

邏輯設計實驗 *final report* 第18組

Topic - Double pong game



大綱

- Introduction
- Motivation
- Function
- Development Process
- AI Training
- Game Engine
- Client FPGA
- Experience result
- Conclusion

Introduction

Double pong game是一款類似pong game的遊戲，但他將pong game擴大化，他將帶給你的快樂絕對是遠遠大於兩倍的pong game，你可以在裡面訓練你的左右腦與手眼協調，或是體驗AI與人對抗的快感，從中找到屬於你的一片天。

Motivation



Pong game算是一個學習training RL moduel會遇到的練習題，但我個人覺得普通的pong game其實沒什麼訓練AI的必要，寫一個算法讓板子一直追著球跑就好了，因此才會衍生出今天這款遊戲，當球不只有一個，板子也不只有一個的時候，怎麼樣分配板子的工作成了最大的問題，並且更進階的問題是如何透過回擊疊高球的速度並且在適當的時機打回去給對手，讓對手措手不及。

Gmae spec

- 維持原本pong game的特色(球的移動、操作板子...)
- 將球數量增加到5顆，板子數量增加到2
- 球不管哪一面碰到板子都會回彈，並且根據彈的方向變換顏色，
- 雙方血量條製作
- 可以選擇1P對AI，或是2P玩家互打
- 可以隨時暫停遊戲或開始遊戲
- 可以根據玩家個人喜好選擇適當的板子速度

Development Process

為了節省時間，我們決定在電腦上先訓練好AI，再將訓練好的結果搬到FPGA上面。

我們使用Python來訓練AI，所以在最一開始先用Python將我們遊戲的原型實作出來，之後遵照Gym的格式將遊戲重構一次，最後就是將AI模型跟Gym版本的遊戲互動來訓練。

然後將訓練好的AI權重轉移到FPGA上面去。

所以我們開發的順序依次如下

Python Game



在開發上為了能更好渲染遊戲，因此我們採用pygame來繪製遊戲狀況，然後自己實作遊戲運作模式。上圖是開發完後用python在本機端運行的樣子。

Gym

在之前的努力下我們已經有了一個python的原型，但為了更好訓練AI，所以我們將遊戲依照常見的規格製作API，方便AI模型取得資料。

以下是我們Gym可以使用的API。

- Init
- Reset
- Step
- Render
- Action_space
- Observation_space

FPGA design

基本上的遊戲架構遵照實作python時的設計，最後再依照spec調整遊戲顯示畫面，並且處理一些python實作上未注意到的小問題。

AI Training

- Basic Settings
- PPO

- Q-Learning
- Model Transfer

Basic Settings

在一開始訓練時我們希望能不要給AI任何人為的影響，所以我們將我們能從遊戲中取得最原始的資料給予AI做判斷。


- State & Action design
 - Observation space :
24個離散的數字，範圍 $((-640,640))$ ，前20個數字代表編號1~5號球的 (x,y) 座標和 $((V_x,V_y))$ 速度，後面4個數字代表著左邊板子1的位置 $((x,y))$ 和板子2的位置 $((x,y))$ 。
 - Action space :
4個離散的數字，範圍 $(([0,4))$ ，分別代表左邊板子1往上、下和左邊板子2往上、下。然後為了training方便，統一都訓練左邊。
- Reward design
如果失分，reward-10，如果打到球並且是往右邊打就reward+5+球速，如果是往左邊打reward-5-球速。
這樣設計的原因是希望AI能學到如何通過反彈去疊加球的速度，但又希望在疊的時候不要不小心失分，因此只要是左邊打的都會扣分，而右邊打的才會加分，並且球打越快會越高分。
- Opponent behavior settiing
為了模仿球反彈得情況發生，我們對手行為是設定有45%會往左反彈，10%往右反彈。
不用策略追蹤原因是因為我們也無法構造出人類在判斷多顆球和多板子的策略。

PPO

- 模型選用:
 - nikhilbarhate99/PPO-PyTorch
 - Ray rllib PPO
 - Ray rllib DQN
- 效能比較:

模型	average reward
Ray rllib DQN	-1000 ~ -800
Ray rllib PPO	-400 ~ -300
Custom PPO	-150 ~ -30

• Ray rllib



Ray v2.0.0.dev0

Search the docs ...

What is Ray?

OVERVIEW OF RAY

RAY CORE

MULTI-NODE RAY

RAY SERVE

RAY DATA

RAY WORKFLOWS

RAY TUNE

RAY RLlib

RLlib: Industry-Grade Reinforcement Learning

RLlib Table of Contents

← ↻ ⌵ Contents

RLlib in 60 seconds


Feature Overview

Customizations

We're hiring!

The RLlib team at [Anyscale Inc.](#), the company behind Ray, is hiring interns and full-time **reinforcement learning engineers** to help advance and maintain RLlib. If you have a background in ML/RL and are interested in making RLlib the industry-leading open-source RL library, [apply here today](#). We'd be thrilled to welcome you on the team!

