大部分人都有俩眼睛。我们通过俩眼睛看到的图像去帮助我们判断物体的远近。存在着很多深度的队列，这就包括到聚焦点的深度，光照下的深度，在我们的视点在物体间做相对运动的时候的深度。OpenGL可以根据你的显示设备，生成两张图片，这两张图片可以分别显示给你的左右眼看，这样一来就增加了场景图像的深度信息了。这里可以用的设备其实挺多的，其中就包括了那些通过不同的物理设备给你提供双目显示的显示设备，还比如需要你带上眼睛去看的到效果的显示设备，以及一些不需要你去穿戴任何装置就可以看到立体场景的显示设备。OpenGL并不关心那些图片是如何被显示出来的，它只关心你希望渲染的某个场景的两个画面-一个是左眼看到的，另一个是右眼看到的。

我们需要窗口系统或者是操作系统的协助来创建立体的显示图像，因此，这套创建出立体显示场景的玩法是依赖特定的平台的。这些技术的实施细节都是由具体的平台实现的，并且与那些平台的框架绑定。现在，我们可以使用sb7的应用程序框架代码帮助我们去创建立体的窗口。在你的程序中，你可以重载sb7::application::init，先调用基类的这个函数，然后如清单9.14那样设置info.flags.stereo为1。因为有些OpenGL的实现需要你的程序去覆盖整个显示器， 你也可以设置info.flags.fullscreen这个标志位，把它弄成一个全屏的窗口。

void my\_application::init()

{

info.flags.stereo = 1;

info.flags.fullscreen = 1; // Set this if your OpenGL

// implementation requires

// fullscreen for stereo rendering.

}

记住，不是所有的显示器都支持立体画面输出的，并且不是所有的OpenGL实现都允许你去创建3D窗口的。然而，如果你是在支持3D渲染的显示设备和OpenGL实现上运行程序的话，你应该已经可以看到一个3D的窗口了。现在我们需要去 往画面里渲染东西。最简单的渲染到3D画面的方法就是把整个场景绘制两次，在我们渲染到左眼画面中之前，我们要调用：

glDrawBuffer(GL\_BACK\_LEFT);

当我们想渲染到右眼画面中去的时候，我们需要调用：

glDrawBuffer(GL\_BACK\_RIGHT);

为了产生一组让人能感受到深度效果的图像，你需要根据左右眼的姿态构建左右眼它们各自的矩阵。记住，我们的模型矩阵把模型转换到了世界坐标系里，世界坐标系是全局的，这一步操作对于双眼来说都是一样的。然而，视口矩阵的作用是把世界坐标系里的东西转换到你的视口中来。所以两只眼睛的视口矩阵是不一样的。也因此，当我们渲染到左眼视图里去的时候，我们使用的是左眼的视口矩阵，并且当我们渲染到右眼里面去的时候，我们使用的是右眼视口矩阵。

最简单的得到立体视口矩阵的方式是简单的把左右眼在水平方向上分开，然后去计算矩阵。作为可选项，你可以把矩阵转动一下，来瞄准视口的中心。另外，你也可以使用vmath::lookat函数去生成视口矩阵。简单的把左眼的位置和左眼的视点传进去得到左眼的视口矩阵，右眼的视口矩阵计算同理。清单9.15展示了这样的样本代码：

void my\_application::render(double currentTime)

{

static const vmath::vec3 origin(0.0f);

static const vmath::vec3 up\_vector(0.0f, 1.0f, 0.0f);

static const vmath::vec3 eye\_separation(0.01f, 0.0f, 0.0f);

vmath::mat4 left\_view\_matrix =

vmath::lookat(eye\_location - eye\_separation,

origin,

up\_vector);

vmath::mat4 right\_view\_matrix = vmath::lookat(eye\_location + eye\_separation,origin,up\_vector);

static const GLfloat black[] = { 0.0f, 0.0f, 0.0f, 0.0f };

static const GLfloat one = 1.0f;

// Setting the draw buffer to GL\_BACK ends up drawing in

// both the back left and back right buffers. Clear both.

glDrawBuffer(GL\_BACK);glClearBufferfv(GL\_COLOR, 0, black);

glClearBufferfv(GL\_DEPTH, 0, &one);

// Now, set the draw buffer to back left

glDrawBuffer(GL\_BACK\_LEFT);

// Set our left model-view matrix product

glUniformMatrix4fv(model\_view\_loc, 1, left\_view\_matrix \* model\_matrix);

// Draw the scene

draw\_scene();

// Set the draw buffer to back right

glDrawBuffer(GL\_BACK\_RIGHT);

