基于稀疏体素有向无环图的光照计算加速结构

袁昱纬[1,2]，全吉成[1,2]，吴 晨[1]，刘 宇[2]，王宏伟[2]

[1]海军航空工程学院电子信息工程系，山东 烟台 264001；

[2]空军航空大学航空航天情报系，吉林 长春 130022

摘要 提出了一种基于稀疏体素有向无环图(SVDAG)的光照计算加速结构。通过自下而上地合并相同节点，将稀疏体素八叉树转化为 SVDAG；同时，利用给定节点的遍历路径和子掩码，消除了空间位置的多义性。针对闭合几何体，基于双层深度图算法可自适应合并位于闭合区域的节点，在保证光照计算性能的前提下，进一步减小了存储开销。提出了一种基于时域相关性的 SVDAG 帧间复用方法，利用动态场景的全部帧构成 SVDAG 加速结构整体，提高了更新速度。实验结果表明，新算法提高了三维场景的绘制效率，在面对高分辨率的动态场景时，仍能获得较高的帧速率。

关键词 光计算；光学数据处理；加速结构；有向无环图；双层深度图；帧间复用

中图分类号 TN29 **文献标识码** A

doi：10.3788/AOS201737.0820001

Illumination-Computation Acceleration Structure Based on Sparse Voxel Directed Acyclic Graph

Yuan Yuwei[1,2], Quan Jicheng[1,2], Wu Chen[1], Liu Yu[2], Wang Hongwei[2]

[1] *Department of Electronic and Information Engineering, Naval Aeronautical and Astronautical University, Yantai, Shandong* 264001, *China*;

[2] *Department of Aeronautic and Astronautic Intelligence, Aviation University of Air Force, Changchun, Jilin* 130022, *China*

Abstract An illumination-computation acceleration structure based on the sparse voxel directed acyclic graph (SVDAG) is proposed. By merging the same nodes from bottom to top, the sparse voxel octree is converted into a SVDAG, and the polysemy of spatial positions can be eliminated by using the traversal paths and the child masks of the given nodes. Aiming at the closed geometry, an algorithm based on the double depth maps can be used to merge adaptively the nodes located in the closed region, which can further reduce the storage cost while the performance of illumination computation is maintained. A inter-frame multiplex method of SVDAG based on the time correlation is proposed in which all frames of the dynamic scene are used to constitute an integral SVDAG acceleration structure, which can improve the update rate. The experimental results indicate that the rendering efficiency of three-dimensional scene based on the new algorithm is enhanced. When a high resolution dynamic scene is conducted, a relatively high frame rate still can be obtained.

Key words optics in computing; optical data processing; acceleration structure; directed acyclic graph; double depth maps; inter-frame multiplexing

OCIS codes 200.4560; 110.1758; 100.2000

1 引 言

空间加速结构通过对三维场景构建某种层次性的数据结构，为光线跟踪、光子映射、阴影体算法等需要进行光照计算的绘制算法提供了空间查找和遍历功能，可加快三维场景的绘制速度，因此其被广泛应用于光

收稿日期：2017-02-24；**收到修改稿日期**：2017-04-27

基金项目：吉林省自然科学基金(20130101069JC)、军内武器装备重点科研基金(KJ2012240)

作者简介：袁昱纬(1988—)，男，博士研究生，主要从事装备仿真与虚拟现实方面的研究。E-mail：yyw57156@hotmail.com

导师简介：全吉成(1960—)，男，博士，教授，博士生导师，主要从事三维可视化方面的研究。

E-mail：jicheng_quan@126.com(通信联系人)

学计算[1]、图像处理[2]和光通信[3]等领域中。但空间加速结构是通过牺牲一定的存储空间来缩短计算时间，因此如何在保证较高加速性能的同时，尽可能地减小存储开销，成为空间加速结构的一个重要研究方向。

典型的空间加速结构包括八叉树[4-5]、KD 树(K-Dimensional Tree)[6]、包围体层次(BVH)[7]、均匀格网[8]等。近年来，体素化的三维场景逐渐被国内外学者采用，基于稀疏体素八叉树(SVO)的加速结构具有较高的光照计算效率[9]，所需的存储空间与场景分辨率成正比。Laine 等[10]提出了高效的稀疏体素八叉树(ESVO)结构，但其存储开销很高，其中约 40%的存储空间用于节点的编码。Crassin 等[11]将 SVO 应用于锥跟踪算法，但是在处理包含材质信息的分辨率为 512 pixel×512 pixel×512 pixel 的场景时，消耗了近 1 GB(Byte，字节)内存。Hornung 等[12]提出了基于 Morton 码的八叉树节点编码，该编码方法能够减少部分指针，但其规则的存储方式会造成相同节点之间的冗余存储。Liu 等[13]提出一种紧凑的 SVO，其节点压缩思路对本文有一定的借鉴意义。

对于加速结构的快速更新，国内外学者大多采用并行计算，尤其是利用基于中央处理器(CPU)多核架构[14]或图形处理器(GPU)[15]来提高算法的实时性。Bittner 等[16]提出一种基于 BVH 加速结构的增量更新并行算法，但是该算法的实现过程较为复杂。李静等[17]提出一种混合结构来实现大规模动态场景的绘制，但其仅针对空节点进行了优化。Ma 等[18]提出的差分树算法可以实现帧间复用，但是每帧内部的加速结构没有进行优化。

本文针对体素化场景的 SVO，通过寻求相同节点并自底向上地进行逐层合并，提出了一种基于稀疏体素有向无环图(SVDAG)的加速结构，减少了相同节点造成的冗余存储；同时，保持了 SVO 的体素遍历和光照计算性能，并通过给定节点的遍历路径和子掩码，消除了 SVDAG 的空间位置多义性。针对闭合几何体内部，基于双层深度图算法将闭合区域的节点与相邻节点进行合并，进一步简化了 SVDAG 加速结构。根据动态场景的时域相关性，提出了一种 SVDAG 帧间复用方法，避免了动态场景的每一帧都完全重构整个加速结构，加快了更新速度，改善了动态场景的绘制性能。

2　基于 SVDAG 的加速结构

2.1　SVO

在构建 SVO 之前，需要将三维场景转化为最接近的体素化形式，以获得体素化后的三维场景表示。SVO 的构建算法与传统八叉树的类似，其深度决定了场景体素化的分辨率。SVO 与传统八叉树的区别在于，传统八叉树的叶子节点表示的是场景模型的几何面片，而 SVO 的叶子节点仅存储了表示该节点是否存在对应模型体素的“1”和“0”。

在大多数场景中，特别是在高分辨率条件下，SVO 的空节点个数一般远多于包含模型体素的节点个数。SVO 的层次格网可以高效地编码场景的空白区域，这是因为可以利用 SVO 的稀疏性，通过子掩码表示空白区域对应的节点，避免了对空白节点的存储。

在 SVO 构建过程中，每个节点的子掩码一共 8 bit，“0”表示子节点为空，“1”表示子节点为实节点。因此，指向空节点的指针可以完全被剔除，仅需要保留指向第一个非空子节点的指针，其子节点可以按照约定顺序(如广度优先顺序)连续地存储在内存空间中，层级内部可采用 Morton 码顺序，也不需要保留指针，从而达到减小存储开销的目的，如图 1 所示。因此，只要知道子掩码和约定的遍历顺序，就可以通过节点的连续存储顺序恢复出 SVO 的完整结构。

