The Evolution of the Graphics Pipeline

Arrian Chi
alienchi@ucla.edu

***Abstract*—The graphics pipeline is an algorithm used to convert 3 dimensional data from a 3D model into a 2D image. However, the construction of the algorithm depends on many graphics innovations made from the 1950s to the 2000s. This paper will discuss the evolution of the graphics pipeline, starting from the first algorithms to the modern graphics pipeline.**

I. Introduction

Computer graphics is a fundamental branch of computer science. Its existence has supported the growth of the gaming, film, aerospace, and medical industries. In modern times, we see its use in AI/ML for image/model generation and computer vision. From this, it is valid to say ”where there is virtual visualization, there is computer graphics in some way or another”.

The fundamental algorithm that allows the aforementioned industries to thrive is the graphics pipeline. To put simply, the graphics pipeline is an algorithm that converts 3D data from a 3D model into a 2D image. The individual steps of the pipeline emerged gradually as people invented technologies and discovered new algorithms. The pipeline serves as an artifact of the history from the dawn of computer graphics. In the following sections, I will review the graphics pipeline,uncover the primitive beginnings of the pipeline, discuss the innovations that exponentially increased the development of the pipeline, and report on the competition that paved the way for GPU rendering [21].

An Overview of the Graphics Pipeline

Input Data

The input data for the graphics pipeline is a 3D object, specifically, a set of vertices each defined by a vector-like type. Each vertex may also have other data associated with them such as color, normal vectors, texture coordinates, etc. The pipeline must also take in a set of indices of the vertices, which will be used in input assembly to form primitive triangles. Finally, uniform variables (such as camera matrices, light positions, time variables, etc.) are utilized by shaders to store data that is constant for all execution contexts.

Vertex Shader

The vertex shader is the first stage of the graphics pipeline. It takes in the vertex data and the uniform data from an application and outputs new vertex data per vertex. In modern times, the shader is a fully programmable program, so programmers may encode any logic they want as long as they obey the shading language’s syntax. However, there were times in history where this stage was fixed function, implemented purely in highly configurable hardware. The term shader applies to both cases, and we will see why programmability was desirable as time went on. Some common operations done on the vertex shader includes projection/clipping transformations, shading/lighting calculations, and normal computations [3].

Rasterization

This fixed-function stage first constructs triangles from the outputted vertices (input assembly). It then traverses them to determine which pixels (or fragments) on the screen are lit up by checking whether part of the pixel overlaps the triangle. This stage also interpolates the data (from the previous stage) among each pixel between the vertices of the triangles for calculations in the next stage[1].

Pixel Shader

Following rasterization, we run a pixel shader program on each pixel outputted. Like the vertex shader, this stage is also a fully programmable program that was historically fixed-function. The output of the pixel shader is the actual color of the pixel on the screen, along with the depth and, optionally, an opacity [1]. These outputs are then sent to a buffer, where merging occurs.

Merging

The merging stage takes in the fragment values and tests/blends them with current values on a framebuffer. Depending on the result of that operation, the stage either replaces the current value on the buffer with the incoming value, discards the incoming value, or creates and places a new value from the incoming and current values. After that, display hardware (the screen) will read the values from the buffer and exhibit that value (the color and intensity) in the corresponding pixel. When all the pixels on the screen glow, we see an image of the 3D model we started with[1].

I would like to clarify two subtleties for the reader. First, the pipeline may seem slow, but it is actually done multiple times per second. Optimally, values should be generated as fast as the refresh rate of the display hardware. If not, there will be visual artifacts such as lagging and screen tearing. This leads to the next subtlety that the pipeline is called a pipeline because each stage may be executed in parallel, decreasing the latency per vertex stream. This was not true when the pipeline started out, however.