// Set the right model-view matrix product

glUniformMatrix4fv(model\_view\_loc, 1, right\_view\_matrix \* model\_matrix);

// Draw the scene... again.

draw\_scene();

}

很明显，清单9.15的代码渲染了整个场景两次。这个玩法可能会因为场景的复杂度而变得非常的耗性能。一种可能的策略是在你的GL\_BACK\_LEFT和GL\_BACK\_RIGHT绘图缓冲区之间交替的进行绘制你场景中的每一个物体。这就意味着， 那些对于状态的改变只能够执行一次，但是改变绘制缓冲区跟我们改变其他的状态一样昂贵。如同我们在本章的前面学到的，我们可以一次性渲染到多个缓冲区上去。事实上，可以想想，如果你的fragment shader可以输出两个数据的时候 会发生什么事情，我们调用下面的代码

static const GLenum buffers[] = { GL\_BACK\_LEFT, GL\_BACK\_RIGHT }

glDrawBuffers(2, buffers);

在这之后，fragment shadear会写入左眼缓冲区，第二个输出会被写入到右眼缓冲区。太精彩了！现在我们一次性把俩眼睛的画面都渲染出来了。然而，并不是很快。记住，即使fragment shader可以输出到很多不同的绘图缓冲区， 但是每个缓冲区里的数据的那些坐标位置是一样的。那么我们如何把不同的画面绘制到每一个缓冲区里去呢？

我们可以做的就是使用Geometry shader，然后把画面渲染到两个layer中去，一个是左眼的，另一个是右眼的。我们将会使用geometry shader的instancing技术执行两次geometry shader，然后通过调用的index来标记到底渲染 目标是哪个layer。每一次调用geometry shader的时候，我们可以选择model-view矩阵中的一个并且在geometry shader中完成那些本来应该在vertex shader中完成的工作。当我们渲染完毕整个场景后，framebuffer的两个layer中 就会有我们的左眼和右眼的画面了。现在我们需要做的就是通过全屏四边形技术，把这两个layer的画面读出来，然后把结果写入到它的两个输出变量中，就是这俩变量把数据分别输出到了左眼视图和右眼视图中去的。清单9.16 展示了一个简单的geometry shader，这个shader是将被应用到我们应用程序中去的shader，它通过一个通道就把我们的立体场景渲染到了两个视图中去了。

#version 450 core

layout (triangles, invocations = 2) in;

layout (triangle\_strip, max\_vertices = 3) out;

uniform matrices

{

mat4 model\_matrix;

mat4 view\_matrix[2];

mat4 projection\_matrix;

};

in VS\_OUT

{

vec4 color;

vec3 normal;

vec2 texture\_coord;

} gs\_in[];

out GS\_OUT

{

vec4 color;

vec3 normal;

vec2 texture\_coord;

} gs\_out;

void main(void)

{

// Calculate a model-view matrix for the current eye

mat4 model\_view\_matrix = view\_matrix[gl\_InvocationID] \* model\_matrix;

for (int i = 0; i < gl\_in.length(); i++){

// Output layer is invocation ID

gl\_Layer = gl\_InvocationID;

// Multiply by the model matrix, the view matrix for the

// appropriate eye, and then the projection matrix

gl\_Position = projection\_matrix \* model\_view\_matrix \* gl\_in[i].gl\_Position;

gs\_out.color = gs\_in[i].color;

// Don't forget to transform the normals...

gs\_out.normal = mat3(model\_view\_matrix) \* gs\_in[i].normal;

gs\_out.texcoord = gs\_in[i].texcoord;

EmitVertex();

}

EndPrimitive();

}

现在，我们已经把场景渲染到了我们的framebuffer的layer中去了，我们可以把结果数组纹理通过全屏四边形的方式，通过一个shader把数据分别渲染到左右眼的缓冲区里去。清单9.17展示了这样的一个样本shader

#version 450 core

layout (location = 0) out vec4 color\_left;

layout (location = 1) out vec4 color\_right;

in vec2 tex\_coord;

uniform sampler2DArray back\_buffer;

void main(void)

{

color\_left = texture(back\_buffer, vec3(tex\_coord, 0.0));

color\_right = texture(back\_buffer, vec3(tex\_coord, 1.0));

}

图9.8展示了我们这个程序在一个立体显示器上显示的时候的效果。这里需要拍一张照片来展示，因为屏幕截图无法同时展示立体视图的两个画面。然而，通过立体渲染方式渲染出来的画面在照片中可以清晰的被看到，另一个效果更好的 图片可以在Color Plate 2中找到。

