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()

Arrays of Sprites

# Basic Program

Setup()

Loop()

Frames()

# 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.

Images of ground

These are enumerated in an enum called GroundType, as shown below. The first element in an enumeration is implicitly assigned a value of zero and subsequent elements increase in value by one. An array of images has also been declared with the images arranged in the same order as the enumeration allowing us to use the enumeration elements as index values to retrieve the images.

enum GroundType {

Flat,

Bump,

Hole,

};

const byte \*ground\_images[] = { ground\_flat, ground\_bump, ground\_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++) {

Sprites::drawSelfMasked((i \* 32) - groundX, GROUND\_LEVEL, ground\_images[ground[i]], frame);

}

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++;

# Moving and Rendering Steve

When Steve is not standing around waiting for the game to begin he may be running, ducking or simply be dead. These various states or stances are shown below.

Images of Steve

These stances are enumerated in an enum called Stance. As with the ground types detailed above, a matching array of dinosaur images has been declared with the images arranged in the same order as the Stance enumeration. A second array of masks has also been declared – again with the masks in the same order as the Stance enumeration – to allow us to use the Stance values as indexes.

enum Stance {

Standing,

Running1,

Running2,

Ducking1,

Ducking2,

Dead1,

Dead2,

};

const byte \*steve\_images[] = { dinosaur\_still, dinosaur\_running\_1, dinosaur\_running\_2, dinosaur\_ducking\_1, dinosaur\_ducking\_2, dinosaur\_dead\_1, dinosaur\_dead\_2 };

const byte \*steve\_masks[] = { dinosaur\_still\_mask, dinosaur\_running\_1\_mask, dinosaur\_running\_2\_mask, dinosaur\_ducking\_1\_mask, dinosaur\_ducking\_2\_mask, dinosaur\_dead\_2\_mask, dinosaur\_dead\_2\_mask };

All of the details relating to Steve’s position and current stance are stored in a single structure. Note that in structures, we can include fields using the common data types (integers, Booleans and so forth) and ones that refer to the enumerations we have already defined.

struct Steve {

int x;

int y;

Stance stance;

bool jumping;

byte jumpIndex;

const byte \*image;

const byte \*mask;

};

Once the structure has been declared, we can create an instance of it and initialise all of the elements in a single line as shown below. We now have a container to track changes in Steve’s position.

Steve steve = {0, STEVE\_GROUND\_LEVEL, Stance::Standing, false, false, dinosaur\_still, dinosaur\_still\_mask };

Our Steve structure has two fields that point to the image and mask that represent his current stance. Later we will use these to determine whether Steve has crashed into a cactus or pterodactyl. For now, we can populate these by looking up looking up the image and mask from the array of images using Steve’s current stance as an index.

Steve can then be rendered using the Sprites::drawExternalMask() function with Steve’s current position and the two image references. As Steve can be standing or ducking, I have assumed that his coordinates represent the lower left position. By default, images are rendered from the top left position and therefore I have subtracted the height of the image from Steve’s Y position to accommodate this.

void drawSteve() {

steve.image = steve\_images[steve.stance];

steve.mask = steve\_masks[steve.stance];

Sprites::drawExternalMask(steve.x, steve.y - getImageHeight(steve.image), steve.image, steve.mask, frame, frame);

}

Now that we can draw Steve on the screen, we need to be able to move him around. Our structure includes an X and Y coordinates plus fields that represent his current stance and whether or not he is jumping.

Handling user input is generally handled in the main loop of the program. The code below tests first whether Steve is jumping and, if not, allows the player to control him. The Arduboy library provides four commands for detecting the pressing and releasing of buttons - justPressed(), justReleased(), pressed() and noPressed(). The first two commands are used in conjunction with the command pollButtons().

A call to pollButtons() at the start of the main loop captures the current state of the Arduboy’s buttons. The justPressed()and justReleased() commands will return true if the user has pressed a button since the last time pollButtons() was called.

