**作业三**

# 一、编译与运行代码

直接运行即可，会在控制台内有一些提示性输出，通过按键与鼠标控制来实现一些功能。

可以用命令行参数：e，t，r选择TimeStepper。分别代表欧拉法，梯形法，RK。当然如果不选择会默认为RK4。

# 二、合作与交流

在初期制作时，对于粒子链条的力一直无法计算正确。然后和同学一起探讨问题。得出正确的结论。即公式： X(t + h)=X + hf(X,t)处，将x与f分别拓展为2为向量时。需要注意f与x处具体对应的是什么。对于位置：x1=x0+h\*v。对于速度：v1=v0+h\*f。之前是在计算位置时直接带入的力，因此单摆从高处释放后到垂直时会戛然而止，因为按照错误的公式会达到多力平衡状态，而不会再向另一边回摆。

在与同学交流粒子系统中，发现自己之前遗漏了阻尼力，之前以为是弹簧的相互作用力，因此只是带入的弹簧的k。所以正确的做法是设置多个k值，并进行调参。

# 三、参考文献

主要是在实现鼠标拖拽功能处查阅了部分相关资料。

1.首先查阅了OpenGL如何实现鼠标拖拽，发现是需要使用其中自带的拾取模式。于是查阅了拾取模式的流程，以及如何实现。因此先跑了一遍网上的代码，看看效果。

链接：<http://blog.sina.com.cn/s/blog_73428e9a0101fic2.html>

2.然后就按照具体教程改了改，发现还是不能达到效果。问题就是已经可以在二维拾取，但是旋转后位置不正确。因此看了看一些深入理解的文章。但是好像帮助不太大。

链接：<https://www.cnblogs.com/icmzn/p/7373663.html>

3.花了很长时间一通乱改之后还是不对，最终根据更具体的问题找了找文章。

链接：<https://bbs.csdn.net/wap/topics/330075291>

4.发现应该是需要在拾取模式时考虑到矩阵的变换，使得拾取模式与当前看到的点绘制相同。因此最终找到了可能是设置视点的问题。

链接：https://bbs.csdn.net/topics/340059961

# 四、已知问题

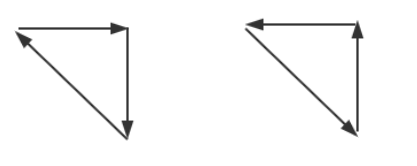
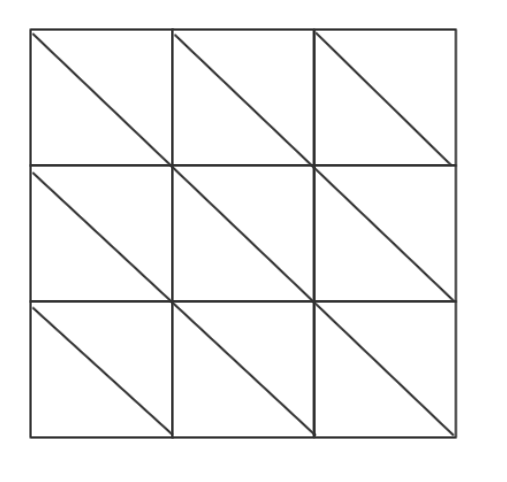
1.对于弹簧的k值，重力，点之间间距调的还是不太理想，布料显示比较松。

2.在与求进行碰撞时。我认为如果是无摩擦碰撞那么速度应该保持原状，但是有些情况下，布料与球持续挤压，布料的速度保持不变但是总是被球弹回来会导致模型爆炸。一种解决方法是把反弹回的点的速度设为0。那么不会存在爆炸的情况，即不会产生剧烈抖动。但是会使得一些点粘连到球体上。

# 五、实现附加分

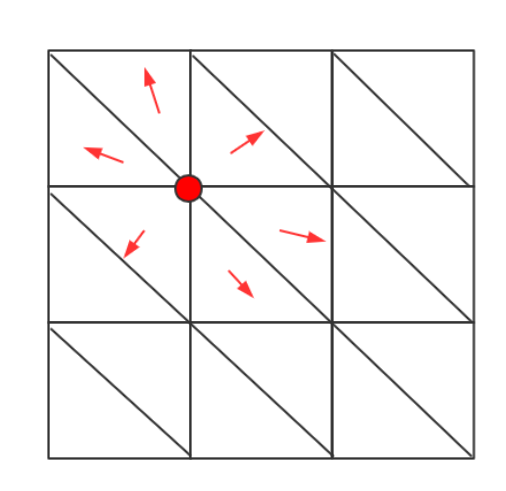
## 1. 布料光滑渲染

一开始是通过画三角面实现布面的显示。但是遇到一个问题，就是反面是没有任何显示的。最终我想到了作业1中的解决方案。就是再重复用三角形画一遍反面。但是需要调换位置。我个人理解的是画三角形时是由正反面的，例如本题中可能就是左手螺旋大拇指指向的方向就是正面。



