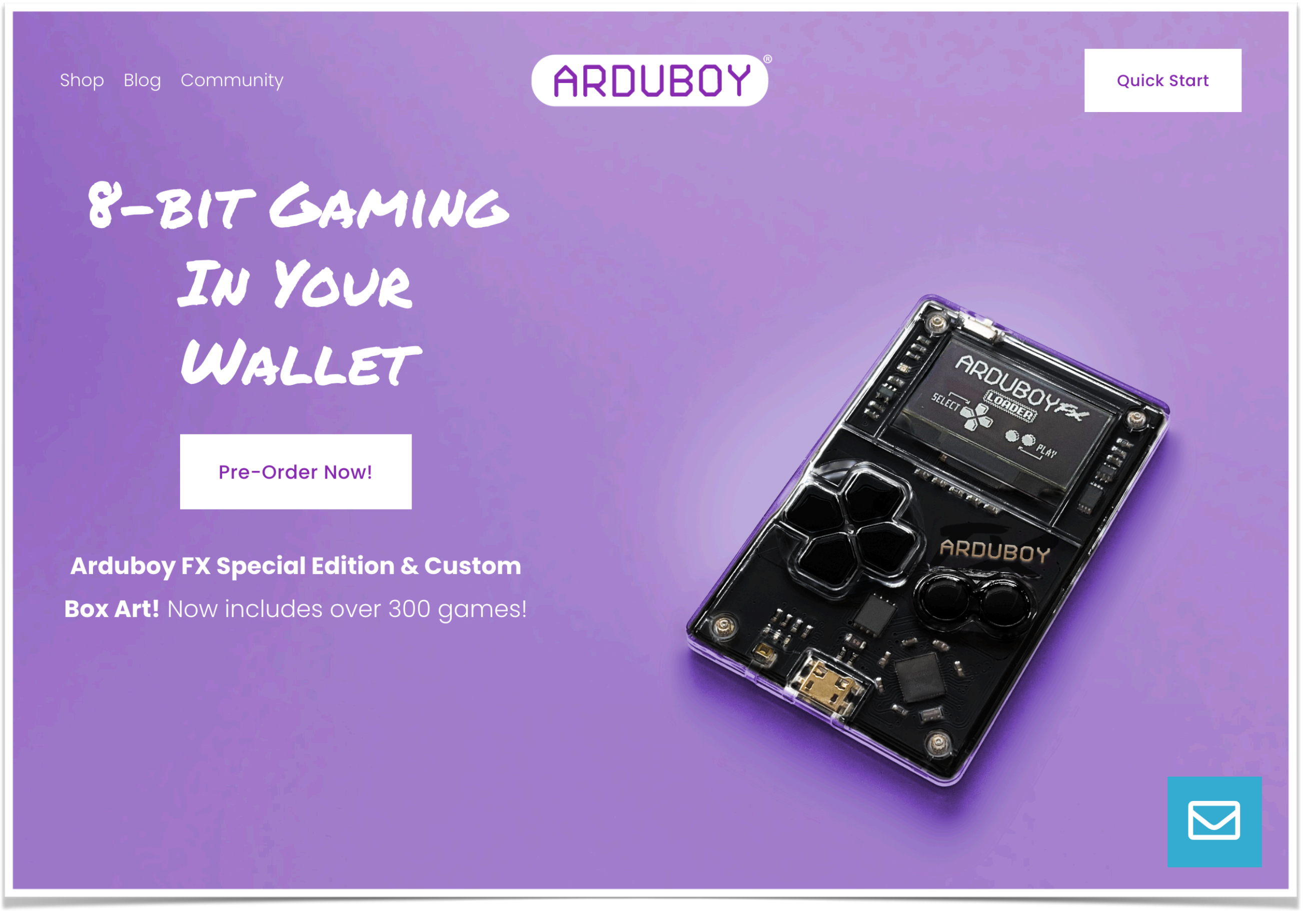
Arduinix

A game with 151 byte stack

Jeremy Bridon, Spring 2024

Inspiration



Shop Blog Community

ARDUBOY®

Quick Start

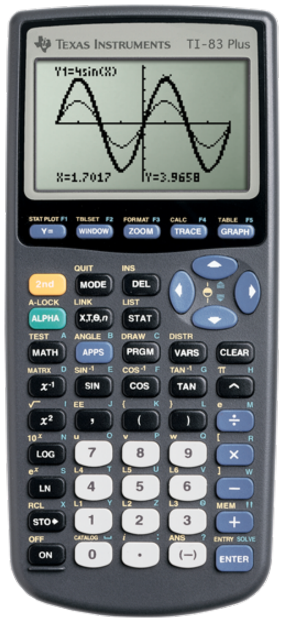
8-BIT GAMING IN YOUR WALLET

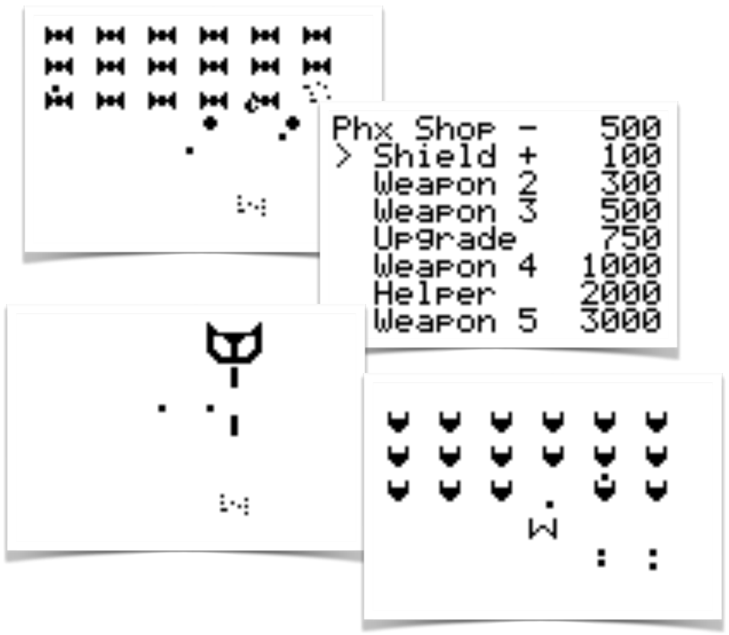
Pre-Order Now!

Arduboy FX Special Edition & Custom Box Art! Now includes over 300 games!

Inspiration

- Programming challenge: write a game with tiny memory constraints
- TI-83+ games: Phoenix
  - 64 x 96 Pixels, 24 KB RAM
  - Z80 Processor, 6 MHz, 1976





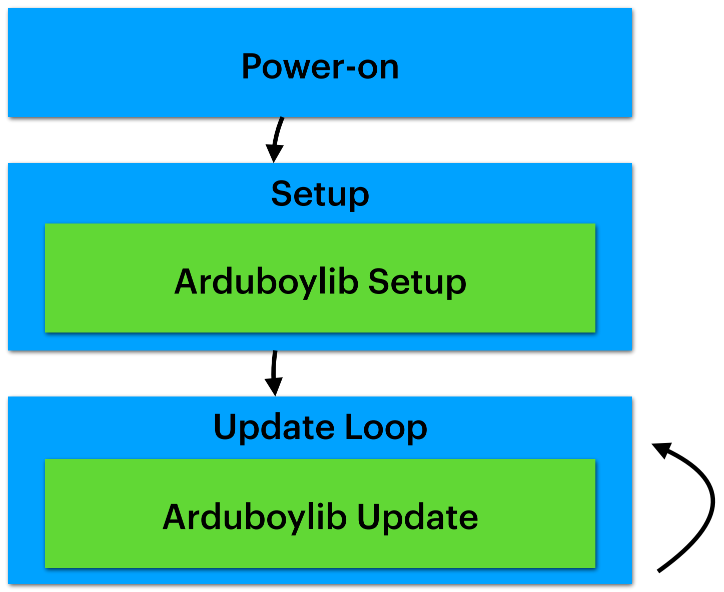
```
Phx Shop -    500
> Shield +    100
  Weapon 2    300
  Weapon 3    500
  Upgrade     750
  Weapon 4   1000
  Helper     2000
  Weapon 5   3000
```

Inspiration

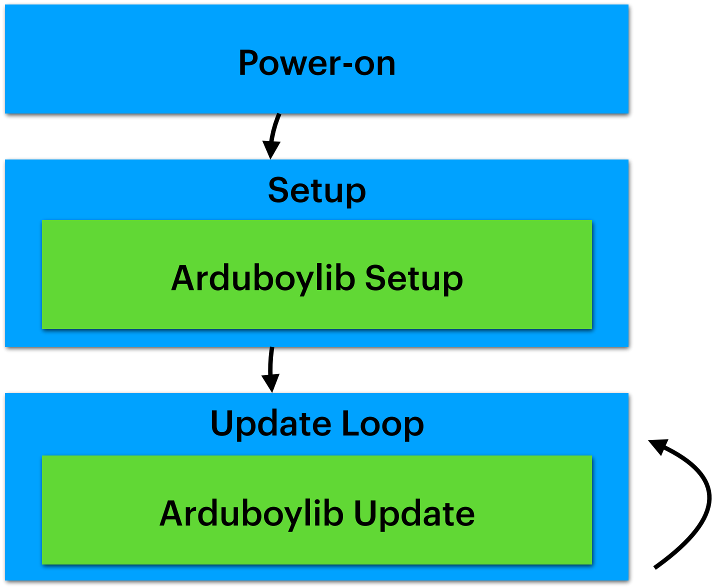
Arduboy

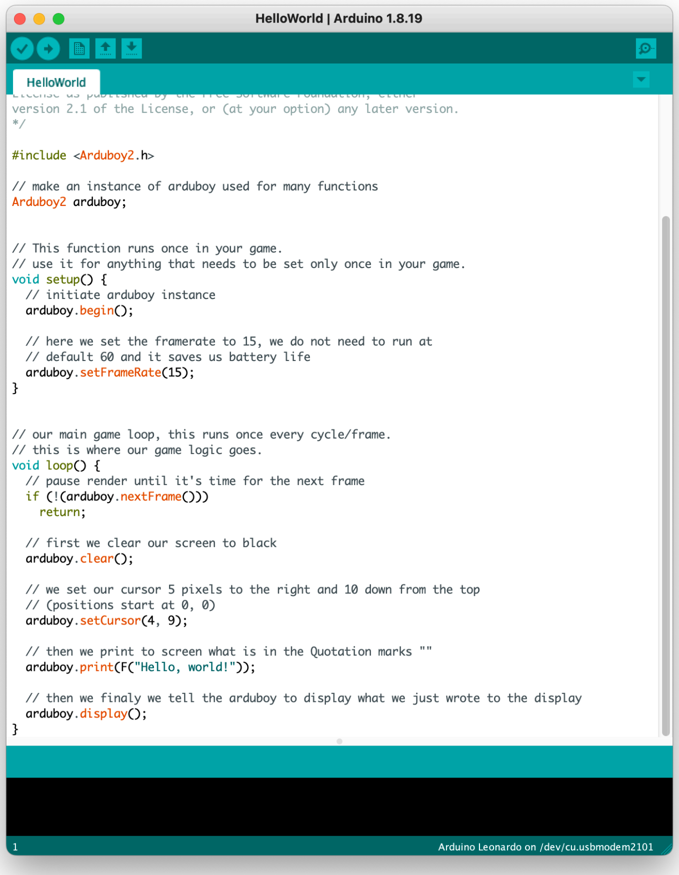
- ATMega32u4 8-bit 8Mhz RISC microcontroller
  - 32 X 8-bit registers
  - 2.5 KB RAM, 32 KB Flash ROM
  - No branch prediction, no pipelining
- 128x64px 1-bit OLED, 40Hz update
- Based on Arduino - C / C++ / Processing, IDE

Normal runtime



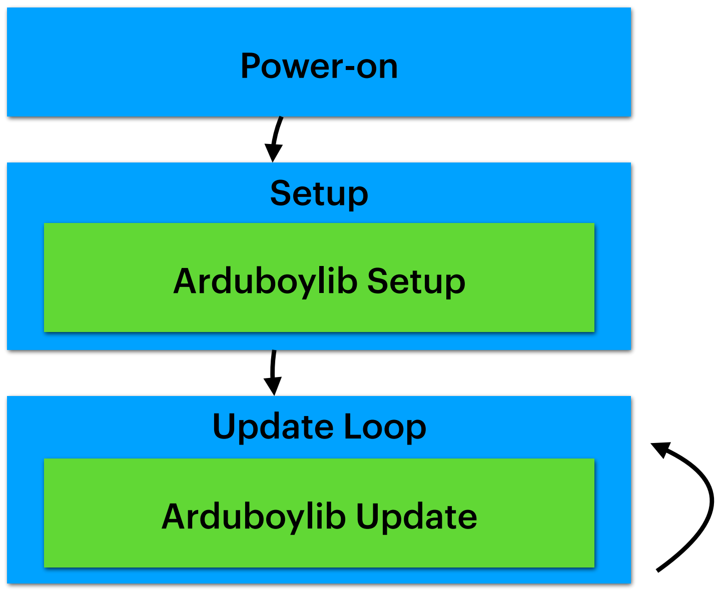
Normal runtime





```
version 2.1 of the License, or (at your option) any later version.
*/

#include <Arduboy2.h>

// make an instance of arduboy used for many functions
Arduboy2 arduboy;


// This function runs once in your game.
// use it for anything that needs to be set only once in your game.
void setup() {
  // initiate arduboy instance
  arduboy.begin();

  // here we set the framerate to 15, we do not need to run at
  // default 60 and it saves us battery life
  arduboy.setFrameRate(15);
}


// our main game loop, this runs once every cycle/frame.
// this is where our game logic goes.
void loop() {
  // pause render until it's time for the next frame
  if (!(arduboy.nextFrame()))
    return;

  // first we clear our screen to black
  arduboy.clear();

  // we set our cursor 5 pixels to the right and 10 down from the top
  // (positions start at 0, 0)
  arduboy.setCursor(4, 9);

  // then we print to screen what is in the Quotation marks ""
  arduboy.print(F("Hello, world!"));

  // then we finaly we tell the arduboy to display what we just wrote to the display
  arduboy.display();
}
```

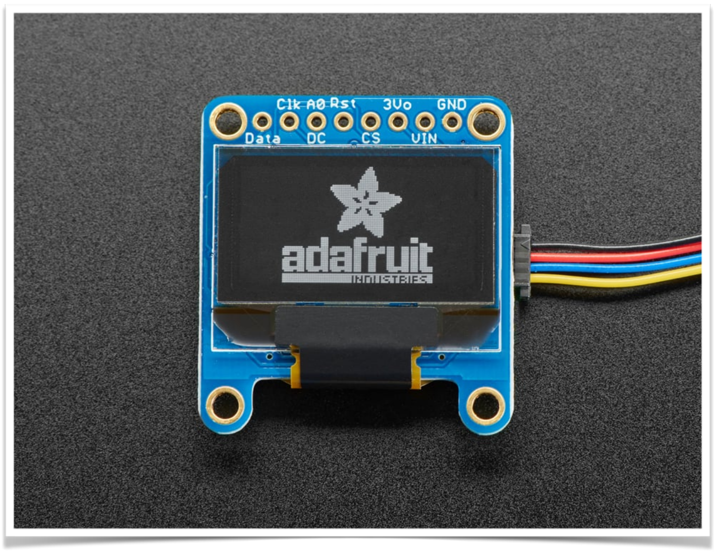
Normal runtime

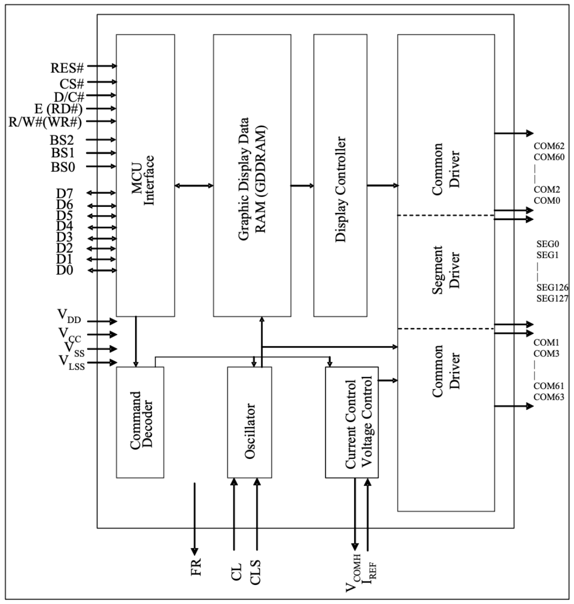


```
#include <Arduboy2.h>
Arduboy2 arduboy;

void setup() {
  arduboy.begin();
}
void loop() {
  arduboy.clear();
  arduboy.setCursor(4, 9);
  arduboy.print(F("Hello, world!"));
  arduboy.display();
}
```

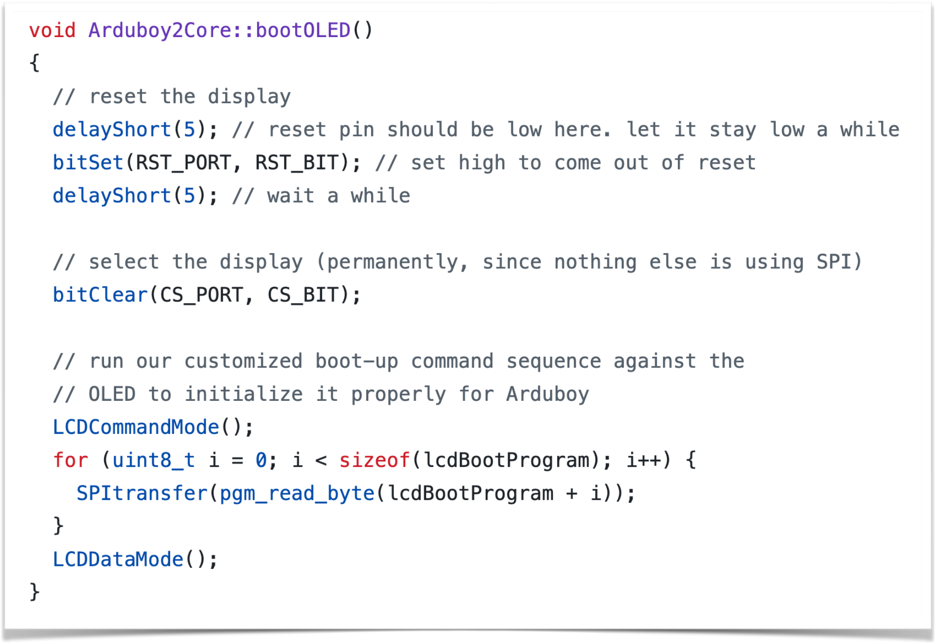
Drawing?

Monochrome 0.96" 128x64 OLED Graphic Display

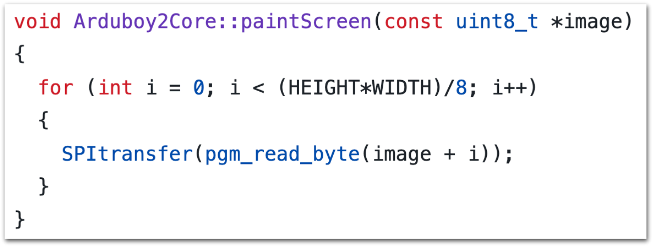




Drawing?



```
void Arduboy2Core::bootOLED()
{
  // reset the display
  delayShort(5); // reset pin should be low here. let it stay low a while
  bitSet(RST_PORT, RST_BIT); // set high to come out of reset
  delayShort(5); // wait a while

  // select the display (permanently, since nothing else is using SPI)
  bitClear(CS_PORT, CS_BIT);

  // run our customized boot-up command sequence against the
  // OLED to initialize it properly for Arduboy
  LCDCommandMode();
  for (uint8_t i = 0; i < sizeof(lcdBootProgram); i++) {
    SPItransfer(pgm_read_byte(lcdBootProgram + i));
  }
  LCDDataMode();
}
```



```
void Arduboy2Core::paintScreen(const uint8_t *image)
{
  for (int i = 0; i < (HEIGHT*WIDTH)/8; i++)
  {
    SPItransfer(pgm_read_byte(image + i));
  }
}
```

First Challenge:

“Global variables use 1197 bytes (46%) of dynamic memory, leaving 1363 bytes for local variables. Maximum is 2560 bytes.”

“No-code” binary is 149 bytes

Arduboy2 takes 1,048 bytes!

