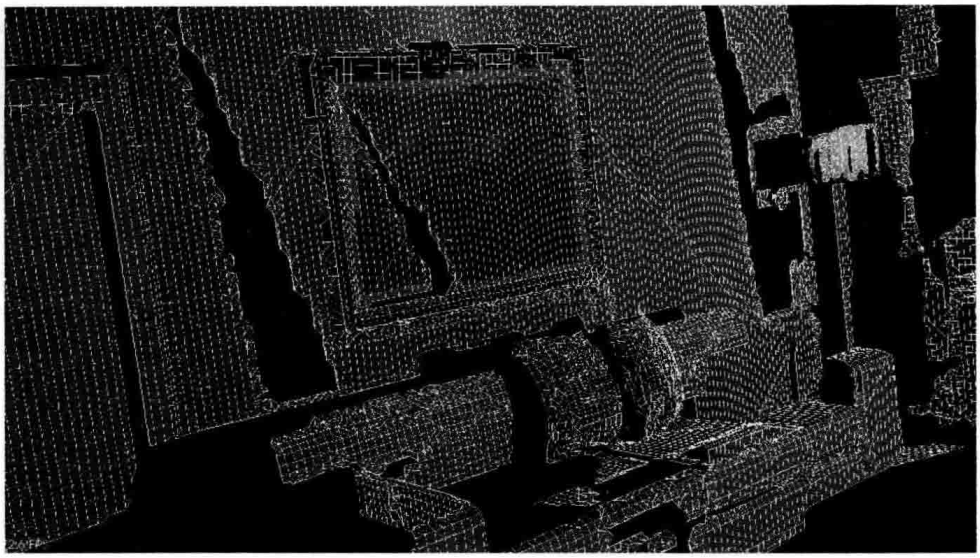


图 12-10 从点云重建得到的表面和网格模型

12.4.3 八叉树地图

下面介绍一种在导航中比较常用的、本身有较好的压缩性能的地图形式：**八叉树地图**（Octomap）。在点云地图中，我们虽然有了三维结构，也进行了体素滤波以调整分辨率，但是点云有几个明显的缺陷：

- 点云地图通常规模很大，所以 pcd 文件也会很大。一幅 640 像素 ×480 像素的图像，会产生 30 万个空间点，需要大量的存储空间。即使经过一些滤波后，pcd 文件也是很大的。而且讨厌之处在于，它的“大”并不是必需的。点云地图提供了很多不必要的细节。我们并不特别关心地毯上的褶皱、阴暗处的影子这类东西，把它们放在地图里是在浪费空间。由于这些空间的占用，除非我们降低分辨率，否则在有限的内存中无法建模较大的环境，然而降低分辨率会导致地图质量下降。有没有什么方式对地图进行压缩存储，舍弃一些重复的信息呢？
- 点云地图无法处理运动物体。因为我们的做法里只有“添加点”，而没有“当点消失时把它移除”的做法。而在实际环境中，运动物体的普遍存在，使得点云地图变得不够实用。

接下来我们要介绍的就是一种灵活的、压缩的、能随时更新的地图形式：八叉树（Octotree）[132]。

我们知道，把三维空间建模为许多个小方块（或体素）是一种常见的做法。如果我们把一个小方块的每个面平均切成两片，那么这个小方块就会变成同样大小的八个小方块。这个步骤可以不断地重复，直到最后的方块大小达到建模的最高精度。在这个过程中，把“将一个小方块分成同样大小的八个”这件事，看成“从一个节点展开成八个子节点”，那么，整个从最大空间细分

到最小空间的过程，就是一棵八叉树。

如图 12-11 所示，左侧显示了一个大立方体不断地均匀分成八块，直到变成最小的方块为止。于是，整个大方块可以看作根节点，而最小的块可以看作“叶子节点”。于是，在八叉树中，当我们由下一层节点往上走一层时，地图的体积就能扩大为原来的八倍。我们不妨做一点简单的计算：如果叶子节点的方块大小为 1 cm^3，那么当我们限制八叉树为 10 层时，总共能建模的体积大约为 $8^{10}cm^3 = 1,073m^3$，这足够建模一间屋子了。由于体积与深度呈指数关系，所以当我们用更大的深度时，建模的体积会增长得非常快。

