國立暨南國際大學資訊工程研究所

碩士論文

具有整合性場景管理系統的3d成像引擎之開發

Developing a Real-Time 3d Rendering Engine with an Integrated Scene Management System

指導教授：陳履恆 博士

研究生：黃禹翔

中華民國九十七年六月

論文名稱：具有整合性場景管理系統的 3d 成像引擎之開發

校院系：暨南國際大學科技學院資訊工程系　頁數：41

畢業時間：九十七年六月　學位別：碩士

研究生：黃禹翔　指導教授：陳履恆 博士

中文摘要

在近代，3D 虛擬實境已擁有大量的應用；例如工業上的輔助設計，或是娛樂業的電影 CG 動畫與近期蓬勃發展的遊戲產業。而在遊戲產業龐大的市場商機下，也讓 GPU 的發展速度超過了著名的摩爾定律。

雖然 GPU 擁有龐大的運算能力，然而我們認為，要能有效發揮 GPU 的運算效能還必須搭配上良好的繪圖引擎。而良好的繪圖引擎必須要有好的場景管理系統；因為即使現在的 GPU 效能已有很大的提升在即時運算的情況下仍然不能將所有的資料全部送到 GPU 中進行繪製。因此業界的繪圖引擎為了減少送到 GPU 處理的資料量都會使用各自的場景管理演算法來進行 culling。

一個良好的場景管理系統將能快速的決定場景中的可視物件以減少將不可視物件送入 GPU 中進行繪製所造成的浪費；帶來的益處即為能有效提高單一 frame 的場景精緻與複雜度。

因此在一個高效率的繪圖引擎中，場景管理系統將扮演一個非常重要的腳色。然而，在這些商業化的主流繪圖引擎中，通常並不會公開他們的演算法細節。因此我們打算進行研究並公開我們的演算法以降低進入這項領域的門檻。

關鍵詞：CSG、場景管理、BSP Tree、Portal rendering

Title of Thesis: Developing a Real-Time 3D Rendering Engine with an Integrated Scene Management System

Name of Institute: Department of Computer Science and Information Engineering, National Chi Nan University Pages: 41

Graduation Time: 2008/06 Degree Conferred: Master

Student Name: Yu-Hsiang Huang Advisor Name: Dr. Lieu-Hen Chen

Abstract

In recent years, the virtual 3d system brings many kinds of application. Take something for example, computer-aided design for industry or fantasy CG movies and video games for entertainment. As the markets of video games grow hugely, the improvement of GPU speed is fast than the famous Moore's law.

Nowadays GPU can do large data computing; however we think in order to use it well without any wasting, we must have an effective 3D engine for applications. An effective 3D engine must have good scene management system, because GPU still can' t render all objects even its data computing is more powerful than past. To reduce the objects rendered by GPU, most of commercial render engines will use their culling algorithms to achieve the goal.

The main purpose of scene management is to fast determine which object is visible in the scene, and save the computing of objects that are invisible. The benefit is that we can render more complex objects and more detail scene in single frame.

According to the above reasons, scene management plays an important role in an effective rendering engine. However, the commercial rendering engine will not make their algorithm public. Hence we want to research and development an effective rendering system with integrated scene management algorithm.

Keywords: CSG, Scene management, BSP tree, portal rendering

內容目錄

中文摘要 I

Abstract II

內容目錄 III

第一章 序論.. 1

1.1 研究背景 1
1.2 研究方法 3

第二章 系統理論與實做 4

2.1 CSG 原理與方法 4
2.2 封閉偵測與 Edge tracing 演算法 10
2.3 Edge tracing 與 T-Junction 修正 17
2.4 BSP tree 理論簡介 19
2.5 建立不分割場景 polygon 的 BSP tree 26
2.6 系統流程 30
2.7 結果比較 32

第三章 視覺效果 34

3.1 Lighting model 34
3.2 Normal mapping 36
3.3 Environment mapping 37
3.4 Shadow effect 39

第四章 結論 40

參考文獻 41

第一章 緒論

1.1 研究背景

由於場景管理在即時繪圖引擎中扮演極重要的腳色，因此想要開發一高效率的繪圖引擎就必須從此開始做起。而從國外的技術開始研究將能快速切入此領域，因此底下將說明目前業界主要常用的方法。

Frustum culling

視截頭體演算法，這是最基本的 culling 演算法，藉由算出視者可見區域來對物件進行可視偵測。如果一個物體在視截頭體裡則將此物體送入 GPU 中進行繪製，反之則否。而為了簡化每個物體的偵測運算量，通常會使用 bounding volume 來簡化一個複雜的物體，常見的有 bounding box、bound sphere 等等。

另外除了 occlusion culling 外其他的 culling 演算法都需要 frustum 來協助進行 culling，因此是最基本也最必要的 culling 演算法。

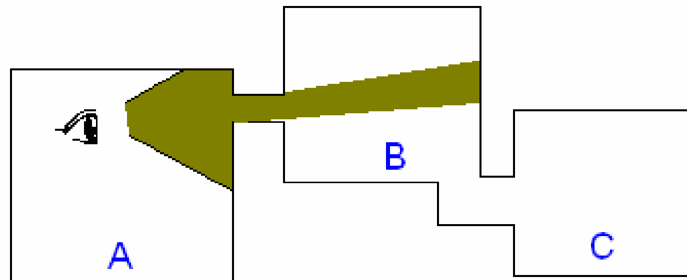
BSP culling

BSP tree 為一不斷將空間以平面分割為二的樹狀結構，其歷史最早可以追溯到兩篇由 Fuchs, Kedem and Naylor 提出的學術論文，第一篇並非提出 BSP tree 結構但卻是前置作業導致後來提出 BSP tree 成為解決方案。而 BSP tree 結構在 1996 年由 id software 知名的 3D engine 設計大師 John Carmack 應用在其 quake 遊戲 engine 後，使得 BSP tree 成為遊戲業內經常使用的加速結構；另外光跡追蹤也常用此結構做加速，並且也依分割面選擇法的不同發展出 kd-tree 結構。

Portal culling

我們認為，在複雜的室內環境中，使用 portal culling 是最好的方

法；由於室內空間經常藉由門口、窗戶與其他房間相連，而當由窗或門口看入其他空間時可縮小 frustum 的範圍，因此可以很快並且不需要龐大的計算即能做到 occlusion culling 的效果。



Portal culling 示意圖

Octree or Quadtree culling

Octree 為將空間分為等大小的八個子區域，非常適合用來管理戶外場景，而若空間中物件分部高度沒有太多變化則可以使用 Quadtree 即可。然而室外場景通常具有大量的物件，因此某些時候須要搭配 occlusion culling 以減少 GPU 繪製的物件數。

Occlusion culling

在一 3D 場景中，由觀察者的位置視野可以將場景中的物體分為兩種情形，遮蔽物(occluder)與被遮蔽物(occludee)，遮蔽物代表相對離觀察者位置較近的物體，而被遮蔽物則較遠。通常在一廣大戶外場景中將會有大量的物件在觀察者的視野中無法被移除掉，所以 occlusion culling 在此情況顯得格外重要。若能偵測哪些物體被遮蔽將能減少不必要的繪製作業，而這種情形前面提到的演算法都無法進行偵測並移除。早期的 occlusion 偵測都由 CPU 來做，但由於需要對物體依遠近做排序的動作，因此需要 CPU 進行大量的運算而很難自由應用在即時繪圖中。

並且使用時通常都需要預先指定較大的物體為 occluder。然而自從

GPU 支援 per-pixel image based occlusion 偵測後，在設計 occlusion culling 的演算法上簡化了不少。不過在使用 GPU 進行 occlusion 偵測卻不利一般 CPU 與 GPU 平行運算的設計架構。一般繪製流程 CPU 主要的工作只需要準備必須的資料然後交給 GPU 做運算，之後 GPU 會將工作指令存在 Queue 中依序完成。一但 CPU 要從 GPU 得到某一工作的結果，就必須等待直到 GPU 將此工作完全處理完成為止，因此在設計上還是需要特別注意。

一般來說為了減少等待的時間，仍然會使用簡化的 Bounding Volume 來代替複雜的模型做偵測以減少資料處理的運算量。另外由於 CPU 的發展已經走向多核心的設計，使用 multithread 將有機會成為未來的主流解決辦法之一。

1.2 研究方法

由上面的介紹我們可以知道每種演算法都各自擁有其長處與短處，要使用單一的演算法將很難有效率地管理一個龐大的場景；因此我們希望整合上面的演算法來實現高效率的繪圖引擎。

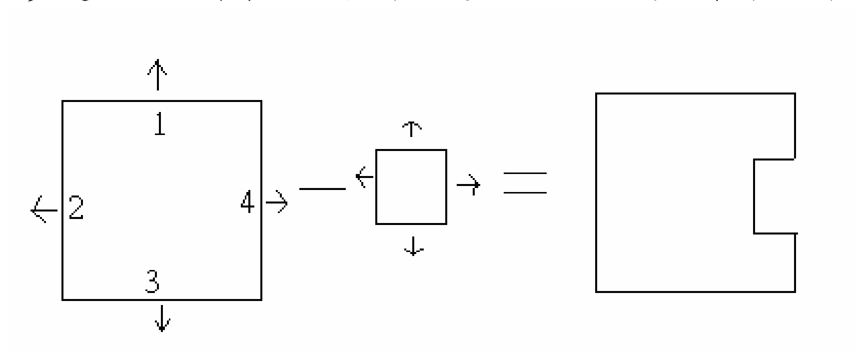
顯然地為了要實現不同的管理方式，我們將需要把場景劃分由各子 zone 構成，主要用意就是實現所謂的 Divide-and-Conquer。而依據 zone 的特性可以使用不同的演算法。我們將用 BSP + portal 管理室內場景，而 Octree 管理室外場景。為了實現這樣的想法，場景的建立將是一個必須要解決的問題，因此我們自行開發了 CSG 場景編輯器來進行研究。

第二章 系統理論與實做

2.1 CSG 原理與方法

Constructive Solid Geometry〈CSG〉是現在許多製作 3D 模型軟體常用的方式，主要是利用各種簡單的幾何原型〈geometric primitives〉經過 "加減" 操作之後拼湊出複雜的模型，那麼加減操作是怎麼回事？以下將簡單說明。

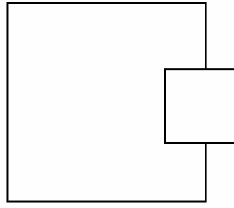
常見的 CSG 幾何原型有 Cube、Cone、Sphere、Cylinder......等等，很多應用軟體會稱他們為 Brush。〈通常為凸多邊形，因為所有凹多邊都可以用凸多邊組合出來〉，如下圖就是一個 CSG 減法操作的圖示：



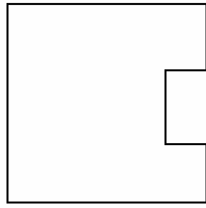
在大方塊的右方減去一個小方塊

那麼這些加減的操作是怎麼做的呢？為了說明方便，所用圖示將以 2D 的方式來表示，但在 3D 中原理都是相同的，只是實作時會比較複雜一點。

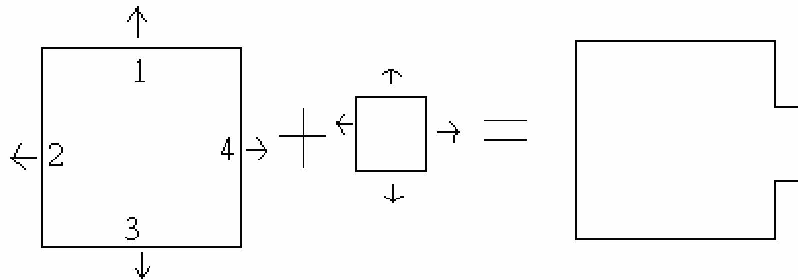
首先上圖為兩個實心的矩形箭頭代表各邊的正法向量，將它定為朝向方塊的外部，假設想要在大方塊的右邊減出一個空心的洞，首先對大方塊的每一個邊進行測試，測試其是否在小方塊的內部，很明顯的只有第四個邊會被檢測出，那麼這個邊就是非法的，應該消除。所以結果就變成下面這樣：



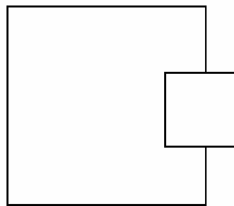
　　但是這還不是我們想要的結果，此時我們要對小方塊所有的邊作一次檢測，測試是否有邊落在大方塊的外部，如果有代表此邊是非法的，應該要消除。所以結果變成下面這樣：



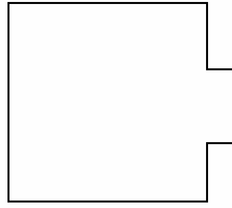
　　此時的形狀就是我們想要的結果了。而 CSG 另一加法操作也是利用同樣的方法，參考下圖：



現在我們想要在大方塊的右方加上一個小方塊，我們先對大方塊每一個邊作測試，一樣是測試是否有邊落在小方塊的內部，如果有就是非法的，必須將他消去，所以我們得到以下的圖示：



　　接下來對小方塊所有的邊作測試，這次則是要測試是否有邊落在大方塊的內部，如果有就是非法的，必須將它消除，經過這次測試就可以得到下圖的正確結果。



知道 CSG 加減操作的方式之後現在問題又來了，到底要怎樣檢測一個邊是否在一多邊形的內部或外部並且將不合法的部份消除呢？這個問題其實不難，因為前人已經想出來了，這個演算法有優點也有缺點，在此先介紹這個方法。請參考下圖：

