UNIVERSITI TEKNOLOGI MALAYSIA

PROGRAMMING TECHNIQUE II (SECJ1023)

SEMESTER 1 2025/2026

GROUP PROJECT

RHYTHM REFLEX

PHASE 4

PROJECT FINAL REPORT

NUR AQMAL IMANI BIN HASSNOR (A24CS8009)

MUHAMMAD ISYRAF BIN MD ISRAK (A25CS0278)

MUHAMMAD HAFIZ BIN MOHD HATTA (A25CS0105)

FADHIL ATHA RAMADHAN (A24CS4093)

SECTION 01

Lecturer:

DR. FAZLIATY EDORA FADZLI

DATE

16/01/2026

The work in this page has been done by: **HAFIZ** *Date:* **13/1/2026**

SECTION A: PROJECT SYNOPSIS

Introduction:

This project focuses on developing a simple reaction-based game using Object-Oriented Programming (OOP) concepts in C++. The purpose of this game is to enhance player reflexes and hand-eye coordination through precise timing. The game includes various OOP principles such as classes, objects, encapsulation, constructors, and functions to simulate game logic, scoring, and timing.

Synopsis of the Game:

The game is a circular timing reaction challenge where the player must press the SPACE key when a rotating pointer enters a highlighted zone. As the game progresses, the pointer speed increases, making the game more challenging. The player earns points for accurate timing. The round lasts for a fixed time, and the final score will reflect the player's performance.

General Concept of the Game:

1. The game features a circular play area.
2. A small circular pointer rotates along the inner edge of the circle.
3. A highlighted target zone appears on the circle and changes position and size randomly after each correct hit.
4. Scoring starts at +2 points and increases based on consecutive hits:
    - 1st hit: +2
    - 2nd hit: +4
    - 3rd and onward: +6
    - If the player misses, the score resets back to +2 on the next hit.
5. After each successful hit, a rotating encouragement message appears (e.g., "Amazing!", "Keep it up!", "WOW!").
6. A 60-second countdown timer is shown at the top right corner.
7. If the player presses at the wrong time (outside the zone), the game freezes for about 2 seconds. The pointer stops, but the timer continues.

How to Play:

1. When the game starts, the pointer will begin rotating around the inside of the circle.
2. Watch for the highlighted zone on the circle, this is the target area.
3. Press the SPACE key only when the pointer is inside the highlighted zone.

4. If you press at the correct time, you will earn points and the highlighted zone will move and change size.
5. If you press outside the zone, the game will freeze for about 2 seconds, but the timer will continue running.
6. Keep hitting accurately to build up your score multiplier and reach the maximum points per hit.
7. The round ends when the 60-second timer reaches zero, try to score as high as you can before time runs out.

Game Mission:

The objective of the game is to react quickly and accurately to achieve the highest score possible before the timer ends. The faster and the more accurate the player presses the key, the higher the score they can earn.

Scoring Method:

Action	**Points Awarded**
1st successful hit	2 points
2nd successful hit	4 points
3rd and onward successful hits	6 points per hit

The work in this page has been done by: **HAFIZ , FADHIL** *Date:* **13/1/2026**

SECTION B: SYSTEM DESIGN & IMPLEMENTATION

Flowchart:

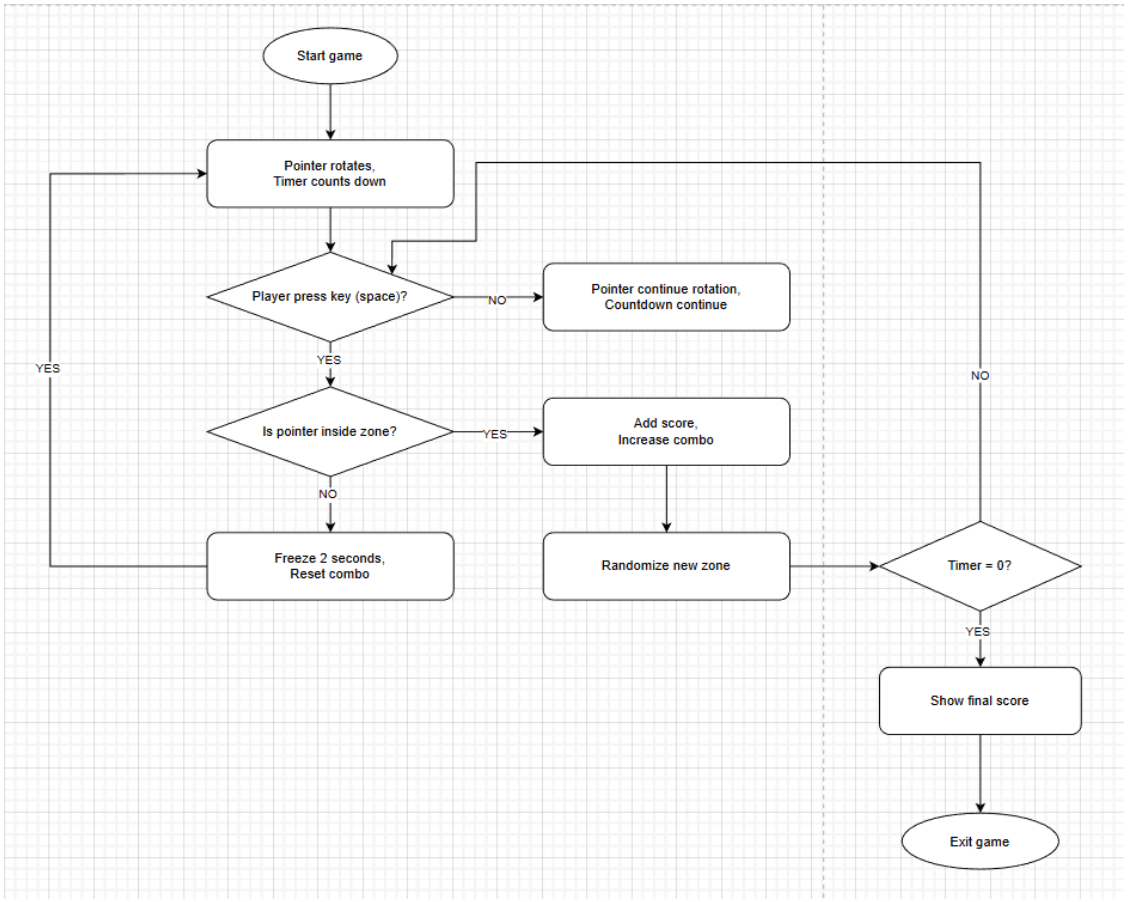
