Assumptions:

You have read Craits tutorial.

Discussion of game

The endless runner.

Discussion of Sprites

Sprites and masks

Encoding sprites

Encoding masks

Drawing a sprite

Example 1 ground, 1 dinosaur drawOverwrite showing how it erases the ground

Example 2 ground, dinosaur and mask showing how it does not erase the ground.

Making the dinosaur run.

Sprite and frames

Different size sprites.

Making the ground move.

Flintstones

Sprites

A sprite is simply an image or graphic that represents a player, an enemy or even background elements of your game and can be drawn or moved around as a single object. The Arduboy library provides a number of functions to render sprites to the screen. In this tutorial, we will concentrate on two functions drawOverwrite() and drawExternalMask().

But before we get into those, let’s recap how we define a sprite. The image below shows Steve in his upright, ready-to-run position.

The array definition for this sprite is shown below.

byte myFirstSprite[] = {

8, 8,

126, 231, 231, 129, 129, 231, 231, 126,

};

I have formatted the array to make it a little more readable. The first line contains the width and height of the array, in this case 8 pixels by 8 pixels. The remaining 8 bytes contain the pixel data for each column of the sprite and are calculated using a simple formula shown below.

Notice how I have labelled the side of the graphic with 1, 2, 4 and so on. To calculate that the first column’s value is 126, I simply added up all of the values adjacent to the pixels I want to be turned on (white). 2 + 4 + 8 + 16 + 32 + 64 = 126. The remaining columns are calculated in exactly the same way.

The sprite is drawn by simply overwriting what was already there. A bit set to 1 in the frame will set the pixel to 1 in the buffer, and a 0 in the array will set a 0 in the buffer. In the example below, the black corners of the ball are visible as the ball passes into the white area.

When rendering a sprite, bits set to 1 in the mask indicate that the pixel will be set to the value of the corresponding image bit. Bits set to 0 in the mask will be left unchanged. This can be seen clearly as the ball moves into the right hand side of the background. The top-left and bottom-right corners of the image are rendered as black as the mask is set to 1 in these areas which in turn ensures that the images pixels (both zeroes and ones) are rendered on the background.

* Dino sprite
* drawOverwrite()
* drawErase()
* drawExternalMask()
* drawPlusMask()
* drawSelfMasked()

# Moving the Ground

The illusion of movement in an endless runner is important to game play. To provide a little variety, I have designed three separate graphics which include flat land, a bump and a pot-hole. The variations are irrelevant to the game play.

These are enumerated in an enum called GroundType, as shown below:

enum GroundType {

Flat,

Bump,

Hole,

};

The ‘ground’ itself is made up of five images that are 32 pixels wide to give a combined width of 160 pixels. As you will see in a moment, the array of images will be rendered across the page overlapping the 128 pixels of the screen width. Moving the images a pixel to the left and re-rendering them will provide the illusion that the ground is moving.

The array is declared and initialised as shown below:

GroundType ground[5] = {

GroundType::Flat,

GroundType::Flat,

GroundType::Hole,

GroundType::Flat,

GroundType::Flat,

};

When rendering the ground for the first time, the first four images are rendered at X position 0, 32, 64 and 96 respectively. The fifth image is also rendered but as its X position is 128 it is not visible off to the right of the screen.

The following code renders the ground. It loops through the ground array and draws the five elements 32 pixels apart.

for (byte i = 0; i < 5; i++) {

switch (ground[i]) {

case GroundType::Flat:

Sprites::drawSelfMasked((i \* 32) - groundX, 55, ground\_flat, frame);

break;

case GroundType::Bump:

Sprites::drawSelfMasked((i \* 32) - groundX, 55, ground\_bump, frame);

break;

case GroundType::Hole:

Sprites::drawSelfMasked((i \* 32) - groundX, 55, ground\_hole, frame);

break;

}

}

In the code above, the variable groundX is used as an offset and is initially set to zero so has no affect. To scroll the ground to the left, the ground variable is incremented. Assuming the value is now one, this results in the five images being rendered at X positions -1, 31, 63, 95, and 127 respectively. The left most pixels of the first image are no longer visible and the left most pixels of the right are now rendered on the right most side of the screen.

The ground can be continued to be scrolled until the offset equals 32 (our image width) at which point the images are being rendered at the X positions -32, 0, 32, 64 and 96 respectively. At this point the first image is completely off screen. At this point, we need to move the elements of the ground array to the left one position and randomly select an image for the fifth position.