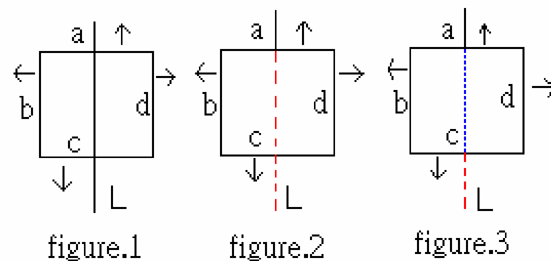
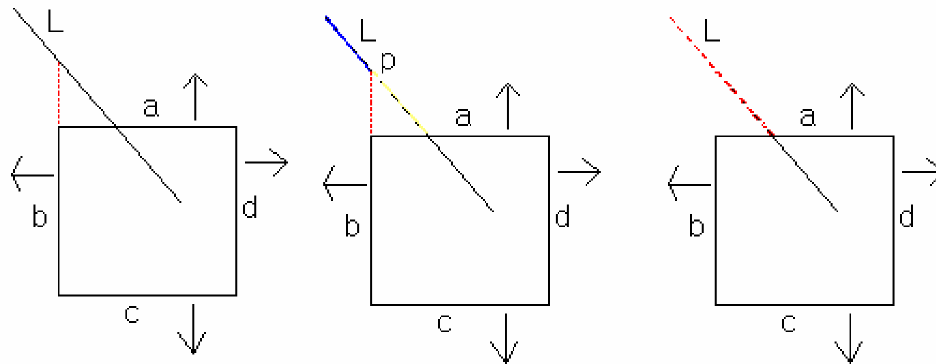


figure.1 figure.2 figure.3

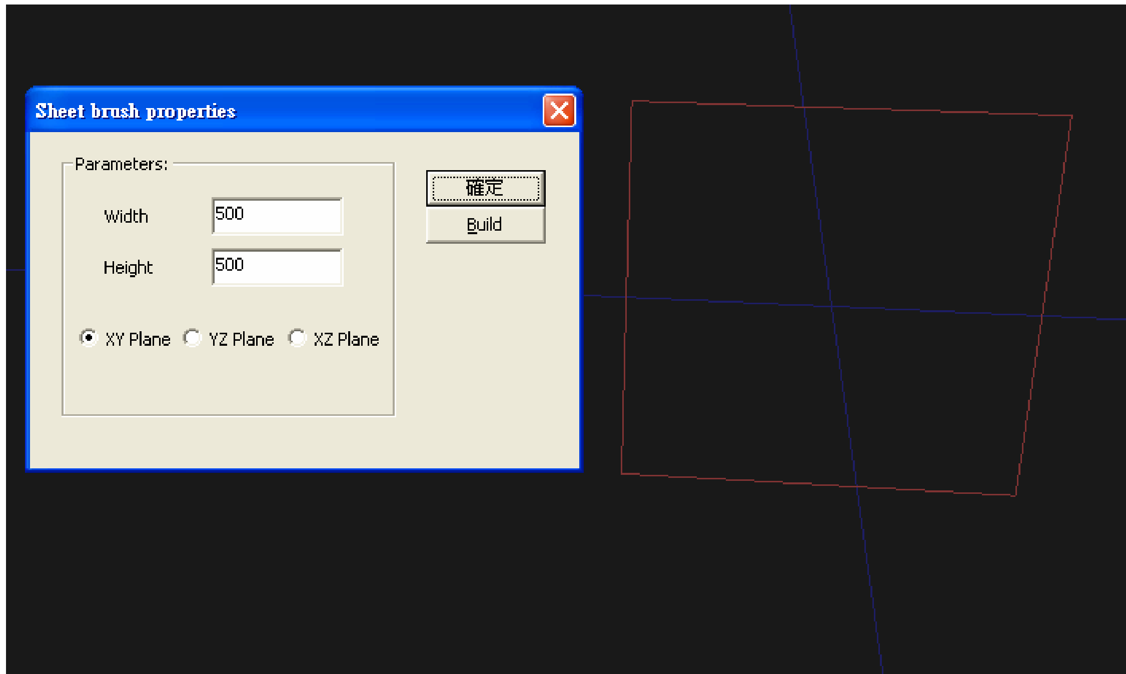
上圖 figure.1 中有一條 polygon L 穿過了方塊的四個邊〈在此因為視點與 polygon 共面所以看起來像一個線段，方塊的四個邊其實也是 polygon〉，一開始先將這多邊形與方塊的 a polygon 來做測試，很明顯兩 polygon 互相交錯，用程式檢驗只要將 polygon L 的所有頂點帶入 a polygon 的平面方程式，然後再比較正負號就可以知道 polygon L 與 polygon a 的關係；如果穿過了 polygon a 我們就將它分割，以 polygon a 分割成前方與後方。此時如圖 2，polygon L 黑色的部分已經可以確定一定是在方塊的外部，那麼我們接下來只要對紅色 polygon 的部份作測試就可以了。將 polygon L 紅色虛線的部分與 polygon b 做測試，結果是在 polygon b 的後方，暫時先不用處理這個情形；接著與 polygon c 作測試，發現又是交錯的情形，因此再一次分割紅色虛線的部份，如上圖 3 所示；原本紅色虛線現在被分割成藍色與紅色的部分，同樣的紅色部分已經確定是在方塊的外部了，因此接下來只要將藍色線段與 polygon d 作測試就行了；測試結果是在 polygon d 的後方，由於 d 是方塊的最後一邊，我們終於可以確定藍色部份是在方塊的內部。簡單來說其實就是如果一 polygon 在一多邊形體所有 polygon 的後面，那麼他一定在多邊形的內部〈當然多邊形體要為凸多邊形才行〉。

這個演算法的好處就是用在 3D 處理時相當直觀容易實做，但是無法用來處理凹多面體上。另外 CSG 演算法有個普遍的缺點，就是可能會在測試多邊形是否合法時做出多餘的分割，參考下圖：

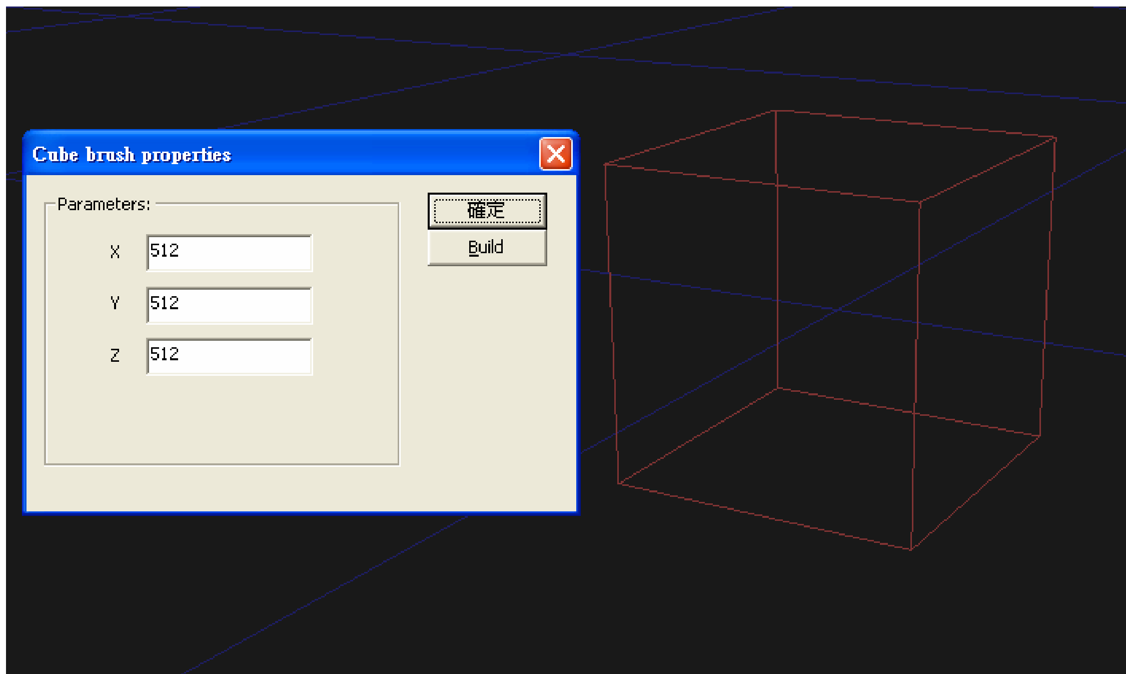


上圖左方代表一 polygon L 與方塊交錯的情形，假設我們以 b、c、d、a 的次序來檢驗 polygon L，我們會先將 L 以 b 來分割，而當最後與 a 做分割時檢驗結束。此時在方塊外面的 L 就變成兩個 polygon 了，如上圖中間的圖示。但若我們以 a、b、c、d 的順序時來測試 polygon L 時，結果總共只要分割一次，並且在方塊外面的 polygon L 只有一個！所以用 b、c、d、a 的次序來檢驗實際上產生了多餘的分割，一次分割代表產生了新的頂點，而在 3D 繪圖中一個頂點常需要經過一次或多次的座標轉換，產生多餘的頂點就代表必須付出多餘的計算，因此這是要想辦法避免的；而解決辦法就是在之後進行 polygon 的合併測試作業。

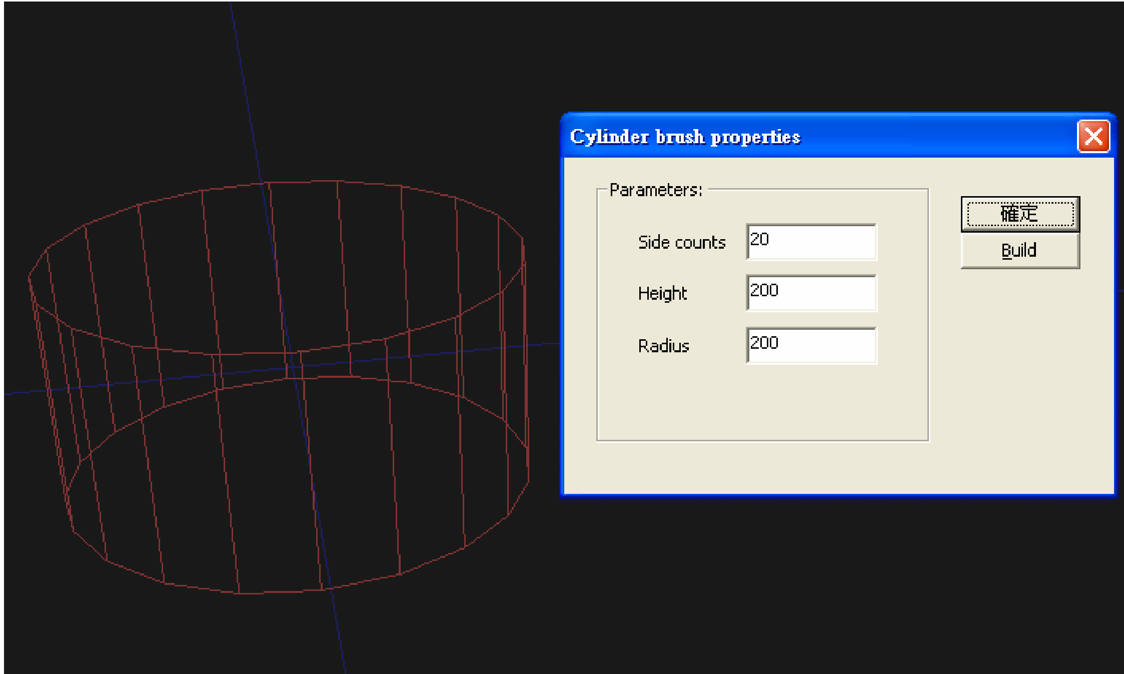
由於 CSG 在製作模型上非常容易學習，加上已經被業界使用多年不需其他驗證。因此我們的場景編輯器將使用 CSG 演算法，目前支援以下幾種參數化幾何原型：



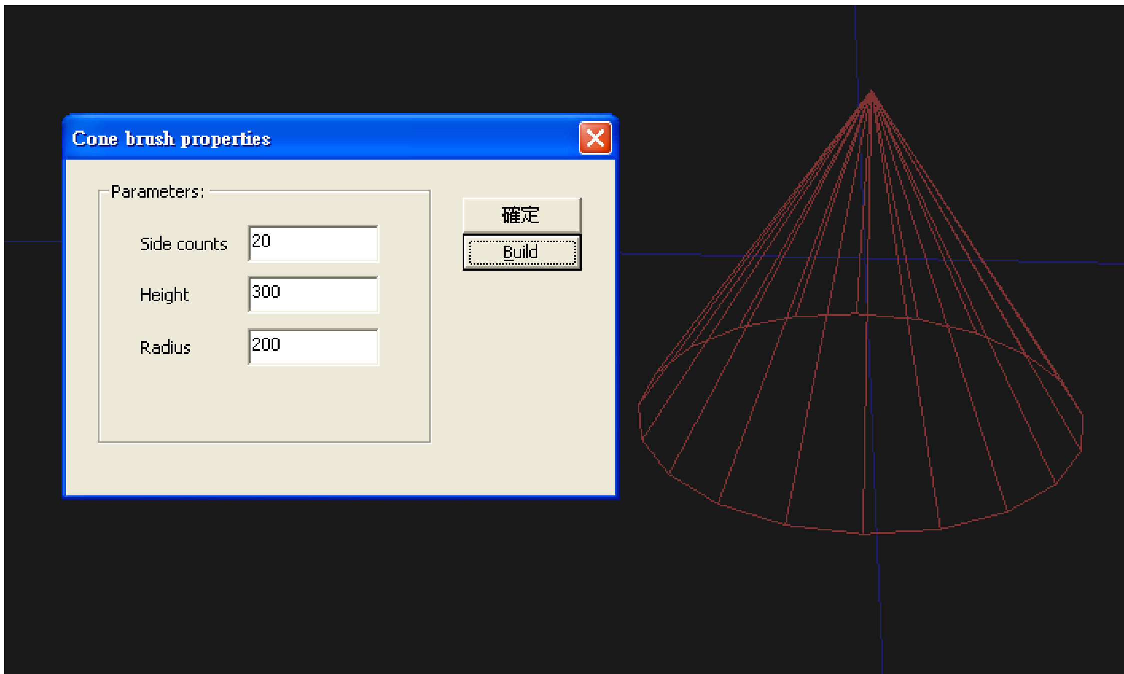
單一平面



立方體



圓柱



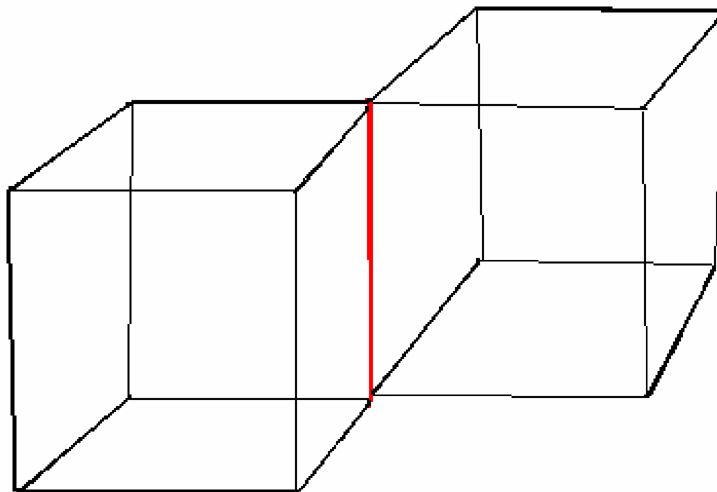
圓錐

2.2 封閉偵測與 Edge tracing 演算法

利用 CSG 建構出場景骨架後，在實做 portal rendering 之前，我們必須要知道一個 zone 內部的所有 polygon，但是真正的問題在於，如何知道這些 polygon 形成封閉的區域(zone)。

