常用数据结构

1 KD Tree

1.1 基本概念

KD Tree 是一种用于场景划分的数据结构，可用于快速查询场景中的物体。它要求划分平面垂直于某一个坐标轴，并把空间划分为左右两个子空间。具体理论可以看下PBRT-v3的解析。

1.2 CPU 端的 KD Tree

可参考nanoflann，这是一个头文件库，实现简单，性能也很好。

2 Hash Grid

2.1 基本概念

Hash Grid 是一种用于快速查询场景数据的数据结构，它的原理是把空间划分为多个格子，每个格子存储场景数据（如 GI 等），通过 hash 函数计算索引，根据索引获取格子中的数据。

优点：存储空间小，查询效率高；不用定义实际的 3D voxel，而是将每个三维坐标映射到 hash index，然后根据 hash index 获取数据。

2.2 SHARC

SHARC（Spatially Hashed Radiance Cache）是 NVIDIA 提出的一种空间 hash 算法，主要应用于路径追踪，便于计算在每个 hit point 处的 radiance。

2.2.1 Hash Grid 设计

Hash Grid 初始化参数 HashGridParameters：

（1）cameraPosition：相机位置

（2）logarithmBase：控制 voxel 的分布密度，相邻 level 的 voxel 相对大小

（3）sceneScale：控制 voxel 的 size

（4）levelBias：选择 level 时的 bias，避免在不必要的时候选择了细粒度的 level

2.2.2 hash index 和坐标转换

（1）将 sample 的 world position 转化为 hash index

（1.1）选择存储的 level

```
float HashGridLogBase(float x, float base)
{
    return log(x) / log(base);
}

uint HashGridGetLevel(float3 samplePosition, HashGridParameters gridParameters)
{
    const float distance2 = dot(gridParameters.cameraPosition - samplePosition,
            gridParameters.cameraPosition - samplePosition);

    return uint(clamp(0.5f * HashGridLogBase(distance2, gridParameters.logarithmBase)
        + gridParameters.levelBias, 1.0f, float(HASH_GRID_LEVEL_BIT_MASK)));
}
```

（1.2）计算 level 对应的 voxel size

```
float HashGridGetVoxelSize(uint gridLevel, HashGridParameters gridParameters)
{
    return pow(gridParameters.logarithmBase, gridLevel) /
        (gridParameters.sceneScale * pow(gridParameters.logarithmBase,
        gridParameters.levelBias));
}
```

（1.3）计算 HashGridKey

HashGridKey 使用的是 64 位无符号整型，其中用分别用三个 17bit 存储 sample 对应的 voxel 坐标，10 个 bit 存储 level；如果使用了法线，用 3 个 bit 存储法线方向。

```
int4 HashGridCalculatePositionLog(float3 samplePosition, HashGridParameters gridParameter
{
    samplePosition += float3(HASH_GRID_POSITION_BIAS, HASH_GRID_POSITION_BIAS, HASH_GRID_

    uint   gridLevel     = HashGridGetLevel(samplePosition, gridParameters);
    float  voxelSize     = HashGridGetVoxelSize(gridLevel, gridParameters);
    int3   gridPosition  = int3(floor(samplePosition / voxelSize));

    return int4(gridPosition.xyz, gridLevel);
}

HashGridKey HashGridComputeSpatialHash(float3 samplePosition, float3 sampleNormal, HashG
{
    uint4 gridPosition = uint4(HashGridCalculatePositionLog(samplePosition, gridParameter

    HashGridKey hashKey =
        ((uint64_t(gridPosition.x) & HASH_GRID_POSITION_BIT_MASK)
            << (HASH_GRID_POSITION_BIT_NUM * 0)) |
        ((uint64_t(gridPosition.y) & HASH_GRID_POSITION_BIT_MASK)
            << (HASH_GRID_POSITION_BIT_NUM * 1)) |
        ((uint64_t(gridPosition.z) & HASH_GRID_POSITION_BIT_MASK)
            << (HASH_GRID_POSITION_BIT_NUM * 2)) |
        ((uint64_t(gridPosition.w) & HASH_GRID_LEVEL_BIT_MASK)
            << (HASH_GRID_POSITION_BIT_NUM * 3));

#if HASH_GRID_USE_NORMALS
    uint normalBits =
        (sampleNormal.x + HASH_GRID_NORMAL_BIAS >= 0 ? 0 : 1) +
        (sampleNormal.y + HASH_GRID_NORMAL_BIAS >= 0 ? 0 : 2) +
        (sampleNormal.z + HASH_GRID_NORMAL_BIAS >= 0 ? 0 : 4);

    hashKey |= (uint64_t(normalBits)
       << (HASH_GRID_POSITION_BIT_NUM * 3 + HASH_GRID_LEVEL_BIT_NUM));
#endif // HASH_GRID_USE_NORMALS