由于没有法向量，因此最初看到的布是十分暗淡的。然后按照之前写过的程序，在布料上加了一些面法向量。看到的是鳞片状的效果。因此需要加入点发向量。

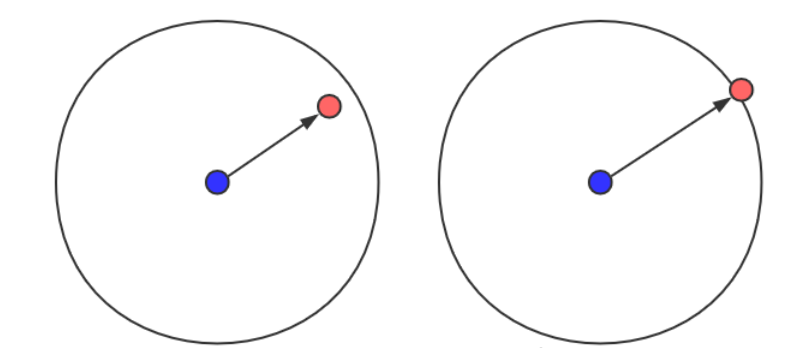
某一点的法向量是把所有的以此点构成三角形的面的法向量进行求和最后进行归一化即可。需要注意所有面法都要先进行归一化保存起来，确保权重相同。此处主要考虑一些边界值以及如何减少重复计算来提高效率。比如一个点的正反面只是一个负号之差，因此只需要计算一遍即可。



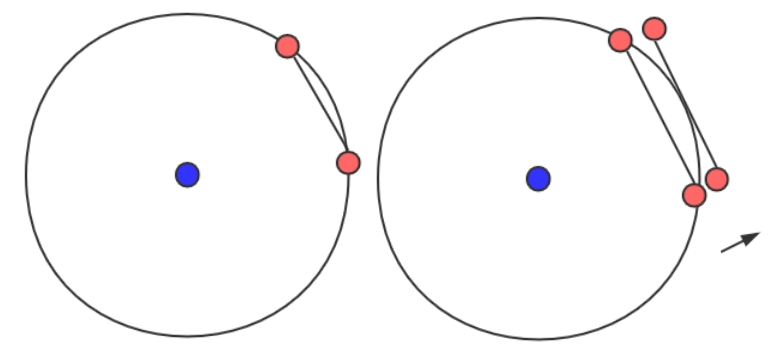
## 2.球体无摩擦碰撞

画出一个球体。记录好球的球心。每当TimeStepper计算完新的状态，需要设置新的状态前，需要判断当前点与球心的距离是否小于球的半径。如果大于球的半径则保持原状态，如果过小则把当前点移到球的表面。也就是在球心与当前点之间设置一个向量。并按照当前方向移动点即可。

设当前点状态为X1，球心坐标为X0，则向量V1为X1-X0，即左图向量。将此向量进行正则化，然后乘上球的半径就可以得到新的向量V2，即右图向量。V2=X2-X0，也就是球表面的位置就是X2=V2-X0，即用得到的向量减去球心坐标即可。