我們試著去想著各種方法，包括從 2D 方面開始著手。但最後發現在 3D 環境應用上都會碰到難以解決的問題，因此我們使用最直觀的演算法，稱它為"邊緣追蹤"(Edge tracing)。

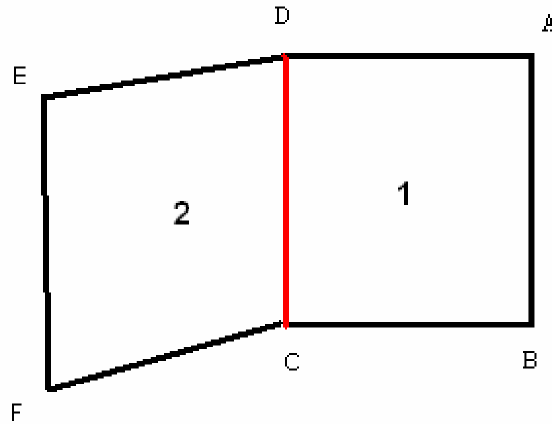
主要想法其實很簡單，首先從多邊形集合中任意選擇一個 polygon，再從此 polygon 的 edge 去找相連的 polygon，利用 polygon 的相連性進行追蹤。然而使用者自訂的場景將有可能出現多個 polygon 共邊的情形，例如下圖：



上圖由兩個方塊構成，紅色的邊界形成四個 polygon 共邊的情形。而單一 edge 共邊情形還有更複雜的情況。為了正確處理所有可能的情形我們依下面的規則作處理：

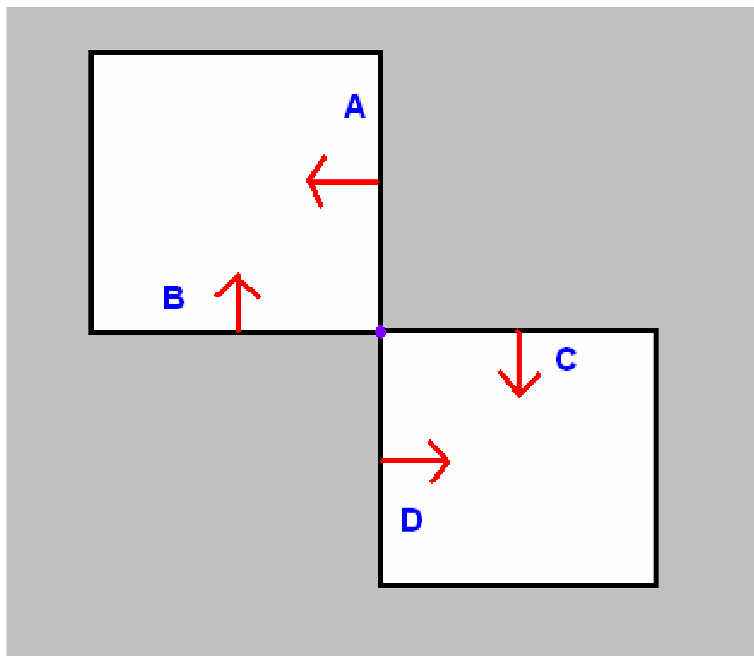
1. 由於我們的場景編輯器採用封閉的 brush 建構，因此所有的 polygon edge 至少會與另一 polygon 共邊，即不存在 open edge。

2. 正確的共邊規則為當一 polygon 的 edge 與另一 polygon 形成共邊時，此時兩 polygon 共用 edge 的頂點構成順序必然相反，如下圖



多邊形 1 與多邊形 2 共用紅色的 edge，polygon 1 的頂點順序為 ABCD，polygon 2 則為 DCFE，因此共用的紅色 edge 對多邊形 1 其 CD 向量與多邊形 2 的 DC 向量方向相反。

3. 當符合規則 2 的情形仍然可能會有多個，參考下圖：

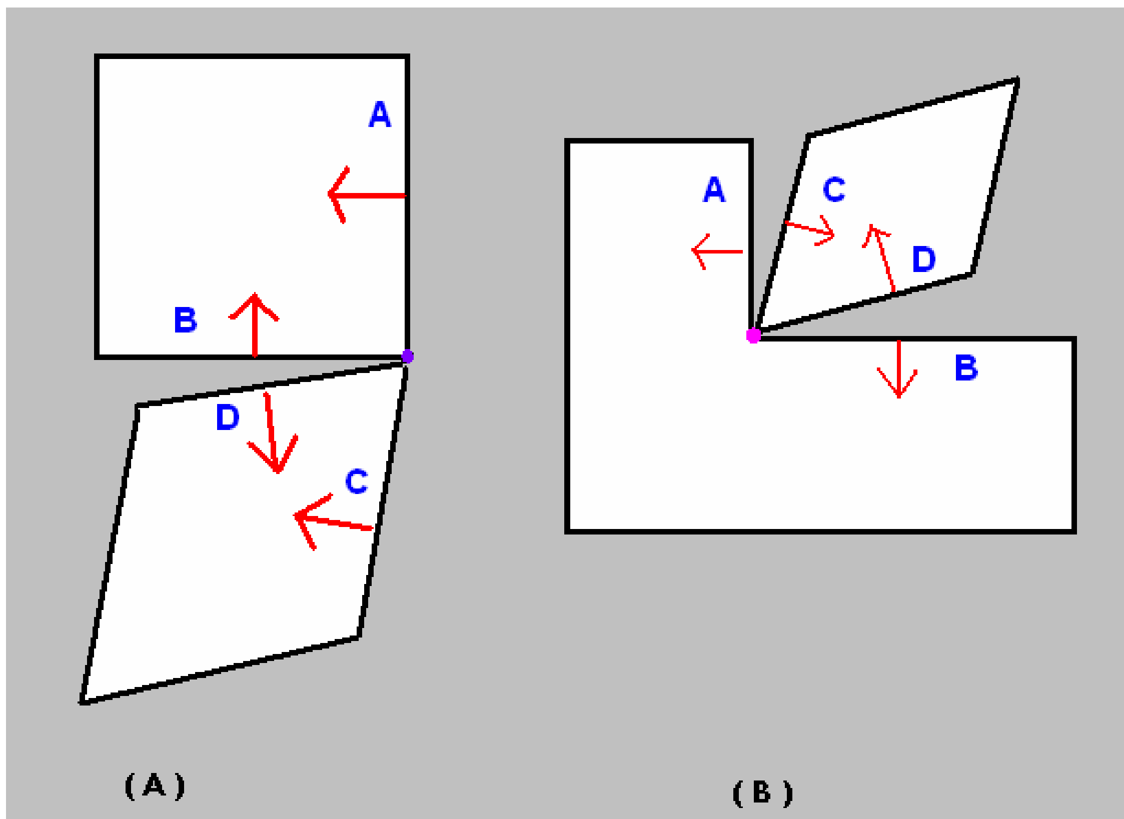


上圖由兩個中空的方塊構成並且視點由它們的正上方往下看去，紅色箭頭為各邊的頂點法向量。其中多邊形 A、B、C、D 共用紫邊；若從

多邊形 A 做 Edge 追蹤時依照規則 2 此時只有多邊形 D 不符合因此會被排除。然而此時必定要有辦法從多邊形 B 與多邊形 C 選擇出正確的相鄰多邊形。要選擇出正確的 polygon 我們由相對 polygon A 的位置可以區分成兩種情形，在 A polygon 的前方與後方。

要知道 Polygon B 與 Polygon C 相對 A 的位置相同容易，只要將 Polygon B 與 Polygon C 除了共用 Edge 的任一其他頂點帶入 Polygone A 的平面方程式即可得知。由於上圖只有 Polygon B 位於 Polygon A 的前方，因此正確相鄰 Polygon A 紫色 Edge 的 Polygon 為 B。

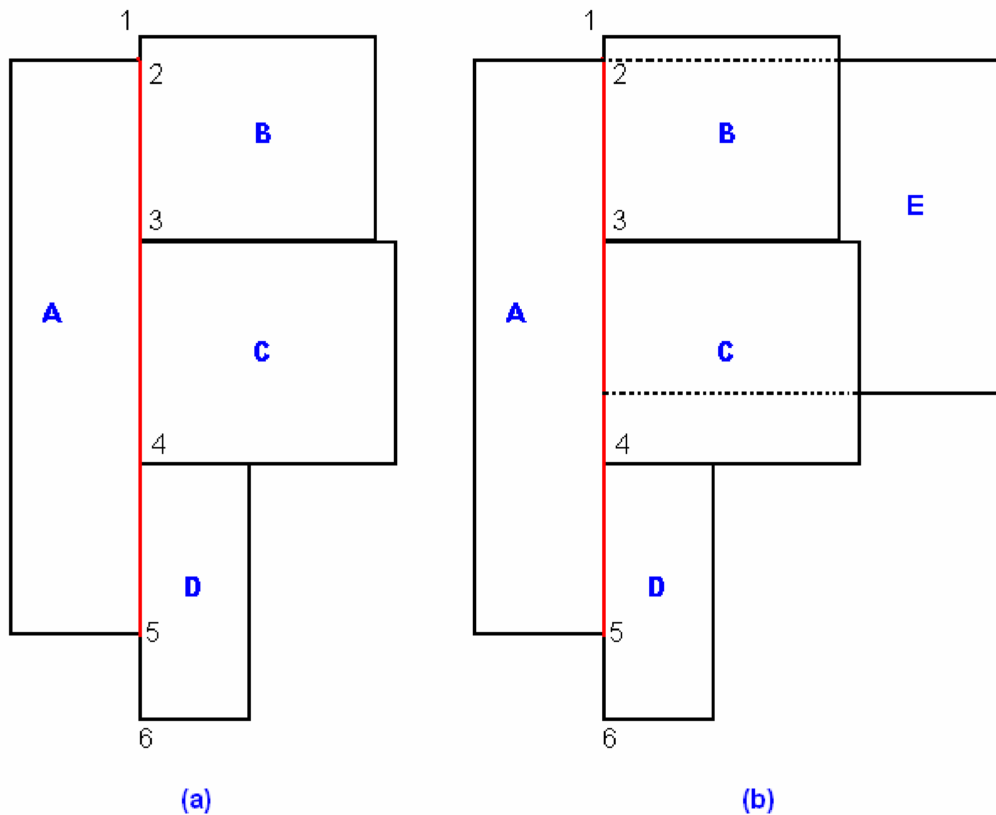
然而只由相對位置去選擇仍不能滿足所有情況，參考下圖(A)情形：



此時 Polygon B 與 Polygon C 都在 A 的前方，這種情形可以使用多邊形的法向量來輔助選擇。由於兩多邊形都在 Polygon A 的前方，因此我們可以使用法向量的夾角大小來判定何者為與 Polygon A 正確的共邊 Polygon。

由向量公式知 A dot B = |A| * |B| cos(theta)；因此當兩向量夾角小於 180 度時，若夾角越大則其 dot 值越小。我們將 normalized polygon B Normal 與 Polygon C Normal 分別與 polygon A normal 做內積。此時若兩法向量夾角越大，相鄰 Edge 的兩 polygon 形成的夾角越小。因此此時所要取的 polygon normal dot 值應該取較小者為正確的相鄰 Polygon。而若當相鄰的可能的邊集合都在 polygon A 後方時(上圖(B)情形)，此時情況正好相反。此時法向量夾角越小者其與 Polygon A 共 Edge 所形成的夾角越小，因此當可能的邊集合都在 polygon A 的後方時，我們要選擇其 normal 與 polygon A normal 內積值較大者為正確的相鄰 polygon。

4. 一 polygon 的單一 edge 可能與數個 polygon 形成共用 edge，如下圖：



polygon A 的紅色 edge 與 polygon B、C、D 共邊。若從 polygon A 做 edge tracing，則依 Polygon A 的頂點順序可將頂點 2 定為起始點，頂點 5 定為終點。而共邊的 Polygon B、C、D 起點和終點順序則相反，但為了方便之後的說明我們定為相同的順序以便說明。

首先由 polygon A 的頂點 2 去找符合以下條件的 polygon；由於 polygon A 的起始點所對應的另一 polygon 的起始點可能在 edge 的外部，因此此時所要找的 polygon 只要確定其 edge 的終點至少在頂點 2 的外部，符合條件則將 polygon 加入。

在上圖(b)中，我們最後找到的可能相鄰多邊形為 B 與 E，此時由之前的規則 3 選擇出正確的多邊形；假設 Polygon B 為正確的相鄰 polygon，我們將 Polygon B 的終端頂點 3 做為新的起始點開始找尋下一個相鄰 polygon，另外由於頂點 3 已位於 Polygon A 的 edge 內部，所以此時要找的 polygon edge 起始點必須和頂點 3 同位置。這時找到的 Polygon 只有 C 因此不需要額外的判斷，並由 polygon C 的終端頂點 4 繼續找尋相鄰的 Polygon。接下來找到的 polygon 將為 D，另外由於 Polygon D 的終端頂點已經超過 Polygon A 的 edge 終點 5，因此我們已經完成對 polygon A 紅色 edge 的追蹤作業。