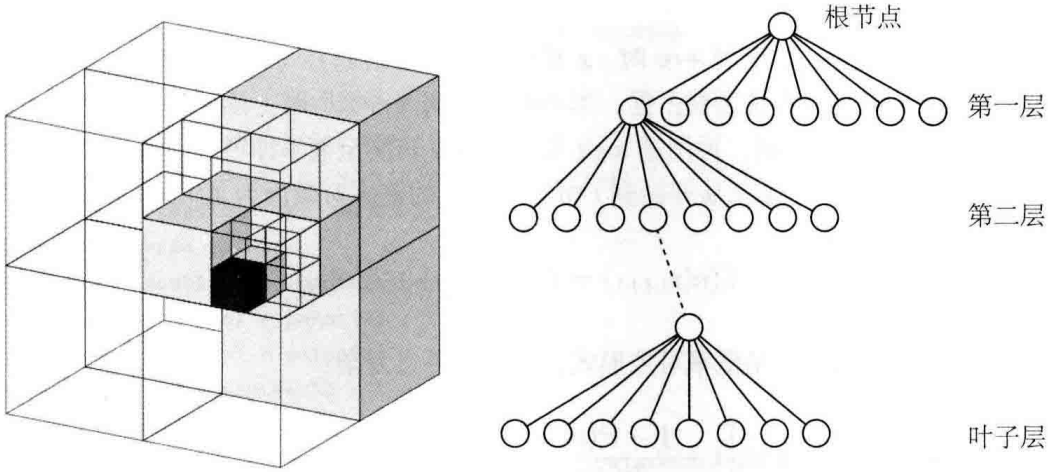


图 12-11　八叉树示意图

读者可能会疑惑，在点云的体素滤波器中，我们不是也限制了一个体素中只有一个点吗？为何我们说点云占空间，而八叉树比较节省空间呢？这是因为，在八叉树中，在节点中存储它是否被占据的信息。当某个方块的所有子节点都被占据或都不被占据时，**就没必要展开这个节点**。例如，一开始地图为空白时，我们只需一个根节点，不需要完整的树。当向地图中添加信息时，由于实际的物体经常连在一起，空白的地方也会常常连在一起，所以大多数八叉树节点无须展开到叶子层面。所以说，八叉树比点云节省大量的存储空间。

前面说八叉树的节点存储了它是否被占据的信息。从点云层面来讲，自然可以用 0 表示空白，1 表示被占据。这种 0-1 的表示可以用一个比特来存储，节省空间，不过显得有些过于简单了。由于噪声的影响，可能会看到某个点一会儿为 0，一会儿为 1；或者多数时刻为 0，少数时刻为 1；或者除了“是”“否”两种情况，还有一个“未知”的状态。能否更精细地描述这件事呢？我们会选择用**概率**形式表达某节点是否被占据的事情。例如，用一个浮点数 $x \in [0,1]$ 来表达。这个 x 一开始取 0.5。如果不断观测到它被占据，那么让这个值不断增加；反之，如果不断观测到它是空白，那就让它不断减小即可。

通过这种方式，我们动态地建模了地图中的障碍物信息。不过，现在的方式有一点小问题：

如果让 x 不断增加或减小，它可能跑到 $[0,1]$ 区间之外，带来处理上的不便。所以我们不是直接用概率来描述某节点被占据，而是用概率对数值（Log-odds）来描述。设 $y \in \mathbb{R}$ 为概率对数值，x 为 0~1 的概率，那么它们之间的变换由 logit 变换描述：

