在動態場景下
使用 SVO 的一個成像方法

黃琮偉 鄭進和

輔仁大學資訊工程學系

摘 要

Sparse Voxel Octree (SVO)是一種應用在儲存靜態大場景的資料結構，其結構簡單有規律的優點，成為常用的場景儲存方式，但對於動態場景卻不然，必須不斷的重新建構 SVO 樹以因應場景中物體位置的不停轉換，這樣一再地重建 SVO 樹會降低動態場景成像的動態效果。在本論文中我們考慮在動態場景中使用 SVO 資料結構，在避免一再重建物體的 SVO 樹情況下，我們提出一個簡單的成像方法，將動態場景的隱藏面去除與光影效果呈現出來。我們的方法是把場景中的每一個物體當作獨立的個體並建立一棵 SVO 樹，如果物體是擁有連續姿勢的動態物體，則將每個姿勢各建一棵 SVO 樹，利用 Z-buffer 與 Shadow z-buffer 的技巧，將場景的隱藏面去除與光影效果成像出來。最後，我們也實驗證明我們的方法正確而且實作容易。

關鍵字：Sparse Voxel Octree、Z-buffer、Shadow Z-buffer、隱藏面去除、光影效果

1. 簡介

Sparse Voxel Octree (SVO) [2 , 3 , 5 , 6]適用於儲存靜態大場景的一種資料結構，是一種能有效減少記憶體需求的儲存方式。在 2010 年 Laine 與 Karras [5 , 6]提出的新資料結構 Efficient Sparse Voxel Octrees (ESVO)，在節點上儲存物體的輪廓資訊，並使用資料壓縮方式儲存資料以減少記憶體需求量。由於建構一棵 SVO 樹需要一些時間，這是利用 SVO 樹的一項缺點，現今一般應用皆在動態場景之中，因此必須頻繁的重建 SVO 樹以因應物體位置的變動，為此想用 SVO 樹儲存大量場景資料，在動態場景中如何避免物體的 SVO 樹一再的重建變成一個重要課題。所以本篇論文主要目的就是在動態場景下，使用 SVO 樹儲存場景資料，避免一再重建 SVO 樹，快速產生隱藏面去除與光影效果。

我們的成像方法就是將一個大場景每一個物體看成獨立的個體，在局部座標系中，分別建構一棵 SVO 樹，如果是具有連續動作的物體，則將其每個姿勢的模型各建一棵 SVO 樹，往後在處理成像時，再依時間採用其中一個適當的 SVO 樹即可。在本論文中，我們使用兩個座標系：局部座標系與世界座標系，物體的模型資料是建構在局部座標系中，而整個場景則是在世界座標系中，因此每個物體從局部座標系中被置入世界座標系中，就需一個幾何轉換動作，我們可以有一個 4X4 矩陣表示該轉換動作，此矩陣稱之為 model view 矩陣。假設某物體之 model view 矩陣為 M，而 P 點是局部座標系中模型上之一點 (以 homogeneous coordinates [9]表示，也被視為一個4×1矩陣)。$M \cdot P$即為 P 點在世界座標系的座標位置，本論文所考慮的動態場景允許物體在世界座標系中做平移或旋轉運動，假設某物體做了上述幾何運動，其相對的運動矩陣為M' (是一4×4矩陣)，則該物體所對應之 model view 矩陣 M 應更新為$M' \cdot M$，這更新後的矩陣 M 即是該物體從局部座標系轉到世界座標系的轉換矩陣，換言之，從世界座標系轉換至局部座標系之轉換矩陣為M^{-1} (為矩陣 M 的反矩陣)，這矩陣M與M^{-1}為在局部座標系與世界座標系間轉換座標的矩陣。

在本論文中我們運用 Z-buffer 與 Shadow Z-buffer 的技巧，提出一個簡單又容易實作的成像方法，將在世界座標系中動態場景迅速成像把隱藏面去除與光影效果顯示出來，在當中所有涉及到找交點的計算，皆是轉到局部座標系下，與物體的 SVO 樹找交點。如此就達到避免因物體運動而需重建該物體的 SVO 樹，只是要即時更新該物體的 model view 矩陣。

本論文的組織結構如下：第二節說明相關研究；第三節說明我們的成像方法；在第四節說明實驗結果；最後，在第五節說明結論與未來展望。

2. 相關研究

2.1 Sparse Voxel Octrees

Sparse Voxel Octrees (SVO) [2 , 3 , 5 , 6]是一種樹狀結構，把整個物體當成被一個Voxel包起來，此Voxel相當於一個節點，並切割成八塊對應成八個子節點來分別儲存各個的資料。然後以遞迴方式分割下去，適合用來儲存大量場景資料又兼具有level-of-detail特性，對成像速度具有優勢表現。最早在1987年由Amanatides 與Woo [1]提出一種Voxel追蹤法。而後在2006年由Knoll、Wald、Parker 與Hansen [4]所提出的一種用於Octree結構的光線追蹤，此方法類似KD-Tree的搜索方式，利用不同層級的概念，遞迴的對物體做光線追蹤，它的方法不適合用於GPU實作。

在2009年由Crassin、Neytet、Lefebvre 與Eise [3]等人提出可在GPU上做Voxel成像的演算法，他們的演算法包含如何解決CPU和GPU之間的資料管理。