RLlib: Industry-Grade Reinforcement Learning



RLlib is an open-source library for reinforcement learning (RL), offering support for production-level, highly distributed RL workloads while maintaining unified and simple APIs for a large variety of industry applications. Whether you would like to train your agents in a multi-agent setup, purely from offline (historic) datasets, or using externally connected simulators, RLlib offers a simple solution for each of your decision making needs.

You **don't** need to be an **RL expert** to use RLlib, nor do you need to learn Ray or any other of its libraries! If you either have your problem coded (in python) as an **RL environment** or own lots of pre-recorded, historic behavioral data to learn from, you will be up and running in only a few days.

RLlib is already used in production by industry leaders in many different verticals, such as climate control, manufacturing and logistics, finance, gaming, automobile, robotics, boat design, and many others.

Algorithm	Frameworks	Discrete Actions	Actions	Agent	Model Support
A2C, A3C	tf + torch	Yes +parametric	Yes	Yes	+RNN, +LSTM auto-wrapping, +Attention, +
ARS	tf + torch	Yes	Yes	No	
BC	tf + torch	Yes +parametric	Yes	Yes	+RNN
CQL	tf + torch	No	Yes	No	
ES	tf + torch	Yes	Yes	No	
DDPG, TD3	tf + torch	No	Yes	Yes	
APEX-DDPG	tf + torch	No	Yes	Yes	
Dreamer	torch	No	Yes	No	+RNN
DQN, Rainbow	tf + torch	Yes +parametric	No	Yes	
APEX-DQN	tf + torch	Yes +parametric	No	Yes	
IMPALA	tf + torch	Yes +parametric	Yes	Yes	+RNN, +LSTM auto-wrapping, +Attention, +
MAML	tf + torch	No	Yes	No	
MARWIL	tf + torch	Yes +parametric	Yes	Yes	+RNN
MBMPO	torch	No	Yes	No	

Ray rllib是一款python用來訓練AI的套件，而該套件庫也有提供很多算法，但因為我們只有學過DQN和PPO，因此只以這兩個做測試訓練，不過因為初期的訓練狀況比個人客製化的還爛，所以最後選擇不採用。

• Custom PPO

詳細模板是參考nikhilbarhate99/PPO-PyTorch的，但這個模型原本是用在訓練CartPole-v1，所以我有對比我們的遊戲做一下參數上的調整。

• Tune 過後的PPO

一開始使用的方法是Ray提供的tune工具，這個工具是你只要提供你參數的範圍和特性，他可以幫你用一些策略性

的算法窮舉，並且這個tune是平行化的程式，因此可以一次用多顆CPU和GPU做參數搜尋。(在此特別感謝國網中心有提供server可以讓我們運行)

Tune過的結果的參數有點不太具有解釋性，但是還是比直接使用Ray的PPO去訓練的結果來的要好。

- PPO訓練成果

<https://drive.google.com/file/d/1x6rLgueXDMcohJSwJJ4KCHIW8OzDxrWr/view?usp=sharing>

(<https://drive.google.com/file/d/1x6rLgueXDMcohJSwJJ4KCHIW8OzDxrWr/view?usp=sharing>).

- 調整遊戲設定

為了讓我們的AI變得更加強大，我們也開始嘗試調整遊戲設定，而歸類出以下重點：

- 在測試中發現如果將球的軌道固定為2種發射路徑，將可將平均reward提高到50~100，但如果路徑越多元化平均reward也會降低。
- 如果先讓AI在單一發射路徑的遊戲訓練過一陣子後，再將它轉移到隨機路徑上會有比直接在隨機路徑下訓練的結果還要好。
- 對手的行為也會有重大影響，如果對手打到的機率夠大也會讓訓練難度提升。

總結來說一完全不調整遊戲資訊的狀況下對PPO來說這個遊戲還是具有一定的難度，因此我們決定來透過些微的人為處理來幫助AI學習。

Q-Learning

當初為了能讓AI超越人類，所以希望給AI的資訊是完全沒有被人工加工過的，但因為之前訓練狀況不佳，因此打算將State做一些人工上的判斷，不過由於經過人工判斷後state數量下降許多，因此評估就算用q-learning也做得出來，因此之後就嘗試使用q-learning來做訓練。

