Digital System Design (Fall 2022)

Final Report of Term Project

Group Number (組別) : 23

Group Member 1 (組員 1) : Student ID 109550059 Name 黃彥傑

Contribution (貢獻度) 100 %

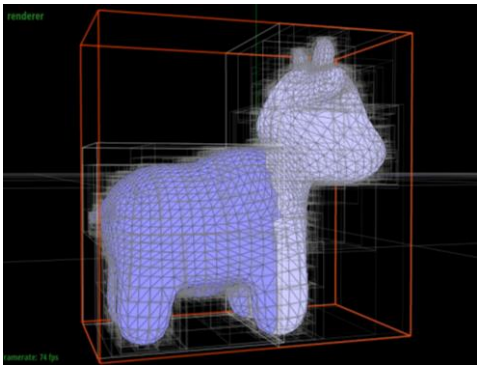
Title (標題) : 包圍體階層走訪及相交測試電路設計

A. Problem Description (問題敘述):

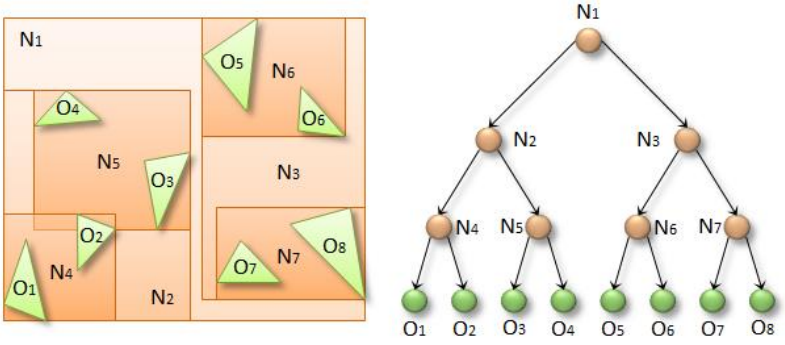
在計算機圖學領域中，光線追蹤（ray tracing [1]）是一個是能產生擬真圖像的渲染技術，其在電腦遊戲及動畫產業上有舉足輕重的地位。然而，光線追蹤的計算複雜度相當龐大，在通用處理器（包括 CPU、GPGPU）上無法充分發揮其運算效能，導致光線追蹤往往無法在通用處理器上達到實用的目的（例如在電腦遊戲中保持高幀率或應用於高複雜度的動畫電影渲染）。

也因為如此，顯示卡廠商包括 NVIDIA、AMD，在近年的產品都增加了專用電路加速光線追蹤的運算[2][3]。在光線追蹤渲染技術中，其中一個相對耗時且複雜的運算即是包圍體階層（bounding volume hierarchy，簡稱 BVH [4][5]）的走訪（traversal）及相交測試（intersection test），其目的為計算光線射出至擊中 3D 模型的相交資訊，作為後續渲染之用。

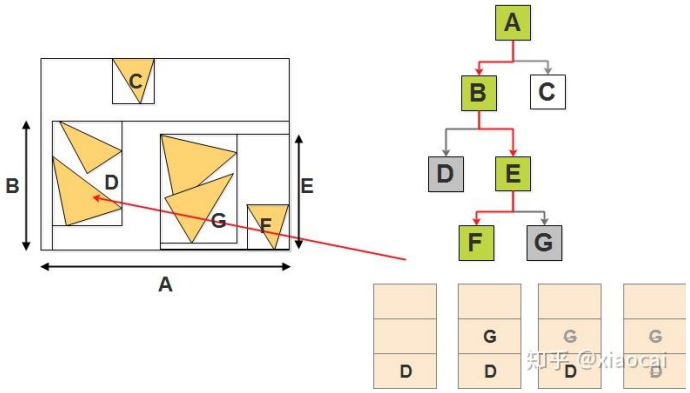
3D 模型通常是以大量的三角形所組成，而包圍體階層的主要核心概念即是利用許多包圍體（通常為立方體）將這些三角形包住，且包圍體之間是階層式的（包圍體中含有包圍體），如下圖所示：



由於包圍體階層是階層式的，我們通常會使用樹的形式儲存，如下圖：



如果我們要計算光線射出至擊中 3D 模型的相交資訊，一個有效率的方法即是透過包圍體階層，階層式地走訪上述所提及的樹結構。如下圖所示：



其中左圖為欲渲染的場景，右圖為包圍體階層的樹狀表示。左圖的紅色箭頭為光線的軌跡，黑色長方形為包圍體；右圖的紅色箭頭為走訪的路徑。

在計算光線射出至擊中 3D 模型的相交資訊時，除了走訪之外，我們也會需要對三角形執行相交測試。相交測試指的是給定光線資訊（光線原點、方向等）及三角形資訊（三角形座標、法向量等），計算光線是否擊中三角形並求取相交資訊(例如相交座標)。通常我們會使用高效的演算法，例如 Möller–Trumbore ray-triangle intersection algorithm，我在這次 project 中也是採用這個演算法。

在開始進行包圍體階層走訪及相交測試之前，包圍體階層的樹狀結構需要先被建造（construction）出來。然而，建立包圍體階層涉及許多複雜的運算（例如排序），不容易進行硬體加速，因此包圍體階層建造不在這次 project 的進行範圍。我會使用開源的包圍體階層函式庫[6]預先建造包圍體階層並儲存成檔案，以 Verilog 讀取並進行包圍體階層走訪及相交測試。

B. Flowchart or Procedure (流程圖或運作程序) :

為了說明演算過程，我將包圍體階層走訪演算法使用虛擬碼描述出來，如下圖所示：

Algorithm 1 BVH Traversal

Input: $ray, bounds, triangle\_indices, left\_child\_indices, triangles$
Output: $intersected, t, u, v, n_x, n_y, n_z$