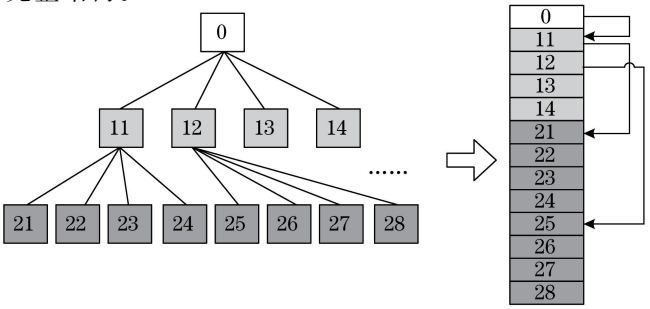

图 1　SVO 节点排列顺序及指针安排的二维示意图

Fig. 1　Two-dimensional schematic of order and pointer setting of nodes in SVO

2.2　稀疏体素八叉树转化为有向无环图

根据 SVO 的构建过程可知，每个节点仅保留了指向第一个非空子节点的指针，因此，确定某个体素的不是指针而是遍历到该体素的路径和路径上的子掩码。即使节点重新排列，只要遍历路径和路径上访问到的子掩码是相同的，仍能获得和原来一样的几何体。因此，根据节点的遍历路径和子掩码，将 SVO 中的相同子节点进行合并，并将指向相同子节点的指针指向同一个节点，仅保留该唯一节点。由于相同节点的合并操作，某些节点可能拥有多个父节点，因此，SVO 就从树形结构扩展为 SVDAG。

按照上述思路，将 SVO 转化为 SVDAG 最简单的方法就是从根节点开始，搜索是否存在完全相同的子树，并继续递归每一个子树，直至遇到叶子节点。这个方法虽然简单，但是递归操作需要将整个场景的所有节点推入堆栈后进行回溯，分辨率较高时内存开销的峰值很大。因此，为了避免递归，提出了如下一种自下而上的方法以实现从 SVO 到 SVDAG 的转换。

1）从 SVO 的底部开始，搜索相同的叶子节点并进行合并，如图 2 所示。在 SVO 中，由于叶子节点仅表示对应空间位置上是否存在对应体素，因此合并后的叶子节点最多两种。

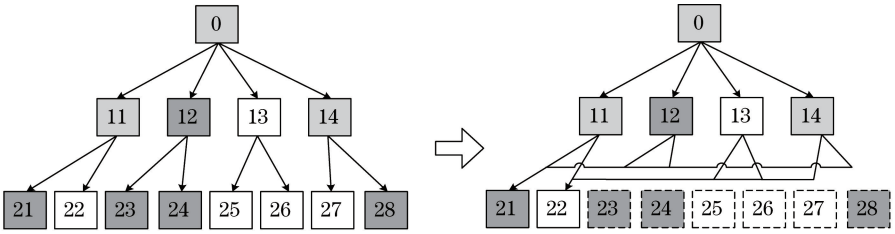

图 2　搜索相同叶子节点并进行合并

Fig. 2　Searching for same leaf nodes and merging

2）继续往上层处理，搜索子掩码和指针完全相同的节点，这样的节点具有相同的子树，可以进行合并，如图 3 所示。需要注意的是，压缩这些层级的节点时，只要考虑子树的根节点，这是因为合并操作是自下而上进行的，节点本身具有相同的子掩码和指针即说明它们的下级子树直至叶子节点都是相同的。

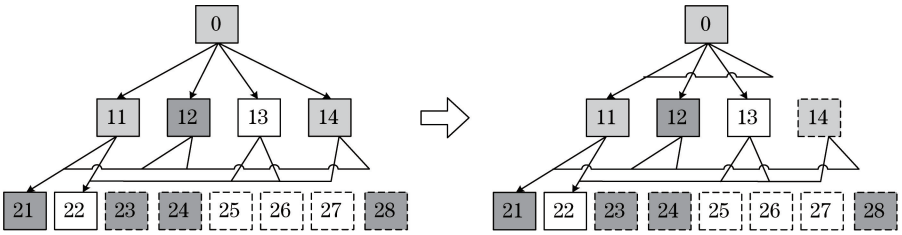

图 3　自下而上地将 SVO 转化为 SVDAG 的示意图

Fig. 3　Schematic of converting SVO to SVDAG from bottom to top

3）自下而上进行搜索和合并操作，参与合并的节点越来越多，直到没有节点可以合并或到达根节点，获得最终的 SVDAG，如图 4 所示。

该算法同样适用于深度不均匀的 SVO，同时也可对不同子树分别进行转化操作然后再组合。由于整个 SVO 的数据规模可能远大于对应的 SVDAG 的数据规模，可以利用这一特性，避免一次性完成整个 SVO 的构建，有效减小了内存需求的峰值。

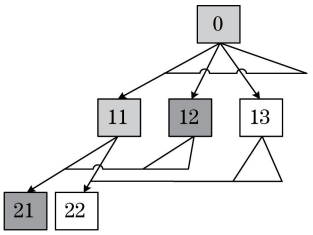

图 4　相同节点合并后的 SVDAG 示意图

Fig. 4　Schematic of SVDAG after merging same nodes

为了在 SVDAG 存储时剔除指向空节点的指针，减小存储资源的开销，可以采用 2.1 节所述的 8 bit 子掩码，按子掩码顺序在内存中仅存储指向非空节点的指针。为了查找方便，子掩码和每个指针都分别占用 4 B，一个节点占用的内存大小为 8～36 B。最小的节点仅包含一个子节点，即存储了一个 8 bit 的子掩码和一个 32 bit 的指针。之所以在 8 bit 的子掩码之前填充空位使其保持与指针一样的字节数，是因为查找时仅需根据线性存储中的索引号，就可以很快计算出所需指针在内存中的存储地址，即所需地址为入口地址加上四倍索引号。

2.3　SVDAG 节点空间位置的多义性

SVO 的节点与空间区域是一一映射的关系，但是由于 SVDAG 的节点合并，一个 SVDAG 节点可能代表空间中若干个不同区域，因此 SVDAG 中的节点所代表的空间位置出现了多义性。因为存在多义性，所以进行基于空间位置的节点搜索是极不方便的。为了消除这个多义性，通过给定遍历到此节点的路径和子掩码确定该节点的空间位置，即遍历到此节点的不同路径和子掩码可以确定不同的空间位置，并与空间位置一一对应。确定给定节点空间位置的方法如下。

1）从根节点开始，依次查找给定的遍历路径上的节点，每个节点相对于其父节点的空间区域由其子掩码确定，节点所在的空间位置范围被逐步缩小。子掩码中各比特代表的空间区域已在构建时约定，如按 Morton 码顺序。

2）继续上一步的操作，直至在遍历路径上访问到给定节点，逐步确定给定节点在空间中的对应位置。

上述过程如图 5 所示，给定的遍历路径为 $A\rightarrow B\rightarrow C$，左图中的数字为对应节点的子掩码，可以恢复出节点 C 在右图所示的 SVO 中的位置，并同时确定节点 C 对应的空间位置。

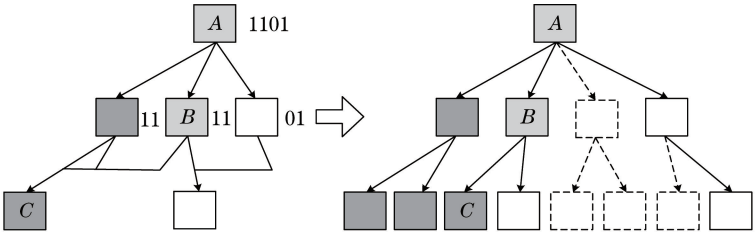

图 5　SVDAG 节点的空间位置确定过程

Fig. 5　Determining process for spatial positions of nodes in SVDAG

3　面向闭合几何体的 SVDAG 结构优化

在三维场景绘制技术中，光照计算主要用于渲染几何体的阴影、材质等效果，本文将基于 SVDAG 的光照计算加速结构应用于三维场景绘制中的阴影计算，基于阴影体算法进一步优化 SVDAG，减小存储开销。

