2.26

请教了学长关于这方面的一些问题，并且拿到了实验器材，和代码以及一些需要了解的论文。

2.27

在电脑上安装Kinect程序，并且初步进行实验。

2.28

正式开始看论文，《Real-time 3D Reconstruction at scale using voxel hashing》

详细了解各种方法利弊，优点和不足。

了解signed distance field 算法。

了解ICP算法，对于确定旋转矩阵的方法有些不太明白。

3.1

继续深入了解SVD奇异值分解。

特征值分解和奇异值分解有所区别，也有所联系。

继续看《《Real-time 3D Reconstruction at scale using voxel hashing》》(下用《RVH》代替)，发现需要对KinectFusion有一个大致了解，所以通过网上资料来进行了解。

KinectFusion 算法采用 **frame-to-model** （通过当前帧深度图像转换得到的点云，和根据上一帧相机位姿从模型投影获取的深度图像转换得到的点云进行配准）的方式，而不是采用 frame-to-frame （通过当前帧深度图像转换得到的点云，和上一帧深度图像转换得到的点云进行配准）的形式计算两帧位姿，作者论文里也验证了采用 frame-to-model 的形式重建要更加准确。

KinectFusion重建表面的时候用到的的SDF的方法，这种方法能保证一定纹理精细程度并且再投影的过程中不会有过大变形。同时用到volumetric data structure也就是上面我们所说的model，这个model就是密集网格，每个网格所赋予的数值就是网格到达重建表面的距离。

《RVH》一文中提到，Curless and Levoy 的方法（implicit volumetric fusion），改进SDF为TSDF，为了节省计算开销，减少噪声，存储signed distance的时候剔除了视线锥之外的的voxels,只存储观察范围附近的。这个范围大小可以调整。

文中反复提到哈希索引和哈希表，所以先对其进行了解。

可以看到voxel hashing算法最大的创新点在于voxels的存储，与传统方法不同的是，它将voxels以voxel block为单位存入hash表中，相邻的block在表中未必相邻。定义的哈希函数能很好的提高查找效率并且可以减少哈希冲突。

再不断地更新TSDF时，会不断的剔除那些距离重建表面的太远的voxel block以保证整个存储结构的精简。

在哈希表中，每一个voxel blocks可以通过世界坐标系的坐标整数值来检索。

解决哈希冲突的方法：首先对于一个世界坐标系下的具体坐标，通过哈希函数计算hash值来确定其落入哪一个bucket，每个hash bucket有多个hash entry，这样做的目的就是为了解决哈希冲突，由于每个entry中也存储对应的具体坐标，可以通过简单的比较在hash bucket中找到真正的对应的entry从而得到指针来指向具体存储的vexel block。这里需要注意bucket的容量来确保不会溢出。如果溢出则需要相邻bucket的空间来借用。在溢出的bucket中最后一条entry中设置offset值来指向下一条hash值相同的但是溢出对应bucket的entry，像这样每一条指向下一条可以把所有的entry联系起来。需要注意的是再其他bucket中寻找见缝插针的空间时要注意避开最后一个位置，因为那个位置通常用来作为link list head。

3.2

Voxel hashing算法的voxel的存储结构昨天已经了解，昨天看到定位voxel block 和拼接融合，感觉有些地方不太明白，今天继续把这一块搞清楚。

**Voxel block allocation:**

这个步骤是相机经过移动，得到新一帧图像之后，确定相机视场角内的vexel block的位置为下一步的拼接融合做准备。这里用到的切割方法是DDA，这与计算机图形学中画直线的方法类似。

这里需要说明的是：所有的block，无论视场内还是视场外，在显存中的存储空间都是事先划分的，在 GPU 中用一个 list 存储还没有分配的 block，新 block 分配空间时，从 list 的末尾元素获取存储地址，当一个 block 删除时，删除的 block 的地址放到 list 的末尾。

这里我想做个不恰当的比喻：一家酒店，为将要到来的客人预留了200个空房间，但是具体哪个客人住哪一间还不确定，客人都是一批一批来的，这一批就好比是视场角内的voxel blocks，每来一批都要分配房间，也就是**Voxel block allocation**。

继续往下看，下面很多的内容与Kinectfusion类似。

在存储视场角之外的重建表面数据的时候，需要考虑GPU内存，为了解决这一个问题，文中提到GPU-HOST双向的数据流（streaming scheme）。

当voxel block离开active region 这时候就不必在GPU存储其数据，这时候可以将其转移到host,在host中，其数据结构也不再是哈希表，而是以chunks的形式存储，voxel的entry数据和具体的voxel数据都用链表存入chunks。这样节省了GPU内存并且存储了数据以重复使用并且获得最终三维模型。

对于 Host 向 GPU 的移动，找到完全在相机视场内的 chunk，GPU-to-CPU 是以 block 的单位移动的，GPU-to-CPU 以 chunk 为单位移动。移动时，每次只移动和相机视角最近的单个 chunk。这是为了保证效率。