2010年由Laine與Karras [5 , 6]所提出的Efficient Sparse Voxel Octrees (ESVO)，有別於原本的SVO，他們對於在每一個Voxel中的模型輪廓 (surface)，只儲存包夾在此Voxel內模型輪廓之兩平行面 (bounding plane)，所以ESVO的架構中會有節點內沒有存資料的情況，在往後的成像處理能加快執行速度。另外，它也利用了壓縮方是來儲存資料使記憶體需求量少一些。最後用光線追蹤法在GPU上使用CUDA[7]了來做平行運算，處理成像，並且產生很好的效果。

針對大資料量場景關於SVO樹建構的研究，過去在2010年Laine和Karras [5, 6]提出上而下slice-based的建構方法。在2014年Baert、Lagae與Dutré [2]與在2015年Patzold與Kolb [8]分別提出out-of-core的建構方法，希望使用有限的記憶體有效且快速建構SVO樹，但這些方法也僅限於靜態場景的SVO樹建構。

2.2 Z-Buffer

Z-buffer [9 , 10 , 12]演算法是一個利用深度來做比較遠近的隱藏面去除演算法，它的原理是在眼睛前方之投影面上有一長方形區域 (稱為觀景視窗)，將此觀景視窗分割成數格，其維度與螢幕上顯示視窗之解析度維度相同，亦即觀景視窗上每一格皆有螢幕視窗相對位置上一像素與之對應，而且從眼睛往觀景視窗上每一格中心點方向看出去 (拉射線)，看到場景中物體最近點的距離，稱為該格的深度值(又稱為Z深度)，所以Z-buffer演算法需要一個深度表來儲存觀景視窗上每一格的深度值，為求得最後之深度表，就必須將場景中每一多邊形依序投影到觀景視窗中，並計算投影區域內每一格的深度值，再把這深度值與先前深度表中相對應格內之深度值做比較，較小者留下來。在深度表中之每一格除了儲存深度值外，還儲存該深度值所對應3D點座標與顏色值。有了最後之深度表，即可呈現3D場景中物體

間隱藏面去除的效果。可是Z-buffer的主要問題就是需要使用大量的記憶體(儲存深度表),和精確度的問題,但隨著硬體設備的發展和更多改善精確度的方法出現後,Z-buffer的使用也越來越普遍了。

2.3 Shadow Z-Buffer

Shadow Z- buffer [12]的原理和Z-buffer一樣,不同的是把光源位置當作眼睛位置,為處理光影效果所需。假設在光源之前有一張虛擬的表格Shadow Z-buffer深度表 (類似Z-buffer 深度表)。同時,在光源之前也有一個長方形平面區域(類似Z-buffer演算法之觀景視窗),稱之為光源的觀景視窗,此視窗被分割的維度與Shadow Z-buffer深度表的維度一樣,光源的觀景視窗上每一格皆與Shadow Z-buffer深度表上其中一格相對應。而Shadow Z-buffer深度表上每一格儲存值就是從光源位置發射一條光線經過該格所對應之光源的觀景視窗格中心點,首先打到物體時其間的距離,可以利用這張Shadow Z-buffer深度表去決定場景中任一位置是否在陰影區。從眼睛射出的視線光線所看到的點,比對此點與光源的距離是否大於該點在Shadow Z-buffer深度表中對應格的深度值,如果是大於則代表此點為在陰影區中;反之,則此點不在陰影區。

2.4 光線追蹤

光線追蹤(Ray Tracing)是一種成像的方法,最早由Whitted [11]在1980年提出,在眼睛之前有一投影視窗(以長方形表示),將投影視窗分割成與螢幕視窗一樣的維度,因此每一投影視窗的一格對應螢幕上的一個像素(pixel),光線追蹤法是計算螢幕上每一像素的顏色,為此目的,從眼睛往此螢幕像素所對應在投影視窗上之格子中心拉一視線,找到此視線交到3D場景中的第一點位置。如果從眼睛這一視線所看到的這一點,光源無法直接照到,則此點位置位在陰影區。倘若考慮3D物體材質,則需往下處理從這一點為起點的反射線與折射線(如果物體材質是透明或半透明)往下找交點,進而計算眼睛在這視線方向上看到第一點位置的顏色與光源照射的效果,必須遞迴的往下計算反射與折射的效果[9]。

3. 研究方法

本論文中我們使用 SVO 樹在動態場景環境下產生即時隱藏面去除與光影效果,以往的 SVO 樹主要用於儲存龐大三角片數量的靜態場景,對於動態場景的處理往往需要隨著場景資料變動而重建物體的 SVO 樹,所以在本論文中,我們提出的成像方法只需要在建立一次 SVO 樹的情況下,就可以處理在動態場景下計算出隱藏面去除與光影效果,我們的方法分別在以下各節中敘述。

3.1 座標轉換

由於 SVO 樹可以有效的組織儲存大場景資料，其結構又具有 level-of-detail 特性，因此在進行成像處理時具有相當的優勢，只是在建構 3D 物體的 SVO 樹時需花一些時間，在本論文中，由於考慮動態場景，因此有些 3D 物體會做運動，在本論文中只考慮平移與旋轉運動，不希望 3D 物體一變換位置就重建一棵 SVO 樹，以提升時間效率。所以我們必須將在局部座標系(X_L, Y_L, Z_L)之原始物體模型資料經幾何轉換，包含平移與旋轉合成運動，以4×4矩陣 M 儲存，放入動態場景之世界座標系(X_W, Y_W, Z_W)中，也就是原始模型上任一點$P(X_L, Y_L, Z_L, 1)$ (以 homogeneous coordinates 表示)，經$M \cdot P$之後就進入世界座標系中，我們稱矩陣 M 為該物體之 model view 矩陣。當物體在世界座標系中會做一些運動如平移或旋轉，其幾何轉換矩陣以M'代表，則在世界座標系中之物體模組資料，就是經由在局部座標系中之每一點$P(X_L, Y_L, Z_L, 1)$，經$M' \cdot M \cdot P$運算後得到。