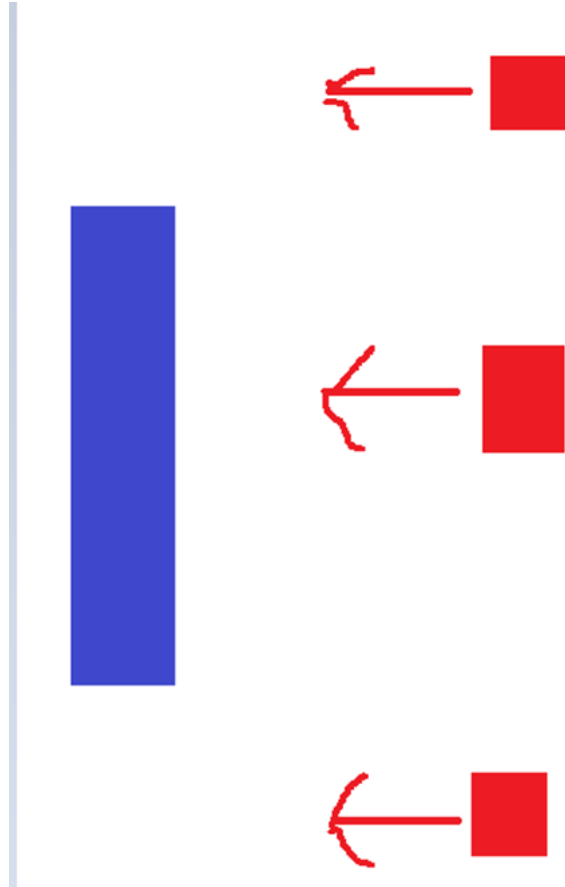
- new state design

新的state是設定每種球都可以有三種狀況，會被目前板子打到，在目前板子上面和在目前板子下面，這樣即可讓AI拿到有關y軸部分的資訊，而球的x軸資訊我們利用x的大小來排序state的順序由小到大，也就是最接近板子的球它的資訊會在第一個，離板子最遠的球它的資訊會是最後一個。而這樣子的可能性會是 $(4 * (3 * 3)^{10})$ ，經過計算後

是可以直接在FPGA裡面建表的。

不過上面這個的訓練結果也不太好，推斷是AI沒辦法確認球最終是往哪裡飛，而如果要把速度的資訊也加進state會造成表太大，因此最終的state設定改成預測3個偵數後的球會是位於板子的哪個部份來讓AI做判斷。

以下面這張圖為例第一顆球會被判斷為狀態1第二顆球是狀態2第三顆球是狀態3。



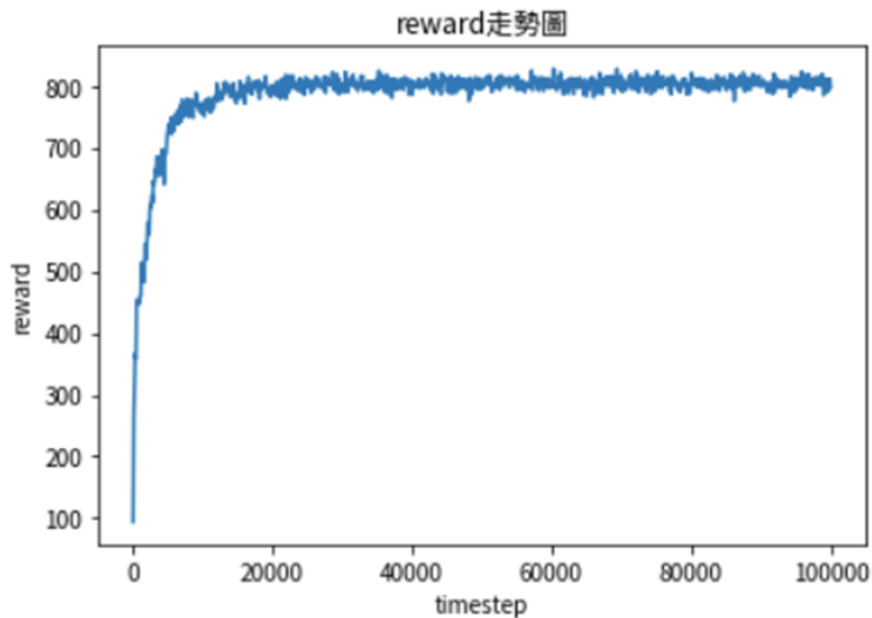
- Q-learning訓練小技巧

由於有了之前PPO給的經驗，因此我們將 γ 設定在 (0.9) 附近的範圍，並且為了更精準的訓練需要將 α 調小至差不多 (0.001) 才会有比較好的結果，我們嘗試過指數下降 α 可以最快train出結果。

- 訓練成果

AI在對手完全不動的狀況下可以每場平均失球 $(17) \sim (20)$ 顆，在對手有機會反彈的狀況平均失球 $(46) \sim (50)$ 顆。

不過有很多球的失分原因是跟本身遊戲機制不完全有關，因此決定停止訓練。



成果影片(建議載下來觀看)

<https://drive.google.com/file/d/18CF99d9PWTa90YMwmssxf2DSNjCCe21n/view?usp=sharing>

(<https://drive.google.com/file/d/18CF99d9PWTa90YMwmssxf2DSNjCCe21n/view?usp=sharing>).