根據以上情形，我們的 Edge trace 演算法總結大致如下：

1. 將所有多邊形區分為三種情形：已分類、未分類與待測集合，而一開始所有多邊形都存在未分類集合中。

2. 若未分類的 Polygon set 中不存在 polygon，則結束作業。否則從中取一 Polygon A 出來對其所有 Edge 做以下的追蹤：

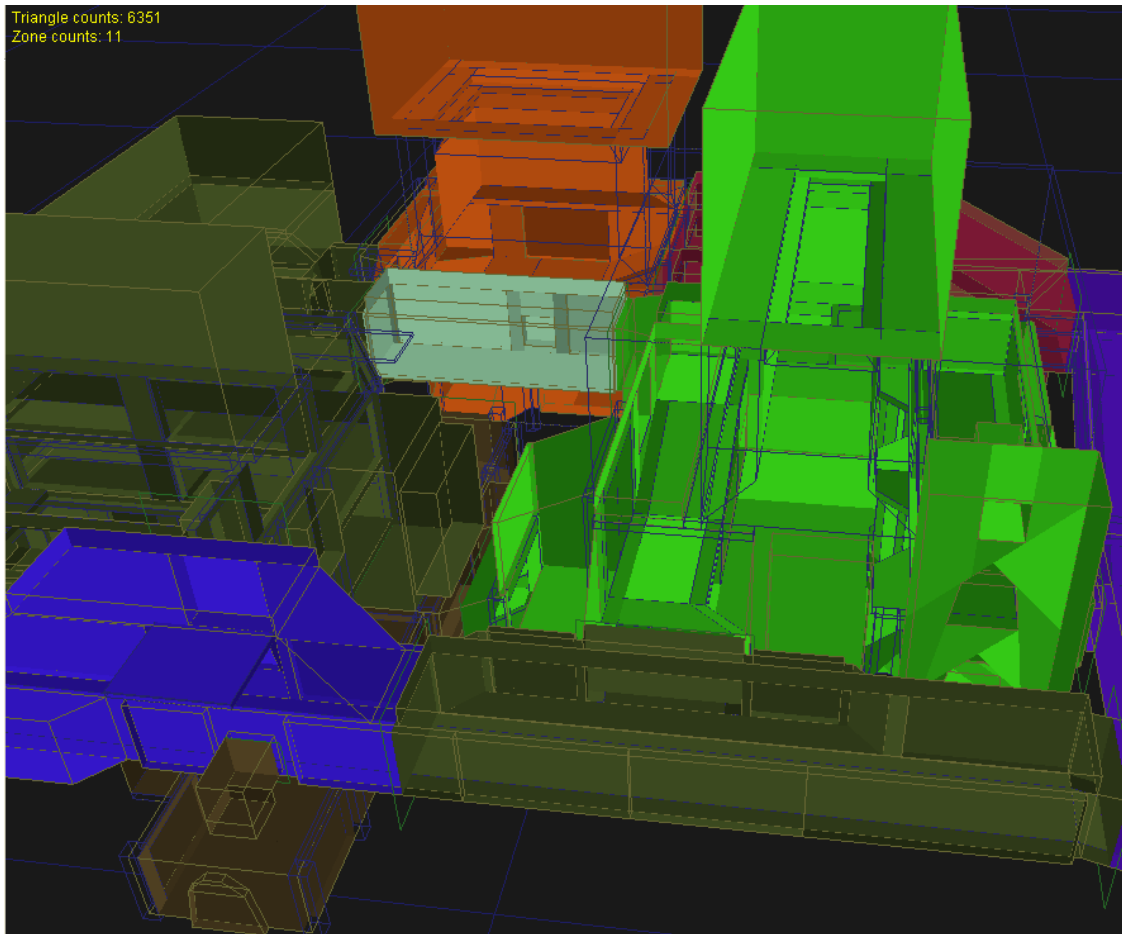
(1)若 edge 落在 portal polygon 上則略過此 edge 的搜尋作業，從 Polygon A 的下一個 edge 開始進行。

(2)對所有其他 Polygon 的每一 Edge 檢查是否有與 Polygon A 共 Edge;如果共 Edge 則檢查多邊形的 edge 向量是否與原 Polygon A 相反，如為真則將其加入 possible set 中。

(3)對 possible set 中依照上述狀況 4 的規則取得正確的相鄰 polygon 並將其置入待測集合中。

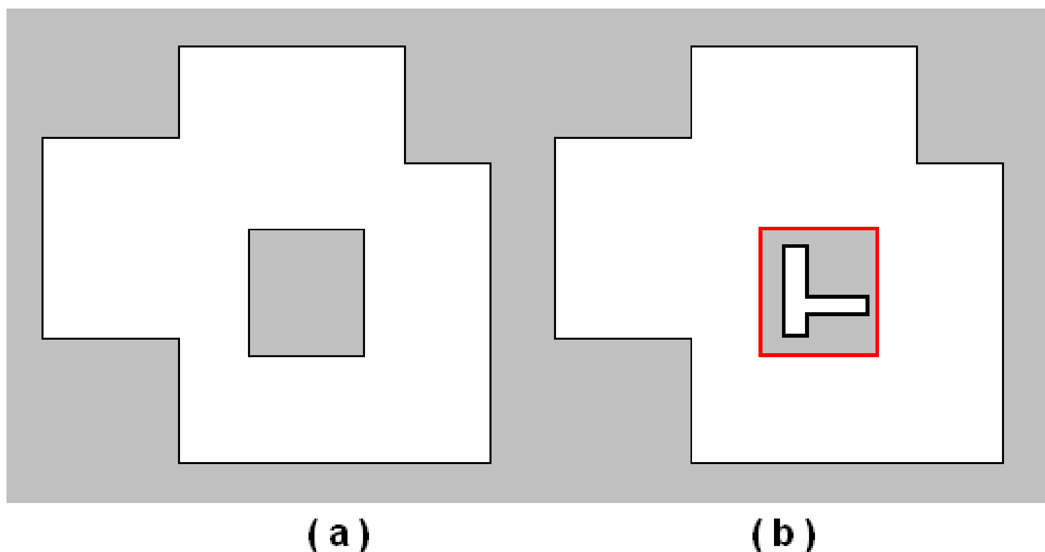
(4)之後將 polygon A 置入已分類集合中。

3. 若待測集合中存在 polygon，則取出一 Polygon 做為新的 Polygon A 再度進行步驟 2 的作業；若無 Polygon 在待測集合中則代表我們已經將一個 zone 所包含的 polygon 依照其相連性收集完成；此時已分類集合中的 polygon 即構成一封閉的 zone。儲存結果後將其清空回到步驟 2。



一複雜場景經過劃分後產生 11 個 zone

至此自動偵測空間劃分為 zone 的步驟已大致完成，然而事情並沒有這麼圓滿，經過測試後我們發現新的問題，參考下圖(a)：



為了簡化描述問題我們採用 2D 圖形來表示，但實際上每個邊都應當由 polygon 構成。由於在 zone 之中的方塊並沒有任何 polygon 和外邊的 polygon 相連；因此由 edge tracing 後自然會產生出兩個不同的 zone。

要解決這種情形，我們可以由內部 zone 的任一 polygon 由其 noraml 向外投射一射線，而射線所碰到的 polygon B 將有兩種情形；一種是該 polygon B 與原射出射線的 polygon 皆在相同的 zone 中，這並非我們要的結果因此跳過並從下一個 polygon 投射出新的射線。另外一種即 polygon B 所屬之 zone 為實際包圍方塊的 zone，該 zone 即為內部方塊其應所屬之 zone，所以將此兩 zone 進行合併的作業。

而對於這種情形的偵測，我們可以使用 zone 的 Bounding box，唯有一個 zone 的 Bounding Box 完全存在於另一 zone 的 Bounding Box 之中時，才需要另外進行投射 ray 的測試，眼看一切問題似乎都解決了，然而我們又發現了特別的情況，參考上圖(b)。

在中間固態空間中，紅色的方塊內又有一自成封閉區域的 zone，因此根本沒有進行 ray 測試的必要，要判斷此種情形依舊可以藉由 ray 測試。然而倘若中間的場景極為複雜，我們將不能找到一個明確的判斷條件來證明中間的區域形成一封閉的 Zone。在經過反覆思考觀察後，我們發現以下事實作為分類依據：

規則 1. 在由凸 polygon 所形成的封閉空間中，如果我們可以找到一個 polygon 使得其他所有的 polygon 都在其後方，那麼此 polygon 集合所形成的內部必為固態空間。

規則 2. 在由凸多邊行所形成的封閉空間中，如果我們可以找到一個 polygon 使得其他所有的 polygon 都在其前方，那麼此 polygon 集合所形成的外部必為固態空間。

在發現上面兩個規則後，解決上圖(b)的問題就容易了。

當我們發現某一 zone 的 aabb 被另一 zone 的 aabb 包圍時，我們立刻對此 zone 做測試，如果此 zone 的內部為非固態空間，那代表此 zone 為一單獨的 zone，因此不需要做 ray 測試。而若此 zone 的內部為固態空間，此時才需要做額外 ray 投射測試。

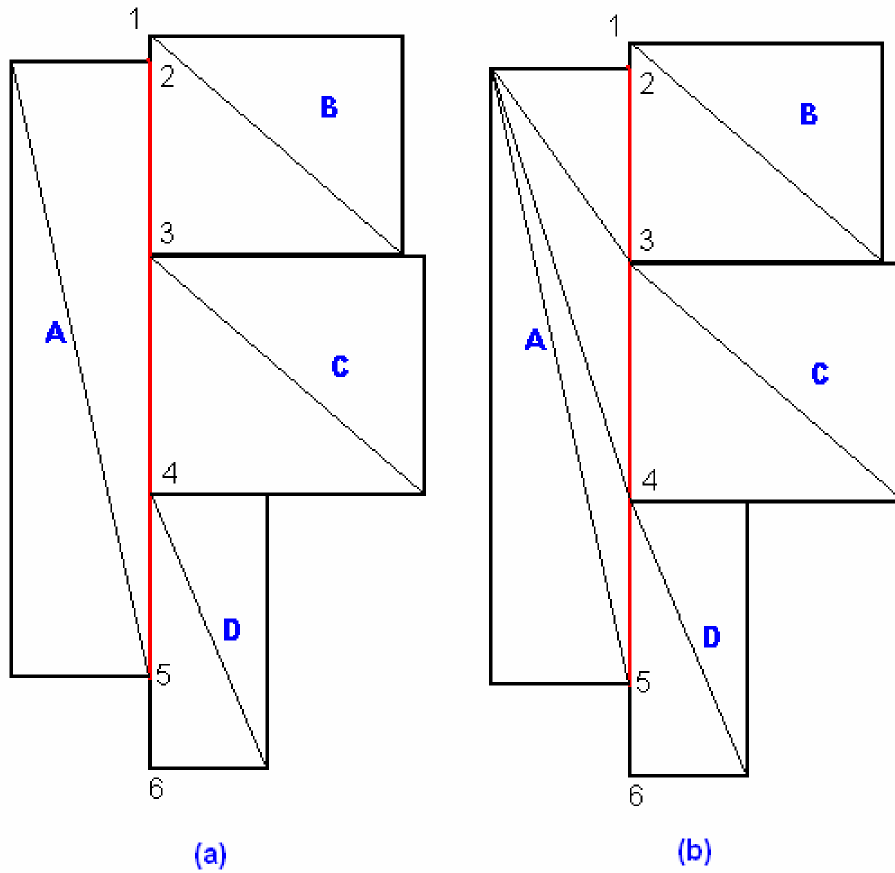
至此，我們的 edge tracing 演算法也總算完成了，經過各種複雜場景測試都為正確的結果，除了某些極端的情況會因浮點數精確度的問題產生錯誤。為此我們的場景編輯器特別使用 64bits 浮點數來代表資料以減少該情形發生；即使該錯誤發生時 scene editor 也會自行偵測並且回報使用者。

2.3 Edge tracnig 與 T-Junction 修正

參考下圖(a)，由於使用 CSG 產生的 polygon 可能會形成電腦圖學領域中俗稱 T-Junction 情況發生，因此在繪圖時可能會因為浮點精確度誤差的情況而出現間縫閃爍影響繪圖的品質。

要修正 T-Junction，我們發現在做 edge tracing 時將會是個很好的機會。由於 edge tracing 將對場景中的 BSP polygon 的每一 edge 進行追蹤，而當一 polygon A 的 edge 找到其他相鄰 polygon B 時，若 polygon B 的 edge 的終點位於 polygon A 的 edge 之中，我們即將此終點插入原先的 polygon A 的 edge 中。參考下圖(b);polygon A 與 polygon B、C、D 相鄰，修正後的結果將頂點 3、4 插入 polygon A 的 edge 中，

使得 polygon A 產生新的三角形；而新的三角形由於頂點與相鄰的 polygon 位置相同，因此可以避免 GPU 內部浮點運算誤差而產生的邊縫閃爍結果。



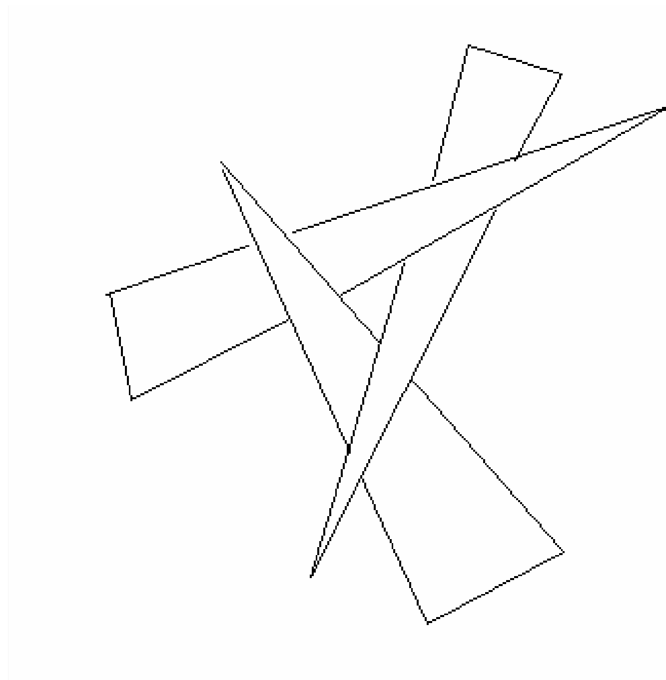
(a)圖中 polygon A 因紅色 edge 與相鄰頂點 3 與 4 形成 T-Junction，因此 edge 可能產生閃爍的隙縫；(b)為修正後的情況

