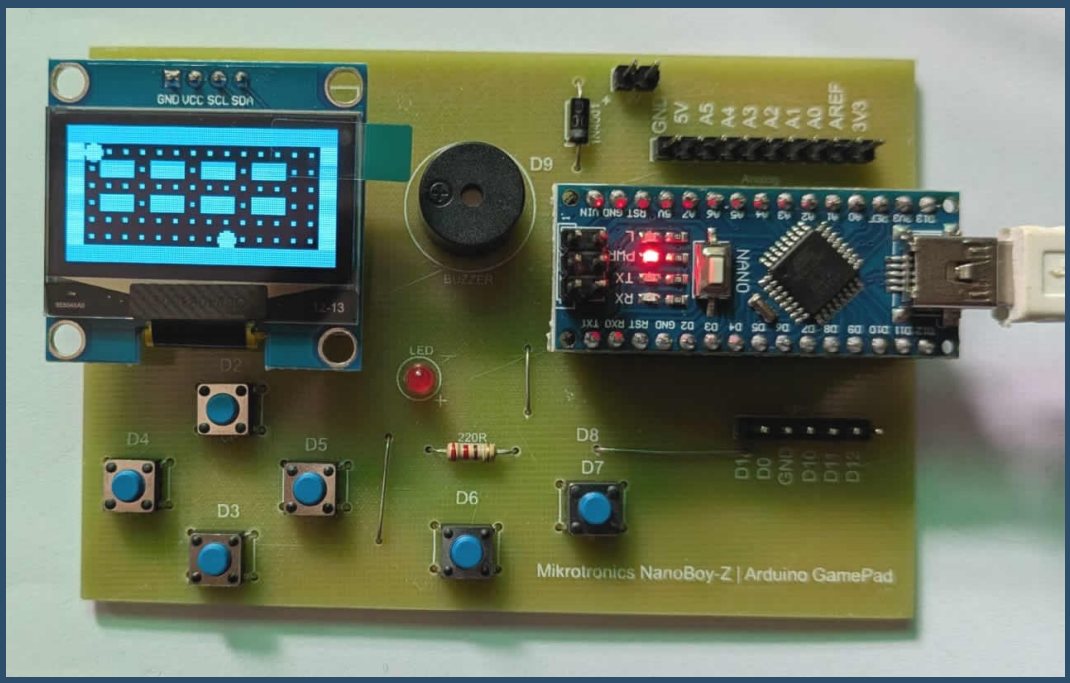
NanoBoy

Arduino Game Development

Learn Embedded Systems Programming the Fun Way



Amer Iqbal Qureshi

Mikrotronics Pakistan | https://mikro.pk

Preface

Electronics and programming have always fascinated curious minds. For many, the journey begins not in a laboratory, but on a small desk with a handful of components, a soldering iron, and an eagerness to learn. *Mikrotronics Pakistan* was born out of such a passion.

Founded in **2005** as a hobby project, Mikrotronics Pakistan has since grown into a vibrant community of **electronics enthusiasts, hobbyists, and students** from different walks of life. Over the years, it has evolved from a small group of tinkerers into a recognized platform for promoting **electronics education** and **embedded systems programming**.

The mission of Mikrotronics Pakistan is simple yet powerful:

- To help students and hobbyists of all backgrounds get started with electronics.
- To provide learning resources and affordable development boards.
- To encourage creativity and innovation through **hands-on projects**.
- To make embedded systems accessible to non-engineers, empowering everyone to turn ideas into reality.

About the Author

This book is written by **Dr. Amer Iqbal Qureshi**, one of the **founding members** of Mikrotronics Pakistan.

Dr. Qureshi is a **Professor of Cardiac Surgery** in a public sector hospital in Pakistan. His professional life is dedicated to performing complex **adult cardiac surgeries** and mentoring **MS, FCPS, and PhD students** in their medical careers.

Alongside his demanding medical career, Dr. Qureshi has always been deeply passionate about **electronics, microcontrollers, and technology**. He has authored several books on **microcontroller programming for non-engineers**, introducing countless students and hobbyists to the exciting world of embedded systems. Over the years, he has also designed and developed a number of **training boards and educational kits** that simplify learning for beginners.

The **NanoBoy** project is yet another step in this journey. Conceived as a fun and educational platform, it is designed to make learning **embedded systems programming** enjoyable through the lens of **game development**. By combining play with learning, NanoBoy helps beginners grasp concepts like graphics, input handling, and system logic in a way that feels natural and engaging.