The pressed() command returns true if the nominated button is pressed or remains pressed over a period of time. You can see that I am using the justPressed()command to detect a jump as it is a one off event whereas I am allowing the user to hold the left and right buttons to move continuously by using the pressed() command .

arduboy.pollButtons();

…

if (!steve.jumping) {

if (arduboy.justPressed(A\_BUTTON)) {

steve.jumping = true;

steve.jumpIndex = 0;

}

if (arduboy.justPressed(B\_BUTTON)) {

if (steve.stance != Stance::Ducking2) {

steve.stance = Stance::Ducking1;

};

}

if (arduboy.pressed(LEFT\_BUTTON) && steve.x > 0) {

steve.x--;

}

if (arduboy.pressed(RIGHT\_BUTTON) && steve.x < 100) {

steve.x++;

}

if (arduboy.notPressed(B\_BUTTON) && (steve.stance == Stance::Ducking1 || steve.stance == Stance::Ducking2)) {

steve.stance = Stance::Running1;

}

}

The last clause of the code above detects whether the player has let go of the B button (the duck button) and returns Steve to a running stance if he was already ducking. The second part of the condition is important as the notPressed() command - as the name suggests - will return true when the user is not pressing the button and we do not want Steve’s stance to be changed unless he was actually ducking.

At the end of each loop, we need to update Steve’s position if he is jumping or change the image displayed to give the illusion of his feet moving.

**Jumping**

Previously, we detected if the user had pressed the A button and set the jumping flag to true. If it is true we want Steve to jump an over oncoming obstacles in an arc that somewhat simulates the effects of gravity – fast acceleration at the start of the jump, deceleration as he reaches full height followed by acceleration as he falls to earth,

The array below describes the Y positions of Steve as his jump progresses. It starts and ends at Y = 55 (or ground level) and reaches a height of 19 in the middle.

unsigned char jumpCoords[] = {55, 52, 47, 43, 40, 38, 36, 34, 33, 31, 30, 29, 28, 27, 26, 25, 24, 24, 23, 23, 22, 22, 21, 21, 20, 20, 20, 20, 19, 19, 19, 19, 19, 20, 20, 20, 20, 21, 21, 22, 22, 23, 23, 24, 24, 25, 26, 27, 28, 29, 30, 31, 33, 34, 36, 38, 40, 43, 47, 51, 55 };

The code below updates Steve’s position when jumping. In addition to the jumping flag, the data structure that maintains Steve’s details also includes an array index, named jumpIndex, which is used to keep track of the current position in the array. On each pass of the main loop, adjust Steve’s height to the value within the array that the index points to. We then increment the array index up until the last value of the array is reached at which point Steve has returned to the ground and the jump is over. Steve’s jumping property is set to false and the array index is set to zero in anticipation of his next jump.

void updateSteve() {

if (steve.jumping) {

steve.y = jumpCoords[steve.jumpIndex];

steve.jumpIndex++;

if (steve.jumpIndex == sizeof(jumpCoords)) {

steve.jumping = false;

steve.jumpIndex = 0;

steve.y = STEVE\_GROUND\_LEVEL;

}

}

else {

...

}

}

**Running or Ducking**

The illusion of running is achieved by alternating the images used when rendering Steve. If we alternate the images every frame (or every time we pass through the main loop), Steve’s legs will appear as a blur as this is too fast. To slow this down, we can use an Arduboy library function called everyXFrames(n) which returns true on every nth frame.

The code below alternates Steve’s stance every 3 frames. If his current stance is Stance::Running1 then it becomes Stance::Running2 – if it was already Stance::Running2 then it reverts back to the original value. This is true of the ducking images but not true of the dead images – once you are dead, you remain dead.

void updateSteve() {

if (steve.jumping) {

...

}

else {

if (arduboy.everyXFrames(3)) {

switch (steve.stance) {

case Stance::Running1:

steve.stance = Stance::Running2;

break;

case Stance::Running2:

steve.stance = Stance::Running1;

break;

case Stance::Ducking1:

steve.stance = Stance::Ducking2;

break;

case Stance::Ducking2:

steve.stance = Stance::Ducking1;

break;

case Stance::Dead1:

steve.stance = Stance::Dead2;

break;

default:

break;

}

}

}

}

Now that we have Steve running, jumping and ducking we can move on to the obstacles he will need to avoid.

