SUNG-EUI YOON, KAIST

RENDERING

FREELY AVAILABLE ON THE INTERNET

http://sglab.kaist.ac.kr/~sungeui/render

First printing, July 2018

Part I

Rasterization

Rasterization is one of most popular rendering techniques developed for computer graphics. It simply projects triangles in a scene into a viewing space and color pixels overlapped with those triangles. This approach is very simple and thus can be implemented efficiently in specialized hardwares. Especially, many graphics hardware and GPUs support this rasterization scheme.

It, however, does not simulate the natural interaction between light and materials. Simply speaking, in reality, objects are not projected into our eyes! Due to this issue, rasterization schemes have fundamental drawbacks of simulating various rendering effects such as shadows, transparency, and so on. Nonetheless, thanks to its fast performance, many techniques and fixes have been proposed to improve its rendering quality.

In this part, we discuss the fundamental engine of rasterization, which is developed in many graphics library such as OpenGL and DiretX accelerated by GPUs. In other parts , we study global illumination that physically simulates interactions between lights and materials.

1.4 Related Materials

Many useful resources for rasterization techniques are available. Some of them are listed here:

- OpenGL Programming Guide. OpenGL is one of very popular computer graphics library that can be used in a wide variety of computing platform including Windows, Linux, and mobile OS. OpenGL provides various useful low-level graphics APIs, and they are well explained in this book and in its reference book. Early version of these books are available on free at internet. We also explain some of OpenGL APIs and their concepts, when we explain concepts of rasterization for delivering concrete examples.
- Real-time rendering [9] and its resource. This book covers a vast amount of topics that are related to rasterization and real-time rendering techniques. Its resource cite [10] has many useful web pages and links.
- OpenGL tutorials. Many OpenGL tutorials exist at Web. Some of them are based on the legacy OpenGL, but `http://www.opengl-tutorial.org/` discusses useful tutorials based on a recent OpenGL (ver. 3.3 and later).

[9] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., 2008

[10] `http://www.realtimerendering.com/`

2
Rendering Pipeline

Rendering triangles for scenes requires an excessive amount of computation time, since there could be many triangles representing scenes, and each triangle can map to hundreds of pixels in the screen space. As a result, carefully designed steps, known as rendering pipeline, has been proposed.

2.1 Classic Rendering Pipeline

Let us first discuss the classic rendering pipeline, before studying a modern, but complex one.

Fig. 7.1 shows an example of a classic rendering pipeline running on a GPU. An graphics application runs on a CPU in general and sends geometry of the scene and a camera setting that its user wants to see to a GPU by using a graphics library such as OpenGL. The rendering pipeline implemented in a GPU processes such requests and computes an output image displayed in a screen.

In general, the rendering pipeline consists of many steps for drawing an image from the user's camera position and orientation in an efficient manner. At a high level, they usually breaks into vertex processing and pixel processing units. The vertex processing step transforms input geometry into ones mapped in the screen space. Those ones are converted into pixels with appropriate colors by the pixel processing step, and this step is commonly known as the rasterization step.

Historically, these steps take a high computation time and thus are implemented in a chip in a hard-wired manner. These steps, therefore, are rather fixed functions and invoked through graphics APIs. As we have more processing power and developers request more flexibility on programming, the GPU implementing these steps become more general like CPU and can run various graphics programs such as OpenGL shaders.

While more accurate rendering techniques (e.g., global illumi-

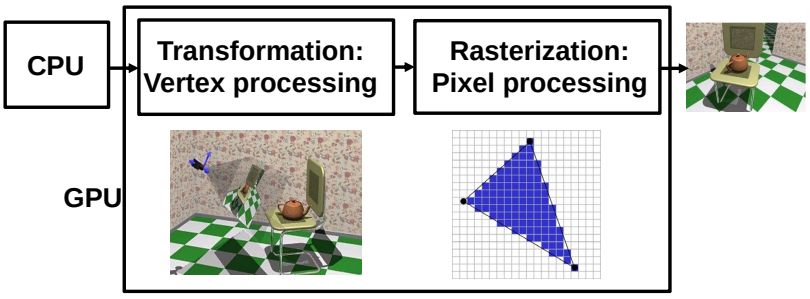

Figure 2.1: This shows a schematic diagram of classic rendering pipeline consisting only two steps: vertex and pixel processing steps.

nation) have been proposed with high performance, rasterization scheme is one of the most efficient rendering algorithms specializing on local illumination, which considers the light energy transfer between a surface and a light source. We therefore study this scheme in a detailed manner in Chapter 3 and 7.

2.2 *Modern Rendering Pipeline*

Fig. 7.2 shows a schematic view on a modern rendering pipeline adopted in OpenGL 3.0. While this differs a lot from the classical one, it shares both vertex and pixel (e.g., fragment) processing steps.

- **Vertex specification.** Vertices and triangles are defined and passed to the following step.
- **Vertex processing.** Each vertex is processed by a vertex shader, a program working on each vertex. It performs various modeling transformation, viewing, and projection transformations.
- **Vertex post-processing.** It performs various basic operations after the vertex processing step and serves as a setup stage for the following steps such as rasterization. It includes clipping (Sec. 6.4), homogeneous divide (Sec. 4.2.1), and viewport transformation (Sec. 3.1).
- **Primitive assembly.** Face culling is performed in this step.
- **Rasterization.** This step converts a triangle represented by vertices into a number of fragments.
- **Fragment shader.** It also processes each fragment generated by the prior rasterization step.

