FPS Aim Trainer

Class: Programming II – COP3003
Professor: Deepa Devasenapathy
Florida Gulf Coast University

by Xavier Seron

Content

- Abstract
- Goal of the Project
- Software Requirements
- Introduction
- Why You Chose This Project?
- Existing Technique & Improvisations
- Project Concept & Procedure
- Result Discussion
- Conclusion
- Appendix & References

Abstract

We built an FPS Aim Trainer in C++ using SDL3 and Visual Studio 2022. It features a modular state machine with a main menu, Gridshot and Tracking modes, settings, and credits. Gridshot challenges you to click timed targets; Tracking tests continuous accuracy, both include streaks and an optional challenge mode. A settings panel lets you tweak sensitivity, FOV, and crosshair appearance, with all preferences and high scores saved via JSON.

Goal of the Project

- Implement two game modes in C++/SDL3.
- Demonstrate OOP concepts: inheritance, polymorphism, abstraction.
- Provide user-friendly GUI & persistent high scores.

Software Requirements

- Visual Studio 2022 (C++17)
- SDL 3.2.10
- Windows 11 (haven't tested any other OS)
- 4GB of ram

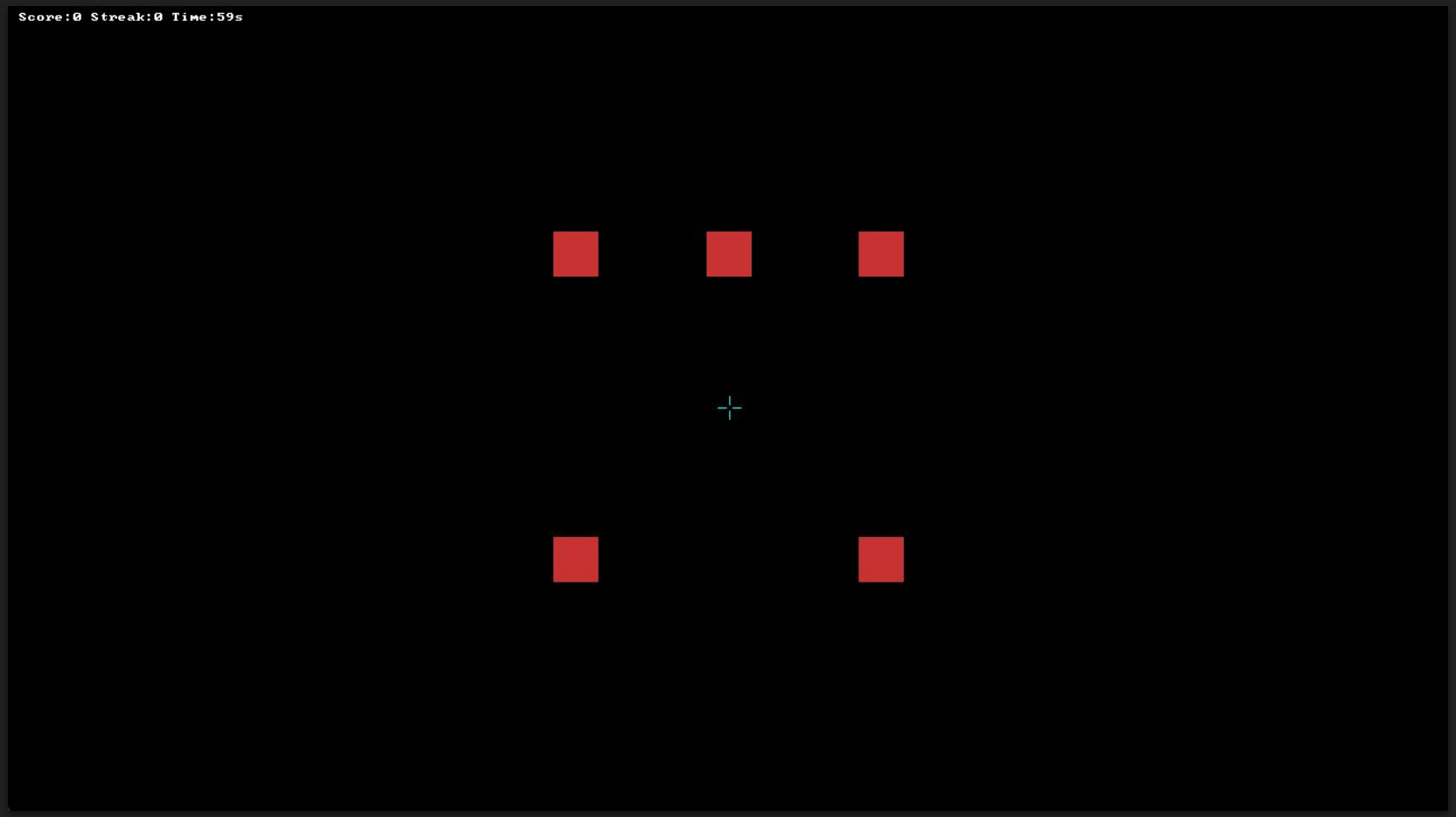
Introduction

For this project, we set out to build a simple but fully featured FPS aim trainer using C++ and SDL3 in Visual Studio 2022. In the next few slides, we'll cover:

- Goals & Motivation: Why an aim trainer and what we hoped to learn
- Core Features: Two practice modes (Gridshot and Tracking), streak bonuses, challenge toggle
- Customization & Persistence: Slider‑driven settings panel and JSON‑backed save/load of your preferences and high scores
- Technical Highlights: State‑driven class design, relative mouse control, lightweight JSON parsing, and clean event‑loop architecture

Why We Chose This Project?

