Chapter 3

Graphics Programming

We find interactive computer graphics in a wide variety of computational platforms, ranging from low-power mobile devices to high-powered gaming systems to clouds of remote graphics servers. Graphics programming relies on graphics libraries that simplify programming and make graphics programs more portable among these different platforms. Interactive graphics often requires high level concepts, such as the simulation of physical motion, but relies on low level operations, such as setting the color of an individual pixel. Graphics libraries provide common low level processing routines so that the programmer can focus on the higher level behavior of a graphics application.

A variety of graphics libraries are available when programming graphics applications on a personal computer, and the two most popular ones are OpenGL and Direct3D. OpenGL grew out of the IrisGL graphics library used for Silicon Graphics workstations in the 90's, and is now an established standard controlled by a consortium of companies through an architecture review board (ARB), and is available across almost all computer platforms and operating systems. Direct3D is the 3-D graphics component of Microsoft's DirectX library, and is used primarily for videogames for both Windows PC's and the XBox game console. Direct3D can focus on efficient implementation on Windows platforms with quick adoption of recent advances, whereas OpenGL provides more general purpose 3-D graphics support across a broader collection of platforms.

In addition to programming the CPU, these libraries also include support for directly programming the graphics processor (GPU). GPU programs are based on specialized instructions designed for graphics and parallel processing, and so rely on specialized langauges for their programming. OpenGL uses a language called GLSL (Graphics Library Shading Language) for GPU programming, whereas Direct3D uses a similar language called HLSL (High Level Shading Language).

The GPU can also be programmed independently from a graphics library, and can be used to compute tasks other than graphics, using a process which is termed GPGPU (General Purpose GPU) programming.

Modern GPU's have a so-called "compute" mode that shuts down their specialized graphics circuitry and turns the GPU into a general purpose parallel processor. Languages have been designed specifically for this purpose, including NVIDIA's CUDA language and the multiple-company OpenCL language and library.

In addition to PC's, we now see powerful graphics capabilities in mobile devices, such as cellphones and tablets. While such devices have vendor-specific graphics libraries, many of them are also supported by a cross-platform OpenGL/ES (Embedded System) library, which is a scaled back version of OpenGL with some fundamental differences.

We can also program graphics for the web. The HTML5 standard includes a "Canvas" tag that enables sophisticated 3-D graphics. The WebGL standard, supported by most browsers, is essentially an implementation of the OpenGL/ES standard for use by javascript on the browser.

3.1 GPU Programming

The graphics processing unit (GPU) is a special purpose computational engine designed to accelerate the real-time rendering of three-dimensional meshed shapes.

The modern GPU is a high-performance parallel processor. We call a single GPU activity, such as transforming a vertex or shading a pixel, a *thread.* A thread is a sequence of operations applied to an individual data element (e.g. a vertex or a pixel). The GPU collects multiple copies of these threads, each operating on different data, into a *warp.* On modern GPU at the time of this writing, a warp consists of 32 threads. The GPU processes a warp in SIMD (single instruction multiple data) by running the same instruction in each thread simultaneously, with each of these threads modifying its own data.

Hence the threads in a warp run in sync, in lock step with each other. This provides great speed benefits especially for graphics, since we tend to want to run the exact same process on all of our vertices and pixels. However, this SIMD approach can become problematic, for example when an if-then-else instruction is processed. The condition of an if-then-else usually depends on data, and the thread data might differ from one another, so some threads in a warp might need to execute the "then" clause while others would execute the "else" clause. Since all threads in the warp always execute the same instructions, the GPU must execute both the "then" and "else" clauses on all of the threads, and throw away the result of the other clause for each of the threads. This creates a *control flow divergence* that can be costly. In the worst cases, each of a warp's threads needs to execute a unique sequence of instructions, effectively causing the GPU to *serialize* the threads.

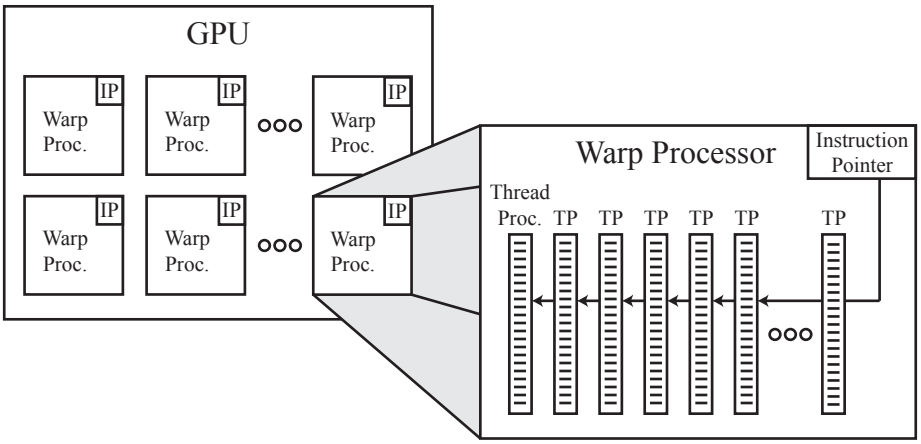

Figure 3.1: A GPU consists of several warp processors, each with its own instruction pointer. A warp processor contains several thread processors that share the same instruction pointer. The thread processors run identical programs in parallel in lock step with each other within the warp processor. The warp processors, however, run independently of each other.

