LearnOptix 系列 6-变换节点与实例化

Dezeming Family

2023 年 4 月 28 日

DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 DezemingFamily 的系列书，可以从我们的网站 [https://dezeming.top/] 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

目录

一　图节点的基本知识点

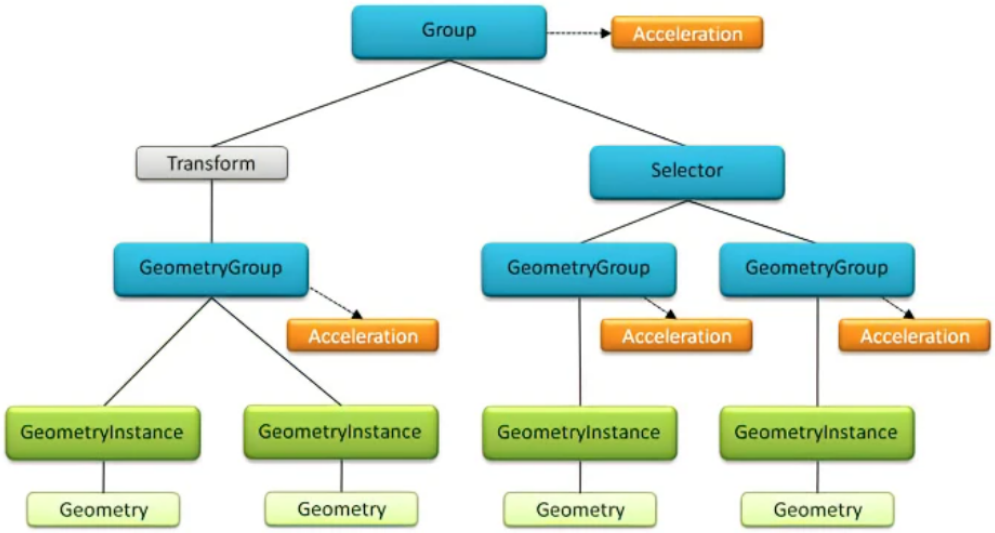
1 1　图节点基本概念

当使用 rtTrace 函数从程序跟踪光线时，会给出一个指定图的根 (root of the graph) 的节点。主机应用程序通过组装 OptiX API 提供的各种类型的节点来创建此图。图的基本结构是层次结构，底部是描述几何对象的节点，顶部是对象的集合 (collections)。

图结构并不意味着是经典意义上的场景图 (scene graph)。相反，它是将不同的程序或动作绑定到场景部分的一种方式。由于每次调用 rtTrace 都指定一个根节点，因此可以使用不同的树或子树。例如，阴影对象或反射对象可能会使用不同的表示——用于特殊的表现或艺术效果。

图节点是通过 rt*Create 调用创建的，该调用将 Context 作为参数。由于这些图节点对象由 context 所有，而不是由其在图中的父节点所有，因此对 rt*Destroy 的调用将删除该对象的变量 (variable)，但不会进行任何引用计数或自动释放其子节点。

下图显示了一个图可能是什么样子的示例。以下部分将介绍各个节点类型。



下表指示了哪些节点可以是其他节点的子节点，包括与加速度结构和材质节点的关联。

Parent node type	*Child node types*	*Associated node types*
Geometry	AccelerationStructure	*none* Material
GeometryInstance	Geometry GeometryTriangles	Material
GeometryGroup	GeometryInstance	AccelerationStructure
TransformNode	GeometryGroup TransformNode	*node*
GroupNode	GeometryGroup	AccelerationStructure

先解释一下上面的图。最顶层的 Group 一定要附加一个加速器结构（哪怕是不定义加速器。也需要”NoAccel” 设定）：

```
1 rtGeometryGroupSetAcceleration( m_geometrygroup, acceleration->get() )
```

GeometryInstance 最重要的意义在于将几何和材质绑定。创建和绑定用下面的函数：

```
1 // 创建GeometryInstance
2 rtGeometryInstanceCreate
3 // 设置几何
4 rtGeometryInstanceSetGeometry
5 // 设置材质
6 rtGeometryInstanceSetMaterial
```

后面介绍的内容很多我们之前早就已经使用和了解过了，这里会提供更多细节上的介绍，我们也会给出一些 optixu 的实现。

1 2 Geometry

Geometry 节点是描述几何对象的基本节点：一组用户定义的基元，光线可以与这些基元相交。几何体节点中包含的基元数是使用下面函数指定的：

```
rtGeometrySetPrimitiveCount
// optixu函数:
GeometryObj::setPrimitiveCount()
```