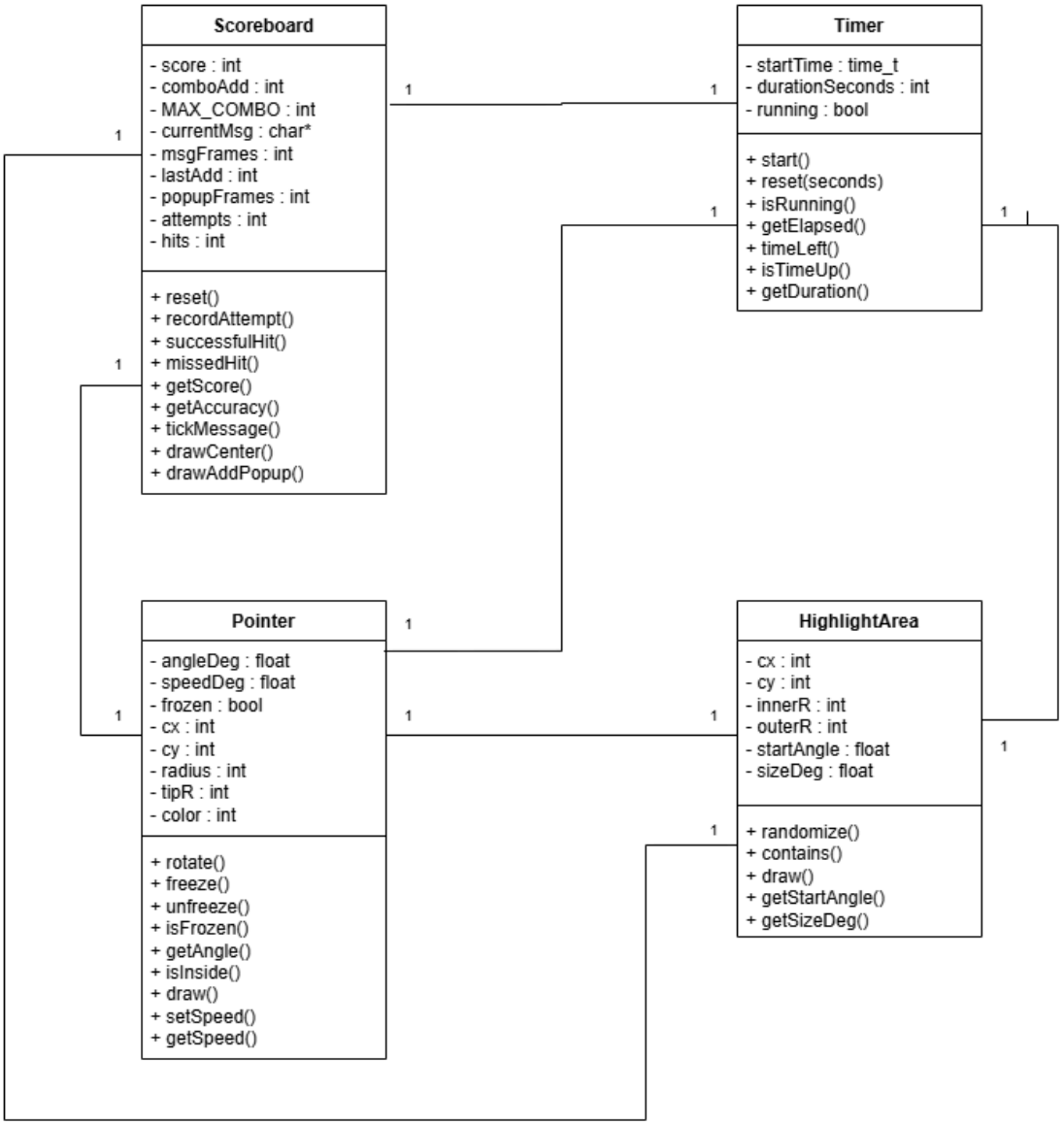


