PART 10: “Arrays”

Introduction

In previous episodes we generally worked with one object at a time: one ball, one grain of sand, and so on. In practice, we will generally want several of these. So, to store the locations of two balls, we could type:

int ballx1 = …, bally1 = …;
int ballx2 = …, bally2 = …;

Obviously, this quickly gets annoying. This is where *arrays* come in handy:

int ballx[2], bally[2];

Now, variable ballx doesn’t contain a *single* integer, but *two*; and even better: we can replace ‘2’ by pretty much any number we want.

Let’s give this a try. Copy the following code into a fresh template:

```
#include "precomp.h"

int x[4096], y[4096];

void Game::Init()
{
for( int i = 0; i < 4096; i++ )
    x[i] = IRand( 800 ), y[i] = IRand( 512 );
}

void Game::Shutdown() {}

void Game::Tick( float deltaTime )
{
    screen->Clear( 0 );
    for( int i = 0; i < 4096; i++ )
        x[i] = (x[i] + 800 + (((i & 1) * 2) - 1)) % 800,
        y[i] = (y[i] + 512 + ((((i >> 2) & 1) * 2) - 1)) & 511,
        screen->GetBuffer()[x[i] + y[i] * 800] = 0xffffff;
}
```

This probably requires some explanation. ☺ First, the arrays: two arrays are allocated to store x and y coordinates. There is room for 4096 integers in each array. This is not a random choice: 4096 is 2^{12}, so actually (for a computer) a nice round number. The arrays are filled with random numbers in the `Init` function. A simple template function is used for this; check it out in `template.h`. To keep the code brief I used a trick here: a comma links the two variable assignments, which is perfectly fine in C++, and sometimes very handy, e.g. in

```
for( int i = 0, j = 0, k = 0; i < 10; i++, j++, k++ )
```

In the above code, it simply allows us to skip the curly brackets, which are not needed for a for-loop that executes a single ‘instruction’, reducing `Init` to just two lines.

The magic is happening in `Tick` however.

Each of the 4096 particles moves, but always diagonally:

- half the particles go left, half go right;
- half the particles go up, half go down.

The decision for movement direction is made based on the *index* of the particle. Specifically: 'even' particles go left, 'uneven' particles go right. This is done using *bitmasking* (which we used for colors before). 'Even' particles are numbered 0, 2, 4, …; these numbers have in common that their first bit is set to 0. For uneven particles, the first bit is 1. So, we take this bit, multiply it by 2 (to get 0 or 2), and subtract 1 (to get -1 or 1). This value is then added to the x-coordinate of the particle. To modify y, the same trick is applied, but using a different bit of the index.

Finally, we need to make sure that the particles stay on the screen. A modulo ('%') is used for this. I am never sure how '%' behaves with negative numbers, so the screen width is simply added to the current coordinate, and then the % is used. This solves off-screen problems on the left *and* the right, using a single operation.

After this, commas are used to safe on curly brackets, and a pixel is plotted by directly accessing the pixel buffer.

Smooth

This runs pretty smooth. But how smooth exactly? Try this: start FRAPS, and increase the particle count until the frame rate drops below 60.

There's a slight problem with that request: the number 4096 appears four times in the code. To ease the pain, add a #define at the top of the code:

```
#define PARTICLES 4096
```

Now, instead of typing `4096` you can type `PARTICLES`, and changing the count is much easier. The `#define` is a *macro*: C++ will replace each occurrence with the specified value. Just imagine the mayhem when you start using this to obfuscate code…

Smoother

Let's try another program:

```
#include "precomp.h"

float x = 400, y = 256;

void Game::Init() {}
void Game::Shutdown() {}

void Game::Tick( float deltaTime )
{
	screen->Clear( 0 );
```

```
        screen->Line( mousex, 0, mousex, 511, 0xff0000 );
        screen->Line( 0, mousey, 799, mousey, 0xff0000 );
}
```

This also requires some changes in game.h:

- Replace the MouseMove function by:
  `void MouseMove( int x, int y ) { mousex = x, mousey = y; }`
- Add two variables at the end of the class definition:
  `int mousex, mousey;`

When you start the program you will notice there is a bug in the template: the lines do not actually follow your mouse. You can fix this in `template.cpp`, line 312: replace `xrel` by `x`, and `yrel` by `y`. Now you have some crosshairs that follow the mouse.

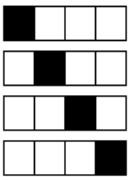
Let's have some fun with the mouse. Add these lines at the end of the Tick function:

```
float dx = x - mousex, dy = y - mousey;
float dist = sqrtf( dx * dx + dy * dy );
if (dist < 50) x += dx / dist, y += dy / dist;
screen->Plot( (int)x, (int)y, 0xffffff );
```

Now we have a pesky white dot that is afraid of the mouse.

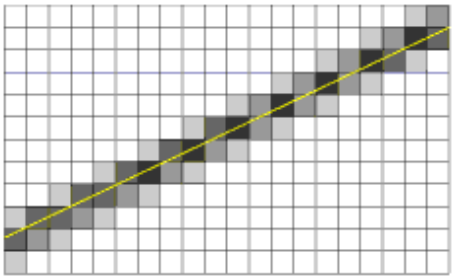
Silky Smooth

The white dot has floating point coordinates, but when we plot it, it is plotted to an integer position. That's only logical: it cannot move by less than a pixel at a time. *Or can it?* - Let's zoom in on a pixel moving from the left to the right:



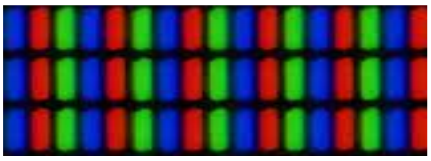
..And now, let's make it move smoother:

The two images may not be entirely convincing, but what we apply here is the fundamental principle of anti-aliasing: when a 1x1 pixel is located at a coordinate that is not an integer, it essentially 'overlaps' multiple pixels. It thus contributes to the color of multiple pixels, which our eyes in turn interpret as sub-pixel movement. Here's an anti-aliased line to make the effect more visible:

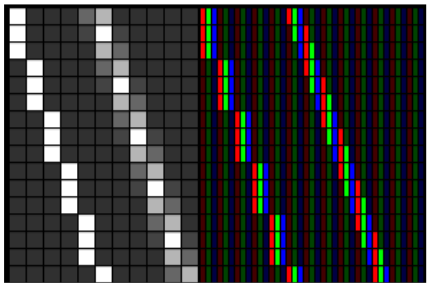


Smootherder

A final trick to make things move even smoother requires a closer look at computer screens:



As you know, a white pixel is obtained by setting red, green and blue to their maximum values. But, what if we set green, blue and red instead? On a screen that uses the three color components in the layout shown above, we can move a white pixel by one third of a pixel… This is the core idea behind the ClearType technology.



Assignment

PART 1 – REGULAR:

This episode started with a discussion of arrays. Make the mouse-evading pixel into a pixel plague: add a large amount of pixels, all exhibiting the same annoying behavior.

PART 2 – HARD:

You probably noticed that there is no code in this episode to make the pixel move at the sub-pixel level. That is because this is your second assignment for today. Some hints:

- A pixel located at position x, y affects pixels (x,y), (x+1,y), (x,y+1) and (x+1,y+1).
- The brightness of each of these four pixels is proportional to the area of overlap.
- The sum of these areas is 1.

Also notice that a single pixel may be affected by multiple particles. This means that the brightness you want to write to a pixel should actually be *added* to it. You can find a useful function for this in `surface.h`: function `AddBlend` takes two Pixel colors and returns the summed color.

END OF PART 10

Next part: "Tiles"