▯ ARCADE SHOOTER Live Coding Session

Step-by-Step Teacher Guide with Runnable Code

Step	What Students Learn	Result After Running
1	Window creation, game loop basics	Empty black window appears
2	Structs, data organization	Green player rectangle visible
3	Keyboard input, movement	Player moves left/right with A/D
4	Arrays, object spawning	Yellow enemies fall from top
5	Shooting mechanics, cooldown	White bullets fire with SPACE
6	Collision detection	Bullets destroy enemies
7	Game state, lives system	Player loses lives on collision
8	Pickups, power-ups	Cyan life drops, weapon upgrades
9	UI text display	Score, lives, weapon shown
10	Sound effects	Laser and explosion sounds
11	Background music	Complete game with music!

Required Files: arial.ttf, laser.wav, explosion.wav, easy_music.wav

```
COMPILE & RUN: g++ -std=c++17 main.cpp -o game -lsfml-graphics -lsfml-window -lsfml-system
-lsfml-audio && ./game
```

STEP 1: EMPTY WINDOW + GAME LOOP

👨‍🏫 TEACHER NOTES:
- Start with the absolute minimum — just a window
- Explain: Game loop runs 60 times per second (60 FPS)
- Show the structure: Event handling → Update → Render
- Ask students: 'What happens if we remove setFramerateLimit?'

Type this code (main.cpp):

```
#include <SFML/Graphics.hpp>
#include <optional>

int main()
{
    sf::RenderWindow window(
        sf::VideoMode({1024, 768}),
        "Arcade Shooter"
    );
    window.setFramerateLimit(60);

    // GAME LOOP
    while (window.isOpen())
    {
        // 1. EVENT HANDLING
        while (const std::optional event = window.pollEvent())
            if (event->is<sf::Event::Closed>())
                window.close();

        // 2. UPDATE (empty for now)

        // 3. RENDER
        window.clear();
        window.display();
    }

    return 0;
}
```

```
COMPILE & RUN: g++ -std=c++17 main.cpp -o game -lsfml-graphics -lsfml-window -lsfml-system
-lsfml-audio && ./game
```

▶ RUN CODE NOW — Show students:
✓ Black empty window appears (1024x768)
✓ Window title shows 'Arcade Shooter'
✓ Close button works (X button)
✓ Window stays open until closed

STEP 2: ADD PLAYER (STRUCT + DRAWING)

👨‍🏫 TEACHER NOTES:

- Introduce struct — grouping related data together
- Explain sf::RectangleShape — SFML's rectangle drawable
- Show coordinate system: (0,0) is TOP-LEFT corner
- Player positioned at bottom center of screen
- Ask: 'Why do we subtract 30 from player X position?'

Add BEFORE main():

```
struct Player
{
    sf::RectangleShape shape;
    float speed{};
    int lives{};
};
```

Add INSIDE main(), before game loop:

```
// Create player
    Player player;
    player.shape.setSize({60.f, 20.f});
    player.shape.setFillColor(sf::Color::Green);
    player.shape.setPosition({1024 / 2.f - 30.f, 768 - 30.f});
    player.speed = 6.f;
    player.lives = 3;
```

Add in RENDER section (after window.clear()):

```
window.draw(player.shape);
```

▶ RUN CODE NOW — Show students:

✓ Green rectangle appears at bottom center

✓ This is our player ship!

✓ It doesn't move yet — that's next step

STEP 3: PLAYER MOVEMENT (KEYBOARD INPUT)

👨‍🏫 TEACHER NOTES:

- Difference: pollEvent() vs isKeyPressed()
- pollEvent = one-time events (key pressed once)
- isKeyPressed = continuous state (holding key down)
- For smooth movement, we need isKeyPressed
- Clamping = keeping value within bounds
- Demo: Remove clamping, show player going off-screen

Add in UPDATE section (before RENDER):

```
// Player movement
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::A))
            player.shape.move({-player.speed, 0});
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::D))
            player.shape.move({player.speed, 0});

        // Keep player on screen (clamping)
        auto pos = player.shape.getPosition();
        if (pos.x < 0) pos.x = 0;
        if (pos.x > 1024 - 60) pos.x = 1024 - 60;
        player.shape.setPosition(pos);
```

▶ RUN CODE NOW — Show students:

✓ Press A — player moves LEFT

✓ Press D — player moves RIGHT

✓ Player stops at screen edges (can't go off-screen)

✓ Movement is smooth (60 updates per second)

STEP 4: ENEMIES (ARRAYS + SPAWNING)

TEACHER NOTES:
- Introduce arrays — multiple objects of same type
- Object pooling pattern: 'active' flag instead of create/destroy
- Why pooling? Avoids memory allocation during gameplay
- Random spawning: std::rand() % 100 == 0 means 1% chance per frame
- Enemies spawn above screen (y = -20) and fall down

