OpenGL作业文档

数媒1501 王宇轩 U201517168

本人郑重申明：所呈交的作品（包括文档、代码、纹理、演示视频）是本人独立进行研究所取得的研究成果。本论文不包含任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本申明的法律后果由本人承担。

签名：

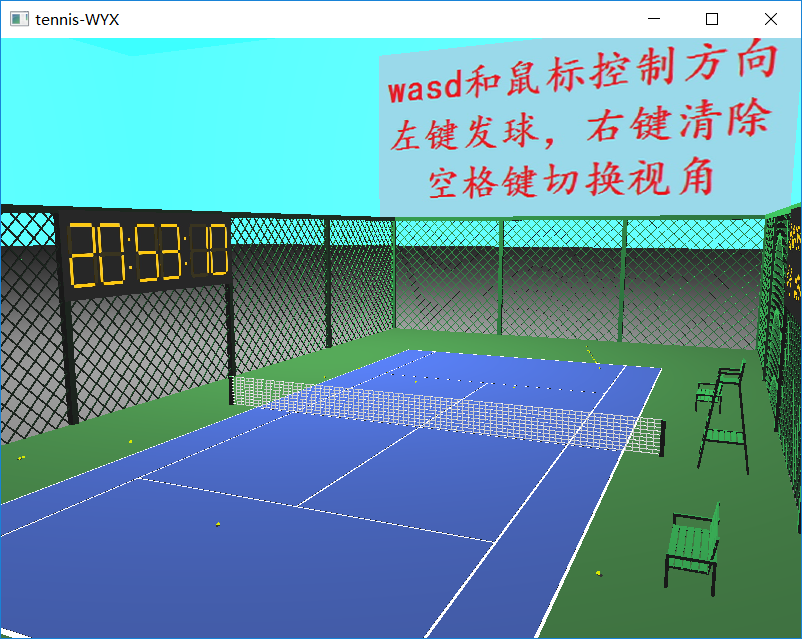
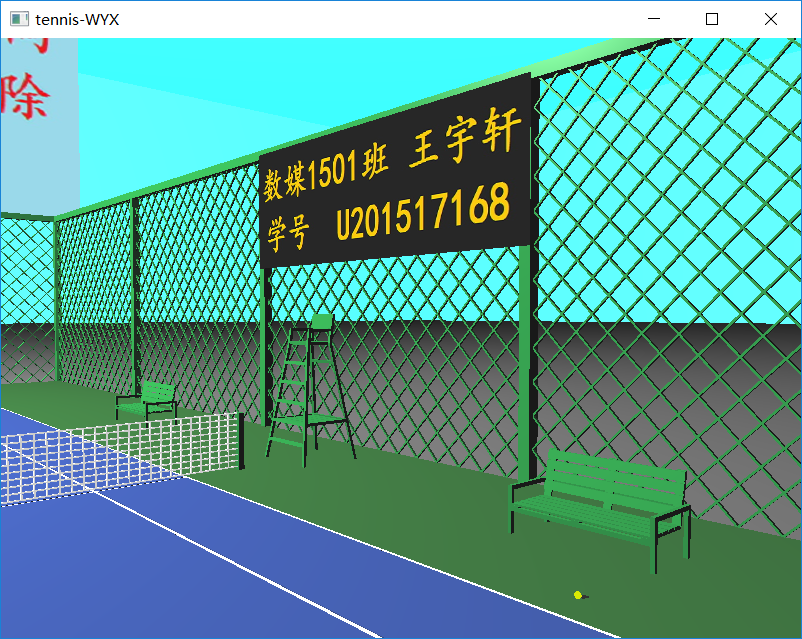
正文

1. **场景总览**

我的作品是一个网球场模型，在VS2015中采用OpenGL3.3编写。里面的元素包括：网球场地的各色区域以及格线、网球场中间的拦网、网球场边的围网、裁判椅、运动员休息椅、时间显示屏、网球及阴影。其中，除了运动员休息椅和时间显示屏外，其他所有元素均按照国际标准尺寸绘制。

使用鼠标和键盘可以完成各项操作，包括：视野的平动及转动（键盘WASD以及鼠标移动）、高低视角的切换（空格键）、从当前位置发射网球（鼠标左键）、清空场地内所有网球并释放其内存（鼠标右键）。