The code below detects when the offset has reached the 32 and randomly selects a number between 0 and 5 and assigns the ground type accordingly. One thing to note about the random() function is that the lower bound is inclusive whereas the upper bound is inclusive. A value of 3 or lower results in a flat section of ground whereas the values 4 and 5 are mapped to a bump and a pothole. This approach ensures that many more flat sections of ground are generated.

Finally, the elements of the array are shuffled to the left and the newly generated ground type is assigned to the fifth element.

if (groundX == 32) {

groundX = 0;

byte type = random(0, 6);

switch (type) {

case 0 ... 3:

type = GroundType::Flat;

break;

case 4:

type = GroundType::Bump;

break;

case 5:

type = GroundType::Hole;

break;

}

ground[0] = ground[1];

ground[1] = ground[2];

ground[2] = ground[3];

ground[3] = ground[4];

ground[4] = (GroundType)type;

}

groundX++;

Launching an Obstacle

Steve the Dinosaur must avoid four type of obstacles - a single, double and triple cactus and a flying pterodactyl. These various obstacle types are enumerated in an enum called ObstacleType as shown below. The first three elements are self-explanatory - the two pterodactyl elements are used to represent the animal (I was about to say bird but they were in fact reptiles!) with its wing up and down. Later we will see how we animate the image but for now you can ignore the second element, Pterodactyl2.

enum ObstacleType {

SingleCactus,

DoubleCactus,

TripleCactus,

Pterodactyl1,

Pterodactyl2,

Count\_CactusOnly = 3,

Count\_AllObstacles = 4,

};

The details of a single obstacle are stored in a structure as defined below. In addition to the obstacle’s position, the structure also contains the object type, an enabled flag and a reference to the image that will be used when rendering it. As mentioned earlier, structures are a great mechanism for capturing related data together.

struct Obstacle {

int x;

int y;

ObstacleType type;

bool enabled;

const byte \*image;

};

At any time during game play, two or even three obstacles may be visible on the screen.

Obstacle obstacles[NUMBER\_OF\_OBSTACLES] = {

{ 0, 0, ObstacleType::Pterodactyl1, false, pterodactyl\_1 },

{ 0, 0, ObstacleType::Pterodactyl1, false, pterodactyl\_1 },

{ 0, 0, ObstacleType::Pterodactyl1, false, pterodactyl\_1 },

};

#define NUMBER\_OF\_OBSTACLES 3

#define OBSTACLE\_LAUNCH\_DELAY\_MIN 90

#define OBSTACLE\_LAUNCH\_DELAY\_MAX 200

// Should we launch another obstacle?

--obstacleLaunchCountdown;

if (obstacleLaunchCountdown == 0) {

for (byte i = 0; i < NUMBER\_OF\_OBSTACLES; i++) {

if (!obstacles[i].enabled) {

launchObstacle(i);

break;

}

}

obstacleLaunchCountdown = random(OBSTACLE\_LAUNCH\_DELAY\_MIN, OBSTACLE\_LAUNCH\_DELAY\_MAX);

}

#define PTERODACTYL\_UPPER\_LIMIT 27

#define PTERODACTYL\_LOWER\_LIMIT 48

void launchObstacle(byte obstacleNumber) {

ObstacleType type;

if (score < SCORE\_START\_PTERODACTYL) {

type = (ObstacleType)random(ObstacleType::SingleCactus, ObstacleType::Count\_CactusOnly);

}

else {

type = (ObstacleType)random(ObstacleType::SingleCactus, ObstacleType::Count\_AllObstacles);

}

switch (type) {

case ObstacleType::SingleCactus:

case ObstacleType::DoubleCactus:

case ObstacleType::TripleCactus:

obstacles[obstacleNumber].type = type;

obstacles[obstacleNumber].enabled = true;

obstacles[obstacleNumber].x = WIDTH - 1;

obstacles[obstacleNumber].y = CACTUS\_GROUND\_LEVEL;

break;

case ObstacleType::Pterodactyl1:

obstacles[obstacleNumber].type = ObstacleType::Pterodactyl1;

obstacles[obstacleNumber].enabled = true;

obstacles[obstacleNumber].x = WIDTH - 1;

obstacles[obstacleNumber].y = random(PTERODACTYL\_UPPER\_LIMIT, PTERODACTYL\_LOWER\_LIMIT);

break;

}

}