```

```
    return hashKey;
}
```

（2）将 HashGridKey 转化为坐标

```
float3 HashGridGetPositionFromKey(const HashGridKey hashKey,
    HashGridParameters gridParameters)
{
    const int signBit   = 1 << (HASH_GRID_POSITION_BIT_NUM - 1);
    const int signMask  = ~((1 << HASH_GRID_POSITION_BIT_NUM) - 1);

    int3 gridPosition;
    gridPosition.x = int((hashKey >> (HASH_GRID_POSITION_BIT_NUM * 0))
            & HASH_GRID_POSITION_BIT_MASK);
    gridPosition.y = int((hashKey >> (HASH_GRID_POSITION_BIT_NUM * 1))
            & HASH_GRID_POSITION_BIT_MASK);
    gridPosition.z = int((hashKey >> (HASH_GRID_POSITION_BIT_NUM * 2))
            & HASH_GRID_POSITION_BIT_MASK);

    // Fix negative coordinates
    gridPosition.x = (gridPosition.x & signBit) != 0 ?
        gridPosition.x | signMask : gridPosition.x;
    gridPosition.y = (gridPosition.y & signBit) != 0 ?
        gridPosition.y | signMask : gridPosition.y;
    gridPosition.z = (gridPosition.z & signBit) != 0 ?
        gridPosition.z | signMask : gridPosition.z;

    uint   gridLevel = uint((hashKey >> HASH_GRID_POSITION_BIT_NUM * 3)
        & HASH_GRID_LEVEL_BIT_MASK);
    float  voxelSize = HashGridGetVoxelSize(gridLevel, gridParameters);
    float3 samplePosition = (gridPosition + 0.5f) * voxelSize;

    return samplePosition;
}
```

（3）计算 base slot

实际分配的 hash grid 的容量为 capacity，base slot 当前 HashGridKey 在 hash grid 对应的索引。

```
float HashGridLogBase(float x, float base)
{
```

```
    return log(x) / log(base);
}

// http://burtleburtle.net/bob/hash/integer.html
uint HashGridHashJenkins32(uint a)
{
    a = (a + 0x7ed55d16) + (a << 12);
    a = (a ^ 0xc761c23c) ^ (a >> 19);
    a = (a + 0x165667b1) + (a << 5);
    a = (a + 0xd3a2646c) ^ (a << 9);
    a = (a + 0xfd7046c5) + (a << 3);
    a = (a ^ 0xb55a4f09) ^ (a >> 16);

    return a;
}

uint HashGridHash32(HashGridKey hashKey)
{
    return HashGridHashJenkins32(uint((hashKey >> 0) & 0xFFFFFFFF))
        ^ HashGridHashJenkins32(uint((hashKey >> 32) & 0xFFFFFFFF));
}

uint HashGridGetBaseSlot(const HashGridKey hashKey, uint capacity)
{
    uint hash = HashGridHash32(hashKey);
    uint slot = hash % capacity;

    return min(slot, capacity - HASH_GRID_HASH_MAP_BUCKET_SIZE);
}
```