3.1　面向体素化阴影体的加速结构

与三维场景中的几何体体素化类似，在对三维场景进行光照计算并绘制阴影前，需要基于阴影体算法将场景的光线空间体素化，即阴影体的体素化。在对光线空间进行体素化细分时，由于阴影内部都是均匀的阴影空间，因此只对处于阴影体表面的体素进行下一级细分，直至达到所需的分辨率，如图 6(a)所示。对光源可见的区域（即被光源照亮的区域）用“0”表示，如图 6(a)中白色格网所示；对光源不可见的区域（即处于阴影中的区域）用“1”表示，如图 6(a)中灰色格网所示。

按照前面所述的 SVDAG 构建方法，首先对场景的阴影体体素构建 SVO，SVO 结构的大小与阴影体的表面积成正比，然后从叶子节点开始进行自下而上的相同节点的合并操作，无论在哪一级分辨率层次上，加速结构中都仅存储唯一的节点，重复的节点都被舍去，并更新指针，相同的子节点可以被其父节点通过指针和子掩码实现共享，最终完成光线空间中 SVO 向 SVDAG 的转化。光线空间的 SVDAG 加速结构如图 6(b)所示，其中每种颜色代表合并后的一个节点。由于仅需要存储位于阴影体素表面的唯一叶子节点，因此存储这些叶子节点所需的空间比存储整个场景阴影所需的空间小。

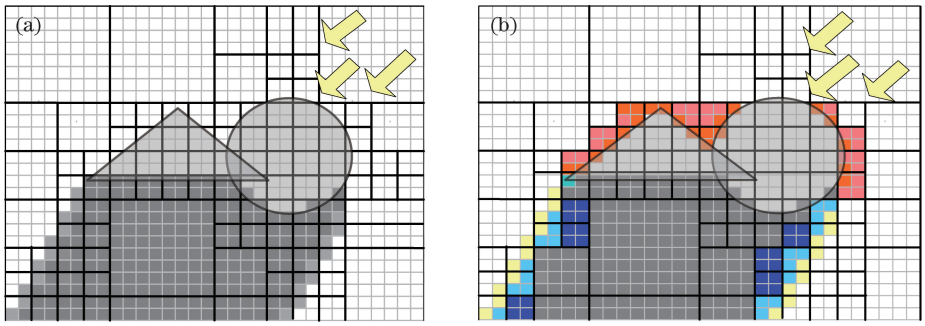

图 6　体素化的(a)光线空间和(b)SVDAG 加速结构

Fig. 6　(a) Space of light rays and (b) SVDAG acceleration structure of voxel

在光线空间的 SVDAG 加速结构中，每一个节点都由子掩码和一组指针组成。关于子掩码的设定，Kämpe 等[19]在将场景几何体表面体素化时，考虑了节点可能与三维模型表面相交的情况。借鉴文献[19]的子掩码表示方式，光线空间 SVDAG 加速结构的节点子掩码包含如下三种情况：1)体素完全位于阴影之内，用“0”表示；2)体素完全位于阴影之外，用“1”表示；3)体素与阴影体边界相交，用“2”表示。因此，面向光线空间 SVDAG 的子掩码需要占用 16 bit，而不是 2.1 节所述的 8 bit。

3.2　基于双层深度图的闭合几何体 SVDAG 的优化

对于绝大多数场景，几何体模型大多是闭合的，观察者或者相机的视角一般不会进入这些几何体模型的内部，因此，建立的 SVDAG 加速结构在这些闭合几何体内部没有价值。针对这些闭合的几何体，提出了一种基于双层深度图的节点自适应合并算法，进一步优化光线空间的 SVDAG 加速结构，使 SVDAG 结构进一步简化，减小其占用的内存空间。

在进行 SVDAG 加速结构的构建时，位于闭合几何体内部的空间区域被认为是未定义的，完全位于未定义空间的节点既可以设置为对光源可见，也可以设置为不可见，究竟如何设置取决于能否获得一个更加简化的 SVDAG。这个面向闭合几何体的 SVDAG 自适应优化过程如下。

1) 基于光源视点，为三维场景建立深度图，其分辨率与 SVDAG 加速结构的分辨率一致，并计算光源到场景的深度值。

2) 在计算深度图的遍历过程中，当光线与几何体相交时，建立第二层深度图，记录光源投射的光线离开几何体表面并再次进入空白区域的情况。光线从空白区域进入闭合区域时深度值为 D_1，存储在第一层深度图中；光线离开闭合区域进入空白区域时深度值为 D_2，第二层深度图记录深度差值(D_2-D_1)，该值表征了闭合区域的深度范围，如图 7(a)所示，其中深绿色线条表示第一层深度图，蓝色线条表示第二层深度图。

3) 基于双层深度图，可以将体素化的光线空间划分为对光源可见、对光源不可见和未定义三种区域，其中未定义区域即为闭合几何体包围的区域，如图 7(b)所示，其中白色区域为光源可见区域、灰色为光源不可见(阴影)区域、浅蓝色为未定义区域。

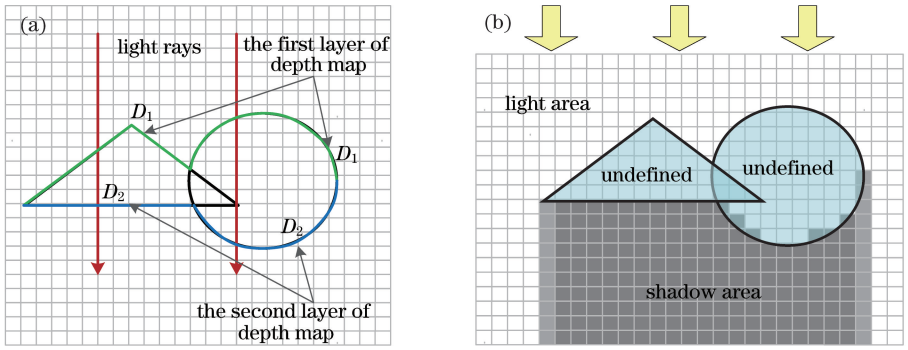

图 7　(a)双层深度图和(b)对应的空间划分

Fig. 7　(a) Double depth maps and (b) their corresponding spatial partition

4) 在为体素化的光线空间构建 SVO 时，仅沿着阴影体的轮廓往下细分，其他节点不再细分，如图 8(a)所示。

5) 在构建过程中，如果某个节点的子节点仅包含光源可见和未定义两种区域，则将这个节点设置为光源可见，并将其子节点合并，不需要再细分；如果某个节点全部位于未定义区域，则将该节点仍设置为未定义；如果某个节点仅包含光源不可见和未定义两种区域，则将这个节点设置为阴影，并将其子节点合并，不需要再细分。

6) 重复上述自适应的合并过程，直至到达根节点，完成整个 SVO 的构建。经过子节点类型的设置与合并操作后，SVO 的结构简化了很多，简化后的 SVO 如图 8(b)所示。