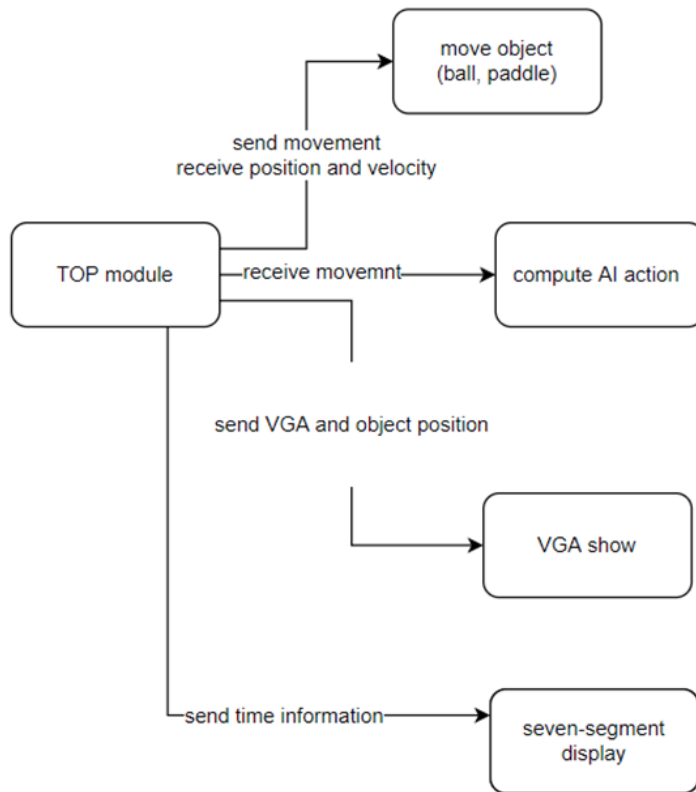
Model Transfer

因為我們是利用q-learning去train出我們的AI，因此我們只需要單純地將我們所training出來的q-table直接取每個狀態最好的操作轉成.coe檔的格式，透過block memory generator存進去FPGA裡面，再design 相對應的AI module 去執行查表的行為並action即可。

Game Engine

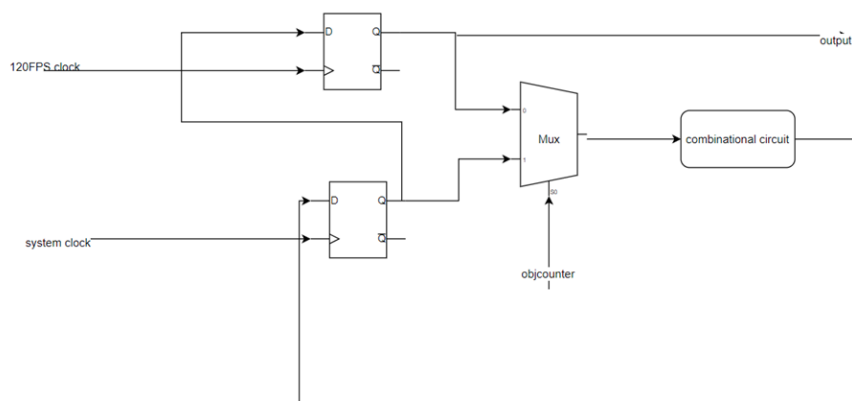
- Game server design
- Main FPGA I/O
- VGA顯示

Game server design



上圖是我們主要FPGA的架構圖

- 遊戲基礎參數設計
 - 遊戲更新率：120 FPS
 - 畫面更新率：60 FPS
 - 場景 長：640 px 寬：480 px
 - 球半徑：6px
 - 板子 長：80 px 寬：4 px
- 遊戲模擬實作



由於遊戲更新頻率是120Hz，但在每次更新有許多東西要計算，所以設計成有兩個DFF，其中一個會記錄每次遊戲更新資訊，而另一個則是根據目前計算狀況做更新，然後每1/120秒將資訊更新進遊戲裡面，而其中combinational circuit就由一個counter紀錄目前要計算的資訊是要從哪個

DFF拿取。

因為我們有5顆球，每顆球的動作模式都差不多，所以這樣設計希望能減少combination circuit的用量，每次combination circuit就只對我們要處理的球做運算，不需要每顆球都生成新的combination circuit。

○ 模擬流程

- 更新球新的位置
- 判斷是否撞到上面或下面的牆壁
- 判斷是否有撞到板子
- 判斷是否有撞到底線

○ 注意事項:

-Clock與儲存計算

- 因為每次更新時都是抓上一次的資訊，所以在判斷上要十分注意是否有更新過。
- 或是每一顆球更新完要到下一個state的時候的clock要特別小心處理。

■ Signed 與 Unsigned

- 因為球的方向有左有右，所以我們設計往左(x軸)和往上(y軸)為負，往右(x軸)和往下(y軸)為正，因此跟球運作有關的參數都要使用signed。

```
parameter signed WIDTH = 11'd640;
parameter signed HEIGHT = 11'd480;
```