球场前侧、左右两侧放置了显示荧屏，分别写有操作指南、作者信息、当前系统时间，均为贴有纹理的平面四边形。特别的，系统时间显示板的绘制，由纹理及普通的四边形共同组合实现。

1. **物体的绘制**

场景内，所有的物体均由三角形组成。两个等腰直角三角形组合成一个正方形，六个正方形组合成一个正方体。对每一个正方体进行特定的矩阵变换，可以在任何位置绘制任何形状、任何大小的平行六面体。

除了人们所熟知的平移变换、伸缩变换、旋转变换，我还使用了切变变换。切变变换是一个很重要的变换，它能够使横平竖直的长方体变形成各式各样的平行六面体，在本场景中，裁判椅上倾斜的楼梯、休息椅上后仰的靠背、围网上斜向交叉的铁丝，均涉及到切变变换。由于glm.hpp文件中并没有切变变换相关的函数，所以需要自己定义切变矩阵



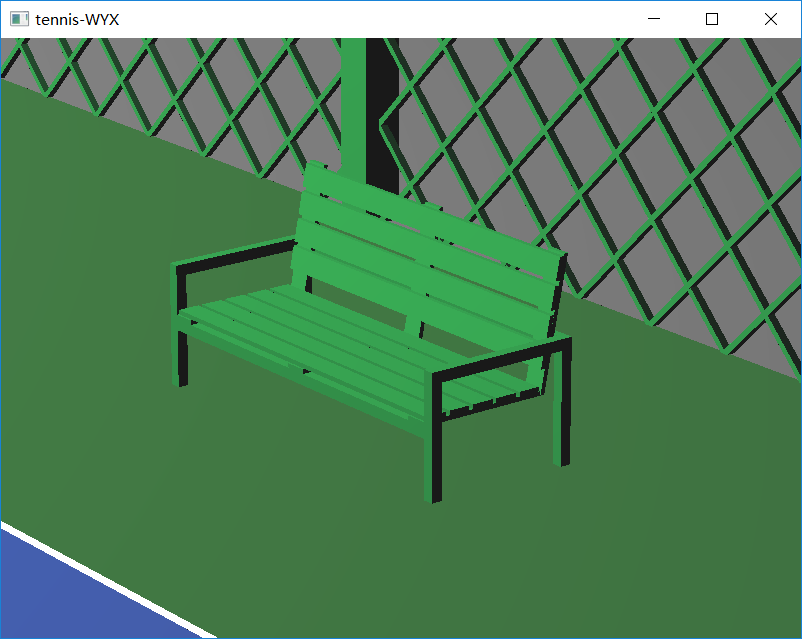
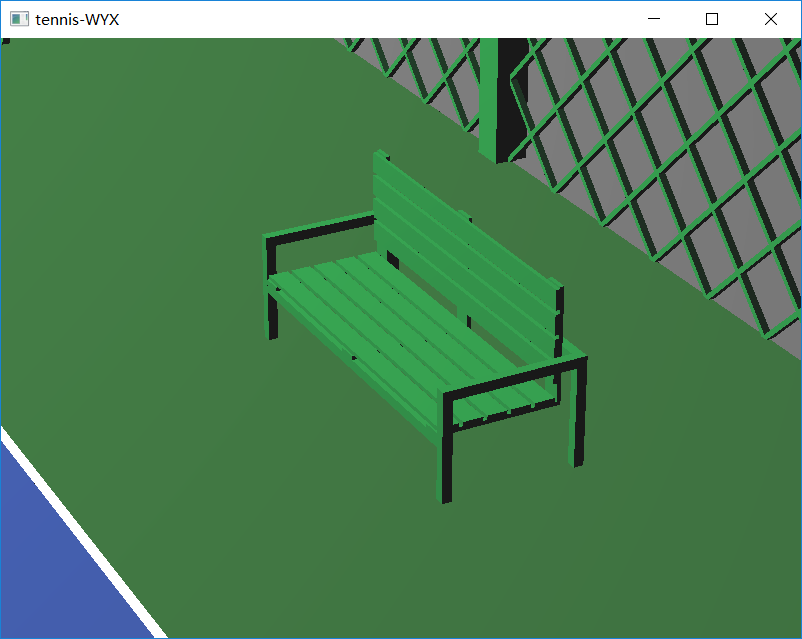
然后对切变矩阵的某一列赋值