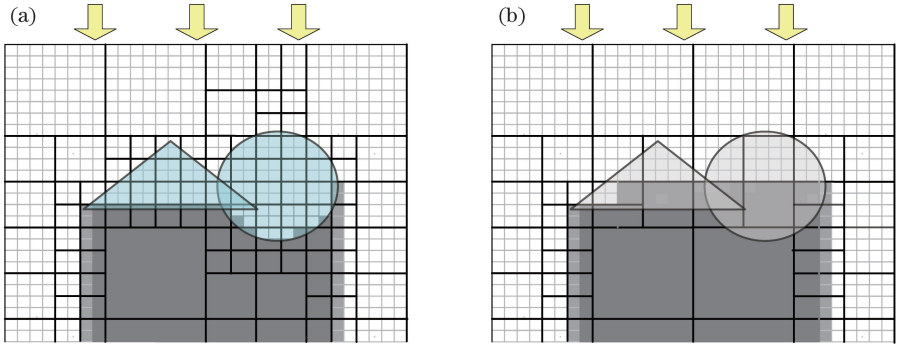

图 8　基于双层深度图的加速结构简化示意图。(a)光线空间中的 SVO；(b)基于节点属性自适应合并后的 SVO

Fig. 8　Schematic of acceleration structure simplification based on double depth maps. (a) SVO in space of light rays; (b) SVO after merging adaptively based on node attributes

7) 将简化后的 SVO 再按照 2.2 节方法进行相同节点的合并操作，生成对应的 SVDAG，如图 9 所示，其中每种颜色代表合并后的一个节点。

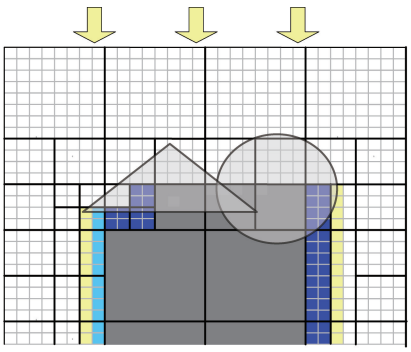

图 9　优化后的光线空间 SVDAG 加速结构

Fig. 9　Optimized SVDAG acceleration structure in space of light rays

优化后的光线空间 SVDAG 仅需要在阴影轮廓边缘保持所需的分辨率，同时可以通过牺牲闭合几何体内部的加速结构划分精度，达到减少节点数量的目的，而优化前的加速结构需要将整个阴影的分辨率精确地定义为所需的分辨率。本文的优化方式与文献[20]的有相似之处，都是基于多层深度图判断几何体对光源的可见性，但本文的方法可以合并更多的节点，对加速结构拥有更明显的简化和压缩效果，并支持投射在动态几何体上的阴影计算。

假设场景光线空间的体素数量为 $O(n)$(n 为节点数)，在构建和优化 SVDAG 加速结构过程中，所有节点均要遍历一遍，且仅与已经存在的唯一节点进行比较，因此该过程的时间复杂度在最佳情况下为 $O(n)$，最坏情况下为 $O(n^2)$。此外，由于相邻体素之间没有重叠，相同节点合并后的节点之间也没有重叠区域，空间开销只减不增，因此 SVDAG 的构建和优化过程的空间复杂度在最坏情况下为 $O(n)$。

4　基于时域相关性的动态场景 SVDAG 帧间复用方法

动态场景的帧与帧之间仍然存在静止或简单运动的几何体，因此动态场景存在一定的时域相关性和帧间冗余，利用这一特性将提出的 SVDAG 加速结构扩展到随时间变化的动态场景中，实现 SVDAG 的帧间复用和局部更新，减少总体数据量，并避免每帧都完全重建整个加速结构。

本文不仅仅针对某一时刻场景的相同节点进行合并生成 SVDAG，还在所有帧中进行搜索，找到帧与帧之间的相同节点，然后基于这些帧间相同的节点，实现 SVDAG 的帧间复用，减少不同帧 SVDAG 之间的节点冗余，具体方法如下。

1）按照时间顺序排列所有帧对应的 SVO 加速结构，并将这些加速结构的每一层节点看作一个列表，如图 10 所示，上半部分为场景的 SVO 序列，下半部分为每一层级（L）对应的列表。

2）自下而上地在列表的每一层中分别搜索相同的节点，这些相同的节点可能分别位于不同的空间区域和不同时刻的帧。在寻求 SVO 序列中的相同节点时，仅需要比较每个节点包含的子掩码和至多 8 个指向子节点的指针。

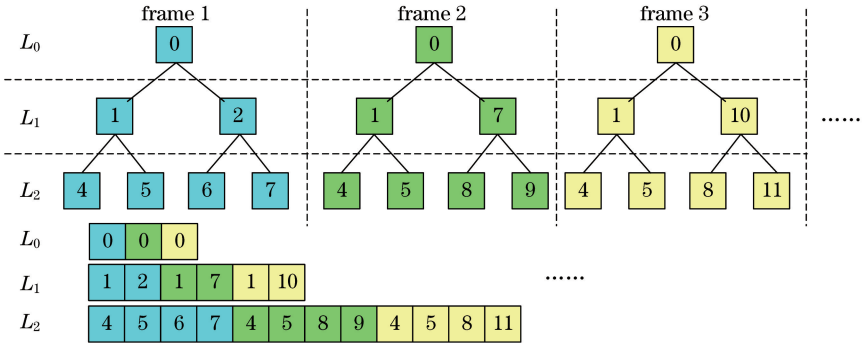

图 10　动态场景的 SVO 序列各层对应的列表示意图

Fig. 10　Schematic of list corresponding to each layer of SVO sequence in dynamic scene

3）进行相同节点的合并，自下向上地将父层级节点包含的指针更新为指向相应的唯一节点，相同子节点仅保留一个。继续向上合并，直到根节点的子节点，获得唯一节点的集合。即使当序列中的两个 SVO 完全相同时，仍然保留根节点不被合并，以用来代表不同的帧时刻，并按根节点的时间顺序进行排序。当完成除根节点层外的所有相同节点的合并和指针更新后，就得到了 SVO 序列对应的 SVDAG 加速结构，如图 11 所示。

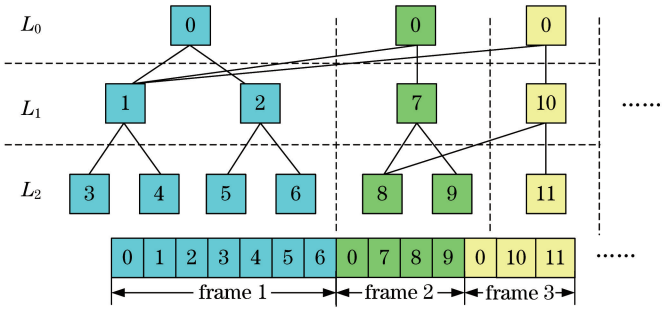

图 11　SVDAG 的帧间复用示意图

Fig. 11　Schematic of inter-frame multiplexing in SVDAG

如果全部帧的 SVO 序列同时进行 SVDAG 转换操作，内存需求会达到所有 SVO 占用内存的总和。为了避免这样的内存需求峰值，提出两种方案。一种是先将 SVO 向 SVDAG 的转换方法应用于每帧，然后将每帧的 SVDAG 一次性合并成为最终的 SVDAG，这样操作的运算量最少。另一种是逐帧进行 SVDAG 合并，即先对每帧的 SVO 进行转化并获得每帧的 SVDAG，然后通过合并第 1 帧和第 2 帧的 SVDAG 来获得前两帧所需的 SVDAG，依次类推，将第 n 帧合并入前（n-1）帧的 SVDAG 中，获得前 n 帧所需 SVDAG，直至