- 當我們今天要用一些signal組合出我們要的數字時，如果直接用數學運算比如{9'b0, out2[4], out2[6]} * -11'b1的寫法就有可能會出問題。所以在組合上就直接組合出負數。

```
next_ball_vel[id][0] = {9'b11111111, out2[4], out2[6]};
```

- Unsigned和signed做運算:兩個不同型態的signal相互做運算可能會導致無法預期的問題，為了避免這樣的問題我們統一採用下圖的coding style去解決

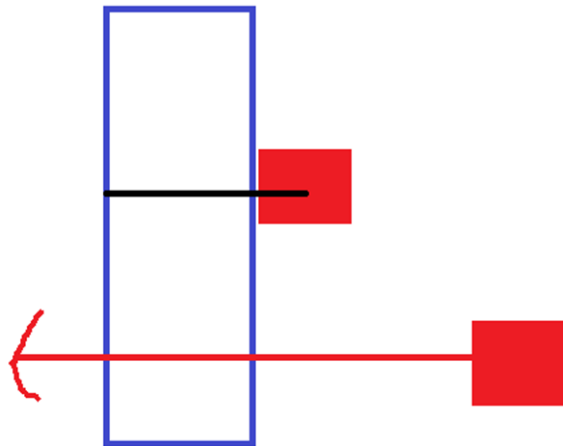
```
{11'd320 + ({2'b0, Play1_S} - {2'b0, Play2_S}) * 11'd2);
{11'd320 - ({2'b0, Play2_S} - {2'b0, Play1_S}) * 11'd2);
```

- Signed的正負性判別:對於Signed正負性判斷，因為我們在design module時，有發現單純判斷signal > 0有時會出現奇怪的bug，因此我們統一改為判斷MSB為準

- 遊戲中所遇到的問題：

- 碰撞穿牆問題：

很多網路上的pong game範例在做碰撞處理都是使用球的位置做判定，只要球的圓心加上半徑超過板子就反彈，但這種方法在我們的遊戲裡會有問題，如果一個球的速度過快會導致她在前一幀還沒碰撞，但下一幀則會直接穿過板子。



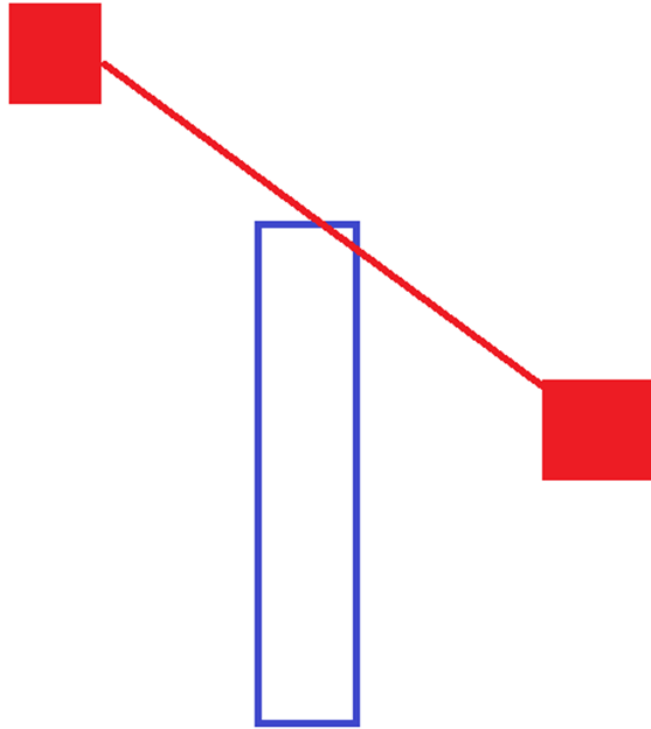
如上圖大部分會判斷黑線的長度決定是否反彈，但如果單次位移超過黑線長度就會穿牆

- 解決方法:

因此我們在做碰撞是檢測他下一幀是否有超過板子，並且判斷目前的位置在板子前面。

- 斜對角穿射問題：

如果有一個球他在前一幀的y軸還在板子的下面，但它下一幀卻飛到板子上。在這種情況如果位置的更新是直接座標加上速度，然後用目前的座標或是之後的座標去判斷碰撞都無法得到正確的結果。