$$y = \text{logit}(x) = \log\left(\frac{x}{1-x}\right). \tag{12.16}$$

其反变换为

$$x = \text{logit}^{-1}(y) = \frac{\exp(y)}{\exp(y)+1}. \tag{12.17}$$

可以看到，当 y 从 $-\infty$ 变到 $+\infty$ 时，x 相应地从 0 变到了 1。而当 y 取 0 时，x 取 0.5。因此，我们不妨存储 y 来表达节点是否被占据。当不断观测到“占据”时，让 y 增加一个值；否则就让 y 减小一个值。当查询概率时，再用逆 logit 变换，将 y 转换至概率即可。用数学形式来说，设某节点为 n，观测数据为 z。那么从开始到 t 时刻某节点的概率对数值为 $L(n|z_{1:t})$，$t+1$ 时刻为

$$L(n|z_{1:t+1}) = L(n|z_{1:t-1}) + L(n|z_t). \tag{12.18}$$

如果写成概率形式而不是概率对数形式，就会有一点复杂：

$$P(n|z_{1:T}) = \left[1 + \frac{1-P(n|z_T)}{P(n|z_T)} \frac{1-P(n|z_{1:T-1})}{P(n|z_{1:T-1})} \frac{P(n)}{1-P(n)}\right]^{-1}. \tag{12.19}$$

有了对数概率，就可以根据 RGB-D 数据更新整个八叉树地图了。假设在 RGB-D 图像中观测到某个像素带有深度 d，就说明：**在深度值对应的空间点上观察到了一个占据数据，并且，从相机光心出发到这个点的线段上应该是没有物体的**（否则会被遮挡）。利用这个信息，可以很好地对八叉树地图进行更新，并且能处理运动的结构。

12.4.4 实践：八叉树地图

下面通过程序演示八叉树地图的建图过程。首先，请读者安装 octomap 库，在 Ubuntu 18.04 版本之后，八叉树地图和对应的可视化工具 octovis 已经集成在仓库中，可以通过如下命令安装：

终端输入：

```
sudo apt-get install liboctomap-dev octovis
```

我们直接演示如何通过前面的 5 张图像生成八叉树地图，然后将它画出来。

slambook/ch13/dense_RGBD/octomap_mapping.cpp（片段）

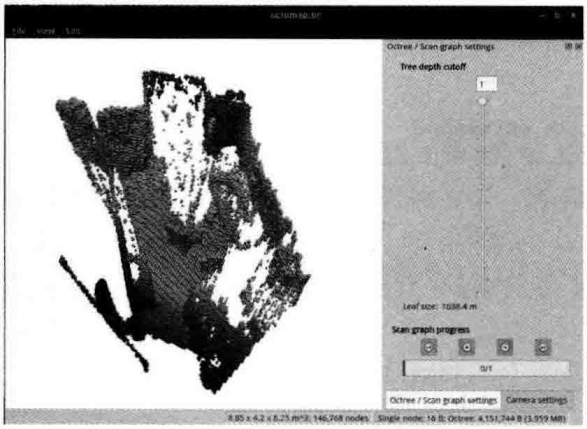
```
// octomap tree
octomap::OcTree tree(0.01); // 参数为分辨率

for (int i = 0; i < 5; i++) {
    cout << "转换图像中: " << i + 1 << endl;
    cv::Mat color = colorImgs[i];
    cv::Mat depth = depthImgs[i];
    Eigen::Isometry3d T = poses[i];

    octomap::Pointcloud cloud;  // the point cloud in octomap

    for (int v = 0; v < color.rows; v++)
    for (int u = 0; u < color.cols; u++) {
        unsigned int d = depth.ptr<unsigned short>(v)[u]; // 深度值
        if (d == 0) continue; // 为0表示没有测量到
        Eigen::Vector3d point;
        point[2] = double(d) / depthScale;
        point[0] = (u - cx) * point[2] / fx;
        point[1] = (v - cy) * point[2] / fy;
        Eigen::Vector3d pointWorld = T * point;
        // 将世界坐标系的点放入点云
        cloud.push_back(pointWorld[0], pointWorld[1], pointWorld[2]);
    }

    // 将点云存入八叉树地图，给定原点，这样可以计算投射线
    tree.insertPointCloud(cloud, octomap::point3d(T(0, 3), T(1, 3), T(2, 3)));
}

// 更新中间节点的占据信息并写入磁盘
tree.updateInnerOccupancy();
cout << "saving octomap ... " << endl;
tree.writeBinary("octomap.bt");
```