II. The Beginnings of Computer Graphics

Whirlwind I (1951) and SAGE (1953)

Our story begins with military defense (as many computer innovations were). The United States was in the Cold War

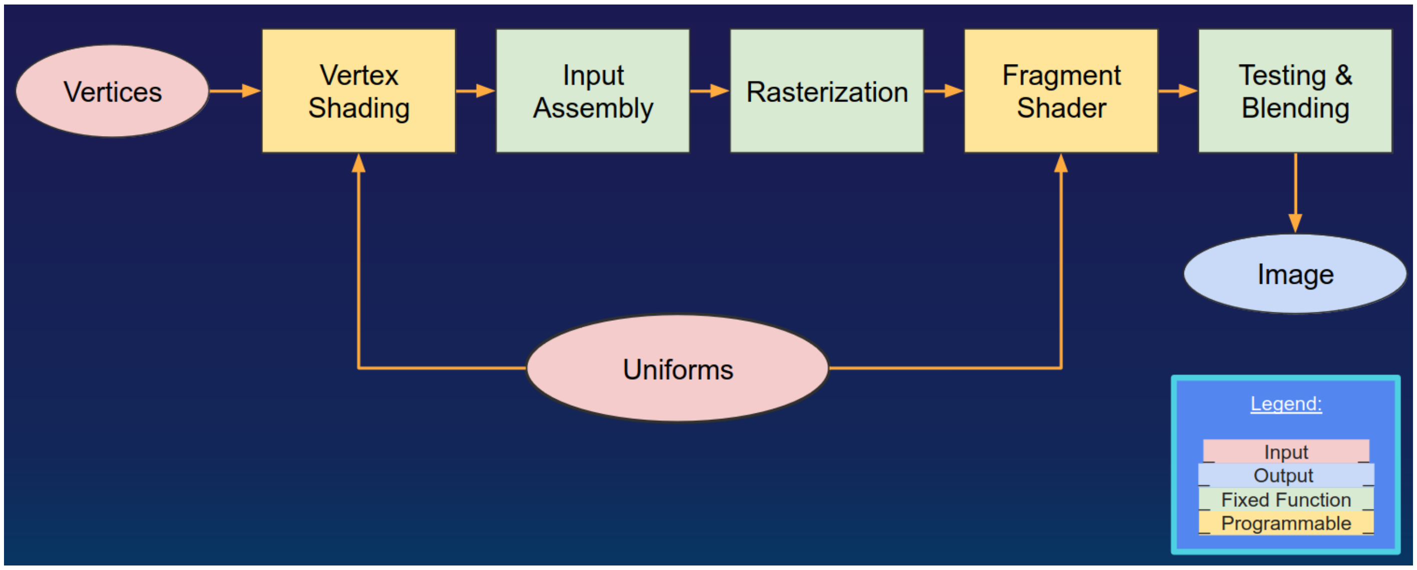

Fig. 1: The modern graphics rendering pipeline

against Russia and were anticipating an aerial attack from them. The US Air Force wanted to develop a mechanism that can alert stationed pilots about nearby enemy jets and their locations [9] [22].

Thus sparked the invention of the prototype system, the Whirlwind I, by MIT in 1951 and its successor, the Semi-Automatic Ground Environment (SAGE for short) by IBM, in 1953. The idea was to have an operator sit in front of a radar screen, which displayed points corresponding to nearby aircraft. They would then use a light pen to select the aircraft on the screen and in response, the system would display information about the aircraft [17].

This invention is regarded as the first-ever system in computer graphics because it was the first time a computer was connected to a CRT (cathode-ray tube) display. It is also worth mentioning that this is the first ever work in human computer-interaction because of its interactivity with a human operator [16].

Sketchpad (1963)

Following the trend of interactive systems, in 1963, Ivan Sutherland wrote his doctorate dissertation on Sketchpad, an interactive drawing system. His vision was to enable easier use of computers by providing an interactive interface for the user, specifically designers [26]. So his system included many features useful for them, which would prove to be seminal computer graphics ideas in the future.

First, complex figures in Sketchpad could be composed of simpler figures. To give an example, a polygon is interpreted as multiple lines and each line is interpreted as 2 connected points each with an x-y coordinate. This concept is eerily similar to the idea of object-oriented programming, where we modularize data into objects in order to create multiple instances of them[25]. Next, instead of manually drawing primitives, Sketchpad allowed users to specify constraints to draw. For instance, users only needed to specify the end points to draw lines. So if users ever needed to change the design, they only needed to change the endpoint instead of redrawing the entire line. Finally, the system could perform the simple 2D transformations: translations, scaling, and rotations. This enabled users to view and zoom into different parts of their drawings and edit them on the fly. This also implies that the system had screen-mapping capabilities to determine which parts of the drawing to display [26].