How to Use this Book

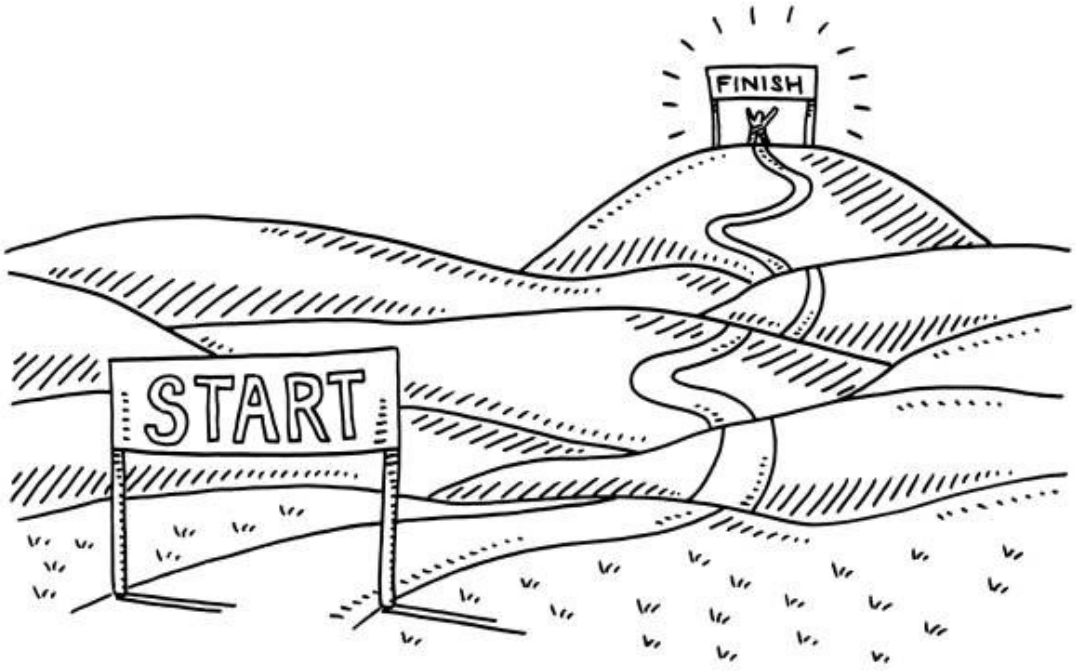
This book is written with **learners in mind**—whether you are a student, a hobbyist, or simply curious about embedded systems.

- **No prior experience required:** We start from the basics, gradually introducing new concepts with practical examples.
- **Hands-on learning:** Every chapter includes code snippets and complete programs that you can run on the NanoBoy development board.
- **Step-by-step progression:** The book begins with simple programs (like displaying text and shapes) and gradually advances to **full game development** projects.
- **Beyond games:** While the primary focus is on games, the concepts and library functions you will learn are equally useful for building **interactive applications and user interfaces**.

Tip for learners: Don't just read—type the code, experiment, and modify it. The real power of NanoBoy lies in *exploring and creating your own ideas*.

By the end of this book, you will not only have built a variety of fun games, but also gained the confidence to design and program your own embedded applications.

Part I – Getting Started



Chapter 1
Introduction to NanoBoy

What is NanoBoy?

NanoBoy is a compact, educational **development board** designed for students and hobbyists to learn **embedded systems programming** through a fun, hands-on approach. At its heart, it is powered by the **Arduino Nano**, one of the most popular beginner-friendly microcontrollers. What makes NanoBoy unique is that it comes bundled with everything needed to design and play simple games, experiment with graphics, and build interactive projects.

NanoBoy combines:

- **Arduino Nano** (the brain of the system).
- **OLED display** (to show text, graphics, sprites, and games).
- **Six control buttons** (to interact with games and applications).
- **Buzzer** (for generating sound effects and tones).
- **LED** (for visual signals, debugging, or status indication).

In short, it's a **pocket-sized game console and development platform**, but one that is entirely open, programmable, and designed for **education and creativity**.

The Vision: Learning Embedded Systems Through Fun and Games

Learning microcontroller programming can sometimes feel **too technical and dry**—setting up sensors, blinking LEDs, or reading datasheets may not sound exciting to a beginner.

NanoBoy takes a **different path**:

- It teaches **core embedded programming skills** (loops, functions, arrays, interrupts, etc.) through **games**.
- Instead of writing boring code, students learn by building **Snake**, **Tic Tac Toe**, **Pong**, and even more complex projects like **Checkers**.
- It demonstrates important concepts like **graphics rendering, collision detection, sound generation, and user input** in a playful, engaging manner.

The philosophy is simple:

"When learning feels like playing, knowledge sticks for life."

Overview of NanoBoy Hardware

NanoBoy's design is minimal yet powerful. Here is what you'll find on the board:

- **Arduino Nano**:
  - 16 MHz ATmega328P microcontroller.
  - 32 KB Flash memory for programs.
  - 2 KB SRAM for variables.
  - USB connectivity for uploading sketches.
- **OLED Display (128×64 pixels)**:
  - Small, crisp, monochrome graphics display.
  - Driven by the SSD1306 driver chip.
  - Can show text, shapes, sprites, and animations.
- **Buttons**:
  - Directional pad: Up, Down, Left, Right.
  - Action buttons: A and B.
  - Used for controlling games and navigating menus.
- **Buzzer**:
  - Small piezoelectric buzzer.
  - Generates beeps, sound effects, or simple tunes.
- **LED**:
  - General-purpose indicator light.
  - Can be toggled in code to signal events or provide visual feedback.
- **Expansion Header**
  - All other unused pins are available as header for interfacing with other modules or sensors if required

This combination makes NanoBoy feel like a **mini handheld console**, yet fully open for coding experiments.