因而每一物體只需要更新其 model view 矩陣 M 為$M' \cdot M$(即$M = M' \cdot M$)，以後物體若有其他運動，其運動矩陣假設為M''，更新該物體的 model view 矩陣 M 為$M'' \cdot M$。如此一來，任一時候在世界座標系中之物體模型資料就是把模型在局部座標系上之點$P(X_L, Y_L, Z_L, 1)$拿來做$M \cdot P$計算而得到。在世界座標系中如果眼睛位置在$E(E_X, E_Y, E_Z, 1)$其視線方向為$V(V_X, V_Y, V_Z, 0)$，在此視線方向可看到的點可以藉由世界座標系中眼睛視線方向與物體之相對位置關係，將眼睛位置 E 與視線方向 V 轉換到局部座標系中位置E'與視線方向V'，其中$E' = M^{-1} \cdot E$，$V' = M^{-1} \cdot V$ (當中M^{-1} 是M矩陣的反矩陣)，有了位置E'與V'視線方向，在局部座標系中以E'為起點，V'為視線方向看到物體模型之點 P，即是在世界座標系中看到的點，此點之世界座標為$M \cdot P$ 。

在本論文中，我們在局部座標系中採用上而下 slice-based SVO 樹建構法[5, 6]為每一物體建構一棵 SVO 樹，往後在世界座標系的眼睛位置 E 與在視線方向 V 所看之點的計算，皆是從世界座標系將眼睛位置 E 與視線方向 V 轉至局部座標系中而分別得到E'與V'，然後位置E'往視線方向V'發出一條射線，找到與 3D 模型第一個交點 P 所在，再計算$M \cdot P$後之點即是在世界座標系中眼睛在視線方向 V 所看到之點座標。

3.2 隱藏面去除

在本論文中隱藏面去除的處理採取類似 Z-buffer 方法，將每一物體在世界座標系中其對應的 Z-buffer 深度表做一一的合併，而得到最後的深度表(簡稱為 ZBT 深度表)，以達到成像後有隱藏面去除效果。

由於在本論文中將每一物體在局部座標系中建構好一棵 SVO 樹，並存有一相對幾何運動之4×4 model view 矩陣 M，使得在局部座標系中，物體模型經由矩陣 M 而放入世界座標系的場景中。為求得在世界座標系中之物體的 Z-buffer 深度表，

我們的做法類似光線追蹤(ray tracing)處理方式，從眼睛位置 E 往每一 pixel 所對應之空間位置拉方向為 V 的射線，找此射線與物體第一個交點，再計算此交點與眼睛位置 E 的距離，即是該 pixel 的深度值，由於物體之 SVO 樹是存在局部座標系中，因此必須將眼睛位置 E 與射線方向 V 透過該物體對應的幾何轉換之 model view 矩陣 M，求得局部座標系下的眼睛位置 E' 與射線方向 V'，其中 $E' = M^{-1} \cdot E$ 與 $V' = M^{-1} \cdot V$ (M^{-1} 是 M 矩陣的反矩陣)，再進行從 E' 位置為起點，V' 射線方向拉射線，找局部座標系下物體與此射線的交點 P，P 點與 E' 位置的距離即是所要的該 pixel 所對應之 ZBT 深度表中對應格的深度值。

為求有效率的得到最後 Z-buffer 深度表 ZBT，將起始的 ZBT 深度表設為背景深度表，往後將場景中每一物體之 Z-buffer 深度表 T 合併入表格 ZBT 深度表中，在處理深度表 T 時只考慮該物體在投影到螢幕上的 bounding box 區域內之 pixel 深度，如此即能大大降低時間需求，如圖 1 所示。

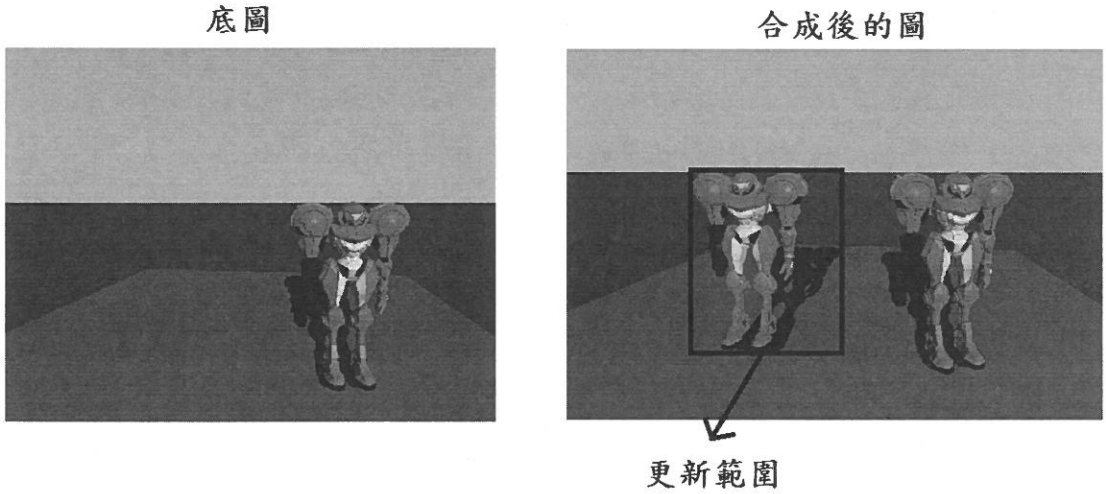


圖 1 檢查更新範圍內的顏色變化。

3.3 Shadow Z-Buffer

如果只呈現 ZBT 深度表內容，其效果只有去除隱藏面，為處理光影效果，在本論文中需要一份 Shadow Z-buffer 深度表(簡稱 SZBT 深度表)，其取得方法是在世界座標系下把光源位置當成眼睛位置，按照隱藏面去除方式找到最後合併所得之 Z-buffer 深度表，此表格即是我們所需要之 SZBT 深度表。