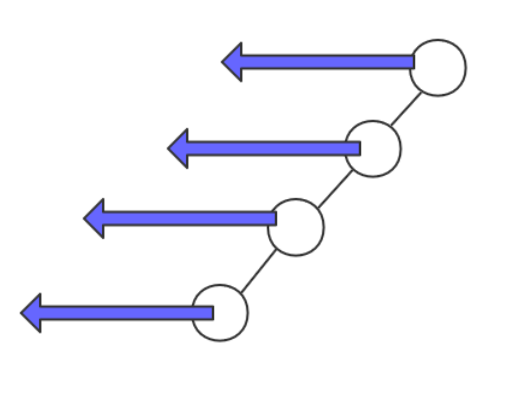
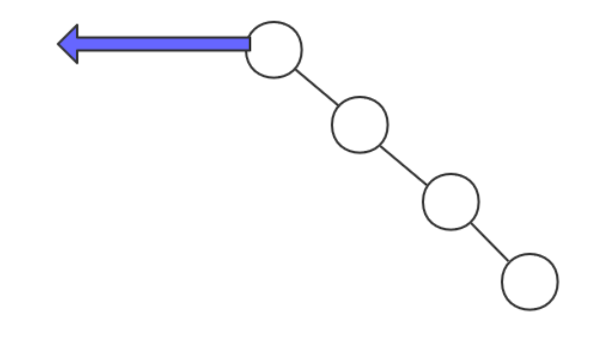
在测试时会发现一个问题，就是虽然一个三角形的三个点都被反弹到球的表面，但是整个面是嵌入到球内的。因为每个边可以看作左图一样。因此为了改进，可以将点向外移动一小段距离，那么面就会浮出来了。不过最好能够做到面与球是相切的，也就是以三角形为一个空间面判断与球心的距离。但是这样考虑的问题比较多，当前点会影响另外两点的位置。因此目前的做法只是稍微移动一小段距离，即一个ε距离。如果在球里面则移动到以此方向从球心移动R+ε距离。



## 3.风力与左右摆动

风力主要是在计算时把每个点都加上一个水平力即可。

对于左右摆动，只是给固定点加一个力。在一个范围来回运动。用固定点拖着其他部分一起运动。

## 4.鼠标拖拽

### 4.1前言：

鼠标拖拽点这一部分做的时间比较长，最终做的不是特别满意，而且仍然不太清楚原理…

实现效果：布料是由多个点构成的，那么鼠标点击布料上一个点的位置就可以进行一个在3维方向的移动。然后在选中的过程中持续拖动鼠标就可以实现一种拖动的效果。并且为了显示具体的拾取位置，或者说证明你是否选中了某一个点，题目中是把点变成红色以示区分。

对于拖拽不足的地方在于，在相机伸缩变换时，没有重新调整拾取框大小，导致可能无法正确拾取到，或者点到空白位置拾取到错误的点。另外，目前只是实现了拾取单个点的功能，对于多个点堆积到一起的时候，是无法失去任何一个点的。这里应该是增大拾取数量或者是根据深度或者是名字栈的顺序来选取。

### 4.2拾取模式的细节与做法：

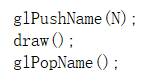
### 4.2.1初步了解--拾取模式

本题的拖拽主要使用OpenGL的拾取模式。主要原理就是将当前三维世界投影成二维像素图片。然后根据当前鼠标的位置设置一个矩形框。如果框到一些物体，那么就说明是选中它了，

首先，在拾取模式中，并不是直接把当前绘制的图形进行投影的，是需要在模式中重新绘制一遍，只是绘画出的物体并不会显示在界面中，而是画在内部，便于计算。因此，对于我们当前已经显示出的物体，需要在拾取模式中重新绘制一遍。

### 4.2.2标识物体信息--名字栈

另一个问题，就是OpenGL如何知道我们选择的是什么物体？这里用到了名字栈。个人理解使用方式和矩阵栈是一样的，都有push与pop，和load。名字栈就是标识在当前栈顶名字下画出的物体的编号即姓名是什么。那么对于此题，我们是每画一个点就给出一个名字，方便区分。并且名字栈的好处是对于画一个涉及到多个点的物体，可以在同一组push，pop中绘画，那么他们的名字不需要重复设置，就可以是全部相同的名字。



### 4.2.3三维世界与二维窗口对应--投影与视口

接下来，如何进行选取呢？根据我自己的理解，首先我们需要将三维投影到二维中来。在拾取模式这里我们和绘画模式保持一致。例如本题看到camera中是透视投影，且角度为50。那么这里我们也做出此操作。



PS：这里困扰了我很久，因为在参考题目用到了正交投影，不知道对于本题的具体的参数如何设置。最终就干脆调成一样效果还是比较不错的。

投影到二维之后，我们需要设置一个选取范围，也就是设置一个小的矩形框，如果物体在这个框中，我们就认为它被选中了。该矩形框是以当前鼠标坐标作为中心（由于在屏幕中是左上角为原点，而OpenGL中是左下角作为原点，因此对于y周需要进行调整）。然后自己设置一个长宽的像素范围。



