```
147     "particles", [width, height]
148   )
149   .exit_on_esc(true)
150   .build()
151   .expect("Could not create a window.");
152
153   let mut world = World::new(width, height);
154   world.add_shapes(1000);
155
156   while let Some(event) = window.next() {
157     world.update();
158
159     window.draw_2d(&event, |ctx, renderer, _device| {
160       clear([0.15, 0.17, 0.17, 0.9], renderer);
161
162       for s in &mut world.particles {
163         let size = [s.position[0], s.position[1], s.width, s.height];
164         rectangle(s.color, size, ctx.transform, renderer);
165       }
166     });
167   }
168 }
```

清单 6.9 是一个非常长的代码示例，希望此示例中没有你觉得陌生的代码。在此示例代码末尾，我们用到了 Rust 的闭包语法。此处是对 `window.draw_2d()` 的调用，调用的第二个参数是用两个竖线包围的几个变量名（`|ctx, renderer, _device|{...}`）。这两个竖线里面放的是该闭包的参数，后面紧跟的花括号中放的则是闭包体。

闭包是一个以内嵌（in-line）的方式来定义的函数，它能够访问到相邻作用域中的变量。通常也叫作匿名函数或者 lambda 函数。

闭包是 Rust 惯用代码中的常见特性，但是本书会尽量避免使用闭包，这是为了让那些有命令式编程背景和面向对象编程背景的程序员更易于理解。有关闭包的内容，我们会在第 11 章中详细介绍。就现在而言，把闭包当作定义函数的一种便捷方式就足够了。接下来，让我们收集一些证据，来证明在堆上分配变量（数百万次），可能会对代码性能产生的影响。

6.3.4　分析动态内存分配的影响

如果在一个终端窗口中运行清单 6.9 中的程序，在这个终端窗口中，你就会看到有两列数字立刻就把这个终端填满了。这两列数字分别表示，每次分配的字节数量以及完成这次分配请求所花费的时长，时长的单位是纳秒。你可以将这些输出的信息发送到一个文件中，以供进一步分析之用。清单 6.10 给出了具体的操作步骤，通过重定向标准错误，把 ch6-particles 的输出发送到一个文件中。

清单 6.10　创建一份内存分配的报告

```
$ cd ch6-particles
```