3.4 成像方法

就任一瞬間而言，所有物體在世界座標系上是靜止的，要算出當時的成像有隱藏面去除與光影效果，就要利用 ZBT 深度表與 SZBT 深度表，檢驗 ZBT 深度表中每一格所對應螢幕之 pixel P_f 是否在陰影區，其做法如下：

檢驗 ZBT 深度表陰影區

For each pixel P_f {

1. 從 ZBT 深度表中找到 P 點座標(世界座標)，其中 P 點是眼睛所看到之點而且投影到 pixel P_f；
2. P 點與光源位置 L 連線找到 P 點投影到 SZBT 深度表中對應格所存之深度值 depth；
3. **if** P 點與光源位置 L 距離 > depth
   then 將 pixel P_f 塗黑; //P_f 是在陰影區
   }

當每一瞬間把靜止狀態的場景成像完畢之後，接著更新每一個會動物體的 model view 矩陣 M，在已知每個會動物體下一瞬間會做何種運動(平移或旋轉情況下)，假設某一物體它的運動矩陣是M′，則更新它的 model view 矩陣 M 為M′·M。接著計算新的 ZBT 深度表與 SZBT 深度表，進行下一瞬間畫面的成像工作。最後，將我們的成像方法敘述如下：

我們的成像方法

1. 對每一物體在局部座標系建立一棵 SVO 樹；
2. 起始化每個物體的 model view 矩陣；
3. **For** each 單位時間 **do**
   3.1 更新每個物體的 model view 矩陣；
   3.2 計算每個物體的 Z-buffer 深度表；//藉由光線追蹤法完成
   3.3 合併所有 Z-buffer 深度表成為一個 ZBT 深度表；
   3.4 計算每個物體的 Shadow Z-buffer 深度表；//藉由光線追蹤法完成
   3.5 合併所有 Shadow Z-buffer 深度表成為一個 SZBT 深度表；
   3.6 檢驗 ZBT 深度表陰影區，並設定在陰影區之 pixel 為黑色；
   3.7 將 ZBT 深度表內容呈現於畫面；
4. **End**；//**For**

3.5 處理帶有連續動作的物體

帶有連續動作的物體是指物體動作在原地具有一連串的姿勢模型M_1, M_2, M_3, …，為了要呈現在畫面上有連續動作的效果(如圖 2)，在處理瞬間靜止畫面的成像時，針對此類物體，必須輪流的採用各個姿勢模型資料進行成像。因此在整個起始化時，必須把此類物體的每個姿勢模型M_i在局部座標系下建立一個 SVO 樹，在成像時要用到的姿勢模型，就選用其所對應的 SVO 樹，但是這類的每一物體其 model view 矩陣只有一個。

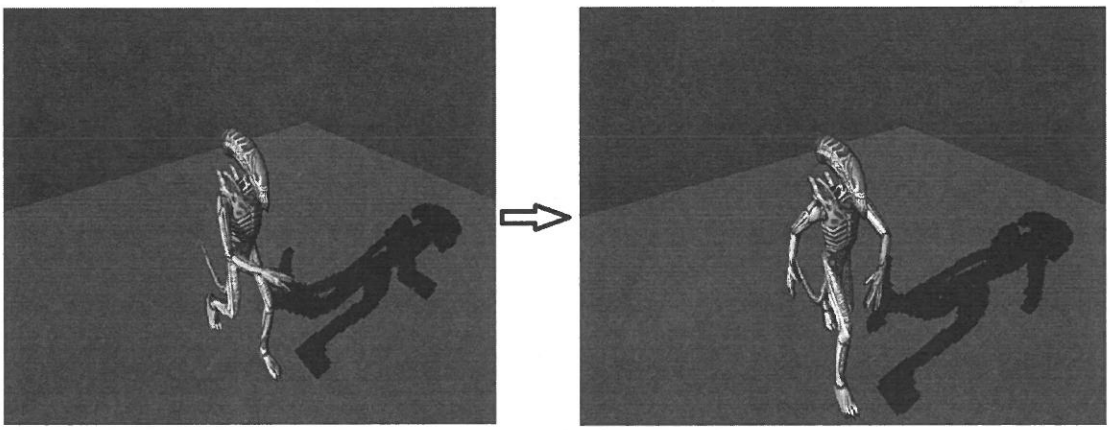


圖 2 帶有連續動作的物體

4. 實驗結果

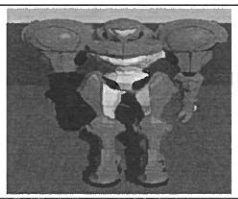
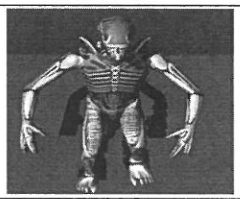
4.1 實驗環境

在本論文中所使用的實驗環境為 CPU：Intel Core(TM)2 Quad CPU Q6600，GPU：NVIDIA GeForce GTX 750 Ti， RAM：DDR3 2G*2，OS：Windows 7 Ultimate 64Bit。開發環境為 Visual Studio 2010，使用的程式語言為 C，畫面解析度為 1920*1080 (Pixels)，場景光源為一個點光源。