Add to includes at top:

```
#include <cstdlib>
#include <ctime>
```

Add struct BEFORE main():

```
struct Enemy
{
    sf::RectangleShape shape;
    float speed{};
    bool active{};
};
```

Add at START of main():

```
std::srand(static_cast<unsigned>(std::time(nullptr)));
```

Add after player creation:

```
// Create enemy array
    Enemy enemies[6];
    for (auto& e : enemies)
    {
        e.shape.setSize({40.f, 20.f});
        e.shape.setFillColor(sf::Color::Yellow);
        e.active = false;
    }
```

Add in UPDATE section:

```
// Enemy spawning and movement
        for (auto& e : enemies)
        {
            if (!e.active && std::rand() % 100 == 0)
            {
                e.active = true;
                e.shape.setPosition({float(std::rand() % 984), -20});
                e.speed = 2.f;
            }
            else if (e.active)
            {
                e.shape.move({0, e.speed});
                if (e.shape.getPosition().y > 768)
                    e.active = false;
            }
        }
```

Add in RENDER section:

```
for (auto& e : enemies)
            if (e.active) window.draw(e.shape);
```

▶ RUN CODE NOW — Show students:

✓ Yellow rectangles spawn randomly at top

✓ They fall downward continuously

✓ When they exit bottom, they deactivate

✓ New enemies spawn randomly over time

STEP 5: SHOOTING (BULLETS + COOLDOWN)

👨‍🏫 TEACHER NOTES:

- sf::Clock for timing — measures elapsed time
- Cooldown prevents shooting too fast (250ms between shots)
- Bullets also use object pooling pattern
- Bullet spawns at player position, moves upward
- Ask: 'Why do we need cooldown? What happens without it?'

Add struct BEFORE main():

```
struct Bullet
{
    sf::RectangleShape shape;
    float speed{};
    bool active{};
};
```

Add after enemy array creation:

```
// Bullet array and fire timer
    Bullet bullets[20];
    for (auto& b : bullets) b.active = false;
    sf::Clock fireClock;
```

Add in UPDATE section:

```
// Shooting
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Space) &&
            fireClock.getElapsedTime().asMilliseconds() > 250)
        {
            fireClock.restart();
            for (auto& b : bullets)
                if (!b.active)
                {
                    b.active = true;
                    b.speed = 10.f;
                    b.shape.setSize({5.f, 15.f});
                    b.shape.setFillColor(sf::Color::White);
                    b.shape.setPosition({
                        player.shape.getPosition().x + 27.f,
                        player.shape.getPosition().y - 15.f
                    });
                    break;
                }
        }

        // Move bullets
        for (auto& b : bullets)
            if (b.active)
            {
                b.shape.move({0, -b.speed});
                if (b.shape.getPosition().y < 0)
                    b.active = false;
            }
```

Add in RENDER section:

```
for (auto& b : bullets)
            if (b.active) window.draw(b.shape);
```

▶ RUN CODE NOW — Show students:

✓ Press SPACE — white bullet fires upward

✓ Can't spam — 250ms cooldown between shots

✓ Bullets disappear when exiting top of screen

✓ Multiple bullets can be active at once

STEP 6: COLLISION DETECTION

TEACHER NOTES:
- AABB collision: Axis-Aligned Bounding Box
- findIntersection() returns overlap rectangle if collision
- In boolean context: true = collision, false = no collision
- We check every bullet against every enemy (nested loop)
- Both objects deactivate on collision
- Add score variable to track hits

Add after player creation:

```
int score = 0;
```

Add in UPDATE section (after bullet movement):

```
// Bullet-Enemy collision
        for (auto& b : bullets)
            for (auto& e : enemies)
                if (b.active && e.active &&
                    b.shape.getGlobalBounds().findIntersection(
                        e.shape.getGlobalBounds()))
                {
                    b.active = false;
                    e.active = false;
                    score++;
                }
```

▶ RUN CODE NOW — Show students:
✓ Shoot at enemies — they disappear on hit!
✓ Bullet also disappears (doesn't pass through)
✓ Score increases (we'll display it later)
✓ Multiple enemies can be destroyed

STEP 7: PLAYER DAMAGE (LIVES SYSTEM)

□□ TEACHER NOTES:
- Same collision detection, different outcome
- Enemy hitting player = lose a life
- When lives reach 0, game ends (window closes)
- Enemy deactivates after hitting player
- Demo: Let enemy hit player, show game ending

Add in UPDATE section (after bullet-enemy collision):

```
// Enemy-Player collision
        for (auto& e : enemies)
            if (e.active &&
                e.shape.getGlobalBounds().findIntersection(
                    player.shape.getGlobalBounds()))
            {
                e.active = false;
                player.lives--;
                if (player.lives <= 0)
                    window.close();
            }
```