Comparison with Other Platforms

NanoBoy draws inspiration from earlier hobbyist handheld consoles:

- **Arduboy**
  - A credit-card sized gaming device based on ATmega32u4.
  - Offers similar monochrome graphics and buttons.
  - Primarily focused on gaming, with a large community.
- **Gamebuino**
  - A more advanced open-source console.

    - Features a color screen, SD card, and more advanced hardware.
    - Designed for larger, more complex games.
- **NanoBoy**
    - Simpler, cheaper, and more accessible.
    - Focused on **learning and experimentation**.
    - Directly based on the **Arduino Nano**, making it compatible with Arduino libraries and tools.
    - Great for **teaching beginners**, but still powerful enough for creative game projects.

NanoBoy fills the gap between **basic Arduino projects** and **dedicated handheld consoles**. It's the ideal stepping stone for anyone who wants to **learn programming by making games**.

Not only games but many other projects, since it can interface with external hardware, like sensor modules, and output devices.

Chapter 2
Getting Started with NanoBoy

Before we can dive into making games and applications with NanoBoy, we need to set up the tools that allow us to communicate with the board. This chapter is about preparing your computer, installing the Arduino environment, and connecting the NanoBoy library so that you can begin writing your very first programs. By the end of this chapter, you will have written a small test program that displays a message on the screen, and another that responds to button presses and makes sound through the buzzer. These simple steps will give you the confidence that everything is working correctly and prepare you for the more advanced projects in later chapters.

Setting Up the Arduino IDE

NanoBoy is based on the Arduino Nano microcontroller, and therefore it can be programmed using the popular Arduino IDE. The IDE, short for Integrated Development Environment, is a simple program where you write, edit, and upload code to the board. If you have never used Arduino before, don't worry — this process is designed for beginners. The Arduino IDE is free to download from the official website, and versions are available for Windows, macOS, and Linux. Once installed, you connect the NanoBoy to your computer using a USB cable.

Inside the IDE, the correct board and port need to be selected. Since NanoBoy is essentially an Arduino Nano with an OLED screen and extra components, you will choose "Arduino Nano" as your board. Depending on the version of the chip, you may also need to select the processor type as "ATmega328P (Old Bootloader)". This ensures that the IDE knows how to talk to your specific hardware.

For those who do not have access to the NanoBoy hardware, we have developed a simulation tool, that can run your code and display results. Though not as good and smooth as using a real hardware, but still it can give you an idea as to how your software will behave in real world. The simulation file and how to use it will be mentioned in appropriate section of this chapter.

Installing the NanoBoy Library

The real magic of NanoBoy lies in its library. While the Arduino IDE gives you the basic ability to program the microcontroller, the NanoBoy library takes care of all the difficult details: talking to the screen, scanning the buttons, making sounds, and turning

the LED on and off. Without the library, you would have to write dozens of lines of code just to light a pixel on the OLED. With the library, you can do it in a single command.

The library can be downloaded from the Mikrotronics Pakistan GitHub page. Once you

https://github.com/ameriqbalqureshi/nanoboy

have the ZIP file, extract it. It has the actual library (as another .zip file 'nanoboy.zip') and some document files to help you find the proper functions. you open the Arduino IDE, go to the menu "Sketch → Include Library → Add .ZIP Library 'nanoboy.zip' ", and select the file. The IDE will install the NanoBoy library and make its examples available. A quick way to confirm installation is to look under "File → Examples" and check if a NanoBoy section appears.

NanoBoy library depends upon popular Adafruit SSD1306 and its associated libraries. In Library explorer just search for 'Adafruit SSD1306' and install that library, it will ask if you want to install other needed libraries, click All.

Writing Your First Program

Once everything is set up, the first program we write is traditionally called "Hello World." On a normal computer, this would simply print text to a terminal window. On

```
#include <NanoBoy.h>
NanoBoy nb;
void setup() {
  nb.begin();
  nb.clear();
  nb.setCursor(10, 20);
  nb.print("Hello, NanoBoy!");
  nb.update();
}
void loop() {
  // Nothing here for now
}
```

NanoBoy, our "Hello World" will appear on the OLED screen.

When you upload this program, the board should greet you with the message "Hello,

NanoBoy!" displayed neatly on the tiny screen. This is your first taste of how easy the library makes it to control the hardware.

Now let's go through it step by step.

```
#include <NanoBoy.h>
```

This line tells the compiler to include the **NanoBoy library**. Without this, the Arduino would know nothing about the screen, buttons, LED, or buzzer. The library contains all the special functions that make NanoBoy easy to use.