# Launching an Obstacle

Steve the Dinosaur must avoid four type of obstacles - the single, double and triple cacti 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\_AllObstacles = 4,

};

Unless otherwise specified, elements in an enumeration are assigned values starting from zero. In the enumeration above, the SingleCactus element has a value of 0 and the Pterodactly2 element has a value of 4. Although there are five elements, when we randomly launch objects there are only four types to choose from as the Pterodactyl1 and Pterodacytl2 elements describe the same thing. The element Count\_AllObstacles is used to define the number of options available. Note that I have explicitly assigned it a value of 4. Enumerations do not have to have contiguous element values and multiple elements can have the same values.

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. To cater for this, I have created an array of obstacles and initialised them with default values. Note that all of the obstacles are disabled by default.

#define NUMBER\_OF\_OBSTACLES 3

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 },

};

When launching obstacles, we need to make sure that the obstacles are randomly placed but not too close together otherwise Steve may not be able to land between them and jump again. To facilitate this, we generate a random delay and store this into the variable obstacleLaunchCountdown which is decremented each pass of the main game loop. When this variable reaches zero, a simple loop passes through the obstacles[] array looking for the first inactive obstacle in the collection.

#define LAUNCH\_DELAY\_MIN 90

#define LAUNCH\_DELAY\_MAX 200

--obstacleLaunchCountdown;

if (obstacleLaunchCountdown == 0) {

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

if (!obstacles[i].enabled) {

launchObstacle(i);

break;

}

}

obstacleLaunchCountdown = random(LAUNCH\_DELAY\_MIN, LAUNCH\_DELAY\_MAX);

}

The actual code to launch a new obstacle is shown below. The input parameter, obstacleNumber, defines which of the three obstacles in the array to activate. To help the player get accustomed to the various obstacles, they are introduced slowly as the player’s score increases.

The first section of the routine calculates which of the elements in the ObstacleType enumeration can be chosen based on the player’s score. When the player’s score is less than 100, only the single cacti is valid. When the player’s score is less than 200, the single and double cacti images are valid - likewise a score less than 300 allows all three cacti obstacles to be chosen. Once the score exceeds 300 all obstacles including the pterodactyl are valid.

If a pterodactyl obstacle is chosen then a flying height is randomly selected between an upper and lower limit as defined by the two constants PTERODACTYL\_UPPER\_LIMIT and PTERODACTYL\_LOWER\_LIMIT. In contrast, cacti are all launched at ground level.

#define PTERODACTYL\_UPPER\_LIMIT 27

#define PTERODACTYL\_LOWER\_LIMIT 48

void launchObstacle(byte obstacleNumber) {

// Randomly pick an obstacle ..

ObstacleType randomUpper = ObstacleType::SingleCactus;

switch (score) {

case 0 ... 99:

randomUpper = ObstacleType::SingleCactus;

break;

case 100 ... 199:

randomUpper = ObstacleType::DoubleCactus;

break;

case 200 ... 299:

randomUpper = ObstacleType::TripleCactus;

break;

default:

randomUpper = ObstacleType::Count\_AllObstacles;

break;

}

ObstacleType type = (ObstacleType)random(ObstacleType::SingleCactus, randomUpper + 1);

// Launch the obstacle ..

obstacles[obstacleNumber].type = type;

obstacles[obstacleNumber].enabled = true;

obstacles[obstacleNumber].x = WIDTH - 1;

if (type == ObstacleType::Pterodactyl1) {

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

}

else {

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

}

}

It’s worth pointing out the use of ranges in the case statement in the above example. Ranges must be specified in the format shown with three decimal points between them and the ranges cannot overlap. Although this syntax is valid in the Arduino / Arduboy environment, it is not valid in most C++ implementations.