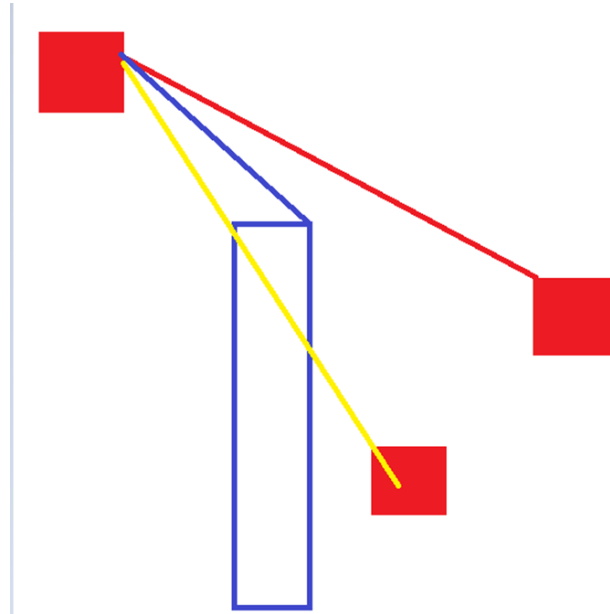
如上圖我們只用最初或最終球的位置是無法判斷中途有沒有撞到。而當初python遊戲在設計時就是因為沒考慮到這點，才會倒置AI夠強後還是會掉那麼多球。

- 解決方法：

- 向量外積：

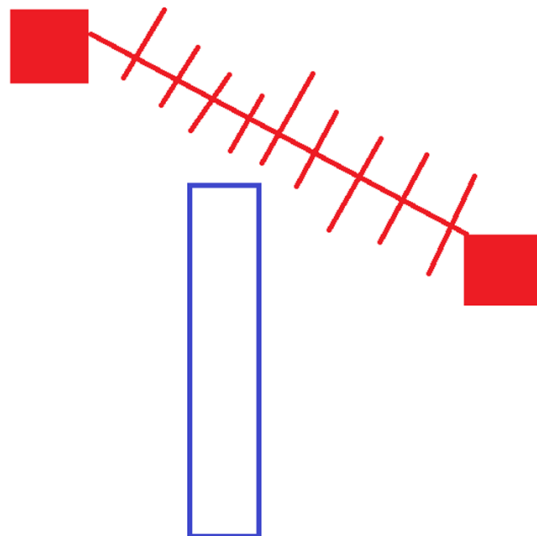
對於這類幾何問題常見的解法就是用向量的內積和外積，我們可以用兩個向量，分別是新的球到舊球的向量，和新球到板子的向量。然後將兩個向量做外積，然後在依據外積結果的正負判斷兩個向量誰在上面就可得知球的路徑是否會撞到板子，然後再用內積判斷他是在那裡撞到的怎麼反彈，但在verilog寫數

學式明顯不是一個好作法。



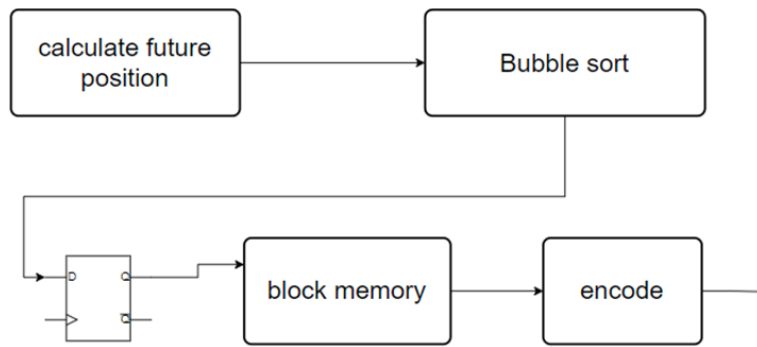
■ 更新方式調整：

仔細思考一下問題，會發現主要造成這個的原因是y軸的單位變化量太大了，所以假設我們每次更新都只更新1px，然後少量多次的位移就可以解決這個問題了，並且根據場地的大小推斷速度不可能超過640，且最高5顆球，所以絕對能在一幀內更新完畢。



● 遊戲AI計算

○ 架構設計



首先先將原本遊戲給的資訊轉換成AI要的state，然後再去block memory裡面查表，根據結果在做對應到遊戲所使用的action規格。

■ bubble sort

```

always @(*) begin
    for (i = 3'd0; i < 3'd5; i = i + 3'd1) begin
        array[i+1] = i;
    end

    for (i = 5; i > 0; i = i - 1) begin
        for (j = 1; j < i; j = j + 1) begin
            if (future_ball_pos[array[j]][0] > future_ball_pos[array[j + 1]][0]) begin
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            end
        end
    end

    for (i = 0; i < 5; i = i + 1) begin
        sorted_bus[i][0] = future_ball_pos[array[i+1]][0];
        sorted_bus[i][1] = future_ball_pos[array[i+1]][1];
    end
end
  
```

為了排序state，我們使用bubble sort來排序，每次對相鄰的兩個index去比較他們位置的y軸，最後再將排序好的資料傳送進DFF裡面。

■ Encode and Decode

■ Encode

每個球可以有 (3×3) 種狀況，也就是 **(第一個板子的上中下)** 乘以 **(第二個板子的上中下)**，並且有五顆球，所以有 $((3 \times 3)^5)$ 。因此將sort好的state跟板子比較方向，然後定義上為 (0) 中為 (1) 下為 (2) ，再將結果以3進位存起來。

