

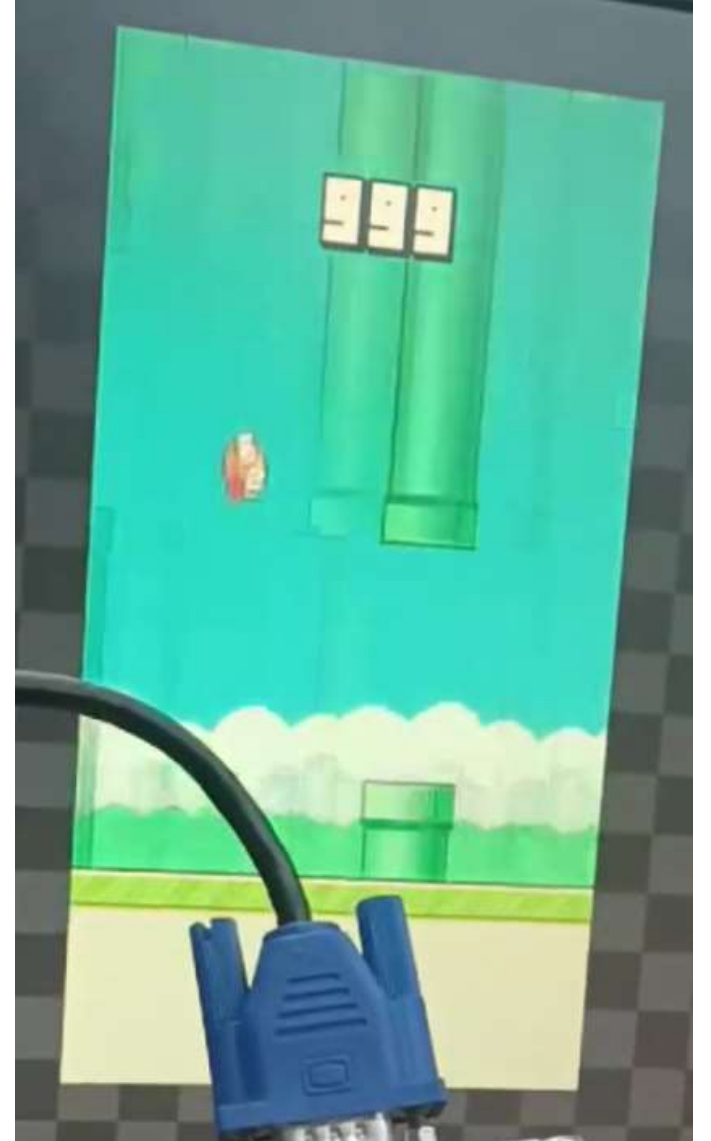
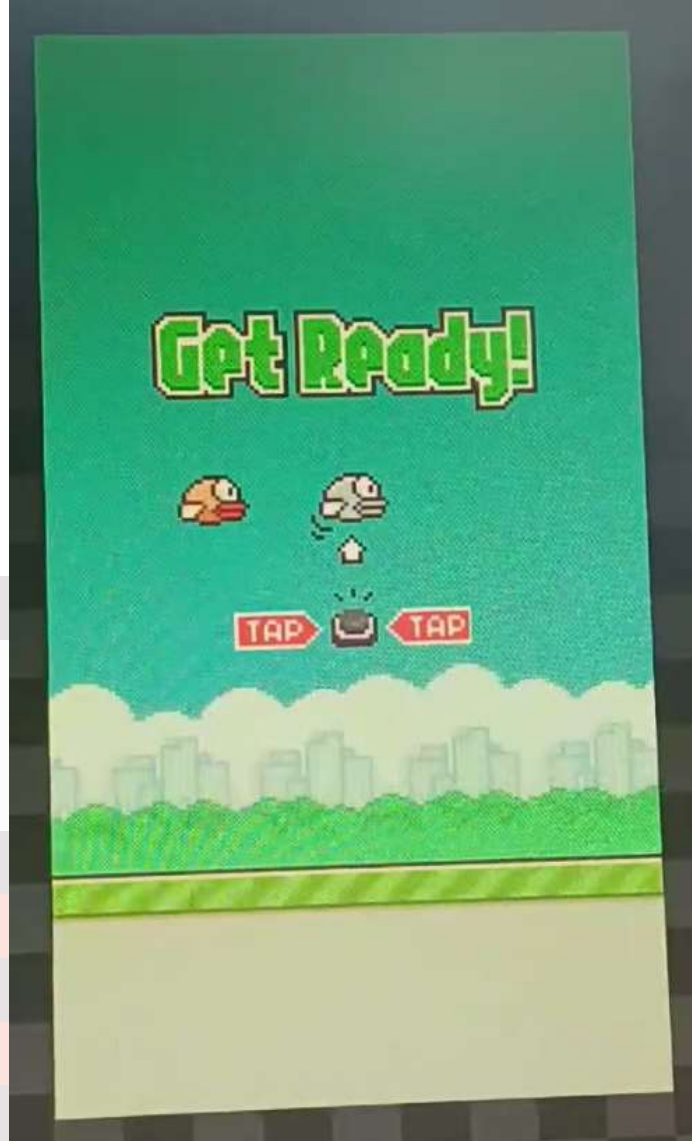
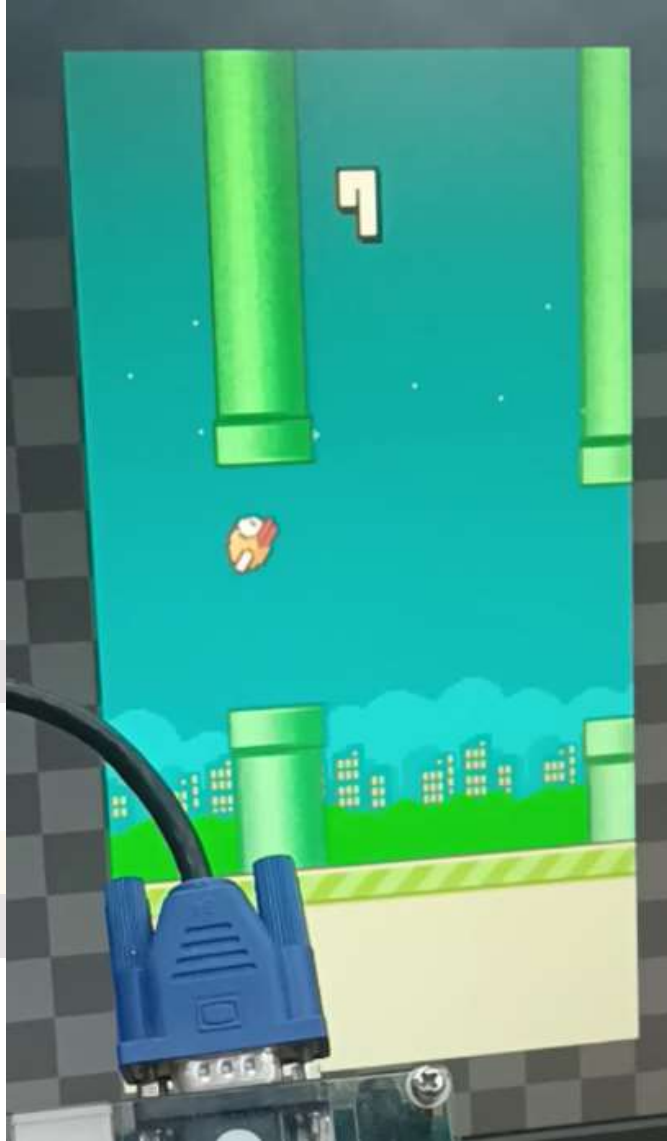
A pixelated bird character, likely a Flappy Bird, is positioned on the left side of the slide. It has a yellow body, grey wings, and a grey beak. The bird is facing right, and its wings are spread as if it is about to flap. The background is white.