Although his work was incredibly seminal, it did not catch on with the public due to the high costs of CAD (computer-aided design) systems. Sutherland's work would only see light in the aerospace and automobile industries at the time. Additionally, there were also many limitations of his system. But before presenting that, the reader should understand the significance of vector-scan displays and raster-scan displays, which I will briefly explain below.

Vector scan Displays

The aforementioned systems used vector scan (a.k.a. random scan) displays with cathode ray tube technology. A display processor reads instructions from a refresh buffer in its display file. These instructions are used to direct an electron gun sitting in the back of the tube to fire a beam at a phosphorescent screen only towards parts of the screen where the picture is drawn. When the beam hits the screen, the screen emits light for a few seconds, so for the image to persist, the picture must be retraced. So the display processor reads from the refresh buffer multiple times per second to keep the image on the screen. To modify the image displayed, instructions are inserted, removed, or changed on the buffer [10].

At the time, vector scan displays were enough to get the job done on graphical systems. Their main advantage was that they created very smooth and straight lines on the screen. However, vector scan displays suffered from some major flaws. To begin,

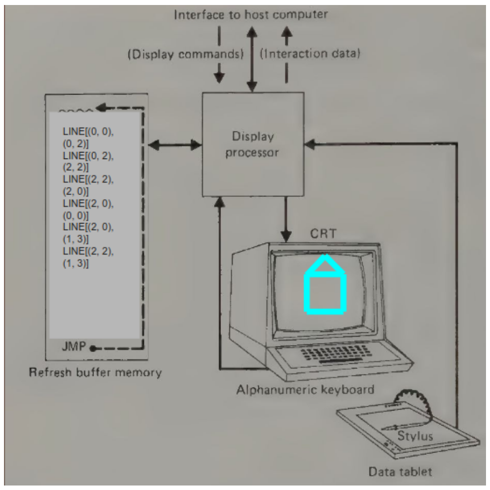

Fig. 2: A vector scan display. The beam only strikes the regions of the screen with showing the image.

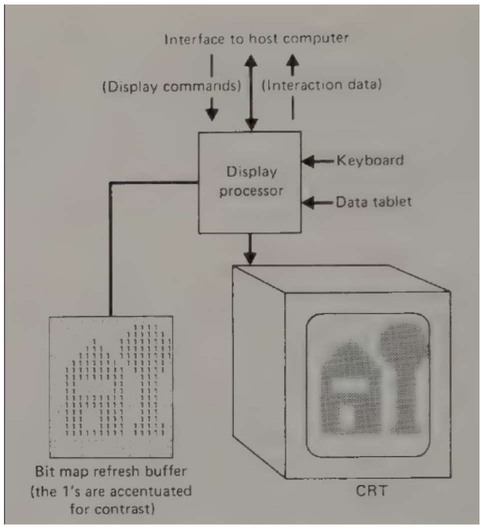
Interface to host computer
(Display commands)
(Interaction data)
Display processor
Keyboard
Data tablet
Bit map refresh buffer (the 1's are accentuated for contrast)
CRT

Fig. 3: A raster scan display. Notice that display technology is independent of display method.

the display flickered especially when many lines were drawn in a region. Although rotations and scaling was possible, they were expensive operations that took time when manipulating the buffer. Finally, there was no way to reliably create color. These flaws made it impossible for vector scan displays to do two ambitious tasks: 3 dimensional drawings and animation [26].

Raster-scan Displays

As the price of memory when down in the early 1970s, vector scan displays were replaced with raster scan displays. In raster scan displays, the screen is divided into an array of picture elements (a.k.a. pixels). The refresh buffer stored the intensity and color of every pixel on the screen. As the refresh buffer was updated, each pixel would be updated in a scan-line fashion, left-to-right and top-to-bottom. Again, the process was repeated multiple times per second [10].

When analyzing the advantages of vector scan displays and raster scan displays, it's clear why raster scan displays are superior. It provided color and was flicker-free, but it was also capable of displaying continuous tone images (images where the color transitions smoothly). Because of the cheap costs of memory, raster scan displays were much cheaper than their vector scan counterparts. This all came at a small sacrifice: lines were no longer truly smooth as a consequence of dividing the screen. The visual artifacts are called jaggies due to their jagged edges [17]. In the modern age, this problem is mitigated with anti-aliasing techniques, such as fast approximate anti-aliasing (FXAA) or multisample anti-aliasing (MSAA).