- We play video games and are all considering the game design field as a future career path.
- Wanted to challenge ourselves with something unique that didn’t already have a youtube tutorial made.
- Combine real-time rendering with data persistence.
- Create a fun tool that simulates an FPS game without making an FPS game. It helps players improve aim through measurable exercises.
- Apply full OOP curriculum in a fun, interactive tool.



Gridshot -

Gameplay: 3×3 grid; initially 5 (or 2 in challenge mode) targets “active” at random.

Flow: Countdown phase (3 s big number) | Running (60 s timer) | On click, checks if aim (yaw/pitch converted to screen coords) is within angular tolerance ⟶ score++ & streak++ ⟶ respawn a new dot. Miss resets streak.

When time’s up, pushes score into config.gridshotScores and returns to main menu.

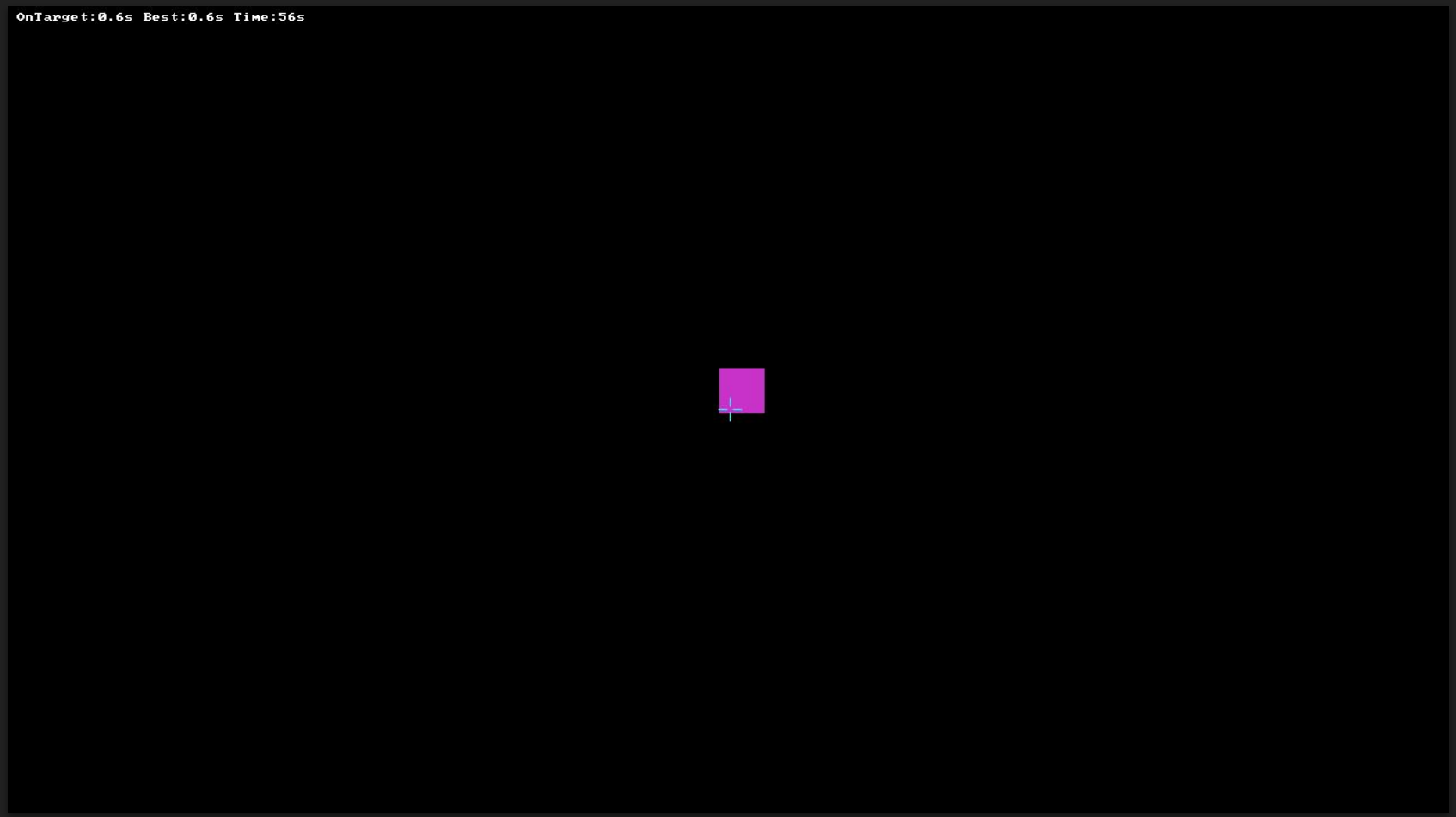
Rendering:

Clears screen black, shows countdown or active targets as red squares.