Figure 1. Flowchart of the Rhythm Reflex game

Description:The game starts by initializing all variables, after which the pointer continuously rotates while the timer counts down. During this time, the system repeatedly checks whether the player presses the space key; if no key is pressed, the pointer continues rotating and the countdown continues. When the player presses the key, the system checks whether the pointer is inside the target zone. If it is, the player is rewarded by gaining points and increasing the combo count, and a new target zone is randomly generated to continue the game; if it is not, the game applies a penalty by freezing for two seconds and resetting the combo count. After each action, the system checks whether the timer has reached zero; if the timer has not ended, the game loop continues, but once the timer reaches zero, the final score is displayed and the game exits.

CLASS DIAGRAMS:



The work in this page has been done by: **ISYRAF ISRAK, NUR AQMAL IMANI, FADHIL ATHA RAMADHA, HAFIZ** *Date:* **14/1/2026**

Code Class Main.cpp:

```cpp
#include <graphics.h>
#include <conio.h>
#include <ctime>
#include <cstdlib>
#include <cmath>
#include <windows.h>
#include <mmsystem.h>

#include "Timer.h"
#include "Scoreboard.h"
#include "HighlightArea.h"
#include "Pointer.h"
#include "UIHelper.h"

#pragma comment(lib, "winmm.lib")

static void delay_ms(int ms) {
    Sleep(ms);
}

// Draw solid donut (ring) with inner background
static void drawSolidDonut(int cx, int cy, int outerR, int innerR, int ringColor, int bgColor) {
    setfillstyle(SOLID_FILL, ringColor);
    setcolor(ringColor);
    fillellipse(cx, cy, outerR, outerR);

    setfillstyle(SOLID_FILL, bgColor);
    setcolor(bgColor);
    fillellipse(cx, cy, innerR, innerR);
}

int main() {
    srand((unsigned)time(0));

    int gd = DETECT, gm;
    initgraph(&gd, &gm, (char*)"");

    setbkcolor(WHITE);
    cleardevice();

    const int cx = getmaxx() / 2;
    const int cy = getmaxy() / 2;

    const int innerR = 90;
    const int outerR = 140;
    const int pointerPathR = (innerR + outerR) / 2;

    HighlightArea zone(cx, cy, innerR, outerR);
    Pointer pointer(cx, cy, pointerPathR, 10, WHITE);
    Scoreboard scoreboard;

    bool retry = true;
    int page = 0;
    setvisualpage(page);
    setactivepage(page);

    while (retry) {
        Timer timer(30);
        timer.start();

        scoreboard.reset();
        zone.randomize();
        pointer.randomizeSpeed(4.0f, 9.0f);

        int shakeFrames = 0;
```

```
// ========================
// START SCREEN
// ========================
page = 0;
setvisualpage(page);
setactivepage(page);

while (true) {
    setactivepage(page);
    cleardevice();

    UIHelper::drawLargeTitleImage(cx, cy - 50);
    UIHelper::drawTextCentered(cx, cy + 70, "Press ENTER to start", BLACK, 2);

    setvisualpage(page);
    page = 1 - page;

    if (kbhit()) {
        int key = getch();
        if (key == 13) break; // ENTER
    }
    delay_ms(16);
}

// ========================
// MAIN GAME LOOP
// ========================
while (!timer.isTimeUp()) {
    setactivepage(page);
    setviewport(0, 0, getmaxx(), getmaxy(), 1);
    clearviewport();

    int dx = 0, dy = 0;
    if (shakeFrames > 0) {
        dx = (rand() % 7) - 3;
        dy = (rand() % 7) - 3;
        shakeFrames--;
    }

    setviewport(dx, dy, getmaxx() + dx, getmaxy() + dy, 1);

    drawSolidDonut(cx, cy, outerR, innerR, COLOR(80, 235, 240), WHITE);
    zone.draw(COLOR(55, 125, 170));

    pointer.rotate();
    pointer.draw();

    setviewport(0, 0, getmaxx(), getmaxy(), 1);

    // UI
    UIHelper::drawSmallTitleImage(cx, cy);

    char t[32];
    UIHelper::makeTimerText(t, sizeof(t), timer.timeLeft());
    UIHelper::drawTextRight(getmaxx(), 10, t, RED, 2, 10);

    scoreboard.drawCenter(cx, cy);
    scoreboard.drawAddPopup(cx, cy, outerR);

    UIHelper::drawTextCentered(getmaxx() / 2, getmaxy() - 40,
        "Hit [SPACE] when the pointer is in the zone," BLACK, 2);

    scoreboard.tickMessage();

    setvisualpage(page);
    page = 1 - page;
```

```
            // ========================
            // INPUT HANDLING
            // ========================
            if (kbhit()) {
                int key = getch();

                if (key == ' ') {
                    while (kbhit()) getch(); // clear buffer
                    pointer.freeze();

                    bool ok = pointer.isInside(zone);
                    scoreboard.recordAttempt(ok);

                    if (ok) {
                        scoreboard.successfulHit();
                        zone.randomize();
                        PlaySound("point_up2.wav", NULL, SND_FILENAME | SND_ASYNC);
                    } else {
                        scoreboard.missedHit();
                        PlaySound("point_miss2.wav", NULL, SND_FILENAME | SND_ASYNC);
                        shakeFrames = 15;
                        delay_ms(1000);
                    }

                    pointer.unfreeze();
                } else if (key == 'q' || key == 'Q') {
                    timer.reset(0);
                }
            }

            delay_ms(16);
        }

        // ========================
        // END SCREEN
        // ========================
        page = 0;
        setvisualpage(page);
        setactivepage(page);

        bool waiting = true;
        while (waiting) {
            setactivepage(page);
            cleardevice();

            UIHelper::drawTextCentered(cx, 160, "TIME'S UP!", RED, 4);

            char finalScore[64];
            sprintf(finalScore, "Final Score: %d", scoreboard.getScore());
            UIHelper::drawTextCentered(cx, 220, finalScore, BLACK, 2);

            char acc[80];
            sprintf(acc, "Accuracy: %.1f%%", scoreboard.getAccuracy());
            UIHelper::drawTextCentered(cx, 245, acc, BLACK, 2);

            UIHelper::drawTextCentered(cx, 300, "Press R to Retry or Q to Quit", BLACK, 2);

            setvisualpage(page);
            page = 1 - page;

            if (kbhit()) {
                int key = getch();
                if (key == 'r' || key == 'R') {
                    retry = true;
                    waiting = false;
                } else if (key == 'q' || key == 'Q') {
                    retry = false;
                    waiting = false;
                }
            }
            delay_ms(16);
        }
    }

    closegraph();
    return 0;
}
```