1: $intersected \leftarrow$ false
2: $t \leftarrow \infty$
3: $stack \leftarrow$ empty stack
4: $node\_index_l \leftarrow 1$
5: **while** true **do**
6: $\quad node\_index_r \leftarrow node\_index_l + 1$
7: $\quad$ **for** $i \in \{l, r\}$ **do**
8: $\quad\quad$ **for** $j \in \{x, y, z\}$ **do**
9: $\quad\quad\quad tt_{i,j,min} \leftarrow (bounds[node\_index_i]_{j,min} - ray.origin_j) \div ray.dir_j$
10: $\quad\quad\quad tt_{i,j,max} \leftarrow (bounds[node\_index_i]_{j,max} - ray.origin_j) \div ray.dir_j$
11: $\quad\quad\quad entry_{i,j} \leftarrow tt_{i,j,max}$ if $ray.dir_j < 0$ else $tt_{i,j,min}$
12: $\quad\quad\quad exit_{i,j} \leftarrow tt_{i,j,min}$ if $ray.dir_j < 0$ else $tt_{i,j,max}$
13: $\quad\quad$ **end for**
14: $\quad\quad entry_i \leftarrow \max(0, entry_{i,x}, entry_{i,y}, entry_{i,z})$
15: $\quad\quad exit_i \leftarrow \min(t, exit_{i,x}, exit_{i,y}, exit_{i,z})$
16: $\quad\quad node\_intersected_i \leftarrow$ false
17: $\quad\quad$ **if** $entry_i < exit_i$ **then**
18: $\quad\quad\quad$ **if** $triangle\_indices[node\_index_i] \neq \emptyset$ **then**
19: $\quad\quad\quad\quad$ **for** $triangle\_index$ in $triangle\_indices[node\_index_i]$ **do**
20: $\quad\quad\quad\quad\quad triangle \leftarrow triangles[triangle\_index]$
21: $\quad\quad\quad\quad\quad$ **if** triangle_intersect(ray, t,$triangle$) **then**
22: $\quad\quad\quad\quad\quad\quad intersected \leftarrow$ true
23: $\quad\quad\quad\quad\quad\quad$ update t, u, v, n_x, n_y, n_z
24: $\quad\quad\quad\quad\quad$ **end if**
25: $\quad\quad\quad\quad$ **end for**
26: $\quad\quad\quad$ **else**
27: $\quad\quad\quad\quad node\_intersected_i \leftarrow$ true
28: $\quad\quad\quad$ **end if**
29: $\quad\quad$ **end if**
30: $\quad$ **end for**
31:
32: $\quad$ **if** $node\_intersected_l$ **then**
33: $\quad\quad$ **if** $node\_intersected_r$ **then**
34: $\quad\quad\quad$ **if** $entry_l < entry_r$ **then**
35: $\quad\quad\quad\quad$ push $left\_child\_indices[node\_index_r]$ into $stack$
36: $\quad\quad\quad\quad node\_index_l \leftarrow left\_child\_indices[node\_index_l]$
37: $\quad\quad\quad$ **else**
38: $\quad\quad\quad\quad$ push $left\_child\_indices[node\_index_l]$ into $stack$
39: $\quad\quad\quad\quad node\_index_l \leftarrow left\_child\_indices[node\_index_r]$
40: $\quad\quad\quad$ **end if**
41: $\quad\quad$ **else**
42: $\quad\quad\quad node\_index_l \leftarrow left\_child\_indices[node\_index_l]$
43: $\quad\quad$ **end if**
44: $\quad$ **else if** $node\_intersected_r$ **then**
45: $\quad\quad node\_index_l \leftarrow left\_child\_indices[node\_index_r]$
46: $\quad$ **else**
47: $\quad\quad$ **if** $stack$ is empty **then**
48: $\quad\quad\quad$ break
49: $\quad\quad$ **end if**
50: $\quad\quad node\_index_l \leftarrow$ top element of $stack$
51: $\quad\quad$ pop one element from $stack$
52: $\quad$ **end if**
53: **end while**

為了使狀態易於設計，在底下的狀態圖設計中，我將第 7 行及第 8 行的 for 迴圈展開。在經過這樣的轉換之後，整個演算法基本上只剩下大量的 if-else 分支，因此設計狀態會較為容易。