所有节点都被并入 SVDAG。这样逐帧进行合并虽然需要相对较多的计算量，但是内存需求的峰值最小，同时能够按照时间顺序渐进获取和处理三维场景。

上述基于时域相关性的 SVDAG 帧间复用方法利用动态场景的全部帧构成统一的 SVDAG，有效地减少了节点的帧间冗余，并支持加速结构的预构建。绘制时，能够避免动态场景的每一帧都完全重构整个加速结构，只需在各层级中修改和重建与上一帧不同的部分节点，保证了加速结构的更新速度。

5　实验与分析

5.1　SVDAG 加速结构的测试

SVDAG 加速结构的性能测试包括创建性能和加速性能两个方面。SVDAG 是基于体素场景建立的，将其与同样基于体素场景的 SVO[9] 和 ESVO[10] 加速结构进行对比。实验素材为 Stanford 大学提供的 Bunny、Buddha 和 Dragon 场景，并将其体素化，场景渲染后的结果如图 12 所示。

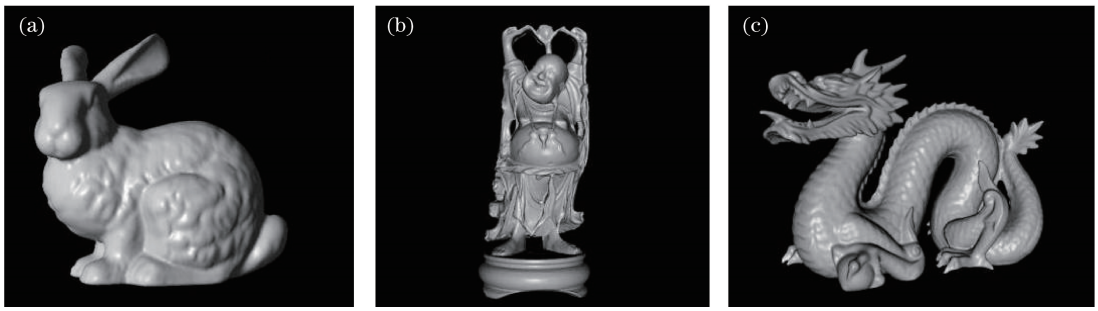

图 12　实验场景。(a) Bunny 场景；(b) Buddha 场景；(c) Dragon 场景

Fig. 12　Test scenes. (a) Bunny scene; (b) Buddha scene; (c) Dragon scene

5.1.1　加速结构的创建性能

在节点和指针数量方面，分别统计 SVO、ESVO 和 SVDAG 加速结构在不同场景中的节点总数和指针总数，结果见表 1，其中的数据是以 1000 pixel×1000 pixel×1000 pixel 的分辨率对实验场景进行体素化并构建加速结构时的情况。

表 1　节点数量和指针数量的对比

Table 1　Comparison of total number for nodes and pointers

Scene	Parameter	SVO	ESVO	SVDAG
Bunny	Total number of nodes /10^4	2307	2543	17.1
	Total number of pointers /10^4	529	701	4.8
Buddha	Total number of nodes /10^4	1577	1837	145.1
	Total number of pointers /10^4	438	487	43.4
Dragon	Total number of nodes /10^4	890	221	14.2
	Total number of pointers /10^4	214	53	3.8

从表 1 中可以看出，所有场景下 SVDAG 加速结构的节点总数和指针总数都显著减少，说明该结构可以有效地合并相同节点。兔子场景的节点数量减少幅度最大，减少到约为 SVO 的 1/135，这是因为该场景中几何体比较规则，产生了大量的相同节点。对于佛像场景，几何体的规律性相对较弱，但仍然可以获得可观的压缩率，节点总数减少到原来的 1/11 左右。

在内存开销方面，表 2 比较了 SVO、ESVO 和 SVDAG 加速结构的内存开销情况，这里对 SVO 采用了文献[21]提出的仅需更少字节的 SVO 算法。SVDAG 每个节点占用 8～36 B，ESVO 每个节点占用 8 B，SVO 每个节点仅需 1 B。在实验时忽略几何体的材质等信息。

表 2　存储开销对比

Table 2　Comparison of storage costs

Scene	Resolution /(pixel×pixel×pixel)	Storage cost /MB		
		SVO	ESVO	SVDAG
Bunny	512×512×512	5.82	31.93	1.06
	1000×1000×1000	22.42	123.50	3.32
Buddha	512×512×512	3.49	36.57	7.44
	1000×1000×1000	13.94	126.60	26.30
Dragon	512×512×512	0.25	0.47	0.08
	1000×1000×1000	0.82	1.49	0.31

从表 2 可以看出，SVDAG 的内存开销明显优于 ESVO 的。对于节点数量压缩率较高的场景，如兔子场景等，SVDAG 的内存开销明显优于 SVO 的内存开销，但是对于节点数量压缩率较低的场景，如佛像场景，SVDAG 的内存开销略多于 SVO 的，这是由于 SVDAG 的每个节点需要较多的字节数。虽然文献[21]的方法可以使 SVO 每个节点仅占用 1 B，对某些场景具有良好的压缩性能，但是这种结构在没有解压并添加指针之前无法被遍历，因此只能应用于离线存储等方面，无法应用于三维场景的实时绘制。

在 SVDAG 加速结构的创建性能方面，本文在不同体素分辨率下测试了从 SVO 转化为 SVDAG 所需的时间及其中查找相同子节点所需的时间，结果见表 3。对于分辨率超过 256 pixel×256 pixel×256 pixel 的场景，先将对应层级下的子树转换为子 SVDAG，然后再将这些子 SVDAG 与树的根部相连接，再次进行转换并生成最终的 SVDAG。

在表 3 中，t_A表示由 SVO 转化为 SVDAG 所消耗的时间，t_B表示转化操作中用于查找相同子节点所消耗的时间，可以看出，当分辨率从 256 pixel×256 pixel×256 pixel 增大到 512 pixel×512 pixel×512 pixel 时，时间开销增大较多，这是由于先完成子 SVDAG 的转换再合并的方法增加了一定的计算开销，但是对于更高分辨率的场景，这个开销在总时间开销中的比例较小。

表 3　加速结构的构建时间对比

Table 3　Comparison of build time of acceleration structures

Resolution /(pixel×pixel×pixel)	Bunny		Buddha		Dragon	
	t_A /ms	t_B /ms	t_A /ms	t_B /ms	t_A /ms	t_B /ms
128×128×128	6.14	2.29	8.75	3.06	11.36	4.31
256×256×256	24.06	5.07	33.55	8.07	40.25	11.30
512×512×512	250	30	340	40	400	50
1000×1000×1000	1020	130	1420	190	1600	220
2000×2000×2000	4250	490	5800	780	6530	890

在 SVDAG 加速结构的构建过程中，由于需要指针调整和子掩码生成等计算，因此生成的总时间必然比单独构建 SVO 的总时间长，但由于 SVDAG 加速结构可以预生成，在场景绘制时只要能够在 SVDAG 加速结构中高效地遍历和搜索，就可以保证绘制速度。

5.1.2　加速结构的加速性能

在不同测试场景中，对 SVO、ESVO 和 SVDAG 加速结构进行了光线跟踪实验，记录了单根光线访问的平均节点数和光线相交测试所需的时间。

由于 SVDAG 加速结构只是共用了相同的子节点，子节点代表的区域并没有扩大和改变，光线的相交测试过程依然要按照 SVO 的遍历过程，从根节点按照子掩码和指针进行逐层遍历，因此单根光线访问的平均节点数并没有变化，与 SVO 和 ESVO 的相同。

在光线相交测试所需时间方面，不同加速结构的实验结果如图 13 所示。测试时，以一个固定的视点向场景随机发出 10 条连续的采样光线，计算光线在不同加速结构中到达相交体素所需的平均时间，实验时仅考虑主光线，不考虑其他类型的光线(如光线反射、折射等)。