# Moving Obstacles

The obstacles in our game move from right to left, one pixel per frame. The code below checks to see which of our obstacles are enabled and decrements their x coordinate by one. As the objects move out of view on the left hand side of the screen, they are disabled which allows them to be relaunched in the future.

We apply the same technique used to animate Steve’s feet to animate the pterodactyl’s wings.

void updateObstacles() {

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

if (obstacles[i].enabled == true) {

switch (obstacles[i].type) {

case ObstacleType::Pterodactyl1:

case ObstacleType::Pterodactyl2:

if (arduboy.everyXFrames(2)) {

if (obstacles[i].type == Pterodactyl1) {

obstacles[i].type = Pterodactyl2;

}

else {

obstacles[i].type = Pterodactyl1;

}

}

obstacles[i].x--;

break;

case ObstacleType::SingleCactus:

case ObstacleType::DoubleCactus:

case ObstacleType::TripleCactus:

obstacles[i].x--;

break;

}

// Has the obstacle moved out of view ?

if (obstacles[i].x < -getImageWidth(obstacles[i].image)) {

obstacles[i].enabled = false;

}

}

}

}

# Saving Scores

The Arduboy includes a small amount of non-volatile memory, known as EEPROM, which can store and retain information even when the unit is turned off.

EEPROM is ideal for saving user settings, high scores and other information between sessions. EEPROM stands for Electrically Erasable Programmable Read-Only Memory but this is a misnomer as the memory can actually be updated. EEPROMs have a limited life and will eventually fail after they have been erased and rewritten too many times – this number may be in the millions of operations but a poorly written program that attempts to use it as working memory could easily reach that.

The EEPROM class provides three basic functions to read and write a single byte of memory, as shown below. The memory location can be anywhere in the 1Kb and equates to a value between 0 and 1023. The update() function differs from the write() function in that it checks the value to be written against what is already stored in order to minimize the number of updates thus prolonging the life of the EEPROM.

EEPROM.read(memory\_location);

EEPROM.update(memory\_location, value);

EEPROM.write(memory\_location, value);

The library also offers two other functions that can save and retrieve datatypes other than a byte, such as a float, integer or even a structure.

EEPROM.put(memory\_location, value);

EEPROM.get(memory\_location, value);

Using these functions, we can save Steve’s top scores. We can save it anywhere in the 1Kb range however the first 16 bytes are reserved for storing Arduboy system details including the current sound state (on / off), the unit name and other bits and pieces.

The Arduboy library defines a constant, EEPROM\_STORAGE\_SPACE\_START, which indicates the first memory location free for user information. The code below allows us to save and retrieve Steve’s score into the lowest available EEPROM memory location.

EEPROM.get(EEPROM\_STORAGE\_SPACE\_START, highScore);

EEPROM.put(EEPROM\_STORAGE\_SPACE\_START, highScore);

Depending on what other games we have been playing previously, these memory locations may contain invalid data that can cause an error or, at worst, report unrealistically high scores. To overcome this, I like to store two fixed characters in front of my application’s data. When the application starts, it checks in the EEPROM memory for the two characters and if it does not find them clears out the memory it plans to use. It then populates the two characters so future checks do not clear the score again.

This is achieved using the following code:

#define EEPROM\_START\_C1 EEPROM\_STORAGE\_SPACE\_START

#define EEPROM\_START\_C2 EEPROM\_START\_C1 + 1

#define EEPROM\_SCORE EEPROM\_START\_C1 + 2

void initEEPROM() {

unsigned char c1 = EEPROM.read(EEPROM\_START\_C1);

unsigned char c2 = EEPROM.read(EEPROM\_START\_C2);

if (c1 != ‘S’ || c2 != ‘T’) {

EEPROM.update(EEPROM\_START\_C1, ‘S’);

EEPROM.update(EEPROM\_START\_C2, ‘T’);

EEPROM.put(EEPROM\_SCORE, (unsigned int)0);

}

}

# Putting it all together