用 FPGA 做 Flappy Bird !

设计思路分享答辩

2024/12/29





项目分工说明

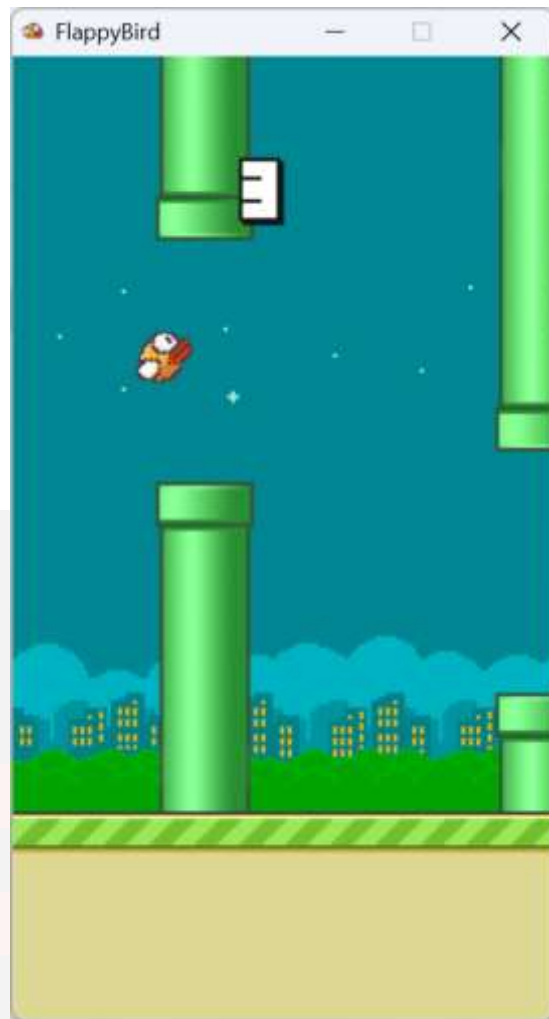
@rubatotree

- 负责整个游戏框架的规划设计；
- 编写绘制部分的代码。

@tinatanhy

- 负责一部分逻辑模块的具体实现；
- 编写与板子、外设交互的代码，如 DST、按钮、数码管等。

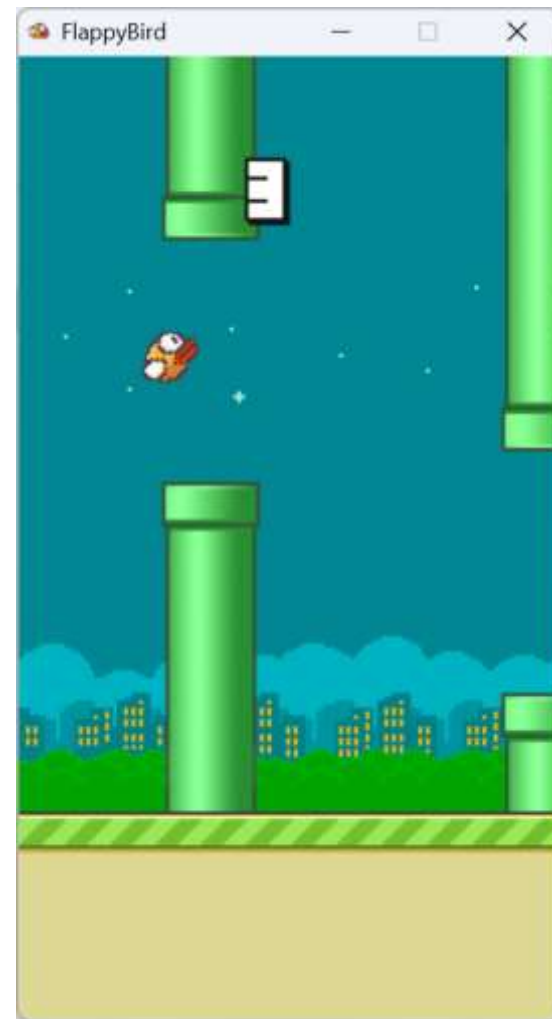
1 | 从 CPU 上的复刻版到 FPGA……



1 | 从 CPU 上的复刻版到 FPGA……

- 硬件编程思维

- 一切“并行”计算出结果
- 没有“函数调用”，没有“面向对象”
- 一切资源都是静态的



1 | 从 CPU 上的复刻版到 FPGA……

- 硬件编程思维

- 一切“并行”计算出结果
- 没有“函数调用”，没有“面向对象”
- 一切资源都是静态的

- ROM 空间有限

为什么塞一个小视频进去要压缩那么多！



1 | 从 CPU 上的复刻版到 FPGA……

- 硬件编程思维

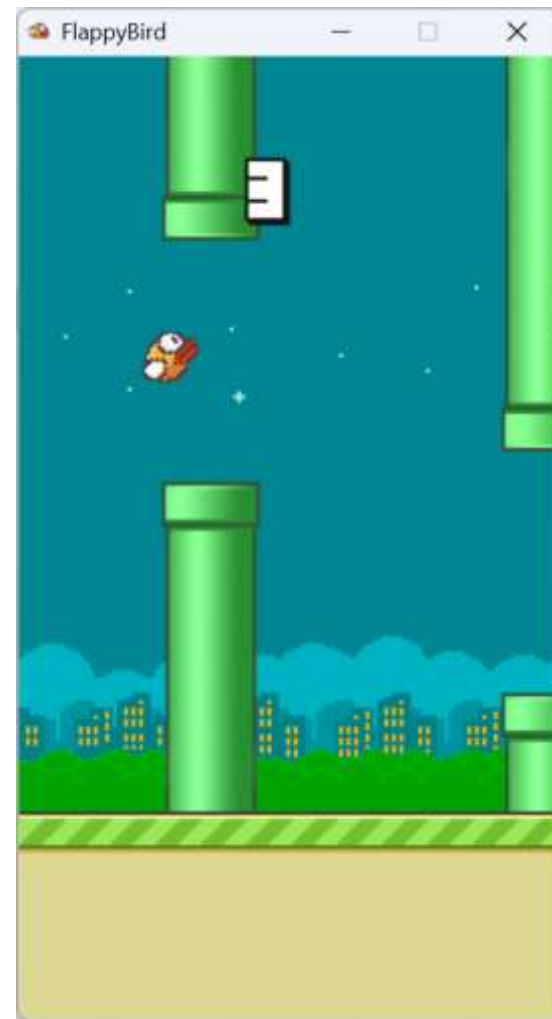
- 一切“并行”计算出结果
- 没有“函数调用”，没有“面向对象”
- 一切资源都是静态的

