高等計算機圖學 HW4 Report &操作說明

ntu_r06922063 邱政凱

(對於複雜的模型(bunny)，我的程式可以達成約 300 倍的加速)

使用教學:

可以直接執行 hw4.exe。它會自動讀取同個路徑下名為 input.txt 的檔案當作輸入。若要自行指定檔名，可使用 cmd

./hw4 “檔名”

的指令，以參數的形式把要使用的 input 檔名傳入。(預設 input.txt 是複製 rabbit.txt 的內容)

(老師提供的兩個 input 我分別命名為 monkey.txt 和 rabbit.txt)

程式執行完會輸出一個名為"colorOutput.ppm"的 ppm 圖片檔，請使用 infraview 來檢視。

專案中有一個 codeblocks 的專案檔 hw4.cbp，可以使用此專案檔配合 code blocks 來開起來編譯。

Hw4.cpp： 這次作業的程式本體

algebra3.cpp: 這次作業使用老師提供的 algebra3 函式庫

imageio.cpp : ppm 檔案相關的讀寫操作。

並且，我有稍微更改了 input 的格式

在每個"單一"物件的所有 triangle mesh 的敘述出現之後之後加上一個字元"O"的 line 來代表以上的所有 MESH 代表一個單一物件，需要有自己的 KD-TREE。這樣做的目的是為了避免 bounding box 差距太大的兩個物體共用同一個 kd-tree，會讓 kd-tree 的加速效果退化非常多。
例如老師的範例場景中，動物模型再靠右側，但是地板平面幾乎橫跨畫面的 x 方向，如果兩者共用同一個 KD-tree，會變成動物模型任何一個 kd-node 中只要有包含到地板平面的 mesh，他的 bounding box 幾乎就會橫跨整個畫面的 x 軸，造成很容易打中某些 Bounding box，但是實際上該 bounding box 的 mesh 只有兩個三角形。

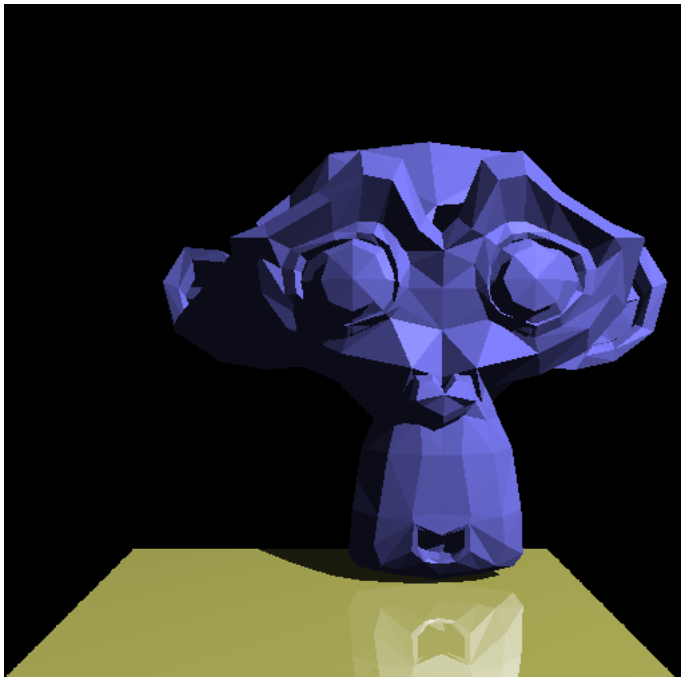
我把老師的 monkey 和 bunny 模型的模型本體跟地板的 plane 分別弄成兩個不同的物件
要測試請直接使用資料夾中的 monkey.txt 和 rabbit.txt，裡面為我已經加上額外兩行"O"的物件描述的版本

這次的渲染加速作業，我實作了 KD-Tree，將場景動態依照每個物件個別化分成不同的 kd-tree，然後建構自己的 tree(在範例 input 中，底下的 plane 跟動物模型兩者就是不同的 object，所以兩個檔案各有兩個 kd-tree)，詳細步驟與心得在結果下面。

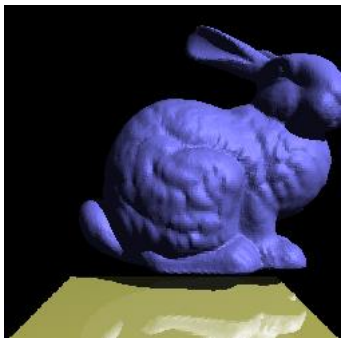
結果圖:　(原尺寸，使用 Left-hand rule)