First "Pipeline"

An image of our first pipeline is shown in Figure 4. Input data is sent to the application program, transformed and sent to e a graphics system, where it sits in a refresh buffer. Regardless of display method or display technology, the display processor reads the refresh buffer multiple times per second and displays the output image on the display hardware.

The mechanism shares some similarities with our modern pipeline. For instance, transformations are done on the data before it is displayed. This logically makes sense; data needs to be converted to the right form before it may be processed in the next stage. Additionally, the refresh buffer is functionally the same as the framebuffer in our modern pipeline. The only difference here is that the framebuffer has more features and can deal with 3D data.

Despite the similarities, they have glaring differences. First, the graphics system acts as a pass-through between the input and application data. We will see that input gets modularized out of our pipeline as time goes on. Second, the mechanism is not a pipeline. The concept of pipelining was not invented until the 1980s, so our mechanism was done sequentially with no parallelization.

III. Growth and Development

Innovations at the University of Utah (1970s)

The 1970s saw many innovations in 3D computer graphics, particularly from the great minds at the University of Utah[12]. Their work brought to light the problems that needed to be solved by the graphics rendering pipeline. To mention some notable work:

- *Realistic Lighting Models:* Henri Gourand and Bui Tuong Phong invented Gourand shading and Phong shading models in 1971 and 1973, respectively. Gourand shading was an improvement upon flat shading (specifying a color per every flat surface) by interpolating the normals across the surface of the object between vertices [2]. Phong shading was an improvement upon Gourand shading, which missed highlights and had Mach banding, by computing lighting per-pixel, using the point each pixel projects onto the surface [4].
- *Texture Mapping:* Edwin Catmull invented texture mapping in 1974, a technique that maps images onto 3D objects to give more detail that is not inherent in the

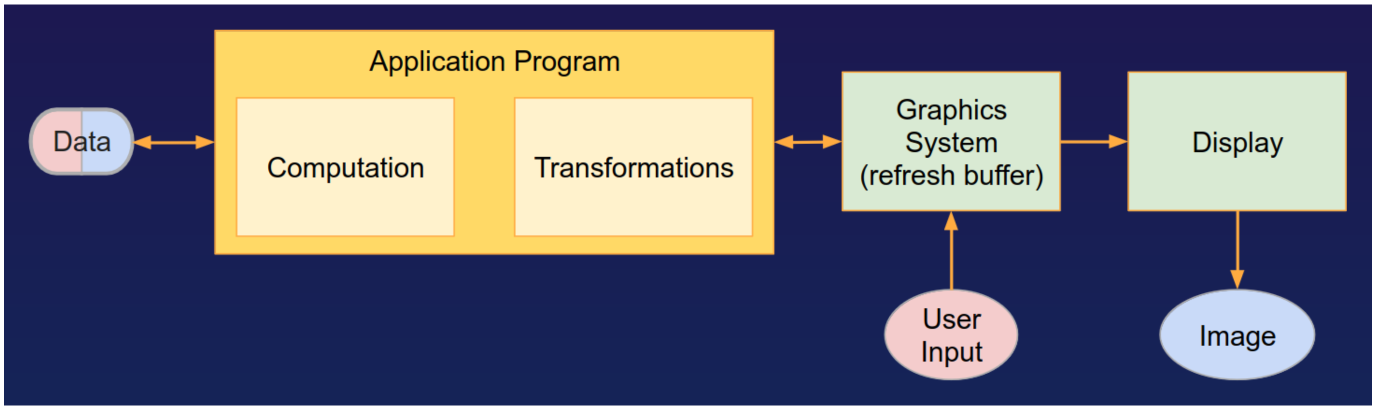

Fig. 4: The first "pipeline". User input is handled by the graphics system and sent to the application program.

geometry. The idea was to specify a coordinate between (0,0) and (1,1) representing the top left and bottom right corners of the images, respectively, at each vertex and map the regions between each vertex to the corresponding region on the image. This could be extended to a technique called bump mapping (invented by James Blinn in 1978) where we use an image to represent the normal at each vertex, giving the object a wrinkly texture [13].