然后在合适的位置乘上切变矩阵



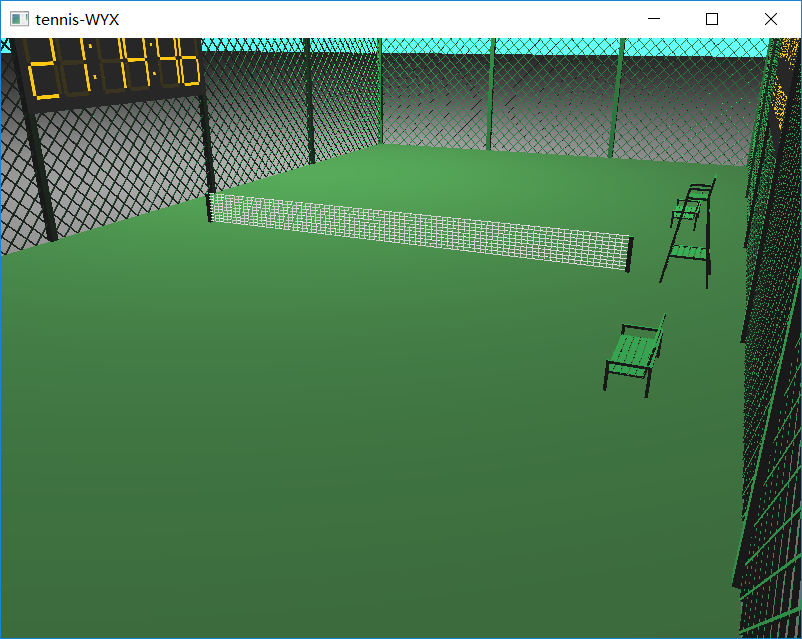
这样就可以绘制我们所期盼的各种规格的平行六面体。



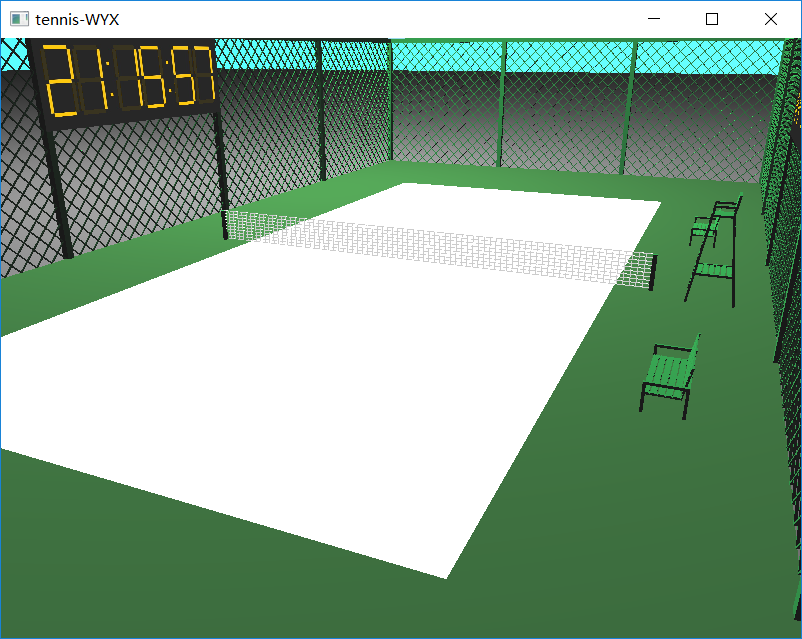
使用了切变变换的座椅（右），明显比未使用切变变换的座椅（左）要真实

对于网球场地上各种颜色区域的绘制，也蕴含着一定程度的技巧。注意到网球场地上是绿色的，而各种击球区域用蓝色标识，区域间用白色线分割。我们固然可以用笨办法，将绿色的区域分割成4块长方体，然后将蓝色的区域分割成10块长方体，然后白线也用若干细长的长方体绘制。但这种方法极为繁琐。为了简化，我采取了一种较为讨巧的方法——层叠法。