▶ RUN CODE NOW — Show students:
✓ Let enemy touch player — enemy disappears
✓ Player has 3 lives (invisible for now)
✓ After 3 hits, game window closes
✓ This is our basic game over condition

STEP 8: PICKUPS (LIFE + WEAPONS)

▯▯ TEACHER NOTES:

- Two new struct types: LifeDrop and WeaponDrop
- enum class WeaponType — type-safe enumeration
- Pickups spawn rarely (1/500 chance per frame)
- Different colors indicate different weapon types
- Collecting pickup changes game state

Add structs BEFORE main():

```
enum class WeaponType { Single, Burst, Heavy };

struct LifeDrop
{
    sf::CircleShape shape;
    bool active{};
    float speed{};
};

struct WeaponDrop
{
    sf::RectangleShape shape;
    bool active{};
    float speed{};
    WeaponType type{};
};
```

Add after bullet array:

```
WeaponType currentWeapon = WeaponType::Single;

    LifeDrop lifeDrop;
    lifeDrop.shape.setRadius(10.f);
    lifeDrop.shape.setFillColor(sf::Color::Cyan);
    lifeDrop.speed = 2.f;
    lifeDrop.active = false;

    WeaponDrop weaponDrop;
    weaponDrop.shape.setSize({20.f, 20.f});
    weaponDrop.speed = 2.2f;
    weaponDrop.active = false;
```

Add in UPDATE section:

```
// Life drop
        if (!lifeDrop.active && std::rand() % 500 == 0)
        {
            lifeDrop.active = true;
            lifeDrop.shape.setPosition({float(std::rand() % 1004), -20});
        }
        if (lifeDrop.active)
        {
            lifeDrop.shape.move({0, lifeDrop.speed});
            if (lifeDrop.shape.getPosition().y > 768) lifeDrop.active = false;
        }

        // Weapon drop
        if (!weaponDrop.active && std::rand() % 600 == 0)
        {
            weaponDrop.active = true;
            weaponDrop.shape.setPosition({float(std::rand() % 1004), -20});
            int w = std::rand() % 3;
            weaponDrop.type = static_cast<WeaponType>(w);
            if (w == 0) weaponDrop.shape.setFillColor(sf::Color::Cyan);
            else if (w == 1) weaponDrop.shape.setFillColor(sf::Color::Magenta);
            else weaponDrop.shape.setFillColor(sf::Color::Red);
        }
        if (weaponDrop.active)
        {
            weaponDrop.shape.move({0, weaponDrop.speed});
            if (weaponDrop.shape.getPosition().y > 768) weaponDrop.active = false;
        }

        // Pickup collisions
        if (lifeDrop.active &&
            lifeDrop.shape.getGlobalBounds().findIntersection(
                player.shape.getGlobalBounds()))
        {
            lifeDrop.active = false;
            player.lives++;
        }
        if (weaponDrop.active &&
            weaponDrop.shape.getGlobalBounds().findIntersection(
                player.shape.getGlobalBounds()))
        {
            currentWeapon = weaponDrop.type;
            weaponDrop.active = false;
        }
```

Add in RENDER section:

```
if (lifeDrop.active) window.draw(lifeDrop.shape);
        if (weaponDrop.active) window.draw(weaponDrop.shape);
```

▶ RUN CODE NOW — Show students:

✓ Cyan circles = life pickups (gives +1 life)

✓ Colored squares = weapon pickups

✓ Cyan = Single, Magenta = Burst, Red = Heavy

✓ Touch pickup to collect it

STEP 8b: WEAPON TYPES (UPDATE SHOOTING)

👨‍🏫 TEACHER NOTES:

- Now make weapons actually different!
- Single = 1 bullet, Burst = 3 bullets, Heavy = 1 big slow bullet
- Lambda function for bullet spawning (code reuse)
- Demo each weapon type after collecting pickup

REPLACE the shooting code with this:

```
// Shooting with weapon types
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Space) &&
            fireClock.getElapsedTime().asMilliseconds() > 250)
        {
            fireClock.restart();

            auto spawn = [&](float offset, float spd, sf::Vector2f size)
            {
                for (auto& b : bullets)
                    if (!b.active)
                    {
                        b.active = true;
                        b.speed = spd;
                        b.shape.setSize(size);
                        b.shape.setFillColor(sf::Color::White);
                        b.shape.setPosition({
                            player.shape.getPosition().x + offset,
                            player.shape.getPosition().y - size.y
                        });
                        break;
                    }
            };

            if (currentWeapon == WeaponType::Single)
                spawn(27.f, 10.f, {5.f, 15.f});
            else if (currentWeapon == WeaponType::Burst)
            {
                spawn(10.f, 10.f, {5.f, 15.f});
                spawn(27.f, 10.f, {5.f, 15.f});
                spawn(44.f, 10.f, {5.f, 15.f});
            }
            else
                spawn(23.f, 6.f, {14.f, 20.f});
        }
```

▶ RUN CODE NOW — Show students:

✓ Single (default) = 1 small fast bullet

✓ Burst = 3 bullets spread horizontally

✓ Heavy = 1 big slow bullet

✓ Collect weapon pickups to change weapon!