Think of this like opening a toolbox — once you include it, you have all the tools available for your project.

```
NanoBoy nb;
```

This line is very important. Here we are **creating an object** called `nb` of type `NanoBoy`.

- `NanoBoy` is the **class** (defined inside the library).
- `nb` is the **object** (an instance of the class).

Why is this declared **outside of any function** (globally)? Because we want to use this object everywhere in our program — both in `setup()` and in `loop()`. If we declared it inside a function, it would only exist while that function is running. Declaring it globally means it is always available.

Whenever you see `nb.something()`, you are calling a **method** (a function) that belongs to the `NanoBoy` object.

```
void setup() { ... }
```

This function runs **only once** when the board is powered on or reset. Here, we prepare the display and show our message.

`nb.begin();` This initializes the NanoBoy hardware. It sets up the OLED display, the pins for the buttons, buzzer, and LED. You **must call this once** at the start of every program, otherwise nothing will work.

`nb.clear();` This clears the screen, turning all pixels off. It ensures you start with a blank canvas.

`nb.setCursor(10, 20);` This sets the "text cursor" to coordinates `(10,20)` on the OLED display. The numbers mean:

`10` → pixels from the left edge

`20` → pixels from the top edge

If you don't set the cursor, text will usually begin at the top-left corner `(0,0)`.

`nb.print("Hello, NanoBoy!");` This prints the text to the display buffer. Notice the text is

written inside double quotes `""`, which means it is a **string literal**.

`nb.update();` The NanoBoy uses a "buffered" display system. When you draw or print something, it first goes into memory (the buffer). It will only appear on the screen when you call `update()`. This allows you to draw multiple things and then refresh the screen all at once, reducing flicker.

```
void loop() { ... }
```

This function runs **again and again forever**. In our first program, we don't want anything to change, so it is left empty. Later, we will use `loop()` to update game graphics,

Summary of New Commands

`NanoBoy nb;` → Creates the NanoBoy object (toolbox).
`nb.begin();` → Prepares NanoBoy hardware.
`nb.clear();` → Clears the display.
`nb.setCursor(x, y);` → Sets position for text.
`nb.print("text");` → Prints text to the screen buffer.
`nb.update();` → Refreshes the display to show changes.

move sprites, check button presses, and so on.

*Tip: If your text does not appear, always check that you have called both `nb.setCursor()` **before** printing, and `nb.update()` **after** printing.*

Interacting with Buttons and Buzzer

A development board is not much fun if it can only display text. Let us now make it respond to your input. In the next short program, pressing the A button will light up the LED and make the buzzer beep.

```
#include <NanoBoy.h>

NanoBoy nb;

void setup() {
  nb.begin();
}

void loop() {
  if (nb.buttonPressed(BTN_A)) {
    nb.setLED(true);
    nb.beep(500, 200); // 500 Hz for 200 milliseconds
```

```
  } else {
    nb.setLED(false);
  }
}
```

Now, when you press the A button, the LED will glow and the board will produce a short tone. This small demonstration combines input (the button), output (the LED), and sound (the buzzer). In other words, you have already experienced the three key channels of interaction available on NanoBoy.

let's now take the **Buttons Demonstration** program and walk through it in the same "beginner-friendly" style, but focusing only on the **new commands** we encounter compared to the *Hello World* program.

New Commands and Functions

Compared to our **Hello World**, here we meet **three important new things**:

`nb.buttonPressed(BTN_A)`

Checks if the **A button** is currently pressed.

Syntax:

```
nb.buttonPressed(BTN_NAME)
```

Parameter:

`BTN_A` → constant representing the A button pin.

Return:

`true` if pressed, `false` if not.

Here it's used inside an `if` statement:

```
if (nb.buttonPressed(BTN_A)) {
  // do something when button is pressed
}
```

`nb.setLED(true)`

Turns the **onboard LED** on or off.

Syntax:

```
nb.setLED(state)
```

Parameter:

`true` → LED ON

`false` → LED OFF

This is useful for simple feedback, like indicating when a button is pressed.

`nb.beep(1000, 200)`

Makes a short beep sound on the buzzer.

Syntax:

```
nb.beep(frequency, duration)
```

Parameters:

`frequency` → Pitch of the sound in Hz (e.g. 1000 Hz = medium pitch).

`duration` → How long the sound lasts in milliseconds (e.g. 200 = 0.2 seconds).

So `nb.beep(1000, 200);` plays a 1 kHz tone for 0.2 seconds.

How It All Works Together

The `if` statement constantly checks if **BTN_A** is pressed.

If pressed:

Turn on the LED (`nb.setLED(true)`)

Play a beep (`nb.beep(1000, 200)`)

If not pressed:

Turn off the LED (`nb.setLED(false)`).

This is your very first **interactive program** with NanoBoy: pressing a button makes a sound and lights the LED, just like real game controllers give visual and audio feedback.