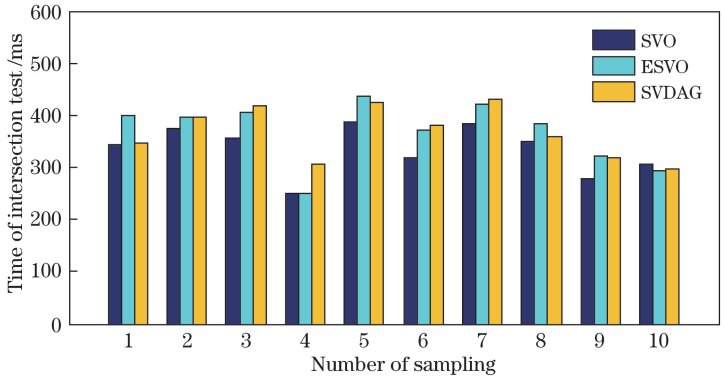

图 13　在不同加速结构中进行光线相交测试所需的时间

Fig. 13　Time required for tests of light rays intersection in different acceleration structures

从图 13 可以看出，基于 SVDAG 加速结构进行光线跟踪的计算时间与其他加速结构的基本持平，相交测试所需的时间没有因为内存开销的减少而增加。

5.2　基于双层深度图的 SVDAG 加速结构的优化测试

将基于双层深度图的优化后的 SVDAG 分别与未经优化的 SVDAG 和常用的不构建加速结构的阴影图算法进行对比。测试分为两部分：基于双层深度图的 SVDAG 压缩性能测试和光照计算性能测试。实验素材包括 Chalmers 大学提供的两个大规模三维场景，Closed citadel 和 Villa 场景以及一个三维地形场景 Terrain，其中，Closed citadel 和 Villa 场景中的几何体都是闭合的，而 Terrain 场景中闭合的几何体较少。三个实验场景体素化并渲染后的结果如图 14 所示。

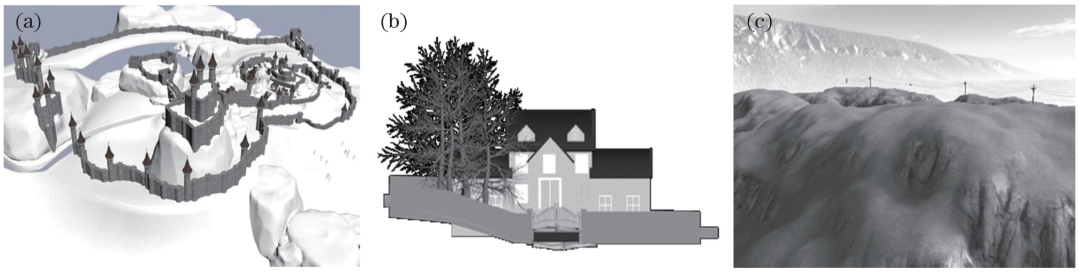

图 14　实验场景。(a) Closed citadel 场景；(b) Villa 场景；(c) Terrain 场景

Fig. 14　Test scenes. (a) Closed citadel scene; (b) Villa scene; (c) Terrain scene

5.2.1　基于双层深度图的 SVDAG 的压缩性能

对于各个实验场景，分别在内存中建立阴影图、SVDAG 和基于双层深度图优化后的 SVDAG，获取它们在不同分辨率下的存储开销 S_1、S_2、S_3，结果见表 4。为保证实验数据的可对比性，阴影图分辨率的设定与 SVDAG 的分辨率相当，且不进行压缩。

表 4　不同算法在不同场景下的存储开销对比

Table 4　Comparison of storage costs for different algorithms in different scenes

Resolution / (pixel×pixel×pixel)	Closed citadel			Villa			Terrain		
	S_1/MB	S_2/MB	S_3/MB	S_1/MB	S_2/MB	S_3/MB	S_1/MB	S_2/MB	S_3/MB
1000×1000×1000	2.30	0.493	0.231	2.30	0.462	0.146	2.30	0.592	0.538
2000×2000×2000	9.27	1.62	0.532	9.27	2.47	0.336	9.27	2.37	2.03
4000×4000×4000	37.20	5.10	1.23	37.20	7.78	0.772	37.20	7.50	6.64
16000×16000×16000	605.72	47.47	6.46	605.72	85.29	4.09	605.72	91.20	81.59
32000×32000×32000	2422	127.14	13.87	2422	249.04	9.40	2422	183.23	163.92

从表 4 可以看出，加速结构的存储开销随分辨率的增大而增大。阴影图算法的存储开销仅与场景分辨率有关，且随场景规模的增大呈几何级数增加，面对 32000 pixel×32000 pixel×32000 pixel 的高分辨率场景时，需要的存储空间高达 2.4 GB 以上。采用 SVDAG 加速结构时，即使没有面向闭合几何体的优化，仍然可

以获得非常高的压缩比，例如当场景分辨率为 2000 pixel×2000 pixel×2000 pixel 时，SVDAG 所需内存空间仅为阴影图算法的 1/5 左右，大大地减小了内存开销，且分辨率越大的场景，压缩比也越大。

由于实验场景 Closed citadel 和 Villa 中几乎所有几何体都是闭合的，当对它们采用基于双层深度图的 SVDAG 加速结构后，与阴影图算法相比，最高的压缩比能够达到 174.6。Terrain 场景中的闭合几何体较少，虽然 SVDAG 可以获得较高的压缩比，但是进一步优化后的 SVDAG 的存储空间相对于优化前的仅平均缩小了 10%左右。

图 15 描述了三种算法的存储开销随分辨率变化的情况，可以看出，随着分辨率的增大，阴影图算法存储开销的增长倍数一直保持在 4，而 SVDAG 存储开销的增长倍数逐渐减小，由 3 趋近于 2。

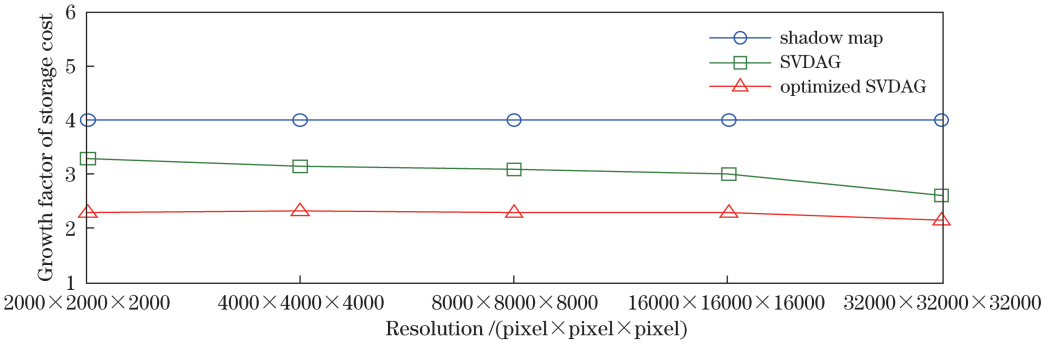

图 15　存储开销的增长倍数与分辨率间的关系

Fig. 15　Relationship between growth factor of storage cost and resolution

5.2.2　基于双层深度图的 SVDAG 的光照计算效率测试

实验时，视点位于场景上方，分别采用阴影图算法、SVDAG 加速结构和基于双层深度图的 SVDAG 加速结构，记录每一帧所需的阴影计算时间。图 16、图 17 和图 18 分别以 Closed citadel 场景、Villa 场景和 Terrain 场景为例，横坐标为连续变化视角对应的帧数，纵坐标为计算所需的时间。