要定义基元，将使用下面程序为几何体节点指定一个相交程序：

```
rtGeometrySetIntersectionProgram
// optixu函数:
GeometryObj::setIntersectionProgram()
```

求交程序的输入参数是基元索引和光线，程序的工作是返回两者之间的交点。

与程序变量相结合，这提供了必要的机制来定义可以与射线相交的任何基元类型。一个常见的例子是三角形网格，其中相交程序从 buffer 读取三角形的顶点数据（根据基元在 buffer 中的索引读取顶点，通过变量传递给程序），并执行射线三角形求交。

为了在任意几何体上建立加速结构，OptiX 有必要查询单个基元的边界。因此，必须使用下面的函数提供一个单独的边界程序：

```
rtGeometrySetBoundingBoxProgram
// optixu
GeometryObj::setBoundingBoxProgram()
```

该程序只需计算所需图元的边界框，然后 OptiX 将其用作加速结构构建的基础。

以下示例显示如何使用单个基元构造描述球体的几何体对象。假设相交和边界框程序取决于指定球体半径的单个参数变量：

```
RTgeometry geometry;
RTvariable variable;
rtGeometryCreate( context, &geometry );
rtGeometrySetPrimitiveCount( geometry, 1 );
rtGeometrySetIntersectionProgram(
    geometry, sphere_intersection );
rtGeometrySetBoundingBoxProgram(
    geometry, sphere_bounds );
// Declare and set the radius variable.
rtGeometryDeclareVariable(
    geometry, "radius", &variable );
rtVariableSet1f( variable, 10.0f );
```

1 3 GeometryTriangles

GeometryTriangles 是三角形基元的一种特殊类型，它提供了比自定义基元（Geometry）更高效的内置求交程序。特别是，GeometryTriangles 能够充分利用图灵体系结构引入的 RT 核的光线跟踪硬件支持。我们会在下一本小书中介绍三角形。

1 4 Material

材质封装了当光线与与给定材质关联的基元相交时所采取的操作。此类操作包括：计算反射颜色、跟踪附加光线、忽略交点和终止光线。通过声明程序变量，可以向材材质提供任意参数。

两种类型的程序可以分配给一个材质，closest-hit programs 和 any-hit programs。这两种类型在执行的时间和频率上有所不同。closest-hit programs 类似于经典渲染系统中的着色器，对于光线与场景的最近交点，每条光线最多执行一次该程序（注意，继续跟踪的附加光线不再属于这条光线）。它通常执行涉及纹理查找、反射率颜色计算、光源采样、递归光线跟踪等操作，并将结果存储在 ray payload 数据结构中。

对于光线遍历过程中发现的每个潜在的最近交点，都会执行 any-hit program。执行程序的交点可能不会沿着光线排序，但如果需要，最终可以枚举光线与场景的所有交点（通过对每个交点调用 rtIgnoreIntersection，则所有的交点都可以被依次访问到，而不会像之前计算阴影那样，让阴影光线被遮挡时直接终止）。any-hit program 的典型用途包括早期终止阴影光线（使用 rtTerminateRay）和二进制透明度 (binary transparency)，例如，通过基于纹理查找忽略交点。

需要注意的是，这两种类型的程序都指定给每种光线类型的材质，这意味着每种材质实际上可以容纳多个 closest-hit programs 和 any-hit programs。如果应用程序能够识别出某种射线仅执行特定操作，则这一点非常有用。例如，单独的光线类型可以用于阴影光线，阴影光线仅用于确定场景中两点之间的二元可见性。在这种情况下，附加到该光线类型索引下的所有材质的简单 any-hit program 可以立即终止这些光线，并且可以完全省略 closest-hit program。这一概念允许对单个光线类型进行高效的专门化。

通过调用下面的函数将 closest-hit program 和 any-hit program 分配给材质，如果省略了，则默认为空程序：

```
1 rtMaterialSetClosestHitProgram
2 rtMaterialSetAnyHitProgram
```

1 5 GeometryInstance

几何体实例表示单个几何体/几何体三角形节点 (geometry triangles node) 与一组材质的耦合。实例引用的几何体对象是使用下面的函数指定的：

```
1 rtGeometryInstanceSetGeometry
2 rtGeometryInstanceSetGeometryTriangles
```

