三維幾何圖形即時壓縮之研究

A Study on Real Time Geometry Compression for 3D Objects

鍾斌賢* 林聰武+ 羅坤松*

*:中原大學 資訊工程學系

+:東吳大學 資訊科學學系

摘要

幾何圖形的壓縮與解壓縮技術可以減少網路傳輸時間與解決繪圖顯像硬體中記憶體匯流排頻寬不足的瓶頸，但在即時的要求之下，其速度仍是一大考驗。我們演算法的觀念是由一個邊可以找到對應的頂點，形成三角片，加上同心圓的構想，將幾何模型的三角片表示成二元樹的架構，針對此二元樹適度地編碼，可以完整地紀錄下三角片間的關係，而且又不會浪費太多記憶體空間，處理過程相當簡單快速。我們將此方法與現有的局部網格化演算法作實例比較，不論在壓縮比或是執行速度方面均比該方法優異。

關鍵字：幾何壓縮、電腦繪圖、即時壓縮

一、簡介

圖學影像技術蓬勃發展，虛擬實境（Virtual Reality）應用範圍日益廣泛，舉凡一些民生所接觸到的事物，都可以見到虛擬實境技術的影子。而這幾年也紅得發紫的網路多媒體技術，應用在3D圖形的表現上，最常見的莫過於建築物的導覽，如虛擬博物館、樣品屋導覽等等。對於這類的3D幾何圖形，大都以三角片組成，而這些由點、線、面爲內容的檔案容量，其實是不小的，在現今有限的網路頻寬之下，是相當不利的。若要將這些大量的3D模型即時地描繪出來，目前面臨最大的瓶頸，在於繪圖顯像硬體中的記憶體匯流排頻寬（Memory Bus Bandwidth）的不足。如文獻[4]中所舉的一個例子：假如1秒鐘要描繪三角片數目爲100萬片的幾何圖形30次，而每片三角片保守估計大約需要24個位元組的記憶體，那麼在中央處理器（CPU）與繪圖管線（Graphics Pipeline）之間的記憶體匯流排，每秒將需要720萬位元組的頻寬。對一個效能比較好的工作站來說，面對如此高的頻寬是不成問題；但是，對於較低階的工作站來說，卻是很難達到的。

基於上述的原因，如果將想要透過網路傳送的3D幾何圖形先行壓縮，將其檔案容量縮小，那麼傳輸的時間將大大地降低。若要描繪的3D模型，也事先予以壓縮而儲存在主記憶體中；當繪圖管線要進行描繪時，先透過即時的解壓縮處理器，就可以描繪出正確的模型；雖然多了壓縮和解壓縮的動作，但是，記憶體匯流排頻寬不足的問題卻可以迎刃而解，使得較低階的工作站也能克服原先無法處理的問題了。

二、相關研究討論

最先提出3D幾何圖形壓縮的是Deering[1]，利用一個容量爲16的暫存器，儲存會被重複利用的點資訊，這些資訊包含點的座標值、材質屬性、法向量等，而這個暫存器，被稱爲網格暫存器（Mesh Buffer）。因此，他將一般化帶狀三角片（Generalized Triangle Strip）的表示法加上網格暫存器的配合使用，就稱爲一般化三角網格（Generalized Triangle Mesh）的表示。雖然他最先提出一般化三角網格的觀念，但是並未詳細描述如何找出一般化三角網格。Taubin及Rossignac [3]首先提出一般化三角網格的產生方法：先對幾何模型的拓樸結構加以分析，由三角片的頂點建成一棵生成樹（Spanning Tree），因此可得到頂點生成樹的邊。然後根據這些邊，如同剝橘子皮的方式，將幾何模型作線條狀的切割；最後可以得到一串一串的帶狀三角片（Triangle Strip），形成三角片生成樹。這種方法的結果雖然很好，但是，演算法過於複雜，加上解壓縮的程序需要相當大的記憶體空間，造成整體的成本提高，非常不適合應用於即時的硬體設