Monkey 256*256

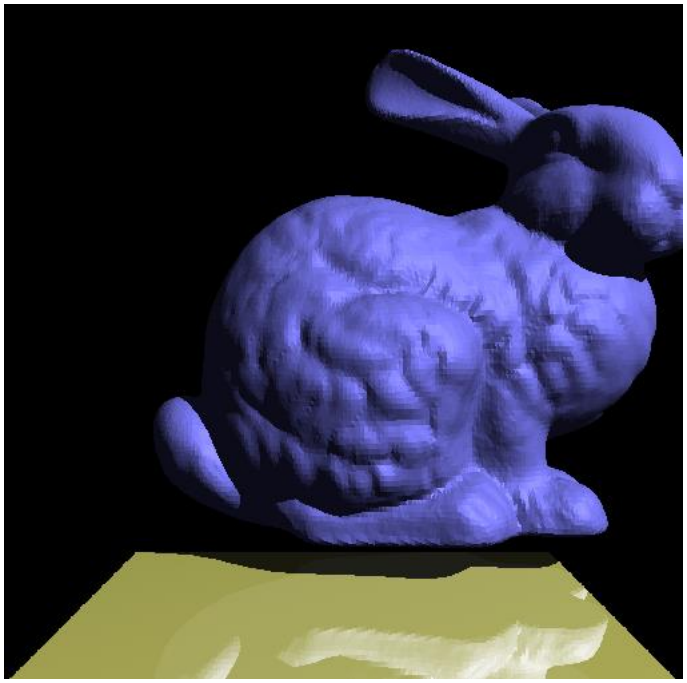
Monkey 512 * 512



Bunny 256 * 256



Bunny 512 * 512



執行時間 & 相關數據： (原本的速度請參照 HW3 Report，量測時間只計算 render time，不包含建構 kd-tree 的 pre processing 時間)

設備: Interl core i5-7200U 2.5GHz

Monkey 256 * 256

render time: 2.137 seconds (原本: 14.293 SEC)

total kd-tree node count: 138
max depth of kd-tree subdivision: 9
average ray-mesh Intersection check count: 133.332

Monkey 512 * 512

render time: 7.833 seconds (原本: 73.435 SEC)

total kd-tree node count: 138
max depth of kd-tree subdivision: 9
average ray-mesh Intersection check count: 133.237

Bunny 256*256

render time: 5.576 seconds (原本: 1627.135 SEC)

total kd-tree node count: 18094
max depth of kd-tree subdivision: 17
average ray-mesh Intersection check count: 128.707

Bunny 512 * 512

render time: 21.031 seconds (原本: 5780.73 SEC)

total kd-tree node count: 18094
max depth of kd-tree subdivision: 17
average ray-mesh Intersection check count: 128.655

從以上的數據可以看到，在小型的模型(monkey)裡，KD-tree 的深度只有 9，總共建構了 138 個 kd-node，並且對於每次要進行 ray-triangle intersection 時，平均需要檢查的數量從原本的 980 個(與所有 mesh 進行檢查)變成只需要檢查 133.33 個。相較於原本的速度大約快了 7~10 倍。

在大型的模型(bunny)中，KD-Tree 的深度是 17，總共建構了 18094 格 kd-node，並且對於每次要進行 ray-triangle intersection 時，平均需要檢查的數量從原本的 70000 個 MESH 變成只需要 128.655 個

mesh，因此速度提升了大約 300 倍！

儘管對於較小的模型，可能因為平均 ray-triangle intersection 檢查數量減少所提升的效能不足以完全彌補檢查 bounding box 與 ray 的 intersection 花的時間(大約平均提升 7~10 倍的速度)，但是實際上真正的效能瓶頸一般還是大型、複雜的物件，我的實作確實有效提升了對於大型複雜物件的渲染速度到達約 300 倍的加速。

實作方法&心得:

我實作 KD-Tree 的方法主要是先把單一物件中的所有 mesh 放進 root kd-node 中，接著從這些 mesh 中估計出一個可以把所有 mesh 包覆在內的 bounding box，接著依照此 bounding box 中最長的 axis 來當作劃分左右子樹的依據，找出此 kd-node 中所有 Mesh 對應 axis 的值，進行排序，並且取出中位數當作左右子樹的臨界值 (例如 x 軸最長，則從所有 Mesh 中找出他們的 x 值的中位數，把所有 mesh(只要有任何一個頂點)中 x 值小於此值的 mesh 丟到左子樹，大於此值的 mesh 丟到右子樹)，並且遞迴地建構 kd tree，直到某一層他劃分出來的左右子樹中的 mesh 的重疊比例高過一定的 threshold(根據模型大小計算)就停止繼續建構左右子樹。(因為 triangle mesh 有三個頂點，所有有時單一 mesh 會同時被丟到左右子樹中，當同時存在左

右子樹裡的 mesh 的比例跟總 mesh 樹的筆直高到一定程度，我們可以想見再繼續建構子樹也沒有太大的意義，故停止)。這個比值的 threshold 是根據模型大小來計算的，取法為 “同時存在於左右子樹中的 mesh 數” / (父節點 mesh 數 / log10(total mesh number))。

以上是建構 kd-tree 的部分，而實際在進行 hit test 的方法，主要就是每次要判斷某個 ray 會打到場上哪些物件時，就對於所有的 object 的 root KDnode 做 hit test，(hit test 演算法參照的是: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection 這個網址的實作方式)，若有打中 root node，則進一步檢查是否有打中左右子樹的 bounding box，若有，則往該方向的子樹進一步進行以上的步驟，遞迴地直到檢查到 leaf 節點為止。到達 leaf 節點時，則對於節點中包含的所有 mesh 進行 ray-triangle intersection test。藉由這樣的方法，我們可以有效的減少需要檢查 triangle intersection test 的 mesh 數量。