与实例关联的材质数量由下面的函数设置：

```
1 rtGeometryInstanceSetMaterialCount
```

单个材质由函数指定：

```
1 rtGeometrityInstanceSetMaterial
```

必须指定给几何图元实例的材质数量由参照几何图元的相交程序可能报告的最高材质索引确定，报告材质索引的函数我们早就遇到过很多次了：

```
1 static __device__ bool rtReportIntersection (unsigned int material)
```

特殊规则适用于三角形，这些规则在三角形的“Multi-materials”中有详细说明，本文暂不介绍。

注意，允许多个几何体实例引用单个几何体对象，从而可以实例化具有不同材质的几何体对象。同样，材质可以在不同的几何体实例之间重复使用。

此示例配置了一个几何体实例，使其第一个材质索引为 mat_phong，第二个材质索引是 mat_diffuse，这两个对象都被假定为已分配适当程序的 RTmaterial 对象。该实例是指 RTgeometry 对象 quad_mesh:

```
1 RTgeometryinstance ginst;
2 rtGeometryInstanceCreate( context, &ginst );
3 rtGeometryInstanceSetGeometry( ginst, quad_mesh );
4 rtGeometryInstanceSetMaterialCount( ginst, 2 );
5 rtGeometryInstanceSetMaterial( ginst, 0, mat_phong );
6 rtGeometryInstanceSetMaterial( ginst, 1, mat_diffuse);
```

1 6 GeometryGroup

几何体组 GeometryGroup 是用于任意数量的几何体实例的容器。使用函数设置包含的几何体实例的数量：

```
rtGeometryGroupSetChildCount
// optixu的函数：
GeometryGroupObj::setChildCount()
```

并使用函数分配实例：

```
rtGeometryGroupSetChild
// optixu的函数
GeometryGroupObj::setChild()
```

还必须为每个几何体组分配一个加速结构：

```
rtGeometryGroupSetAcceleration
```

几何体组的最小示例用例是只为其分配一个几何体实例：

```
RTgeometrygroup geomgroup;
rtGeometryGroupCreate( context, &geomgroup );
rtGeometryGroupSetChildCount( geomgroup, 1 );
rtGeometryGroupSetChild( geomgroup, 0, geometry_instance );
```

允许多个几何体组共享子对象，也就是说，一个几何图形实例可以是多个几何体组的子对象。

1 7 Group

组 Group 表示图形中较高级别节点的集合。它们用于编译最终传递给 rtTrace 与射线相交的图结构 (graph structure)（我们以前定义的最高节点是 GeometryGroup 对象，而不是一个 Group 对象）。组可以包含任意数量的子节点，这些子节点本身必须是 RTgroup、RTgeometrygroup、RTtransform 或 RTselector 类型。

组中的子级数量由函数设置：

```
rtGroupSetChildCount
```

单个子级使用函数分配：

```
rtGroupSetChild
```

每个组还必须分配一个加速结构：

```
rtGroupSetAcceleration
```

组的一个常见用例是收集几个相对于彼此动态移动的几何图形组。单个位置、旋转和缩放参数可以由 TransformNode 表示，因此在调用 rtContextLaunch 之间需要重建的唯一加速结构是顶级组的加速结构。这通常比更新整个场景的加速结构计算量少得多。

注意，一个组的子节点可以与其他组共享，也就是说，每个子节点也可以是另一个组（或其为有效子节点的任何其他图节点）的子节点。这允许非常灵活和轻量级的实例化场景，尤其是与共享加速结构 (shared acceleration structures)（第二节会介绍）相结合。

我们可以把之前写的程序改为最高节点是 Group 的形式：

```
Group group = context->createGroup();
group->setChildCount(1);
```

```
group->setChild(0, geometrygroup);
group->setAcceleration(context->createAcceleration("NoAccel"));

context["top_object"]->set(group);
context["top_shadower"]->set(group);
```

1 8 TransformNode

TransformNode 用于表示其基础场景几何体的投影变换。必须使用如下函数为转换分配一个类型为 RTgroup、RTgeometrygroup、RTtransform 或 RTselector 的子级。

```
rtTransformSetChild
```