（4）在 Hash Grid 内进行查找

以 $HASH\_GRID\_HASH\_MAP\_BUCKET\_SIZE$ 为查找范围，在 hash grid 为当前的 HashGridKey 找到合适的位置，没找到则返回 false。

```
bool HashMapFind(in HashMapData hashMapData, const HashGridKey hashKey,
    inout HashGridIndex cacheIndex, out uint bucketOffset)
{
    const uint baseSlot = HashGridGetBaseSlot(hashKey, hashMapData.capacity);
    for (bucketOffset = 0; bucketOffset < HASH_GRID_HASH_MAP_BUCKET_SIZE;
        ++bucketOffset)
```

```
    {
        HashGridKey storedHashKey = BUFFER_AT_OFFSET(hashMapData.hashEntriesBuffer,
            baseSlot + bucketOffset);

        if (storedHashKey == hashKey)
        {
            cacheIndex = baseSlot + bucketOffset;
            return true;
        }
    }

    return false;
}

HashGridIndex HashMapFindEntry(in HashMapData hashMapData, float3 samplePosition,
    float3 sampleNormal, HashGridParameters gridParameters)
{
    HashGridIndex cacheIndex = HASH_GRID_INVALID_CACHE_INDEX;
    const HashGridKey hashKey = HashGridComputeSpatialHash(samplePosition,
        sampleNormal, gridParameters);
    uint hashCollisionsNum;
    bool successful = HashMapFind(hashMapData, hashKey, cacheIndex, hashCollisionsNum);

    return cacheIndex;
}
```

（5）在 Hash Grid 内进行插入

```
// 平台支持64位原子操作
void HashMapAtomicCompareExchange(in HashMapData hashMapData, in uint dstOffset,
    in uint64_t compareValue, in uint64_t value, out uint64_t originalValue)
{
#if SHARC_ENABLE_GLSL
    // GLSL对应atomicCompSwap
    originalValue = InterlockedCompareExchange(
        BUFFER_AT_OFFSET(hashMapData.hashEntriesBuffer, dstOffset), compareValue, value);
#else // !SHARC_ENABLE_GLSL
    // HLSL
    InterlockedCompareExchange(BUFFER_AT_OFFSET(hashMapData.hashEntriesBuffer, dstOffset)
        compareValue, value, originalValue);
```

```
#endif // !SHARC_ENABLE_GLSL
}

// 平台不支持64位原子操作
void HashMapAtomicCompareExchange(in HashMapData hashMapData, in uint dstOffset,
    in uint64_t compareValue, in uint64_t value, out uint64_t originalValue)
{
    // ANY rearangments to the code below lead to device hang if fuse is unlimited
    const uint cLock = 0xAAAAAAAA;
    uint fuse = 0;
    const uint fuseLength = 8;
    bool busy = true;
    while (busy && fuse < fuseLength)
    {
        uint state;
        InterlockedExchange(hashMapData.lockBuffer[dstOffset], cLock, state);
        busy = state != 0;

        if (state != cLock)
        {
            originalValue = BUFFER_AT_OFFSET(hashMapData.hashEntriesBuffer, dstOffset);
            if (originalValue == compareValue)
                BUFFER_AT_OFFSET(hashMapData.hashEntriesBuffer, dstOffset) = value;
            InterlockedExchange(hashMapData.lockBuffer[dstOffset], state, fuse);
            fuse = fuseLength;
        }
        ++fuse;
    }
}

bool HashMapInsert(in HashMapData hashMapData, const HashGridKey hashKey,
    out HashGridIndex cacheIndex)
{
    const uint baseSlot = HashGridGetBaseSlot(hashKey, hashMapData.capacity);
    for (uint bucketOffset = 0; bucketOffset < HASH_GRID_HASH_MAP_BUCKET_SIZE;
            ++bucketOffset)
    {
        HashGridKey prevHashGridKey;
        HashMapAtomicCompareExchange(hashMapData, baseSlot + bucketOffset,
```

```
            HASH_GRID_INVALID_HASH_KEY, hashKey, prevHashGridKey);

        if (prevHashGridKey == HASH_GRID_INVALID_HASH_KEY || prevHashGridKey == hashKey)
        {
            cacheIndex = baseSlot + bucketOffset;
            return true;
        }
    }

    cacheIndex = hashMapData.capacity - 1;

    return false;
}
```

2.2.3 数据存储

（1）GI 数据，可以考虑用球谐函数

（2）每个 voxel 内部样本数量，新样本如何和旧样本进行混合

（3）帧间数据复用

2.2.4 参考

NVDIA-RTX/SHARC

2.3 个人应用记录

本人在离线渲染引擎的开发工作中使用过 Hash Grid，主要应用于两个方面：（1）GI 信息存储；（2）Photon Map 存储。能满足上线需求，主要问题在于：

（1）处理 hash 冲突；

（2）需要存储法线，减少 artifact；

（3）原子操作耗时。