- *Z-buffering:* Catmull and Wolfgang Straßer discovered z-buffering (a.k.a. depth testing) independently in 1974. It was a technique to determine what color should be shown on the image if objects occlude each other from view. Fragments looking to replace their correspondent on the framebuffer in the same render loop are successful, only if their z-value is smaller than the correspondent's (meaning that fragment of the surface is closer to the camera view than the other) [2].
- *Curves:* Martin Newell constructed one of the first models from Bézier curves, the Utah Teapot, in 1975. This enabled authoring objects with equations as opposed to explicitly specifying points, which is tedious.

As previously said, these stages brought about problems and solutions in computer graphics and thus changed our graphics rendering mechanism. It added new stages like shading and texturing and cemented others like buffering. With so many algorithms, programmers desired a standard to organize and keep implementation simple, leading to the next feat of the 1970s.

Standardization of Computer Graphics APIs (1980-1995)

It was around the mid-1970s that industry leaders and pioneers decided to come together and form standards. The overall motivation was portability. Up until now, hardware vendors each had their own set of graphics features. Programmers needed to learn how each machine operated before they could create applications on the machine. This was both tedious and wasteful because programmers who wanted their application to work on two machines from different vendors needed to expend time to study the workings of both machines. A standard would require the vendors to provide common functionality that the programmer would be familiar with and significantly reduce the time-to-market of graphics applications [6]. In practice, however, standards were seldom agreed upon when business incentives is involved.

Among the first were *ACM Core* in 1975, *Graphics Kernel System* (GKS) in 1977 and the *Programmer's Hierarchical Interactive Graphics System* (PHIGS) in 1979. These standards specified APIs (application programming interfaces) that did not encapsulate all that was capable at the time, and because of the rapid development of graphics algorithms, they did not survive (despite PHIGS's attempts to create a 3D version later on). However, the development of PHIGS pressured Silicon Graphics Inc. (SGI) to create OpenGL in 1992, an open source version of their proprietary graphics library IrisGL. OpenGL was originally only adopted by SGI's workstation machines, but as time went on, there was a widespread adoption among a majority of CAD workstations [6]. Microsoft would release their own API, DirectX, in 1992 for use on personal computers. The years following that would see direct competition between the two APIs, each trying to support a feature the other doesn't have. Silicon Graphics Inc. would appoint a committee to manage the standard before going bankrupt, while DirectX would continue supporting a wider range of hardware. When the committee was dissolved and management was given to the Khronos Group, DirectX had cemented itself in the world of graphics APIs. It took many revisions of OpenGL to get it up to par with DirectX.Nowadays, the main APIs used in the industry are Vulkan (from Khronos), OpenGL(used for legacy support and education), DirectX 18 (from Microsoft) and Metal (from Apple) [18].

Pixel Planes (1981) & The Geometry Engine (1982)

The 1980s also saw the advent of parallel processing. *Pixel Planes*, a project started at the University of North Carolina by Henry Fuchs and John Poulton, set out to create a VLSI system that would permit the use of computer graphics in fields outside of computing such as chemistry and biology. Their mantra was "One processor per pixel!" with goals of improving the rendering time. Their system was essentially parallelizing rasterization, pixel processing and buffering with

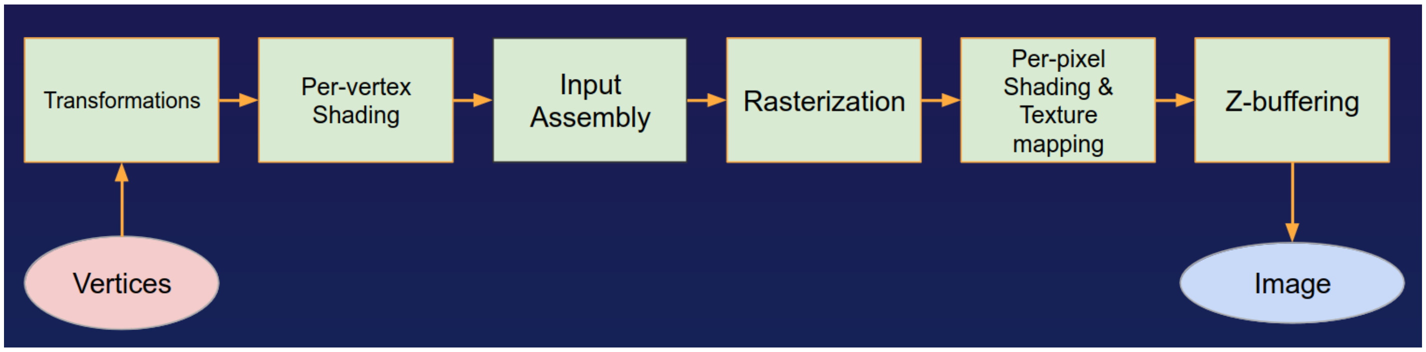

Fig. 5: The second pipeline. Notice how everything is fixed-function.

hardware because it was capable of identifying pixels that lie in the particular polygon, determine their visibility, and shade the pixel simultaneously. As a result, they iterated over the polygons of the scene rather than the pixels of the screen during runtime, similar to modern graphics cards. In the end, their system was estimated to process 15,000 to 30,000 polygons per second [11].