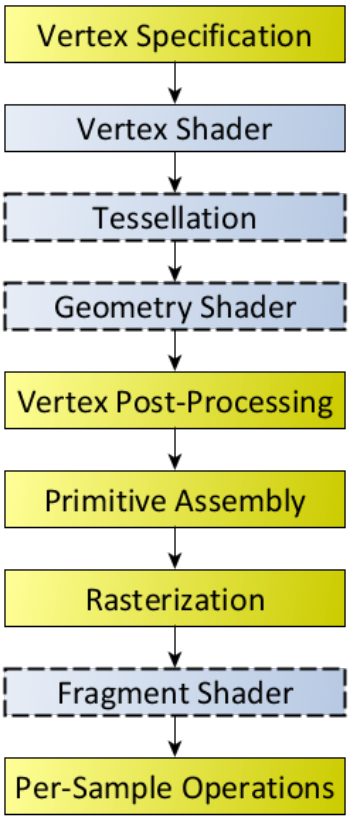

Figure 2.2: This shows a rendering pipeline adopted in OpenGL 3.0. This image is excerpted from the OpenGL homepage.

2.3 OpenGL and Other Tools

The rendering pipeline has been implemented and accelerated in GPUs. To enjoy such hardware acceleration, we use OpenGL and DirectX. OpenGL is more widely available in different operation systems and devices, since DirectX depends on Windows OS. Most concepts and techniques that are covered in this part are available at such APIs. Nonetheless, it is useful to know what other tools related to graphics are available and their goals. Fig. 2.3 shows other tools and languages that can utilize various features of GPU other than the rasterization.

Recently, Vulkan was introduced for achieving even higher performance on mobile phones [1] that have lower performance than PCs. For achieving its goal, Vulkan allows users to various low-level APIs with low overheads and multi-tasking. Nonetheless, it comes with certain costs such as higher programming burdens to users.

[1] G. Sellers and J.M. Kessenich. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Addison Wesley, 2016

While these APIs provide the full features of the rendering pipeline, they are rather low-level APIs. When we want to develop high-level applications such as a game, we need to utilize a more powerful set of tools and SWs. This is a gap that modern game and rendering engines such as Unity try to fill in. Additionally, in graphics applications (e.g., games and movies), content creation is one of main tasks, and many modeling and animation tools are available.

Initially, GPU is designed as a specialized hardware to accelerate the rendering process, which is captured in the rendering pipeline. However, as the performance of GPU is getting higher and various demands on programmability on the rendering pipeline arise. As a result, parts of vertex and fragment stages can be programmable through a dedicated language, i.e., GLSL and HLSL.

While these shading languages are designed to effectively utilize functions of GPUs for graphics applications, non-traditional needs on using GPUs for non-graphics applications keep increasing, thanks to its higher performance on streaming tasks than CPUs. To accommodate such demands, a general purpose language for utilizing GPUs has been proposed, and CUDA and OpenCL are two examples.

2.3.1 Common Questions

What if we have new input devices (e.g., joystick, or multiple input devices used in PlayStation or XBox)? How can we handle those devices in OpenGL programs? OpenGL does not have any functionality to support those various input devices. GLUT library supports some of basic input devices such as keyboard and mouses. For other devices, you need to use other external libraries that support those

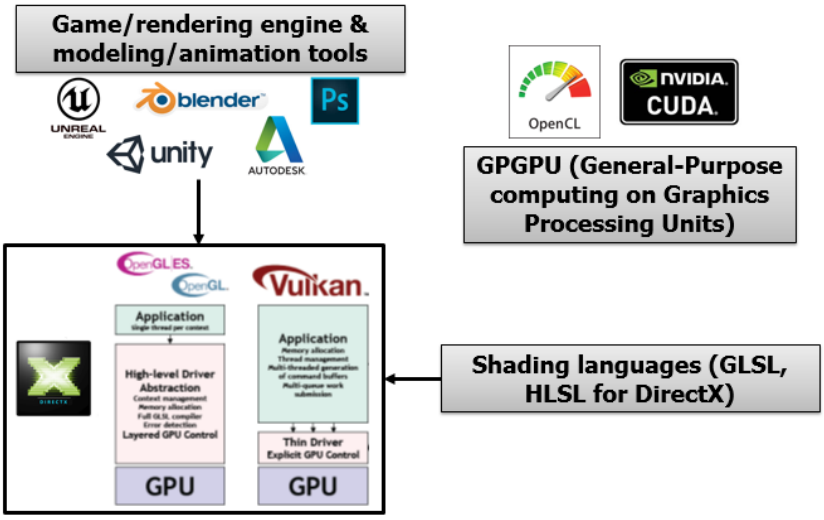

Figure 2.3: This figure shows other APIs, SWs, and languages that are related to OpenGL and computer graphics. In this book, we mainly discuss the core rendering pipeline that rasterizes input models. Nonetheless, many game and rendering engines (e.g., Unity) are commonly used as convenient, high-level tools. Also, shading languages are used in recent OpenGL verions, to add various details on rendering results. Additionally, general purpose computing languages for GPU (e.g., CUDA) are also used for implementing arbitrary programs on GPUs. Images are excerpted from the Vulkan overview and Google images.

devices.

In what cases, is OpenGL used rather than DirectX? OpenGL is cross-platform graphics API, while DirectX is proprietary library for Windows. Because of the openness of OpenGL, it, more specifically, OpenGL ES, is widely used for many embedded systems including mobile phones.

In what portions of my OpenGL program are executed in CPU and GPU? In a typical OpenGL program, rendering parts (e.g., portions started with glBegin and ended with glEnd) are performed in GPU, graphics hardware, if your computer is equipped with such GPU. All the control parts, e.g., calling OpenGL functions and handling events, are performed in CPU. In other words, various functionality inside OpenGL APIs are commonly performed in GPU, while all the other parts are performed in CPU.