Easy fix

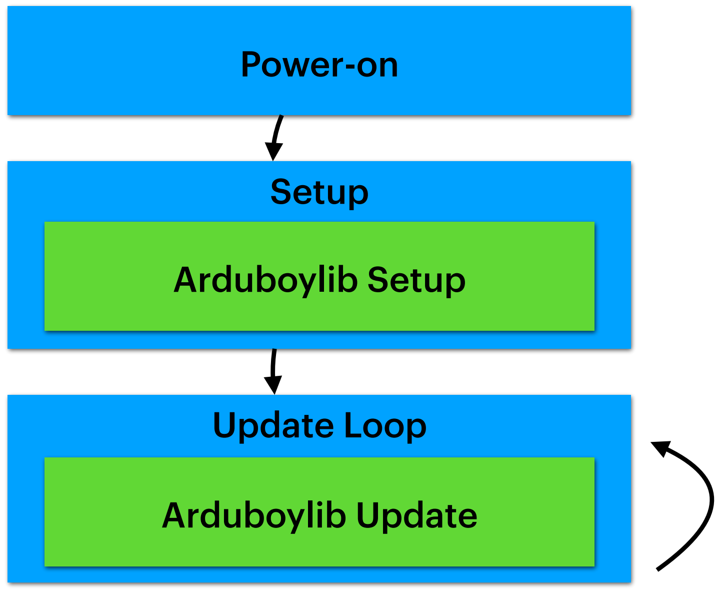
- Slim Arduboy2 lib down
  - Clone repo
  - Remove unnecessary text rendering
    - *Personally offended*
  - Slimmed down to ~500 bytes
    - ~2,000 bytes left!

Second challenge:

- Arduino IDE…
  - Awful text editor: switch to Vim / TextEdit?
  - Iteration time is about 30 seconds
    - Not too bad, but has no debugger, and flakey serial I/O

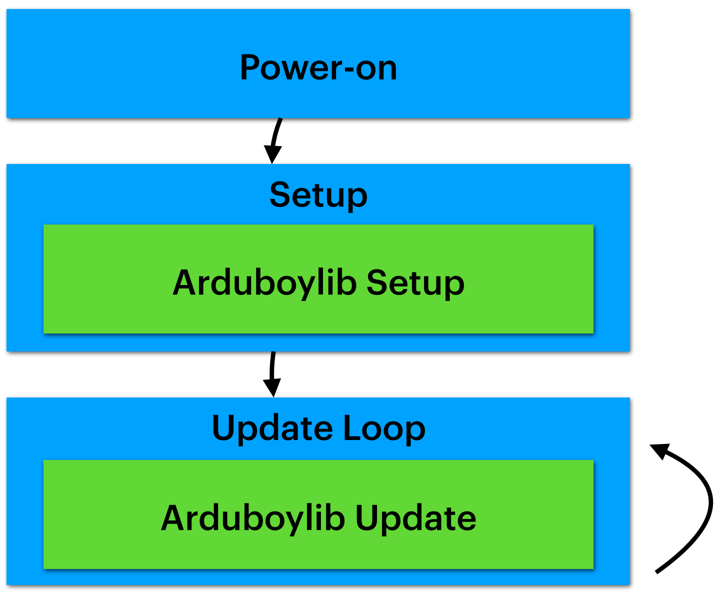
Harder fix:

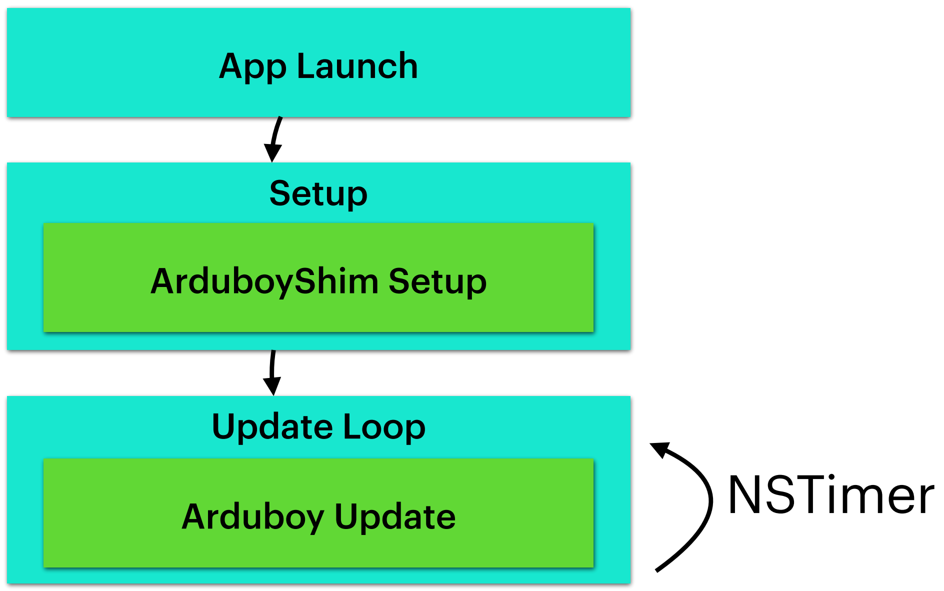
- Build an AppKit app that “simulates” the platform



Harder fix:

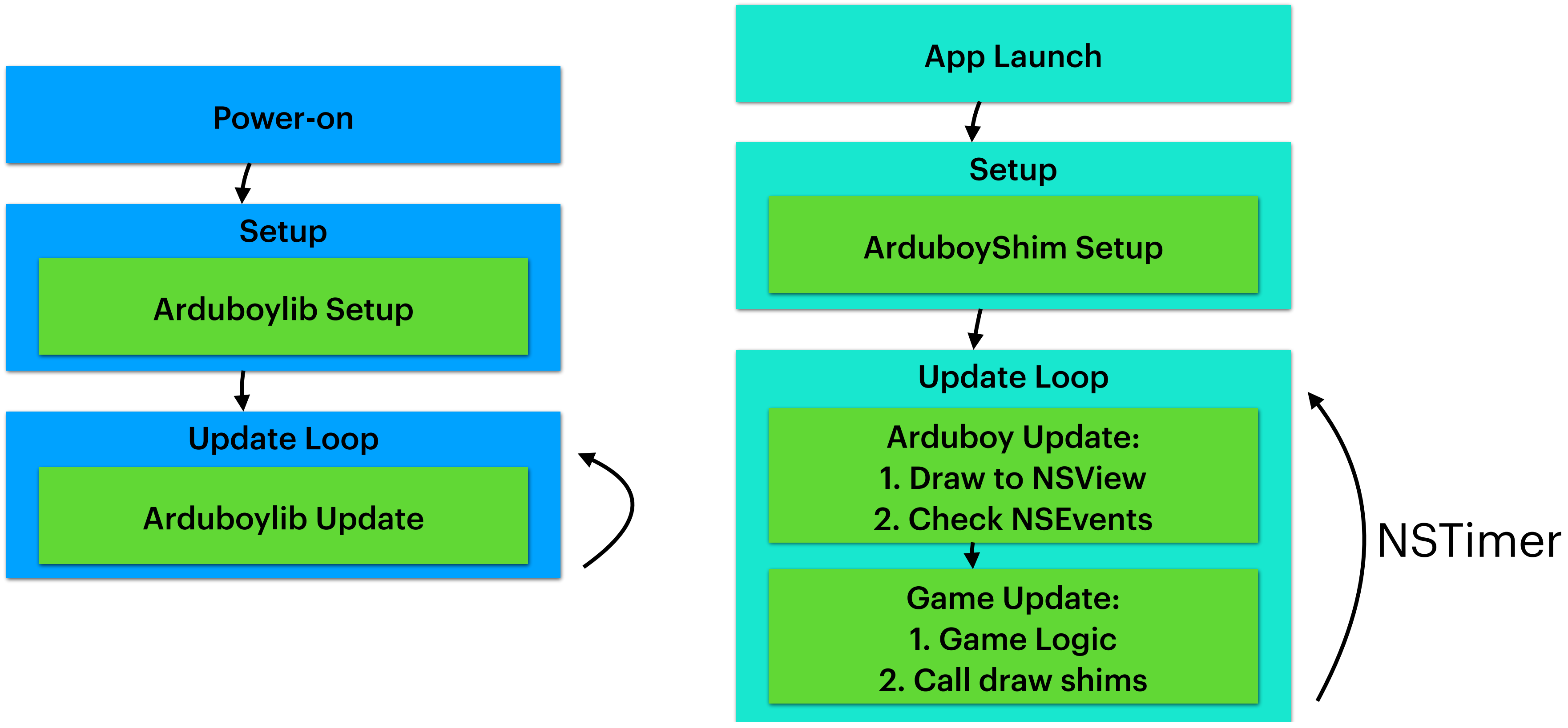
- Build an AppKit app that “simulates” the platform



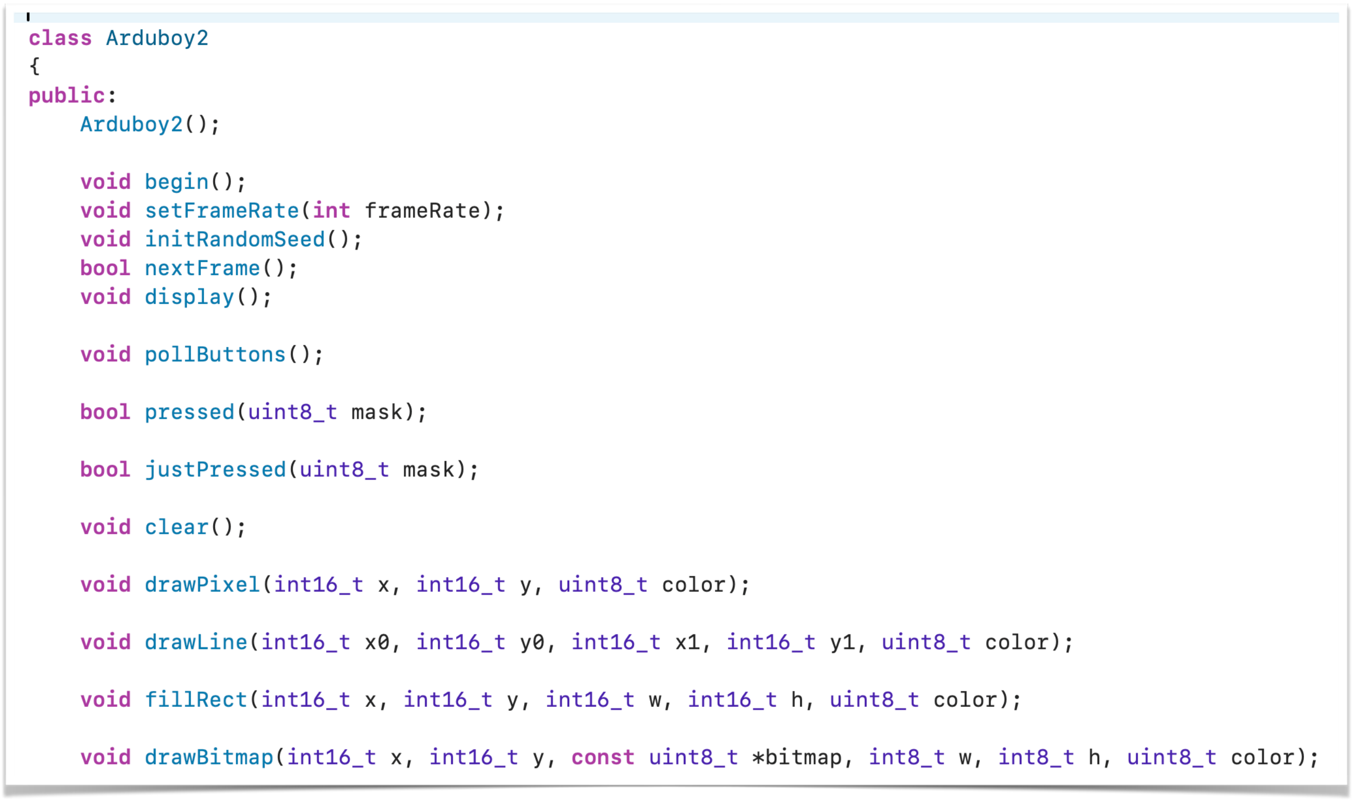


Harder fix:

- Build an AppKit app that “simulates” the platform

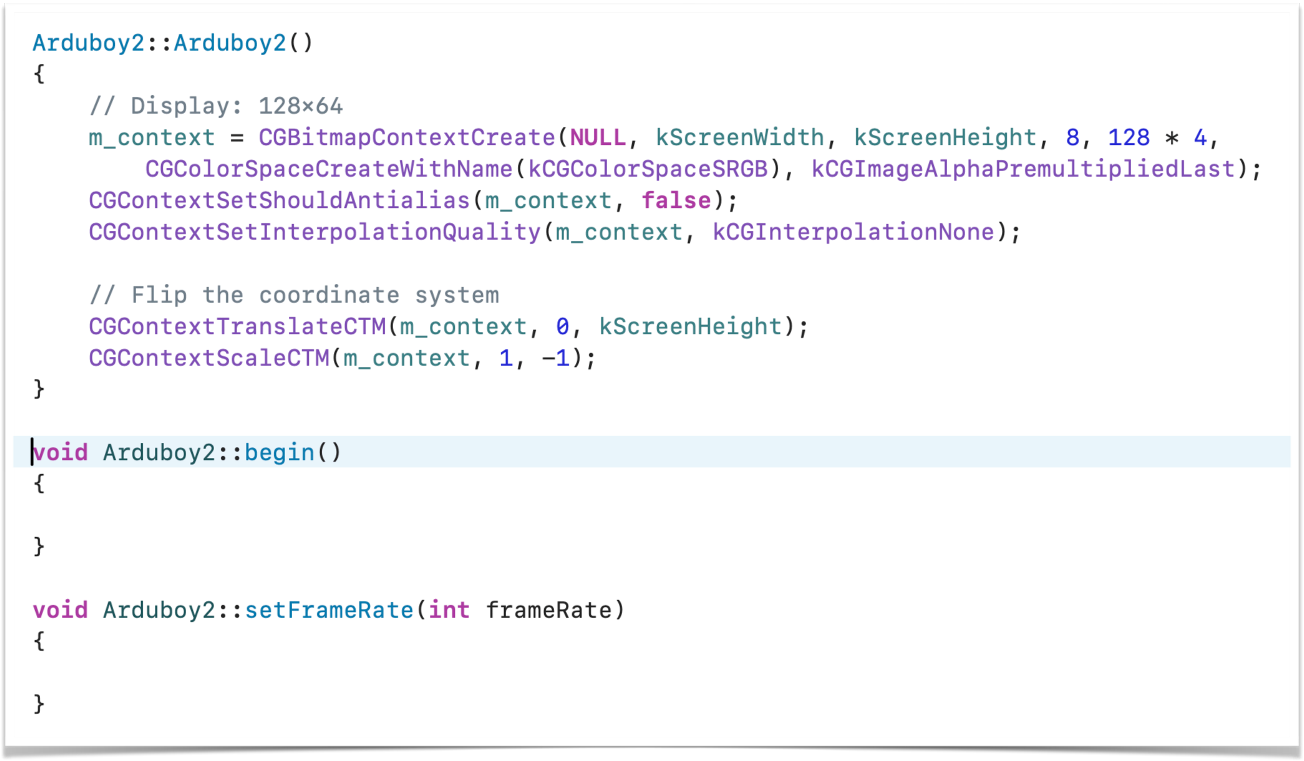


Harder fix:



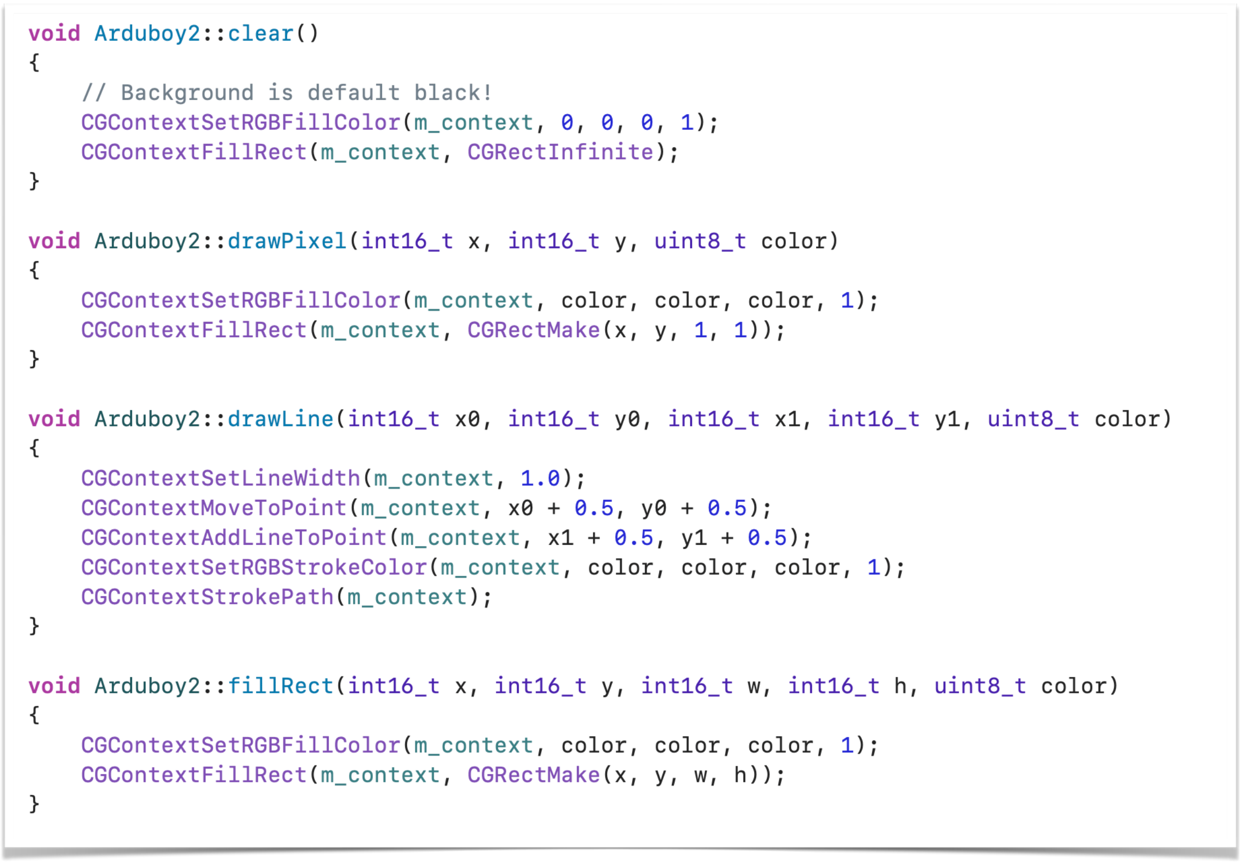
```
class Arduboy2
{
public:
    Arduboy2();

    void begin();
    void setFrameRate(int frameRate);
    void initRandomSeed();
    bool nextFrame();
    void display();

    void pollButtons();

    bool pressed(uint8_t mask);

    bool justPressed(uint8_t mask);

    void clear();

    void drawPixel(int16_t x, int16_t y, uint8_t color);

    void drawLine(int16_t x0, int16_t y0, int16_t x1, int16_t y1, uint8_t color);

    void fillRect(int16_t x, int16_t y, int16_t w, int16_t h, uint8_t color);

    void drawBitmap(int16_t x, int16_t y, const uint8_t *bitmap, int8_t w, int8_t h, uint8_t color);
```

Harder fix:



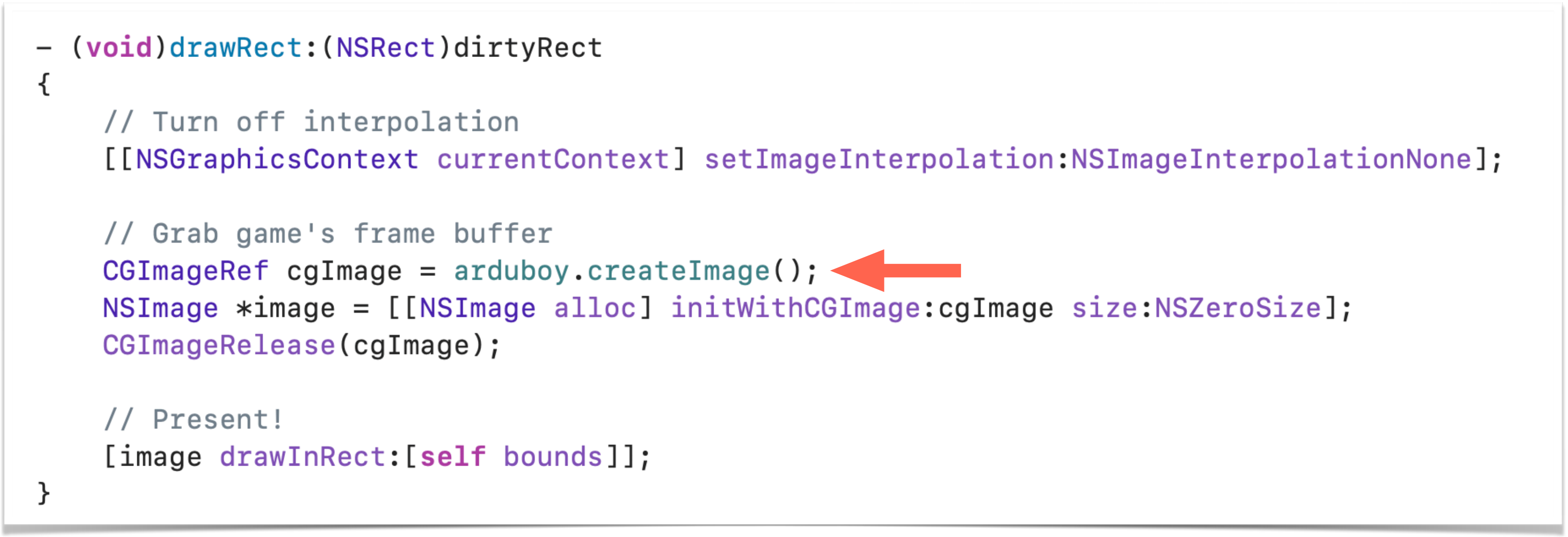
```
Arduboy2::Arduboy2()
{
    // Display: 128x64
    m_context = CGBitmapContextCreate(NULL, kScreenWidth, kScreenHeight, 8, 128 * 4,
        CGColorSpaceCreateWithName(kCGColorSpaceSRGB), kCGImageAlphaPremultipliedLast);
    CGContextSetShouldAntialias(m_context, false);
    CGContextSetInterpolationQuality(m_context, kCGInterpolationNone);

    // Flip the coordinate system
    CGContextTranslateCTM(m_context, 0, kScreenHeight);
    CGContextScaleCTM(m_context, 1, -1);
}

void Arduboy2::begin()
{

}

void Arduboy2::setFrameRate(int frameRate)
{

}
```

Harder fix:



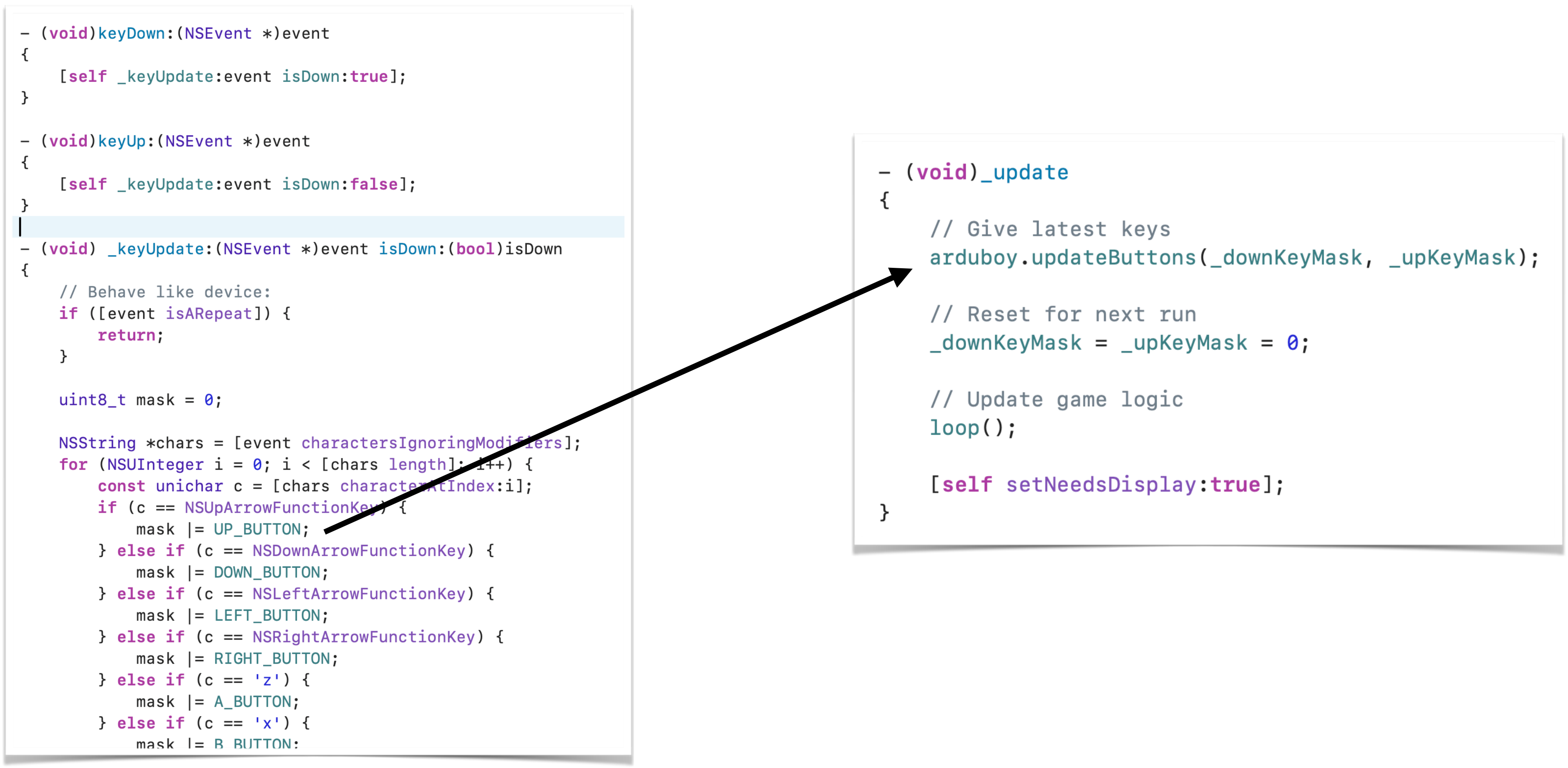
```
void Arduboy2::clear()
{
    // Background is default black!
    CGContextSetRGBFillColor(m_context, 0, 0, 0, 1);
    CGContextFillRect(m_context, CGRectInfinite);
}

void Arduboy2::drawPixel(int16_t x, int16_t y, uint8_t color)
{
    CGContextSetRGBFillColor(m_context, color, color, color, 1);
    CGContextFillRect(m_context, CGRectMake(x, y, 1, 1));
}

void Arduboy2::drawLine(int16_t x0, int16_t y0, int16_t x1, int16_t y1, uint8_t color)
{
    CGContextSetLineWidth(m_context, 1.0);
    CGContextMoveToPoint(m_context, x0 + 0.5, y0 + 0.5);
    CGContextAddLineToPoint(m_context, x1 + 0.5, y1 + 0.5);
    CGContextSetRGBStrokeColor(m_context, color, color, color, 1);
    CGContextStrokePath(m_context);
}

void Arduboy2::fillRect(int16_t x, int16_t y, int16_t w, int16_t h, uint8_t color)
{
    CGContextSetRGBFillColor(m_context, color, color, color, 1);
    CGContextFillRect(m_context, CGRectMake(x, y, w, h));
}
```

Harder fix:



```objc
- (void)drawRect:(NSRect)dirtyRect
{
    // Turn off interpolation
    [[NSGraphicsContext currentContext] setImageInterpolation:NSImageInterpolationNone];

    // Grab game's frame buffer
    CGImageRef cgImage = arduboy.createImage();
    NSImage *image = [[NSImage alloc] initWithCGImage:cgImage size:NSZeroSize];
    CGImageRelease(cgImage);

    // Present!
    [image drawInRect:[self bounds]];
}
```

Harder fix:

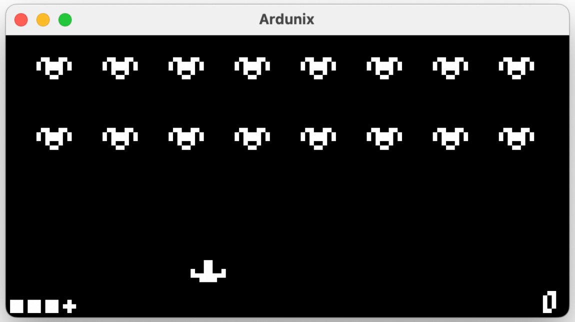


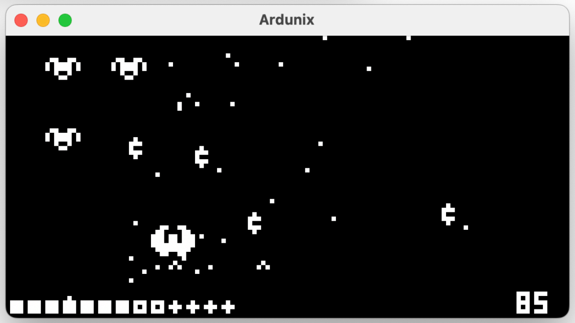
```
- (void)keyDown:(NSEvent *)event
{
    [self _keyUpdate:event isDown:true];
}

- (void)keyUp:(NSEvent *)event
{
    [self _keyUpdate:event isDown:false];
}

- (void) _keyUpdate:(NSEvent *)event isDown:(bool)isDown
{
    // Behave like device:
    if ([event isARepeat]) {
        return;
    }

    uint8_t mask = 0;

    NSString *chars = [event charactersIgnoringModifiers];
    for (NSUInteger i = 0; i < [chars length]; i++) {
        const unichar c = [chars characterAtIndex:i];
        if (c == NSUpArrowFunctionKey) {
            mask |= UP_BUTTON;
        } else if (c == NSDownArrowFunctionKey) {
            mask |= DOWN_BUTTON;
        } else if (c == NSLeftArrowFunctionKey) {
            mask |= LEFT_BUTTON;
        } else if (c == NSRightArrowFunctionKey) {
            mask |= RIGHT_BUTTON;
        } else if (c == 'z') {
            mask |= A_BUTTON;
        } else if (c == 'x') {
            mask |= B_BUTTON;
```

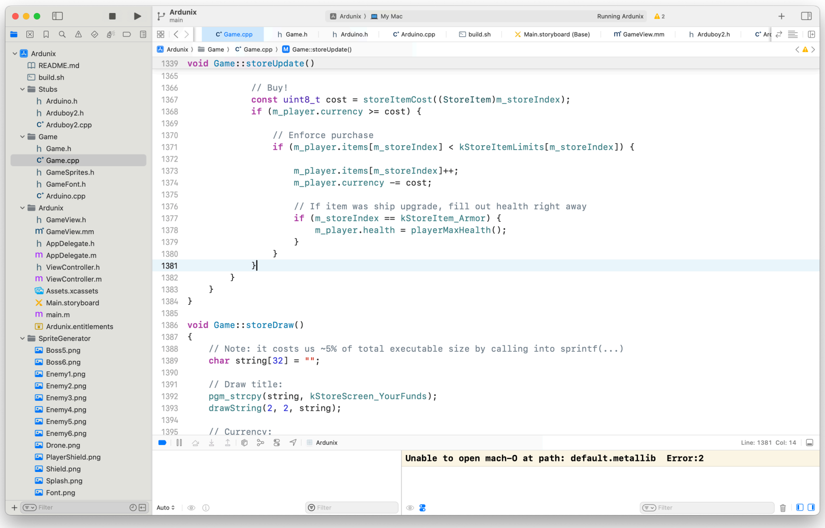
```
- (void)_update
{
    // Give latest keys
    arduboy.updateButtons(_downKeyMask, _upKeyMask);

    // Reset for next run
    _downKeyMask = _upKeyMask = 0;

    // Update game logic
    loop();

    [self setNeedsDisplay:true];
}
```

Harder fix:

- 3 second iteration time
  - And a debugger!
  - … but no memory measurement 😔





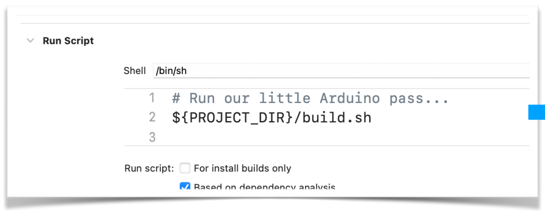


```
void Game::storeUpdate()

            // Buy!
            const uint8_t cost = storeItemCost((StoreItem)m_storeIndex);
            if (m_player.currency >= cost) {

                // Enforce purchase
                if (m_player.items[m_storeIndex] < kStoreItemLimits[m_storeIndex]) {

                    m_player.items[m_storeIndex]++;
                    m_player.currency -= cost;

                    // If item was ship upgrade, fill out health right away
                    if (m_storeIndex == kStoreItem_Armor) {
                        m_player.health = playerMaxHealth();
                    }
                }
            }
        }
    }
}

void Game::storeDraw()
{
    // Note: it costs us ~5% of total executable size by calling into sprintf(...)
    char string[32] = "";

    // Draw title:
    pgm_strcpy(string, kStoreScreen_YourFunds);
    drawString(2, 2, string);

    // Currency:
```

```
Unable to open mach-O at path: default.metallib  Error:2
```

Third Challenge:

- How can I auto-create the Arduino project?
    - Get quick checks on memory usage
    - Makes sure I don't use a C++ feature Arduino doesn't support, etc.



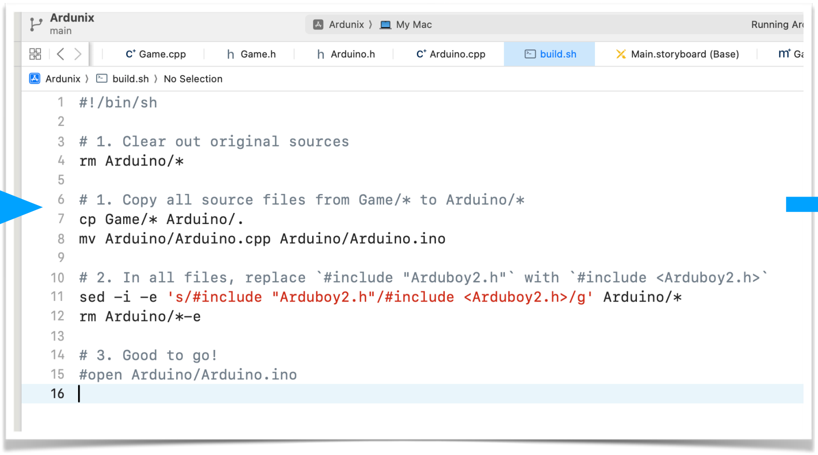
Run Script

Shell: /bin/sh

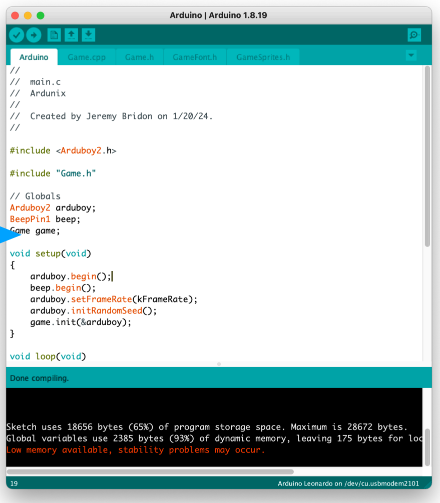
```
# Run our little Arduino pass...
${PROJECT_DIR}/build.sh
```

Run script: [ ] For install builds only

[✓] Based on dependency analysis



```
#!/bin/sh

# 1. Clear out original sources
rm Arduino/*

# 1. Copy all source files from Game/* to Arduino/*
cp Game/* Arduino/.
mv Arduino/Arduino.cpp Arduino/Arduino.ino

# 2. In all files, replace `#include "Arduboy2.h"` with `#include <Arduboy2.h>`
sed -i -e 's/#include "Arduboy2.h"/#include <Arduboy2.h>/g' Arduino/*
rm Arduino/*-e

# 3. Good to go!
#open Arduino/Arduino.ino

```



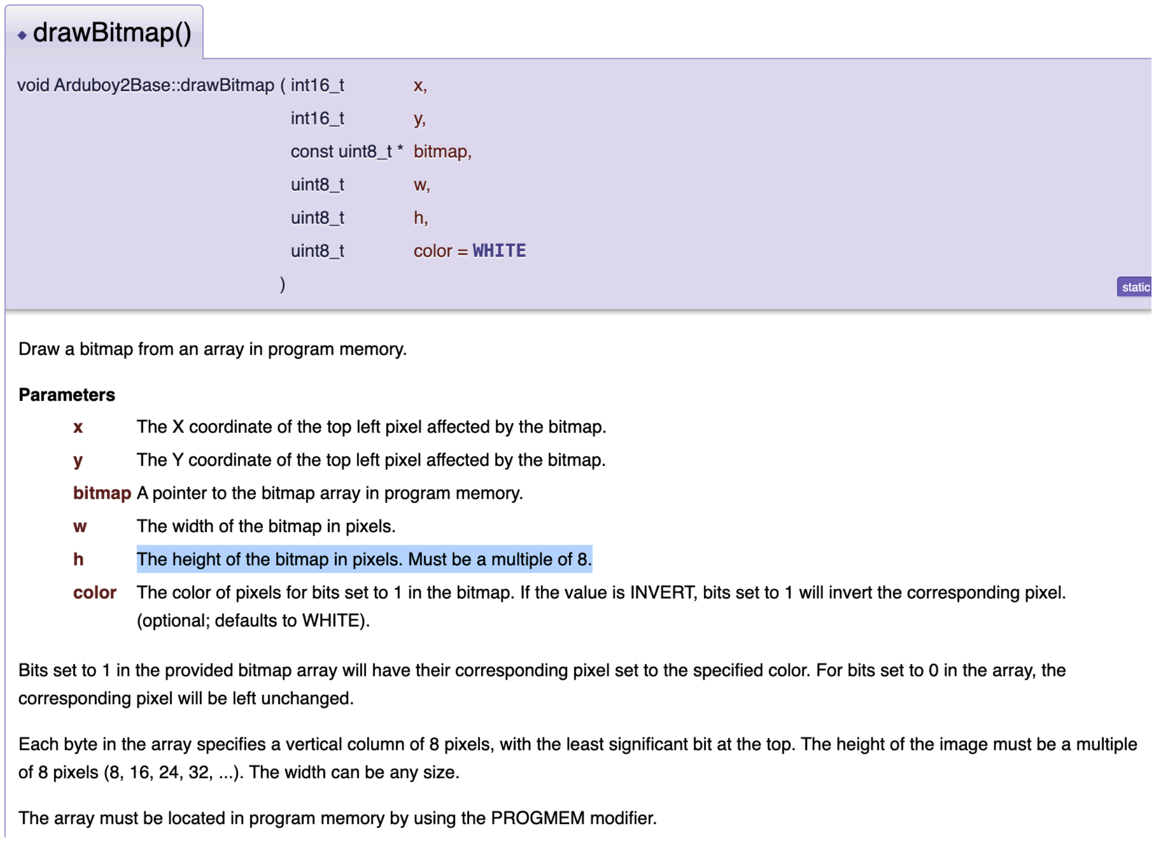
```
//
//  main.c
//  Ardunix
//
//  Created by Jeremy Bridon on 1/20/24.
//

#include <Arduboy2.h>

#include "Game.h"

// Globals
Arduboy2 arduboy;
BeepPin1 beep;
Game game;

void setup(void)
{
    arduboy.begin();
    beep.begin();
    arduboy.setFrameRate(kFrameRate);
    arduboy.initRandomSeed();
    game.init(&arduboy);
}

void loop(void)
```

Done compiling.

```
Sketch uses 18656 bytes (65%) of program storage space. Maximum is 28672 bytes.
Global variables use 2385 bytes (93%) of dynamic memory, leaving 175 bytes for loc
Low memory available, stability problems may occur.
```

Fourth Challenge:

- Sprites!



◆ drawBitmap()

```
void Arduboy2Base::drawBitmap ( int16_t          x,
                                int16_t          y,
                                const uint8_t *  bitmap,
                                uint8_t          w,
                                uint8_t          h,
                                uint8_t          color = WHITE
                              )
```

static

Draw a bitmap from an array in program memory.

Parameters

x	The X coordinate of the top left pixel affected by the bitmap.
y	The Y coordinate of the top left pixel affected by the bitmap.
bitmap	A pointer to the bitmap array in program memory.
w	The width of the bitmap in pixels.
h	The height of the bitmap in pixels. Must be a multiple of 8.
color	The color of pixels for bits set to 1 in the bitmap. If the value is INVERT, bits set to 1 will invert the corresponding pixel. (optional; defaults to WHITE).