Ordinarily, a warp could execute the same instruction on its, say 32, threads in a single step, but if all of the threads have diverged (in the worst case) then the GPU must execute each thread's instruction on all 32 thread processors, throwing away the result for all but the one thread that needed the instruction. Then 32 steps are needed to execute the threads' next instruction and the parallel GPU is running no faster than a serial processor.

The GPU includes useful instructions that replace divergent control flow with data dependent operations. For example the common operation

if $x < 0$ **then**
    $y = 0$;
else if $x > 1$ **then**
    $y = 1$;
else
    $y = x$;

whose control flow divergence could require three steps on all of a warp's threads, is replaced by the single operation

$y =$`clamp(`$x, 0, 1$`);`

executed as a single step simultaneously on all of the warp's threads.

In addition to data parallelism, multiple threads run concurrently. Each thread processor runs a number of threads (usually four) concurrently. Although the thread processor can only run an instruction from one thread

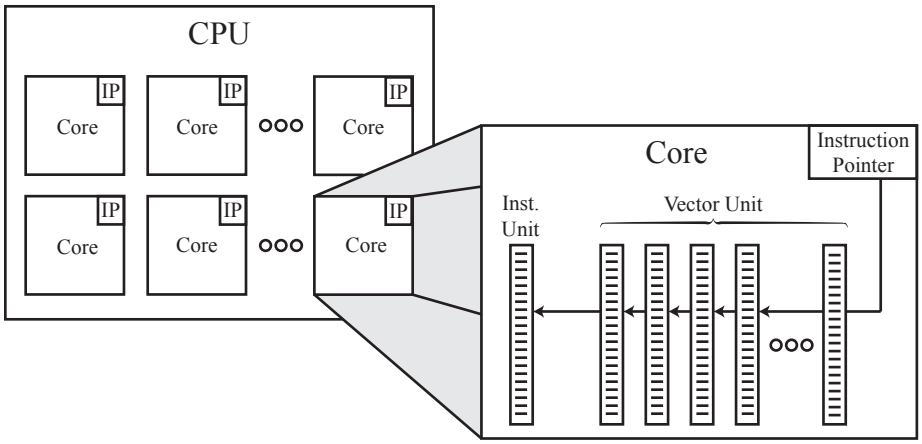

Figure 3.2: Modern multi-core CPU architecture can process similarly to the GPU, with several independent cores each containing a vector unit that applies the same instruction to multiple data streams. In addition to the vector unit, each CPU core has more complex single instruction processing and caching.

at a time, some instructions such as retrieving a value from a memory location require several clock cycles to wait for the memory to deliver the value. The thread processor compensates for this lost time due to memory latency by executing instructions from other threads. Hence, a thread processor alternates between four threads, so each thread runs at one-quarter the speed, but four threads are running concurrently, yielding full utilization of processing power while overcoming delays due to memory latency.

CPU Programming

While there are many differences between CPU design and GPU design, e.g. cache, speculation, etc., there are some similarities. A multi-core CPU consists of several independent processing cores. Each of these processing cores has an instruction unit designed to process a serial stream of instruction, but also has a vector unit that can execute the same instruction on multiple data streams. At this level, the CPU resembles GPU organization, with each core playing the role of a warp processor, and the vector unit processing the equivalent of a warp's simultaneous threads. This comparison is sometimes confused by the language used by the manufacturers. For example GPU manufacturers define a core as a thread processor whereas a CPU manufacturer's "core" is what GPU manufacturers call a warp processor, aka a "streaming multiprocessor" (SM).

When programming the CPU for high-performance applications, especially graphics, one should be sure to take advantage of each CPU core's

vector units, especially when comparing performance to GPU implementations.

3.2 Pipelines

Interactive computer graphics relies on speed, and when possible we can achieve speed through parallelism. When data can be organized into a regular structure, it lends itself well to parallelism. Graphics data is often quite regular. We have already seen that raster images are rectilinear arrays of data, and we conclude this chapter by showing how the triangles we process can also be organized into streams. Hence interactive graphics heavily utilizes parallelism to push the envelop of displaying as much detail as possible as fast as possible.

The processing of shape triangles into image pixels is further accelerated by separating the task into individual stages of a pipeline. This task decomposition isolates the individual steps of the rendering process. This allows each stage to be implemented more efficiently, with better advantage of locality in instruction and data caches. Graphics data can sometime increase or decrease, so pipeline stages can wait until they have a full buffer to process the data. Finally, parallel processors often compete for resources, such as specific memory locations, and distributing processors across different stages of the pipeline allows them to run without getting in each other's way.

There are many graphics pipelines for many purposes. Here we will focus on a pipeline, called the vertex pipeline, that takes a triangle mesh from a 3-D shape model and converts it into 2-D screen triangles for rasterization into pixels. After rasterization, a pixel pipeline processes each rasterized pixel for eventual display in a raster image.