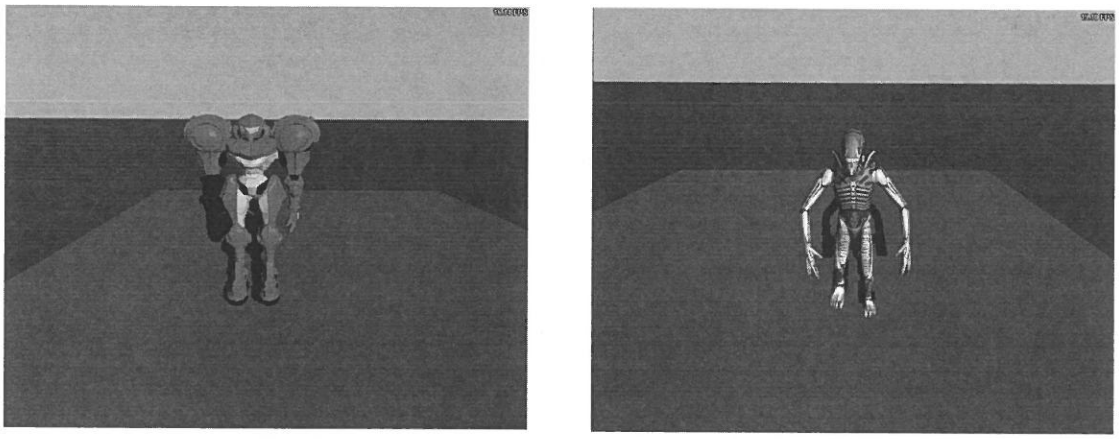
4.2 實驗結果

在本論文中的實驗主要是要呈現擁有大量資料量的動態場景成像，因此我們的實驗是依序比較不同數量下的靜態與動態物體，並取得 FPS (frames per second)資料，來觀察我們的實驗結果。在本論文中使用到的兩個物體(資料如表 1)，其中物體 A 的資料為三角片 16978 片，頂點 41823 個，物體 B 為擁有 50 個連續姿勢之物體，每一動作之模型資料為三角片 2040 片，頂點 1248 個，總共物體 B 擁有三角片 102000(等於 2040x50)片，頂點 62400(等於 1248 x50)個，而在本論文中的實驗，我們分別使用物體(包含物體 A 與物體 B)的數量為 1、3、7、10 個做四個實驗來進一步觀察實驗結果，且載入 SVO 樹的層數都是 10 層，而三角片數量也會隨著物體的數量增加而增加，並呈現不同角度的景像與其光影效果。

表 1 實驗數據

名稱	物體 A	物體 B (共 50 個連續姿勢)
三角片數目	16978	2040/每個姿勢
頂點數目	41823	1248/每個姿勢
模組樣式		

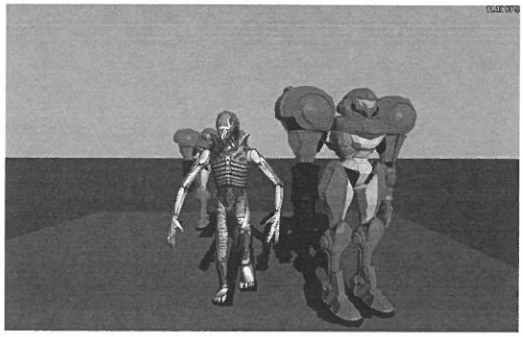
實驗一分別載入一個物體 A 和一個物體 B，物體 A 是沒有連續動作的物體，而物體 B 是有連續動作的物體。對於物體本身的運動，物體 A 與物體 B 都可以進行平移與旋轉，除了可以透過鍵盤控制外，也可以直接設定自體旋轉，來呈現物體的運動。實驗一結果畫面呈現之單一物體A或單一之物體B有相近的FPS表現。在本實驗中，物體 A 的三角片有 16978 片，頂點則有 41823 個，載入完成後實驗結果畫面呈現之FPS介於16~20之間，而物體B為擁有50個連續動作的動態物體，三角片擁有 102000 片，頂點擁有 62400 個，只要完全載入所有的連續動作 SVO 樹後，實驗結果畫面呈現之 FPS 介於 18~22 之間。關於動態光影效果的部分，我們的方法可以產生自然的光影，如圖 3 所示。