計方面。

Chow [4]提出了一個局部網格化（Local Meshify）的演算法；首先，先找到一組網格邊界的邊，然後從這組邊之中的每每相鄰兩邊找到一串相鄰的帶狀三角片，所以整組邊都找完之後，可以得到第一串的帶狀三角片。接著以第一串帶狀三角片邊界的邊，重複剛剛的動作，最後即可以得到許多串的帶狀三角片。邊的數量必須配合網格暫存器的大小，如果網格暫存器大小爲 K，則邊的數量就爲 K-1 個，使得共用點的資訊都能事先暫存在網格暫存器當中，將網格暫存器的功能充分利用。如果要避免有孤立的三角片未尋獲，一般化三角網格就盡量以螺旋狀的方向產生；因此，第一組網格邊界的邊，如果是星形或是扇形三角串的一部份的話，螺旋狀的型式就會出現。

Gumhold [6]提出另一種找尋一般化三角網格的方法，首先，在幾何圖形中任意選一片三角片，然後由三角片的邊依序往外去尋找相鄰的三角片，每搜尋到一片新的三角片，就對該三角片的邊編號。依照編號的順序，一直搜尋相鄰的三角片，直到所有的三角片都找到爲止。

Gumhold[6]依照尋找三角片的過程，定義了一些運算元，我們可以從這些運算元知道哪種情況需要多少個點、邊、以及會不會產生新的三角片；因此，可以將所找到的三角片順序，由三角片的點的邊號、三角片的編號、以及運算元等，組成一個表示式。

三、方法論

對整個 3D 幾何圖形找尋出合適的一般化三角網格的探索過程中，我們最初的靈感是來自於將一顆小石頭投入一面平靜的湖泊中，這時湖面上會激起一環一環的水波，有如同心圓的圖案一般。這樣的尋找過程與 Gumhold[6]的想法類似，但做法不同，更爲簡單；我們很單純地由一個邊，可以找到一個點與之形成一片三角片，不像 Gumhold[6]將尋找的結果分類成許多種情況。現在就以圖 1 來詳細說明尋找一般化三角網格的過程。