In 1981, Jim Clark founded Silicon Graphics Inc. with the intention of creating workstations optimized for CAD. Their systems were based on the *Geometry Engine*, a VLSI processor that optimized the geometric computations of the graphics pipeline. Specifically, it was a four component vector function unit that could perform floating point operations. When 12 of them are pipelined together, the system accelerated matrix multiplication, clipping, and scaling[7]. With respect to the pipeline, the chip proved that transformation can be taken out of the CPU workload and concurrently processed, like in modern GPUs. In fact, this design was inspirational for the development of transform & lighting hardware in the 1990s.

All in all, *Pixel Planes* and the *Geometry Engine* proved that graphics rendering is optimizable and parallelizable, making them precursor technologies of the modern GPU.

Second Pipeline

At this age, the pipeline(Figure 8) is starting to look more like our modern pipeline. The pipeline takes in vertices, transforms them, applies vertex shading, rasterizes the polygons created from the vertices, and applies pixel shading and z-buffering before being displayed on the screen. Additionally, some operations like rasterization and shading are parallelized, compounding the benefits of pipelining. In the next section, we will see chips become better and faster at rendering while also becoming more programmer friendly.

IV. The Birth of the Programmable GPU

Shade Trees (1984)

The idea for programmable graphics hardware started in 1984, when Robert Cook published his paper on *Shade Trees*. Cook noticed that shaders (which were implemented as fixed function hardware at the time) was becoming highly configurable. However, because the hardware was fixed, it forced all surfaces to use the same lighting algorithm. Additionally, hardware resources would be wasted if the user chose to use simple lighting models [8].

Programmability would solve these issues. Associating different surfaces with different programs can create a variety of shading effects in the same scene. Hardware resources are allocated on execution of the program, allowing both simple and complex programs to use only the hardware they need.

To implement this, Cook proposed organizing the shader logic into a syntax tree and evaluate it post order. The leaves of the tree would be input parameters, while the nodes are the operators. The root of the tree would output the final color. This idea implied the need for a shading language with basic mathematical operations [8].

Cook's work was first realized in Pixar's Renderman Interface in 1988. The interface implemented programmable shaders in software, so shaders were run on the CPU. This would set the path for programmable shaders on graphics cards, as we will see in the next section [1].

First GPUs (1996 - 2002)

It would be awhile before programmable shading is realized in hardware. Meanwhile, graphics companies were developing new 3D graphics accelerators every year, trying to best their competitor's specifications as well as their own. Some major chips that were developed over the course of this period includes:

- *3dfx Voodoo I (1996):* This chip was the first consumer 3d graphics accelerators. Its development led to the growth of PC gaming as an industry. [23]
- *3DLabs Oxygen GVX1 (1999):* This chip was the first to integrate programmable transform & lighting (T&L). 3DLabs coined the term GPU (Geometry Processing Unit). [18]
- *NVIDIA Geforce 256 (2000):* This chip was marketed as the world's first GPU (Graphics Processing Unit). It had integrated T&L, but no real programmability yet.[1]
- *ATI Radeon R100 (2000):* This chip was ATI's response to NVIDIA's Geforce 256. It performed significantly better.[20]
- *NVIDIA Geforce 3 (2002):* This chip marked the turning point of graphics controllers. It had true programmable