Now let's extend the demo so that **pressing BTN_A** not only turns on the LED and plays a beep, but also **displays a message on the screen**. This gives students their first taste of combining **input (button)** with **output (LED, sound, text)**.

```
#include <NanoBoy.h>
NanoBoy nb;
void setup() {
  nb.begin();
}
void loop() {
  nb.clear();   // always clear the screen before drawing
```

```
  if (nb.buttonPressed(BTN_A)) {
    nb.setLED(true);                  // turn LED ON
    nb.beep(1000, 200);               // short beep
    nb.setCursor(10, 20);             // move cursor to (x=10,
y=20)
    nb.print("Button A Pressed!");    // show text
  } else {
    nb.setLED(false);                 // turn LED OFF
    nb.setCursor(10, 20);
    nb.print("Press A Button");       // default text
  }

  nb.display();  // refresh the OLED with updated content
}
```

This is a **mini user interface**. Students can immediately see how **hardware feedback** (LED + buzzer) and **visual feedback** (screen text) can be tied to **button input**.

Chapter 3
Working with Graphics

After learning how to display text and read buttons, the next step is exploring how NanoBoy handles **graphics**. Games are all about visuals, and NanoBoy makes it easy to draw simple shapes like lines, rectangles, circles, and pixels. Even though the OLED screen is only **128×64 pixels**, you will be surprised how much can be achieved with just black and white graphics.

In this chapter, we will begin by drawing basic shapes on the screen, and then learn how these can be combined to build more complex images. By the end of this chapter, you will understand how the NanoBoy library lets you "paint" the screen in a structured way, and you will be ready to move on to sprites and tiles in later chapters.

The Coordinate System (x,y)

The screen is like a sheet of graph paper. The **top-left corner** is coordinate `(0,0)`.

`x` increases as you move **right**. X ranges from 0 to 63

`y` increases as you move **down**. Y ranges from 0 to 127

So `(127,63)` is the bottom-right corner of the screen.

```
#include <NanoBoy.h>
NanoBoy nb;
void setup() {
  nb.begin();
}
void loop() {
  nb.clear();
  // Draw a pixel
  nb.drawPixel(10, 10, SSD1306_WHITE);
  // Draw a line
  nb.drawLine(20, 10, 100, 10, SSD1306_WHITE);
  // Draw a rectangle
```

```
        nb.drawRect(20, 20, 40, 20, SSD1306_WHITE);

        // Filled rectangle
        nb.fillRect(70, 20, 40, 20, SSD1306_WHITE);

        // Draw a circle
        nb.drawCircle(40, 50, 10, SSD1306_WHITE);

        // Filled circle
        nb.fillCircle(90, 50, 10, SSD1306_WHITE);

        nb.display();
        delay(1000);
      }
```

Functions Explained

`nb.drawPixel(x, y, color)`

Draws a single pixel.

`x, y` are coordinates, `color` is usually `SSD1306_WHITE`.

`nb.drawLine(x0, y0, x1, y1, color)`

Draws a straight line from `(x0,y0)` to `(x1,y1)`.

`nb.drawRect(x, y, w, h, color)`

Draws the outline of a rectangle.

`(x, y)` is top-left corner, `w` width, `h` height.

`nb.fillRect(x, y, w, h, color)`

Draws a filled rectangle (solid).

`nb.drawCircle(x, y, r, color)`

Draws the outline of a circle.

`(x, y)` is the center, `r` is radius.

`nb.fillCircle(x, y, r, color)`

Draws a filled (solid) circle.

What Students Learn Here

How to “address” different parts of the screen with coordinates.

The difference between drawing outlines and filled shapes.

That graphics are drawn into a buffer, and `nb.display()` is always needed to show the result.

Example: Moving Line Animation

This draws a line whose end point keeps moving, giving the effect of an animated line.

```
#include <NanoBoy.h>
NanoBoy nb;
int x = 0;   // changing coordinate
int y = 0;
void setup() {
  nb.begin();
}
void loop() {
  nb.clear();
  // Draw a static starting point
  nb.drawPixel(0, 0, SSD1306_WHITE);
  // Draw line from top-left (0,0) to (x,y)
  nb.drawLine(0, 0, x, y, SSD1306_WHITE);
  nb.display();
  // update coordinates
  x += 2;
  y += 1;

  // reset if off screen
  if (x > 127) x = 0;
  if (y > 63)  y = 0;
```

```
  delay(50);
}
```

Draw a Line Art

```
#include <NanoBoy.h>
NanoBoy nb;
void setup() {
  nb.begin();
}
void loop() {
  nb.clear();
for(int x=0;x<128;x+=4){
  nb.drawLine(x, 0, 127-x, 63);
  nb.display();

}

}
```

Chapter 4
Sprites and Tiles

Up to now you've been drawing shapes and printing text. That's fine for simple demos, but games need characters that move around and worlds to explore. This is where **sprites** and **tiles** come in.