Bits set to 1 in the provided bitmap array will have their corresponding pixel set to the specified color. For bits set to 0 in the array, the corresponding pixel will be left unchanged.

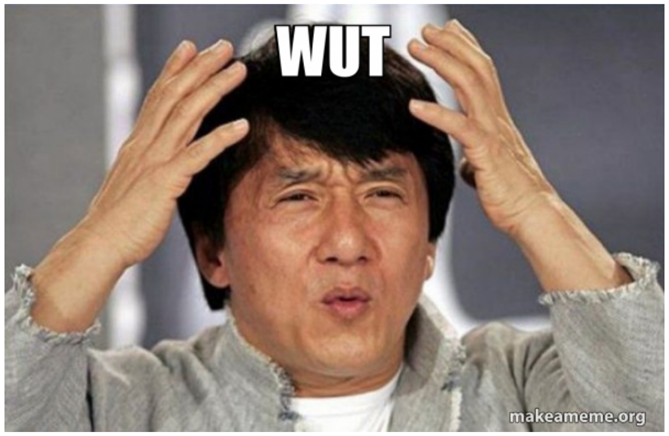
Each byte in the array specifies a vertical column of 8 pixels, with the least significant bit at the top. The height of the image must be a multiple of 8 pixels (8, 16, 24, 32, ...). The width can be any size.

The array must be located in program memory by using the PROGMEM modifier.

Fourth Challenge:

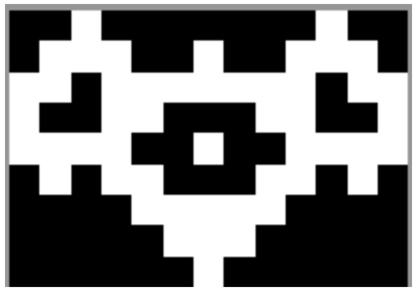
- Sprites!

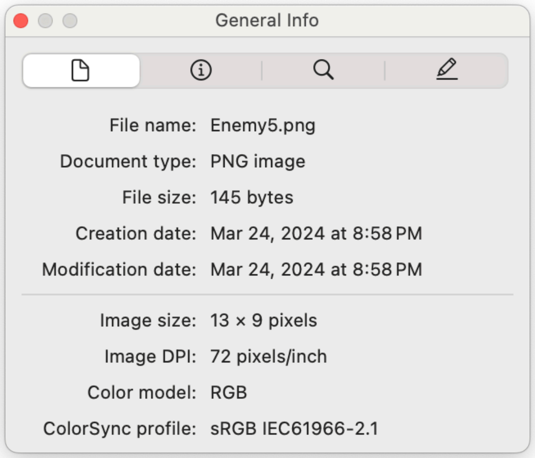
> Each byte in the array specifies a vertical column of 8 pixels, with the least significant bit at the top. The height of the image must be a multiple of 8 pixels (8, 16, 24, 32, ...). The width can be any size.



Sprites!

Part one: encoding





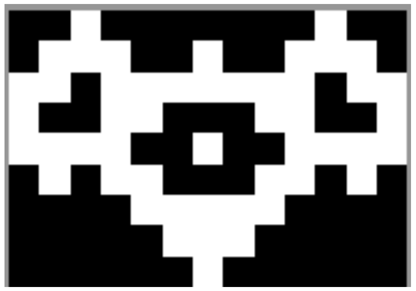
File name: Enemy5.png
Document type: PNG image
File size: 145 bytes
Creation date: Mar 24, 2024 at 8:58 PM
Modification date: Mar 24, 2024 at 8:58 PM

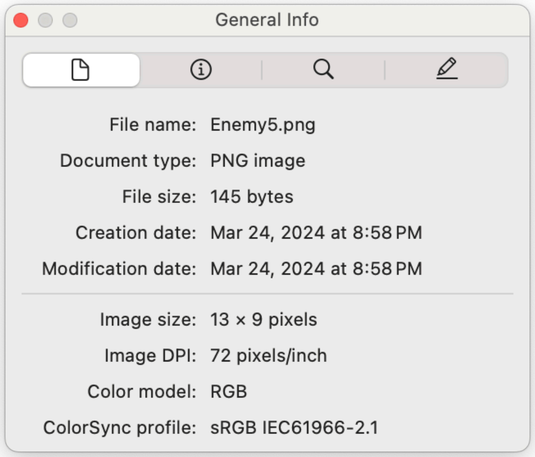
Image size: 13 × 9 pixels
Image DPI: 72 pixels/inch
Color model: RGB
ColorSync profile: sRGB IEC61966-2.1

- Challenge 1: size!
  - 13 x 9 pixels, 4 bytes each = 468 bytes
    - That's nearly 25% of all game-data, um no!
  - 13 x 9 pixels, 1 byte each = 117 bytes
    - ~6%, better!
  - 13 x 9 pixels, 1 bit each = 117 bits, 15 bytes!
    - 0.75% of game data, **GREAT!**

Sprites!

Part one: encoding





General Info

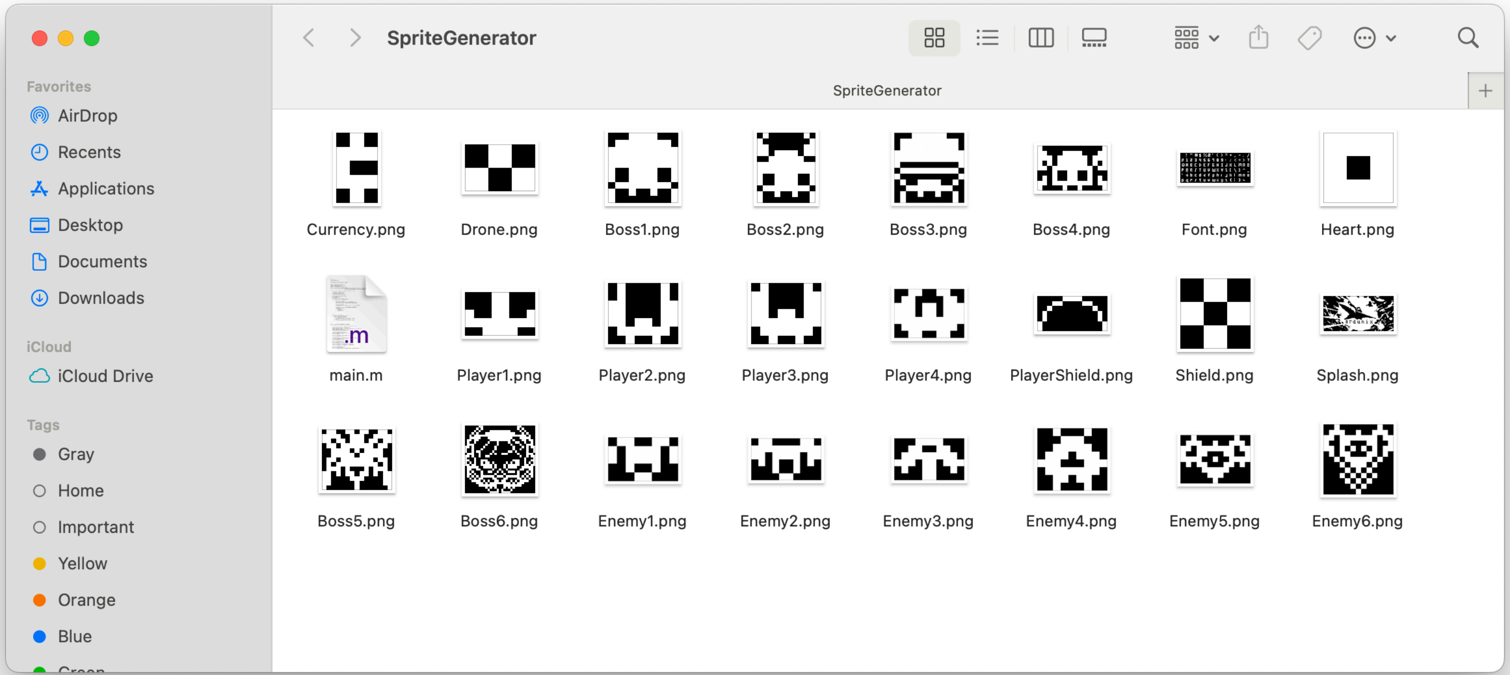
File name: Enemy5.png
Document type: PNG image
File size: 145 bytes
Creation date: Mar 24, 2024 at 8:58 PM
Modification date: Mar 24, 2024 at 8:58 PM

Image size: 13 × 9 pixels
Image DPI: 72 pixels/inch
Color model: RGB
ColorSync profile: sRGB IEC61966-2.1

```
static constexpr uint8_t kEnemy5Width = 13;
static constexpr uint8_t kEnemy5Height = 9;
const uint8_t kEnemy5Bitmap[15] PROGMEM = {
    0x04, 0xc4, 0xc9, 0xed, 0xef, 0x8c, 0xf9,
    0xe4, 0x35, 0x16, 0x7c, 0x00, 0x07, 0x40, 0x00};
```

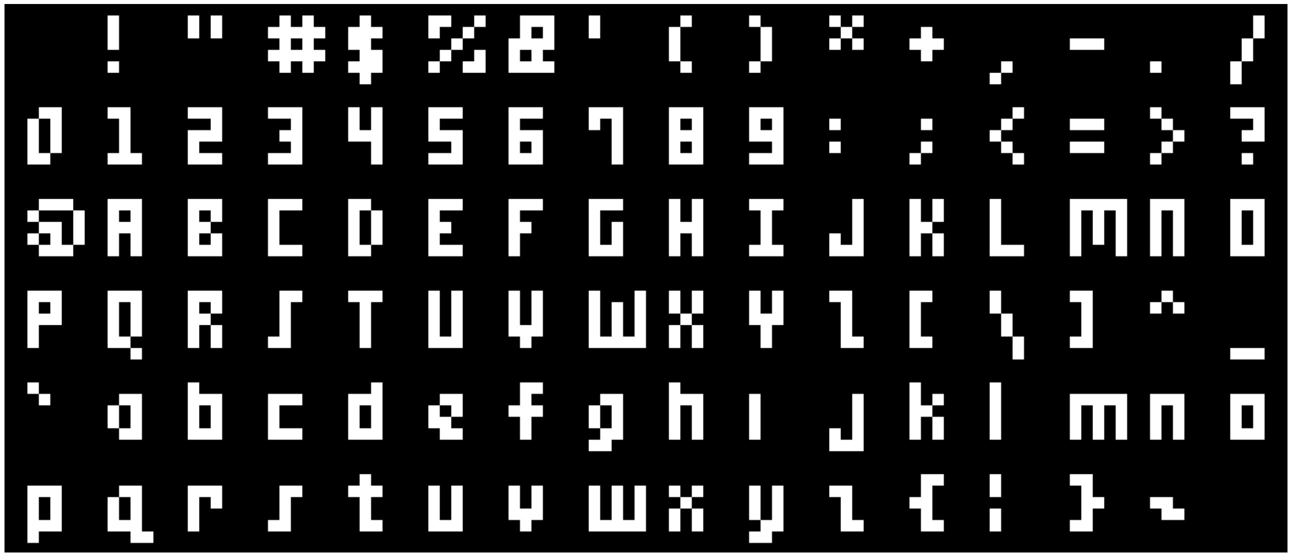
Sprites!

Part deux: automation?



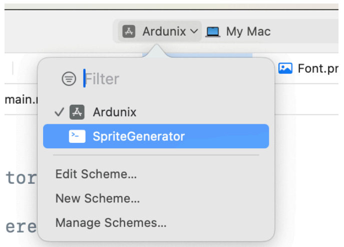
Sprites!

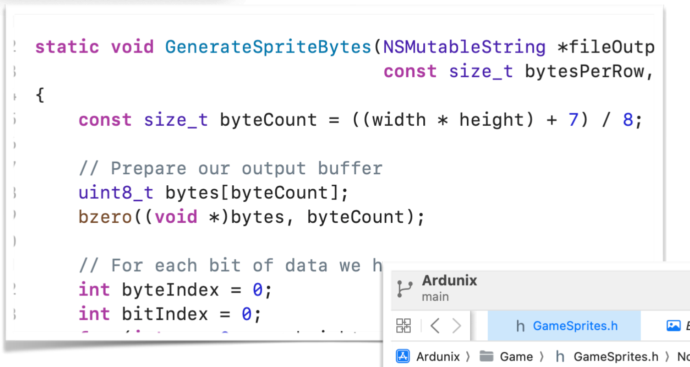
Part deux: automation?



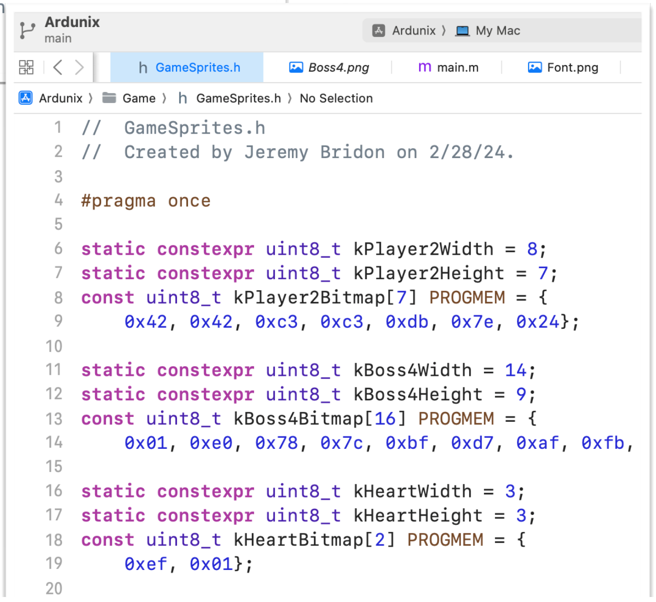
Sprites!

Part deux: automation?





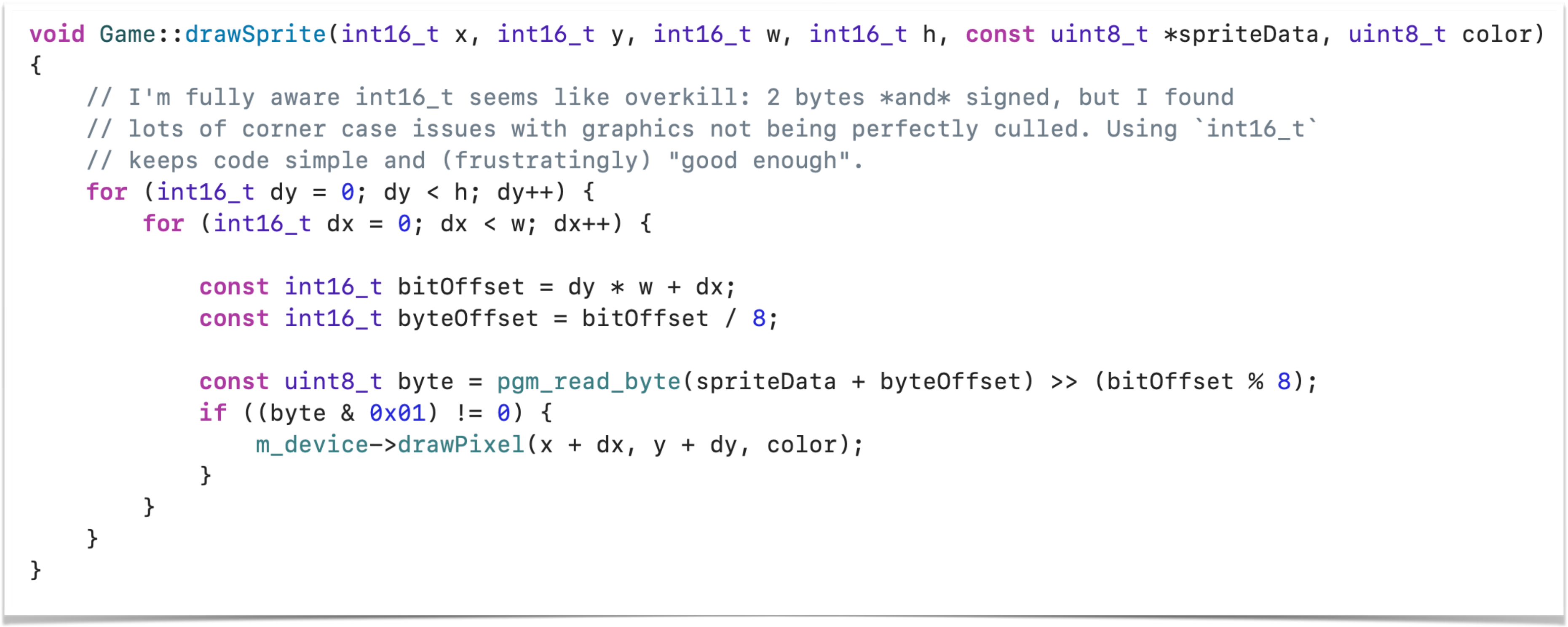
```
static void GenerateSpriteBytes(NSMutableString *fileOutp
                                const size_t bytesPerRow,
{
    const size_t byteCount = ((width * height) + 7) / 8;

    // Prepare our output buffer
    uint8_t bytes[byteCount];
    bzero((void *)bytes, byteCount);

    // For each bit of data we h
    int byteIndex = 0;
    int bitIndex = 0;
```



```
//  GameSprites.h
//  Created by Jeremy Bridon on 2/28/24.

#pragma once

static constexpr uint8_t kPlayer2Width = 8;
static constexpr uint8_t kPlayer2Height = 7;
const uint8_t kPlayer2Bitmap[7] PROGMEM = {
    0x42, 0x42, 0xc3, 0xc3, 0xdb, 0x7e, 0x24};

static constexpr uint8_t kBoss4Width = 14;
static constexpr uint8_t kBoss4Height = 9;
const uint8_t kBoss4Bitmap[16] PROGMEM = {
    0x01, 0xe0, 0x78, 0x7c, 0xbf, 0xd7, 0xaf, 0xfb,

static constexpr uint8_t kHeartWidth = 3;
static constexpr uint8_t kHeartHeight = 3;
const uint8_t kHeartBitmap[2] PROGMEM = {
    0xef, 0x01};
```

- Challenge 2: asset generation!
  - For each image...
    - Load a CGImage, get raw bytes out
    - Build a bit-packed array
    - Emit C array

Sprites!
Part deux: automation?