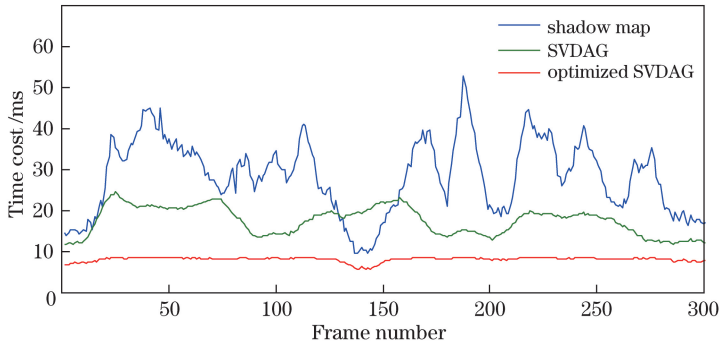

图 16　不同算法在 Closed citadel 场景中的时间开销对比

Fig. 16　Comparison of time costs for different algorithms in Closed citadel scene

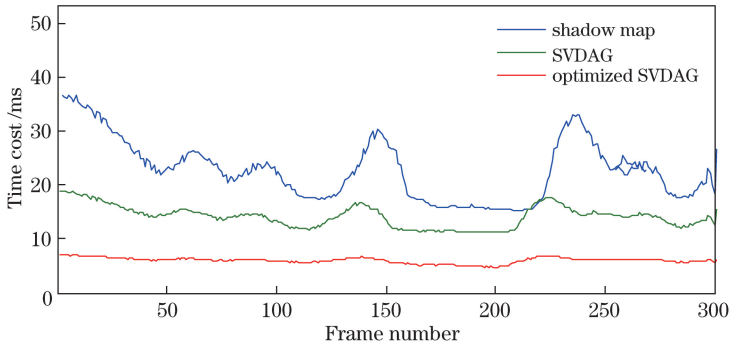

图 17　不同算法在 Villa 场景中的时间开销对比

Fig. 17　Comparison of time costs for different algorithms in Villa scene

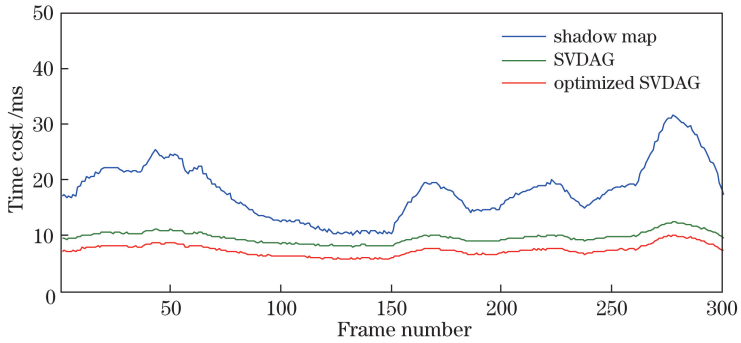

图 18　不同算法在 Terrain 场景中的时间开销对比

Fig. 18　Comparison of time costs for different algorithms in Terrain scene

从图 16～18 可以看出，基于双层深度图的 SVDAG 所需的阴影计算时间明显少于对比算法的，且性能相对稳定，例如在 Closed citadel 场景，新算法与阴影图算法相比，时间开销平均缩短了 67.9%。没有优化的 SVDAG 的算法效率有所降低，与阴影图算法的接近，但在大部分时刻还是高于阴影图算法的。在 Villa 和 Terrain 场景下，新算法也反映出与 Closed citadel 场景下相似的优化性能。

5.3　动态场景的 SVDAG 加速结构的性能测试

在动态场景的性能测试方面，分别采用基于时域相关性的 SVDAG、每帧单独构建的 SVDAG、每帧单独构建的 SVO 以及基于时域相关性的差分树编码进行测试[18]。实验分为节点压缩性能测试和动态场景绘制性能测试两部分。实验素材包括 Chalmers 大学提供的动态三维场景 Kitchen、Robot，以及一个用随机运动粒子描述的火焰场景 Fire，如图 19 所示。

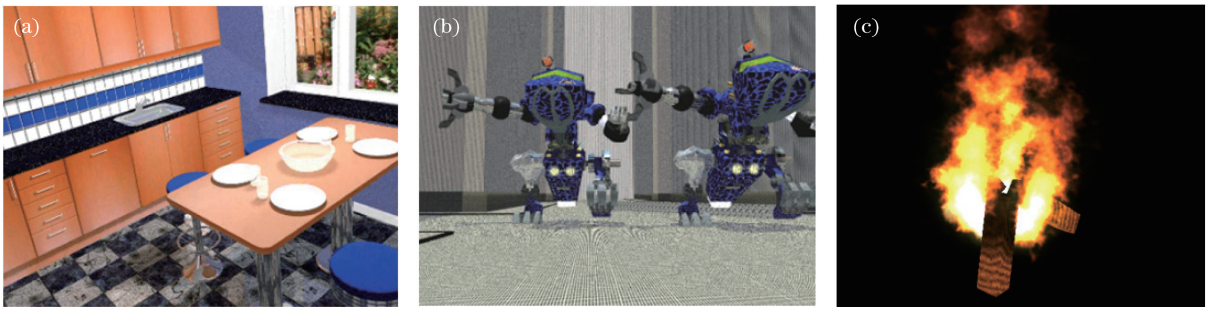

图 19　实验场景。(a) Kitchen 场景；(b) Robot 场景；(c) Fire 场景

Fig. 19　Test scenes. (a) Kitchen scene; (b) Robot scene; (c) Fire scene

5.3.1　基于时域相关性的节点压缩性能

统计不同加速结构在面对动态场景时所包含的节点数量，包括每帧平均节点数 N_p 和节点总数 N_n，结果见表 5。

表 5　每帧平均节点数和节点总数

Table 5　Average number of nodes per frame and total number of nodes

Scene	SVO		Difference tree		SVDAG		Time-correlation SVDAG	
	$N_p/10^3$	$N_n/10^6$	$N_p/10^3$	$N_n/10^6$	$N_p/10^3$	$N_n/10^6$	$N_p/10^3$	$N_n/10^6$
Kitchen	873	685	15.1	1.71	94.8	6.87	5.82	0.36
Robot	321	68.7	224	56.6	50.7	10.0	29.3	7.28
Fire	593	192	433	158	62.1	20.8	44.0	13.66

当场景中包含少量动态几何体，大部分都是静态时，例如 Kitchen 场景，由于基于时域相关性的 SVDAG 和差分树算法可以充分利用场景的时域相关性，节点压缩性能非常明显，与每帧单独构建加速结构相比，减少了大量节点。由于 SVDAG 还可以利用每帧内部的相同节点进行压缩，因此基于时域相关性的 SVDAG 的压缩性能比差分树算法的更好。

当场景中包含少量静态场景，大部分都是动态几何体时，例如 Robot 场景，差分树算法相较于每帧单独构建 SVO 仅减少了 1/3 左右的节点数，性能欠佳。因为场景中几何体的规律性较强，每帧单独构建 SVDAG 获得了较高的压缩比。而基于时域相关性的 SVDAG 节点总数仅为 SVO 的 1/9.4，节点数量的压缩率最高。

当场景中几乎完全都是动态的几何体，仅含非常少量的静态部分时，例如 Fire 场景，由于场景中存在大量随机运动的粒子，能够利用的时域相关性较少，差分树与每帧单独 SVO 相比，需要的节点数量优化程度不大，而每帧构建 SVDAG 和基于时域相关性的 SVDAG 还可以利用每帧内部大量规则粒子的空间相关性，获得了较高的节点压缩比。

