

**Project Report**

**（ 2019 / 2020 1st Semester）**

****

|  |  |
| --- | --- |
| **Course** | Computer Graphics |
| **Date** | 2019年月23日 |
| **School** | School of Computer Science |
| **Teacher** | Dr. Huang Rui |

|  |  |  |  |
| --- | --- | --- | --- |
| Name | 周超轶 | ID | H16000429 |
| School | 南京邮电大学 | Major | 计算机科学与技术 |

第一章 Shadow Mapping及研究背景

1.1 Shadow Mapping的介绍

早在上世纪七十年代末，Williams在他的“Casting Curved Shadows on Curved Surface”一文中提出了名为Shadow Map的阴影生成技术。之后，他人在此基础上针对相关问题做了许多改进。现在，Shadow Map仍被作为主流的阴影生成技术被广泛应用。

Z缓冲在一开始就是Shadow Map技术的实现基础。讨论Shadow Map技术的意义，不仅在于了解一种阴影生成技术，还在于可借此掌握一种很有用的技术手段。物体表面上一点，只有在与光源之间没有障碍阻隔时，它的深度值才会被保存到Z缓冲中。换个角度看，这就相当于，在物体表面上某点的深度值被保存到Z-Buffer之前，用此点与光源间连线与场景中所有对象做了一次碰撞检测。借用Z-Buffer做碰撞检测的这一方法，还可以用来帮助处理许多其它问题。

现有的实时渲染的阴影计算都是tricky的，还没有完美的阴影计算算法。但是有几种很不错的方法来计算阴影，这些方法或多或少都有一些问题，需要我们在实现的时候注意。【1】

1.2 Shadow Mapping的研究背景

实时阴影技术的实现，不论在PC端还是在移动端，都显的非常重要，它也是衡量一个3D引擎渲染能力的一个指标，很多开发者或者是引擎开发者对于如何实现实时阴影的原理并不是很清楚，大部分人使用引擎都是在前人做好的基础上去调用接口，对于它的内部实现一无所知，在面试引擎的工作时就被暴露出来了。作为开发者要知其然知其所以然才能提高自己的能力，这也是我们程序员必须要努力的地方。

本报告主要是给读者介绍关于Shadow Mapping的实现原理，实现阴影的算法很多，我们就由浅入深逐步学习。阴影的出现必须要有光源，这跟我们现实生活是一样的，阴天是看不到自己影子的，白天有太阳，晚上有月亮的情况下才能看到自己的影子，游戏作为虚拟现实也是一样的原理。

阴影是光线被阻挡的结果；当一个光源的光线由于其他物体的阻挡不能够达到一个物体的表面的时候，那么这个物体就在阴影中了。阴影能够使场景看起来真实得多，并且可以让观察者获得物体之间的空间位置关系。场景和物体的深度感因此能够得到极大提升，下图展示了有阴影和没有阴影的情况下的不同：