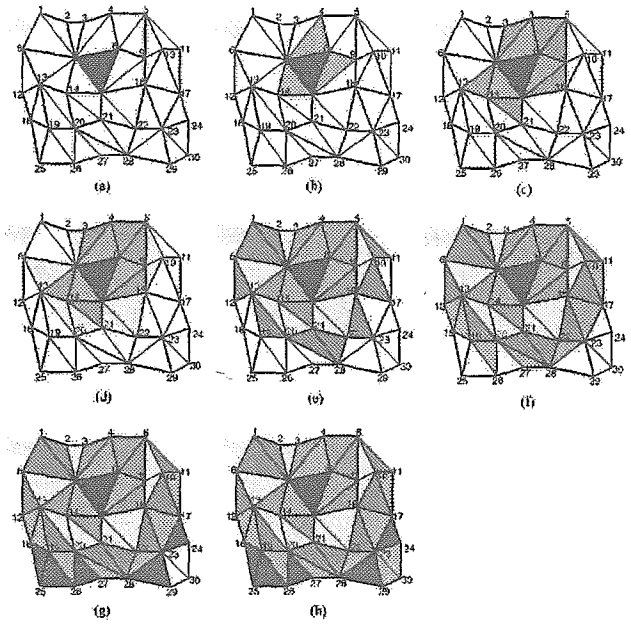


圖 1：以廣度優先（Breadth-First）的方式，對幾何圖形作搜尋

首先，在幾何模型的三角片中，找到一片分支度(Degree)最多的三角片，因此可以得到此三角片頂點（7、8、15）的相關資訊，並且儲存起來，如圖 1(a)；這樣的選擇，一方面增加了該三角片的頂點被共用的機會，另一方面這三個頂點的相關資訊卻不必儲存在網格暫存器中。爲什麼呢？我們可以從圖 1(b)到圖 1(e)得到答案：在圖 1(b)中，我們從第一片三角片的三個邊分別去尋找三個頂點，並將所找到的頂點儲存起來，以得到三片相鄰的三角片，這三個頂點就是 4、9、14，而且這第二層的三角片好比同心圓的第二環。接下來以第二環三角片的邊，繼續往外擴展，尋找相關的頂點以形成第三環的三角片，如圖 1(c)所示，所得到的頂點分別爲 3、5、5、16、21、13，並將它們儲存起來；此時會發現頂點 5 會被重覆使用，所以頂點 5 的相關資訊必須儲存在網格暫存器中。接下來的動作，再以第三環三角片去尋找第四環的三角片，如圖 1(d)；同樣地，可以由第四環的三角片得到第五環的三角片，如圖 1(e)。到此爲止所找到的三角片，已經將第一片三角片的頂點（7、8、15）都環繞住了，而且從第二環至第五環所找到的頂點中，並沒有頂點 7、8、15 的蹤跡；這是因爲我們從一個邊去找到相鄰三角片的頂點，而這個邊的其中一點，已經包含頂點 7、8、15 了，除非頂點 7、8、15 已經被三角片所環繞；所以，找到的第三頂點，當然就不會是 7，8，15。

一直重覆由內環三角片找外環三角片的動作，直到幾何模型中所有三角片都已經拜訪過爲止，這個一般化三角網格的尋找才告結束。統計一下所找到頂點的數目，爲三角片個數加 2，這個數目也就是帶狀三角片的最佳解。這個結果是可以預期的，因爲由一個邊去找所對應的頂點，除了不會忽略掉任何一片三角片之外，所找到的三角串一定是帶狀三角片，因此，頂點數目當然是三角片數目加 2。

在這樣的尋找過程中，其實可以用三棵二元樹所形成的樹林來表示，第一片三角形有三個邊，每個邊分別長出一棵二元樹。圖 2 就是根據圖 1 的尋找過程所建立的一棵二元樹，它具有下列的特性：