Draws configurable crosshair in center.

HUD: “Score:…, Streak:…, Time:…” plus “CHALLENGE MODE” if on.

```cpp
// --- GridshotMode ---
class GridshotMode : public GameMode {
    struct Target { double yaw, pitch;bool active; } t[9];
    int score = 0, streak = 0;
    Uint32 timeRem = 0, countdown = 0;
    bool running = false, challengeMode = false;
    SDL_Color tgtCol{ 200,50,50,255 };
public:
    double targAngRad = 2.0;
    int    targPixRad = 20;

    bool isInCountdown()const { return countdown > 0; }
    bool isRunning()    const { return running; }
    int  getScore()     const { return score; }

    void start()override {
        score = streak = 0;
        timeRem = GAME_DURATION_MS;
        countdown = COUNTDOWN_DURATION_MS;
        running = true;
        std::vector<int> idx(9);
        for (int i = 0;i < 9;++i) idx[i] = i;
        std::random_shuffle(idx.begin(), idx.end());
        int initial = challengeMode ? 2 : 5;
        for (int i = 0;i < 9;++i) {
            t[i].active = (i < initial);
            if (i < initial) {
                int row = idx[i] / 3, col = idx[i] % 3;
                double span = 30.0;
                t[i].yaw = (col - 1) * (span / 2.0);
                t[i].pitch = (1 - row) * (span / 2.0);
            }
        }
    }

    bool handleClick(double cy, double cp) {
        for (int i = 0;i < 9;++i) {
            if (!t[i].active) continue;
            double dy = t[i].yaw - cy;
            if (dy > 180) dy -= 360; else if (dy < -180) dy += 360;
            double dp = t[i].pitch - cp;
            if (fabs(dy) <= targAngRad && fabs(dp) <= targAngRad) {
                score++;streak++;
                t[i].active = false;
            }
        }
        streak = 0;
        return false;
    }

    void update(Uint32 d, double cy, double cp) {
        if (!running) return;
        if (countdown > 0) {
            countdown = (d > countdown ? 0 : countdown - d);
            return;
        }
        if (timeRem > 0) {
            timeRem = (d > timeRem ? 0 : timeRem - d);
            if (timeRem == 0) running = false;
        }
    }

    void render(SDL_Renderer* ren, double cy, double cp) {
        SDL_SetRenderDrawColor(ren, 0, 0, 0, 255);
        SDL_RenderClear(ren);
        if (countdown > 0) {
            int sec = (countdown + 500) / 1000;
            char buf[8];sprintf(buf, "%d", sec);
            SDL_SetRenderScale(ren, 4.0f, 4.0f);
            SDL_RenderDebugText(ren,
                WINDOW_WIDTH / 8 - 4, WINDOW_HEIGHT / 8 - 8, buf);
            SDL_SetRenderScale(ren, 1.0f, 1.0f);
            return;
        }
        double hF = config.fov, asp = double(WINDOW_WIDTH) / WINDOW_HEIGHT;
        double vF = (180.0 / M_PI) * 2 * atan(
            tan(hF * M_PI / 180.0 / 2) * (1.0 / asp));
        int boxRad = challengeMode
            ? (targPixRad / 2)
            : targPixRad;
        for (int i = 0;i < 9;++i) {
            if (!t[i].active) continue;
            double dy = t[i].yaw - cy;
            if (dy > 180) dy -= 360; else if (dy < -180) dy += 360;
            double dp = t[i].pitch - cp;
            if (fabs(dy) > hF / 2 + 5 || fabs(dp) > vF / 2 + 5) continue;
            double xN = tan(dy * M_PI / 180.0) / tan(hF * M_PI / 180.0 / 2);
            double yN = tan(dp * M_PI / 180.0) / tan(vF * M_PI / 180.0 / 2);
            int x = int(xN * (WINDOW_WIDTH / 2) + WINDOW_WIDTH / 2);
            int y = int(-yN * (WINDOW_HEIGHT / 2) + WINDOW_HEIGHT / 2);
            drawRect(ren,
                x - boxRad, y - boxRad,
                2 * boxRad, 2 * boxRad,
                tgtCol);
        }
```



Tracking -

Gameplay: single moving target that bounces within your FOV.

Flow: Same 3 s countdown → 60 s run | Target moves by velocities yv, pv, speed doubled in challenge mode.

If your crosshair is on target, you accumulate “OnTarget” time and streak. Best streak recorded.
At end, saves to config.trackingScores.

Rendering:

Clears screen black, shows countdown or active target a magenta square.

Draws configurable crosshair in center.