Explanation of the code:

The **main.cpp** code implements a rhythm reflex game in C++ using the WinBGIm graphics library, where a pointer rotates around a circular "donut" and the player must press the SPACE key precisely when the pointer is inside a randomly highlighted area. The program begins by initializing the graphics system and setting up key constants like the center coordinates, inner and outer radii of the donut, and the pointer's path radius. Core game objects are instantiated, including a **HighlightArea** representing the active zone, a **Pointer** that moves around the donut, a **Scoreboard** to track score, combos, and accuracy, and a **Timer** for the gameplay duration. The program employs double buffering using **setactivepage** and **setvisualpage** to prevent flickering during animation. The game flow consists of three main phases: a start screen, where the player is prompted to press ENTER; the main gameplay loop, where the pointer rotates at a randomized speed, player input is captured via **kbhit()**/**getch()**, and actions are processed successful hits increment the score and combo, trigger a "+score" popup, play a sound effect, and randomize the zone, while misses reset the combo, play a miss sound, and briefly shake the donut for visual feedback; and an end screen, displaying the final score, accuracy percentage, and retry/quit instructions. During gameplay, the pointer and donut are drawn with optional shake offsets to add visual feedback, while UI elements such as the small title, timer, instructions, and scoreboard are drawn without shake to maintain clarity. Frame updates are regulated with a 16ms delay (~60 FPS), and all graphical drawing, collision detection, and scoring logic are tightly integrated to create a smooth, interactive, and responsive rhythm game experience.

Code Class UILogic.cpp:

```
#include "UIHelper.h"

namespace UILogic {

    void drawTitleTopLeft(const char* text) {
        settextstyle(DEFAULT_FONT, HORIZ_DIR, 2);
        setcolor(BLUE);
        outtextxy(10, 10, (char*)text);
    }

    void makeTimerText(char* out, int outSize, int timeLeft) {
        snprintf(out, outSize, "Time Left: %ds", timeLeft);
    }

    void drawTextRight(int screenW, int y, const char* text,
                       int color, int fontSize, int padding) {
        settextstyle(DEFAULT_FONT, HORIZ_DIR, fontSize);
        setcolor(color);
        int x = screenW - textwidth((char*)text) - padding;
        outtextxy(x, y, (char*)text);
    }

    void drawTextCentered(int cx, int y, const char* text,
                          int color, int fontSize) {
        settextstyle(DEFAULT_FONT, HORIZ_DIR, fontSize);
        setcolor(color);
        int x = cx - textwidth((char*)text) / 2;
        outtextxy(x, y, (char*)text);
    }

    void drawLargeTitleImage(int cx, int cy) {
        int w = 340, h = 150;
        int left = cx - w / 2;
        int top = cy - h / 2;
        int right = cx + w / 2;
        int bottom = cy + h / 2;

        readimagefile("title.bmp", left, top, right, bottom);
    }
```

```
    void drawSmallTitleImage(int cx, int cy) {
        int w = 190, h = 80;
        int left = 10;
        int top = 10;
        int right = left + w;
        int bottom = top + h;

        readimagefile("title.bmp", left, top, right, bottom);
    }

}
```

Explanation of the code:

The code defines a set of functions within the **UILogic** namespace, designed to handle various UI elements in a graphical environment using the graphics library. These functions focus on rendering text and images to the screen, providing a way to interact with the user through visual elements.

The **drawTitleTopLeft** function renders text in the top-left corner using a blue color. The **makeTimerText** formats the remaining time as a string and stores it in the out array. The **drawTextRight** function aligns text to the right edge of the screen, while **drawTextCentered** centers text horizontally at a specified position.

For image rendering, **drawLargeTitleImage** displays a large title image at the center of the screen, and **drawSmallTitleImage** shows a smaller image at the top-left corner.