3.3 The Vertex Pipeline

Scenes in computer graphics are often composed of different shapes, each modeled separately as a triangle mesh. The problem is that each of these shape models is rendered in its own coordinate system. If we have a teapot on a table, the teapot may have its origin at its base, and extend a unit in each direction. The table may have its origin at the bottom of one of its leg also extending a unit in each direction. Simply concatenating the mesh descriptions yeilds a scene with a large teapot absorbing the leg and corner of a small table. Hence we need to accept shape elements in their own "modeling coordinate system" and express them in a single consistent "scene coordinate system" (also called the "world" coordinate system). The first stage of the vertex pipeline converts the coordinates

of the vertices of individual model elements from their element's modeling coordinate system into the scene's single coordinate system.

Once we have a scene filled with triangle mesh shapes whose vertices are based on the same consistent coordinate system, we will want to look at it. We first find a position and orientation for the viewer in the scene. It will make rendering the mesh triangles much simpler if we can express their vertices in a viewer-centered coordinate system, where the viewer is at the origin, looking along the z-axis. Hence the second stage of our pipeline converts vertices from scene coordinates into these viewing coordinates.

The human visual system processes the 2-D image on the retina into a 3-D reconstruction of the scene. This processing depends on a variety of depth cues, which are effects in 2-D that aid in the understanding of the 3-D context. One of these cues is perspective, which indicates depth by making these near the viewer larger and things farther away smaller. Hence we have a stage in our pipeline that implements perspective distortion (or other alternative depth distortions).

Human viewers have a limited field of view, the visual angle defining what appears in a view and peripheral vision horizontally and vertically without moving ones head. Since we are now in a viewer centered coordinate system we can figure out which triangles our outside the display region, or intersect its boundaries. Triangles outside the display region can be eliminated, and those intersecting it can be trimmed, in a pipeline step called clipping.

Clipping allows expensive operations to be avoided for triangle vertices that aren't visible. One of these steps is an expensive division that is the last step for perspective distortion, so a stage of the pipeline can now perform that division now.

At this point, the triangle vertices have been positioned where they need to be to provide a perspective view of the scene. We can call their position in the projection plane their canvas coordinates, which is also (confusingly) called window coordinates. A final stage of the vertex pipeline converts canvas coordinates into pixel coordinates.

3.4 Graphics Libraries

We use graphics libraries to avoid rewriting graphics programs for every new display adapter that comes around. Using a graphics library like OpenGL, we can write a graphics program that runs e.g. in Windows, Linux or on an Apple, and will work with e.g. an NVIDIA, AMD or Intel graphics display adaptor.

OpenGL is an open specification for a graphics library. It is open in that the standard is controlled by an architecture review board (ARB) of major graphics companies. It grew out of Silicon Graphics GL graphics

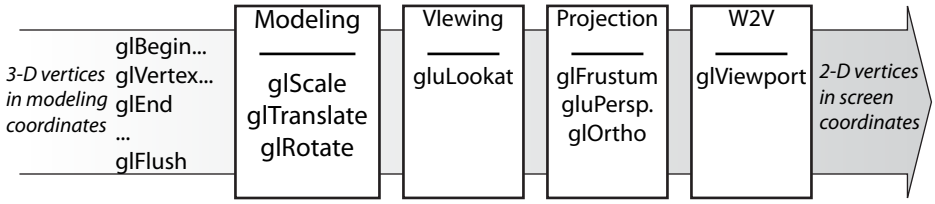

Figure 3.4: OpenGL and other libraries are designed to control the state of the graphics pipeline. As geometry streams through the pipeline, the control settings of each stage can be set by OpenGL.

library used for its graphics workstation when it was at the forefront in the 1980's and 90's.

OpenGL is designed to work across different operating systems and graphics controllers. It relies on an operating system that defines a uniform interface to a device driver that communicates with the specific graphics controller.

OpenGL describes a framework of several libraries working in conjunction with each other. OpenGL consists of two libraries: GL and GLU. The GL library is a core set of routines that need to be implemented for OpenGL to work on a given graphics controller. The GLU library is a set of additional helper routines that are all implemented in terms of GL library calls. Hence when writing a new OpenGL device driver, one needs only implement the GL routines, but when writing a graphics application, one is free to use the broader GLU library.

OpenGL

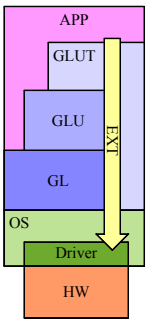

Figure 3.3: OpenGL layers.

While OpenGL supports many graphics operations, it does not include the ability to open a window and define a drawing canvas. OpenGL works with a variety of different applications, but depends on the construction of a drawing canvas, called a graphics context. For simple prototypes, a library called the Graphics Library Utility Toolkit (GLUT) provides an OS independent method for opening a window and accessing other OS services.

The OpenGL library is designed to control a graphics pipelines. If you think of the graphics pipeline as a big conveyor belt through various machine components, then the switch settings for each of these components is called the "state" of the machine. OpenGL is designed to manipulate the state of this machine and to efficiently feed data into it.