在虛擬碼的第 21 行有 triangle_intersect() 這個 function，下圖為 triangle_intersect() 的虛擬碼：

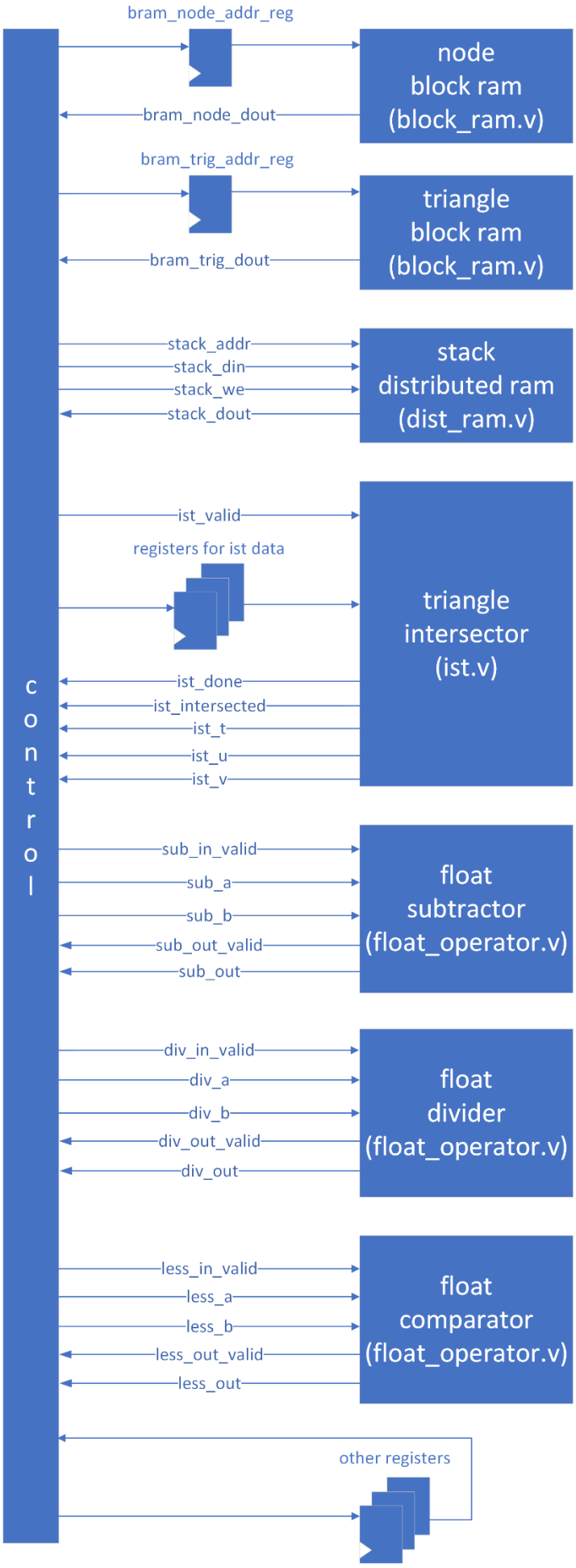
Algorithm 2 Triangle Intersection

Input: $ray, tmax, triangle$
Output: $intersected, t, u, v, n_x, n_y, n_z$

1: $intersected \leftarrow false$
2: $n_x \leftarrow triangle.n_x$
3: $n_y \leftarrow triangle.n_y$
4: $n_z \leftarrow triangle.n_z$
5: $p_x \leftarrow triangle.p_x$
6: $p_y \leftarrow triangle.p_y$
7: $p_z \leftarrow triangle.p_z$
8: $e_{1,x} \leftarrow triangle.e_{1,x}$
9: $e_{1,y} \leftarrow triangle.e_{1,y}$
10: $e_{1,z} \leftarrow triangle.e_{1,z}$
11: $e_{2,x} \leftarrow triangle.e_{2,x}$
12: $e_{2,y} \leftarrow triangle.e_{2,y}$
13: $e_{2,z} \leftarrow triangle.e_{2,z}$
14: $c_x \leftarrow p_x - ray.origin_x$
15: $c_y \leftarrow p_y - ray.origin_y$
16: $c_z \leftarrow p_z - ray.origin_z$
17: $r_x \leftarrow ray.dir_y \times c_z - ray.dir_z \times c_y$
18: $r_y \leftarrow ray.dir_z \times c_x - ray.dir_x \times c_z$
19: $r_z \leftarrow ray.dir_x \times c_y - ray.dir_y \times c_x$
20: $inv\_det \leftarrow 1 \div (ray.dir_x \times n_x + ray.dir_y \times n_y + ray.dir_z \times n_z)$
21: $t \leftarrow inv\_det \times (c_x \times n_x + c_y \times n_y + c_z \times n_z)$
22: $u \leftarrow inv\_det \times (e_{2,x} \times r_x + e_{2,y} \times r_y + e_{2,z} \times r_z)$
23: $v \leftarrow inv\_det \times (e_{1,x} \times r_x + e_{1,y} \times r_y + e_{1,z} \times r_z)$
24: **if** $u \geq 0$ and $v \geq 0$ and $(u + v) \leq 1$ and $0 \leq t$ and $t \leq tmax$ **then**
25: $\quad intersected \leftarrow$ true
26: **end if**

由於 triangle_intersect() 的執行流程較為單純（由上而下循序執行），因此其狀態圖設計上較為簡單，因此在報告中我只會討論包圍體階層走訪（Algorithm 1）的設計，而 triangle_intersect() 的部分則會使用一個 black box module 來表示。

C. Block Diagram (方塊圖) :



D. Definition of Inputs, Outputs, Control Signals, and Status Signals (輸入、輸出、控制訊號、及狀態訊號之定義):

輸入訊號：

valid（輸入訊號是否有效）
origin_x（光線原點的 x 座標，為 32-bit 單精度浮點數）
origin_y（光線原點的 y 座標，為 32-bit 單精度浮點數）
origin_z（光線原點的 z 座標，為 32-bit 單精度浮點數）
dir_x（光線方向向量的 x 分量，為 32-bit 單精度浮點數）
dir_y（光線方向向量的 y 分量，為 32-bit 單精度浮點數）
dir_z（光線方向向量的 z 分量，為 32-bit 單精度浮點數）
clk（時鐘訊號）
reset（重置訊號）

輸出訊號：

done（完成訊號）
intersected（是否相交到物體）
t（相交到物體的時間，為 32-bit 單精度浮點數）
u（相交到的三角形的重心座標 u，為 32-bit 單精度浮點數）
v（相交到的三角形的重心座標 v，為 32-bit 單精度浮點數）
n_x（相交到的三角形的法向量的 x 分量，為 32-bit 單精度浮點數）
n_y（相交到的三角形的法向量的 y 分量，為 32-bit 單精度浮點數）
n_z（相交到的三角形的法向量的 z 分量，為 32-bit 單精度浮點數）

控制訊號：

S_next（下一個狀態）
intersect_result_reg_init（將相交結果的暫存器初始化）
intersect_result_reg_update（將相交結果的暫存器設為對應數值）
counter_reg_reset（將計數器重置）
counter_reg_inc（將計數器加一）
stack_size_reg_reset（將堆疊的 size 重置）
stack_size_reg_inc（將堆疊的 size 加一）
stack_size_reg_dec（將堆疊的 size 減一）
node_data_reg_update（讀出 node 的資料並存入暫存器）
l_num_trigs_reg_dec（將左 node 的三角形數量減一）
r_num_trigs_reg_dec（將右 node 的三角形數量減一）
trig_data_reg_update（讀出三角形的資料並存入暫存器）
entry_exit_xyz_reg_update（將 entry_x、entry_y、entry_z、exit_x、exit_y、exit_z 的暫存器設為對應數值）
entry_exit_reg_update（將 entry、exit 的暫存器設為對應數值）
hit_reg_update（將是否擊中包圍體的資訊存入暫存器）
l_hit_reg_reset（將擊中左 node 的資訊重設）
r_hit_reg_reset（將擊中右 node 的資訊重設）