(1) 二元樹節點的內容爲三角片的三個頂點。

(2) 每一個階層的節點，就是一般化三角網格尋找過程中的每一環三角片。

(3) 節點中斜體底線的數字，代表尋找過程中，由該節點的父節點的邊，所找到可以形成三角片的頂點；樹根的節點除外。

(4) 二元樹的樹葉（Leaf），代表已無法由該葉節點的父節點，去找到相鄰的三角片。

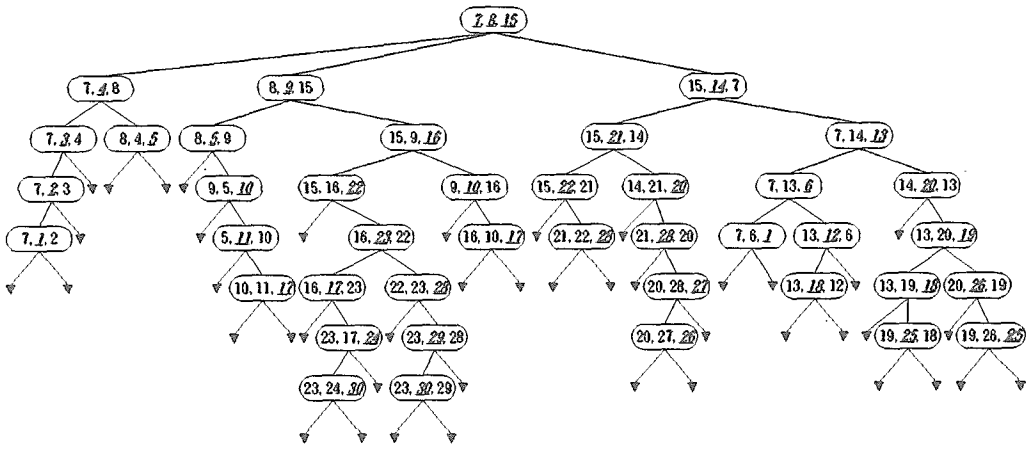


圖 2：根據一般化三角網格尋找過程所建立的二元樹

雖然可以將一般化三角網格的尋找過程建立成三棵二元

樹，以利於三角片的表示與壓縮的處理，但是，由於樹葉節點相當多，記憶體的使用量相對地會提高。基於這個原因，我們針對此二元樹編碼。利用兩個旗標（根節點必須要三個旗標），分別爲左子樹旗標與右子樹旗標，當二元樹節點有左（右）子樹存在，但不爲樹葉的話，該節點之左（右）子樹旗標爲 1；若爲樹葉，左（右）子樹旗標爲 0。而且不需要儲存整片三角片的三個頂點，只需將尋找過程中由邊所找到對應的頂點儲存起來即可。因此，編碼完成的一般化三角網格變成下列的碼：

{7, 8, 15 : 111}、{4 : 11}、{9 : 11}、{14 : 11}、{3 : 10}、{5 : 00}、
{5 : 01}、{16 : 11}、{21 : 11}、{13 : 11}、{2 : 10}、{10 : 01}、{22 : 01}、{10 : 01}、{22 : 01}、{20 : 01}、{6 : 11}、{20 : 01}、{1 : 00}、{11 : 01}、{23 : 11}、{17 : 00}、{28 : 00}、{28 : 01}、{1 : 00}、{12 : 10}、
{19 : 11}、{17 : 00}、{17 : 01}、{28 : 01}、{27 : 10}、{18 : 00}、
{18 : 01}、{26 : 01}、{24 : 10}、{29 : 10}、{26 : 00}、{25 : 00}、
{25 : 00}、{30 : 00}、{30 : 00}

其中的{7,8,15 : 111}代表第一片三角片的碼，（7、8、15）爲三角片的三個頂點；（111）中的第一個 1 表示由邊（7,8）可以找到相鄰的三角片（7,4,8），因此，將頂點 4 紀錄起來；（111）中的第二個 1 表示由邊（8,15）可以找到相鄰的三角片（8,9,15），因此，將頂點 9 也紀錄起來；（111）中的第三個 1 表示由邊（15,7）可以找到相鄰的三角片（15,14,7），因此，將頂點 14 也紀錄起來。第一片三角片的處理比較特別，接下來的{4 : 11}、{9 : 11}、{14 : 11}，由於只紀錄由邊所找到相鄰三角片的一個頂點，那要如何得知該頂點 4、9、14 所對應的邊的兩個頂點，以形成一片三角片呢？很簡單地，我們可以用一個佇列（Queue）來儲存這些三角片的相關資訊，利用廣度優先的尋找方式，配合所得到的編碼，即可以得到圖 2 中劃底線的頂點。

四、壓縮與解壓縮演算法

在這個章節裡，我們除了詳述本系統的核心，也就是壓縮與解壓縮的演算法之外，還會說明整個壓縮與解壓縮的程序。首先，說明演算法中兩個很重要的資料結構：

TriangleCode：爲一個結構陣列，內容爲由邊所找到相鄰三角片的一個頂點（此頂點與該邊形成一片三角片），與對一般化三角網格編碼時的左右子樹旗標。

ReusedVertexCount：爲一個整數陣列，內容爲 TriangleCode 的陣列中頂點的計數值。若該頂點的計數值大於 1，則頂點的相關資訊必須要儲存在 MeshBuffer 中，也就是共用點資訊。