这里需要设定一个视口（viewport）。也就是当前窗口的坐标，左下和右上的两个二维坐标。这样就可以将之前的三维空间投影并对应到二维屏幕中去。

### 4.2.4得到拾取信息--设置缓存数组

之后，如何得到拾取的信息，或者说是名字呢（之前在名字栈中设置的物体名字）。OpenGL中是退出拾取模式就直接返回选中物体的个数与具体信息。所以，我们在进入拾取模式之前需要先做两件事情，一个是设一个GLint型变量，用于接收返回的点的个数。另一个是设一个GLuint型的数组，负责返回具体信息。



这个数组，需要规定长度，且每个拾取到的物体都会被记录5个值，具体有点记不清，不过第五个是物体的名字，有两个是最大最小深度。所以我们只需要得到第五个值就行了。所以这个数组的长度就是5的倍数。



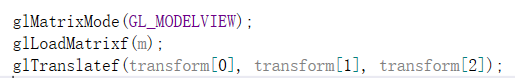
在进入拾取模式前需要先传入数组和他的长度。这类似于引用的传递，当退出拾取模式时，这个数组会更新成拾取的信息。我们就可以直接操作该数组来得到信息了。



### 4.2.5视点变换下的拾取—设置ModelView矩阵

如果之前做的比较都正确的话，基本能实现二维下的选取，这里并不是说只有二维坐标，而是没有任何相机绕周旋转或平移下的操作下，拖拽时是正确的。相机进行缩放是正确的，因为我们在之前设置了透视投影的深度，那么如果在此范围之间都可以拾取到。而对于旋转，我们却无法正确拾取到。在本题中可以看到，真正的点的位置是错位的，点的位置仍然是一个二维的状态下。

这是由于在拾取模式下我们直接先LoadIdentity矩阵。所以对于物体的旋转是在拾取模式下是没有设置的。所以为了保持一致。我们需要将视点设为一致状态，这样投影回来的坐标就是保持一致的。设置视点需要进入ModelView模式下设置当前的视点矩阵。还好camera能够直接返回对应矩阵。不然只能使用glLookAt函数并传入多个参数。我们需要设置当前视点矩阵，并且作出一定的平移变换，才能与绘制时保持一致。



### 4.2.6鼠标移动--鼠标与物体的移动方向如何对应

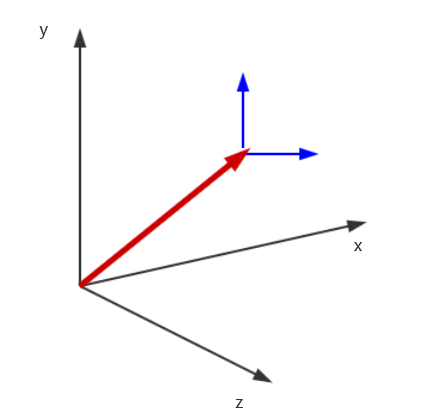
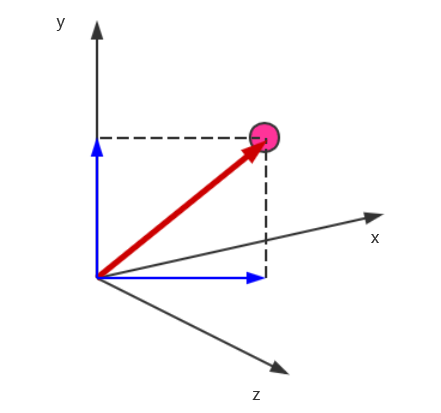
（1）之前的做法。存在错误。！！

鼠标的移动是二维的，即只能上下左右进行移动，而我们需要让点在三维方向移动。因此，介绍一种不太恰当的解决方法，也是目前使用的方法。可以看到下图，有一个三维坐标下的点，注意三个坐标轴是对应OpenGL下的方向。则可以将从原点到此点的向量表示当前点的状态。即左图。

我们此处的目的就是让二维鼠标的上下左右的移动方向对应成三维方向，就是将其中两个方向合二为一，这样变成二对二。为了方便或者说是习惯。可以将鼠标竖直方向即y值变化对应成三维下的y轴方向。将鼠标水平移动方向即x值变化对应成三维下，x轴和z轴合成方向。