STEP 9: UI TEXT DISPLAY

TEACHER NOTES:
- sf::Font loads font file (arial.ttf needed)
- sf::Text displays text on screen
- Update text every frame with current values
- Position text in top-left corner
- Error handling: check if font loaded successfully

Add to includes:

```
#include <string>
```

Add after score variable:

```
sf::Font font;
    sf::Text scoreText(font), livesText(font), weaponText(font);

    if (font.openFromFile("arial.ttf"))
    {
        scoreText.setCharacterSize(22);
        scoreText.setFillColor(sf::Color::White);
        scoreText.setPosition({10.f, 10.f});

        livesText.setCharacterSize(22);
        livesText.setFillColor(sf::Color::White);
        livesText.setPosition({10.f, 40.f});

        weaponText.setCharacterSize(22);
        weaponText.setFillColor(sf::Color::White);
        weaponText.setPosition({10.f, 70.f});
    }
```

Add at END of UPDATE section:

```
// Update UI text
        scoreText.setString("Score: " + std::to_string(score));
        livesText.setString("Lives: " + std::to_string(player.lives));
        std::string wname = (currentWeapon == WeaponType::Single) ? "Single" :
                           (currentWeapon == WeaponType::Burst) ? "Burst" : "Heavy";
        weaponText.setString("Weapon: " + wname);
```

Add in RENDER section:

```
window.draw(scoreText);
        window.draw(livesText);
        window.draw(weaponText);
```

▶ RUN CODE NOW — Show students:

✓ **Top-left shows: Score, Lives, Weapon**

✓ **Score increases when hitting enemies**

✓ **Lives decrease when hit, increase with pickup**

✓ **Weapon name changes when collecting weapon drop**

STEP 10: SOUND EFFECTS

👨‍🏫 TEACHER NOTES:
• sf::SoundBuffer = container for audio data
• sf::Sound = playable sound object
• Sound needs buffer to exist (lifetime dependency)
• setVolume() controls loudness (0-100)
• play() starts the sound immediately

Change first include to:

```
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
```

Add after font setup:

```
sf::SoundBuffer shootBuffer, hitBuffer;
    shootBuffer.loadFromFile("laser.wav");
    hitBuffer.loadFromFile("explosion.wav");

    sf::Sound shootSound(shootBuffer), hitSound(hitBuffer);
    shootSound.setVolume(10.f);
    hitSound.setVolume(10.f);
```

Add in shooting section (after fireClock.restart()):

```
shootSound.play();
```

Add in bullet-enemy collision (after score++):

```
hitSound.play();
```

▶ RUN CODE NOW — Show students:
✓ Shooting makes 'pew' laser sound
✓ Hitting enemies makes explosion sound
✓ Game feels much more alive with audio!

STEP 11: BACKGROUND MUSIC (FINAL!)

👨‍🏫 TEACHER NOTES:
• sf::Music streams from file (for long audio)
• sf::Sound loads entire file to memory (for short sounds)
• Music loops manually by checking status
• Game is now COMPLETE! Celebrate with students!
• Discuss possible extensions and improvements

Add after sound setup:

```
sf::Music music;
    if (music.openFromFile("easy_music.wav"))
    {
        music.setVolume(15.f);
        music.play();
    }
```

Add at START of game loop (before event handling):

```
// Loop background music
        if (music.getStatus() == sf::SoundSource::Status::Stopped)
            music.play();
```

▶ RUN CODE NOW — Show students:
✓ Background music plays continuously
✓ Music loops when it ends
✓ 🎉 GAME IS COMPLETE! 🎉
✓ Full arcade shooter with all features working!

Feature	What We Learned
Window + Loop	sf::RenderWindow, 60 FPS, event handling
Player	struct, sf::RectangleShape, positioning
Movement	sf::Keyboard::isKeyPressed, clamping
Enemies	Arrays, object pooling, random spawning
Bullets	sf::Clock cooldown, movement
Collision	getGlobalBounds(), findIntersection()
Lives	Game state, game over condition
Pickups	enum class, type-safe enumerations
UI	sf::Font, sf::Text, string conversion
Audio	sf::SoundBuffer, sf::Sound, sf::Music