- ROM 空间有限

为什么塞一个小视频进去要压缩那么多！

- 综合实现时间长

每次调试都要等十分钟……



1 | 从 CPU 上的复刻版到 FPGA……

- 浮点数难题

对于 FPGA 而言，浮点数还是太昂贵了……



1 | 从 CPU 上的复刻版到 FPGA……

- 浮点数难题

对于 FPGA 而言，浮点数还是太昂贵了……

解决方案 1：不让它“浮”起来！

用 32 位**整数**存储小数乘上 2^{16} 的值，加减乘运算依然很便利。

1 | 从 CPU 上的复刻版到 FPGA……

- 浮点数难题

对于 FPGA 而言，浮点数还是太昂贵了……

解决方案 1：不让它“浮”起来！

用 32 位**整数**存储小数乘上 2^{16} 的值，加减乘运算依然很便利。

解决方案 2：三角函数难算？以存代算！

1 | 从 CPU 上的复刻版到 FPGA……

- 浮点数难题

对于 FPGA 而言，浮点数还是太昂贵了……

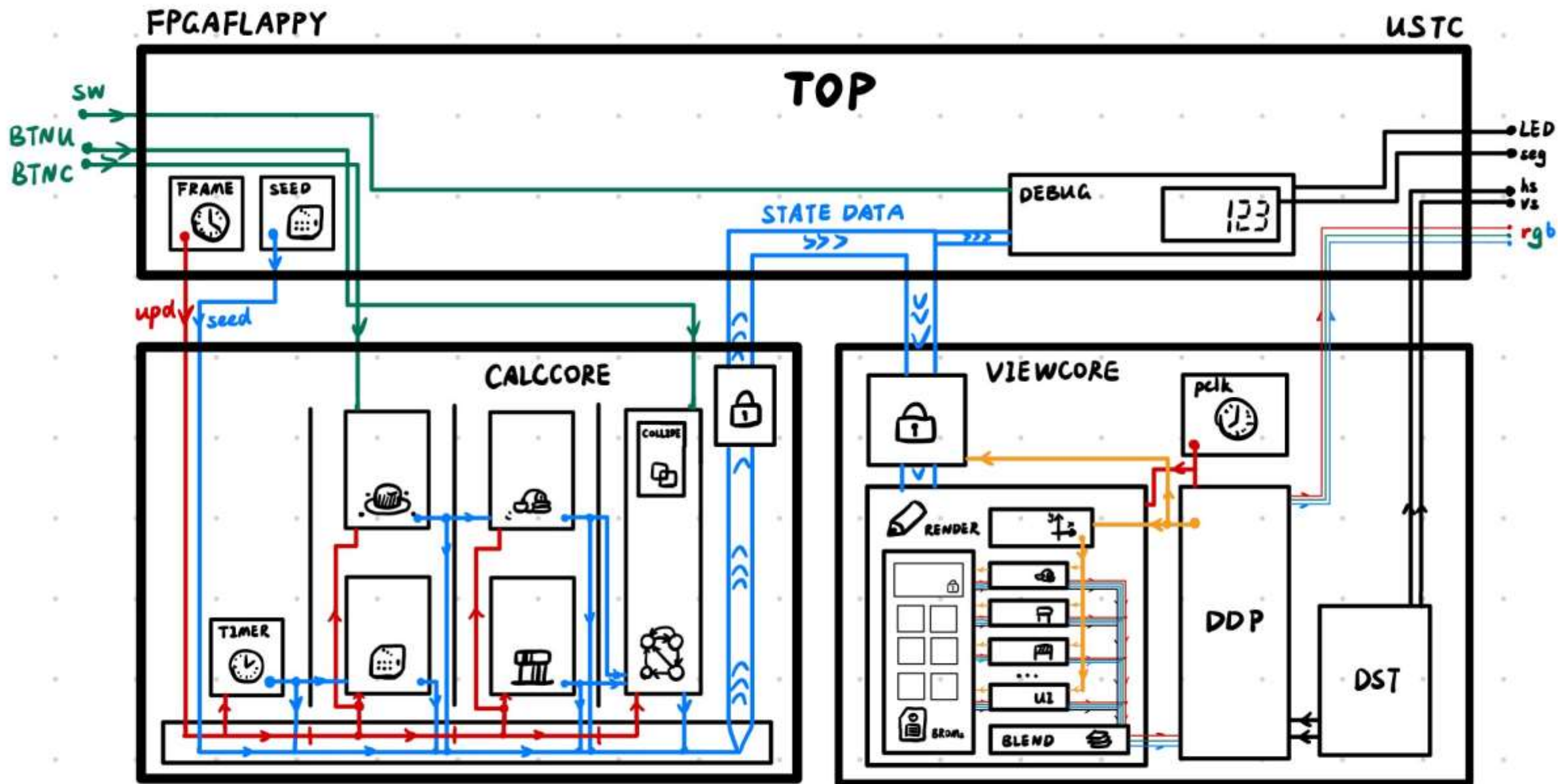
解决方案 1：不让它“浮”起来！

用 32 位**整数**存储小数乘上 2^{16} 的值，加减乘运算依然很便利。

解决方案 2：三角函数难算？以存代算！

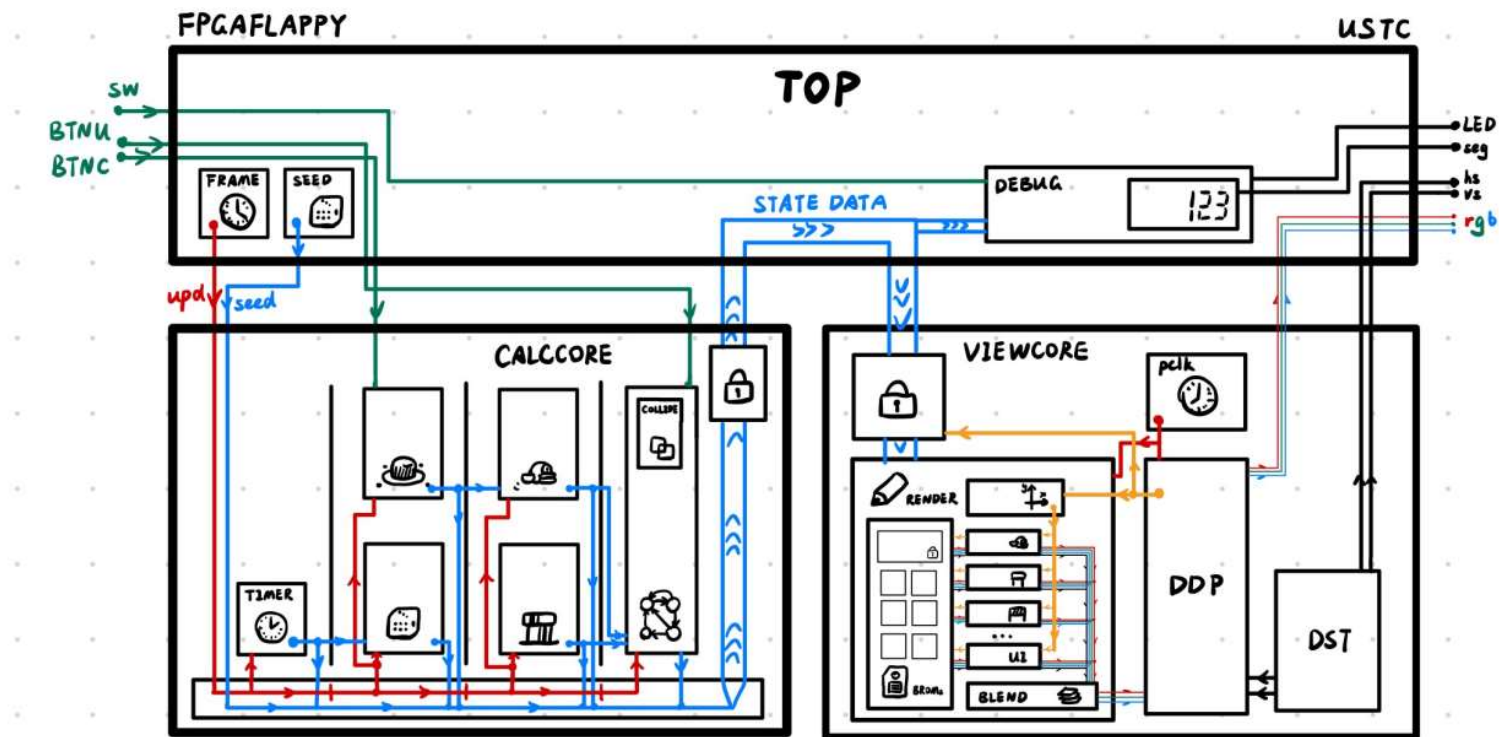


2 | 游戏电路框架结构



2 | 游戏电路框架结构

- 以帧为单位刷新
- 逻辑与绘制分离
- 分时调度计算模块

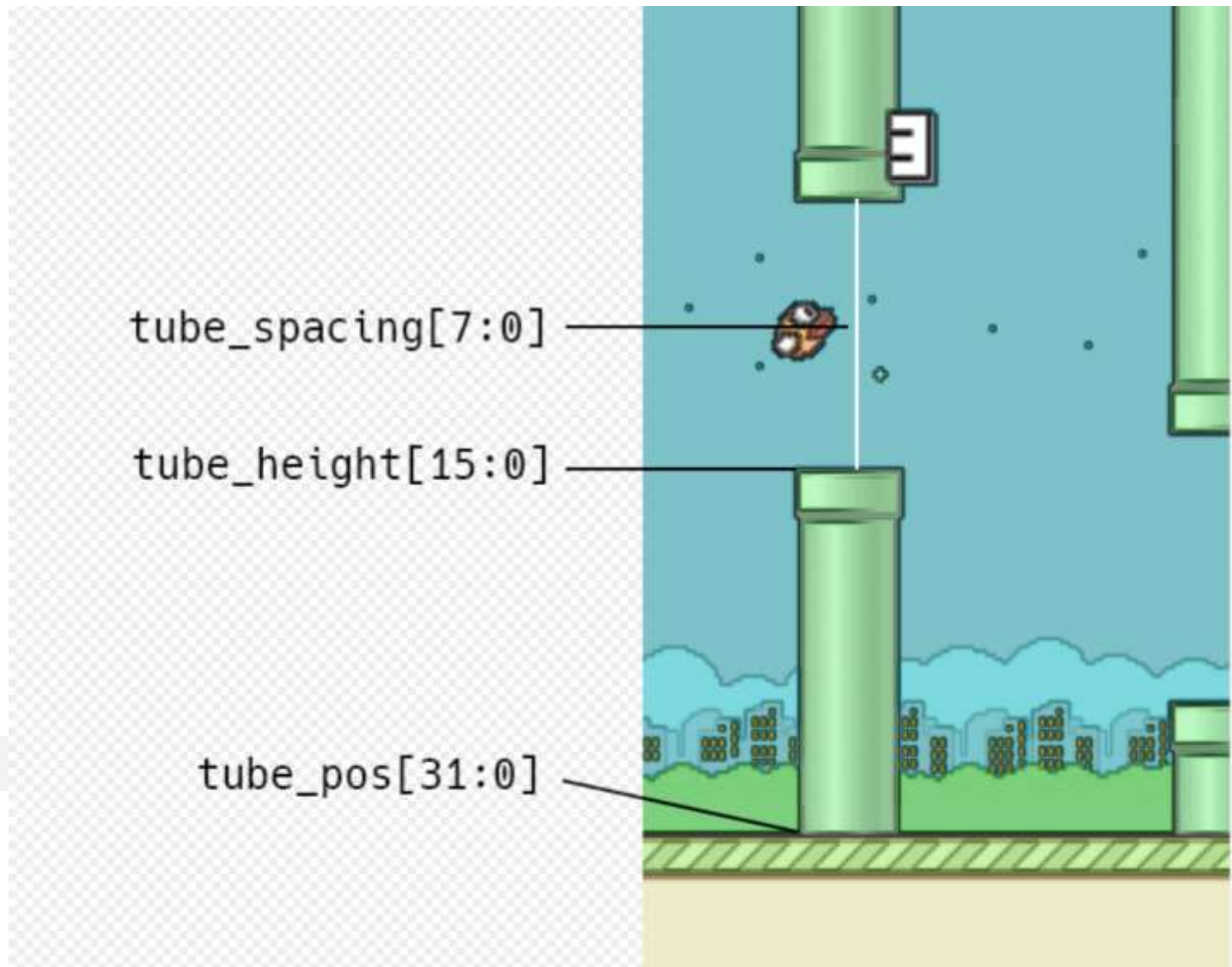


3 | 管道的实现 *逻辑模块的样例



3 | 管道的实现

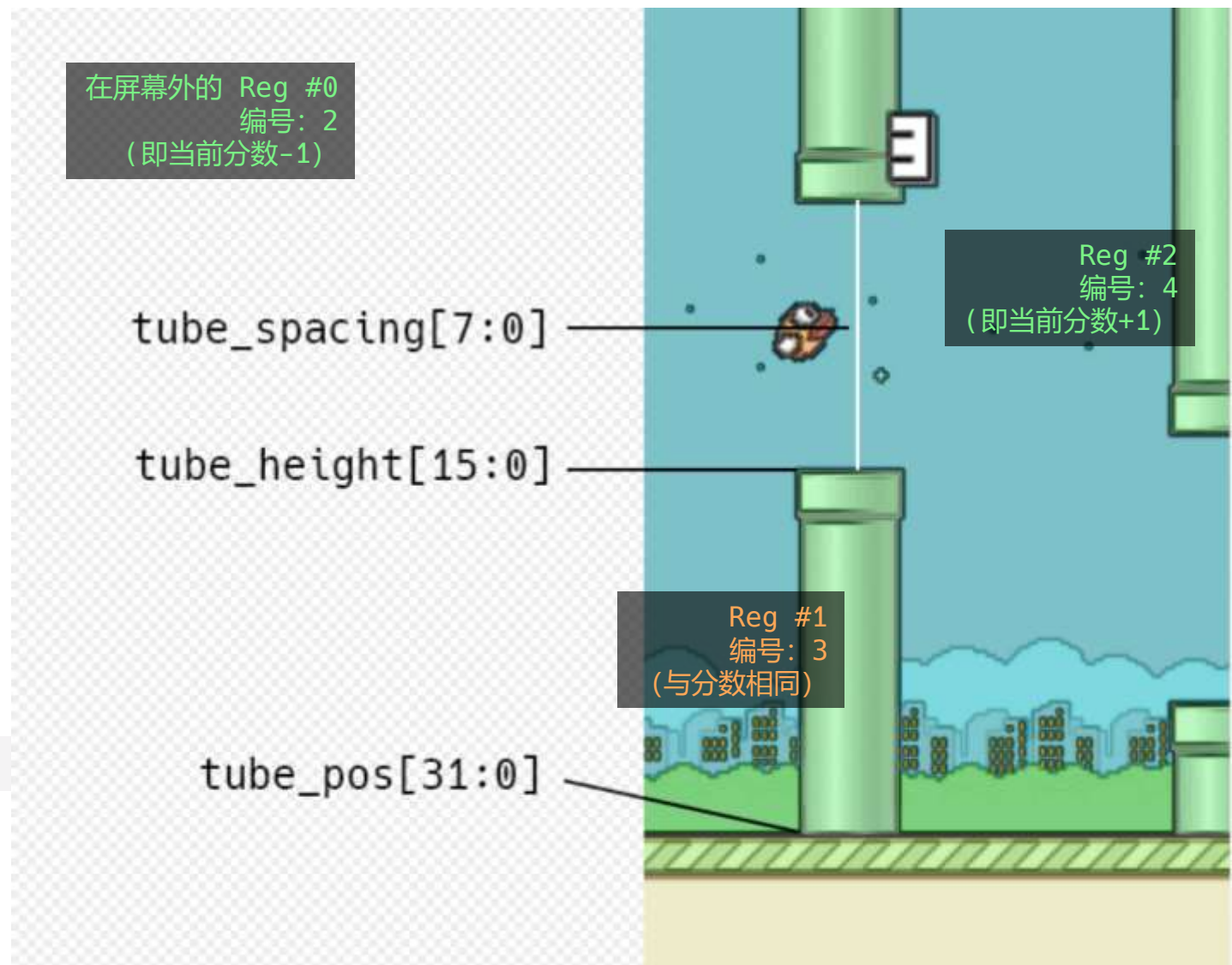