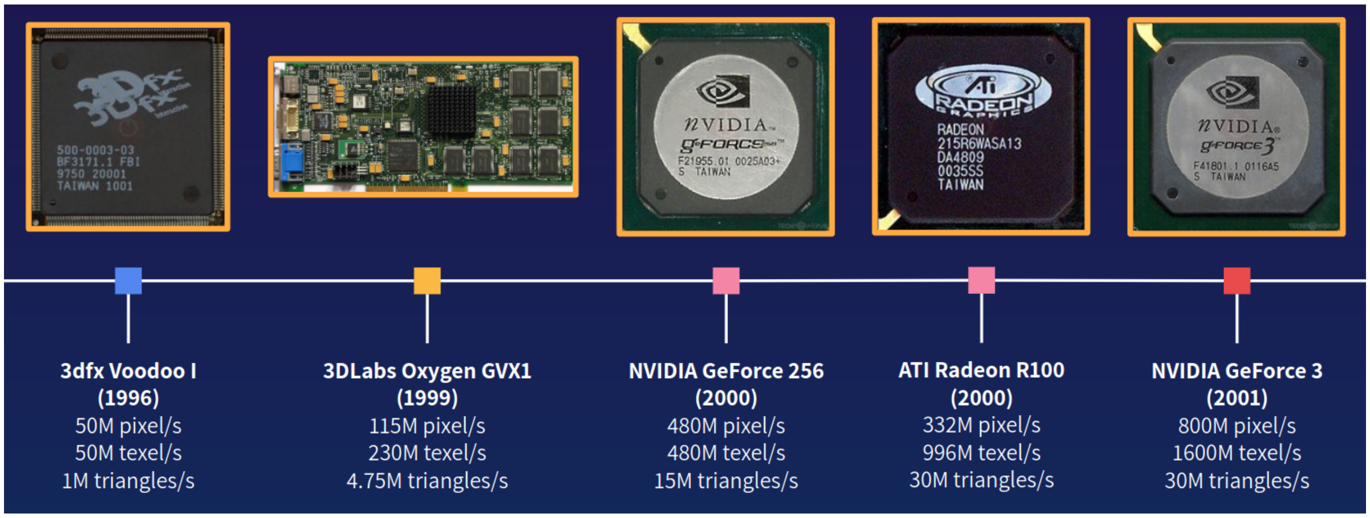

Fig. 6: A timeline of important GPUs developed between the late 1990s and early 2000s [27].

vertex shaders within its vertex engine and highly configurable fragment shaders via texture shaders. [15]

Clearly, the 1990s saw a trend towards higher computing power and more programmer customizability. NVIDIA's success made it a major supplier in the industry as other companies fizzled out of existence.

Trouble with APIs (2000-2006)

This isn't the end though. Although the hardware is programmable they may have features that APIs do not support. Each API had a different way to resolve this issue. For instance, Microsoft made sure to work closely with graphic card manufacturers like NVIDIA to make sure DirectX was consistent with their hardware. This was the case when the Geforce 3 came out with direct support for Shader Model 1 (the shading standard for DirectX 8). Another way to resolve this was to allow manufacturers to create their own extensions to the API as OpenGL did. Although this did allow software to run on that particular piece of hardware, this made the standard counterintuitive since extensions were proprietary.

As an example, the Geforce 3 supported pixel shading, as did ATI's Radeon 8500. To support the feature for both of those cards, Microsoft had made sure their API works for both chips, writing the appropriate drivers. However, to support OpenGL, NVIDIA and ATI each wrote their own extensions because of lack of support on OpenGL's end. For the OpenGL programmer, this meant that if their program used NVIDIA's extension, it would not work on the Radeon 8500 and vice versa. The programmer would be forced to write a new program for the Radeon 8500, which would be no different from when OpenGL didn't exist. This failure on OpenGL's part is one of the reasons why Windows is the de-facto standard OS for gaming (among others like the widespread use of Windows). OpenGL's problems were later resolved when the Khronos Group took over OpenGL's management in 2006 [5].

Unified Shading Model (2006)

Aside from API issues, 2006 marked an important decision to unify shaders. In Microsoft's standard, it was called Shader Model 4.

Previously, vertex engines and fragment engines had highly specific and different instruction sets. This was unideal for manufacturers because they had to develop different hardware for the two features. It was also inefficient because if a workload only used one engine, the hardware for the other engine is wasted [14].

The solution to this problem is to unify the shaders, that is, to make vertex and fragment shading hardware have the same capabilities and use the same instruction sets. This means that we may partition hardware appropriately for vertex and fragment shading based on the demands of each. It was easier to program (Nvidia created CUDA for this) and manufacture and best of all, it didn't affect legacy code. Non-unified APIs can run on unified shader architectures and unified APIs can run on non-unified architectures [19].