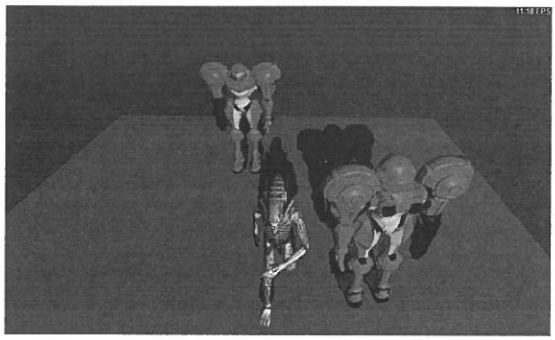
(1)物體 A　　(2)物體 B

圖 3 實驗一場景

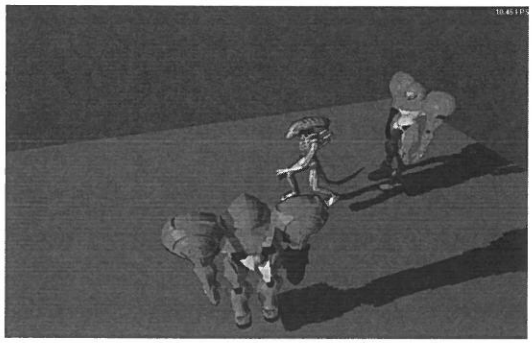
實驗二載入兩個物體 A 和一個物體 B，場景中有一個物體 A 可以進行自轉運動，另一個物體 A 則是靜止狀態，而物體 B 則是由鍵盤控制運動，可以控制平移與旋轉。實驗二中使用兩個物體 A 的三角片共有 33956 片，頂點則有 83646 個，與物體B其擁有50個連續動作的動態物體，三角片共有102000片，頂點共有62400個，實驗結果畫面呈現之 FPS 介於 10~13 之間。圖 4 呈現實驗二在不同的眼睛位置看物體間的遮蔽效果，而圖 5 呈現實驗二物體在旋轉的時候物體本身的面光與背光所造成的光影效果。



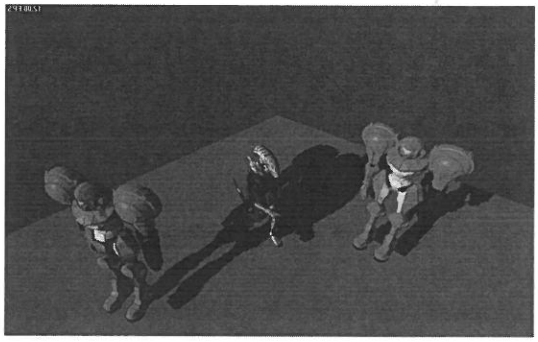
(1)



(2)

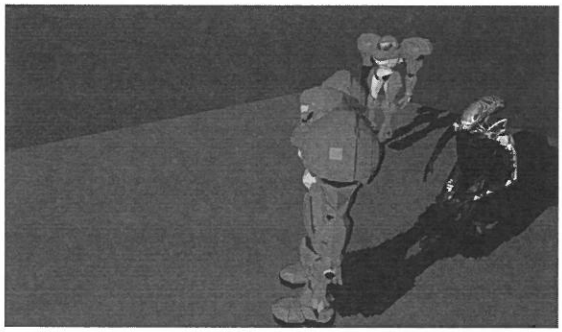


(3)



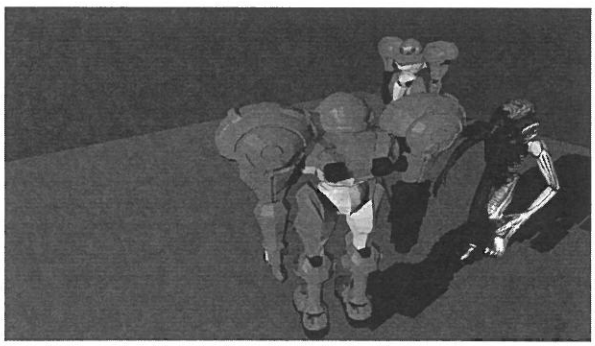
(4)

圖 4 實驗二場景



(1)

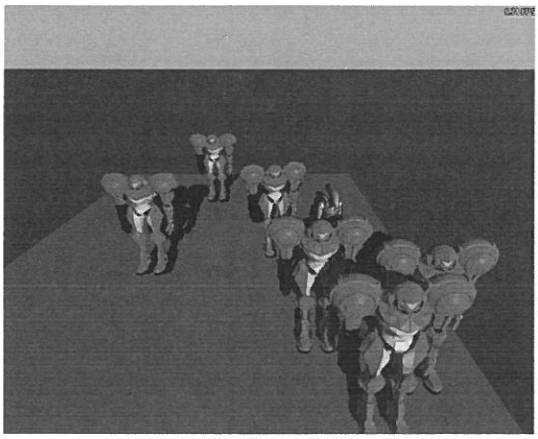
(2)



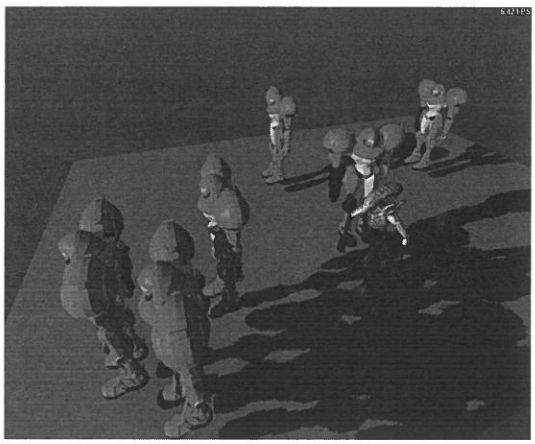
(3)

圖 5 實驗二呈現之物體的旋轉和所產生的連續畫面

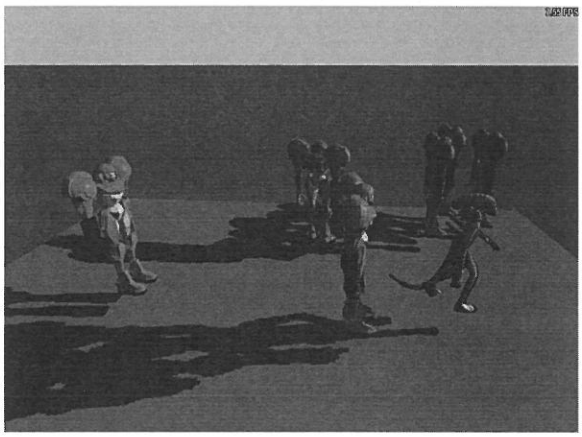
實驗三載入七個物體，包含六個物體 A 和一個物體 B，場景中有一個物體 A 可以進行自轉運動，剩下的物體 A 為靜止狀態，而物體 B 則是由鍵盤控制運動，可以控制平移與旋轉。實驗三的場景資料量包含六個物體 A 的三角片有 101868 片，頂點則有 250938 個，與物體 B(擁有 50 個連續動作的動態物體)的三角片 102000 片和頂點 62400 個。實驗結果的畫面呈現之 FPS 介於 5~8 之間，圖 6 呈現從不同角度觀察成像的結果。