三个参数可以完全决定管道的碰撞与绘制。



3 | 管道的实现

三个参数可以完全决定管道的碰撞与绘制。

用当前**分数**可以确定可能显示在屏幕上的管道**编号**；

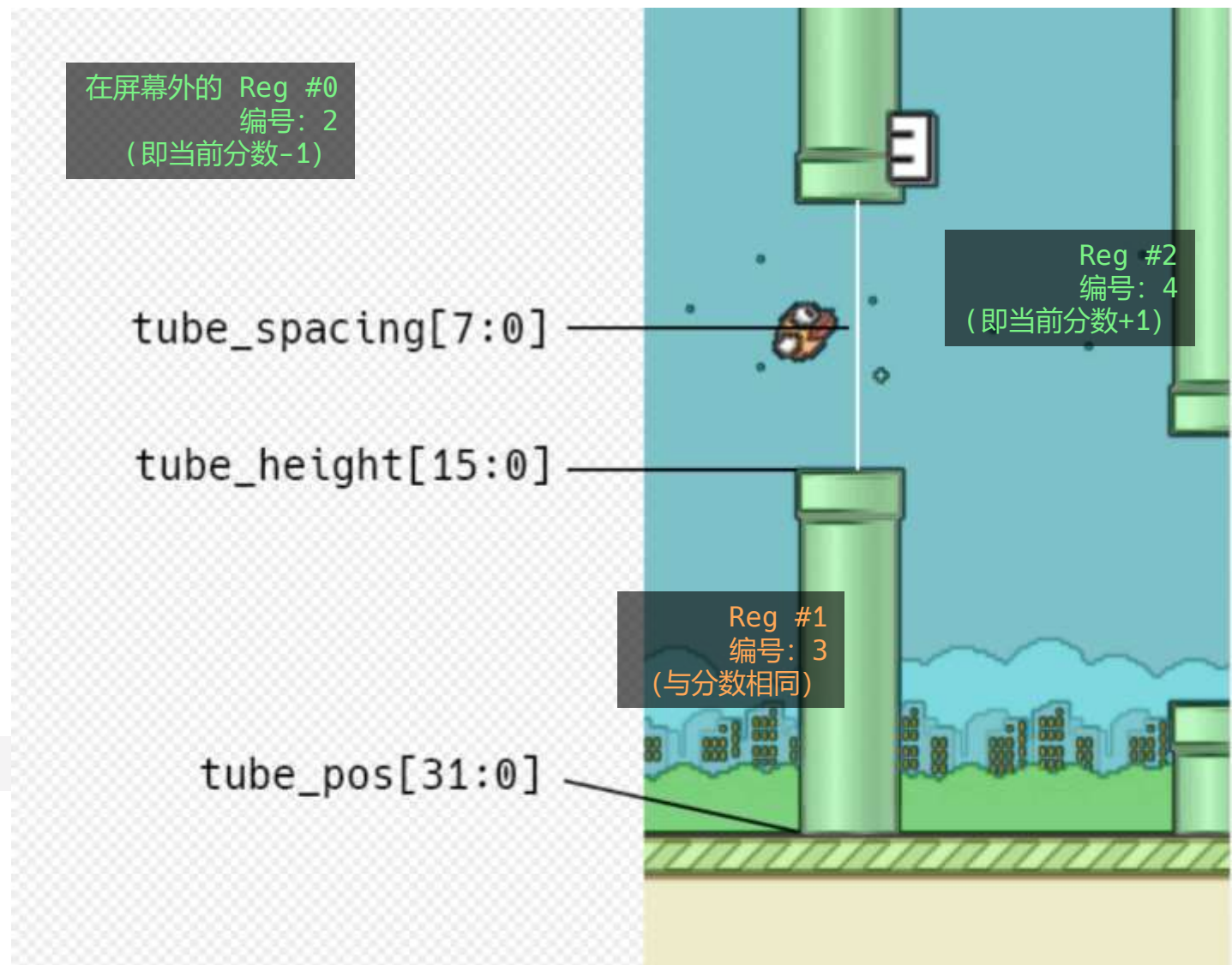


3 | 管道的实现

三个参数可以完全决定管道的碰撞与绘制。

用当前**分数**可以确定可能显示在屏幕上的管道**编号**；

spacing 是常数，pos 是关于分数的简单线性映射；



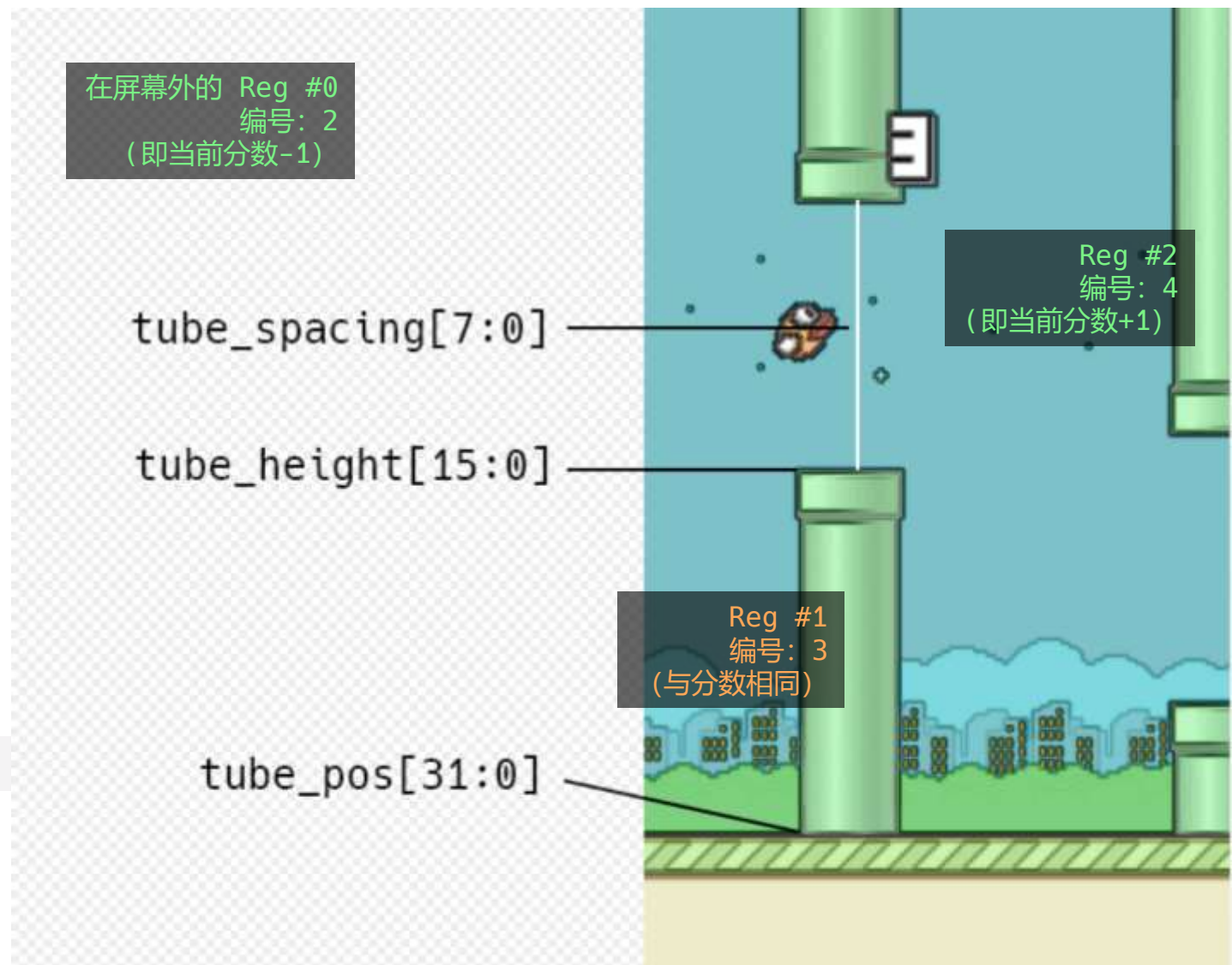
3 | 管道的实现

三个参数可以完全决定管道的碰撞与绘制。

用当前**分数**可以确定可能显示在屏幕上的管道**编号**；

spacing 是常数，pos 是关于分数的简单线性映射；

随机的 height 则由管道编号和世界种子一同送进一个**哈希元件**（时序）得到，避免了存储。



4 | 分数的绘制 *绘制模块的样例

传统游戏开发中，一般会用一张画布存储显示在屏幕上的内容，每个对象有相应的代码将自己绘制在画布上，最后就能呈现出分层的效果。

不过，我们的游戏框架设计中是**不含“画布”**的。

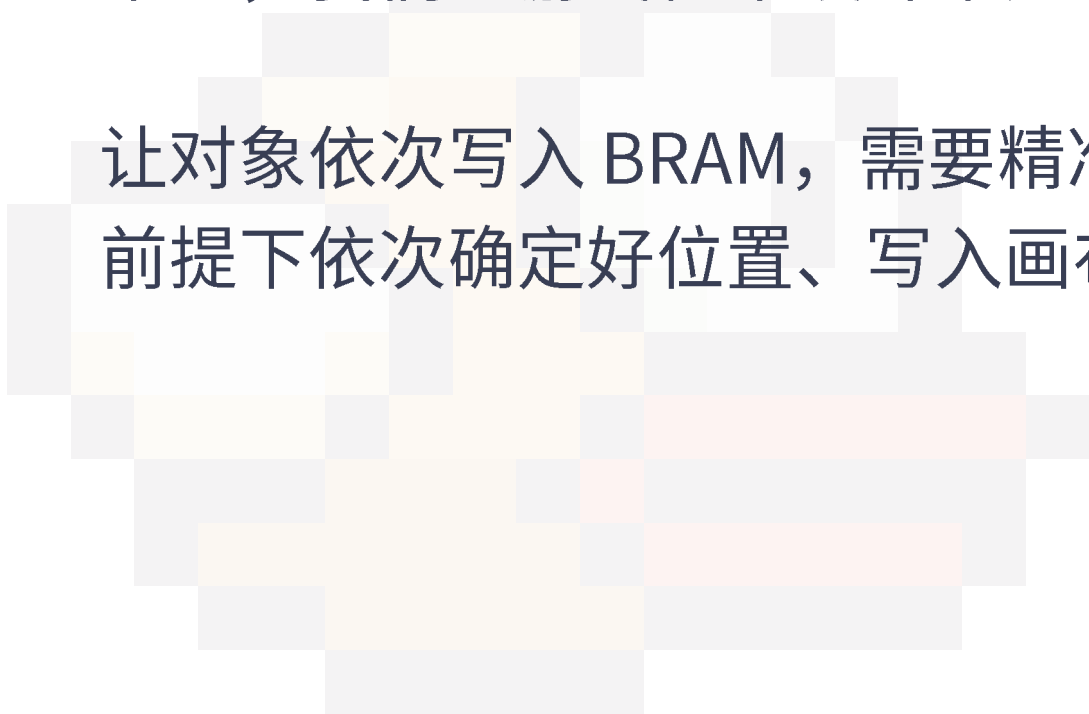


4 | 分数的绘制 *绘制模块的样例

传统游戏开发中，一般会用一张画布存储显示在屏幕上的内容，每个对象有相应的代码将自己绘制在画布上，最后就能呈现出分层的效果。

不过，我们的游戏框架设计中是**不含“画布”**的。

让对象依次写入 BRAM，需要精准地控制时序，在 BROM 有访问延迟的前提下依次确定好位置、写入画布……



4 | 分数的绘制 *绘制模块的样例

传统游戏开发中，一般会用一张画布存储显示在屏幕上的内容，每个对象有相应的代码将自己绘制在画布上，最后就能呈现出分层的效果。