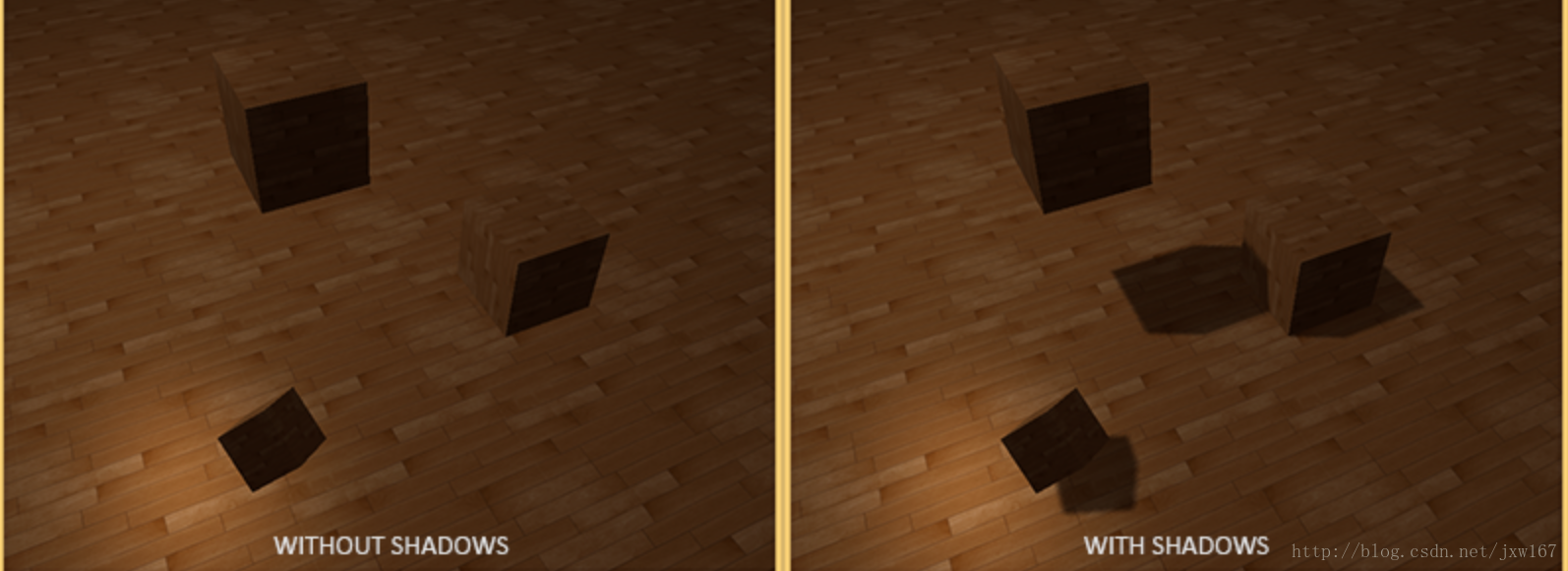


图1 有无阴影的示意图

你可以看到，有阴影的时候你能更容易地区分出物体之间的位置关系，例如，当使用阴影的时候浮在地板上的立方体的事实更加清晰。阴影还是比较不好实现的，因为当前实时渲染领域还没找到一种完美的阴影算法。目前有几种近似阴影技术，但它们都有自己的弱点和不足，这点我们必须要考虑到。视频游戏中较多使用的一种技术是阴影贴图（shadow mapping），效果不错，而且相对容易实现。

第二章Shadow Mapping的理论

2.1 Shadow Mapping的原理

阴影贴图是一种使用深度纹理来为渲染阴影提供解决方案的多通道计算。它的关键是，就是用投射光源代替最终视口来观察场景。通过移动视口到光源位置，可以观察到这个位置每个东西都是明亮的，因为从光的角度来看是没有阴影的。

从光源的角度将场景的深度渲染到一张深度缓冲区中，我们可以在场景中获得一张阴影或者无阴影的贴图，一张阴影贴图。下图很好的解释了Shadow Mapping的原理：

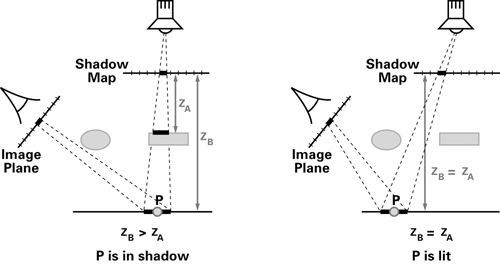


图2 Shadow Mapping的原理

首先将摄像机设置到光源所在的位置(观察方向面对场景，此时叫摄像机观察矩阵为观察矩阵1”，同时设置合理的投影矩阵(这里称此矩阵为投影矩阵1)。

然后对与景进行第二次绘制.此次绘制将每个可以看见的片元(可以观察到指的是片元到光源间无遮挡)到光源的距离记录到一幅纹理图中的对应像素中，供后面的步骤使用。

接着将摄像机恢复到实际摄像机所处的位置，绘制场景。此次绘制时将前面步骤产生的纹理采用投影贴图的方式应用到场景中，进行投影贴图时采用的投影矩阵为投影矩阵1、观察矩阵为观察矩阵1.

绘制每个片元时，根据投影贴图纹理采样的结果换算出光源与此片元连线中距光源最近的片元距离(ZA)，再计算出此片元距光源的实际距离(ZB)。若ZB〉ZA，则需要绘制的片元处于阴影中，采用阴影的颜色着色，否则此片兀不在阴影中，进行既定的光照着色。【2】

Shadow Map实际上比阴影体的原理要简单一些。阴影体是借助Stencil Buffer来做碰撞（观察者视线与阴影体中可能存在的障碍物之间），而Shadow Map则借助Z-Buffer来做碰撞检测。

如图三所示，假设三维空间中，有物体W在光源L照射下形成阴影。空间中的a点位于W与L之间，c 点位于W之后，而b点是W表面上的一点。a、b、c、d经透视投影变换，在屏幕S上对应着a'、b'、c'、d'四个像素区域。

