21–24 Sept. 2021, Minsk, Belarus. – Minsk : UIIP NASB, 2021. – 246 p. – ISBN 978-985-7198-07-8.