那么回到左图，可以看到就是拆成两个方向。再将这两个方向单位化，就变成右图。即对应了鼠标移动方向。竖直方向就是（0,1,0）这个向量。另一个向量需要具体计算。

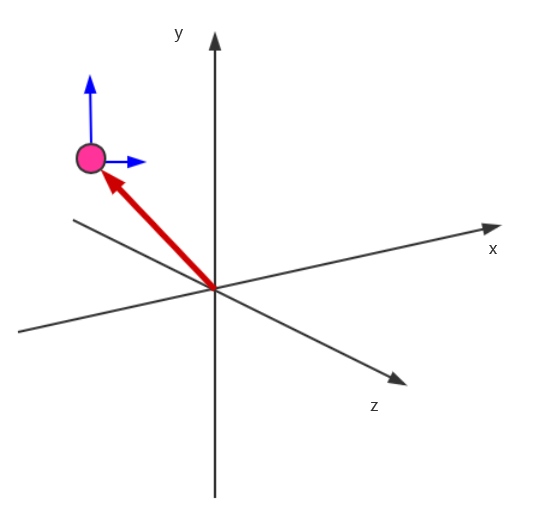
在计算时，就按照以前的方法，设置鼠标前一个状态的固定坐标，通过当前鼠标坐标与固定坐标的差值得到鼠标移动的向量，通过这个向量对应与我们设的单位向量相乘即可得到移动的距离。需要注意仍需要乘上一个恰当的系数，来使得移动不是那么剧烈。



但是此做法有一个错误，就是当一个点在原点时，那个水平向量计算出来为零向量。无法再移动。为了方便解决，如果当前点在原点时，在计算前，先将该点坐标设为

（0.0001, 0, 0.0001）。也就是给一个极小的方向值。那么在原点的细微距离上方向是错误的，但是却能保证点可以在拖拽时经过原点。并且如果跨过原点后，方向仍然是正确的。

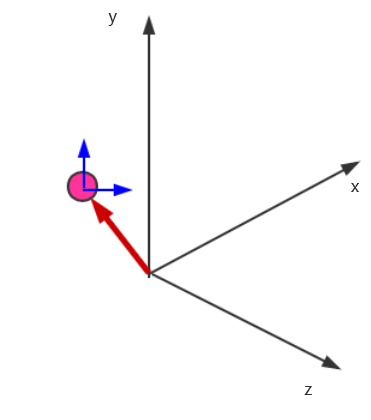
另一个错误，就是当点的x与z全为负的时候，水平移动方向需要取负方向，不然移动的方向是相反的。并且在另外两个象限，即x与z一正一负的情况下，仍然需要考虑具体的移动方向。



（2）新做法！！！

这种移动方法只是对于二维方向的一种稍微的改进。最好的方法是根据视点来求出两个移动方向。也就是水平方向是（1,0,0），数值方向保持（0,1,0）。但是由于相机旋转了，那我们也将水平方向旋转同样状态。就可以得到对应的水平方向了。

需要注意如果相机向左上旋转了一个角度，那么当前物体对于我们来说其实是向右下旋转同样角度的。也就是我们不动，而物体做了一个相对运动。那么根据作业二的逆矩阵的思路。我们只需要将（1，0,0）在前面求一个旋转的逆矩阵即可。就可以得到旋转后的方向。



### 4.2.7拾取模式流程（具体代码流程）

**1.前期操作**

**（1）设置变量：记录hit点个数，缓存hit的信息。**



**（2）设置长度也就是能够返回拾取对象的个数，并设置缓存的返回位置。**



**（3）得到视口**

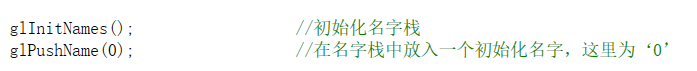


**2.进入选择模式**

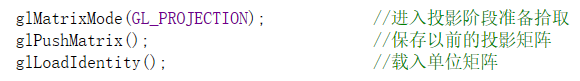
**（1）进入选择模式**



**（2）初始化名字栈**



**（3）投影操作**



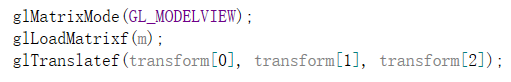
**（4）设置挑选矩形框**



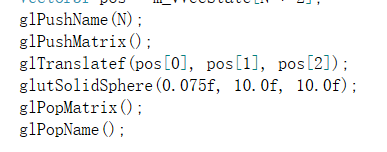
**（5）设置投影**



**（6）注意视点变换**



**（7）名字栈操作，在选择模式下绘画。**



**（8）退出选择模式，并得到拾取的个数与拾取对象信息**

**点个数为hits，而信息存在buffer中。**