bram_node_addr_reg_init(將 node 對應的 block ram 的 address 的暫存器初始化）
bram_node_addr_reg_inc（將 node 對應的 block ram 的 address 的暫存器加一）
bram_node_addr_reg_update_l（將 node 對應的 block ram 的 address 的暫存器設為當前左 node 的左 child 的 address）
bram_node_addr_reg_update_r（將 node 對應的 block ram 的 address 的暫存器設為當前右 node 的左 child 的 address）
bram_node_addr_reg_update_stack(將 node 對應的 block ram 的 address 的暫存器設為堆疊頂端的值）
bram_trig_addr_reg_inc(將三角形對應的 block ram 的 address 的暫存器初始化）
bram_trig_addr_reg_update_l（將三角形對應的 block ram 的 address 的暫存器設為當前左 node 的第一個 triangle address）
bram_trig_addr_reg_update_r（將三角形對應的 block ram 的 address 的暫存器設為當前右 node 的第一個 triangle address）
stack_we（是否寫入堆疊）
ist_valid（是否執行三角形相交測試）
ist_done（是否完成三角形相交測試）
ist_intersected（是否相交到三角形）
sub_in_valid（是否啟用浮點數減法器）
sub_out_valid（浮點數減法是否完成）
div_in_valid（是否啟用浮點數除法器）
div_out_valid（浮點數除法是否完成）
less_in_valid（是否啟用浮點數比較器）
less_out_valid（浮點數比較是否完成）

<u>狀態訊號：</u>
INIT（初始化）
IDLE（閒置）
NODE_ADDR（將 node address 送入 block ram 中）
NODE_LOAD（從 block ram 中讀出 node 資料）
SUB（進行浮點數減法運算，對應於 Algorithm 1 的第 8~9 行）
DIV（進行浮點數除法運算，對應於 Algorithm 1 的第 8~9 行）
MINMAX_A（進行浮點數比較運算，對應於 Algorithm 1 的第 14~15 行）
MINMAX_B（進行浮點數比較運算，對應於 Algorithm 1 的第 14~15 行）
MINMAX_C（進行浮點數比較運算，對應於 Algorithm 1 的第 14~15 行）
HIT（進行浮點數比較運算，對應於 Algorithm 1 的第 17 行）
TRIG_LEFT_ADDR（將左 node 的三角形 address 送入 block ram 中）
TRIG_LEFT_LOAD（從 block ram 中讀出左 node 的三角形資料）
TRIG_LEFT_IST（進行左 node 的三角形相交測試，對應於 Algorithm 1 的第 21 行）
TRIG_RIGHT_ADDR（將右 node 的三角形 address 送入 block ram 中）
TRIG_ RIGHT _LOAD（從 block ram 中讀出右 node 的三角形資料）
TRIG_ RIGHT _IST（進行右 node 的三角形相交測試，對應於 Algorithm 1 的