HUD: “Score:…, Streak:…, Time:…” plus “CHALLENGE MODE” if on.

```
// --- TrackingMode ---
class TrackingMode : public GameMode {
    double yaw = 0, pitch = 0, yv = 20, pv = 15;
    double score = 0, streak = 0, best = 0;
    Uint32 timeRem = 0, countdown = 0;
    bool running = false, challengeMode = false;
    SDL_Color tgtCol{ 200,50,200,255 };
public:
    double targAngRad = 3.0;
    int    targPixRad = 20;
    bool isInCountdown()const { return countdown > 0; }
    bool isRunning()    const { return running; }
    double getScore()   const { return score; }
    void start() override {
        score = streak = best = 0;
        timeRem = GAME_DURATION_MS;
        countdown = COUNTDOWN_DURATION_MS;
        running = true;
        yaw = pitch = 0;
        yv = 20 * ((rand() % 2) ? 1 : -1);
        pv = 15 * ((rand() % 2) ? 1 : -1);
    }
    void update(Uint32 d, double cy, double cp) {
        if (!running) return;
        if (countdown > 0) {
            countdown = (d > countdown ? 0 : countdown - d);
            return;
        }
        double dt = d / 1000.0;
        double factor = challengeMode ? 2.0 : 1.0;
        yaw += yv * factor * dt;
        pitch += pv * factor * dt;
        double hF = config.fov, asp = double(WINDOW_WIDTH) / WINDOW_HEIG
        double vF = (180.0 / M_PI) * 2 * atan(
            tan(hF * M_PI / 180.0 / 2) * (1.0 / asp));
        double maxY = hF / 2 - 5, maxP = vF / 2 - 5;
        if (yaw < -maxY) { yaw = -maxY; yv = -yv; }
        if (yaw > maxY) { yaw = maxY; yv = -yv; }
        if (pitch < -maxP) { pitch = -maxP; pv = -pv; }
        if (pitch > maxP) { pitch = maxP; pv = -pv; }
        double dy = yaw - cy; if (dy > 180) dy -= 360; else if (dy < -18
        double dp = pitch - cp;
        bool on = fabs(dy) <= targAngRad && fabs(dp) <= targAngRad;
        if (on) {
            score += dt * factor;
            streak += dt * factor;
            if (streak > best) best = streak;
        }
        else streak = 0;
```

```
    void render(SDL_Renderer* ren, double cy, double cp) {
        SDL_SetRenderDrawColor(ren, 0, 0, 0, 255);
        SDL_RenderClear(ren);
        if (countdown > 0) {
            int sec = (countdown + 500) / 1000;
            char buf[8];sprintf(buf, "%d", sec);
            SDL_SetRenderScale(ren, 4.0f, 4.0f);
            SDL_RenderDebugText(ren,
                WINDOW_WIDTH / 8 - 4, WINDOW_HEIGHT / 8 - 8, buf);
            SDL_SetRenderScale(ren, 1.0f, 1.0f);
            return;
        }
        double dy = yaw - cy; if (dy > 180) dy -= 360; else if (dy < -180) dy
        double dp = pitch - cp;
        double hF = config.fov, asp = double(WINDOW_WIDTH) / WINDOW_HEIGHT;
        double vF = (180.0 / M_PI) * 2 * atan(
            tan(hF * M_PI / 180.0 / 2) * (1.0 / asp));
        if (fabs(dy) <= hF / 2 && fabs(dp) <= vF / 2) {
            double xN = tan(dy * M_PI / 180.0) / tan(hF * M_PI / 180.0 / 2);
            double yN = tan(dp * M_PI / 180.0) / tan(vF * M_PI / 180.0 / 2);
            int x = int(xN * (WINDOW_WIDTH / 2) + WINDOW_WIDTH / 2);
            int y = int(-yN * (WINDOW_HEIGHT / 2) + WINDOW_HEIGHT / 2);
            drawRect(ren,
                x - targPixRad, y - targPixRad,
                2 * targPixRad, 2 * targPixRad,
                tgtCol);
        }
        SDL_Color cc{ (Uint8)config.cross_r,
                      (Uint8)config.cross_g,
                      (Uint8)config.cross_b,255 };
        int gap = config.cross_gap, len = config.cross_len;
        drawLine(ren, WINDOW_WIDTH / 2 - len, WINDOW_HEIGHT / 2,
            WINDOW_WIDTH / 2 - gap, WINDOW_HEIGHT / 2, cc);
        drawLine(ren, WINDOW_WIDTH / 2 + gap, WINDOW_HEIGHT / 2,
            WINDOW_WIDTH / 2 + len, WINDOW_HEIGHT / 2, cc);
        drawLine(ren, WINDOW_WIDTH / 2, WINDOW_HEIGHT / 2 - len,
            WINDOW_WIDTH / 2, WINDOW_HEIGHT / 2 - gap, cc);
        drawLine(ren, WINDOW_WIDTH / 2, WINDOW_HEIGHT / 2 + gap,
            WINDOW_WIDTH / 2, WINDOW_HEIGHT / 2 + len, cc);
        SDL_SetRenderDrawColor(ren, 255, 255, 255, 255);
        char hud[64];
        sprintf(hud, "OnTarget:%.1fs Best:%.1fs Time:%ds",
            score, best, timeRem / 1000);
        SDL_RenderDebugText(ren, 10, 10, hud);
        if (challengeMode)
            SDL_RenderDebugText(ren, 10, 30, "CHALLENGE MODE");
    }
    void toggleChallengeMode() override {
        challengeMode = !challengeMode;
    }
```

Tracking

Sliders for mouse sensitivity, FOV, crosshair gap/length, and R/G/B color. | Knobs you can drag; numeric values update in real time and show as text.