Think of a **sprite** as a little picture — your player character, an enemy, a coin. You can move it anywhere on the screen. A **tile** is a small square image used to build a larger background, like a mosaic floor. Together they're the building blocks of almost every classic game.

Defining a Sprite

In NanoBoy a sprite is represented by a `Sprite` structure:

```
struct Sprite {
  int x, y, w, h;          // position and size
  const uint8_t *bitmap;   // pointer to image data
  bool active;             // can be used to show/hide
};
```

You provide an image (bitmap), its width and height, and coordinates `(x,y)` on the screen. Here's a very simple 8×8 bitmap of a ball (each byte = 8 pixels wide):

```
// 8×8 ball bitmap stored in PROGMEM to save RAM
const uint8_t ballBitmap[] PROGMEM = {
  0x3C,
  0x7E,
  0xFF,
  0xFF,
  0xFF,
  0xFF,
  0x7E,
```

```
  0x3C
};

Sprite ball = { 60, 30, 8, 8, ballBitmap, true };
```

Now you can draw it with:

```
nb.drawSprite(ball);
```

Moving a Sprite

In your `loop()` you simply change `ball.x` and `ball.y` and redraw. This lets you animate it across the screen.

Building Worlds with Tiles

Drawing every pixel of a maze would be a nightmare. Instead, you store small square images called **tiles** — for example, a floor tile, a wall tile, a door tile. Then you arrange them in a 2-D array called a **map**. NanoBoy can render the whole map with one call.

A tile is typically 8×8 pixels. A tile set is a list of these bitmaps. A map is an array of numbers that index into that tile set.

Example: Small Tile-Based Background with a Moving Sprite

```
#include <NanoBoy.h>

NanoBoy nb;

// Define two 8×8 tiles: floor and wall
const uint8_t tiles[] PROGMEM = {
  // Tile 0: floor (blank)
  0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
  // Tile 1: wall (checker)
  0xAA,0x55,0xAA,0x55,0xAA,0x55,0xAA,0x55
};
```

```
// 4×8 map using tile indices
const uint8_t map[] PROGMEM = {
  1,1,1,1,1,1,1,1,
  1,0,0,0,0,0,0,1,
  1,0,0,0,0,0,0,1,
  1,1,1,1,1,1,1,1
};

// Simple sprite (player) as 8×8 square
const uint8_t playerBitmap[] PROGMEM = {
  0xFF,0x81,0x81,0xFF,0xFF,0x81,0x81,0xFF
};

Sprite player = {16, 16, 8, 8, playerBitmap, true};

void setup() {
  nb.begin();
}

void loop() {
  nb.clear();

  // draw map: 4 rows, 8 columns
  nb.drawTileMap(map, 4, 8, tiles);

  // move player with buttons
  if (nb.buttonPressed(BTN_LEFT))  player.x -= 1;
  if (nb.buttonPressed(BTN_RIGHT)) player.x += 1;
  if (nb.buttonPressed(BTN_UP))    player.y -= 1;
  if (nb.buttonPressed(BTN_DOWN))  player.y += 1;
```

```
    nb.drawSprite(player);
    nb.display();
  }
```

Upload it. You'll see a border of "walls" drawn from tiles and a little square you can move around inside with the arrow buttons.

New Commands Introduced

Command	What it does	Parameters
`nb.drawSprite(const Sprite &sprite);`	Draws a sprite bitmap at its `x`, `y` coordinates.	`sprite` = your Sprite struct
`nb.drawTileMap(const uint8_t *map, int rows, int cols, const uint8_t *tiles);`	Draw a tile-based background given a map array and tile set.	`map` = array of tile indices, `rows`, `cols` = map size, `tiles` = bitmaps of tiles

(Tip: Store your bitmaps and maps in `PROGMEM` to save RAM. Use 8×8 tiles to match the library's default tile size `_T_SIZE`.)

Narrative Transition

With sprites you can create characters and objects. With tiles you can create worlds. Together they form the heart of any game. In the next section you'll add **collision detection** so your player can bump into walls and objects like a real game.

Chapter 5
Collision Detection

So far your sprite could happily float through walls. Real games need rules: a ball bounces, a character stops at a wall, a spaceship collides with an asteroid. This is called **collision detection** — checking whether two objects overlap on the screen.

NanoBoy makes this simple. You don't have to write the math yourself; you just call one function with two sprites and it tells you whether they are touching.

The `checkCollision()` Function

Prototype:

```
bool checkCollision(const Sprite &a, const Sprite &b);
```

It returns `true` if sprite `a` and sprite `b` overlap, `false` otherwise.

Internally it compares the `x`, `y`, `w`, `h` fields of each sprite's rectangle. You can use this to stop movement, bounce objects, play sounds, or increase scores.