5.3.2　动态场景的绘制性能测试

为验证新算法在三维场景绘制时的效率，对三个实验场景分别在不同分辨率下构建基于时域相关性的 SVDAG 加速结构并渲染，统计场景的绘制速率，结果见表 6。其中，帧速率 V_1、V_2 和 V_3 对应的绘制分辨率分别为 1024 pixel×768 pixel、800 pixel×600 pixel 和 400 pixel×300 pixel，在加速结构构建和渲染的过程中均未使用 GPU 进行并行计算。

表 6　实验场景的绘制性能

Table 6　Rendering performance of test scenes

Scene	V_1 /(frame·s^{-1})	V_2 /(frame·s^{-1})	V_3 /(frame·s^{-1})
Kitchen	21.3	28.1	62.9
Robot	12.0	16.1	53.4
Fire	6.20	9.90	27.1

从表 6 可以看出，新算法能够实现 SVDAG 加速结构的快速更新，在未使用 GPU 加速的情况下，可以基本满足动态场景对算法实时性的要求。同时，场景绘制的帧速率随着场景分辨率的增大而减小，也随着场景中动态几何体的增加而减小。当运动几何体较多且进行随机的无规律运动时（如 Fire 场景），帧间的时域相关性较低，相邻帧之间可以复用的节点较少，绘制性能相对较低，但是可以考虑利用 GPU 的并行特性来改善绘制性能。

6　结　　论

提出了一种基于 SVDAG 的光照计算加速结构，通过构建 SVO，自下而上地合并相同节点等方法，将 SVO 转化为 SVDAG，使加速结构的节点数量大幅度减少，并通过给定节点的遍历路径和子掩码，消除了空间位置的多义性。针对闭合几何体，基于双层深度图算法使位于闭合区域的节点进行自适应合并，得到更加简化的 SVDAG 加速结构。提出了一种基于时域相关性的 SVDAG 帧间复用方法，避免动态场景的每一帧都完全重构整个加速结构，并且能够利用全部帧构成统一的 SVDAG 加速结构整体，保证了动态场景的更新速度。最后实验验证了算法的可行性和先进性。将提出的光照计算加速结构应用于光线跟踪、光子映射等算法，同样可以获得较高的计算效率。

参　考　文　献

[1] Li Yang, Xia Xinlin, Chen Xue, *et al*. Simulation study on accelerated pore-scale radiative transfer of Ni foam[J]. Acta Optica Sinica, 2016, 36(11): 1124001.
李　洋, 夏新林, 陈　学, 等. 泡沫镍孔尺度辐射传递加速模拟研究[J]. 光学学报, 2016, 36(11): 1124001.

[2] Zhang Libao, Huang Ying. Image coding algorithm using optimal scaling scheme and quadtree partitioning[J]. Acta Optica Sinica, 2010, 30(12): 3460-3463.
张立保, 黄　颖. 一种结合最优缩放框架与四叉树分割的图像编码算法[J]. 光学学报, 2010, 30(12): 3460-3463.

[3] Qin Panke, Chen Xue, Wang Lei, *et al*. Multi-core shared tree based multipoint to multipoint multicast in multi-domain optical networks[J]. Acta Optica Sinica, 2015, 35(5): 0506001.
秦攀科, 陈　雪, 王　磊, 等. 多域光网络基于多核点共享树的多点对多点组播[J]. 光学学报, 2015, 35(5): 0506001.

[4] Zhang Wensheng, Xie Qian, Zhong Jin, *et al*. Acceleration algorithm in ray tracing by the octree neighbor finding[J]. Journal of Graphics, 2015, 36(3): 339-344.
张文胜, 解　骞, 钟　瑾, 等. 基于八叉树邻域分析的光线跟踪加速算法[J]. 图学学报, 2015, 36(3): 339-344.

[5] Yuan Yuwei, Quan Jicheng, Wu Chen, *et al*. Ray tracing acceleration structure based on octree adaptive volume merging[J]. Acta Optica Sinica, 2017, 37(1): 0120001.
袁昱纬, 全吉成, 吴　晨, 等. 基于八叉树自适应体归并的光线跟踪加速结构[J]. 光学学报, 2017, 37(1): 0120001.

[6] Peicheng Z, Huahu X, Minjie B. Research on parallel KD-tree construction for ray tracing[J]. International Journal of Grid and Distributed Computing, 2016, 9(11): 49-60.

[7] Áfra A T, Szirmay-Kalos L. Stackless multi-BVH traversal for CPU, MIC and GPU ray tracing[J]. Computer Graphics Forum, 2014, 33(1): 129-140.

[8] Kalojanov J, Billeter M, Slusallek P. Two-level grids for ray tracing on GPUs[J]. Computer Graphics Forum, 2011, 30(2): 307-314.

[9] Surhone L M, Tennoe M T, Henssonow S F, *et al*. Sparse voxel octree[M]. Mauritius: Betascript Publishing, 2011.

[10] Laine S, Karras T. Efficient sparse voxel octrees[J]. IEEE Transactions on Visualization and Computer Graphics, 2011, 17(8): 1048-1059.

[11] Crassin C, Neyret F, Sainz M, *et al*. Interactive indirect illumination using voxel cone tracing[J]. Computer Graphics Forum, 2011, 30(7): 1921-1930.

[12] Hornung A, Kai M, Bennewitz M, *et al*. OctoMap: An efficient probabilistic 3D mapping framework based on octrees[J]. Autonomous Robot, 2013, 34(3): 189-206.

[13] Liu H, Zhang F, Hu W. GPU rasterization based octree fast generation algorithm for terrain modeling[J]. IEEE International Geoscience & Remote Sensing Symposium, 2013: 282-285.

[14] Ize T, Wald I, Parker S G. Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures[C]. Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization, 2007: 101-108.

[15] Yang X, Yang B, Wang P, *et al*. MSKD: Multi-split KD-tree design on GPU[J]. Multimedia Tools and Applications, 2016, 75(2): 1349-1364.

[16] Bittner J, Hapala M, Havran V. Incremental BVH construction for ray tracing[J]. Computers & Graphics, 2015, 47(C): 135-144.

[17] Li Jing, Wang Wencheng, Wu Enhua. Ray tracing of dynamic scenes by managing empty regions in adaptive boxes[J]. Chinese Journal of Computers, 2009, 32(6): 1172-1182.
李　静, 王文成, 吴恩华. 基于空盒自适应生成的动态场景光线跟踪计算[J]. 计算机学报, 2009, 32(6): 1172-1182.

[18] Ma K L, Shen H W. Compression and accelerated rendering of time-varying volume data[C]. Proceedings of the 2000 International Computer Symposium-Workshop on Computer Graphics and Virtual Reality, 2000: 82-89.

[19] Kämpe V, Sintorn E, Assarsson U. High resolution sparse voxel dags[J]. ACM Transactions on Graphics, 2013, 32(4): 101-109.

[20] Shen Li, Yang Baoguang, Feng Jieqing. Multi-layered soft shadow mapping via contour back-projection[J]. Journal of Computer-Aided Design & Computer Graphics, 2013, 25(9): 1265-1274.
沈　笠, 杨宝光, 冯结青. 多层阴影图轮廓边背投软影算法[J]. 计算机辅助设计与图形学学报, 2013, 25(9): 1265-1274.

[21] Schnabel R, Klein R. Octree-based point-cloud compression[C]. Eurographics/IEEE VGTC Conference on Point-Based Graphics, 2006: 111-121.