也就是说，变换下面的节点可能只 geometry group 形式的几何体，或者场景的全新子图。

转换本身是通过将 4×4 浮点矩阵（指定为 16 元素一维数组）传递给下面的函数来指定的：

```
rtTransformSetMatrix
```

从概念上讲，可以看到矩阵被应用于所有底层几何体。但是，这种效果是通过在遍历过程中变换光线本身来实现的。这意味着 OptiX 在 TransformNode 发生变化时不会重建任何加速结构。

此示例显示如何创建具有简单平移矩阵的 TransformNode 对象：

```
RTtransform transform;
// Matrices are row-major.
const float x=10.0f, y=20.0f, z=30.0f;
const float m[16] = {
    1, 0, 0, x,
    0, 1, 0, y,
    0, 0, 1, z,
    0, 0, 0, 1
};
rtTransformCreate(context, &transform);
rtTransformSetMatrix(transform, 0, m, 0);
```

注意，TransformNode 的子节点可以与其他图形节点共享。也就是说，TransformNode 的子节点可以同时是另一个节点的子节点。这对于实例化几何体通常很有用。TransformNode 应谨慎使用，因为它们在光线跟踪过程中会降低性能。特别是，强烈建议节点图 (node graphs) 不要超过单个级别的变换深度。也就是说，一个场景的图结构中，最好是最多只有一层变换节点，比如一堆变换节点同时作用于一个草，实现草的实例化，但是在草的父节点（以及更高节点）和子节点（以及更低节点）就不要再有变换节点了，保证变换节点只有一层。

1 9 Selector

SelectorNode 与组 Group 的相似之处在于它是更高级别的图节点的集合。

集合中的节点数由函数设置：

```
rtSelectorSetChildCount
```

各个子节点由函数分配：

```
rtSelectorSetChild
```

有效的子类型为 RTgroup、RTgeometrygroup、RTtransform 和 RTselector。

SelectorNode 和组之间的主要区别在于，SelectorNode 没有关联的加速结构。与此相反，使用下面的指定 visit 程序：

```
1 rtSelectorSetVisitProgram
```

每当光线在图形遍历过程中遇到 SelectorNode 时，都会执行下面的程序。该程序通过调用下面的函数来指定光线应继续遍历的子对象：

```
1 rtIntersectChild
```

SelectorNode 的典型用例是动态（每光线 (per-ray)）细节级别 (level of detail)：场景中的对象可以由多个几何体节点表示，每个节点都包含对象的不同细节级别版本。包含这些不同表示的几何图形组可以指定为 SelectorNode 的子对象。访问程序可以使用任何标准（例如，基于当前光线的足迹（footprint，与光线微分和光线多次反射有关）或长度（光线前进的 t 值））选择要相交的子对象，并忽略其他子对象。对于组和其他图节点，SelectorNode 的子节点可以与其他图节点共享，以实现灵活的实例化。

注意，RTX 模式中不赞成使用选择器节点 (OptiX 6.0 的默认设置)。

二　加速结构

加速结构是加速光线跟踪的遍历和交点查询的重要工具，尤其是对于大型场景。大多数成功的加速结构表示场景几何体的层次分解。然后，该层次用于快速剔除未与光线相交的空间区域。有不同类型的加速结构，每种结构都有各自的优点和缺点。此外，不同的场景需要不同类型的加速结构来获得最佳性能（例如，静态和动态场景之间的差异、通用基元和三角形，等等）。最常见的权衡是在构建速度和光线跟踪性能之间进行，但内存消耗等其他因素也可能起到一定作用。没有一种单一类型的加速结构对所有场景都是最佳的。

为了使应用程序能够平衡权衡，OptiX 支持在几种结构之间进行选择，甚至可以在同一节点图中混合和匹配不同类型的加速结构（只要正确提供包围盒即可）。

2 1　Acceleration objects in the node graph

加速结构是 OptiX 中的 API 对象，称为 RTacceleration。创建加速对象的函数是：

```
rtAccelerationCreate
// optixu的函数：
ContextObj::createAcceleration
```

它将通过函数被指定给一个组或一个几何体组：

```
rtGroupSetAcceleration
RTGeometryGroupSetAcceluration
// optixu的函数：
GroupObj::setAcceleration
GeometryGroupObj::setAcceleration
```