不过，我们的游戏框架设计中是**不含“画布”**的。

让对象依次写入 BRAM，需要精准地控制时序，在 BROM 有访问延迟的前提下依次确定好位置、写入画布……

所以让我们偷个懒，**让像素点自己思考**自己应该是什么颜色。

4 | 分数的绘制 *绘制模块的样例

传统游戏开发中，一般会用一张画布存储显示在屏幕上的内容，每个对象有相应的代码将自己绘制在画布上，最后就能呈现出分层的效果。

不过，我们的游戏框架设计中是**不含“画布”**的。

让对象依次写入 BRAM，需要精准地控制时序，在 BROM 有访问延迟的前提下依次确定好位置、写入画布……

所以让我们偷个懒，**让像素点自己思考**自己应该是什么颜色。

这是类似在 GPU 上编写 Shader 的思路。以分数数字的绘制为例——

4 | 分数的绘制

每个像素点都会经过一遍子渲染器模块。



4 | 分数的绘制

每个像素点都会经过一遍渲染器模块。

输入: `screen_x[15:0]`, `screen_y[15:0]`

// 这是像素点在游戏内屏上的坐标。

`score_decimal[11:0]`

// 8421BCD 码表示的分数值。



4 | 分数的绘制

每个像素点都会经过一遍子渲染器模块。

输入: `screen_x[15:0]`, `screen_y[15:0]`

// 这是像素点在游戏内屏上的坐标。

`score_decimal[11:0]`

// 8421BCD 码表示的分数值。

输出: `mask` // 决定该像素点上该不该绘制分数;

`addr[11:0]` // 在 BROM 中访存的地址。

// 只有 `mask` 高电平时才保证 `addr` 合法。



4 | 分数的绘制