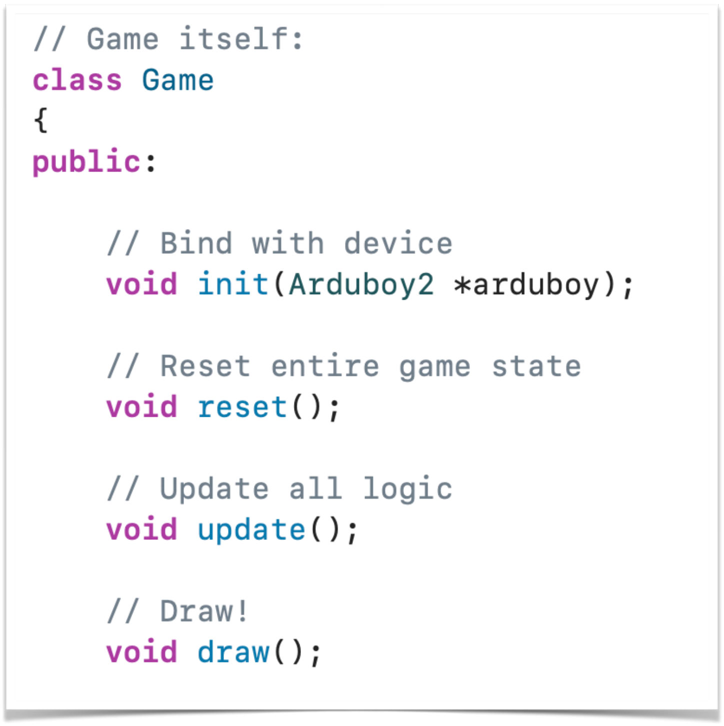


```cpp
void Game::drawSprite(int16_t x, int16_t y, int16_t w, int16_t h, const uint8_t *spriteData, uint8_t color)
{
    // I'm fully aware int16_t seems like overkill: 2 bytes *and* signed, but I found
    // lots of corner case issues with graphics not being perfectly culled. Using `int16_t`
    // keeps code simple and (frustratingly) "good enough".
    for (int16_t dy = 0; dy < h; dy++) {
        for (int16_t dx = 0; dx < w; dx++) {

            const int16_t bitOffset = dy * w + dx;
            const int16_t byteOffset = bitOffset / 8;

            const uint8_t byte = pgm_read_byte(spriteData + byteOffset) >> (bitOffset % 8);
            if ((byte & 0x01) != 0) {
                m_device->drawPixel(x + dx, y + dy, color);
            }
        }
    }
}
```

Fifth & Final Challenge:

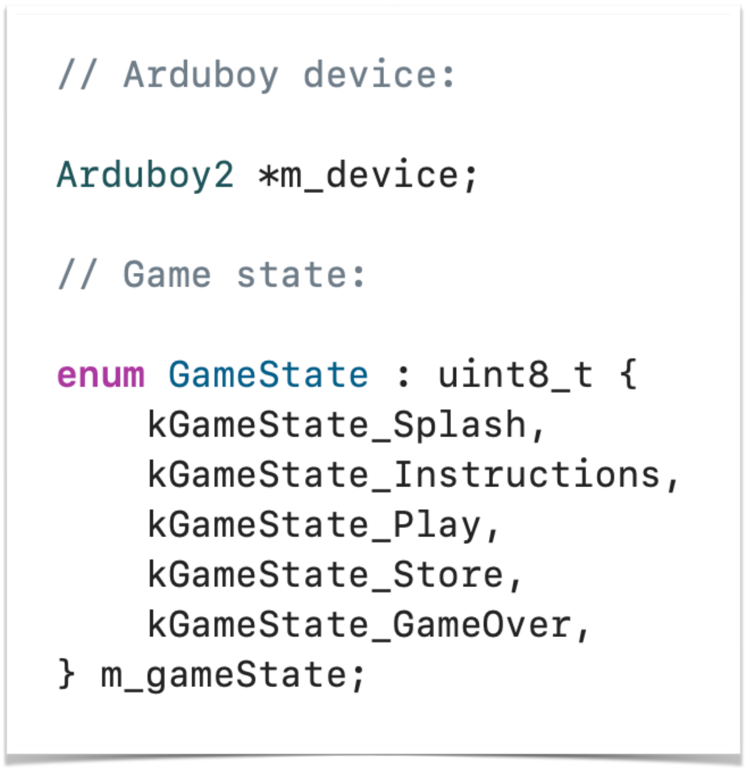
Make the game!

Game Architecture



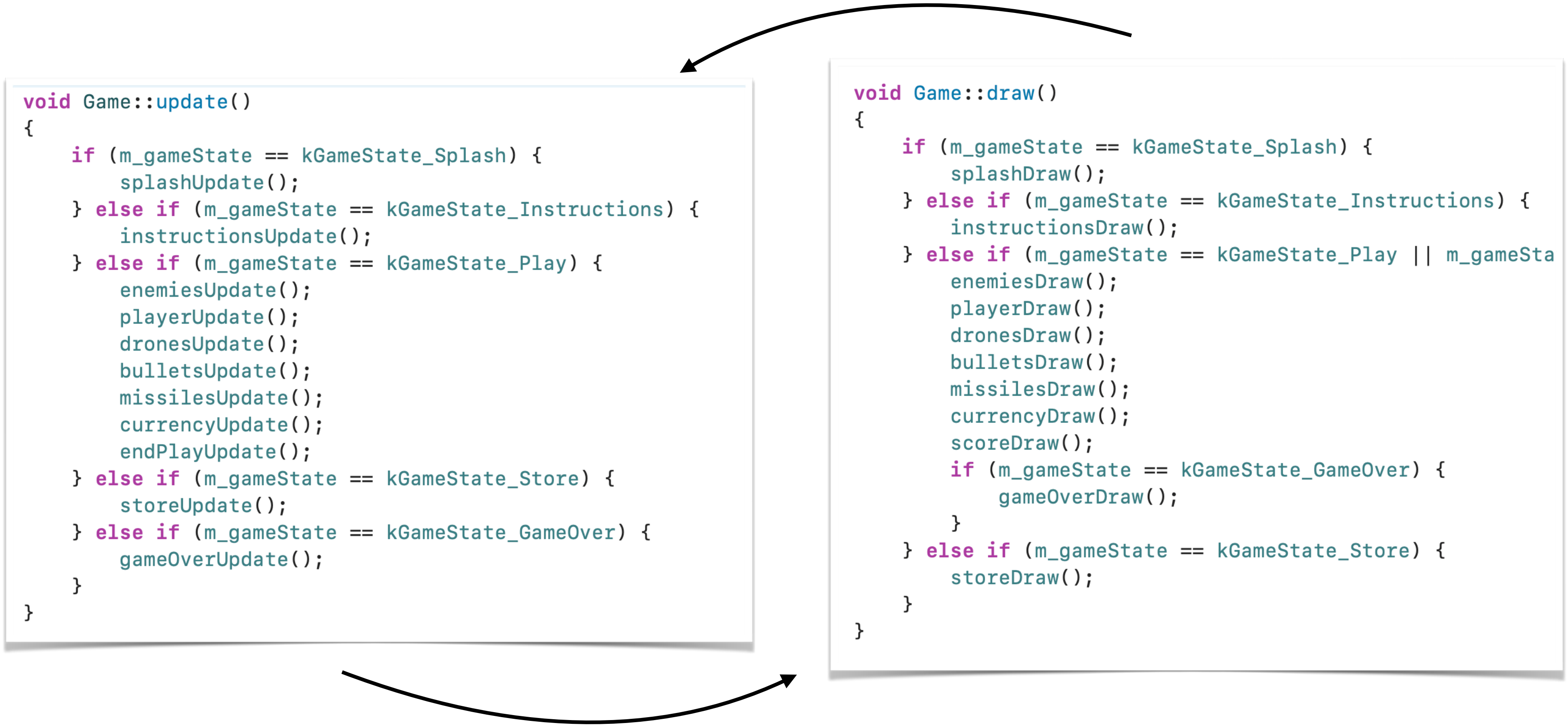
```
// Game itself:
class Game
{
public:

    // Bind with device
    void init(Arduboy2 *arduboy);

    // Reset entire game state
    void reset();

    // Update all logic
    void update();

    // Draw!
    void draw();
```

Game Architecture



```
// Arduboy device:

Arduboy2 *m_device;

// Game state:

enum GameState : uint8_t {
    kGameState_Splash,
    kGameState_Instructions,
    kGameState_Play,
    kGameState_Store,
    kGameState_GameOver,
} m_gameState;
```

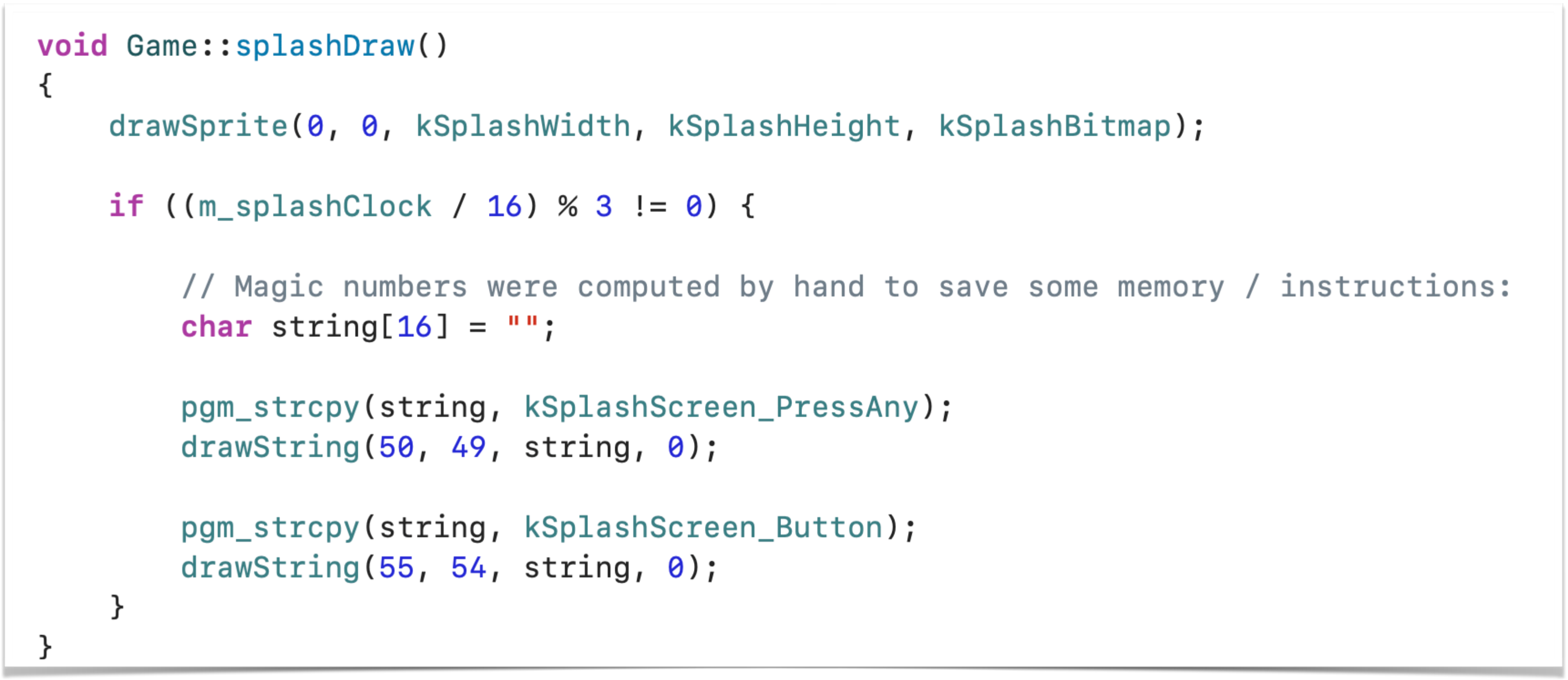
Game Architecture



```
void Game::update()
{
    if (m_gameState == kGameState_Splash) {
        splashUpdate();
    } else if (m_gameState == kGameState_Instructions) {
        instructionsUpdate();
    } else if (m_gameState == kGameState_Play) {
        enemiesUpdate();
        playerUpdate();
        dronesUpdate();
        bulletsUpdate();
        missilesUpdate();
        currencyUpdate();
        endPlayUpdate();
    } else if (m_gameState == kGameState_Store) {
        storeUpdate();
    } else if (m_gameState == kGameState_GameOver) {
        gameOverUpdate();
    }
}
```

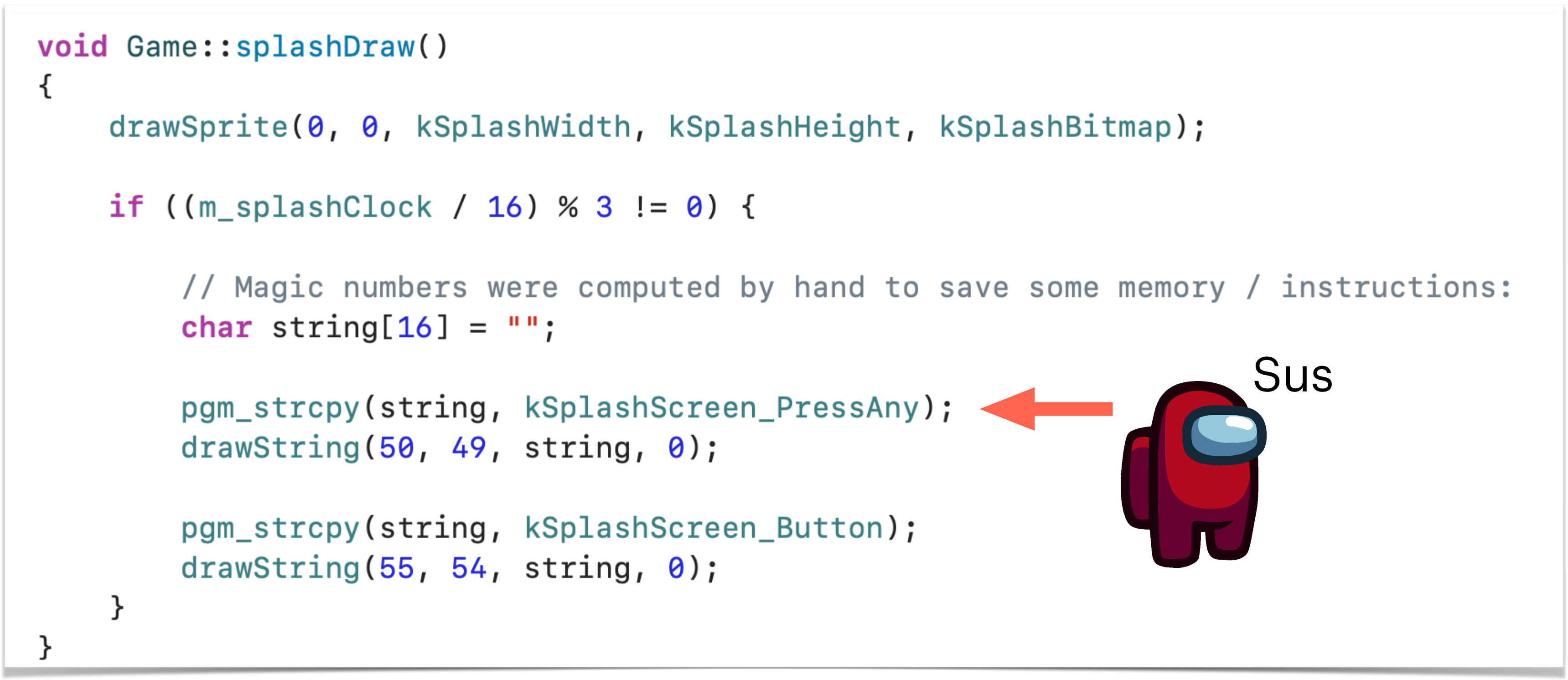
```
void Game::draw()
{
    if (m_gameState == kGameState_Splash) {
        splashDraw();
    } else if (m_gameState == kGameState_Instructions) {
        instructionsDraw();
    } else if (m_gameState == kGameState_Play || m_gameSta
        enemiesDraw();
        playerDraw();
        dronesDraw();
        bulletsDraw();
        missilesDraw();
        currencyDraw();
        scoreDraw();
        if (m_gameState == kGameState_GameOver) {
            gameOverDraw();
        }
    } else if (m_gameState == kGameState_Store) {
        storeDraw();
    }
}
```

Game Architecture

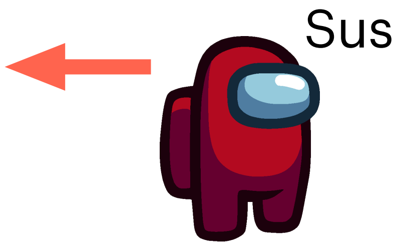


```
void Game::splashDraw()
{
    drawSprite(0, 0, kSplashWidth, kSplashHeight, kSplashBitmap);

    if ((m_splashClock / 16) % 3 != 0) {

        // Magic numbers were computed by hand to save some memory / instructions:
        char string[16] = "";

        pgm_strcpy(string, kSplashScreen_PressAny);
        drawString(50, 49, string, 0);

        pgm_strcpy(string, kSplashScreen_Button);
        drawString(55, 54, string, 0);
    }
}
```

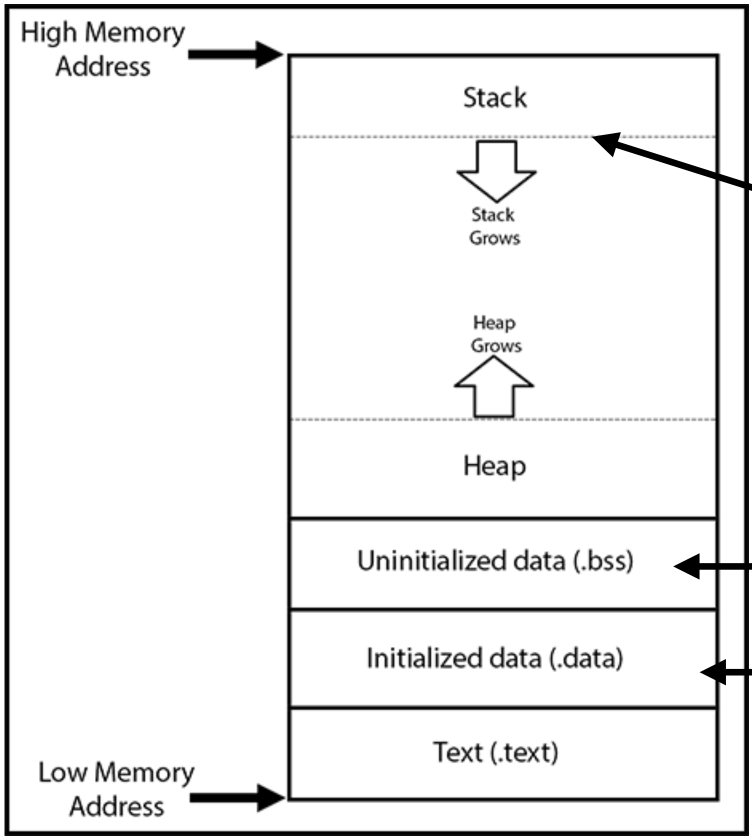
Game Architecture



```
void Game::splashDraw()
{
    drawSprite(0, 0, kSplashWidth, kSplashHeight, kSplashBitmap);

    if ((m_splashClock / 16) % 3 != 0) {

        // Magic numbers were computed by hand to save some memory / instructions:
        char string[16] = "";

        pgm_strcpy(string, kSplashScreen_PressAny);
        drawString(50, 49, string, 0);

        pgm_strcpy(string, kSplashScreen_Button);
        drawString(55, 54, string, 0);
    }
}
```