2.4 BSP tree 理論簡介

BSP Tree 在 3D 圖學中是一個被廣泛應用的重要結構，BSP Tree 的出現最早是要解決 3D 電腦繪圖的"隱面消除問題"。要了解這個問題則要先從畫家演算法講起。一般畫家作畫時，通常會先將遠景畫出，接著在此底層的圖上畫上比較近的場景(物體)，這樣的方式便稱為畫家演算法。

畫家演算法的好處就是不需要找出物體的外框，舉例來說，如果你是先畫近的物體，那麼當你在畫遠的物體時，你要小心的在近物外框外面上色，小心避免蓋住較近的物體，這項動作由人類來做並不會太困難，但要由電腦來做卻不容易。

但是畫家演算法並不適合所有情形,例如下圖就是一個典型的例子:



三角形互相交錯，不論先畫哪個三角形都會將另一個較近的三角形遮住

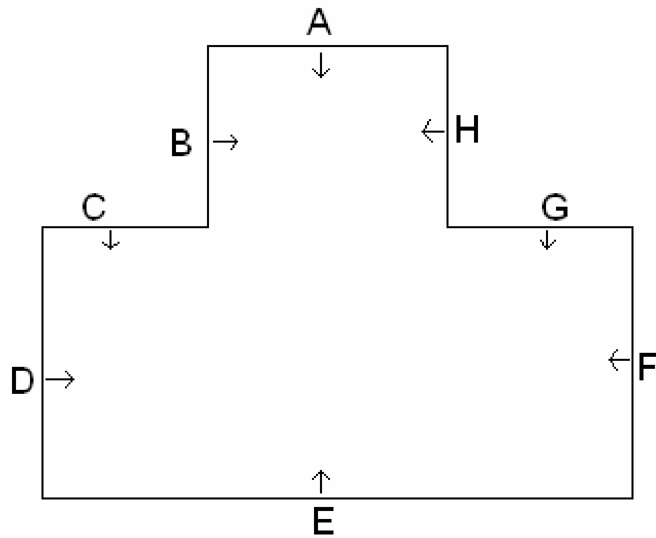
另外畫家演算法還有一個問題，就是在畫所有物體時必須先將場景的物件作排序，如此才能由遠畫到近，這是非常耗時的，尤其場景物體一多，整體效率便會被拖垮，所以不適合即時繪圖。要解決隱面消除問題其實有更好的辦法，就是使用 Z-Buffer。

Z-buffer 是一個和 Frame buffer 相似的記憶體緩衝區，例如你繪圖的解析度為 640 X 480，那麼便有相對應 640 X 480 個 pixel 的浮點數。但是他存的不是對應 pixel 的顏色值 而是對應 pixel 的深度值 (depth value 或稱 Z value)。所謂的深度值可以看成物體離觀察者視點的距離，離你越近深度值越小。

而 Z-Buffer 解決深度問題的方式很簡單，首先將 Z-Buffer 中所有值初始化為 1.0f，代表最遠平面的 Z 值(通常 Z 值用浮點數來表示，範圍由 0 到 1，常見的 hardware Z-buffer 有 16bits & 24 bits 兩種)。而在畫物體時將每個 pixel 的 Z 值與對應 Z-Buffer 的 Z 值作比較，如果比較小代表目前要畫物體的某一 Pixel 離觀察者比較近，所以畫上此 pixel 並更新對應 Z-Buffer 的 Z 值，如此以 pixel 為單位，便可以不照順序的畫上任何物體並且不管物體有沒有交錯。

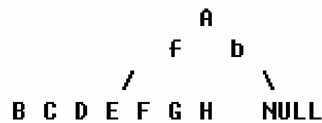
不過早期電腦的顯示晶片並沒有硬體實作 Z-Buffer 的功能，所以所有比較 Z 值的動作都由 CPU 來作，當然早期電腦的 CPU 也不像現在這麼強悍，因此 Z-Buffer 也不適合即時繪圖。

由於以上種種問題便有人想出 BSP Tree 這樣的結構，BSP Tree 演算法的精神實際上就是畫家演算法。但是他解決了畫家演算法的兩個主要問題—排序耗時、物件交錯。在解說他是如何解決問題之前我們必須先了解 BSP tree 結構，BSP tree 就是二元樹結構，不過他每一個 node 存的是一多邊形，此多邊形被要求為凸多邊形，原因等會解釋，以下說明建構 BSP Tree 的步驟。

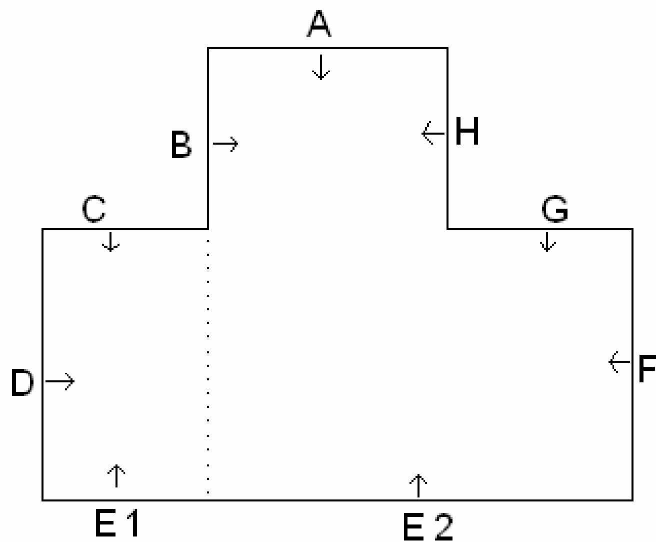


上圖為一張簡單場景的俯視圖，為了方便我們用 2D 的圖示來探討，3D 的原理其實是相同的，不過實作時會複雜一點。圖中箭頭代表此面的正面，以面的正法向量決定。

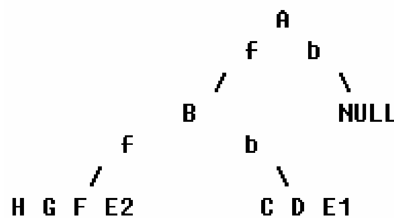
首先我們必須從所有的線條中選出 root 的分割面﹝選擇是要有方法的 不過在此先任意選擇﹞。我們先選擇 A 為 root 接下來以 A 面為分割面將剩下的面分成三類，分別為與此面共面、在此面前方、在此面的後方。這個選擇法導致所有剩下的面都在 A 的前方，這會造成二元樹不平衡，即使無法分出平衡的二元樹，BSP tree 還是非常有效率的結構。以下為目前的樹狀圖：



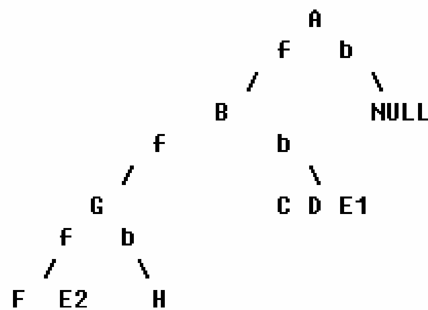
接著我們便由剩下的 BCDEFGH 面中選出分割面，假設我們接下來選擇的分割面為 B 那麼在 B 前面的面有 HGF，在後方有 CD 然而平面 E 卻穿過平面 B，所以此時我們要做分割的動作。將平面 E 以 B 分割成 E1、E2﹝參考下圖﹞，所以在 B 面前面便有 H、G、F、E2，後面有 C、D、E1。



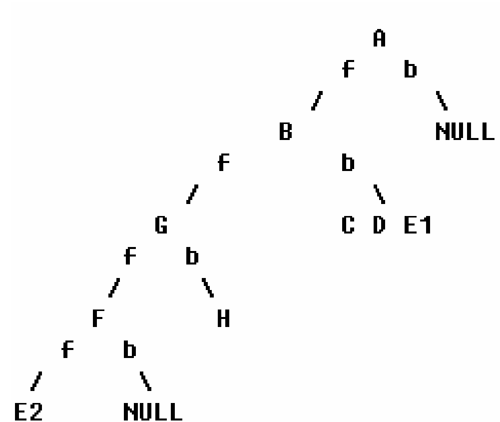
分割可以說是一個非常重要的動作，另外由於平面 E 被規定為凸多邊形，所以我們可以確保分割出來的面只有兩個。另外還有一個理由就是在處理真正的多邊形資料時，並沒有夠好的演算法可以分割凹多邊形，並且分割凹多邊形可能產生出不少面，這是我們在建構 BSP Tree 極須避免的。以下就是目前的樹狀圖：



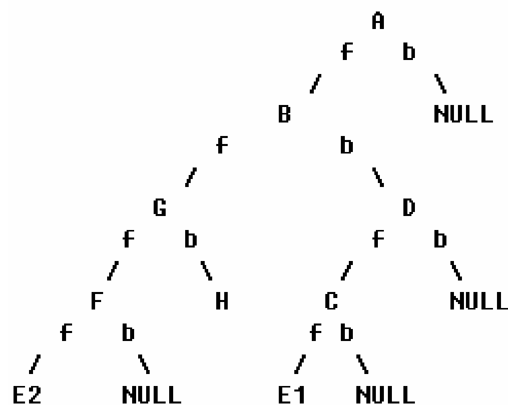
接下來我們再由 B 的前面集合中選出一個做為分割面，由肉眼可以看出選 G 會讓樹比較平衡，所以樹狀圖如下：



再由 F & E2 任意選出 F 做為分割面，樹狀圖如下：



以此類推，我們最終得到的 BSP tree 如下：



當我們建構完成 BSP tree 時，其實就解決了畫家演算法的兩個主要問題。我們已將場景的面排序完成，並且在做分割的動作時解決了物件交錯的問題。既然 BSP tree 的精神是畫家演算法，那麼在畫場景時自然就必須由遠至近將場景畫出。方法很簡單，在由樹的 ROOT 進入後，對每個 node 的分割面判斷觀察者的位置：

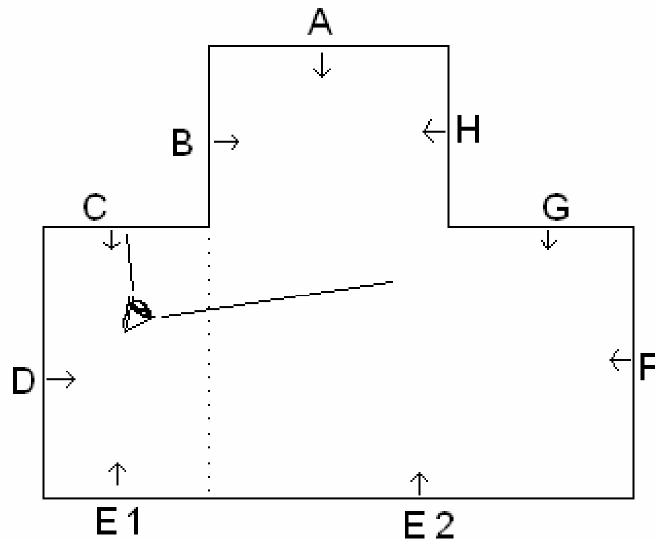
如果在面的後方，則由前、中、後遞回呼叫畫出。
如果在面的前方，則由後、中、前遞回呼叫畫出。

c code 演算法如下：

```
void Draw_BSP_Tree(BSP_tree *tree, point eye)
{
   // 將觀察者的位置以此節點的分割面做分類
   float result = tree->partition.Classify_Point(eye);

   // 觀察者在面的前方
   if(result > 0)
   {
      Draw_BSP_Tree(tree->back, eye);
      tree->polygons.Draw_Polygon_List();
      Draw_BSP_Tree(tree->front, eye);
   }
   // 觀察者在面的後方
   else if (result < 0)
   {
      Draw_BSP_Tree(tree->front, eye);
      tree->polygons.Draw_Polygon_List();
      Draw_BSP_Tree(tree->back, eye);
   }
   // 觀察者在面上，此時任意順序畫皆可
   else // result is 0
   {
      Draw_BSP_Tree(tree->front, eye);
      Draw_BSP_Tree(tree->back, eye);
   }
}
```

我們用上面的例子舉例，參考下圖：



觀察者在圖中以眼睛表示，當我們由樹的 root 進入後，我們先由平面 A 來決定觀察者的位置，顯然的觀察者在 A 的前方，所以我們要先畫 A 後面的圖形，而 A 後面並沒有任何節點，所以我們將 A 畫出，接下來進入 A 前方的節點，也就是平面 B。此時觀察者在平面 B 的後方，所以要先將前節點的圖形畫出。

所以我們可以很明顯的看出，當觀察者在面的前方時，面的後方當然會離觀察者比較遠所以要先畫，而當觀察者在面的後方時，此時剛好相反，面的前方的物體離觀察者比較遠，所以利用這個特性我們就可以使用畫家演算法來完成繪圖而不需使用 Z-Buffer，此外 BSP tree 是以觀察者位置為基礎的演算法，而不用管觀察者的視線。

不過由於現代顯示卡上都有 Hardware Z-Buffer，所以 BSP tree 的主要用途比較不像以前那樣用在解決隱面消除的問題上，但是 Hardware Z-Buffer 也不是萬能的，例如在處理場景透明物上就會發生問題。

想像一下眼前有一扇窗戶，上面帶有半透明的玻璃，窗戶外面是一座山，假設我們使用 Z-Buffer 的方式而不管前後順序，若我們先畫上較近的半透明玻璃，則當要畫山時，因為山比半透明玻璃遠，所以山便不會被畫上去，此時就會出現窗戶上面沒有山的情形。