Buttons:

Apply: copies all slider values into config, toggles challenge mode, and calls saveConfig
Reset: reverts sliders to last-saved config values | Challenge Mode toggle button.

```
};

// --- SettingsMenu ---
class SettingsMenu {
public:
    float sensVal, fovVal;
    int gapVal, lenVal, rVal, gVal, bVal;
    bool challengeVal;
    Rect sensBar, fovBar, gapBar, lenBar, rBar, gBar, bBar;
    Rect sensKnob, fovKnob, gapKnob, lenKnob, rKnob, gKnob, bKnob;
    Rect applyBtn, resetBtn, challengeBtn;
    int dragging, hoverBtn;
    bool hoverChallenge;

    SettingsMenu() {
        sensVal = config.sensitivity;
        fovVal = config.fov;
        gapVal = config.cross_gap;
        lenVal = config.cross_len;
        rVal = config.cross_r;
        gVal = config.cross_g;
        bVal = config.cross_b;
        challengeVal = config.challengeMode;

        sensBar = { WINDOW_WIDTH / 2 - 150,WINDOW_HEIGHT / 2 - 80,300,6 };
        fovBar = { WINDOW_WIDTH / 2 - 150,WINDOW_HEIGHT / 2 - 40,300,6 };
        gapBar = { WINDOW_WIDTH / 2 - 150,WINDOW_HEIGHT / 2,   300,6 };
        lenBar = { WINDOW_WIDTH / 2 - 150,WINDOW_HEIGHT / 2 + 40,300,6 };
        rBar = { WINDOW_WIDTH / 2 - 150,WINDOW_HEIGHT / 2 + 80,300,6 };
        gBar = { WINDOW_WIDTH / 2 - 150,WINDOW_HEIGHT / 2 + 120,300,6 };
        bBar = { WINDOW_WIDTH / 2 - 150,WINDOW_HEIGHT / 2 + 160,300,6 };

        sensKnob.w = sensKnob.h =
            fovKnob.w = fovKnob.h =
            gapKnob.w = gapKnob.h =
            lenKnob.w = lenKnob.h =
            rKnob.w = rKnob.h =
            gKnob.w = gKnob.h =
            bKnob.w = bKnob.h = 12;

        applyBtn = { WINDOW_WIDTH / 2 - 100,WINDOW_HEIGHT / 2 + 200,80,30 };
        resetBtn = { WINDOW_WIDTH / 2 + 20, WINDOW_HEIGHT / 2 + 200,80,30 };
        challengeBtn = { WINDOW_WIDTH / 2 - 100,WINDOW_HEIGHT / 2 + 250,200,30 };

        dragging = 0;hoverBtn = -1;hoverChallenge = false;
        updateKnobs();
    }

    void updateKnobs() {
        auto place = [&](const Rect& bar, Rect& knob, float val, float mn, float mx) {
            float n = (val - mn) / (mx - mn);
            n = CLAMP(n, 0.f, 1.f);
            knob.x = bar.x + int(n * bar.w) - knob.w / 2;
            knob.y = bar.y - knob.h / 2 + bar.h / 2;
            };
        place(sensBar, sensKnob, sensVal, 0.001f, 3.0f);
        place(fovBar, fovKnob, fovVal, 60.0f, 130.0f);
        place(gapBar, gapKnob, float(gapVal), 0.0f, 50.0f);
        place(lenBar, lenKnob, float(lenVal), 0.0f, 100.0f);
        place(rBar, rKnob, float(rVal), 0.0f, 255.0f);
        place(gBar, gKnob, float(gVal), 0.0f, 255.0f);
        place(bBar, bKnob, float(bVal), 0.0f, 255.0f);
    }

    void handleMouseDown(int mx, int my) {
        auto start = [&](const Rect& bar, const Rect& knob, int code) {
            if (pointInRect(mx, my, bar) || pointInRect(mx, my, knob)) {
                dragging = code;return true;
            }
            return false;
            };
        if (start(sensBar, sensKnob, 1))return;
        if (start(fovBar, fovKnob, 2))return;
        if (start(gapBar, gapKnob, 3))return;
        if (start(lenBar, lenKnob, 4))return;
        if (start(rBar, rKnob, 5))return;
        if (start(gBar, gKnob, 6))return;
        if (start(bBar, bKnob, 7))return;

        if (pointInRect(mx, my, applyBtn))hoverBtn = 1;
        else if (pointInRect(mx, my, resetBtn))hoverBtn = 2;
        else if (pointInRect(mx, my, challengeBtn)) {
            challengeVal = !challengeVal;
            return;
        }
    }
    void handleMouseUp() { dragging = 0; }
    void handleMouseMove(int mx, int my) {
        hoverBtn = -1;
        hoverChallenge = pointInRect(mx, my, challengeBtn);
        if (pointInRect(mx, my, applyBtn))hoverBtn = 1;