Game Architecture

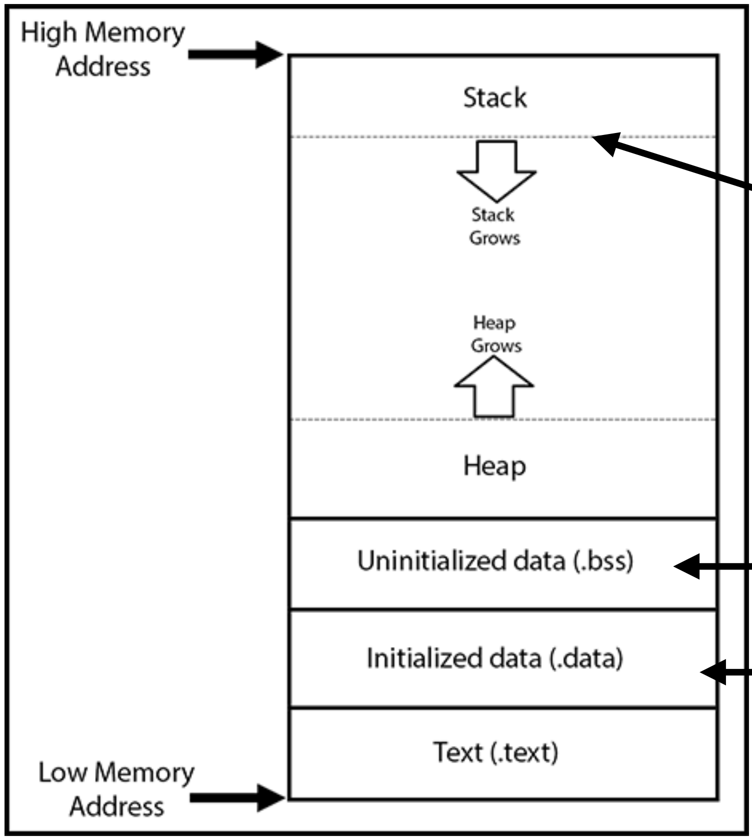


Static Mutable

Static Const

Code

Game Architecture



Function frame

Variables

String buffers

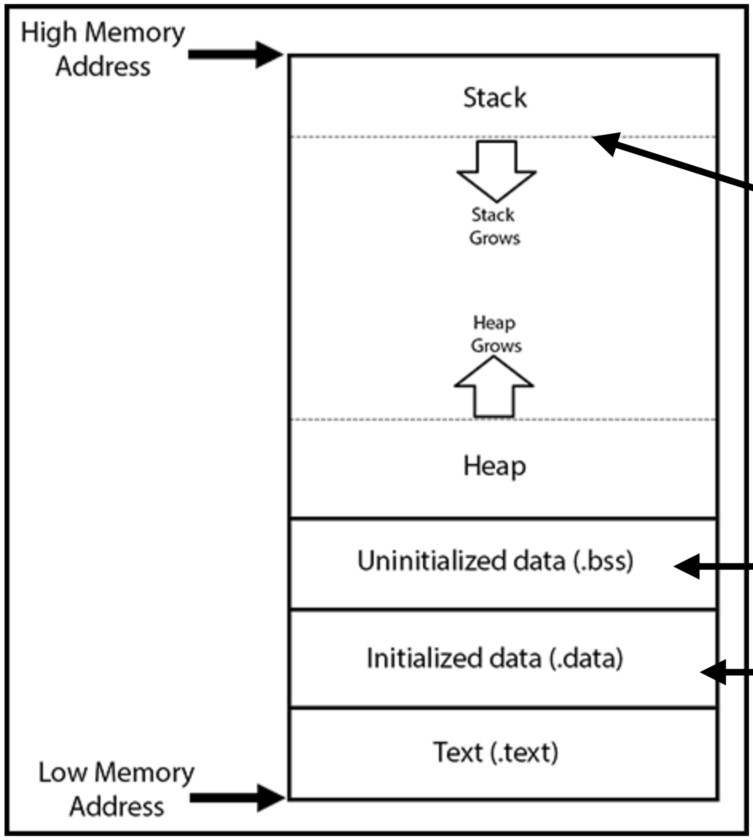
Static Mutable

Static Const

Code

String Literals

Game Architecture



Problem 1

Function frame

Variables

String buffers

Problem 2

Static Mutable

Static Const

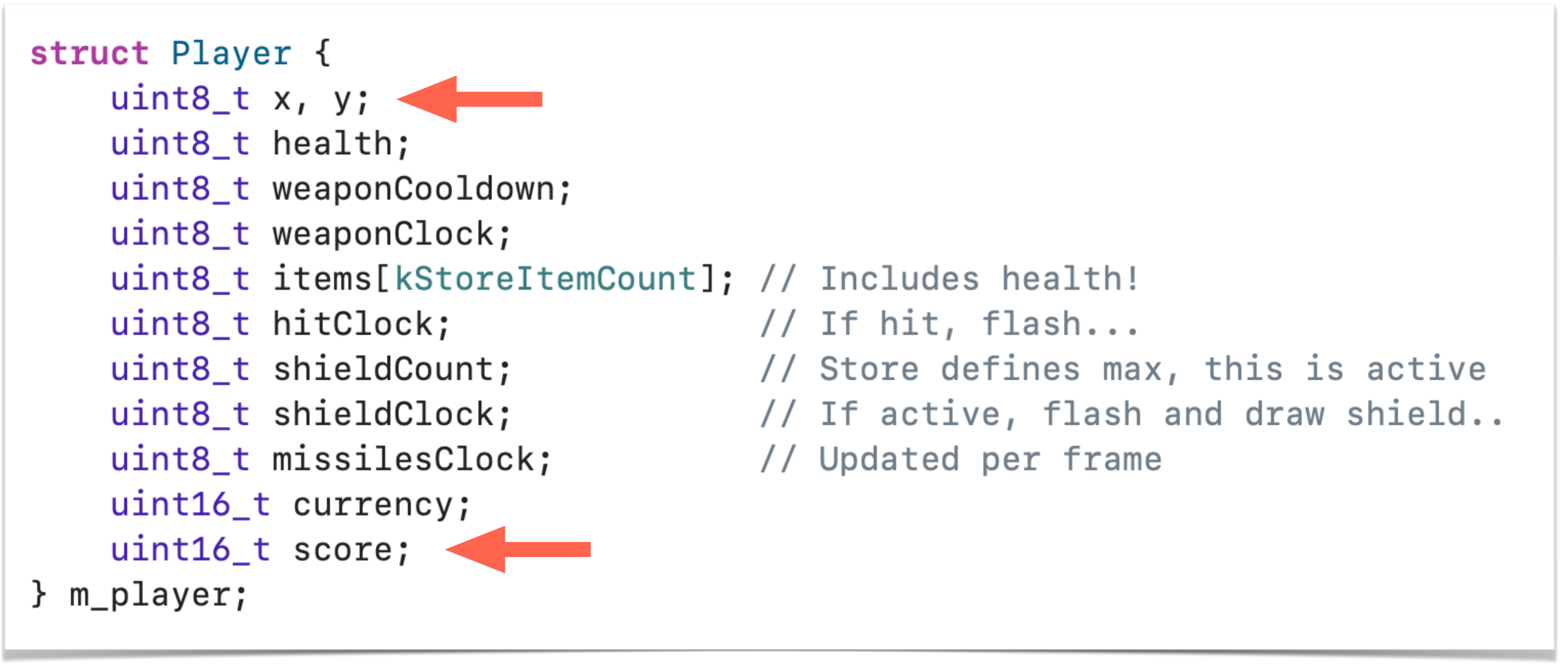
Code

String Literals

Game Architecture

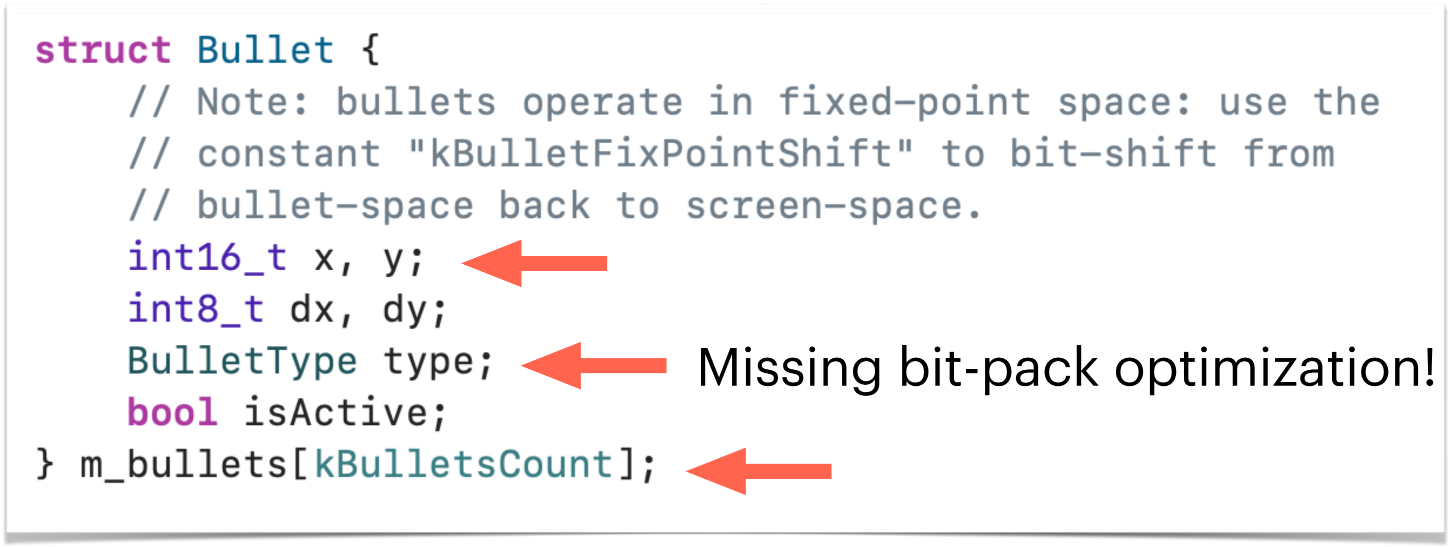
- Use smallest integer possible: uint8_t, uint16_t, etc
    - Avoid ambiguity: `char` isn't always byte, `int` is different on arm64 vs. atmega32
    - Play with `constexpr`; depends on CPU ISA encoding
- Avoid any dynamic allocation: make max stack depth known
    - Pre-allocate all runtime game data on the stack
- Speed: Only use integers, atmega32 doesn't have FPU
    - Consider fixed-point arithmetic
    - Consider bit-packing, but trade-offs

Game Architecture



```
struct Player {
    uint8_t x, y;
    uint8_t health;
    uint8_t weaponCooldown;
    uint8_t weaponClock;
    uint8_t items[kStoreItemCount]; // Includes health!
    uint8_t hitClock;               // If hit, flash...
    uint8_t shieldCount;            // Store defines max, this is active
    uint8_t shieldClock;            // If active, flash and draw shield..
    uint8_t missilesClock;          // Updated per frame
    uint16_t currency;
    uint16_t score;
} m_player;
```

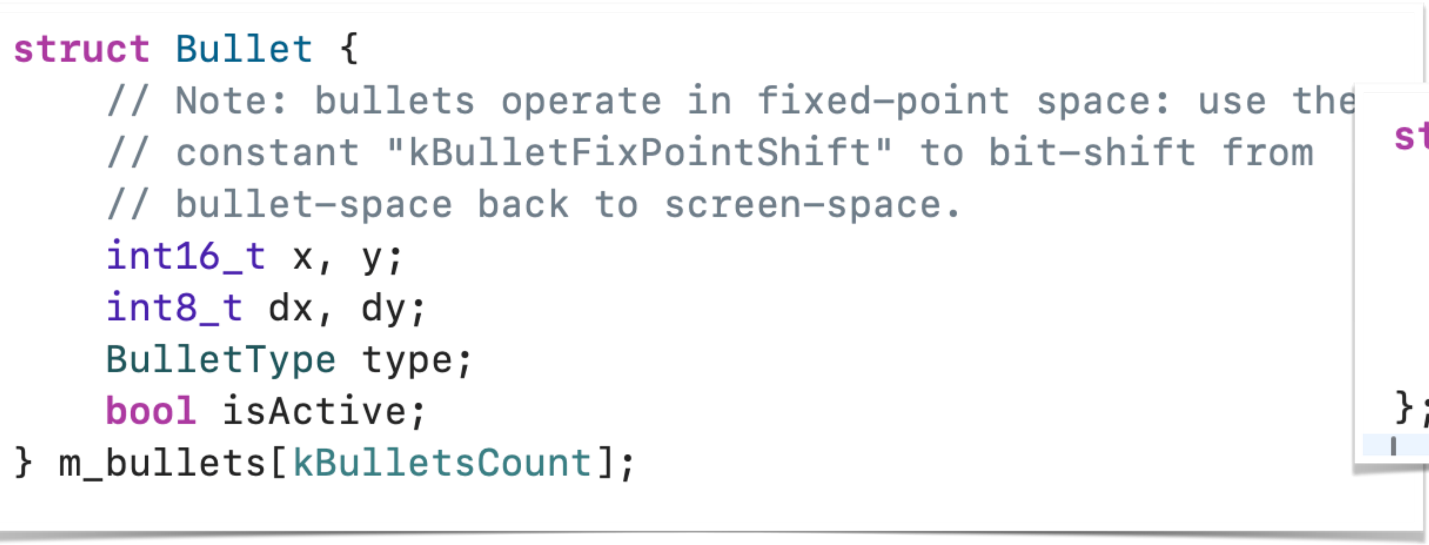
Game Architecture



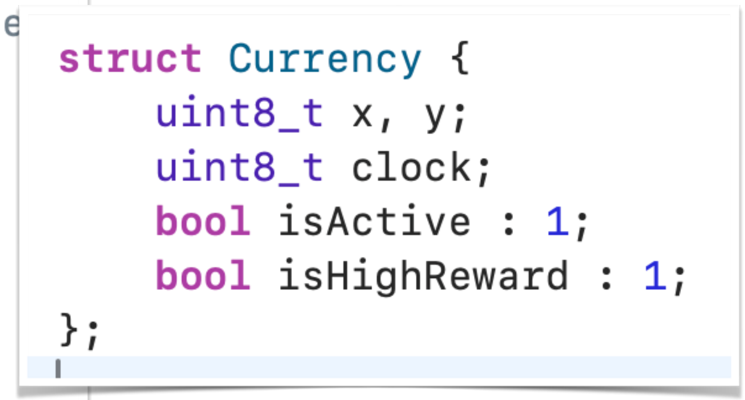
```
struct Bullet {
    // Note: bullets operate in fixed-point space: use the
    // constant "kBulletFixPointShift" to bit-shift from
    // bullet-space back to screen-space.
    int16_t x, y;
    int8_t dx, dy;
    BulletType type;
    bool isActive;
} m_bullets[kBulletsCount];
```

Missing bit-pack optimization!

Game Architecture

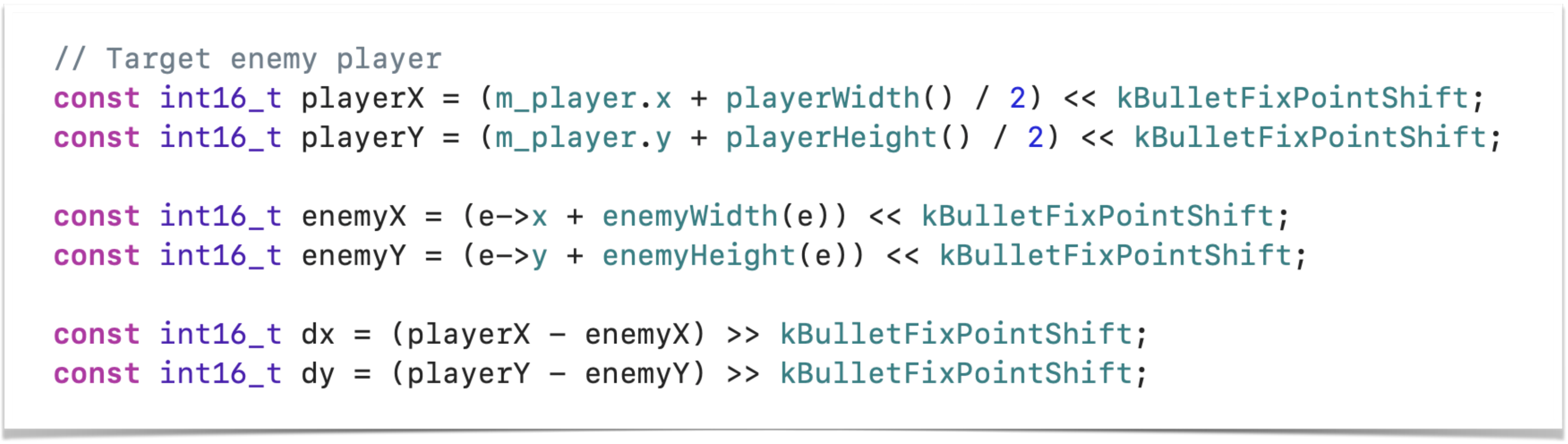


```
struct Bullet {
    // Note: bullets operate in fixed-point space: use the
    // constant "kBulletFixPointShift" to bit-shift from
    // bullet-space back to screen-space.
    int16_t x, y;
    int8_t dx, dy;
    BulletType type;
    bool isActive;
} m_bullets[kBulletsCount];
```



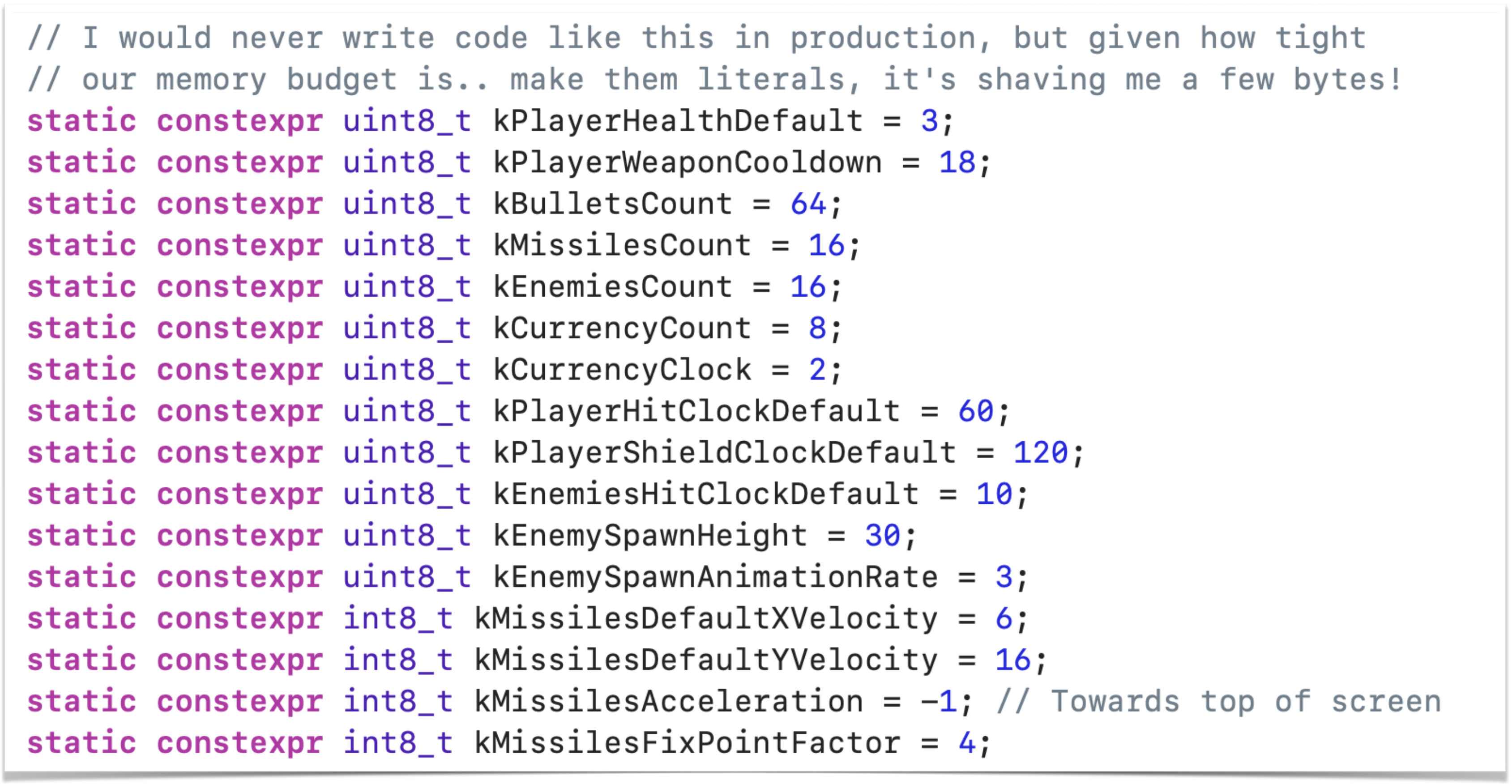
```
struct Currency {
    uint8_t x, y;
    uint8_t clock;
    bool isActive : 1;
    bool isHighReward : 1;
};
```

Game Architecture



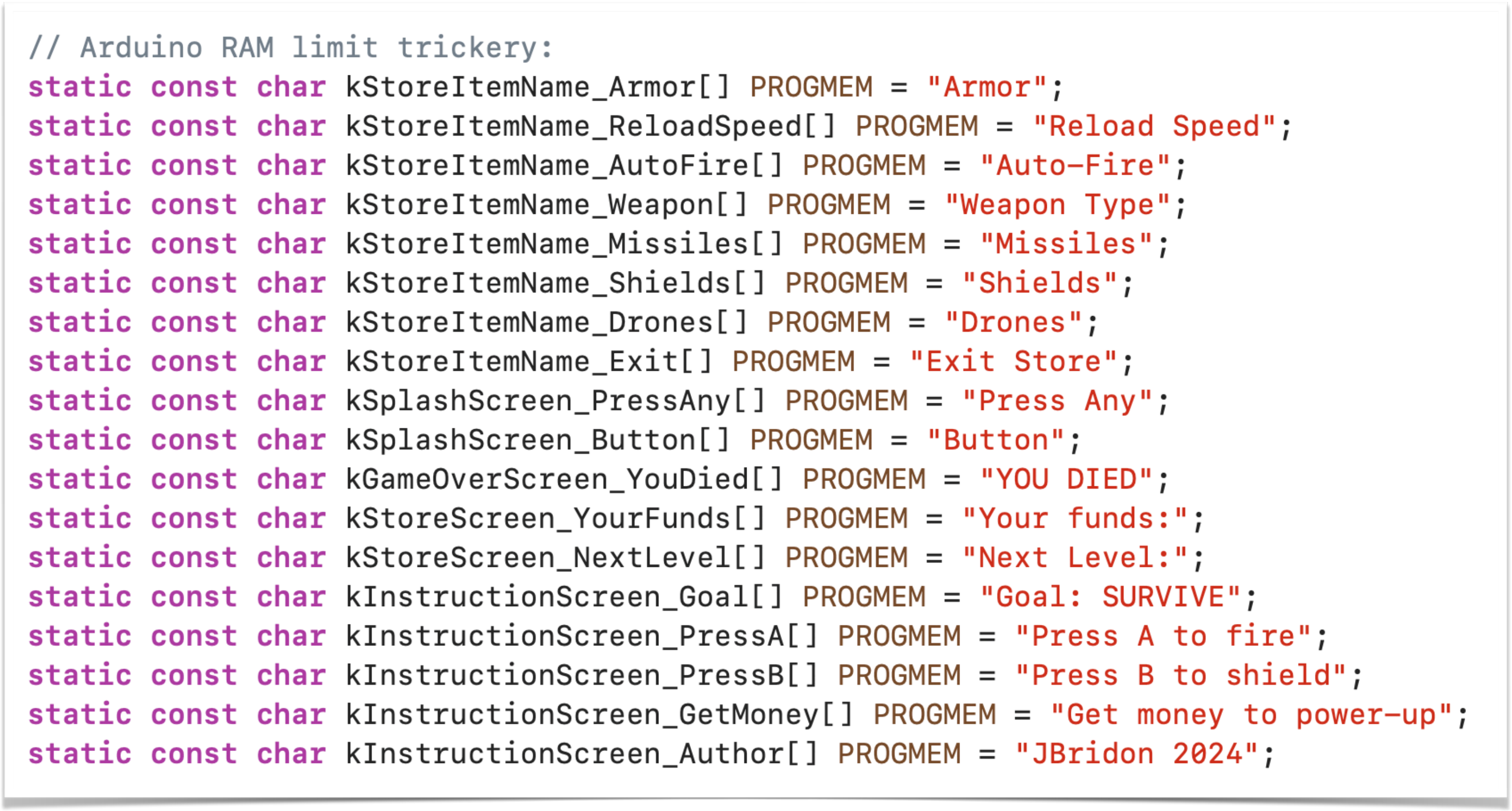
```
// Target enemy player
const int16_t playerX = (m_player.x + playerWidth() / 2) << kBulletFixPointShift;
const int16_t playerY = (m_player.y + playerHeight() / 2) << kBulletFixPointShift;

const int16_t enemyX = (e->x + enemyWidth(e)) << kBulletFixPointShift;
const int16_t enemyY = (e->y + enemyHeight(e)) << kBulletFixPointShift;

const int16_t dx = (playerX - enemyX) >> kBulletFixPointShift;
const int16_t dy = (playerY - enemyY) >> kBulletFixPointShift;
```