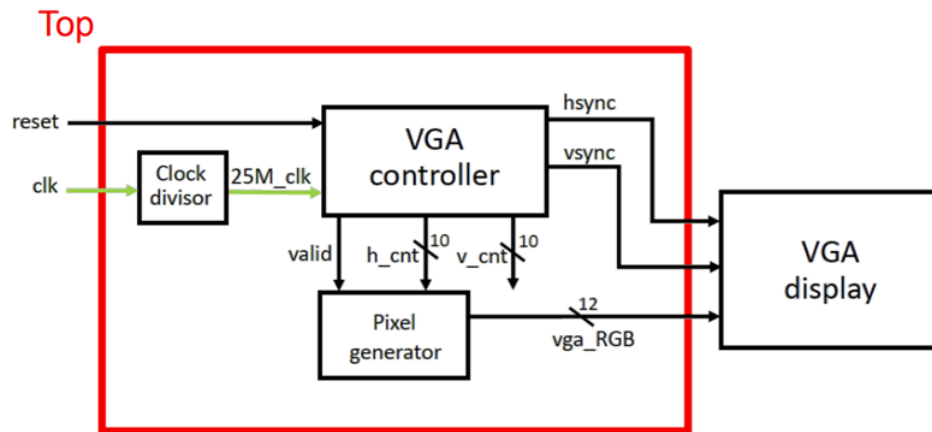
■ Decode

因為AI能進行的操作有4個，分別是控制兩個板子的上下，而block memory所存的值也是0~4，因此要對應到使用者的操作需要將每個的值加一輸出。

Main FPGA I/O

- Input signal
 - reset(U18)
 - clk (W5)
 - play2_M[2:0](J1, L2, J2) 使用者1的輸入
 - play1_M[2:0](A14, A16, B15) 使用者2的輸入
 - start(T17) 控制遊戲開始暫停
 - mode(V17) AI 模式開關
 - pad1_add_vel[1:0](R2, T1) 使用者1對板子速度的調控
 - pad2_add_vel[1:0](U1, W2) 使用者2對板子速度的調控
- Output signal
 - seven[7:0] 七段顯示器訊號
 - an[3:0]
 - play2_S[8:0](R18, P17, M19, L17, G2, H1, K2, H2, G3) 使用者2的分數
 - play1_S[8:0](P18, N17, M18, K17, B16, A15, A17, C15, C16) 使用者1的分數
 - led[15:0] 測試輸出
 - VGAR[3:0] (N19, J19, H19, G19)
 - VGAG[3:0] (D17, G17, H17, J17)
 - VGAB[3:0] (J18, K18, L18, N18)
 - hsync (P19)
 - vsync (R19)

VGA顯示



在VGA的設計大致如課程的VGA顯示一樣。原本有打算使用圖片顯示背景，但畫質實在過於慘不忍睹，所以後面全部採用自己繪畫的方式實作。

- 繪圖

- 板子跟球的更新

根據我們所設計的板子長度、球的半徑，從它們存取的位置點來畫圖，並且給予相對應的色彩，透過判斷球的速度正負，可以讓球在被打到時，更新自己成相對應的顏色。

- 血量條控制

因為server端會分別存取兩位玩家的分數，所以我們同樣可以把分數的signal傳進我們的module中，然後藉有VGA的h_cnt去計算，做出血量條的效果，另外這邊要注意的是計算時，要考慮正負性，所以在實作的過程中，我們先用簡單的if-else語法判別兩位玩家分數的大小再行計算、繪圖。


```

1  if(Play1_S > Play2_S)begin
2      if(h_cnt <= lf || lf < 11'd32
3          pixel = {4'd0, 4'd15,
4      end else begin
5          pixel = {4'd0, 4'd0,
6      end
7  end else if(Play1_S == Play2_S)begin
8      if(h_cnt <= 11'd319)begin
9          pixel = {4'd0, 4'd15,
10     end else begin
11         pixel = {4'd0, 4'd0,
12     end
13 end else begin
14     if(h_cnt <= rf && rf < 11'd32
15         pixel = {4'd0, 4'd15,
16     end else begin
17         pixel = {4'd0, 4'd0,
18     end
19     end

```

○ 遊戲封面

圖片轉成.coe，用memory block generator來存取

○ 倒數、暫停、準備、WIN繪圖

- 準備：當遊戲從封面剛開始進入遊戲時，位於遊戲正中間偏上會出現READY標題
- 暫停：當進入遊戲後，玩家可以透過button使遊戲暫停，此時位於遊戲正中間偏上會出現PAUSE標題
- 倒數：在READY情況下，會在下一幀直接進行倒數，位於遊戲正中間偏上會出現3、2、1標題;在PAUSE情況下，會在玩家按下button後進入倒數。
- WIN：在遊戲結束後，會判斷玩家的分數，在勝方出現WIN標題