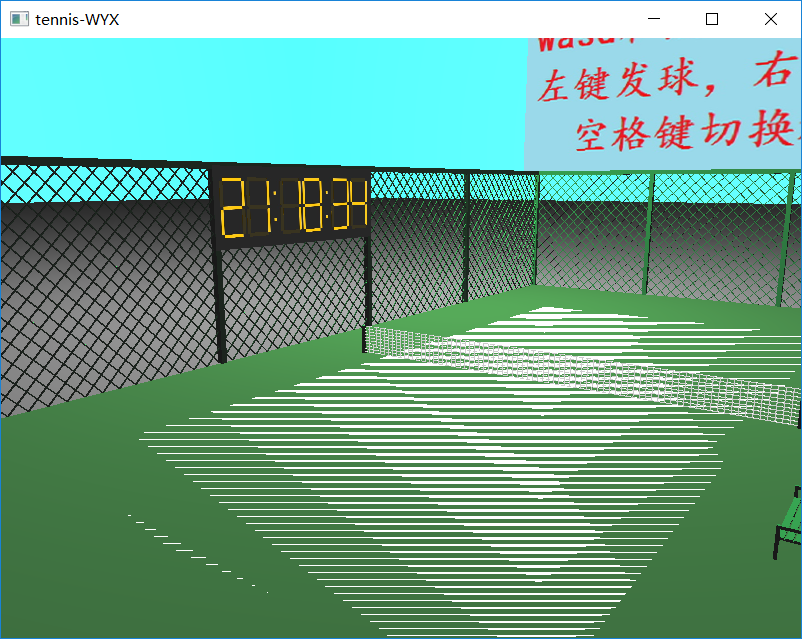
首先我们绘制一个绿色的大草地，它看起来像是这样：



然后我们绘制一个大大的白色的区域：

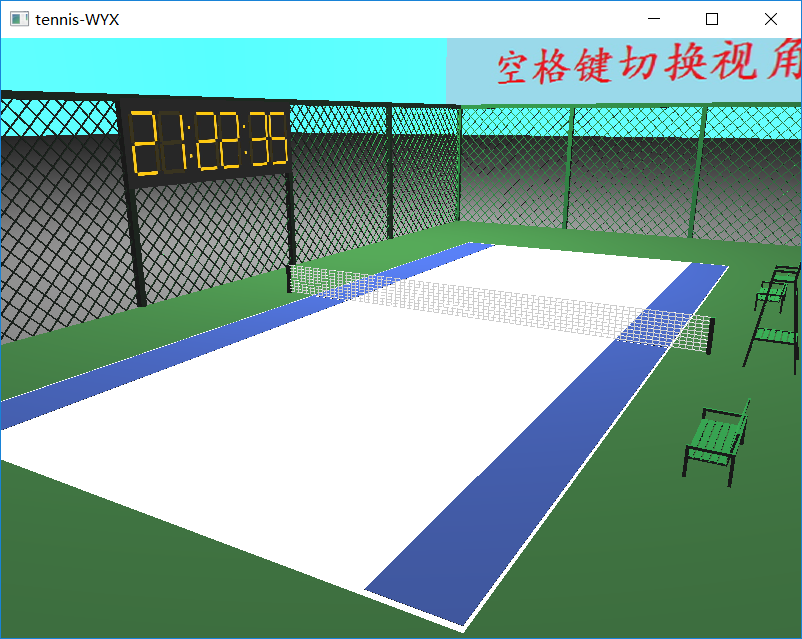
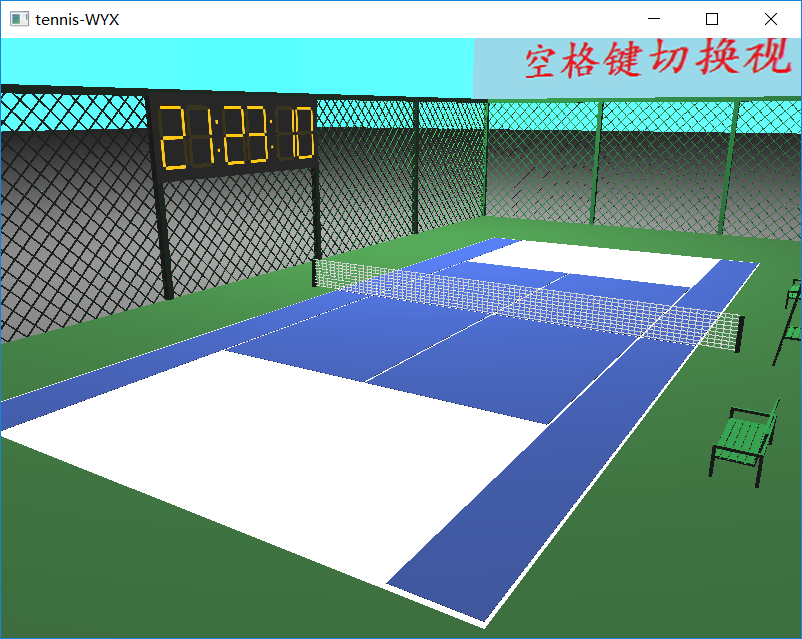
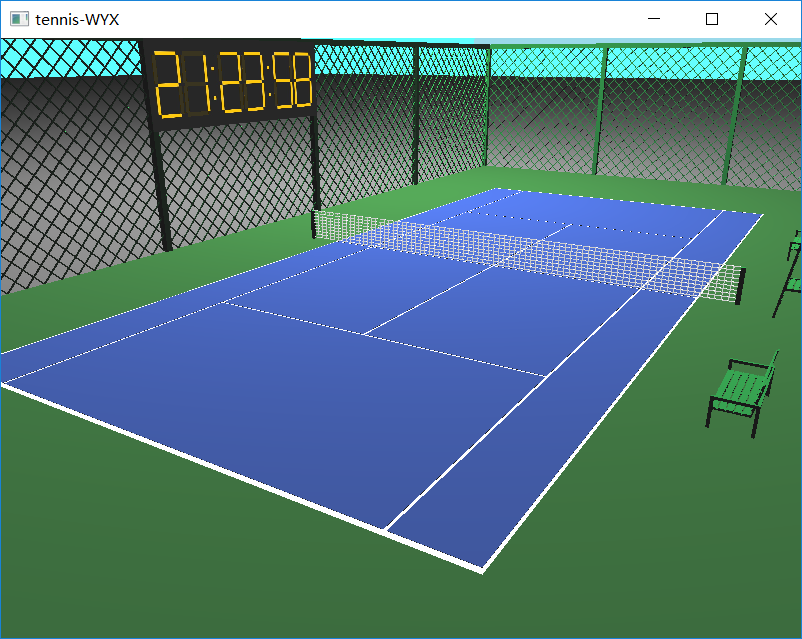


注意：为了让白色区域与绿色区域不至于重叠，我们可以稍稍将白色区域升高。如果不这么做，我们看到的场景可能会是这样的：



这是由于白色区域与绿色区域共用了同一个坐标，系统不知道究竟该在那个坐标上绘制哪个颜色。

接下来，我们用同样的方法，在白色区域上分别放上各种不同规格的蓝色区域，绘制的过程看起来像是这样：

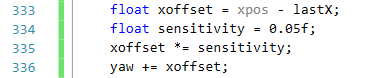
当然，为了让蓝色区域不至于和白色区域重合，同样要把蓝色区域的高度调高一些。

1. **摄像机的改进**

在Learn OpenGL网站上提供了一种鼠标键盘操控摄像机的方法，可以较好地实现摄像机各种形式的移动，它的网址在这里：

<https://learnopengl-cn.github.io/01%20Getting%20started/09%20Camera/#_3>

利用鼠标x坐标改变偏航角yaw的方式，网站上是这么做的：



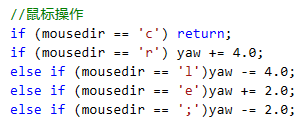
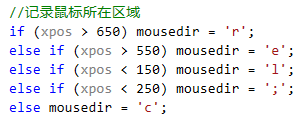
我们看到，偏航角yaw取决于鼠标的横坐标xPos的改变量。偏航角yaw改变的最大范围与窗口的水平宽度直接相关，这种操作其实很不友好。试想，如果由于操作失误，鼠标从窗口右侧离开了窗口，那么偏航角yaw将一直保持某一个正值，当鼠标再次回到窗口，视野将很难调控。