(一)、 壓縮演算法：

```
// Step 1：找一片分支度最多的三角片。
Choose_First_Triangle (num_vertex, &first_index);
// Step 2：由第一片三角片的三個邊找到相鄰三角片的一個頂點。
ThirdVertex1 = Find_Third_Vertex (vertex1, vertex2);
ThirdVertex2 = Find_Third_Vertex (vertex2, vertex3);
ThirdVertex3 = Find_Third_Vertex (vertex3, vertex1);
ReusedVertexCount[vertex1]++;
ReusedVertexCount[vertex2]++;
ReusedVertexCount[vertex3]++;
//Step 3：將所找到的三個三角片的頂點儲存在佇列中。
Insert_Queue_Node (Q, vertex1, ThirdVertex1, vertex2);
Insert_Queue_Node (Q, vertex2, ThirdVertex2, vertex3);
Insert_Queue_Node (Q, vertex3, ThirdVertex3, vertex1);
i = 0;
```

```
while (Remove_Queue_Node (Q, &vertex1, &vertex2, &vertex3)) {
        // Step 4：由佇列中移除掉最先儲存的三角片頂點，再由此三角
                片的兩個邊去尋找相鄰三角片的一個頂點。
        left_vertex = Find_Third_Vertex (vertex1, vertex2);
        right_vertex = Find_Third_Vertex (vertex2, vertex3);
        // Step 5：將所找到的一個頂點儲存以及計數。
        TriangleCode[i].vertex = vertex2;
        ReusedVertexCount[vertex2]++;
        // Step 6：如果 Step 4 可以找到相鄰的三角片，則必須再將新的
                三角片的頂點儲存在佇列中，並且編碼。
        if (left_vertex) {
          Insert_Queue_Node (Q, vertex1, left_vertex, vertex2);
          TriangleCode[i].LeftChildFlag = true;
        }
        else
            TriangleCode[i].LeftChildFlag = false;

        if (right_vertex) {
          Insert_Queue_Node (Q, vertex2, right_vertex, vertex3);
          TriangleCode[i].RightChildFlag = true;
        }
        else
            TriangleCode[i].RightChildFlag = false;
        i++;
}
```

在這個演算法中，最主要的動作就是由一個邊去尋找相鄰的三角片，然後視情況決定是否儲存在佇列中；也就是Find_Third_Vertex ( ) 與 Insert_Queue_Node () 所處理的工作。演算法結束的條件，就是當佇列是空的時候，也就是已經無三角片存在。

(二)、 解壓縮演算法：

```
index = rear_flag = front_flag = 0;
// Step 1：由第一片三角片的碼可以得知相鄰三個三角片的資訊，並且
        存入一個三角片頂點與編碼的暫存器 stripe。
stripe[rear_flag] = FirstTriangle->FirstStripe;
stripe[rear_flag].LeftChildFlag = TriangleCode [index].LeftChildFlag;
stripe[rear_flag++].RightChildFlag=TriangleCode[index++].RightChildFlag;
stripe[rear_flag] = FirstTriangle->SecondStripe;
stripe[rear_flag].LeftChildFlag = TriangleCode [index].LeftChildFlag;
stripe[rear_flag++].RightChildFlag=TriangleCode[index++].RightChildFlag;
stripe[rear_flag] = FirstTriangle->ThirdStripe;
stripe[rear_flag].LeftChildFlag = TriangleCode [index].LeftChildFlag;
stripe[rear_flag++].RightChildFlag=TriangleCode[index++].RightChildFlag;
while ((front_flag-1) != rear_flag) {
        // Step 2：由之前所得到的三角片與碼，可以透過該三角片的邊，
                找到相對的頂點。
        if (stripe[front_flag].LeftChildFlag) {
          stripe[rear_flag].vertex1 = stripe[front_flag].vertex1;
          stripe[rear_flag].vertex2 = TriangleCode [index].vertex;
          stripe[rear_flag].vertex3 = stripe[front_flag].vertex2;
          stripe[rear_flag].LeftChildFlag = stripe[index].LeftChildFlag;
```

```
        stripe[rear_flag++].RightChildFlag = stripe[index++].RightChildFlag;
      }
      if (stripe[front_flag].RightChildFlag) {
        stripe[rear_flag].vertex1 = stripe[front_flag].vertex2;
        stripe[rear_flag].vertex2 = TriangleCode [index].vertex;
        stripe[rear_flag].vertex3 = stripe[front_flag].vertex3;
        stripe[rear_flag].LeftChildFlag = tstripe[index].LeftChildFlag;
        tripe[rear_flag++].RightChildFlag = stripe[index++].RightChildFlag;
      }
      front_flag++;
    }
```