Example: Bouncing Ball Against a Paddle

Here's a small program that uses collision detection to bounce a ball off a paddle you control:

```
#include <NanoBoy.h>


NanoBoy nb;


const uint8_t ballBitmap[] PROGMEM = {
  0x3C,
  0x7E,
  0xFF,
  0xFF,
  0xFF,
  0xFF,
```

```
  0x7E,
  0x3C
};

const uint8_t paddleBitmap[] PROGMEM = {
  0xFF,0xFF,0xFF,0xFF,
  0xFF,0xFF,0xFF,0xFF
};

// Create sprites
Sprite ball   = {60, 30, 8, 8, ballBitmap, true};
Sprite paddle = {50, 50, 16, 4, paddleBitmap, true};

// Ball velocity
int dx = 1;
int dy = 1;

void setup() {
  nb.begin();
}

void loop() {
  nb.clear();

  // Move paddle left/right with buttons
  if (nb.buttonPressed(BTN_LEFT))  paddle.x -= 2;
  if (nb.buttonPressed(BTN_RIGHT)) paddle.x += 2;

  // Move ball
```

```
    ball.x += dx;
    ball.y += dy;

    // Bounce off screen edges
    if (ball.x <= 0 || ball.x + ball.w >= SCREEN_WIDTH)  dx = -
  dx;
    if (ball.y <= 0) dy = -dy;

    // Check collision with paddle
    if (nb.checkCollision(ball, paddle)) {
      dy = -dy; // bounce up
      nb.beep(800, 50); // play sound on hit
    }

    // Draw everything
    nb.drawSprite(ball);
    nb.drawSprite(paddle);
    nb.display();
  }
```

Upload it. You'll see a ball bouncing and your paddle at the bottom. Move the paddle with Left/Right buttons; when the ball hits the paddle it bounces and beeps.

New Command Introduced

Command	What it does	Parameters
`nb.checkCollision(const Sprite &a, const Sprite &b);`	Returns `true` if sprite `a` overlaps sprite `b`.	`a` and `b` are `Sprite` structures

(Tip: You can use `checkCollision()` not only for player–wall interactions but also for collecting coins, hitting enemies, or triggering effects.)

Narrative Transition

Now your NanoBoy games have rules of physics and interaction. You can make characters stop at walls, pick up items, bounce balls, or trigger sounds. In the next section you'll learn how to combine all the pieces — graphics, input, sound, sprites and colli-

sions — into full games like Snake or Tic-Tac-Toe.

Your First Complete Game: Tic-Tac-Toe

You've learned to draw shapes, read buttons, beep, and even detect collisions. Now it's time to put everything together and make a **real game**. We'll start with a classic: Tic-Tac-Toe. This will give you practice with drawing a grid, moving a cursor, reading button input, and storing game state.

Planning the Game

We need:

- A 3×3 grid on the screen.
- A way to move a cursor over the squares with the arrow buttons.
- A button to place "X" or "O" in the square.
- Logic to alternate turns and check for a win.

(Tip box: Before coding, draw your grid and label how the cursor will move. Planning saves time.)

Code: Tic-Tac-Toe on NanoBoy

```
#include <NanoBoy.h>

NanoBoy nb;

int grid[3][3]; // 0 = empty, 1 = X, 2 = O
int currentPlayer = 1;
int cursorX = 0;
int cursorY = 0;

void setup() {
  nb.begin();
}

void loop() {
  // Handle input
  if (nb.buttonPressed(BTN_LEFT) && cursorX > 0)  cursorX--;
```

```
  if (nb.buttonPressed(BTN_RIGHT) && cursorX < 2) cursorX++;
  if (nb.buttonPressed(BTN_UP) && cursorY > 0)    cursorY--;
  if (nb.buttonPressed(BTN_DOWN) && cursorY < 2)  cursorY++;

  // Place mark
  if (nb.buttonPressed(BTN_A) && grid[cursorY][cursorX] == 0) {
    grid[cursorY][cursorX] = currentPlayer;
    currentPlayer = (currentPlayer == 1) ? 2 : 1;
    nb.beep(600, 50);
  }

  nb.clear();

  // Draw grid lines
  for (int i = 1; i < 3; i++) {
    nb.drawLine(i * 42, 0, i * 42, 63); // vertical lines
    nb.drawLine(0, i * 21, 128, i * 21); // horizontal lines
  }

  // Draw marks
  for (int y = 0; y < 3; y++) {
    for (int x = 0; x < 3; x++) {
      int cell = grid[y][x];
      int px = x * 42 + 10;
      int py = y * 21 + 5;
      if (cell == 1) {
        nb.drawText(px, py, "X");
      } else if (cell == 2) {
        nb.drawText(px, py, "O");
      }
```

```
      }
    }

    // Draw cursor rectangle
    nb.drawRect(cursorX * 42, cursorY * 21, 42, 21);

    nb.display();
  }
```

Upload this. You'll see a 3×3 grid. Move the cursor with arrow buttons, press A to place your mark. The marks alternate between X and O.