        else if (pointInRect(mx, my, resetBtn))hoverBtn = 2;

        if (dragging == 1) {
            float n = (mx - sensBar.x) / float(sensBar.w);
            n = CLAMP(n, 0.f, 1.f);
            float lm = log10f(0.001f), lM = log10f(3.0f);
            sensVal = powf(10.0f, lm + n * (lM - lm));
```

```
    if (dragging == 1) {
        float n = (mx - sensBar.x) / float(sensBar.w);
        n = CLAMP(n, 0.f, 1.f);
        float lm = log10f(0.001f), lM = log10f(3.0f);
        sensVal = powf(10.0f, lm + n * (lM - lm));
        updateKnobs();
    }
    if (dragging == 2) {
        float n = (mx - fovBar.x) / float(fovBar.w);
        n = CLAMP(n, 0.f, 1.f);
        fovVal = 60.0f + n * 70.0f;
        updateKnobs();
    }
    auto setVal = [&](int code, int mx, const Rect& bar, 
        if (dragging != code)return;
        float n = (mx - bar.x) / float(bar.w);
        n = CLAMP(n, 0.f, 1.f);
        out = int(mn + n * (mxv - mn));
        updateKnobs();
        };
    setVal(3, mx, gapBar, gapVal, 0.0f, 50.0f);
    setVal(4, mx, lenBar, lenVal, 0.0f, 100.0f);
    setVal(5, mx, rBar, rVal, 0.0f, 255.0f);
    setVal(6, mx, gBar, gVal, 0.0f, 255.0f);
    setVal(7, mx, bBar, bVal, 0.0f, 255.0f);
}

void apply() {
    config.sensitivity = sensVal;
    config.fov = fovVal;
    config.cross_gap = gapVal;
    config.cross_len = lenVal;
    config.cross_r = rVal;
    config.cross_g = gVal;
    config.cross_b = bVal;
    config.challengeMode = challengeVal;
    JSONStorage::saveConfig(config);
}

void reset() {
    sensVal = config.sensitivity;
    fovVal = config.fov;
    gapVal = config.cross_gap;
    lenVal = config.cross_len;
    rVal = config.cross_r;
    gVal = config.cross_g;
    bVal = config.cross_b;
    challengeVal = config.challengeMode;
    updateKnobs();
}
```

```
void render(SDL_Renderer* ren) {
    SDL_SetRenderDrawColor(ren, 0, 0, 0, 255);
    SDL_RenderClear(ren);
    SDL_SetRenderDrawColor(ren, 255, 255, 255, 255);
    char buf[64];
    // Sensitivity
    SDL_RenderDebugText(ren,
        sensBar.x, sensBar.y - 15, "Mouse Sensitivity");
    drawRect(ren,
        sensBar.x, sensBar.y, sensBar.w, sensBar.h, { 200,200,200,255 });
    drawRect(ren,
        sensKnob.x, sensKnob.y, sensKnob.w, sensKnob.h, { 255,255,255,255 });
    sprintf(buf, "%.3f", sensVal);
    SDL_RenderDebugText(ren,
        sensBar.x + sensBar.w + 10, sensBar.y - 4, buf);
    // FOV
    SDL_RenderDebugText(ren,
        fovBar.x, fovBar.y - 15, "Field of View");
    drawRect(ren,
        fovBar.x, fovBar.y, fovBar.w, fovBar.h, { 200,200,200,255 });
    drawRect(ren,
        fovKnob.x, fovKnob.y, fovKnob.w, fovKnob.h, { 255,255,255,255 });
    sprintf(buf, "%.0f", fovVal);
    SDL_RenderDebugText(ren,
        fovBar.x + fovBar.w + 10, fovBar.y - 4, buf);
    // Gap
    SDL_RenderDebugText(ren,
        gapBar.x, gapBar.y - 15, "Crosshair Gap");
    drawRect(ren,
        gapBar.x, gapBar.y, gapBar.w, gapBar.h, { 200,200,200,255 });
    drawRect(ren,
        gapKnob.x, gapKnob.y, gapKnob.w, gapKnob.h, { 255,255,255,255 });
    sprintf(buf, "%d", gapVal);
    SDL_RenderDebugText(ren,
        gapBar.x + gapBar.w + 10, gapBar.y - 4, buf);
    // Length
    SDL_RenderDebugText(ren,
        lenBar.x, lenBar.y - 15, "Crosshair Length");
    drawRect(ren,
        lenBar.x, lenBar.y, lenBar.w, lenBar.h, { 200,200,200,255 });
    drawRect(ren,
        lenKnob.x, lenKnob.y, lenKnob.w, lenKnob.h, { 255,255,255,255 });
    sprintf(buf, "%d", lenVal);
    SDL_RenderDebugText(ren,
        lenBar.x + lenBar.w + 10, lenBar.y - 4, buf);
    // R
    SDL_RenderDebugText(ren,
        rBar.x, rBar.y - 15, "Crosshair R");
    drawRect(ren,
        rBar.x, rBar.y, rBar.w, rBar.h, { 200,200,200,255 });
```