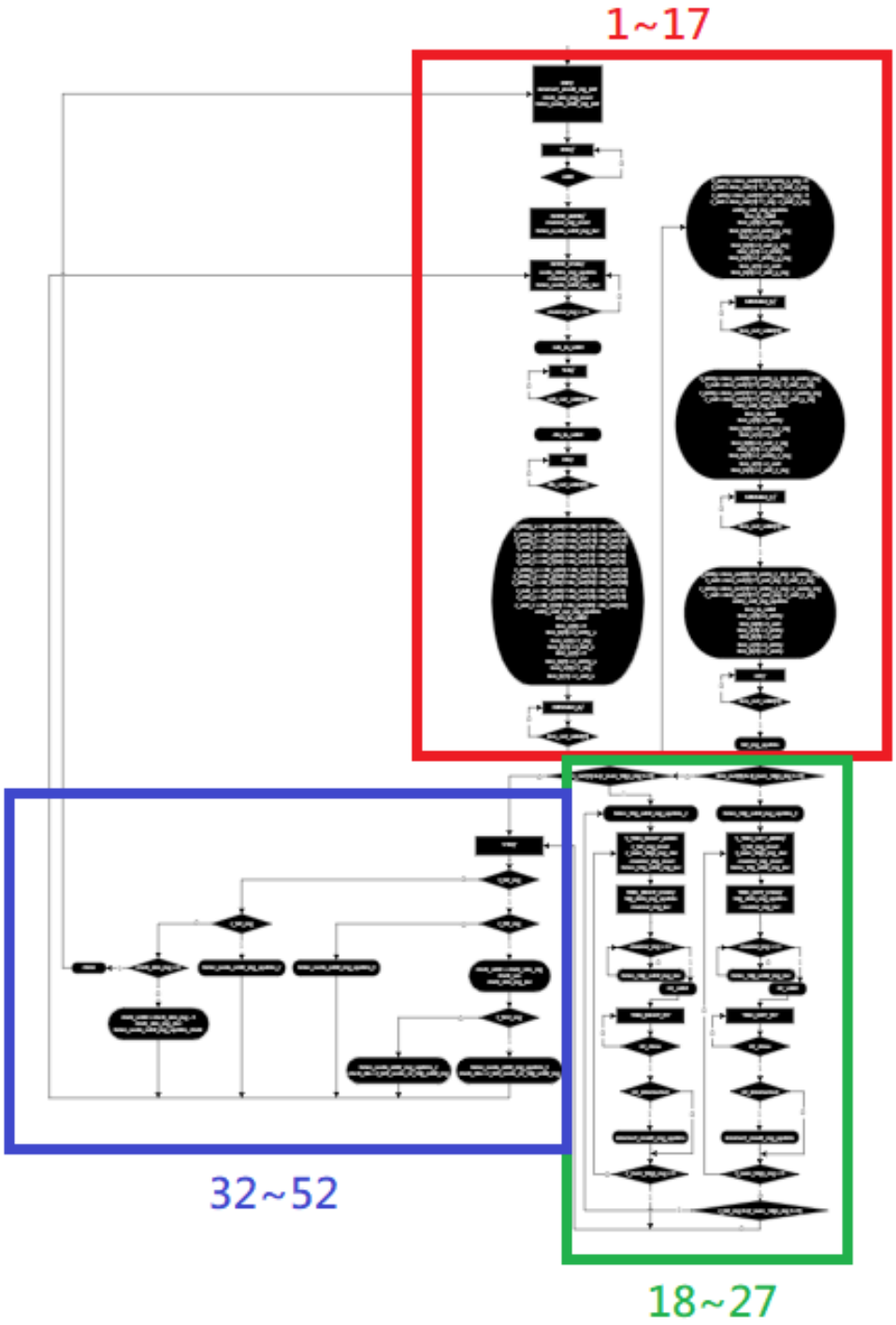
第 21 行）

STEP（決定樹走訪下一步要往哪裡走）

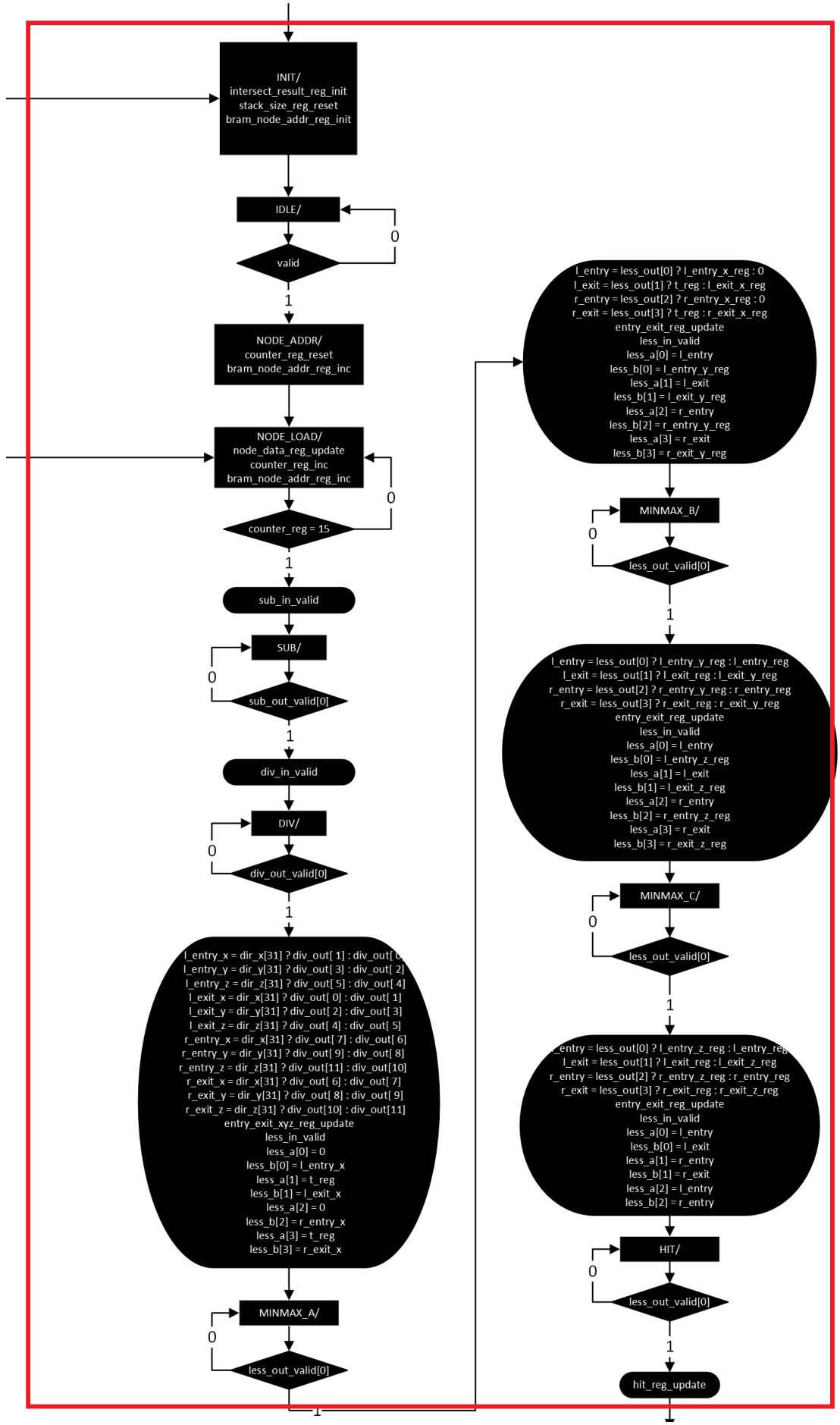
E. State Machine Chart (SM Chart) or State Graph (狀態機器圖或狀態圖):

