2.1 Architecture

In the physical world, the pipeline concept manifests itself in many different forms, from factory assembly lines to fast food kitchens. It also applies to graphics rendering. A pipeline consists of several stages [715], each of which performs part of a larger task.

The pipeline stages execute in parallel, with each stage dependent upon the result of the previous stage. Ideally, a nonpipelined system that is then divided into n pipelined stages could give a speedup of a factor of n. This increase in performance is the main reason to use pipelining. For example, a large number of sandwiches can be prepared quickly by a series of people—one preparing the bread, another adding meat, another adding toppings. Each passes the result to the next person in line and immediately starts work on the next sandwich. If each person takes twenty seconds to perform their task, a maximum rate of one sandwich every twenty seconds, three a minute, is possible. The pipeline stages execute in parallel, but they are stalled until the slowest stage has finished its task. For example, say the meat addition stage becomes more involved, taking thirty seconds. Now the best rate that can be achieved is two sandwiches a minute. For this particular pipeline, the meat stage is the *bottleneck*, since it determines the speed of the entire production. The toppings stage is said to be *starved* (and the customer, too) during the time it waits for the meat stage to be done.

This kind of pipeline construction is also found in the context of real-time computer graphics. A coarse division of the real-time rendering pipeline into four main stages —*application*, *geometry processing*, *rasterization*, and *pixel processing*—is shown in Figure 2.2. This structure is the core—the engine of the rendering pipeline—which is used in real-time computer graphics applications and is thus an essential base for discussion in subsequent chapters. Each of these stages is usually a pipeline in itself, which means that it consists of several substages. We differentiate between the functional stages shown here and the structure of their implementation. A functional stage has a certain task to perform but does not specify the way that task is executed in the pipeline. A given implementation may combine two functional stages into one unit or execute using programmable cores, while it divides another, more time-consuming, functional stage into several hardware units.

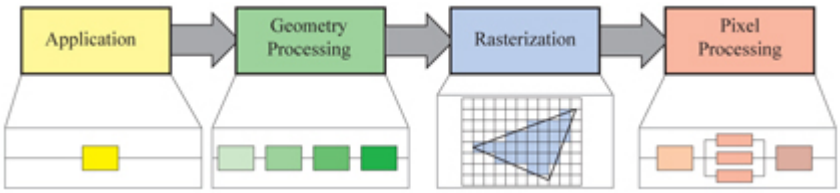

Figure 2.2 The basic construction of the rendering pipeline, consisting of four stages: application, geometry processing, rasterization, and pixel processing. Each of these stages may be a pipeline in itself, as illustrated below the geometry processing stage, or a stage may be (partly) parallelized, as shown below the pixel processing stage. In this illustration, the application stage is a single process, but this stage could also be pipelined or parallelized. Note that rasterization finds the pixels inside a primitive, e.g., a triangle.

The rendering speed may be expressed in *frames per second* (FPS), that is, the number of images rendered per second. It can also be represented using *Hertz* (Hz), which is simply the notation for $1/seconds$, i.e., the frequency of update. It is also common to just state the time, in milliseconds (ms), that it takes to render an image. The time to generate an image usually varies, depending on the complexity of the computations performed during each frame. Frames per second is used to express either the rate for a particular frame, or the average performance over some duration of use. Hertz is used for hardware, such as a display, which is set to a fixed rate.

As the name implies, the *application* stage is driven by the application and is therefore typically implemented in software running on general-purpose CPUs. These CPUs commonly include multiple cores that are capable of processing multiple *threads of execution* in parallel. This enables the CPUs to efficiently run the large variety of tasks that are the responsibility of the application stage. Some of the tasks traditionally performed on the CPU include collision detection, global acceleration algorithms, animation, physics simulation, and many others, depending on the type of application. The next main stage is *geometry processing*, which deals with transforms, projections, and all other types of geometry handling. This stage computes what is to be drawn, how it should be drawn, and where it should be drawn. The geometry stage is typically performed on a graphics processing unit (GPU) that contains many programmable cores as well as fixed-operation hardware. The *rasterization* stage typically takes as input three vertices, forming a triangle, and finds all pixels that are considered inside that triangle, then forwards these to the next stage. Finally, the *pixel processing* stage executes a program per pixel to determine its color and may perform depth testing to see whether it is visible or not. It may also perform per-pixel operations such as blending the newly computed color with a previous color. The rasterization and pixel processing stages are also processed entirely on the GPU. All these stages and their internal pipelines will be discussed in the next four sections. More details on how the GPU processes these stages are given in Chapter 3.