每一个数字的图像大小是 12×18 ，放大一倍后 (24×36) 绘制在屏幕上。

4 | 分数的绘制



每一个数字的图像大小是 12×18 ，放大一倍后 (24×36) 绘制在屏幕上。

因此，对于访问到的每个像素点，可以根据它们在屏幕上的位置：

1. 先确定是否在某个数字的图像上；
2. 再确定在数字图像上的具体坐标，是哪个数字，从而确定访存地址。

4 | 分数的绘制

多位数的情况，只需要分类讨论，最后用一个数据选择器选择分支。



4 | 分数的绘制

多位数的情况，只需要分类讨论，最后用一个数据选择器选择分支。

我们大胆假设没有玩家能玩过 999 分！



4 | 分数的绘制

多位数的情况，只需要分类讨论，最后用一个数据选择器选择分支。

~~我们大胆假设没有玩家能玩过 999 分！~~

所以我们只要提前定义好六种数字即可：

1. 当分数为一位数时的个位；
2. 当分数为两位数时的个位、十位；
3. 当分数为三位数时的个位、十位、百位。

4 | 分数的绘制

先确定每种数字的显示位置……

```
localparam num_w = 12, num_h = 18, img_w = 120,  
             score_y = 54,  
             score_x11 = 144 - 12,  
             score_x21 = 144 - 24,  
             score_x22 = 144 - 0,  
             score_x31 = 144 - 12 - 24,  
             score_x32 = 144 - 12,  
             score_x33 = 144 - 12 + 24;
```