解壓縮的演算法，很簡單地只是對 TriangleCode 的資料，作一個原先的組合，front_flag 與 rear_flag 爲佇列的索引旗標。演算法的結束，也是當佇列爲空的條件成立。

(三)、 壓縮程序：

(a) 讀取 3D 幾何圖形檔案，分析三角片的資訊，以建立 3.5.1 節中的幾何模型的原始資訊一、二的資料結構，並將相關資訊寫入壓縮完成的資料檔頭。

(b) 執行壓縮演算法，以建立 TriangleCode 與 ReusedVertexCount 的資訊。

(c) 根據 TriangleCode 的內容，將相關的頂點座標與法向量加以量化，並利用 ReusedVertexCount 的內容判別屬於何種壓縮完成的資料格式，是否需要將該頂點儲存在 MeshBuffer 中，最後將合適的資料格式寫入壓縮的檔案。

(d) 將壓縮檔案再利用 zlib[9]的壓縮函式庫，作最終的資料壓縮。

(四)、 解壓縮程序：

(a) 將壓縮檔案先利用 zlib[9]的解壓縮函式庫，做最初的解壓縮處理。

(b) 讀取壓縮檔案的資料檔頭，以得知幾何模型的相關資訊。

(c) 依次讀取壓縮資料的格式，並將頂點座標與法向量的資料還原，建立 TriangleCode 的資料，直到壓縮檔案結束。

(d) 執行解壓縮演算法，由 TriangleCode 的內容建立原來三角片的結構，並輸出至檔案。

五、實驗結果與討論

我們將本系統壓縮與解壓縮演算法的結果，與 Chow[4]的局部網格化演算法，在壓縮量與速度兩方面作比較。這些測試的 3D 幾何模型的類型，包括了地圖、動物、人類、建築物、與交通工具等等，種類相當多，檔案型態爲 Wavefront 的 ASCII 格式（*.obj）。在 "http://home.earthlink.net/~mmchow/gcompiler/gcompiler.html" 可以下載 Chow[4]壓縮與解壓縮程式 Sun Solaris 的版本，我們給予 Chow[4]演算法的誤差限度 e = 0.1，這個值也是該程式的初始建議值；在給定誤差限度之後，可以得到相關的座標量化限度，根據此限度爲標準來測試我們的演算法。從實驗結果可以得知 Chow[4]的壓縮率爲 10 至 15 左右，但如果幾何模型檔案中不包含頂點法向量的資訊，壓縮率降爲 2 至 8 左右。相較之下，我們實驗結果的壓縮率大致都比 Chow[4]還好，但是，Chow[4]在壓縮程序的最後階段還經過 Huffman Encoding，降低檔案容量，而我們沒有使用 Huffman Encoding 的情況之下，壓縮率已經可以超越 Chow[4]。因此，爲了比較條件的公平性，最後兩者都經過 Zlib[9]壓縮；如此一來，我們所提出的演算法的壓縮率都比 Chow[4]還高。在執行速度的比較方面，實驗的環境爲 SUN SPARC-40MHz，主記憶體爲 64Mbytes 的工作站。由實驗的數據可以得知，我們的演算法速度相當地快，大大地降低整體壓縮與解壓縮程序的時間。而 Chow[4]在壓縮的處理中，由於局部網格化演算法的關係，必須適度地調整邊的數量；加上他也針對幾何模型做適度量化限度的的分析，因此，整體的執行時間表現上，相對地會增加許多。由於實驗的幾何模型相當多，因此，我

們只選取幾個形狀比較特殊的模型來展示。圖 3 至圖 6 中，分別都有 4 張小圖，編號爲 (a) 至 (d)；其中編號 (a) 爲原圖，編號 (b) 爲原圖的骨架，編號 (c) 爲 Chow[4]所壓縮完畢的結果，編號 (d) 爲我們所壓縮完畢的結果。