V. Conclusion

To end, in this paper, I have discussed how the graphics rendering pipeline is a historical artifact of the innovations in computer graphics. I have discussed how it changed over time and how each stage was formed by a major seminal achievement in computer graphics.

To review the pipeline again, we start with our vertex data, 2D or 3D, that we want to display. The data is processed by a shader program that transforms the vertices and calculates new data for lighting/shading algorithms. A rasterizer uses highly parallelized hardware to assemble the vertices into triangles and determine which pixels in screen space are lit up. We then run a shader on each of these pixels to compute the output of the lighting/shading algorithms. Finally, we place our result in a buffer that can be modified for accuracy before it is read by the display hardware and outputted.

From this, it is clear that the pipeline's development is analogous the innovations in computer graphics and will surely change as time goes on. In fact, we may have new pipelines to serve different purposes such as raytracing (where rays are simulated and bounced around the environment) and GPGPU computing (where the GPU is used as a highly parallelized processor [24]). So when the time comes, I hope someone will appreciate the history of the pipeline and document it as I have done in this paper.

REFERENCES

[1] T. Akenine-Mller, E. Haines, and N. Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018.

[2] T. Akenine-Moller and E. Haines. *Real-time rendering*. A K Peters, 2000.

[3] J. Blinn. *Jim Blinn's corner: a trip down the graphics pipeline*. Morgan Kaufmann, 1996.

[4] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, 1977.

[5] N. Bolas. Why do game developers prefer windows?, Mar 2013. https://softwareengineering.stackexchange.com/questions/60544/why-do-game-developers-prefer-windows/88055#88055.

[6] S. Carson, A. van Dam, D. Puk, and L. R. Henderson. The history of computer graphics standards development. *ACM SIGGRAPH Computer Graphics*, 32(1):34–38, 1998.

[7] J. H. Clark. The geometry engine: A vlsi geometry system for graphics. *ACM SIGGRAPH Computer Graphics*, 16(3):127–133, 1982.

[8] R. L. Cook. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, 1984.

[9] A. Exline. Computer graphics. *IEEE Potentials*, 9(2):43–45, Apr 1990.

[10] J. D. Foley and A. Van Dam. *Fundamentals of interactive computer graphics*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[11] H. Fuchs. Pixel-planes: a vlsi-oriented design for a raster graphics engine. *VLSI Design*, 2(3):20–28, 1981.

[12] J. Goodrich. How the computer graphics industry got started at the university of utah, Nov 2023. https://spectrum.ieee.org/history-of-computer-graphics-industry.

[13] J. F. Hughes. *Computer graphics: principles and practice*. Pearson Education, 2014.

[14] Khronos. History of programmability - opengl wiki. https://www.khronos.org/opengl/wiki/History_of_Programmability.

[15] E. Lindholm, M. J. Kilgard, and H. Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158, 2001.

[16] C. Machover. A brief, personal history of computer graphics. *Computer*, 11(11):38–45, Nov 1978.

[17] C. Machover. Four decades of computer graphics. *IEEE Computer Graphics and Applications*, 14(6):14–19, Nov 1994.

[18] J. Peddie. *The History of the GPU-Eras and Environment*. Springer Nature, 2023.

[19] J. Peddie. *The History of the GPU-New Developments*. Springer Nature, 2023.

[20] J. Peddie. *The History of the GPU-Steps to Invention*. Springer Nature, 2023.

[21] J. Plutte. When a bit became a pixel: The history of computer graphics. https://www.computerhistory.org/revolution/computer-graphics-music-and-art/15/intro/2398.

[22] D. Sevo. History of computer graphics, 2005. https://www.danielsevo.com/hocg/hocg_1970.htm.

[23] G. Singer. The history of the modern graphics processor, Dec 2023. https://www.techspot.com/article/650-history-of-the-gpu/.

[24] S. Soller. Gpgpu origins and gpu hardware architecture. 2011.

[25] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. Technical Report UCAM-CL-TR-574, University of Cambridge, Computer Laboratory, Sept. 2003.

[26] A. Van Dam. Computer graphics comes of age: an interview with andries van dam. *Communications of the ACM*, 27(7):638–648, 1984.

[27] Vlask. Vga legacy mkii, Mar 2015. https://www.vgamuseum.info/.