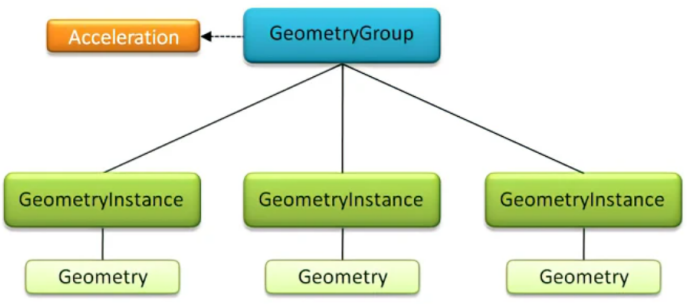
节点图中的每个组和几何体组都需要为光线遍历指定一个加速对象（也就是说如果有组或者几何体组没有指定加速结构，那么就无法正确运行），以与这些节点相交。

本例创建了一个几何体组和一个加速结构，并将两者连接起来：

```
RTgeometrygroup geomgroup;
RTacceleration accel;
rtGeometryGroupCreate( context, &geomgroup );
rtAccelerationCreate( context, &accel );
rtGeometryGroupSetAcceleration( geomgroup, accel );
```

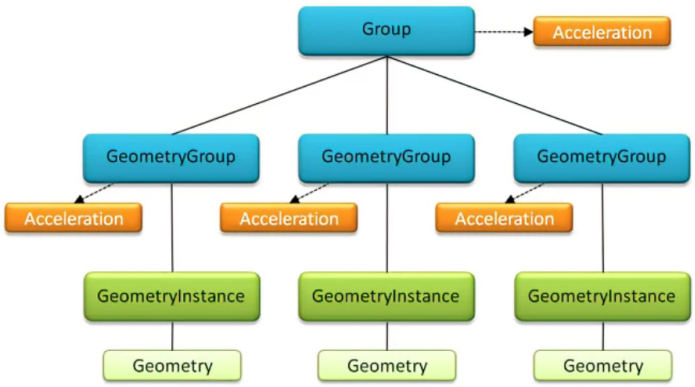
通过在组装节点图时使用组和几何体组，应用程序可以高度控制如何在场景几何体上构建加速度结构。如果考虑场景中多个几何体实例的情况，有多种方法可以将它们放置在组或几何体组中，以适应应用程序的用例。

例如，下图将所有几何图元实例放置在一个几何图元组中。几何体组上的加速结构将在所有几何体的集合上构建：



管理多个几何体实例的不同方法如下图所示。每个实例都放置在其自己的几何体组中，因此每个实例都有一个单独的加速结构。生成的几何图形组集合被聚合到一个顶级组中，该组本身具有加速结构。组上

的加速结构是在子节点的边界体上构建的。因为子节点的数量通常相对较低，所以高层结构通常可以快速更新（一个子节点改变时，不会影响其他子节点，且其父节点下的子节点数量一般不会很高）。这种方法的优点是，当修改其中一个几何实例时，不需要重建其他实例的加速结构。然而，由于更高级别的加速度结构引入了额外的复杂性，并且仅建立在其组的子对象的粗略边界上，因此可能不如前面那样高效。同样，在这种情况下，应用程序需要通过考虑修改单个几何体实例的频率来平衡这一权衡。



2 2 Acceleration structure builders

RTacceleration 有一个生成器。生成器负责收集输入几何体（在大多数情况下，该几何体是由几何体节点的边界框程序创建的边界框），并计算允许加速光线场景求交查询的数据结构。生成器不是应用程序定义的程序。相反，应用程序从下面列表中（OptiX 目前可用的生成器）选择一个合适的生成器。生成器可以随时更换；切换生成器将导致加速结构被标记 (flagged) 以进行重建。

- **Trbvh:** 参考论文【Fast Parallel Construction of High-Quality Bounding Volume Hierarchies】。Trbvh 生成器执行非常快速的基于 GPU 的 BVH 构建。它的光线跟踪性能通常在 SBVH 的百分之几以内，但它的构建时间通常是最快的。对于所有数据集，都应强烈考虑此生成器。除了最终 BVH 所需的内存之外，Trbvh 使用了少量的额外内存。当 GPU 上没有额外的内存时，Trbvh 可能会自动回退到 CPU 上构建。
- **Sbvh:** Split-BVH(SBVH) 是一种高质量的边界体层次。虽然构建时间最高，但由于其高光线跟踪性能，它传统上是静态几何体的首选方法，但可能会被 Trbvh 取代。如果几何体是非均匀 (non-uniform) 的（例如，在不同大小的三角形集合中），则对规则 BVH 的改进尤其明显。该生成器可用于任何类型的几何体，但为了获得三角形几何体的最佳性能，应设置专门的属性（后面会有描述）
- **Bvh:** 参考论文【Spatial Splits in Bounding Volume Hierarchies】。Bvh 生成器构建了一个经典的边界体层次。它具有相对较好的遍历性能，不注重快速构建性能，但它支持为快速增量更新 (fast incremental updates) 进行改造 (refitting)（后面会有描述）。Bvh 通常是建立在组上的加速结构的最佳选择。
- **NoAccel:** 这是一个虚拟生成器，它不构建实际的加速结构。遍历在所有元素上循环，并将每个元素与射线求交。除了非常简单的情况外，这在任何情况下都是非常低效的，但有时会优于构建的加速结构，例如，在具有很少的子节点的组上。