當場景中有透明物體時，我們必須使用畫家演算法的方式才能畫出正確的結果。也因此使用 BSP tree 可以正確的畫出帶有透明物體的場景。

但是 BSP Tree 在現代的繪圖引擎中卻有不同的用法，由於 Hardware Z-Buffer 的普及，使得在畫一般不帶透明物的場景時不需要特別做排序便可畫出物體，但是搭配上 BSP Tree 若由場景的前方畫到後方，將可以提高繪圖效率，因為 Z-Buffer 演算法如下：

```
if(Current Pixel ZValue < Z buffer z value)
    Draw Current Pixel
    Update Z buffer
```

當我們由場景前方往後方畫時，如果有三角形重疊的情況，後方的三角形 Pixel 將不會被畫上去，因此可以提高繪圖效率，即使現今顯示卡的填充率已經非常高了，但是若場景物件太多，或是使用者使用高解析度來繪圖〔甚至使用 FSAA〕，隨時都有可能會讓效率下降，也因此這個技巧也常被應用在各繪圖引擎中〔如著名的 Quake3 engine〕。

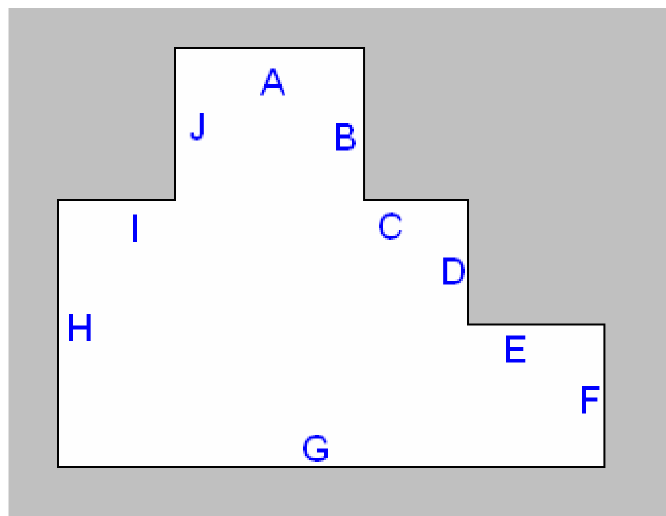
2.5 建立不分割場景 polygon 的 BSP tree

取得構成 zone 的 polygon 集合後，由於 zone 所包含的空間可能相當龐大，因此仍需要對其內部進行管理。而我們對 zone 的管理又分為室內與室外場景；若此 zone 為一室內場景，則我們將使用 BSP tree 演算法。傳統的 BSP tree 由於需要對場景 polygon 進行切割的動作，因此在一複雜的場景中將導致大量多餘的切割。而使用 BSP tree 進行 rendering 也不利於一般的 GPU 加速的原則；由於 GPU 適合處理相同繪製狀態的大筆資料，而 BSP polygon 卻常由於其 material、光源的不同無法一次搜尋出具有相同繪製狀態的大筆資料；最壞的情形就是每個 BSP polygon 都必須單獨的交付 GPU 繪製；另外由於 CPU 將工作指示交由 GPU 也需要工作時間，因此過多的指令傳送不但無法有效利用 GPU 加速，也會浪費過多的 CPU 時間。

因為以上幾點原因，以 BSP tree 管理的場景只適合建構大致的骨架以減少其 polygon 數量。而內部細節再使用其他單一 mesh 進行裝飾；可以想像一個室內空間的牆壁、牆柱、天花板、地板等面積較大的面由 BSP polygon 構成，內部需要精細的物件裝飾如家具等則由其他單一完整的 mesh 構成。

另外由於現在 Hardware Z Buffer 的普及，使得在建構 BSP tree 時已經沒有必要再分割 polygon，因此以下將提出說明建立不分割 polygon 的 BSP tree 演算法。

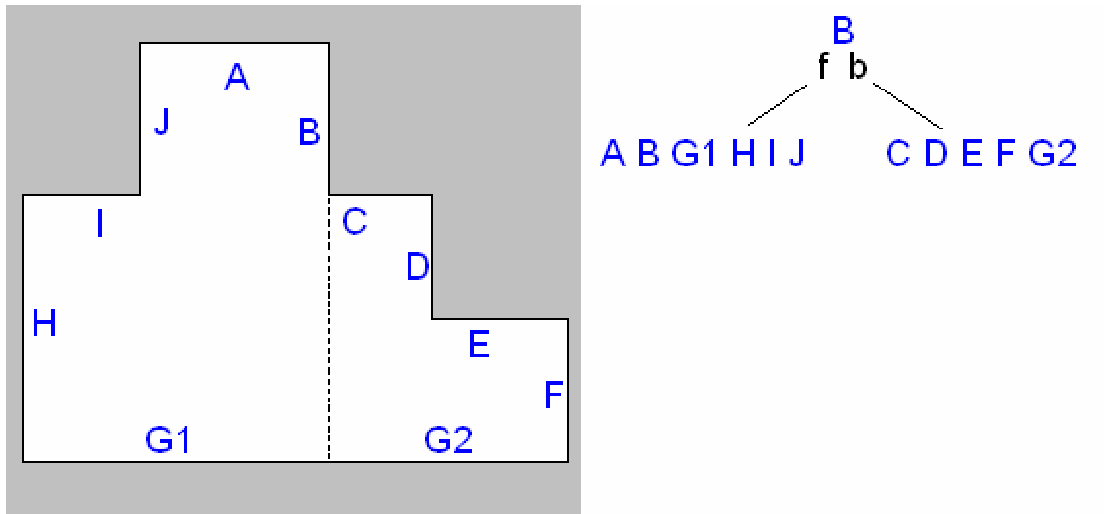
首先在 polygon 結構中加入一指標用以指向原未分割之 polygon，以下稱其為 parent polygon，並且將此指標初始化為 0。



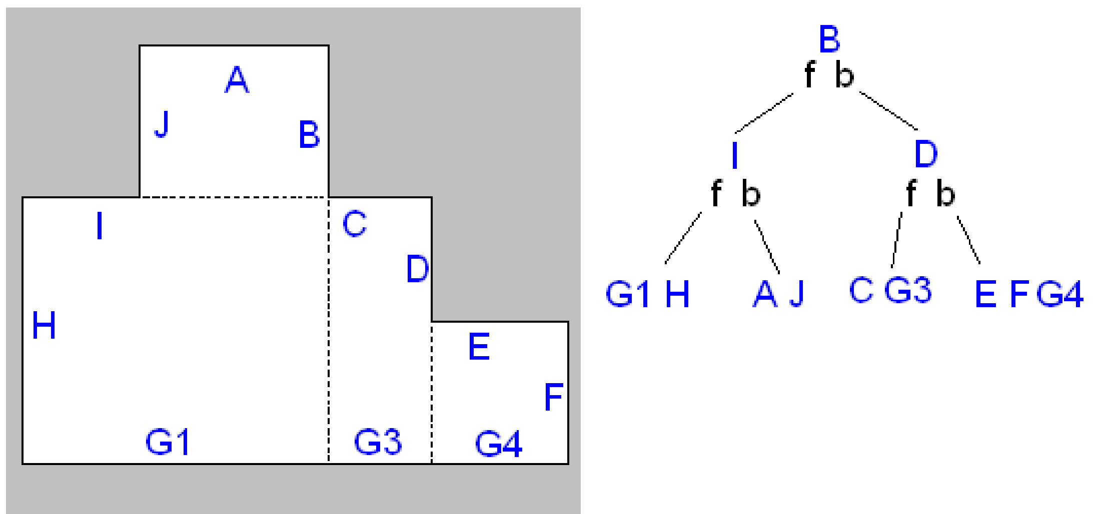
參考上圖，一空間中存在 polygon A、B、C、D、E、F、G、H、I、J 構成一封閉的區域，灰色區域為固態空間，每個 polygon 的 normal 皆指向非固態空間的區域並定此向為 polygon 的前方。

假設一開始我們選擇 polygon B 做為分割面，而 polygon G 因為橫跨過 polygon B 因此被分割為 G1、G2 兩個子 polygon，但我們將保留原 polygon G 並檢查其 parent polygon 指標是否為 0，如為 0 則代表此 polygon 是第一次被分割，所以將兩子 polygon 的 parent polygon 指標指向 polygon G，如果已經有值則代表此 polygon 已經非原 polygon，將此值拷貝到子 polygon 的 parent polygon 指標中；在此的情況因為

polygon G 為第一次被分割，所以將兩子 polygon 的 parent polygon 指標指向 polygon G。結果如下圖：

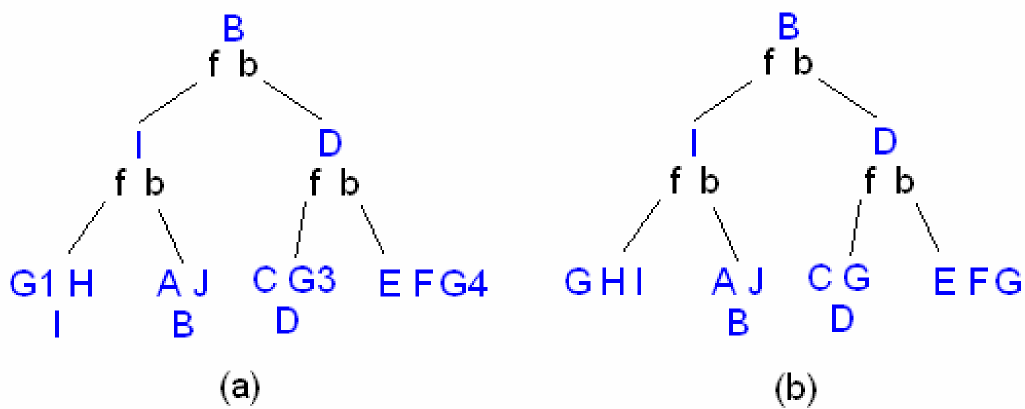


之後我們在 polygon B 的前節點中選擇 I 做為新的分割面，而後節點選擇 D 做為新的分割面；此時將 G2 分割為 G3 與 G4，結果如下。



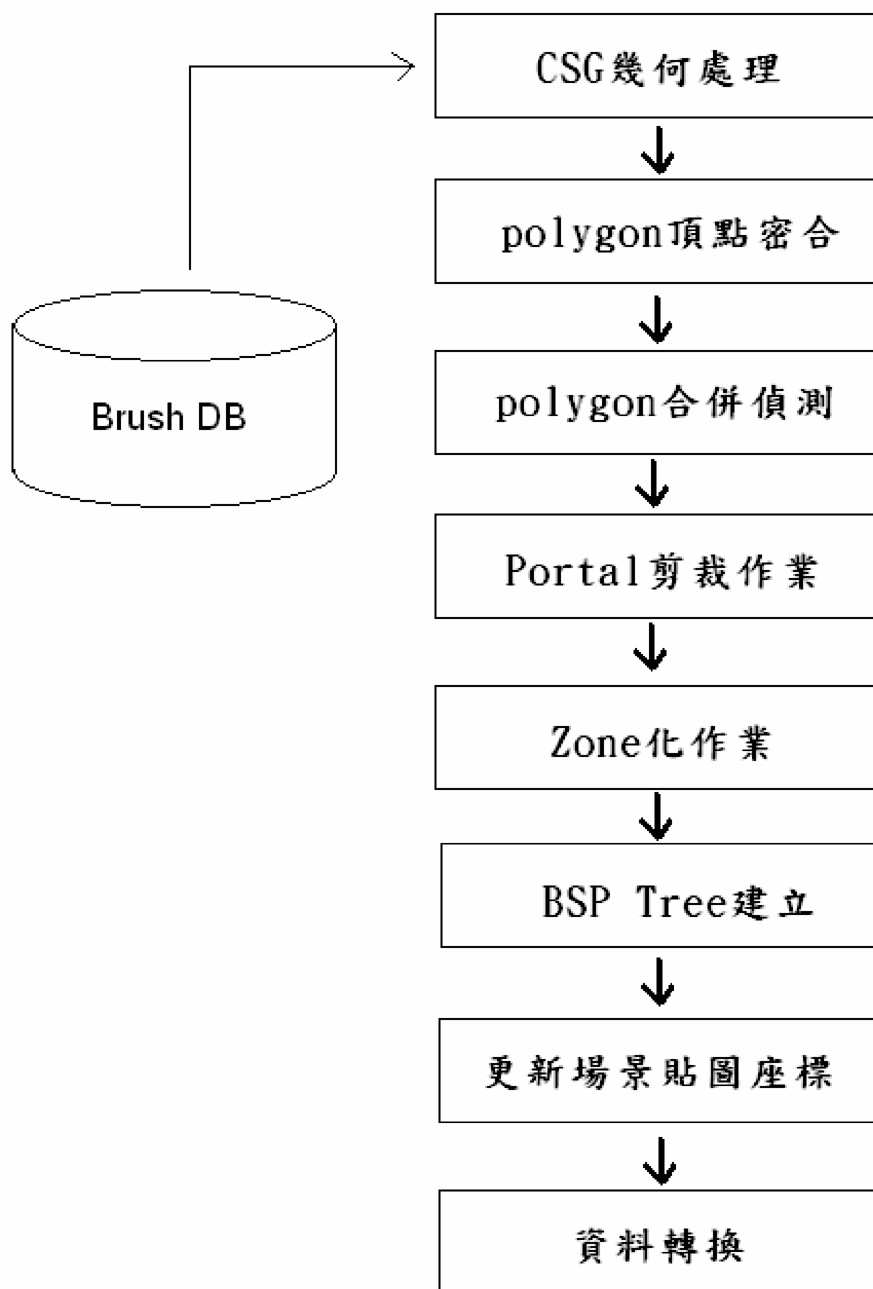
由於 G2 的 parent polygon 指標不為 0，因此將此值〈指向原 polygon G〉拷貝到 G3、G4 的 parent polygon 指標中，之後將 polygon G2 刪除。

由於我們使用 leaf-based BSP tree，因此 tree 中的 node 不存放該分割面的 polygon 只保存 polygon 的平面方程式；而改將分割面的 polygon 放入其前節點中，結果如下圖(a)：