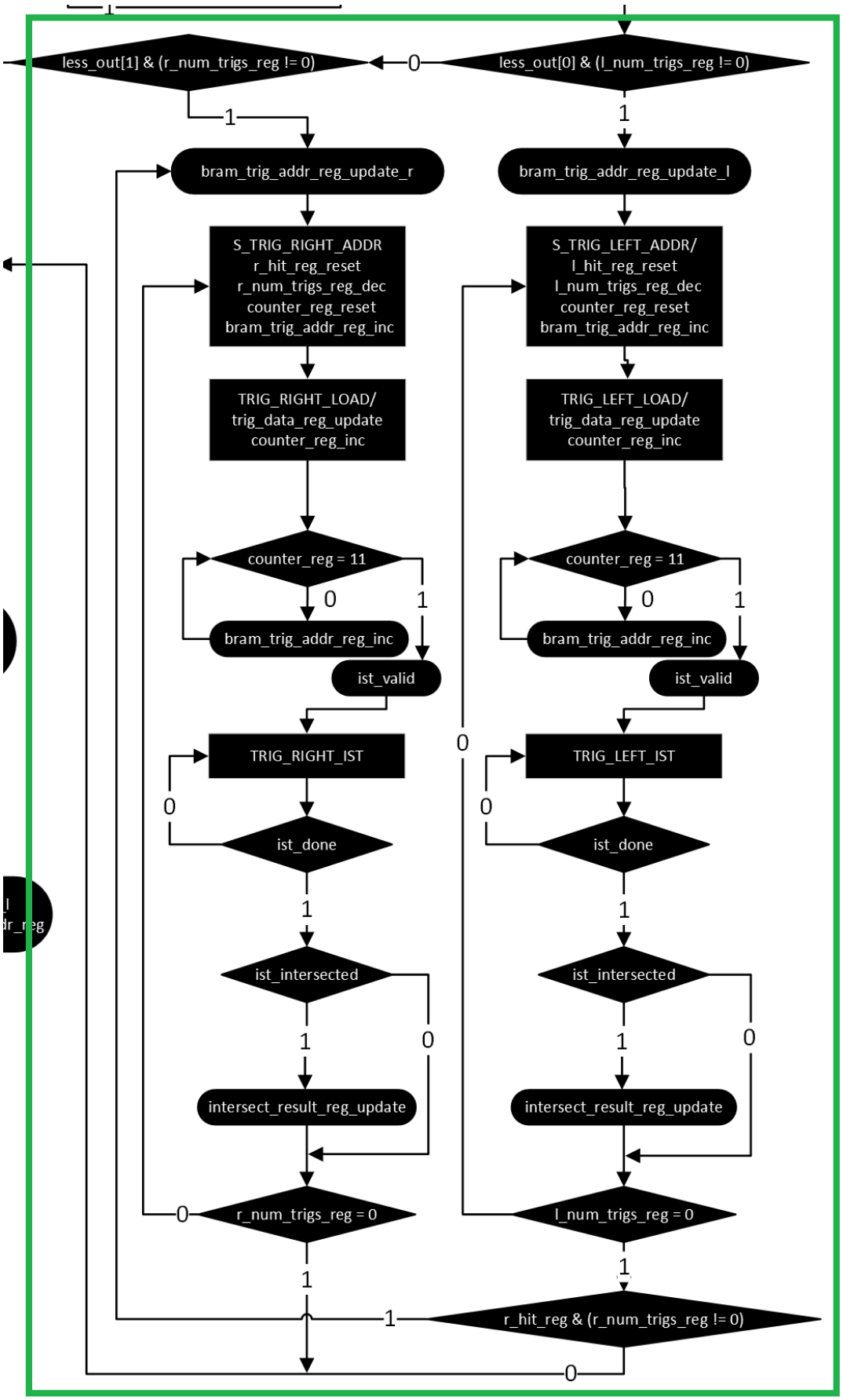
此連結為我使用 Visio 所設計的 SM Chart，是遵照「B. Flowchart or Procedure」所提及的「Algorithm 1」所設計。

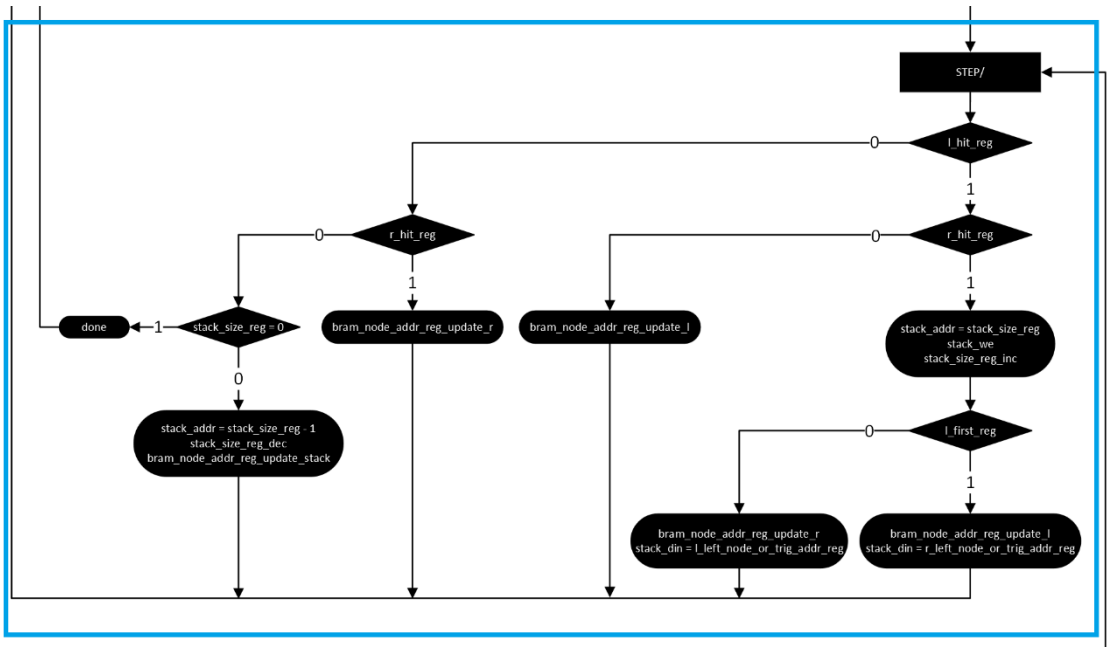
由於這個 project 的 SM Chart 複雜度較高，我將 SM Chart 分成三個部分，如下圖所示（圖上的數字表示對應於「Algorithm 1」的行數）：