(1)



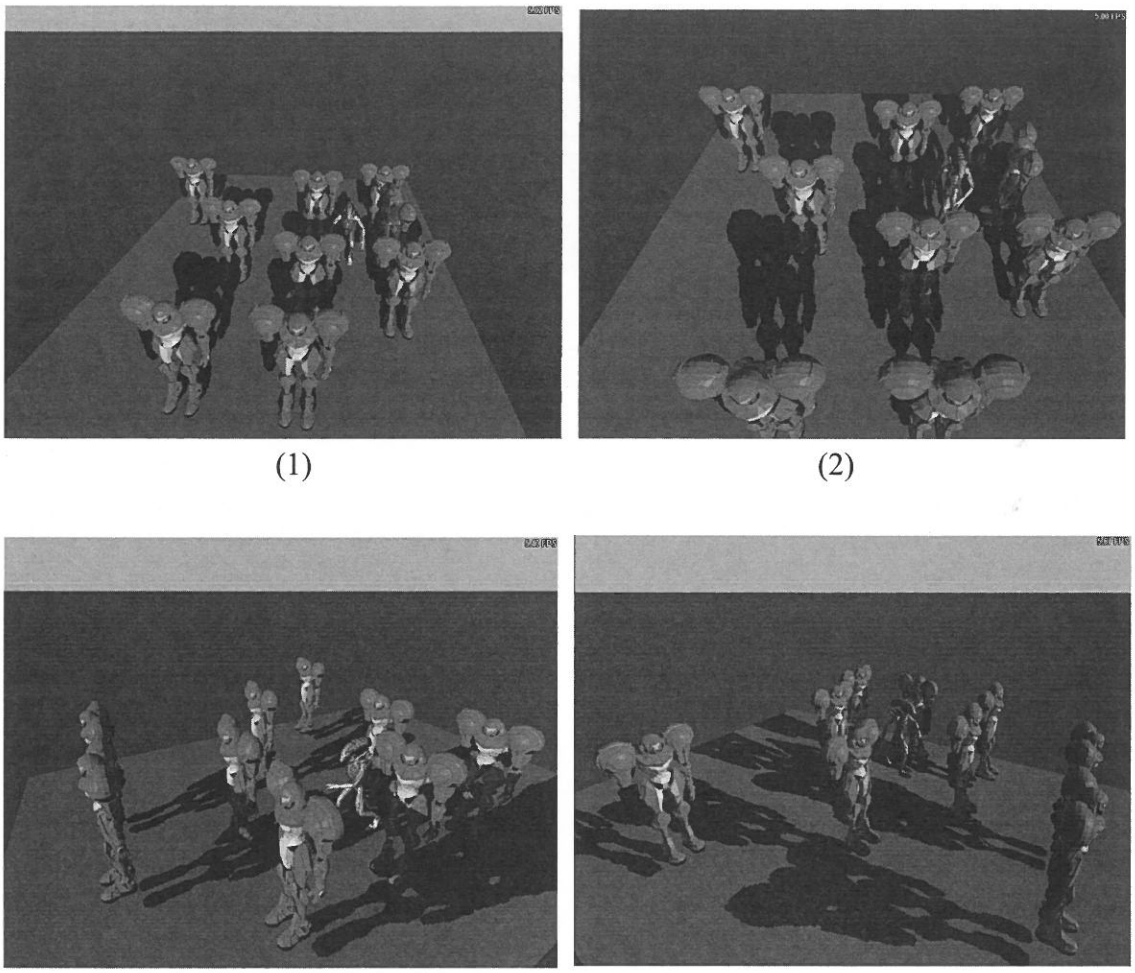
(2)



(3)

圖 6 實驗三場景

實驗四載入十個物體，包含九個物體 A 和一個物體 B 所組成，場景中有一個物體 A 可以進行自轉運動，剩下的物體 A 皆為靜止狀態，而物體 B 則是由鍵盤控制運動，可以控制平移與旋轉。實驗四場景中的九個物體 A 的三角片有 152802 片，頂點則有 376407 個，與物體 B(為擁有 50 個連續動作的動態物體)的三角片 102000 片和頂點 62400 個。實驗結果畫面呈現之 FPS 介於 5~6 之間，圖 7 呈現實驗四從不同角度觀察所有的成像結果。



(1) (2)

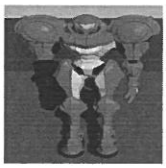
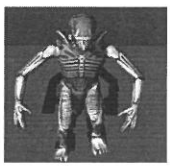
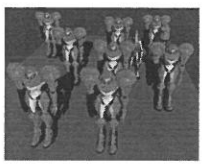
(3) (4)

圖 7 實驗四場景

4.3 討論

從本論文的實驗結果看來(如表 2 所示)，場景內物體數量愈多(即場景的總三角片愈多)，畫面呈現的 FPS 會下降，即使是場景中有做連續動作的物體，雖然它每個姿勢都有一個模組資料，但是在本論文實驗中，在處理隱藏面去除與光影效果時，也只是要決定要用到該物體的那一棵 SVO 樹而已。