最後的結果得到四個 leaf nodes，也就是空間最後被劃分成四個子 convex cell；利用一次遞回並以 post order 的順序求出樹中每個節點的 aabb。最後進行還原 polygon 的步驟，檢查每 polygon 中的 parent polygon 指標是否為 0，若不為 0 則用其取代目前的 polygon，結果如上圖(b)所示。至此我們不對場景的 polygon 做分割的 BSP tree 已建立完畢，並且每個 node 都能保有正確的 aabb 資訊；唯一產生的問題是由於 polygon G 存在於 3 個 leaf nodes 中，若三個 leaf nodes 皆在觀察者視野中將導致 polygon G 被重複繪製 2 次。解決辦法非常容易，只要在 polygon 中設一 flag 用以表示是否繪製過即可。

2.6 系統流程



在由使用者建完模型後，系統經過上圖的流程產生出最後的場景模型，以下依序說明各步驟：

CSG 幾何處理：

一開始整個場景由使用者利用編輯器提供的幾何原型建立出骨架，之後經過 CSG 計算後產生由凸 polygon 構成的場景模型。另外為了加速此步驟的計算，我們對場景中所有的 brush 建立 relative table，紀錄 brush 之間的關係；其可能的關係共有四種：分開、相交、包含、被包含；之後不少作業也能從此 table 獲得加速的作用。

Polygon 頂點密合：

由於 csg 產生 polygon 可能會因為浮點誤差的關係導致某些頂點雖然看起來在 3D 空間中有相同的位置，但實際上的數值卻不相同，因這些微誤差也將可能造成邊縫閃爍的問題。因此接著將場景中所有 polygon 的頂點與鄰近頂點做測試，將所有相近頂點給於同樣的值，也有利於後來進行其他步驟。

polygon 合併偵測：

本文之前曾提到由於 CSG 運算時會產生多餘的切割因而產生更多的 polygon，因此在此進行偵測合併 polygon 作業；對於所有場景中 polygon 而言，如果當兩共平面 polygon 擁有相同的 material 與共同的 edge，並且合併後仍為 convex polygon 時，則將兩 polygon 合併為一 polygon。

Portal 剪裁作業：

此步驟檢查場景中的 polygon 是否和 portal brush 交錯，如果有交錯則將 polygon 以 portal polygon 進行切割。

Zone 化作業：

將場景中的 polygon 利用自行設計的 edge tracing 分析出 zone。

BSP tree 建立：

利用本文之前提到的不分割 polygon 的方式建立 BSP tree。

更新場景貼圖座標：

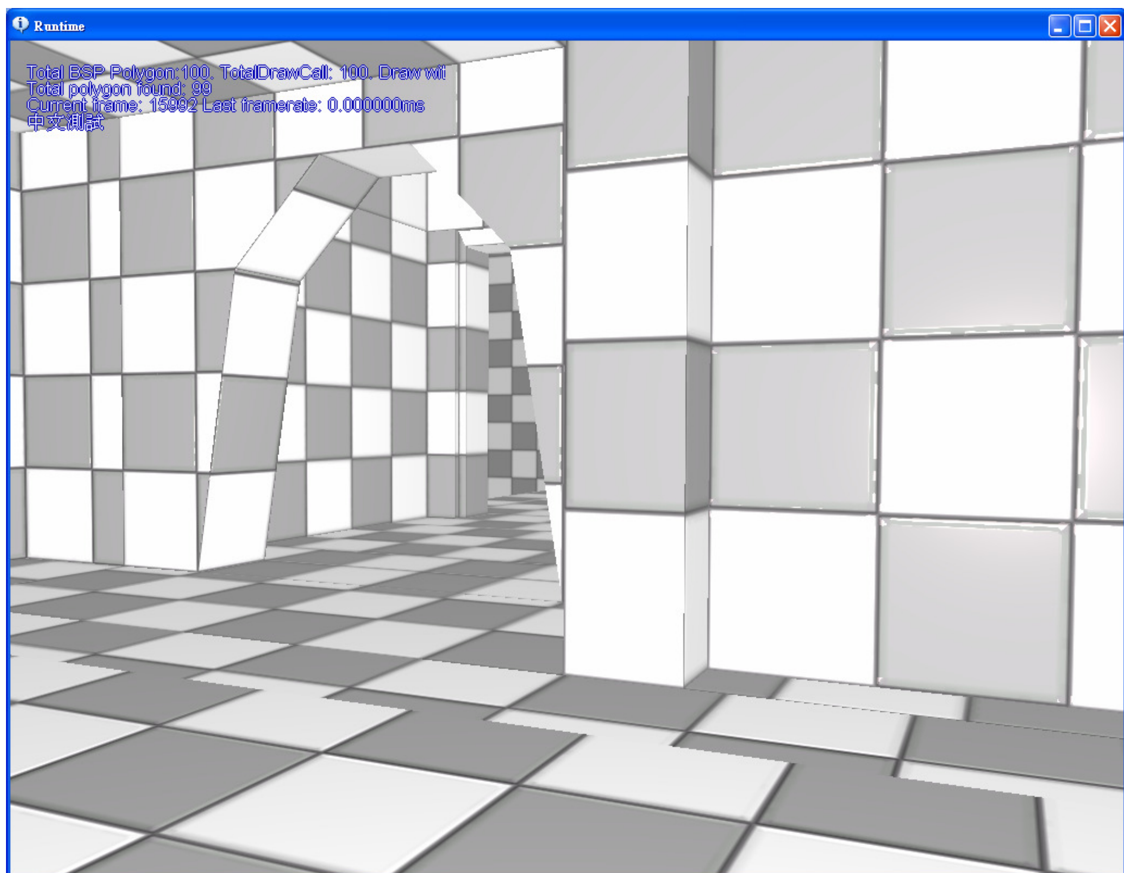
此步驟依據 polygon 的 material 產生貼圖座標，之後再根據 material 的貼圖座標計算每個頂點必要的 tangent、binormal 資料，壓縮後以 byte4 格式儲存。

資料轉換：

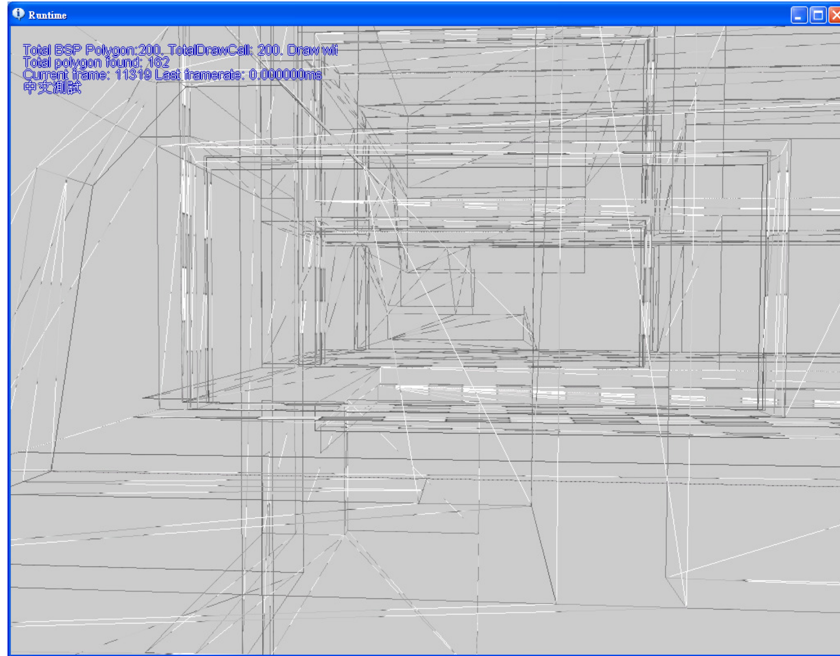
由於編輯器為了方便編輯作業所以常包含實際 runtime 繪製時不需要的資料，並且編輯器內部常使用 list 來儲存資料；但 runtime 為求效率將改用 array 來儲存；此步驟主要產生 runtime 版的資料結構並儲存在檔案中。

2.7 結果比較

以下比較將可明顯看出經由 portal + BSP cell culling 可以明顯減少 GPU 繪製的資料量。

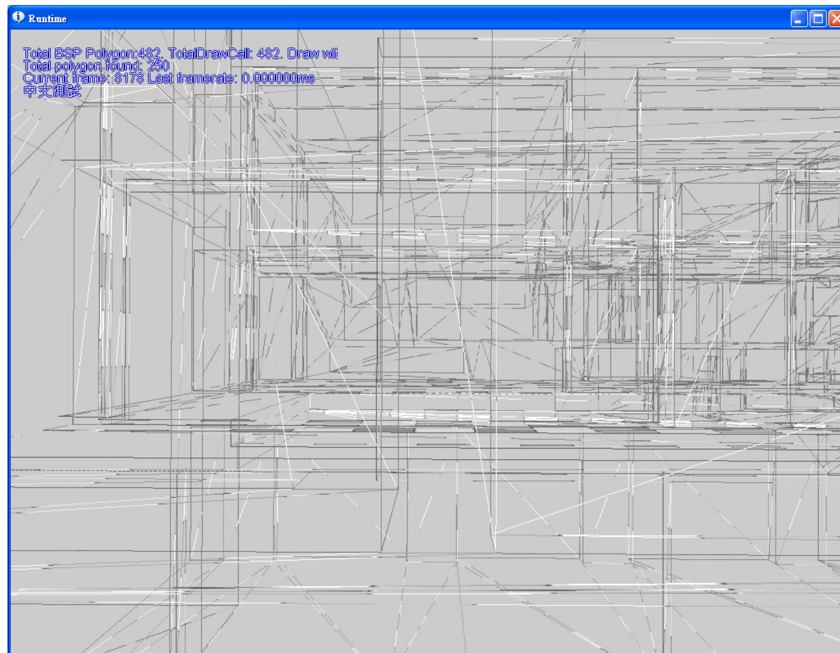


場景原貌圖



開啟 portal culling

上圖為一室內場景的截圖，使用者從 zone 經由 portal 向外部看去，此時畫面中總共存在 200 個 polygon，而關閉之後結果如下，總共有 508 個 polygon 在場景中。



關閉 portal culling

第三章 視覺效果

3.1 Lighting model

要提高整個影像的品質，改善光影效果已經成為重要的課題。例如目前常見的動畫電影，由於不需要考慮即時互動性因此大部分都使用需要大量計算的 Ray-tracing 來做計算光影效果；而使用 Ray-tracing 在處理 Global Illumination 與光線的折射反射後將可產生逼真的效果。

在即時繪圖應用上目前著名的例子為德國學者 Daniel Pohl 將 ray tracing 帶入 quake 3、4 引擎中。然而值得注意的是，由於其為了達到 real time 而捨棄了需要大量運算的效果，如 indirect illumination、soft shadowing，使得 ray tracing 的在畫面擬真度上佔不到明顯的優勢；唯一的優勢就是對於光線折射與反射的計算上有非常正確的結果。

目前即時繪圖中由於 GPU 仍採用傳統 rasterization 的方式設計；雖然也已經有研究成功實做出利用目前 GPU 架構來進行 ray tracing，然而由於硬體終究不是為了實做 ray tracing，因此使用的限制不少；加上各種以目前 GPU 架構提升影像品質的方法不斷被提出，使得短時間內很難有跳脫 rasterization 架構的 GPU 設計出現。因此目前對於場景的光照效果採用逐漸普及的 Phong reflection model 即時去做運算，其一般表示法如下：

$$I_{pixel} = I_a + \sum_{lights} (I_d + I_s) \times att$$

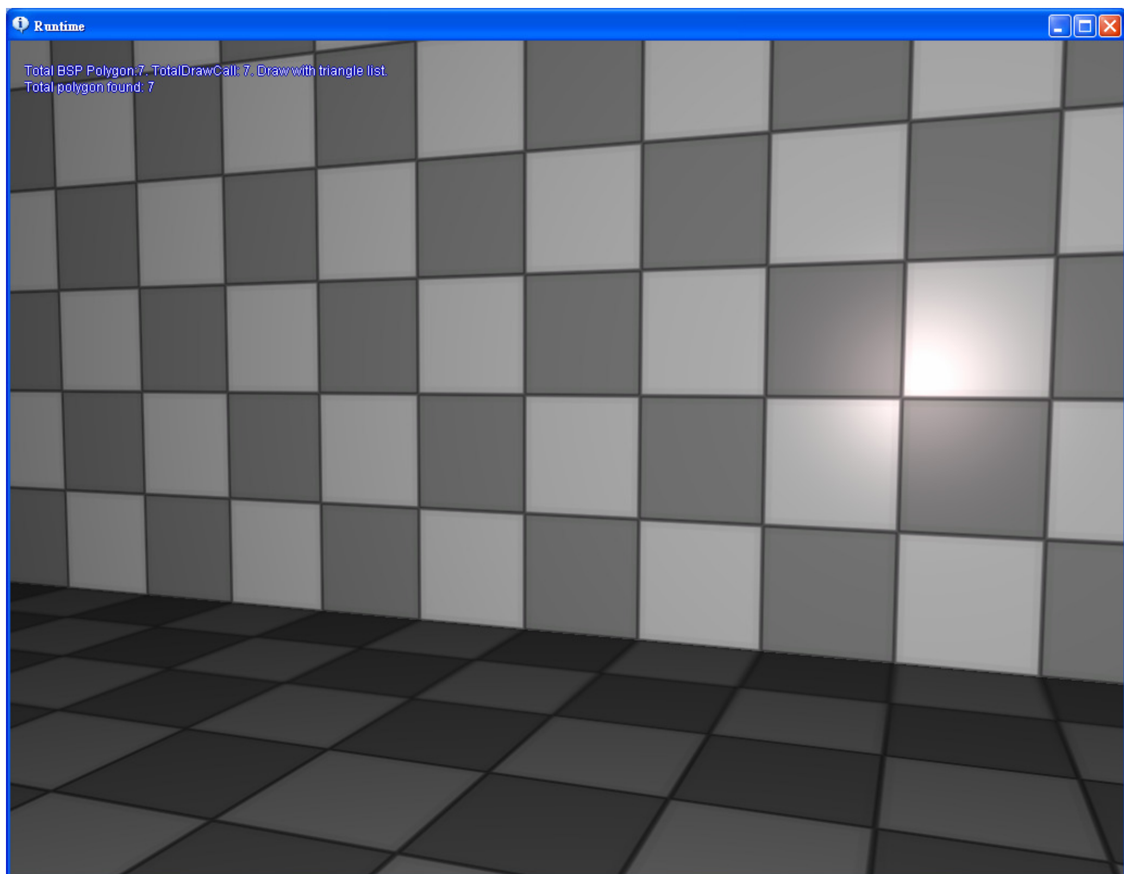
Ia 為場景中的環境光，要擬真計算此值通常需要大量運算如 radiosity，因此需要事先計算儲存在貼圖中；而另一種以 pixel 周遭物體來計算遮蔽率的方法也在 1998 年時提出[ZIK98]，並經過簡化應用在時運算上;即由知名繪圖引擎開發商 Crytek 在 SIGGRAPH 2007 course 中展示 Screen Space Ambient Occlusion (SSAO)方法並應用在知名繪圖引擎 CryEngine 2 中；此方法雖然並沒有很正確的理論基礎，但卻能