4 | 分数的绘制

再确定当前显示的是多少位数……

```
wire score_view_1 = ~(|score_decimal[11:8]) & ~(|score_decimal[7:4]),  
      score_view_2 = ~(|score_decimal[11:8]) & (|score_decimal[7:4]),  
      score_view_3 = (|score_decimal[11:8]);
```



4 | 分数的绘制

再确定当前显示的是多少位数……

```
wire score_view_1 = ~(|score_decimal[11:8]) & ~(|score_decimal[7:4]),  
    score_view_2 = ~(|score_decimal[11:8]) & (|score_decimal[7:4]),  
    score_view_3 = (|score_decimal[11:8]);
```

把遮罩用不等式表示出来……

和对应的十进制位数求与，最后求并，就能得到所要的遮罩了。

```
wire score_mask_y = ($signed(screen_y) >= score_y) && ($signed(screen_y) < score_y + 36);  
wire score_mask_x11 = game_status[1] && score_view_1 && ($signed(screen_x) >= $signed(score_x11)) && ($signed(screen_x - LATENCY) < $signed(score_x11 + 24));  
wire score_mask_x21 = game_status[1] && score_view_2 && ($signed(screen_x) >= $signed(score_x21)) && ($signed(screen_x) < $signed(score_x21 + 24));  
wire score_mask_x22 = game_status[1] && score_view_2 && ($signed(screen_x) >= $signed(score_x22)) && ($signed(screen_x - LATENCY) < $signed(score_x22 + 24));  
wire score_mask_x31 = game_status[1] && score_view_3 && ($signed(screen_x) >= $signed(score_x31)) && ($signed(screen_x) < $signed(score_x31 + 24));  
wire score_mask_x32 = game_status[1] && score_view_3 && ($signed(screen_x) >= $signed(score_x32)) && ($signed(screen_x) < $signed(score_x32 + 24));  
wire score_mask_x33 = game_status[1] && score_view_3 && ($signed(screen_x) >= $signed(score_x33)) && ($signed(screen_x - LATENCY) < $signed(score_x33 + 24));  
wire score_place_mask = score_mask_y & (score_mask_x11 || score_mask_x21 || score_mask_x22 || score_mask_x31 || score_mask_x32 || score_mask_x33);
```

4 | 分数的绘制

计算像素点在每种数字下的图像坐标……

别忘记只有 mask 有效时才要求合法。其他时候无论坐标飞到哪里都没关系！

```
wire [11:0] score_posx_11 = (screen_x - score_x11) >> 1;
wire [11:0] score_posx_21 = (screen_x - score_x21) >> 1;
wire [11:0] score_posx_22 = (screen_x - score_x22) >> 1;
wire [11:0] score_posx_31 = (screen_x - score_x31) >> 1;
wire [11:0] score_posx_32 = (screen_x - score_x32) >> 1;
wire [11:0] score_posx_33 = (screen_x - score_x33) >> 1;
wire [11:0] score_posy    = (screen_y - score_y) >> 1;
```


4 | 分数的绘制

计算每种情形下的访存地址，并用数据选择器选出最后的地址……

然后就能确定像素的颜色啦！如果当前位置有分数，就能绘制出来了。

```
wire [11:0] score_addr_11 = score_posx_11 + img_w * score_posy + score_decimal[3:0] * num_w;
wire [11:0] score_addr_21 = score_posx_21 + img_w * score_posy + score_decimal[7:4] * num_w;
wire [11:0] score_addr_22 = score_posx_22 + img_w * score_posy + score_decimal[3:0] * num_w;
wire [11:0] score_addr_31 = score_posx_31 + img_w * score_posy + score_decimal[11:8] * num_w;
wire [11:0] score_addr_32 = score_posx_32 + img_w * score_posy + score_decimal[7:4] * num_w;
wire [11:0] score_addr_33 = score_posx_33 + img_w * score_posy + score_decimal[3:0] * num_w;

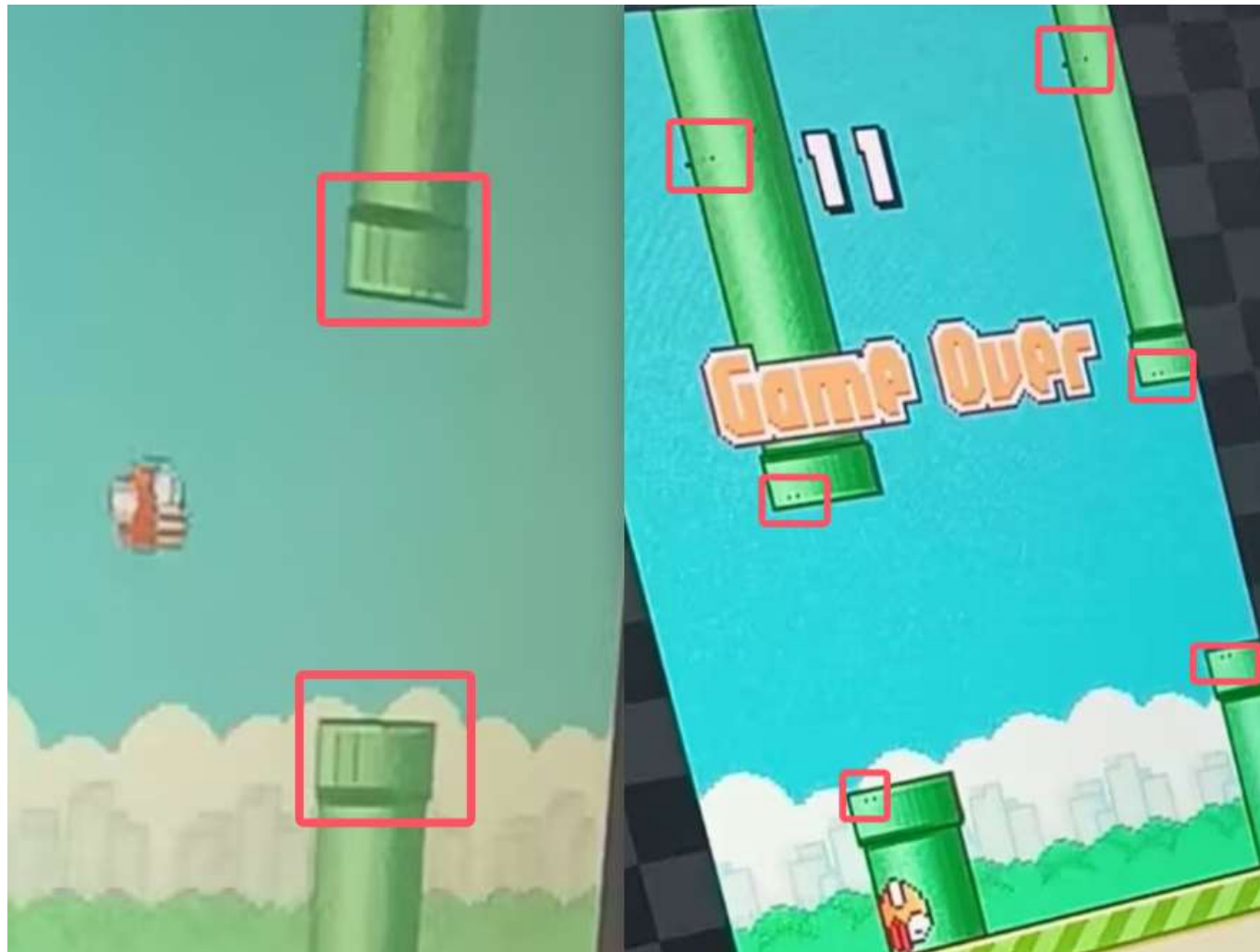
assign addr = score_place_mask ? ({12{score_mask_y}} & (
    {12{score_mask_x11}} & score_addr_11
| {12{score_mask_x21}} & score_addr_21
| {12{score_mask_x22}} & score_addr_22
| {12{score_mask_x31}} & score_addr_31
| {12{score_mask_x32}} & score_addr_32
| {12{score_mask_x33}} & score_addr_33)) : 12'd2161;
```