這個部分所有的標題都是由程式繪製出來的，主要是一個很花時間要去計算像素的工作，而控制流程的部分我是將這些情況作編碼，利用FSM的概念，去實行狀態轉移，並透過和幀數接近的更新頻率去作更新的。

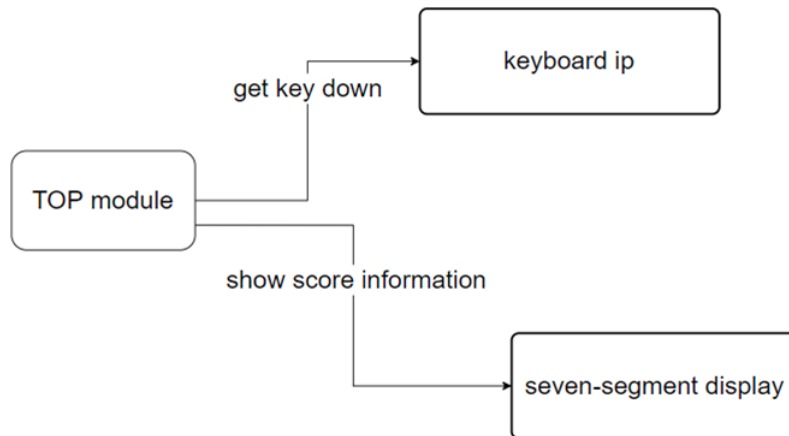
- 繪圖技巧的分析

可以先預設自己呈現出來的畫面會是甚麼樣子，並且透過圖層的概念去切分自己的畫面，一方面是會比較好處理設計，一方面當切分明確時，不太會有自己的圖片被吃掉的bug出現。

Player

- Player design
- Player FPGA I/O

Player design



client架構相對簡單，主要是接收game server的分數資訊和傳輸action回去，top module會接收keyboard資訊然後做decode，並且會把接收到的score傳送給七段顯示器顯示。

- 遇到的問題

- Keyboard問題：

如果直接用課堂上的keyboard的範例去實作會遇到連按的操作失效，要特別對按下去和抬起來做不同操作。除此之外也會有多個操作切換上的問題，因為每次切換時原本按著的鍵並不會再次被檢查到。但實際上總結最大的問題還是受限於傳輸的頻寬，使得我們一次只能傳送一個signal。

- 解決方法：更改keyboard的設計



設置一個counter每120HZ會將action做一次採樣和輸出。

採樣會分採第一個板子和第二個板子，然後根據鍵盤是按下去或沒按來決定是否要輸出操作。

比如如果一次按w和下鍵，那action會1和3分別傳送出去，每個訊號維持1/120秒。如上圖就是只在紅色段檢查板子1，黑色檢查板子2，每個線距離1/120。

Player FPGA I/O

- Input signal
 - score [4:0](P18, N17, M18, K17, B16, A15, A17, C15, C16) 接收game server的score訊號
 - clk
 - reset(U18)
 - PS2_CLK(C17)
 - PS2_DATA(B17)
- Output signal
 - seven[7:0] 七段顯示器訊號
 - an[3:0]
 - player_M[2:0](A14, A16, B15) play的action傳送

Experient result

在這次的專題中我們可以分兩個部分來展示我們的結果，第一部分是AI模型，第二部分是FPGA。

在我們設計的遊戲中隨機移動的策略大約可以獲得-800~-1440的reward，大約一場會漏掉250~300顆球，而我們的AI可以獲得2500~2600的reward一場平均失球18顆，而跟人類比的話大約可以做到一場多100~200的比分，在AI上我們調整過的PPO跟通用rllib的PPO模型比進步了22%的分數，而在我們調整過state後分數提升了接近200%的分數。

而FPGA部分我們除了做出了一般pong game所具備的功能以外還額外擴展了更多的板子和球，除此之外我們還解決了網路上大部分實作在球高速移動時會遇到的穿牆問題，不管是x軸或是y軸的位移我們都有特別做處理。

另外玩家端的地方，我們也在遊戲每偵只能進行一項操作的限制下實現了能讓使用者有同時操作板子的感覺。

Conclusion

我們希望我們的專案是真的能解決一些問題，而不是硬是把一些不需要AI的地方套入AI，所以才會選擇這個題目，將原本pong game的問題擴展並想辦法訓練出AI能打敗玩家，雖然過程中沒有任何可參考的資訊，但透過網路上的相關論文和經驗給予我們前進的方向，最後雖然沒有訓練出可以完美利用我們遊戲特性的AI，但要打敗人類是完全沒有問題的。而在FPGA上面我們從課堂中學到了基本的語法和設計方法，但真正應用在實際問題時仍然會遇到一些我們之前從未遇過的狀況需要去設計合適的架構來處理，其中最大的問題是我們的原型是使用python去設計的，並沒有思考到verilog的特性和限制，因此當我們要用verilog去完成一樣功能的作品時就會遇到許多障礙。但我們最後都一一克服，並且還解決了原本在PYTHON上面未被處理到的問題。