產生很好的效果，因此將來也預計加入系統內，目前則是以常數作計算。Id 與 Is 分別為散射與高反光，計算公式如下：

$$I_{diffuse} = N \bullet L \times L_{diffuse} \times M_{diffuse}$$

$$I_{specular} = (R \bullet V)^{s} \times L_{specular} \times M_{specular}$$

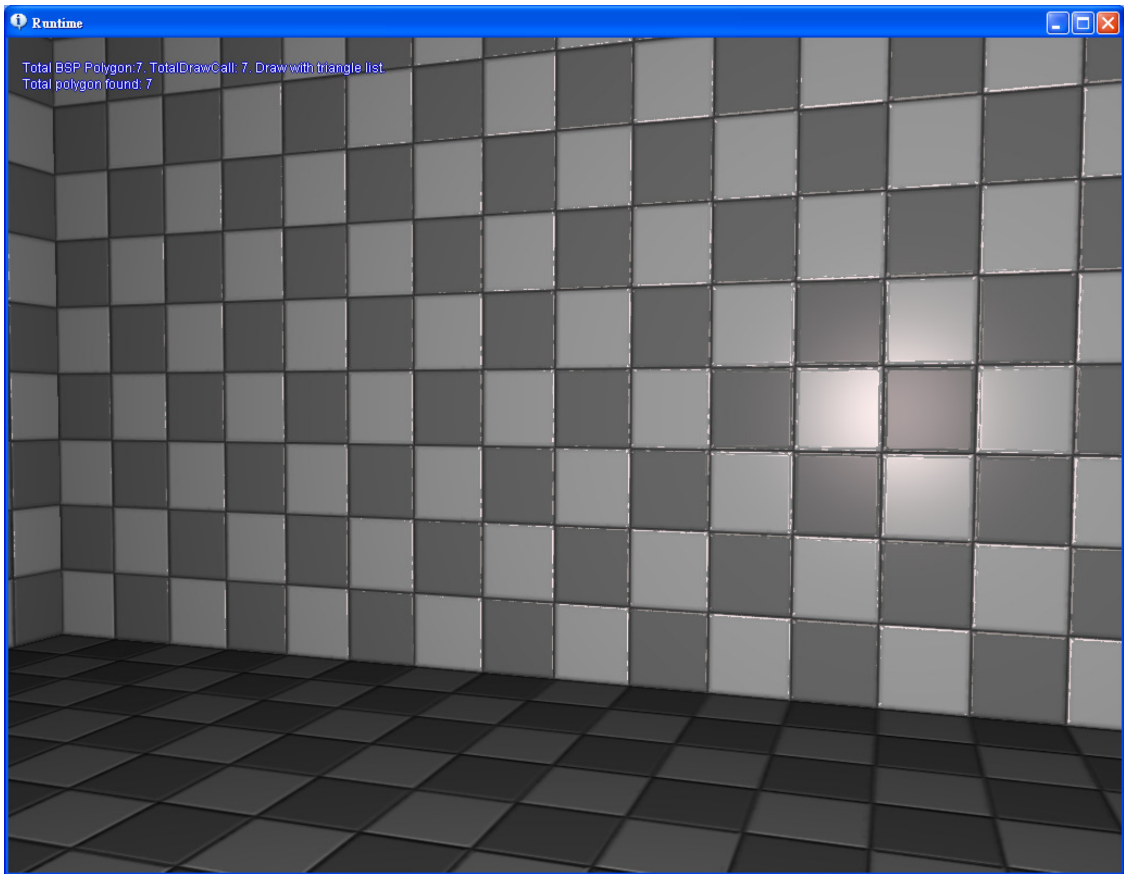
Att 為光衰弱函數，為求方便目前僅與距離平方成反比，下圖為結果。



Real time per-pixel lighting

3.2 Normal mapping

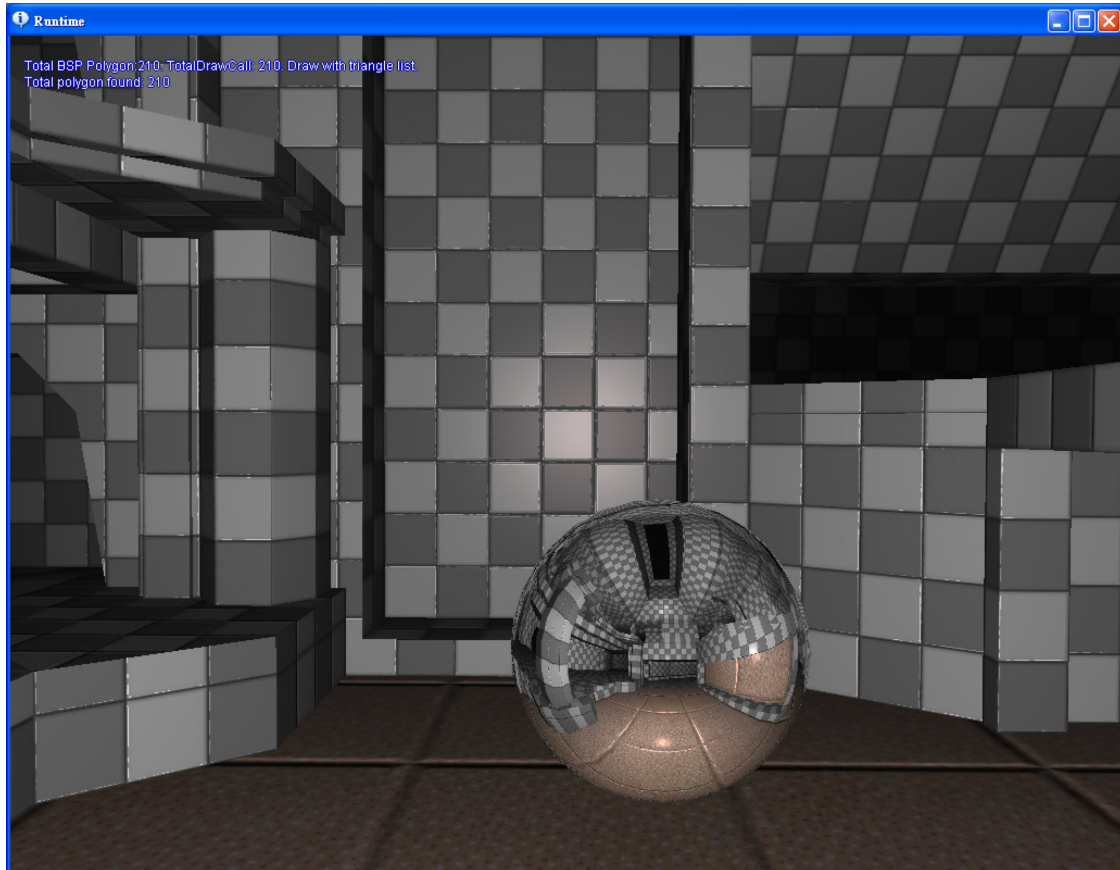
Normal mapping(法向量貼圖)，利用貼圖改變 polygon 表面法向量影響光照計算產生物體表面凹凸感，參考下面的結果與上面的圖示作比較：



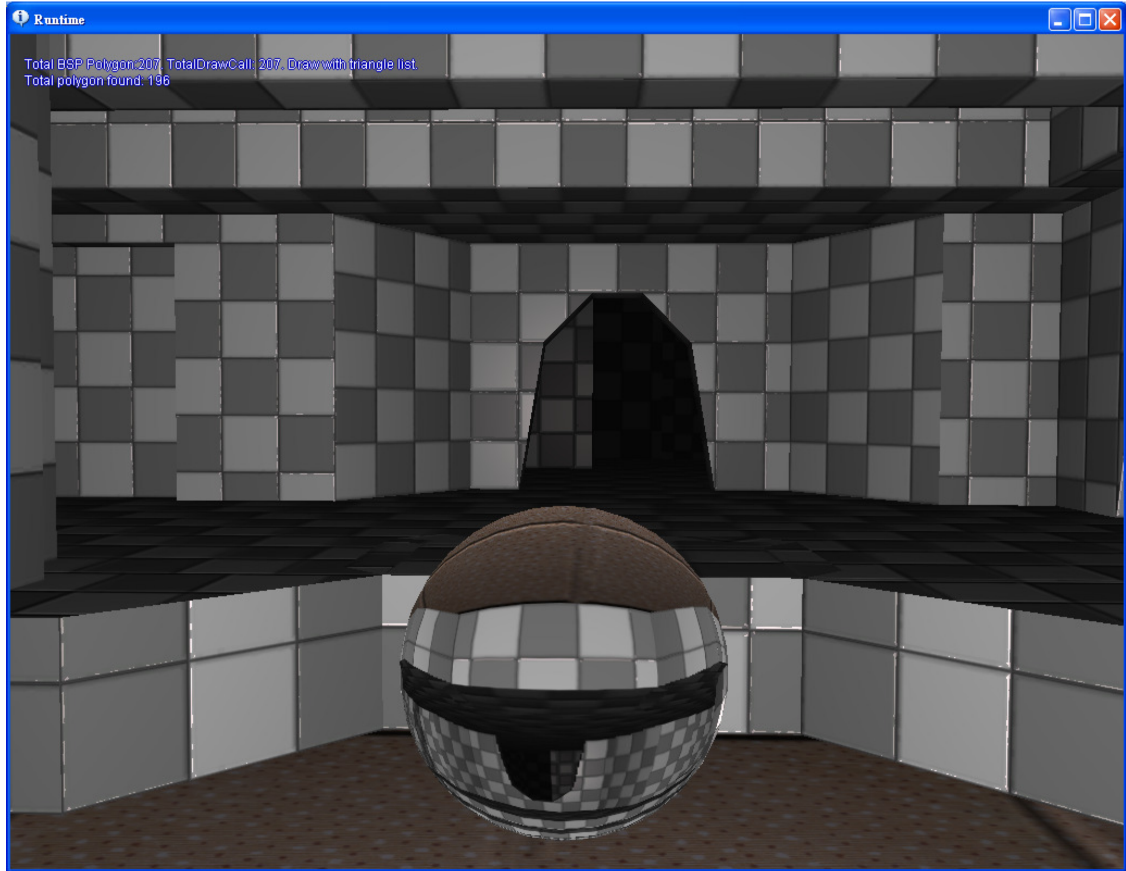
開啟法向量貼圖

3.3 環境貼圖效果

藉助於高效率的場景管理系統，因此能即時從物體的周圍繪製場景六次以取得 cube environment map。之後用其模擬光線折射和反射效果，結果如下：



模擬光線反射效果



模擬光線折射效果

3.4 Shadow effect

系統使用 GPU 加速的 z-fail shadow volumes 來產生陰影，結果如下圖：

Multi-light source and shadow volumes

第四章 結論

我們實作開發出一套具有完整功能的 3D 繪圖引擎，並且套用在一個具有相當複雜度的 3D 場景上。另外開發出一套場景管理工具程式，配合 3D 繪圖引擎，方便開發者使用。並且改善 CSG-oriented BSP Trees 的架構，減少 polygon 的數量，降低場景的複雜度。

由於 3D 繪圖發展快速，使得各種視覺效果大量被提出，然而畢竟個人力量有限，因此很多演算法還待完成。例如戶外場景 terrain rendering，廣大的場景地貌的生成；在陰影處理方面由於 shadow volumes 先天的缺點難以克服：如產生出 hard edge 的 shadow、並且計算量會隨 mesh 的複雜度而改變、加上如果 polygon 的 material 有透空貼圖的話將無法反應該情況。因此 shadow volumes 的使用越來越少，取而代之的是普遍使用的 shadow map 方法，因此整合該演算法顯得格外重要。另外要創造虛擬真實的互動場景，骨架動畫已成為重要的應用；不久前 skeleton animation 由於使用 CPU 來對 vertex 計算座標轉換工作，因此單一 mesh 的頂點數和場景中總 mesh 量受到一定的限制。然而現今 GPU 已經可以利用 vertex shader 單元來取代 CPU 計算，因此使用的 mesh 不但可以增加其細節與複雜度，對於數量限制也放寬不少，因此加入 GPU 加速的 skeleton animation 也成為 future work 之一。

參考文獻

BSP tree FAQ
http://www.opengl.org/resources/code/geometry/bspfaq/

Gordon, D., and Chen, S., Front-to-back display of BSP trees, IEEE Computer Graphics and Applications, 11(5), 79--85, sep 1991.

Sung, K., and Shirley, P., Ray Tracing with the BSP Tree, Graphics Gems III, 271--274, 1992.

CSG 相關
Naylor, B., Interactive solid geometry via partitioning trees, Proceedings of Graphics Interface '92, 11--18, may 1992.

[ZIK98] ZHUKOV, S., IONES, A., AND KRONIN, G. 1998. An ambient light illumination model. In Rendering Techniques ' 98 (Proceedings of the Eurographics Workshop on Rendering), pp. 45 – 55.

MSDN DirectX
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/anch_directx.asp

nVIDIA developer Home
http://developer.nvidia.com/page/home