之後三頁為各個區塊所的 SM Chart 放大圖。


INIT/
intersect_result_reg_init
stack_size_reg_reset
bram_node_addr_reg_init
IDLE/
0
valid
1
NODE_ADDR/
counter_reg_reset
bram_node_addr_reg_inc
NODE_LOAD/
node_data_reg_update
counter_reg_inc
bram_node_addr_reg_inc
0
counter_reg = 15
1
sub_in_valid
SUB/
0
sub_out_valid[0]
1
div_in_valid
DIV/
0
div_out_valid[0]
1
l_entry_x = dir_x[31] ? div_out[1] : div_out[0]
l_entry_y = dir_y[31] ? div_out[3] : div_out[2]
l_entry_z = dir_z[31] ? div_out[5] : div_out[4]
l_exit_x = dir_x[31] ? div_out[0] : div_out[1]
l_exit_y = dir_y[31] ? div_out[2] : div_out[3]
l_exit_z = dir_z[31] ? div_out[4] : div_out[5]
r_entry_x = dir_x[31] ? div_out[7] : div_out[6]
r_entry_y = dir_y[31] ? div_out[9] : div_out[8]
r_entry_z = dir_z[31] ? div_out[11] : div_out[10]
r_exit_x = dir_x[31] ? div_out[6] : div_out[7]
r_exit_y = dir_y[31] ? div_out[8] : div_out[9]
r_exit_z = dir_z[31] ? div_out[10] : div_out[11]
entry_exit_xyz_reg_update
less_in_valid
less_a[0] = 0
less_b[0] = l_entry_x
less_a[1] = t_reg
less_b[1] = l_exit_x
less_a[2] = 0
less_b[2] = r_entry_x
less_a[3] = t_reg
less_b[3] = r_exit_x
MINMAX_A/
0
less_out_valid[0]
1
l_entry = less_out[0] ? l_entry_x_reg : 0
l_exit = less_out[1] ? t_reg : l_exit_x_reg
r_entry = less_out[2] ? r_entry_x_reg : 0
r_exit = less_out[3] ? t_reg : r_exit_x_reg
entry_exit_reg_update
less_in_valid
less_a[0] = l_entry
less_b[0] = l_entry_y_reg
less_a[1] = l_exit
less_b[1] = l_exit_y_reg
less_a[2] = r_entry
less_b[2] = r_entry_y_reg
less_a[3] = r_exit
less_b[3] = r_exit_y_reg
MINMAX_B/
0
less_out_valid[0]
1
l_entry = less_out[0] ? l_entry_y_reg : l_entry_reg
l_exit = less_out[1] ? l_exit_reg : l_exit_y_reg
r_entry = less_out[2] ? r_entry_y_reg : r_entry_reg
r_exit = less_out[3] ? r_exit_reg : r_exit_y_reg
entry_exit_reg_update
less_in_valid
less_a[0] = l_entry
less_b[0] = l_entry_z_reg
less_a[1] = l_exit
less_b[1] = l_exit_z_reg
less_a[2] = r_entry
less_b[2] = r_entry_z_reg
less_a[3] = r_exit
less_b[3] = r_exit_z_reg
MINMAX_C/
0
less_out_valid[0]
1
l_entry = less_out[0] ? l_entry_z_reg : l_entry_reg
l_exit = less_out[1] ? l_exit_reg : l_exit_z_reg
r_entry = less_out[2] ? r_entry_z_reg : r_entry_reg
r_exit = less_out[3] ? r_exit_reg : r_exit_z_reg
entry_exit_reg_update
less_in_valid
less_a[0] = l_entry
less_b[0] = l_exit
less_a[1] = r_entry
less_b[1] = r_exit
less_a[2] = l_entry
less_b[2] = r_entry
HIT/
0
less_out_valid[0]
1
hit_reg_update


less_out[1] & (r_num_trigs_reg != 0)
less_out[0] & (l_num_trigs_reg != 0)
0
1
bram_trig_addr_reg_update_r
bram_trig_addr_reg_update_l
S_TRIG_RIGHT_ADDR
r_hit_reg_reset
r_num_trigs_reg_dec
counter_reg_reset
bram_trig_addr_reg_inc
S_TRIG_LEFT_ADDR/
l_hit_reg_reset
l_num_trigs_reg_dec
counter_reg_reset
bram_trig_addr_reg_inc
TRIG_RIGHT_LOAD/
trig_data_reg_update
counter_reg_inc
TRIG_LEFT_LOAD/
trig_data_reg_update
counter_reg_inc
counter_reg = 11
bram_trig_addr_reg_inc
ist_valid
TRIG_RIGHT_IST
TRIG_LEFT_IST
ist_done
ist_intersected
intersect_result_reg_update
r_num_trigs_reg = 0
l_num_trigs_reg = 0
r_hit_reg & (r_num_trigs_reg != 0)



F. Description of Verilog Code (Verilog 電路模組說明):

這個 project 總共有 5 個 Verilog module：

- block_ram.v
- dist_ram.v
- float_operator.v
- ist.v
- rtunit.v

接下來我會一一說明這 5 個 module 的輸入輸出訊號與功能。

block_ram.v

參數：

- RAM_WIDTH（RAM 中每筆資料的 width）
- RAM_DEPTH（RAM 總共能放多少筆資料）
- INIT_FILE（RAM 初始資料檔案路徑）

輸入訊號：

- clk（時鐘訊號）
- we（write enable 訊號）
- addr（位置訊號）
- din（資料輸入）

輸出訊號：

- dout（資料輸出）

功能描述：

這是一個 Synchronous Read、Synchronous Write 的 RAM。

dist_ram.v

參數：

- RAM_WIDTH（RAM 中每筆資料的 width）

- RAM_DEPTH（RAM 總共能放多少筆資料）
- INIT_FILE（RAM 初始資料檔案路徑）

輸入訊號：

- clk（時鐘訊號）
- we（write enable 訊號）
- addr（位置訊號）
- din（資料輸入）

輸出訊號：

- dout（資料輸出）

功能描述：

這是一個 Asynchronous Read、Synchronous Write 的 RAM。

float_operator.v

參數：

- OPERATION（可以設為”add”、”sub”、”mul”、”div”、”less”，代表欲執行的操作）
- LATENCY（輸入要等多久個 cycle 才會有結果）