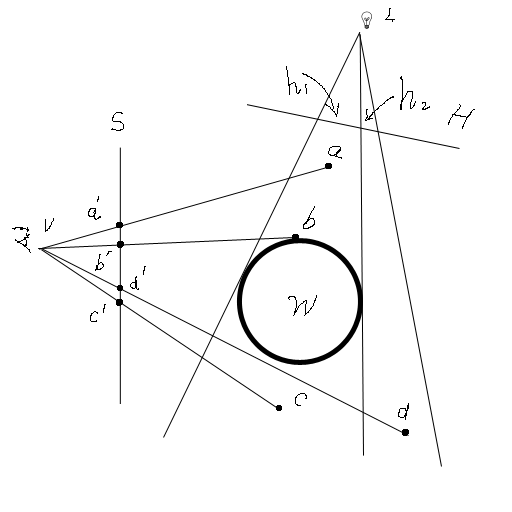


图3 阴影的图像解释

Shadow Map的思想方法是：假设先在光源L处放置一个摄像机（形成所谓的Light Space），则此像机将会把整个场景投影到相应的投影平面H上，其视锥在H平面上的投影是h1和h2两块区域之合。平面H所对应的Z-Buffer保存的是Light Space的所有对象（本例中仅有W）的深度值。在实际生成观察平面S上的像素时，会先将像素对应的空间中的点（如上图中a'、b'、c'、d'所对应的a、b、c、d）转换到Light Space中，投影到H平面上，并将相应的深度值与事先保存在H平面所对应的Z-Buffer的深度值进行比较，以图一为例，a点会投影到区域h1中，由于它位于W之前，其深度值会比H平面的相应Z-Buffer中的值小；b点在h1上的投影点的深度值等于H平面的相应Z-Buffer中的值；c点在h1上的投影点的深度值，则会大于H平面的相应Z-Buffer中的值；由于在生成H平面的投影时，会事先刷新其Z-Buffer的值，刷新值为1，所以在本例中，空间d点在H上的投影的深度值也将小于相应点的Z-Buffer值；因此，通过空间中某一点在平面H上的投影的深度值与H平面原Z-Buffer中的值的比较结果，就可以判断此点是否处于阴影中，并可根据这个判断来设置观察平面S上的相应像素的颜色。

　　考虑这样一种情况，空间中的一点如果处于观察者V的视锥中，同时又位于Light Space的视锥之外，那么显然就无法通过上面的方法来判断它是否被阴影所覆盖。这也是Shadow Map的局限之处。

　　Z-Buffer值一般由图形引擎结合相应硬件，在渲染管线内部计算，用户只需直接调用即可。因此直接使用Z-Buffer的值高效而又方便。但是，通常情况下，Z-Buffer与Stencil Buffer合用4字节空间来描述一个像素，在Shadow Map中用来保存Light Space相应场景对象的深度值一般只有一个字节，而深度值是一个处于0~1之间的浮点数，这样势必会影响到后面的计算精度。这也可看作是传统Shadow Map的另一不足。【3】

绕过Z-Buffer来实现Shadow Map，可以为解决这一问题提供一种方法。

第三章Shadow Mapping的实验

3.1 建一个帧缓冲对象：

GLuint depthMapFBO;

glGenFramebuffers(1, &depthMapFBO);

3.2 创建一个2D纹理，提供给帧缓冲的深度缓冲使用：

const GLuint SHADOW\_WIDTH = 1024, SHADOW\_HEIGHT = 1024;

GLuint depthMap;

glGenTextures(1, &depthMap);

glBindTexture(GL\_TEXTURE\_2D, depthMap);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_DEPTH\_COMPONENT,

SHADOW\_WIDTH, SHADOW\_HEIGHT, 0, GL\_DEPTH\_COMPONENT, GL\_FLOAT, NULL);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

3.3 把我们把生成的深度纹理作为帧缓冲的深度缓冲：

glBindFramebuffer(GL\_FRAMEBUFFER, depthMapFBO);

glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, GL\_TEXTURE\_2D, depthMap, 0);

glDrawBuffer(GL\_NONE);

glReadBuffer(GL\_NONE);

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

3.4 生成深度贴图

glViewport(0, 0, SHADOW\_WIDTH, SHADOW\_HEIGHT);

glBindFramebuffer(GL\_FRAMEBUFFER, depthMapFBO);

glClear(GL\_DEPTH\_BUFFER\_BIT);

ConfigureShaderAndMatrices();

RenderScene();

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

glViewport(0, 0, SCR\_WIDTH, SCR\_HEIGHT);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

ConfigureShaderAndMatrices();

glBindTexture(GL\_TEXTURE\_2D, depthMap);

RenderScene();

第四章 结论

这段代码隐去了一些细节，但它表达了阴影映射的基本思路。这里一定要记得调用glViewport。因为阴影贴图经常和我们原来渲染的场景（通常是窗口解析度）有着不同的解析度，我们需要改变视口（viewport）的参数以适应阴影贴图的尺寸。如果我们忘了更新视口参数，最后的深度贴图要么太小要么就不完整。

第五章 参考文献

【1】Randi J. Rost. *OpenGL shading language*. Addison Wesley, 2004.

【2】Chen, Jim X. *OpenGL Shading Language*. *OpenGL shading language /*. 2010.

【3】Dirk Bartz, Michael Mei?ner, and Tobias Hüttner. "OpenGL-assisted occlusion culling for large polygonal models." *Computers & Graphics*23.5:667-679.