Credits

Simple screen showing version info and authors; click or key any to return to main menu.

main() & Game Loop

Key Parts

1. **Modular design**: Each screen/mode is encapsulated in its own class implementing a simple interface.

2. **Config & persistence**: JSON saved/loaded at start/end of each session; easy to extend for new settings.

3. **Immediate feedback**: Settings sliders show live values; main menu shows best scores.

4. **Relative mouse mode**: Used during gameplay for FPS-style look control, toggled on/off per state.

SDL Initialization -

SDL_SetMainReady();

SDL_Init(SDL_INIT_VIDEO);

SDL_CreateWindowAndRenderer(..., &window,&renderer);

Load Config via JSONStorage::loadConfig(config), enforce minimum sensible values.

Instantiate one of each screen/mode -

MainMenu menu; GridshotMode grid; TrackingMode track;

SettingsMenu settings; CreditsScreen credits;

State Machine -

enum State { MAIN, GRID, TRACK, SETT, CRED } state = MAIN;

Event Handling & Transitions -

In MAIN, mouse clicks on buttons switch state → start corresponding mode & enable/disable relative mouse mode.

In GRID/TRACK, mouse movement adjusts camera yaw/pitch, left-click triggers hits (Gridshot), Esc returns to MAIN.

In SETT, routing of drag, hover, clicks for sliders/buttons.

In CRED, any click or key returns to MAIN.

Per-Frame Updates - Call grid.update() or track.update() if running; when they finish, push score into config, save, and return to MAIN.

Rendering - Depending on state, call the screen/mode's render method.

Present with SDL_RenderPresent, cap frame with a tiny SDL_Delay(1).

Cleanup -

SDL_DestroyRenderer(renderer);

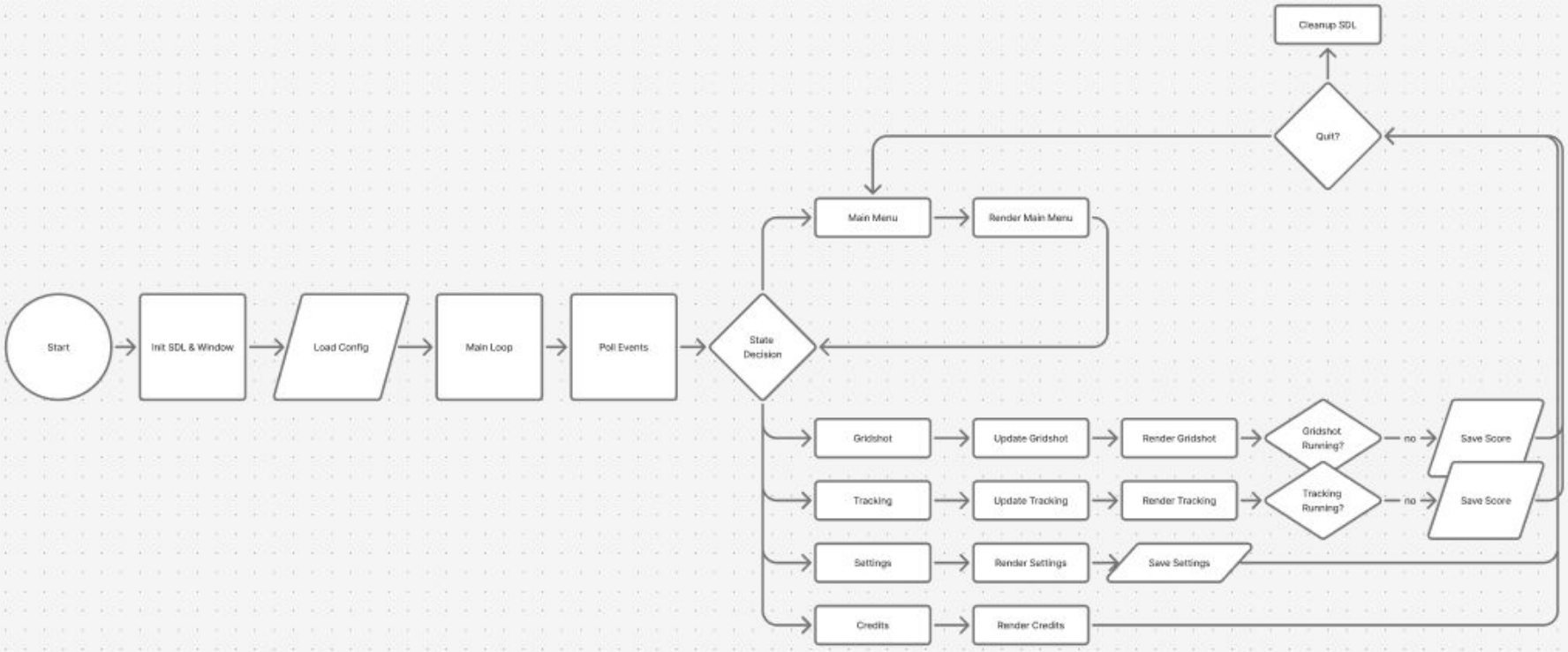
SDL_DestroyWindow(window);

SDL_Quit();

Future Improvisations

- Fully customizable background and targets.
- New modes (simulate being swung)
- Custom themes with different fonts
- Verses mode, compare scores etc.