輸入訊號：

- clk（時鐘訊號）
- valid（當輸入訊號為合法時設為 1）
- a（第一個浮點數輸入）
- b（第二個浮點數輸入）

輸出訊號：

- done（完成訊號）
- result（浮點數輸出）

功能描述：

所有會用到的 32-bit 浮點數操作都在這個 module 內，包括加法、減法、乘法、除法、比較。舉例來說，當我們要將兩個 32-bit 數相加，我們可以 instantiate 一個 float_operator，並將其 OPERATION 設為”add”，輸入接到 a 及 b，並將 valid 設成 1，在 LATENCY 個 cycle 之後，done 會變成 1 且 dout 為運算結果。

這個 module 之中我並沒有實作完整的電路（因為這個 project 主要著重在 rtunit.v），而是採用 Verilog 內建的 real 來做模擬，並使用 shift register 來實作出 LATENCY 的效果。

ist.v

輸入訊號：

- clk（時鐘訊號）
- reset（重置訊號）
- valid（當輸入訊號為合法時設為 1）
- origin_x、origin_y、origin_z、dir_x、dir_y、dir_z、tmax（光線資訊）
- p0_x、p0_y、p0_z、e1_x、e1_y、e1_z、e2_x、e2_y、e2_z、n_x、n_y、n_z（三角形資訊）

輸出訊號：

- done（完成訊號）
- intersected（是否相交）
- t（相交到三角形的時間）
- u（相交到的三角形的重心座標 u）
- v（相交到的三角形的重心座標 v）

功能描述：

這個 module 是在做三角形相交測試的。給定光線及三角形資訊，這個 module 會計算光線是否相交到三角形，並輸出相交資訊（t、u、v）。

rtunit.v (Top Module)

輸入訊號：

- clk（時鐘訊號）
- reset（重置訊號）
- valid（當輸入訊號為合法時設為 1）
- origin_x、origin_y、origin_z、dir_x、dir_y、dir_z、tmax（光線資訊）

輸出訊號：

- done（完成訊號）
- intersected（是否相交）
- t（相交到三角形的時間）
- u（相交到的三角形的重心座標 u）
- v（相交到的三角形的重心座標 v）
- n_x（相交到的三角形的法向量的 x 分量）
- n_y（相交到的三角形的法向量的 y 分量）
- n_z（相交到的三角形的法向量的 z 分量）

功能描述：

給定光線資訊，這個 module 會走訪包圍體階層、使用 ist submodule 進行相交測試，並輸出距離最近的相交資訊。

在這個 project 中，會 instantiate 兩個 block_ram，一個是儲存包圍體階層的節點資訊，另一個是儲存三角形的資訊。為了簡化設計，我使用 C++ 預先將兩個 block_ram 中儲存的資料寫入檔案中，並在 Verilog 中使用$readmemh 為其設定初始值。

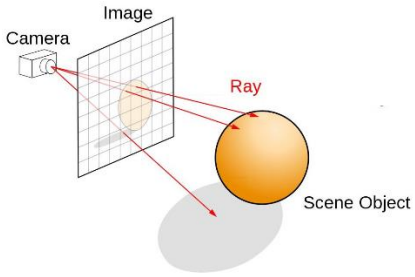
G. Description of Test Bench (Verilog 測試模組說明):

tb_rtunit.v 為這個 project 的 testbench。這個 testbench 會使用物體的法向量渲染出 100*100 的圖像，而物體為計算機圖學中廣為採用的 bunny 模型。

以下我將說明 testbench 的運作流程。對於圖像的每個像素，我會在 testbench 中計算該像素對應到的光線資訊，並將光線資訊送給 rtunit，待 rtunit 計算完畢求得 n_x、n_y、n_z 後套用以下公式計算 RGB 各個 channel 對應的色彩強度（介於 0~1）並寫入圖檔（image.ppm）：

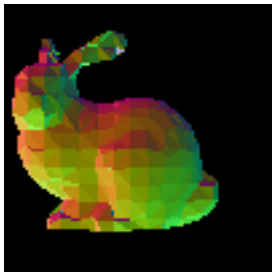
$$tmp = sqrt(n\_x * n\_x + n\_y * n\_y + n\_z * n\_z)$$
$$r = (n\_x / tmp + 1) / 2$$
$$g = (n\_y / tmp + 1) / 2$$
$$b = (n\_z / tmp + 1) / 2$$

其發送光線的流程可用下圖簡要描述：



H. Simulation Results (模擬結果):

testbench 執行完畢後可以得到正確的渲染圖像，如下圖所示：



此外，在 console 中也能看到每條 ray 走訪包圍體階層與進行相交測試的過程。

I. Conclusions and Discussions (心得、感想、結論、及討論):

這個 project 花了好幾天才做完，尤其是在走訪包圍體階層的地方。因為包圍體階層本身屬於樹的結構，因此相較於軟體的實作方法，在硬體上更不易實作（舉例來說在讀資料時會有很多 offset 的細節要考慮），但是在做完這個 project 之後我想我對如何有效地在 Verilog 上實作複雜設計有更深入的理解了。

References (參考資料) : (請說明各參考項目對你的專題提供那方面資料)

[1] Ray tracing (graphics) – Wikipedia
Ray tracing 的維基百科條目。

[2] NVIDIA Turing Architecture Whitepaper
NVIDIA Turing 架構的白皮書。Turing 架構首次增加了 RT Core，是包圍體階層走訪及相交測試的專用電路。

[3] "RDNA 2" Instruction Set Architecture: Reference Guide
AMD RDNA 2 架構的 ISA 指南。其中也增加了關於包圍體階層相交測試的指令集。

[4] Bounding volume hierarchy – Wikipedia
包圍體階層的維基百科條目。

[5] 4.3 Bounding Volume Hierarchies – PBR Book

包圍體階層的詳細介紹。

[6] madmann91/bvh: A modern C++ BVH construction and traversal library
開源的包圍體階層函式庫。

[7] Möller–Trumbore intersection algorithm - Wikipedia
Möller-Trumbore 射線-三角形相交算法