对此，我稍微修改了偏航角yaw的的计算方法。它将不与相关。我参照了几款比较著名的RTS游戏，探求了其中场景移动的规律，并进行改良。

我将xPos的取值分为五个部分：左侧、右侧、中间偏左、中间偏右、中间。当鼠标落在其中不同部分时，偏航角yaw会进行不同程度的变化。



在鼠标移动回调函数中，我们根据鼠标当前的位置给全局变量mousedir赋值，告诉系统当前鼠标所在区域。然后在输入函数processInput中根据mousedir的当前值来计算偏航角yaw。经过多次尝试，最终确定了一个最舒服的灵敏度。它的代码看起来像是这样：

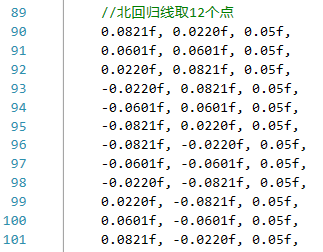
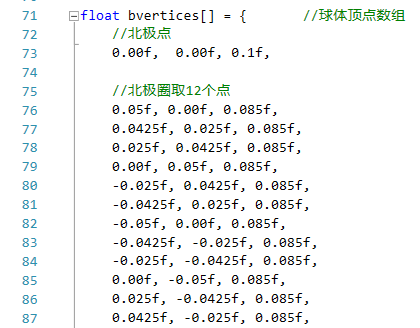


我们用同样的方法优化了俯仰角pitch的计算，优化后的摄像机操作更加顺滑，简直可以和星际争霸媲美。

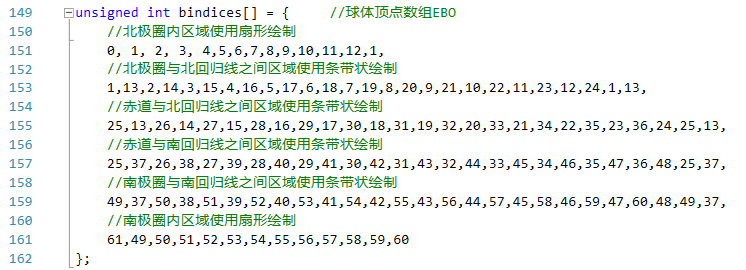
1. **球体的绘制**

球体的绘制是一大难点。OpenGL 3.3的基本图形是三角形。要想绘制球体，就需要把球体分解成若干个三角形。

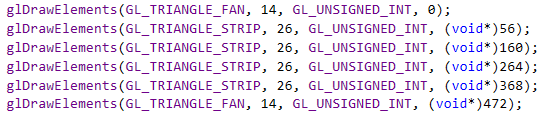
我们借用地球的纬线来更加方便地说明。地球的南北极点、南北极圈、南北回归线，将地球分为六块区域，包括北极圈以北、北极圈与北回归线之间、赤道与北回归线之间，以及南半球的对应区域。我们可以用这种方法近似绘制一个球体。它的部分坐标看起来像是这样：



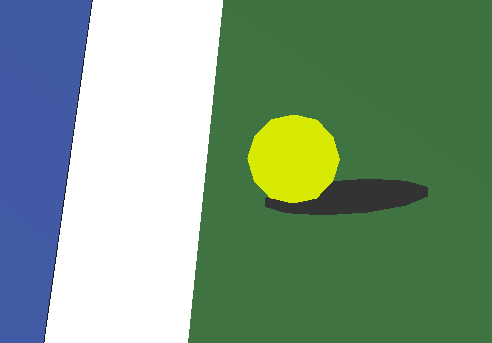
然后对于每一部分区域，我们采取不同的方式去绘制。例如北极圈以内区域，我们选取北极点和北极圈上的12个点，然后采取扇形绘制；而赤道与北回归线区域，我们则选取赤道和北回归线上各12个点，采取条带状绘制。