Game Architecture



```cpp
// I would never write code like this in production, but given how tight
// our memory budget is.. make them literals, it's shaving me a few bytes!
static constexpr uint8_t kPlayerHealthDefault = 3;
static constexpr uint8_t kPlayerWeaponCooldown = 18;
static constexpr uint8_t kBulletsCount = 64;
static constexpr uint8_t kMissilesCount = 16;
static constexpr uint8_t kEnemiesCount = 16;
static constexpr uint8_t kCurrencyCount = 8;
static constexpr uint8_t kCurrencyClock = 2;
static constexpr uint8_t kPlayerHitClockDefault = 60;
static constexpr uint8_t kPlayerShieldClockDefault = 120;
static constexpr uint8_t kEnemiesHitClockDefault = 10;
static constexpr uint8_t kEnemySpawnHeight = 30;
static constexpr uint8_t kEnemySpawnAnimationRate = 3;
static constexpr int8_t kMissilesDefaultXVelocity = 6;
static constexpr int8_t kMissilesDefaultYVelocity = 16;
static constexpr int8_t kMissilesAcceleration = -1; // Towards top of screen
static constexpr int8_t kMissilesFixPointFactor = 4;
```

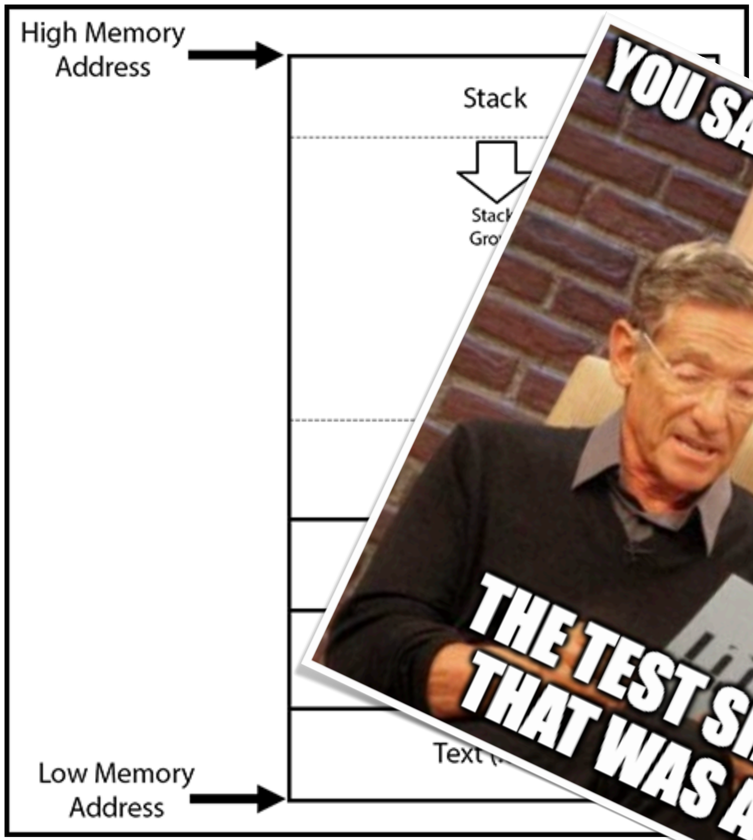
Game Architecture

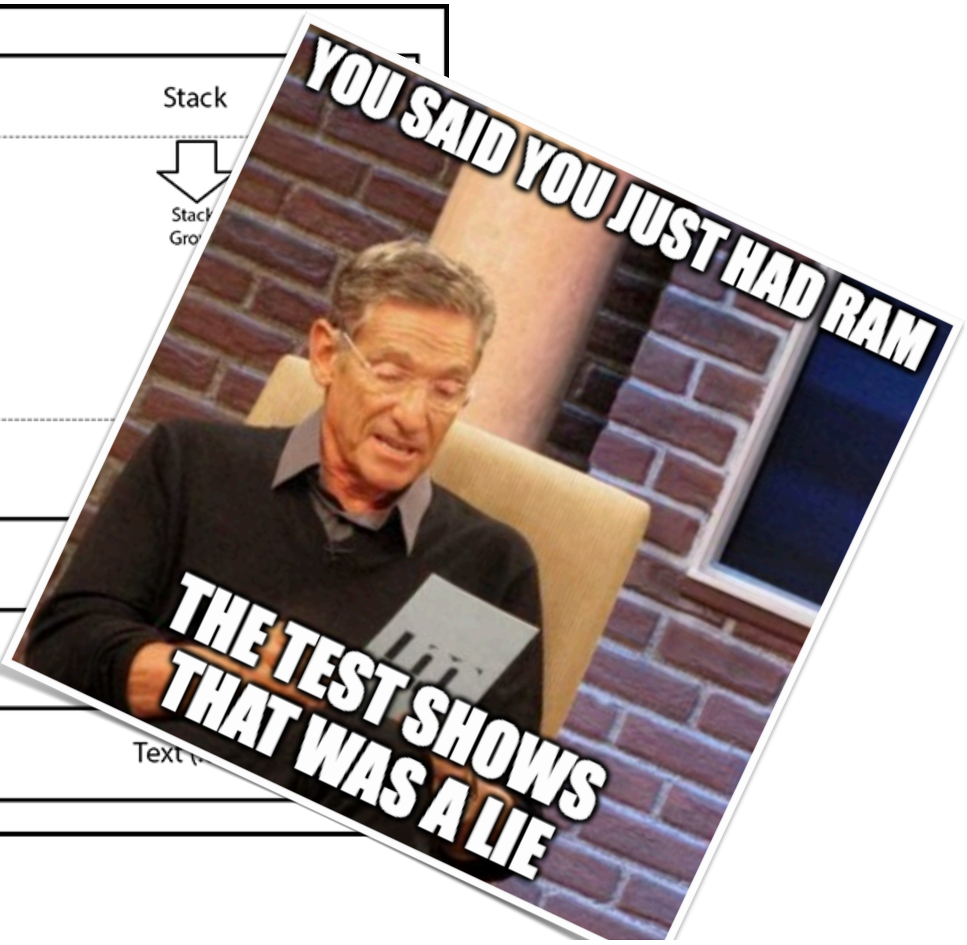


```
// Arduino RAM limit trickery:
static const char kStoreItemName_Armor[] PROGMEM = "Armor";
static const char kStoreItemName_ReloadSpeed[] PROGMEM = "Reload Speed";
static const char kStoreItemName_AutoFire[] PROGMEM = "Auto-Fire";
static const char kStoreItemName_Weapon[] PROGMEM = "Weapon Type";
static const char kStoreItemName_Missiles[] PROGMEM = "Missiles";
static const char kStoreItemName_Shields[] PROGMEM = "Shields";
static const char kStoreItemName_Drones[] PROGMEM = "Drones";
static const char kStoreItemName_Exit[] PROGMEM = "Exit Store";
static const char kSplashScreen_PressAny[] PROGMEM = "Press Any";
static const char kSplashScreen_Button[] PROGMEM = "Button";
static const char kGameOverScreen_YouDied[] PROGMEM = "YOU DIED";
static const char kStoreScreen_YourFunds[] PROGMEM = "Your funds:";
static const char kStoreScreen_NextLevel[] PROGMEM = "Next Level:";
static const char kInstructionScreen_Goal[] PROGMEM = "Goal: SURVIVE";
static const char kInstructionScreen_PressA[] PROGMEM = "Press A to fire";
static const char kInstructionScreen_PressB[] PROGMEM = "Press B to shield";
static const char kInstructionScreen_GetMoney[] PROGMEM = "Get money to power-up";
static const char kInstructionScreen_Author[] PROGMEM = "JBridon 2024";
```

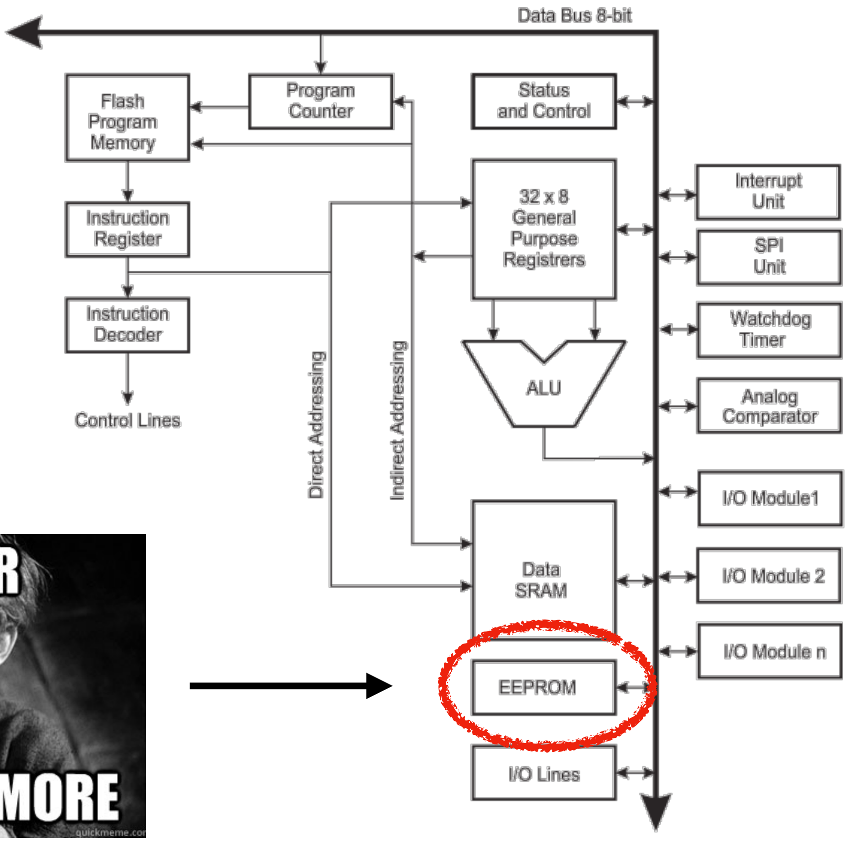
PROGMEM

Game Architecture

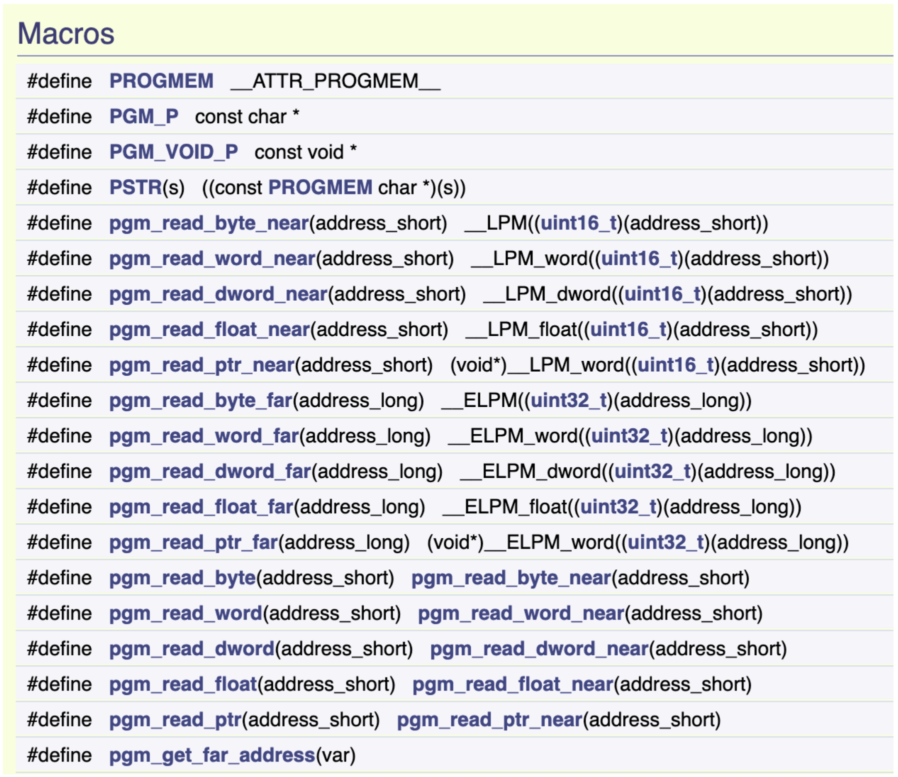




Game Architecture

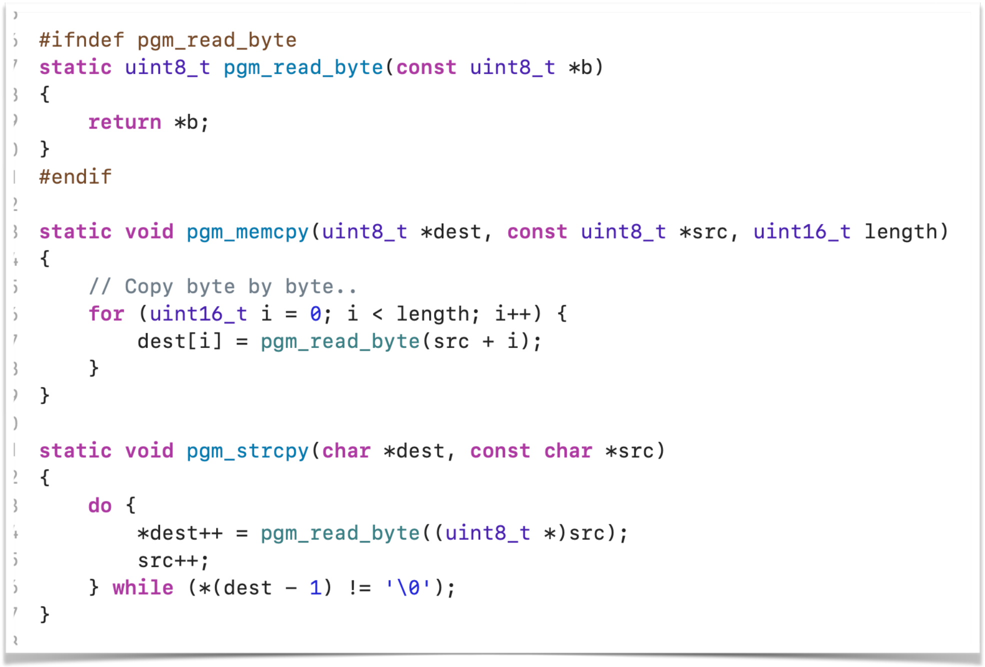


Game Architecture



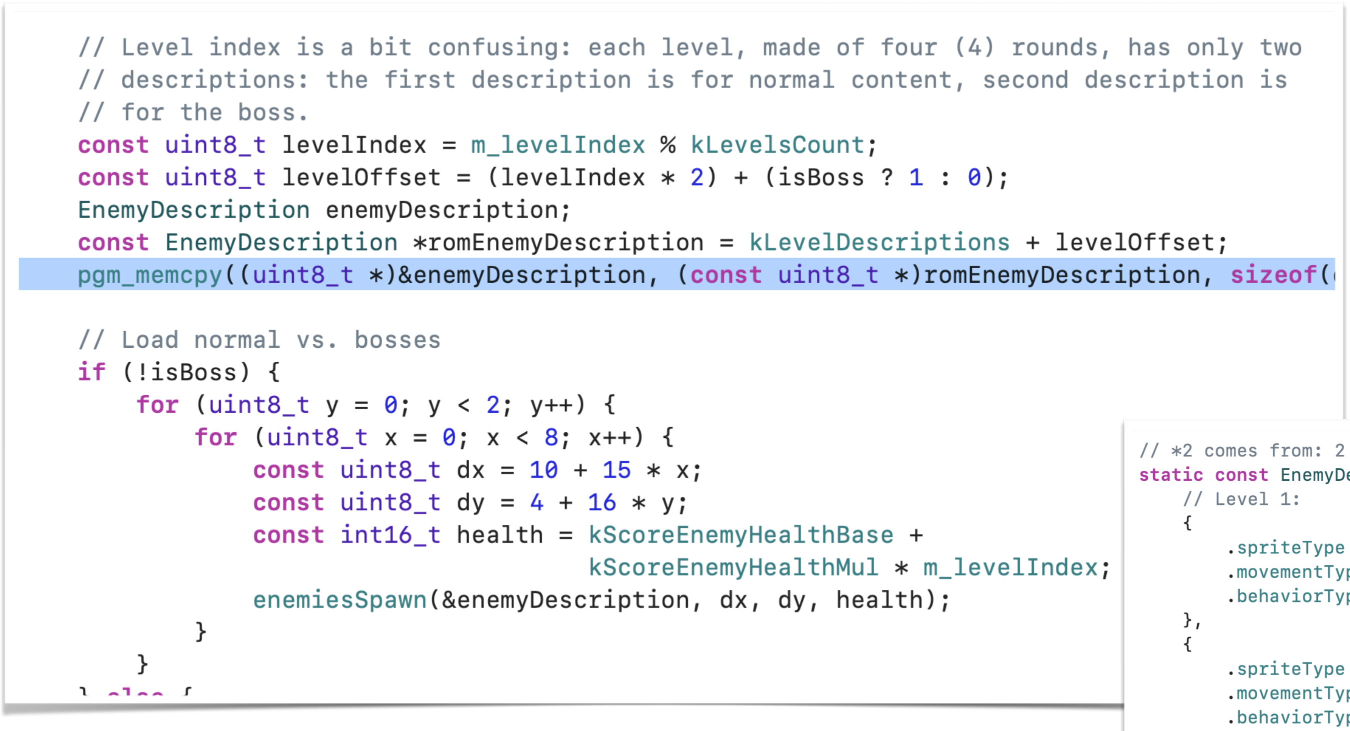
Macros

#define	PROGMEM	__ATTR_PROGMEM__
#define	PGM_P	const char *
#define	PGM_VOID_P	const void *
#define	PSTR(s)	((const PROGMEM char *)(s))
#define	pgm_read_byte_near(address_short)	__LPM((uint16_t)(address_short))
#define	pgm_read_word_near(address_short)	__LPM_word((uint16_t)(address_short))
#define	pgm_read_dword_near(address_short)	__LPM_dword((uint16_t)(address_short))
#define	pgm_read_float_near(address_short)	__LPM_float((uint16_t)(address_short))
#define	pgm_read_ptr_near(address_short)	(void*)__LPM_word((uint16_t)(address_short))
#define	pgm_read_byte_far(address_long)	__ELPM((uint32_t)(address_long))
#define	pgm_read_word_far(address_long)	__ELPM_word((uint32_t)(address_long))
#define	pgm_read_dword_far(address_long)	__ELPM_dword((uint32_t)(address_long))
#define	pgm_read_float_far(address_long)	__ELPM_float((uint32_t)(address_long))
#define	pgm_read_ptr_far(address_long)	(void*)__ELPM_word((uint32_t)(address_long))
#define	pgm_read_byte(address_short)	pgm_read_byte_near(address_short)
#define	pgm_read_word(address_short)	pgm_read_word_near(address_short)
#define	pgm_read_dword(address_short)	pgm_read_dword_near(address_short)
#define	pgm_read_float(address_short)	pgm_read_float_near(address_short)
#define	pgm_read_ptr(address_short)	pgm_read_ptr_near(address_short)
#define	pgm_get_far_address(var)	