我们使用 octomap::OcTree 构建整张地图。实际上，八叉树地图提供了许多种八叉树：有带地图的，有带占据信息的，也可以自己定义每个节点需要携带哪些变量。简单起见，我们使用了不带颜色信息的、最基本的八叉树地图。

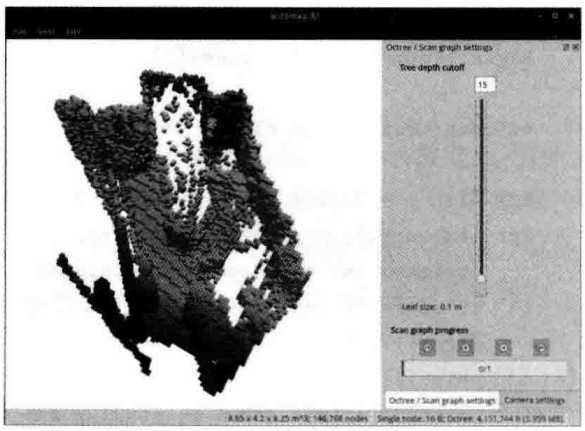
八叉树地图内部提供了一个点云结构。它比 PCL 的点云稍微简单一些，只携带点的空间位置信息。我们根据 RGB-D 图像和相机位姿信息，先将点的坐标转至世界坐标，然后放入八叉树地图的点云，最后交给八叉树地图。之后，八叉树地图会根据之前介绍的投影信息，更新内部的占据概率，最后保存成压缩后的八叉树地图。我们把生成的地图存成 octomap.bt 文件。在之前编

译 octovis 时，我们实际上安装了一个可视化程序，即 octovis。现在，调用它打开地图文件，就能看到地图的实际样子了。

图 12-12 显示了我们构建的地图结果。由于没有在地图中加入颜色信息，所以打开地图时显示为灰色，按“1”键可以根据高度信息进行染色。读者可以慢慢熟悉 octovis 的操作界面，包括地图的查看、旋转、缩放等操作。



八叉树地图（0.05米分辨率）



八叉树地图（0.1米分辨率）

图 12-12 八叉树地图在不同分辨率下的显示结果

操作界面的右侧是八叉树的深度限制条，这里可以调节地图的分辨率。由于构造时使用的默认深度是 16 层，所以这里显示 16 层即最高分辨率，也就是每个小块的边长为 0.05m。当将深度减少一层时，八叉树的叶子节点往上提一层，每个小块的边长就增加一倍，变成 0.1m。可以看到，我们能够很容易地调节地图分辨率以适应不同的场合。

八叉树地图还有一些可以探索的地方，例如，可以方便地查询任意点的占据概率，以此设计在地图中进行导航的方法[133]。读者亦可比较点云地图与八叉树地图的文件大小。12.3 节生成的点云地图的磁盘文件大小约为 6.9MB，而使用八叉树地图的磁盘文件大小只有 56KB，连点云地图的 1% 都不到，可以有效地建模较大的场景。

12.5 * TSDF 地图和 Fusion 系列

在本讲的最后，我们介绍一个与 SLAM 非常相似但又有稍许不同的研究方向：实时三维重建。本节内容涉及 GPU 编程，并未提供参考例子，所以作为可选的阅读材料。

在前面的地图模型中，**以定位为主体**。地图的拼接是作为后续加工步骤放在 SLAM 框架中的。这种框架成为主流的原因是定位算法可以满足实时性的需求，而地图的加工可以在关键帧处进行处理，无须实时响应。定位通常是轻量级的，特别是当使用稀疏特征或稀疏直接法时；相应