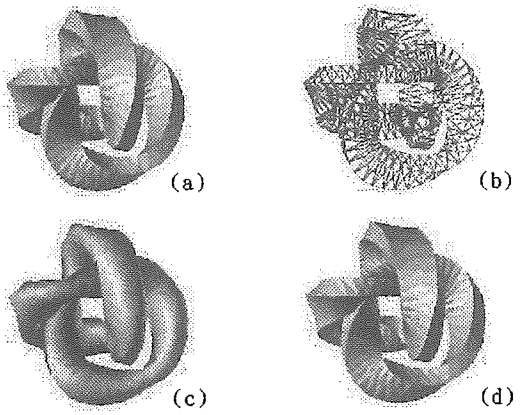


圖 3：幾何模型 tre_twist

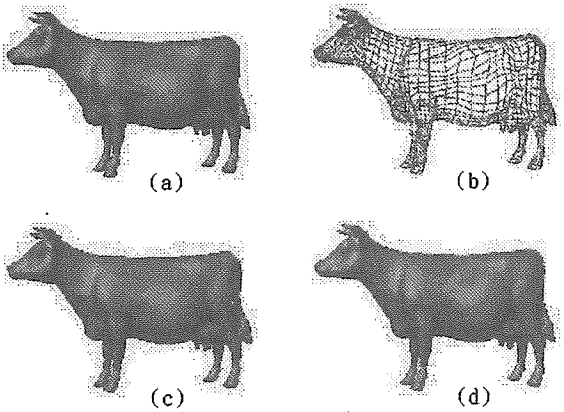
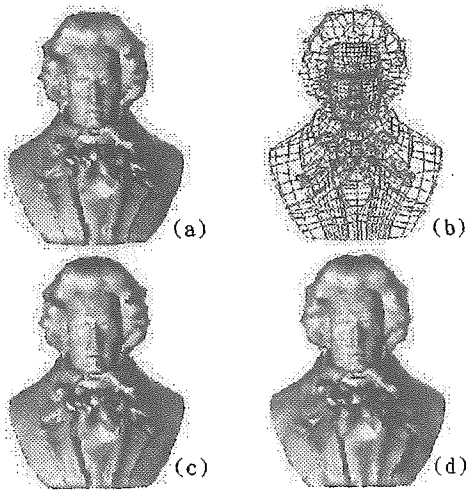


圖 4：幾何模型 cow



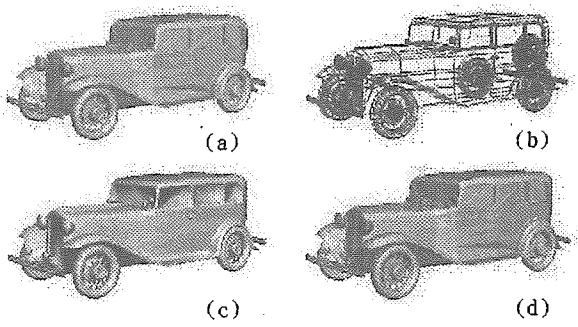


圖 6：幾何模型 32_dodge

Chow[4]的局部網格化演算法中所使用到的網格暫存器，其大小與 Deering[1]所提出的是一樣的，容量都是 16，而且重覆使用的頂點的機率，平均值爲 43%。相較之下，在我們的演算法中所使用到的網格暫存器，其容量爲 256，這個值是經過實驗所得到的。由於我們的觀點在於將一般化三角網格轉換成二元樹，而且處理的過程是以一個邊加上所對應的一個頂點而形成一片三角片，與 Chow[4]以帶狀三角片的臨界邊來找尋一般化三角網格的方式是不同的。Gumhold[6]也提到這種找尋一般化三角網格的方法，其網格暫存器的容量是很難證明出來的。在我們的實驗過程中，當網格暫存器容量爲 128 時，有些幾何模型還會發生頂點參考錯誤的情況，但是，當容量增加到 256 時，這些狀況就不再發生了。這個容量是以目前實驗的幾何模型而得到的數值，雖然幾何模型的形狀各異，三角片數目也有數百片至百萬片不等，但是，我們無法保證這個容量可以正確地處理其他更特殊的幾何模型。因此，基於這個考量，我們將網格暫存器的容量調整爲 16 來作測試，與 Deering[1]、Chow[4]相同。但是，有些應該儲存在網格暫存器中的頂點相關資訊會因爲網格暫存器容量太小而無法存入，必須額外記錄在壓縮的檔案中，而且當有資料需要存入網格暫存器之前，必須先搜尋適合儲存的位置，以免發生資料存取錯誤的情況發生；相對地，幾何圖形的壓縮率也會因