一个简单的实例代码：

```
1 RTacceleration accel;
2 rtAccelerationCreate( context, &accel );
3 rtAccelerationSetBuilder( accel, "Trbvh" );
```

2 3 Acceleration structure properties

根据情况，微调 (Fine-tuning) 加速结构可能是有用的。为此，生成器公开了各种命名属性（有很多项是与三角形有关的，为了更全面，这里都列出了。在后面讲解三角形的内容时会重新提到这里的内容）：

- **属性：refit，可用结构：Bvh Trbvh。**如果设置为 1，则生成器将仅重新调整边界体层次的节点边界，而不是从头开始构建。只有当初始 BVH 已经构建好，并且基础几何形状发生了相对适度的变形时，refit 才有效。在这种情况下，生成器可以在不牺牲太多光线跟踪性能的情况下提供非常快速的 BVH 更新。默认值为 0。

- **属性：vertex_buffer_name，可用结构：Bvh Trbvh。**存储三角形顶点数据的缓冲区变量的名称。每个顶点由 3 个浮点组成。对于 Sbvh 是可选的（但如果几何图形由三角形组成，则建议使用）。默认值为 vertex_buffer。

- **属性：vertex_buffer_stride，可用结构：Bvh Trbvh。**顶点缓冲区中两个顶点之间的偏移量，以字节为单位。默认值为 0，假设顶点是紧密堆积的。

- **属性：index_buffer_name，可用结构：Bvh Trbvh。**存储顶点索引数据的缓冲区变量的名称。该缓冲区中的项是 int 类型的索引，其中每个索引都指向顶点缓冲区中一个项。三个索引的序列表示一个三角形。如果没有给出索引缓冲区，则假设顶点缓冲区中的顶点是三角形列表，一行中每三个顶点形成一个三角形。默认值为 index_buffer。

- **属性：index_buffer_stride，可用结构：Bvh Trbvh。**索引缓冲区中两个索引之间的偏移量，以字节为单位。默认值为 0，这假设索引是紧密封装的。

- **属性：chunk_size，可用结构：Trbvh。**用于分区加速结构 (partitioned acceleration structure) 生成的字节数。如果没有设置 chunk size，或者设置为 0，则会自动选择 chunk size。如果设置为-1，则 chunk size 是无限的。当前最小 chunk size 为 64MB。请注意，指定较小的 chunk size 可以减少 Trbvh 的峰值内存占用，但可能会导致渲染性能降低。

- **属性：compact，可用结构：Bvh Trbvh。**如果设置为 1，生成器将压缩加速结构以消耗所需的最小内存。这发生在加速结构建成后，为 compaction 步骤增加了少量时间。如果同时启用 refit 属性，则无法启用 compact 属性。设置为 0 可禁用。默认值为 1。

属性需要使用下面函数指定：

```
1 rtAccelerationSetProperty
```

它们的值以字符串的形式给出，由 OptiX 进行解析。只有在实际重建加速结构时，属性才会生效。设置或更改属性本身并不标记加速结构来重建；有关如何做到这一点的详细信息会在下一小节介绍。生成器无法识别的属性将被忽略。

```
1 // Enable fast refitting on a BVH acceleration
2 rtAccelerationSetProperty( accel, "refit", "1" );
```

2 4 Acceleration structure builds

在 OptiX 中，当需要重建加速结构时，会对其进行标记（标记为“脏 (dirty)”）。在 rtContextLaunch 过程中，所有标记的加速结构都会在光线跟踪开始之前构建。每个新创建的 RTacceleration 对象最初都被标记为 dirty。