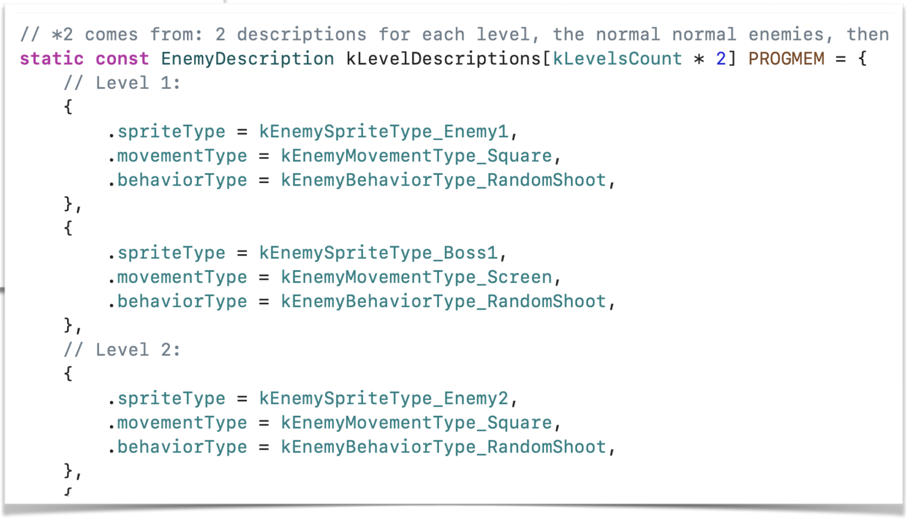


```c
#ifndef pgm_read_byte
static uint8_t pgm_read_byte(const uint8_t *b)
{
    return *b;
}
#endif

static void pgm_memcpy(uint8_t *dest, const uint8_t *src, uint16_t length)
{
    // Copy byte by byte..
    for (uint16_t i = 0; i < length; i++) {
        dest[i] = pgm_read_byte(src + i);
    }
}

static void pgm_strcpy(char *dest, const char *src)
{
    do {
        *dest++ = pgm_read_byte((uint8_t *)src);
        src++;
    } while (*(dest - 1) != '\0');
}
```

Game Architecture

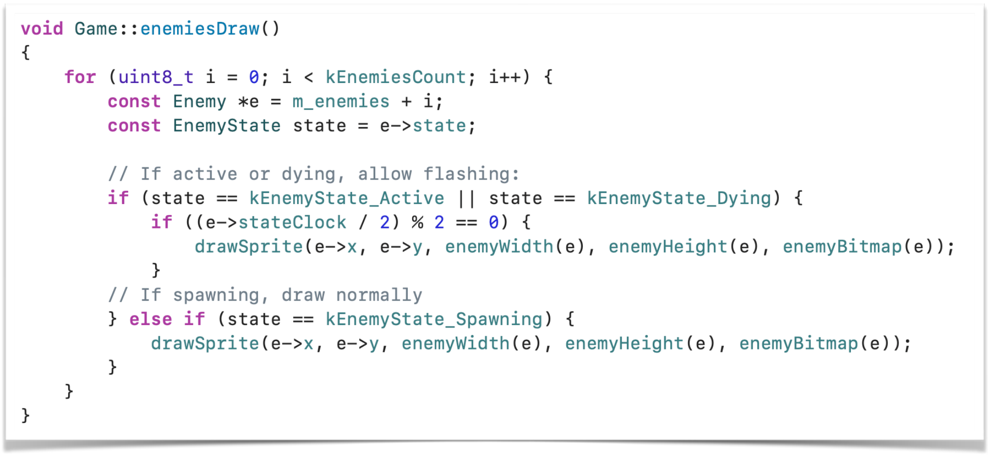


```
// Level index is a bit confusing: each level, made of four (4) rounds, has only two
// descriptions: the first description is for normal content, second description is
// for the boss.
const uint8_t levelIndex = m_levelIndex % kLevelsCount;
const uint8_t levelOffset = (levelIndex * 2) + (isBoss ? 1 : 0);
EnemyDescription enemyDescription;
const EnemyDescription *romEnemyDescription = kLevelDescriptions + levelOffset;
pgm_memcpy((uint8_t *)&enemyDescription, (const uint8_t *)romEnemyDescription, sizeof(

// Load normal vs. bosses
if (!isBoss) {
    for (uint8_t y = 0; y < 2; y++) {
        for (uint8_t x = 0; x < 8; x++) {
            const uint8_t dx = 10 + 15 * x;
            const uint8_t dy = 4 + 16 * y;
            const int16_t health = kScoreEnemyHealthBase +
                                   kScoreEnemyHealthMul * m_levelIndex;
            enemiesSpawn(&enemyDescription, dx, dy, health);
        }
    }
} else {
```



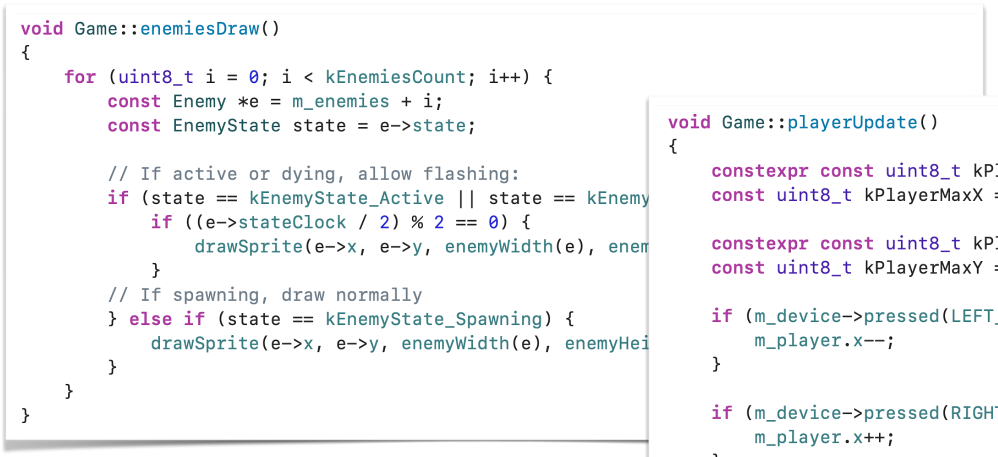
```
// *2 comes from: 2 descriptions for each level, the normal normal enemies, then
static const EnemyDescription kLevelDescriptions[kLevelsCount * 2] PROGMEM = {
    // Level 1:
    {
        .spriteType = kEnemySpriteType_Enemy1,
        .movementType = kEnemyMovementType_Square,
        .behaviorType = kEnemyBehaviorType_RandomShoot,
    },
    {
        .spriteType = kEnemySpriteType_Boss1,
        .movementType = kEnemyMovementType_Screen,
        .behaviorType = kEnemyBehaviorType_RandomShoot,
    },
    // Level 2:
    {
        .spriteType = kEnemySpriteType_Enemy2,
        .movementType = kEnemyMovementType_Square,
        .behaviorType = kEnemyBehaviorType_RandomShoot,
    },
```

Game Architecture

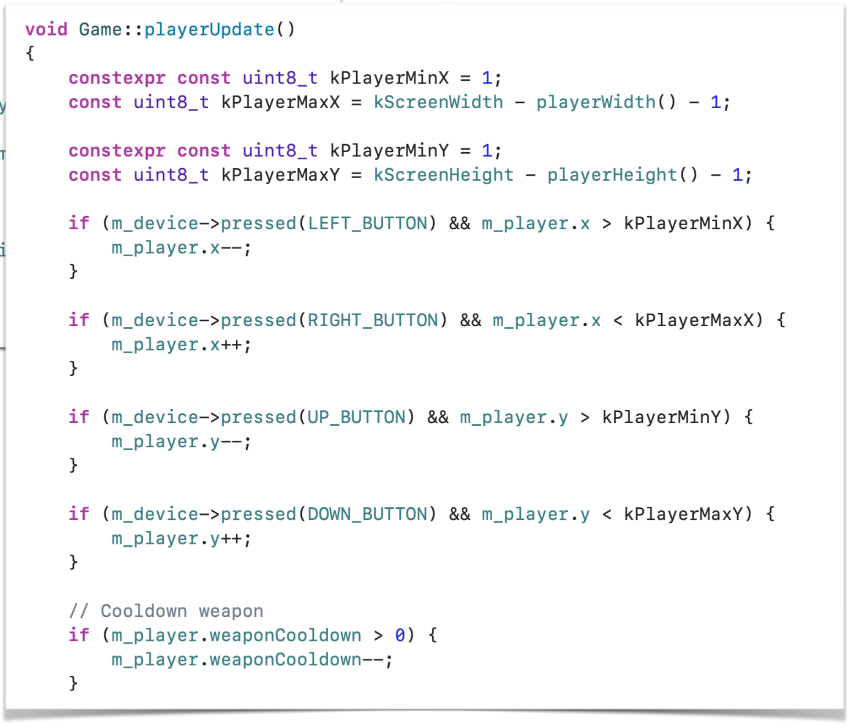


```
void Game::enemiesDraw()
{
    for (uint8_t i = 0; i < kEnemiesCount; i++) {
        const Enemy *e = m_enemies + i;
        const EnemyState state = e->state;

        // If active or dying, allow flashing:
        if (state == kEnemyState_Active || state == kEnemyState_Dying) {
            if ((e->stateClock / 2) % 2 == 0) {
                drawSprite(e->x, e->y, enemyWidth(e), enemyHeight(e), enemyBitmap(e));
            }
        // If spawning, draw normally
        } else if (state == kEnemyState_Spawning) {
            drawSprite(e->x, e->y, enemyWidth(e), enemyHeight(e), enemyBitmap(e));
        }
    }
}
```

Game Architecture

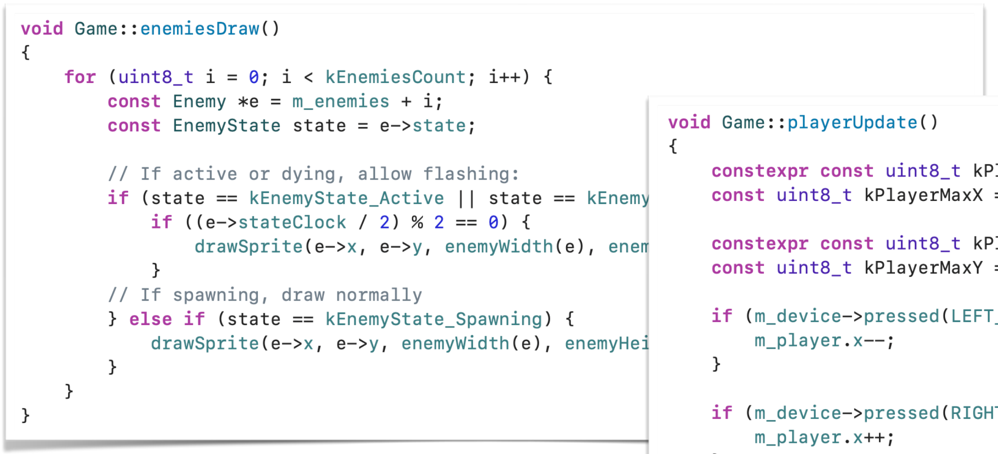


```
void Game::enemiesDraw()
{
    for (uint8_t i = 0; i < kEnemiesCount; i++) {
        const Enemy *e = m_enemies + i;
        const EnemyState state = e->state;

        // If active or dying, allow flashing:
        if (state == kEnemyState_Active || state == kEnemy
            if ((e->stateClock / 2) % 2 == 0) {
                drawSprite(e->x, e->y, enemyWidth(e), enem
            }
        // If spawning, draw normally
        } else if (state == kEnemyState_Spawning) {
            drawSprite(e->x, e->y, enemyWidth(e), enemyHei
        }
    }
}
```

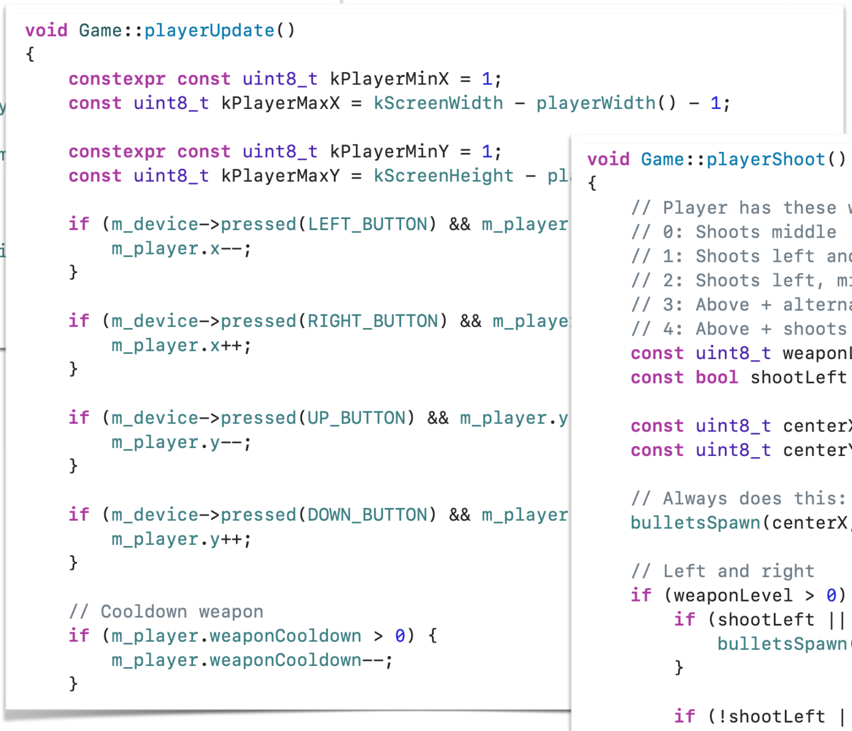


```
void Game::playerUpdate()
{
    constexpr const uint8_t kPlayerMinX = 1;
    const uint8_t kPlayerMaxX = kScreenWidth - playerWidth() - 1;

    constexpr const uint8_t kPlayerMinY = 1;
    const uint8_t kPlayerMaxY = kScreenHeight - playerHeight() - 1;

    if (m_device->pressed(LEFT_BUTTON) && m_player.x > kPlayerMinX) {
        m_player.x--;
    }

    if (m_device->pressed(RIGHT_BUTTON) && m_player.x < kPlayerMaxX) {
        m_player.x++;
    }

    if (m_device->pressed(UP_BUTTON) && m_player.y > kPlayerMinY) {
        m_player.y--;
    }

    if (m_device->pressed(DOWN_BUTTON) && m_player.y < kPlayerMaxY) {
        m_player.y++;
    }

    // Cooldown weapon
    if (m_player.weaponCooldown > 0) {
        m_player.weaponCooldown--;
    }
```

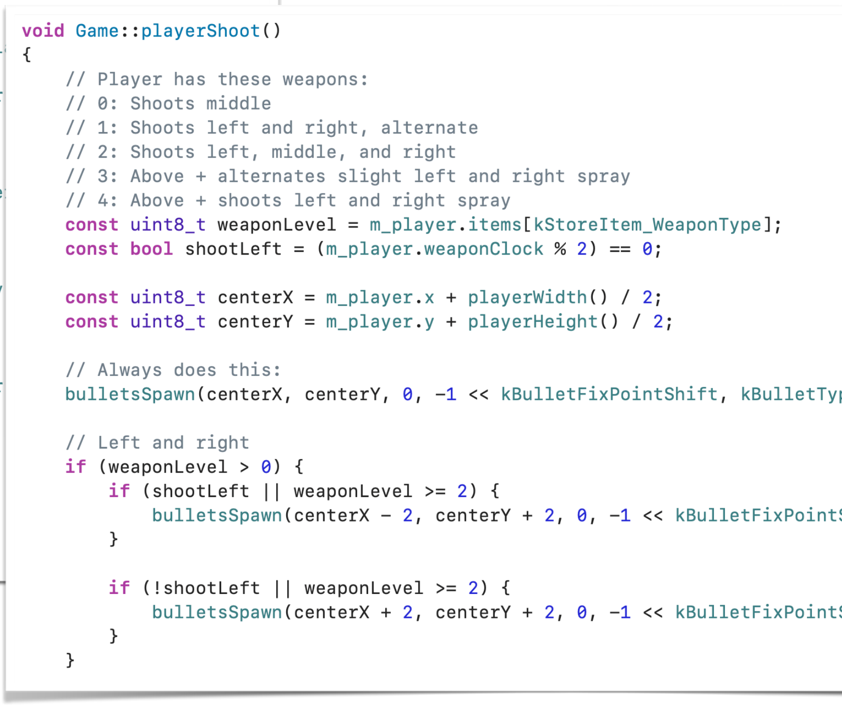
Game Architecture



```
void Game::enemiesDraw()
{
    for (uint8_t i = 0; i < kEnemiesCount; i++) {
        const Enemy *e = m_enemies + i;
        const EnemyState state = e->state;

        // If active or dying, allow flashing:
        if (state == kEnemyState_Active || state == kEnemy
            if ((e->stateClock / 2) % 2 == 0) {
                drawSprite(e->x, e->y, enemyWidth(e), enem
            }
        // If spawning, draw normally
        } else if (state == kEnemyState_Spawning) {
            drawSprite(e->x, e->y, enemyWidth(e), enemyHei
        }
    }
}
```



```
void Game::playerUpdate()
{
    constexpr const uint8_t kPlayerMinX = 1;
    const uint8_t kPlayerMaxX = kScreenWidth - playerWidth() - 1;

    constexpr const uint8_t kPlayerMinY = 1;
    const uint8_t kPlayerMaxY = kScreenHeight - pl

    if (m_device->pressed(LEFT_BUTTON) && m_player
        m_player.x--;
    }

    if (m_device->pressed(RIGHT_BUTTON) && m_playe
        m_player.x++;
    }

    if (m_device->pressed(UP_BUTTON) && m_player.y
        m_player.y--;
    }

    if (m_device->pressed(DOWN_BUTTON) && m_player
        m_player.y++;
    }

    // Cooldown weapon
    if (m_player.weaponCooldown > 0) {
        m_player.weaponCooldown--;
    }
```



```
void Game::playerShoot()
{
    // Player has these weapons:
    // 0: Shoots middle
    // 1: Shoots left and right, alternate
    // 2: Shoots left, middle, and right
    // 3: Above + alternates slight left and right spray
    // 4: Above + shoots left and right spray
    const uint8_t weaponLevel = m_player.items[kStoreItem_WeaponType];
    const bool shootLeft = (m_player.weaponClock % 2) == 0;

    const uint8_t centerX = m_player.x + playerWidth() / 2;
    const uint8_t centerY = m_player.y + playerHeight() / 2;

    // Always does this:
    bulletsSpawn(centerX, centerY, 0, -1 << kBulletFixPointShift, kBulletTy

    // Left and right
    if (weaponLevel > 0) {
        if (shootLeft || weaponLevel >= 2) {
            bulletsSpawn(centerX - 2, centerY + 2, 0, -1 << kBulletFixPoint
        }

        if (!shootLeft || weaponLevel >= 2) {
            bulletsSpawn(centerX + 2, centerY + 2, 0, -1 << kBulletFixPoint
        }
    }
```

Final Stats

- 22x Sprites
- 1x Font Atlas: 6 x 16 glyphs
- 18x levels
- 6 upgrades:
  - Ship / health upgrades, Rate-of-fire upgrades
  - 5 weapon types, Missiles
  - Shield upgrades, 2 drones
- 1x Konami Code easter egg

Final Stats

- Game.h: 488 lines
- Game.cpp: 1,596 lines
- GameSprites.h: 166 lines, 1,343 bytes of sprite data
- GameFont.h: 207 lines, 576 bytes of sprite data
- Final binary:
  - 2,385 bytes for loaded executable, 151 bytes left for stack (93% usage)
  - 18,656 SRAM bundled executable (65% usage)

Thanks for watching!

Questions?