5 | 彩蛋：宇宙射线 Bug

开发中期出现了一个超玄学bug：
管道和地面的 BROM 会**完全随机**
地出现颜色不正常的像素点。

CPU Reset 不起作用，只有**重新
烧板**才能重置，然而过一段时间
它们又会长出来。

助教也不知道这是为什么。我们
一度认为这是宇宙射线引起的。

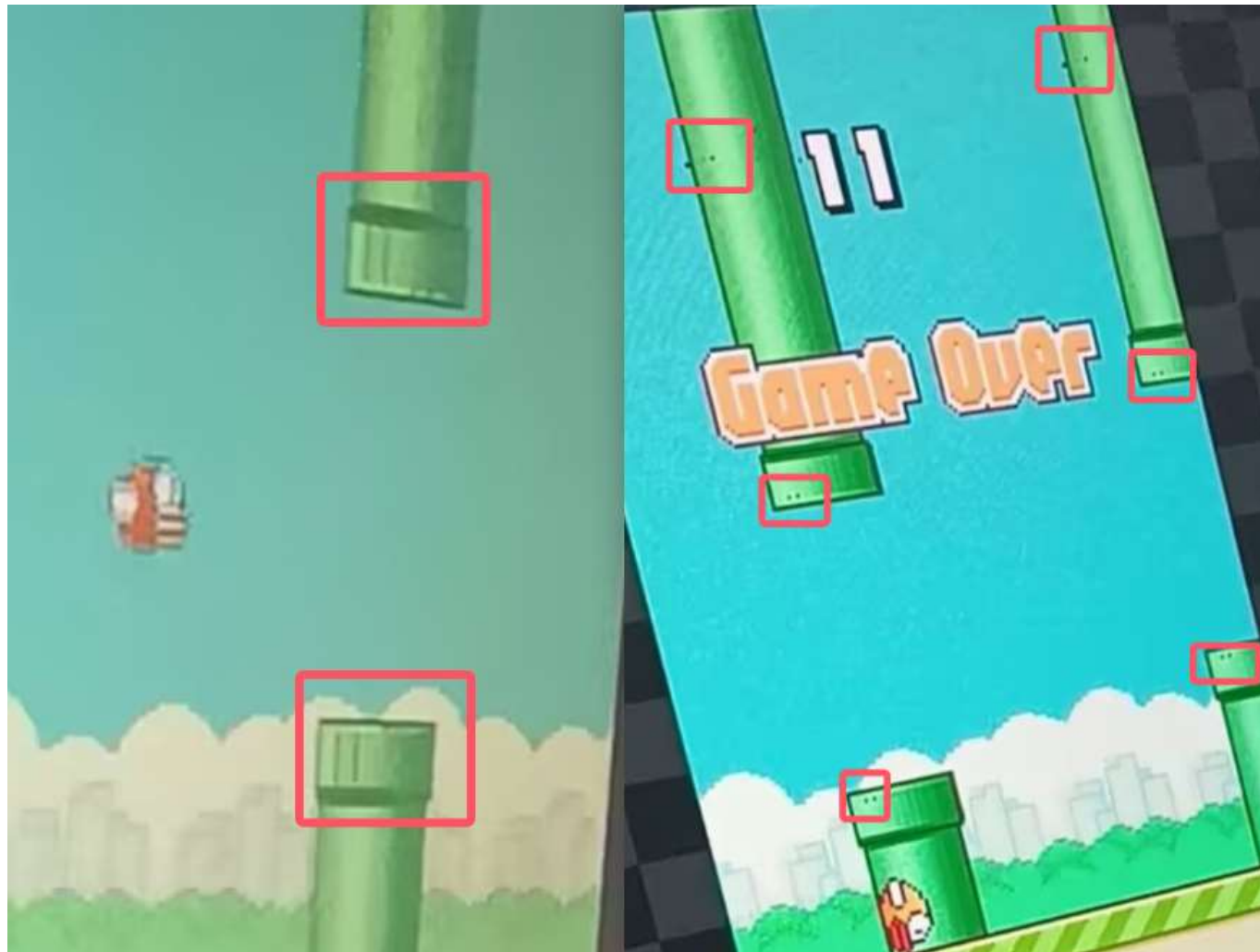


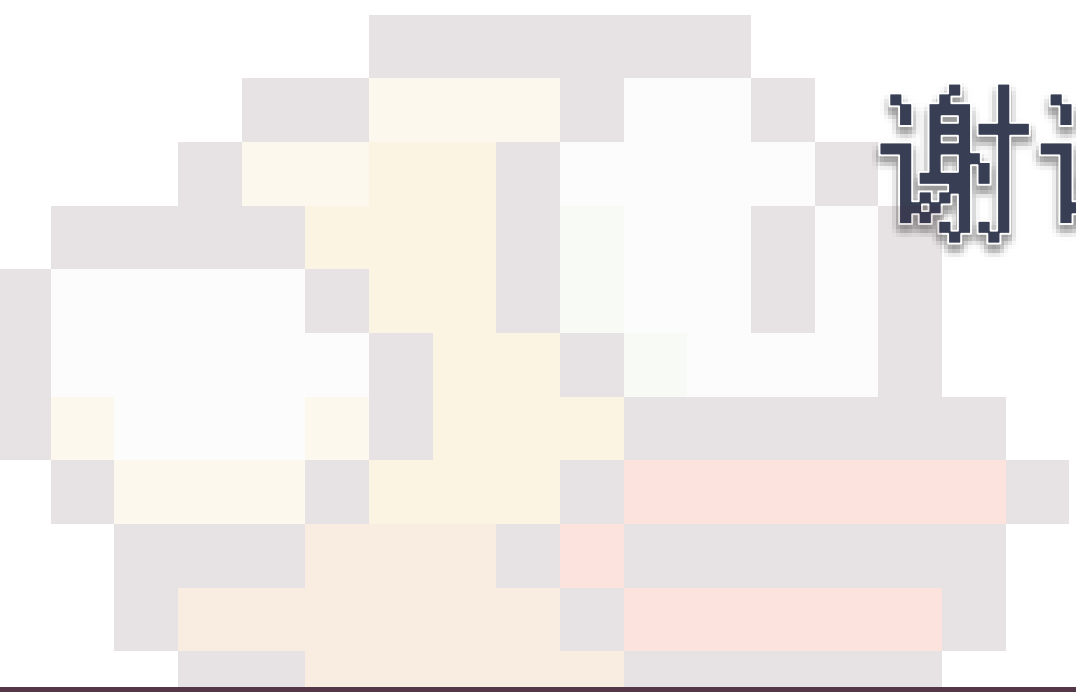
5 | 彩蛋：宇宙射线 Bug

最终发现这是由于在 BROM 的访存延迟期间改变了访问的地址引起的。

最后通过在每个 pclk 的上升沿锁存访存地址解决了。

硬件 Debug 好心累啊啊啊



A pixelated illustration of a yellow bird with an orange beak and feet, positioned on the left side of the slide. The bird is facing right. Behind it are several grey pipes of varying heights. The entire scene is set against a white background with a green and yellow striped ground at the bottom.

谢谢大家！

本项目已经开源在 Github: <https://github.com/tinatanhy/Flappybird>
对项目更详细的解释见 Blog: <http://blog.umb-coffee.icu/fpga-flappy>