New Concepts Used

We didn't introduce new NanoBoy commands here, but you've combined these:

Command	How it's used here
`nb.drawLine()`	Draws the grid lines.
`nb.drawText()`	Draws X and O in each cell.
`nb.drawRect()`	Highlights the current cell with a rectangle cursor.
`nb.buttonPressed()`	Reads input to move cursor and place marks.
`nb.beep()`	Plays a sound when placing a mark.

And you added your own **game logic** with arrays and variables.

Narrative Transition

With just a few dozen lines of code you've built your first complete game. You learned how to mix NanoBoy's drawing functions with your own logic to make an interactive program. In the next chapter we'll build a more dynamic game — Snake — to practice moving sprites, growing arrays, and using collision detection for gameplay.

Game Two: Snake

If Tic-Tac-Toe taught you how to read input and draw on a grid, **Snake** will teach you how to move an object smoothly, grow a body, and detect collisions with walls or food. It's a perfect next step with NanoBoy.

Planning the Game

Snake is a moving line of segments (the body) that grows each time it eats food. You control the direction. The game ends when you hit the wall or your own body.

We'll do a simplified version:

- The playing field is a grid of 8×8 tiles (NanoBoy's default tile size).
- The snake's body is stored as an array of coordinates.
- We'll draw a simple square for each body segment.
- We'll draw a dot for the food.
- You'll use the arrow buttons to steer.

Code: Snake on NanoBoy

```
#include <NanoBoy.h>

NanoBoy nb;

const int TILE = 8;               // size of each cell
const int GRID_W = SCREEN_WIDTH / TILE;
const int GRID_H = SCREEN_HEIGHT / TILE;

int snakeX[64], snakeY[64];       // maximum 64 segments
int snakeLength = 3;
int dirX = 1, dirY = 0;

int foodX, foodY;
unsigned long lastMove = 0;

void placeFood() {
  foodX = random(0, GRID_W);
  foodY = random(0, GRID_H);
}

void setup() {
  nb.begin();
```

```
  randomSeed(analogRead(0));

  // initialise snake in middle
  for (int i = 0; i < snakeLength; i++) {
    snakeX[i] = GRID_W / 2 - i;
    snakeY[i] = GRID_H / 2;
  }
  placeFood();
}

void loop() {
  // read input
  if (nb.buttonPressed(BTN_UP) && dirY == 0)    { dirX = 0; dirY = -1; }
  if (nb.buttonPressed(BTN_DOWN) && dirY == 0)  { dirX = 0; dirY = 1; }
  if (nb.buttonPressed(BTN_LEFT) && dirX == 0)  { dirX = -1; dirY = 0; }
  if (nb.buttonPressed(BTN_RIGHT) && dirX == 0) { dirX = 1; dirY = 0; }

  // move snake every 150 ms
  if (millis() - lastMove > 150) {
    lastMove = millis();

    // move body
    for (int i = snakeLength - 1; i > 0; i--) {
      snakeX[i] = snakeX[i - 1];
      snakeY[i] = snakeY[i - 1];
    }
```

```
    // move head
    snakeX[0] += dirX;
    snakeY[0] += dirY;

    // check collision with walls
    if (snakeX[0] < 0 || snakeX[0] >= GRID_W ||
        snakeY[0] < 0 || snakeY[0] >= GRID_H) {
      nb.beep(200, 500); // game over sound
      snakeLength = 3;
      dirX = 1; dirY = 0;
      for (int i = 0; i < snakeLength; i++) {
        snakeX[i] = GRID_W / 2 - i;
        snakeY[i] = GRID_H / 2;
      }
      placeFood();
    }

    // check collision with food
    if (snakeX[0] == foodX && snakeY[0] == foodY) {
      if (snakeLength < 64) snakeLength++;
      nb.beep(800, 50);
      placeFood();
    }
  }

  // draw everything
  nb.clear();

  // draw food
  nb.fillRect(foodX * TILE, foodY * TILE, TILE, TILE);
```

```
    // draw snake
    for (int i = 0; i < snakeLength; i++) {
      nb.drawRect(snakeX[i] * TILE, snakeY[i] * TILE, TILE, TILE);
    }

    nb.display();
  }
```

Upload it. You'll see a small snake moving across the screen. Steer with the arrow buttons; eat the dot to grow. If you hit the wall, it resets with a beep.

Commands and Concepts Used

Command	How it's used here
`nb.fillRect()`	Draws the food tile.
`nb.drawRect()`	Draws each snake segment.
`nb.buttonPressed()`	Reads arrow input to change direction.
`nb.beep()`	Plays sound when eating or crashing.
`nb.clear()`/`nb.display()`	Clears and updates the screen each frame.

And you learned:

- How to store a moving object as an array of coordinates.
- How to update it every few milliseconds to control speed.
- How to check for collisions with walls or food

Narrative Transition

With Snake you've built your first animated, continuously moving game. You're now combining NanoBoy's drawing, input, sound, and a little timing logic to create real gameplay. From here you can add score counters, high-speed levels, or even detect collisions with the snake's own body to make it more challenging.