而降低，不過，經過這樣的處理，我們可以保證所有的幾何模型不會因爲網格暫存器容量的大小而發生資料存取錯誤的情況。我們將網格暫存器的容量調整爲 16 的實驗結果，三角片數目較少（數百片左右）的幾何模型不會因爲網格暫存器容量的調整而影響到壓縮效果，但是，三角片數目較多的幾何模型，其壓縮率卻因而降低了。在實驗的執行速度方面，網格暫存器容量的調整並沒有造成太大的影響，其壓縮與解壓縮程序執行的時間並無太大的差異。

六、結論與未來展望

在本論文中，我們提出了一個相當簡單又快速的演算法，可以有效地分析 3D 幾何模型的資訊，進而將之編碼成一般化帶狀三角片。經過了相當多的幾何模型測試，壓縮率大約爲 11 至 18 倍左右，執行速度方面，大大地超越 Chow[4]的結果。由於我們的壓縮與解壓縮演算法相當簡單，而且利用對二元樹編碼的技巧，大大地降低記憶體的使用量，因此，相當適合實現在硬體架構上。如此，壓縮與解壓縮程序的速度也將增進許多。

在未來的發展過程中，有下列幾個重點：

1. 在壓縮演算法中，常常需要讀取及查詢幾何模型的資料，而且選定第一片分支度最大的三角片也浪費相當多的時間，因此，需要再進一步研究較合適的資料結構。
2. 因爲量化的技巧可以降低檔案容量，但卻成爲失真的壓縮，所以，利用新的頂點座標、法向量的壓縮技術，在人類視覺及整體壓縮率之間，如何找到最佳的平衡點，將是一個相當重要的研究主題。
3. 針對幾何模型中各種不同區域，給予適合的量化限度。
4. 結合多精度（Level of Detail）的漸進式顯像（Progressive Transmission）。

參考文獻

[1] Deering M. Geometry Compression. Computer Graphics(Proceeding of SIGGRAPH), PP.13-20, August, 1995

[2] Francine Evans, Steven Skiena, and Amitabh Varshney. Optimizing Triangle Strips for Fast Rendering（Proceeding of the IEEE Visualization）, PP.319-326, October, 1996

[3] Gabriel Taubin, and Jarek Rossignac. Geometric Compression Through Topological Surgery. IBM RC-20340（http://www.research.ibm.com/vrml/binary）, 1996

[4] Mike M. Chow. Optimized Geometry Compression for Real-time Rendering（Proceeding of the IEEE Visualization）, PP.347-354, 1997

[5] Richard S. Wright, Jr. and Michael Sweet. OPENGL SUPERBIBLE. Waite Group Press™. 1996

[6] Stefan Gumhold, and Wolfgang StraBer. Real Time Compression of Triangle Mesh Connectivity（Proceeding of SIGGRAPH）, PP.133-140, July, 1998

[7] Sum Microsystems, Inc. Java 3D™ API Specification, Version 1.1（http://java.sun.com/products/java-media/3D/）, December, 1998

[8] 杜家玫. 網際網路上幾何壓縮技術之研究. 國立清華大學 碩士論文. 1998

[9] Jean-loup Gailly and Mark Adler, zlib Version1.1.3（http://www.cdrom.com/pub/infozip/zlib/）