应用程序可以随时决定显式标记加速结构以进行重建。例如，如果几何图形组的基本几何图形 (underlying geometry) 发生更改，则必须重新创建附着到几何图形组上的加速结构。这是通过调用如何函数来实现的：

```
rtAccelerationMarkDirty
```

例如，如果将新的子几何体实例添加到几何体组，或从几何体组中删除子几何体，也需要这样做。

组 Group 上的加速结构也是如此：添加或删除子几何图元、更改组下的变换等操作都需要将组的加速标记为脏。根据经验，对构建加速结构的基本几何体 (underlying geometry)（在组的情况下，该几何体是子轴对齐的边界框）进行修改的每个操作都需要重新生成。但是，例如，如果图形的某些部分在 Tree 下进一步更改，而不影响组的直接子级的边界框，则不需要重新生成。

注意对于一个几何图中的每个单个加速结构，应用程序独立地决定是否需要重建。OptiX 不会尝试自动检测变化，并且将一个加速结构标记为 dirty 不会将 dirty 标志传播到任何其他加速结构。如果在必要时不将加速结构标记为 dirty 的，可能会导致意外行为——通常是错过交点或性能下降。

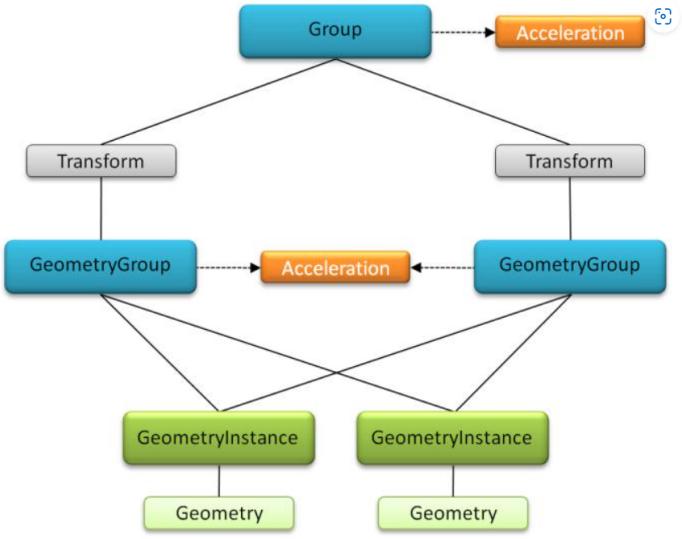
2 5 Shared acceleration structures

诸如将图节点作为子节点附加到多个其他图节点之类的机制使节点图的组成变得灵活，并实现了有趣的实例化应用程序。通过多次引用节点而不是复制节点，实例化可以被视为对场景对象或图形部分的廉价重用。

OptiX 将加速结构解耦为与其他图节点分离的对象。因此，加速结构自然可以在几个组或几何体组之间共享，只要结构所基于的基本几何体 (underlying geometry) 是相同的：

```
// Attach one acceleration to multiple groups.
rtGroupSetAcceleration( group1, accel );
rtGroupSetAcceleration( group2, accel );
rtGroupSetAcceleration( group3, accel );
```

注意应用程序必须确保共享加速结构的每个节点都具有匹配的底层几何体 (underlying geometry)。否则将导致未定义的行为。此外，加速结构不能在组和几何体组之间共享（就是组构建的加速结构，不给共享给几何体组，即使它们的底层几何体都一样也不行）。共享加速结构是一个强大的能力，可以最大限度地提高效率，如下图所示。



图形中心的加速节点附着到两个几何图形组，并且两个几何图元组都引用相同的几何图形对象。这种几何结构和加速结构数据的重用最大限度地减少了内存占用和加速结构的构建时间，可以以同样的方式以非常小的开销添加额外的几何图形组。

三　变换与实例化-代码实现

本节示例代码见 3-1。

我们的场景结构相对复杂一点儿。最顶层是 top_group，它下面有两个子节点，分别是 group 和 geometrygroup2；其中，group 就是一大堆球体构成的实例化，group 下面有 20 个 TransformNode 子节点，这些 TransformNode 子节点都共享同一个几何组 geometrygroup；geometrygroup2 仅仅只有一个地板。

