2 A Rendering Architecture

2.1 Introduction

Writing a production renderer is a large software development project that requires a balance between the creativity of designing algorithms and the discipline of writing robust code. A renderer often starts out as a toy program to demonstrate a new optical effect. Over time, the software evolves into a large, poorly modularized system or gets locked into a particular architecture that is difficult to extend. Starting with a sound design will save effort in the long run.

This chapter presents an object-oriented design for a production renderer with a micropolygon architecture. It should be considered a skeleton upon which the content of the subsequent chapters may be hung. Our sample renderer will adhere to the RenderMan standard (Pixar, 2000) and example header files will be written in C++. We assume that the reader is familiar with both.

The first section will chart the course from the RenderMan Application Programmers' Interface (API) to the front door of the rendering engine. This part is independent from the choice of rendering algorithms, and is applicable to anything from a real-time RIB previewer to a global illumination renderer.

Next, micropolygon architectures and the Reyes pipeline will be discussed from a theoretical point of view. In preference to immediately presenting a Reyes implementation, a supporting cast of classes will be introduced that take on the burden of many steps. Each has been designed to create interfaces at logical points and promote modularity. An overview of the rendering architecture is shown in Figure 2.1.

In the final section, the Reyes framework will be coded in terms of these supporting classes. A ray-tracing framework will then be shown, and the two will ultimately be combined to create a hybrid production renderer. These final steps are simplified by the versatility of the object-oriented design presented earlier in the chapter.

2.2 The Hierarchical Graphics State

Just below the RenderMan API sits the *state* subsystem. It manages the hierarchical graphics state using five stack data structures:

- Mode
- Option

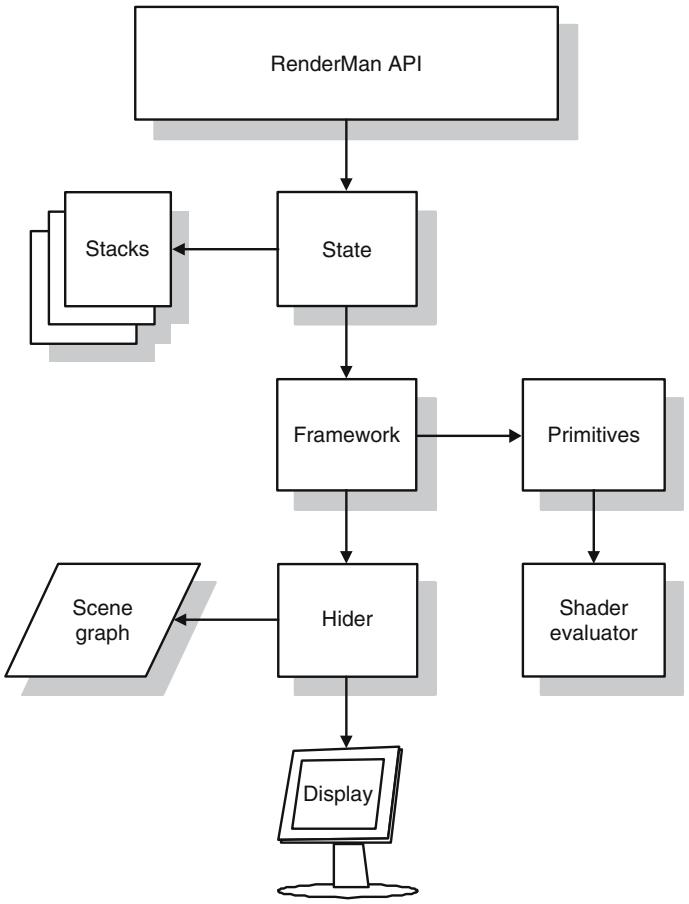

Figure 2.1 An overview of the architecture.

- Attribute
- Transformation
- Object and Light

After examining each of these stacks, we will show how to assemble them into a single class that manages the hierarchical graphics state. Finally, how the state impacts the process of inserting primitives into the scene will be covered.

2.2.1 The Mode Stack

The interface supports the concept of modes to prevent bad input. Functions with names like `RiXxxBegin()` and `RiXxxEnd()` change the mode of the interface. For example, between `RiMotionBegin()` and `RiMotionEnd()` the interface is in motion mode. The complete set of modes are as follows:

`<undefined>`	Before `RiBegin` or after `RiEnd`
`base`	Between `RiBegin` and `RiEnd`