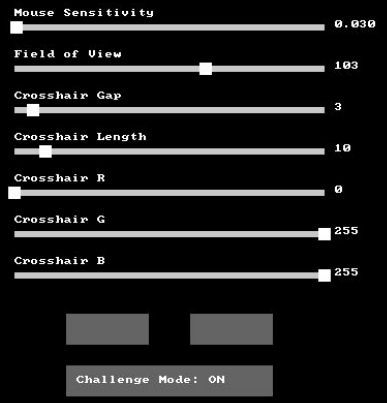
Project Concept & Procedure

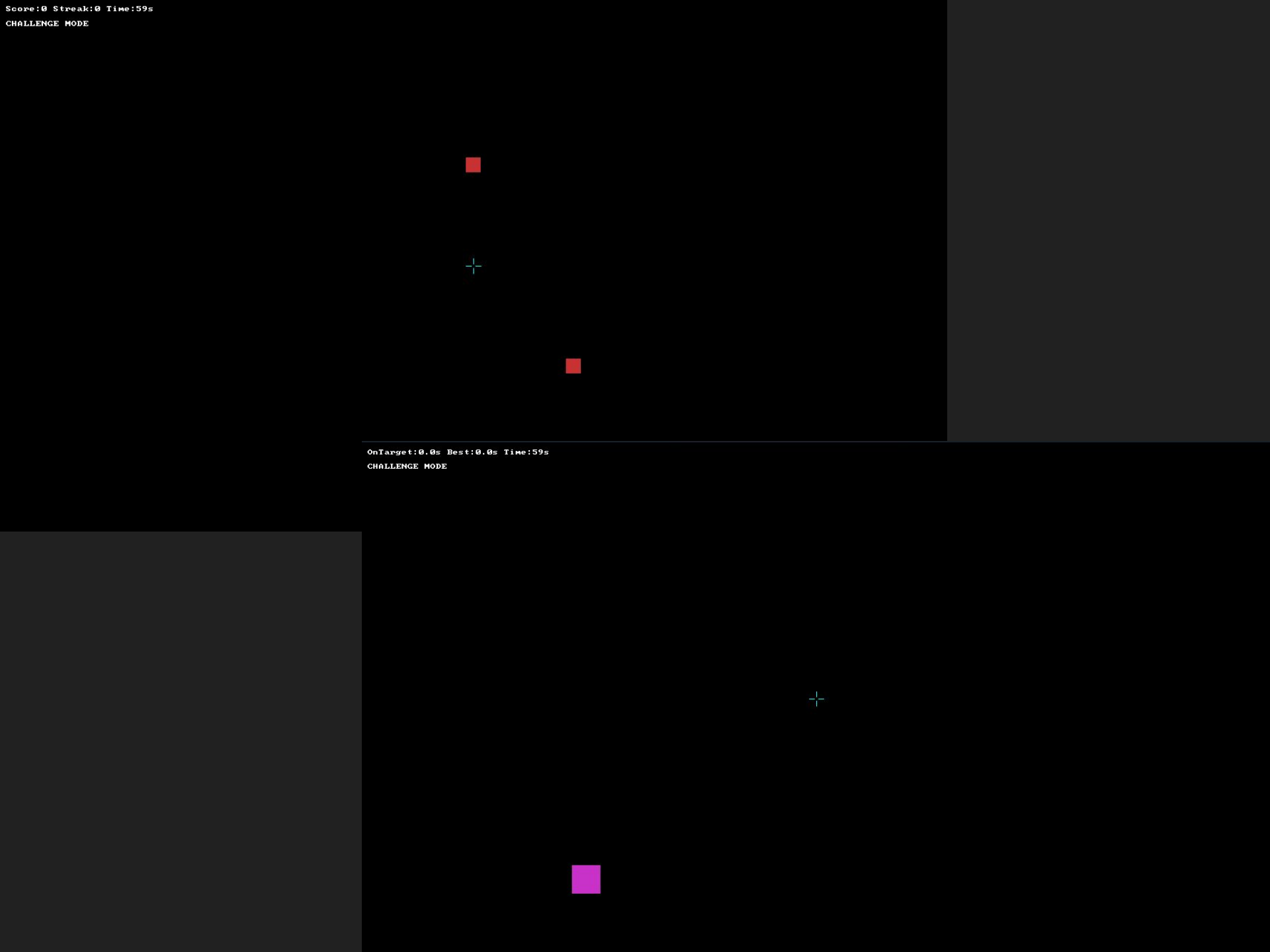


Result Discussion

- Gridshot: score, streak multiplier, best score display.
- Tracking: on-target timer, best streak display.
- Settings: Mouse Sensitivity multiplier, FOV scale, crosshair manipulation
- gameConfig.JSON to read/write/store data, challenge mode

Screenshots / Demo




Score:0 Streak:0 Time:59s
CHALLENGE MODE
OnTarget:0.0s Best:0.0s Time:59s
CHALLENGE MODE

Conclusion

- Met objectives: OOP coverage, GUI, persistence.
- Code is modular, extensible, and user-friendly.

Appendix & References

- Full source code included in project report Appendix.
- References: SDL documentation, C++ tutorial sites, StackOverflow.

examples/demo/woodeneye-008

SDL3/SDL_CreateWindowAndRenderer - SDL Wiki

SDL3/SDL_SetWindowRelativeMouseMode - SDL Wiki

SDL3/SDL_RenderDebugText - SDL Wiki

SDL3/SDL_RenderDebugText - SDL Wiki

What's a good, and easy to use, JSON c++ parser - Software Recommendations Stack Exchange

Game Engine #1 - SETUP (C/C++ Game Engine)