最后的绘制过程像是这样：



最后，我们使用一个链表来记录当前所有球体的位置，这样不论多少个球，我们都可以绘制出来。



可以看出，球的轮廓上仍有一些棱角

1. **球的阴影**

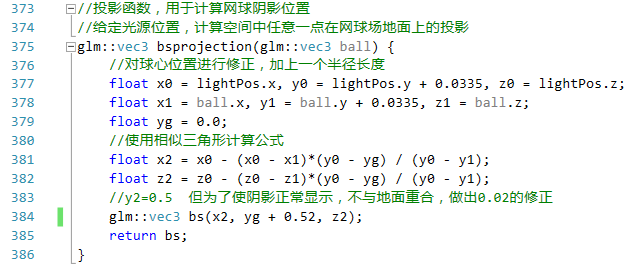
为了使场景看起来更加真实，我们需要给一些物体附加一些投影。在这里，我们重点讨论网球在地面上的投影。

在场景中，地面是一个 y = 0.5f 的水平面，光源位置也已确定（当然光源位置即使不断变化，也能正确绘制阴影）。



我们把光源的纵坐标y0设置得足够高，使得任何一个网球都不可能比光源更高，这样保证了任何一个网球都能在地面上有一个投影。

用简单的相似三角形知识，可以写出这样一个函数，它可以计算出空间中任何一点（高度不超过光源）在光源的照射下投影在地面上的坐标。



在这里，我使用了之前绘制网球场地面的类似伎俩，将阴影做出了0.02的修正，保证阴影和地面不重合。

理论上，我们使用这种方法，然后将球心和绘制球体时所计算的62个顶点都投影一遍，就可以阴影。实际上，这种做法不是很经济。一旦网球数目上升，要投影的顶点个数也将快速上升，在一帧的时间内，我们需要计算成百上千个顶点。于是，我们想得到一种更加简单的计算方法。

我们知道，球的投影其实可以简化为圆的投影，这个圆是球的一个截面，并且垂直于光源与球心的连线。我们只要在这个圆截面上取若干个点，就能够绘制出球体的阴影。

为了得到圆截面上每个点的坐标，我们需要先计算出这个圆截面上的一组正交基，它们的长度均为球的半径。

设正交基为r1, r2。我们可以先直接写出一个基向量：



利用这个基，以及光线的向量，做一次叉乘，能够得到另一个基向量：



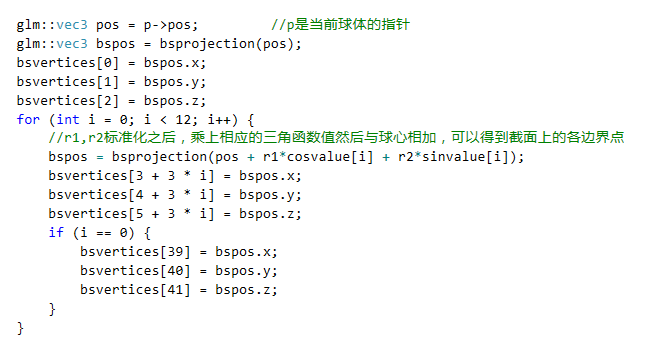
由于光源和网球不等高，所以分母上的y0 - y1 不为零。

接下来对这组基向量的长度调整为网球的半径。网球的标准半径是3.35cm。

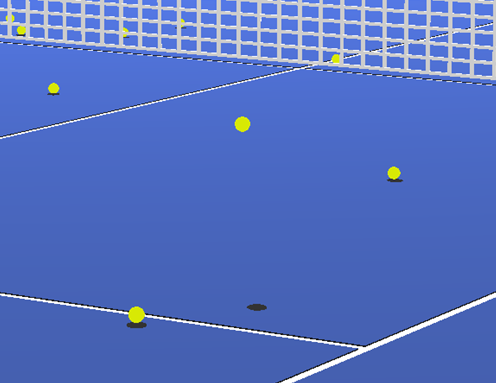


为了让阴影更加明显，认为地将r1放大了0.2倍。对于r2有类似的操作。

在这之后，我们就可以用这组标准化的正交基，以及球心坐标，来表示圆截面上的所有点（在本例中，总共取了12个点），再带入到之前编写的bsprojection函数中，求得地面上阴影的顶点坐标，这段代码看起来像是这样：



最后将计算出的14个点采用扇形绘制，得到网球的阴影。



网球阴影有助于更好地判断网球的高度