表 2 實驗結果

實驗 / 數據	實驗一		實驗二	實驗三	實驗四
物體數量	一個模組 (物體 A)	一個模組 (物體 B)	三個模組 (兩個物體 A 一個物體 B)	七個模組 (六個物體 A 一個物體 B)	十個模組 (九個物體 A 一個物體 B)
場景三角片數量	16978	102000	135956	203868	254802
FPS	16-18	18-22	10-13	5-8	5-6
場景					

在本論文中，成像前需要每一物體的 Z-buffer 深度表，進而合成所有物體 Z-buffer 深度表成為最後的 Z-buffer 深度表以處理隱藏面去除，在此每一物體的 Z-buffer 深度表資料皆由光線追蹤法[9, 11]求得，是一計算量相當大之工作，同樣地，在求得 Shadow Z-buffer 深度表以處理光影效果的過程，也是類似的狀況。因此隨著場景資料量增加與物體數量愈多，成像時間就愈長。

5. 結論與未來展望

SVO 樹主要用於儲存單一的大型靜態場景，對於動態場景往往需要重建整棵 SVO 樹來因應場景的變動，對重建 SVO 樹而言，是很沒時間效益的。在我們的論文中，我們把每一個物體獨立出來，在局部座標系下個別建一棵 SVO 樹，而每個物體也有個別的 model view 矩陣表示如何從局部座標系轉入世界座標系中以反應其目前的位置，物體若有移動，則更新其 model view 矩陣即可，以避開重建 SVO 樹的麻煩而提升成像速度。利用獨立的特性，Z-buffer 和 Shadow Z-buffer 的原理，我們的成像之方法可以簡單的處理隱藏面去除與光影遮蔽等效果。

在動態場景下，我們的成像方法在未來還有幾個優化進步空間：第一，SVO 樹在起始建構時間的縮短；第二，合成多個 Z-buffer 深度表與產生 Shadow Z-buffer 深度表過程時間的縮小，因為在動態場景中，這兩項動作頻繁執行；第三，提升光影效果產生的速度。在可見的未來，一般的應用其動態場景的資料是會愈來愈大量，利用 SVO 樹儲存大量場景資料與其具有 level-of-detail 特性帶給相當優勢，但是其成像速度的提升是一重要課題。

致謝

我們要感謝 NVIDIA Research 的 Samuli Laine 與 Tero Karras 所提供的相關資源。

參考文獻

[1] J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing", In *Eurographics*, pp. 3-10, 1987.

[2] J. Baert, A. Lagae, and Ph. Dutré, "Out-of-core construction of sparse voxel octrees", *Proceedings of the 5th ACM High-Performance Graphics Conference*, pp. 27-32, Jul. 2014.

[3] C. Crassin, F. Neytet, S. Lefebvre, and E. Eisemann, "Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering", *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pp. 15-22, ACM, Feb. 2009.

[4] A. Knoll, I. Wald, S. G. Parker, and C. Hansen, "Interactive isosurface ray tracing of large octree volumes", *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pp. 115-124, Sep. 2006.

[5] S. Laine and T. Karras, "Efficient sparse voxel octree-analysis, extensions, and implementation", NVIDIA Corporation, 2010.

[6] S. Laine and T. Karras, "Efficient sparse voxel octree", *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pp. 55-63, ACM, 2010.

[7] NVIDIA, "Cuda toolkit major components", CUDA Programming Guide 1.0, pp. 1-2, NVIDIA, Sep. 2016.

[8] Martin Patzold and Andreas Kolb, "Grid-free out-of-core voxelization to sparse voxel octrees on GPU", *Proceedings of the 7th Conference on High-Performance Graphics Conference,* pp. 95-103, Aug. 2015.

[9] Peter Shirley and Steve Marschner, Fundamentals of computer graphics, 3nd Ed., A K Peters, 2009.

[10] Theoharis Theoharis, Georgios Papaioannou, and Evaggelia-Aggeliki Karabassi, "The magic of the z-buffer: a survey", *Proceedings of the 2001 WSCG*, 2001.

[11] T. Whitted, "An Improved illumination model for shaded fisplay", *Communication of ACM,* Vol.23, No.6, pp. 343-349, 1980.

[12] Chris Wyman, Rama Hoetzlein and Aaron Lefohn, "Frustum-traced raster shadows: revisiting irregular z-buffers", *Proceedings of the SIGGRAPH '15 ACM SIGGRAPH 2015 Talks Article No. 69*, Aug. 2015.

A Rendering Method Using Sparse Voxel Octrees in Dynamic Scenes

Tsong-Wei Huang and Chin-Ho Cheng

Department of Computer Science and Information Engineering
Fu Jen Catholic University
New Taipei City, Taiwan 24205, R.O.C.

Abstract

Sparse Voxel Octrees (SVOs) are used as the representations of large static scenes. SVO has simple structure and level-of-detail information. So, rendering a large static scene with SVO representation becomes time efficient. But if we consider the dynamic scenes with SVO representation, then we have to reconstruct the SVO in response to the object movement in the scene. Its time is inefficient. In this paper, we propose a simple rendering method for objects with SVO representation in dynamic scenes. Our method avoids regular reconstruction of object's SVO. By using the techniques of Z-buffer and Shadow z-buffer, we render the scene with hidden-surface removal and shadow effect. Experimental results show that our method is correct and that implementation is easy.

Keywords: Sparse Voxel Octree, Dynamic Scene, Rendering, Hidden-Surface Removal, Shadow