对于 group，构建加速结构为 Trbvh：

```
// define group
Group group = context->createGroup();
group->setChildCount(20);
for (int count = 0; count < 20; ++count) {
    float r1 = (float)rand() / (float)(1 + RAND_MAX);
    float r2 = (float)rand() / (float)(1 + RAND_MAX);
    Transform transform = context->createTransform();
    float m[16] = {
        1.0f, 0.0f, 0.0f, (r1 * 32.0f - 16.0f),
        0.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 1.0f, (r2 * 16.0f - 8.0f),
        0.0f, 0.0f, 0.0f, 1.0f
    };
    transform->setMatrix(false, m, NULL);
    transform->setChild(geometrygroup);
    group->setChild(count, transform);
}
group->setAcceleration(context->createAcceleration("Trbvh"));
```

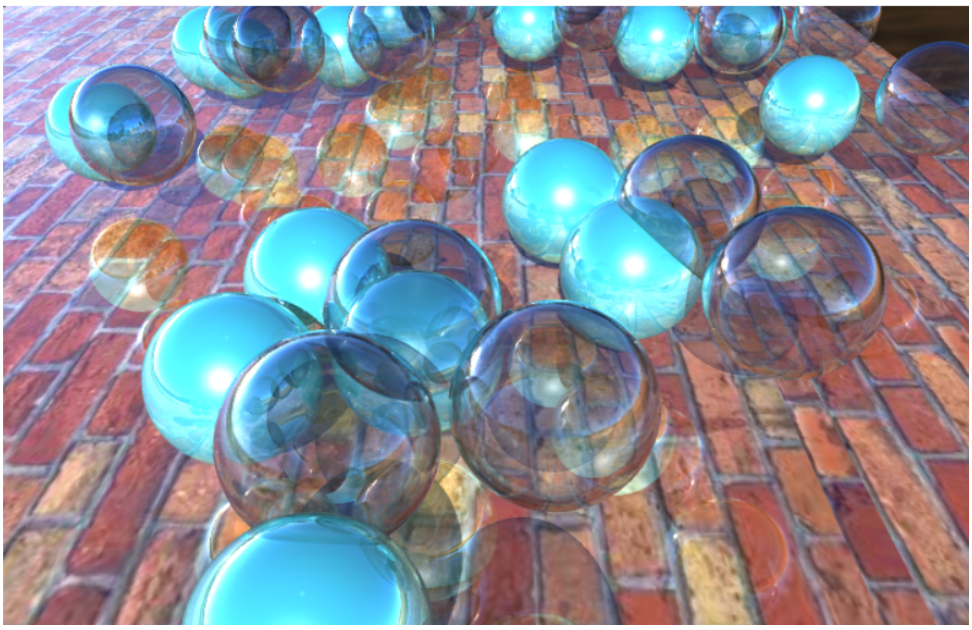
对于 geometrygroup2，不构建加速结构：

```
GeometryInstance gis_floor = context->createGeometryInstance(parallelogram,
   &floor_matl, &floor_matl + 1);
GeometryGroup geometrygroup2 = context->createGeometryGroup();
geometrygroup2->setChildCount(1);
geometrygroup2->setChild(0, gis_floor);
geometrygroup2->setAcceleration(context->createAcceleration("NoAccel"));
```

最顶层的 top_group 不构建加速结构：

```
Group top_group = context->createGroup();
top_group->setChildCount(2);
top_group->setChild(0, group);
top_group->setChild(1, geometrygroup2);
top_group->setAcceleration(context->createAcceleration("NoAccel"));
```

渲染结果如下：



四　小结

本文详细介绍了节点和加速结构的构建，由于前面已经有了一些基础，所以本文的学习也会更加直观和容易理解。

下一本小书将会介绍三角形，介绍完三角形以后，再介绍路径追踪以及 AI 去噪，Optix 教程系列就完结啦！

参考文献

[1] https://developer.nvidia.com/rtx/ray-tracing

[2] https://developer.nvidia.com/rtx/ray-tracing/optix

[3] https://developer.nvidia.com/blog/how-to-get-started-with-optix-7/

[4] https://raytracing-docs.nvidia.com/optix7/index.html

[5] https://raytracing-docs.nvidia.com/optix7/guide/index.html#preface#

[6] https://developer.nvidia.com/designworks/optix/downloads/legacy

[7] https://raytracing-docs.nvidia.com/optix6/guide_6_5/index.html#guide#

[8] https://raytracing-docs.nvidia.com/optix6/api_6_5/index.html

[9] https://raytracing.github.io/books/RayTracingInOneWeekend.html

[10] https://learnopengl.com/Getting-started/Camera