Code Class UIHelper.h:

```
#ifndef UIHELPER_H
#define UIHELPER_H

#include <graphics.h>
#include <cstdio>

namespace UIHelper {

    // title at the top-left corner
    void drawTitleTopLeft(const char* text);

    // format and display the timer text
    void makeTimerText(char* out, int outSize, int timeLeft);

    // draw text aligned to the right
    void drawTextRight(int screenW, int y, const char* text,
                       int color = BLACK, int fontSize = 2, int
padding = 10);

    // draw text centered horizontally
    void drawTextCentered(int cx, int y, const char* text,
                          int color = BLACK, int fontSize = 2);

    // large title image at the center
    void drawLargeTitleImage(int cx, int cy);

    // small title image at the top-left corner
    void drawSmallTitleImage(int cx, int cy);

}

#endif
```

Explanation of the code.

The **UIHelper.h** header file defines a set of function declarations for handling UI elements like text and images in a graphical environment. These functions facilitate UI rendering by providing basic methods for placing text and images in specific areas of the screen, allowing for a dynamic and interactive interface.

Code Class Scoreboard.cpp:

```cpp
#include "Scoreboard.h"

// ---------- Utility ----------
const char* Scoreboard::randomFrom(const char* const* arr, int n) {
    return arr[rand() % n];
}

// ---------- Constructor ----------
Scoreboard::Scoreboard()
    : score(0),
      comboAdd(2),
      currentMsg(nullptr),
      msgFrames(0),
      lastAdd(0),
      popupFrames(0),
      attempts(0),
      hits(0) {}

// ---------- Reset ----------
void Scoreboard::reset() {
    score = 0;
    comboAdd = 2;
    currentMsg = nullptr;
    msgFrames = 0;
    lastAdd = 0;
    popupFrames = 0;
    attempts = 0;
    hits = 0;
}

// ---------- Accuracy ----------
void Scoreboard::recordAttempt(bool success) {
    attempts++;
    if (success) hits++;
}

int Scoreboard::getAttempts() const {
    return attempts;
}
```

```cpp
int Scoreboard::getHits() const {
    return hits;
}

float Scoreboard::getAccuracy() const {
    if (attempts <= 0) return 0.0f;
    return (hits * 100.0f) / attempts;
}

// ---------- Gameplay ----------
void Scoreboard::successfulHit() {
    lastAdd = comboAdd;
    popupFrames = 45;

    score += comboAdd;

    if (comboAdd < MAX_COMBO)
        comboAdd += 2;

    static const char* const hitMsgs[] = {
        "Amazing!", "Keep it up!", "WOW!", "Nice!", "Great!"
    };

    currentMsg = randomFrom(hitMsgs, 5);
    msgFrames = 60;
}

void Scoreboard::missedHit() {
    comboAdd = 2;
    lastAdd = 0;
    popupFrames = 0;

    static const char* const missMsgs[] = {
        "Miss!", "Oops!", "Try again!", "Almost!"
    };

    currentMsg = randomFrom(missMsgs, 4);
    msgFrames = 60;
}
```

```
80
81  // ---------- Getters ----------
82  int Scoreboard::getScore() const {
83      return score;
84  }
85
86  int Scoreboard::getComboAdd() const {
87      return comboAdd;
88  }
89
90  // ---------- Frame Update ----------
91  void Scoreboard::tickMessage() {
92      if (msgFrames > 0) --msgFrames;
93      if (msgFrames == 0) currentMsg = nullptr;
94
95      if (popupFrames > 0) --popupFrames;
96  }
```

```
98  // ---------- Drawing ----------
99  void Scoreboard::drawCenter(int cx, int cy) const {
100     setcolor(BLACK);
101     settextstyle(DEFAULT_FONT, HORIZ_DIR, 6);
102
103     char s[16];
104     sprintf(s, "%d", score);
105
106     int w = textwidth(s);
107     int h = textheight(s);
108
109     outtextxy(cx - w / 2, cy - h / 2 - 10, s);
110
111     if (currentMsg != nullptr) {
112         settextstyle(DEFAULT_FONT, HORIZ_DIR, 2);
113         int mw = textwidth((char*)currentMsg);
114         outtextxy(cx - mw / 2, cy + 40, (char*)currentMsg);
115     }
116 }
117
118 void Scoreboard::drawAddPopup(int cx, int cy, int outerR) const {
119     if (popupFrames <= 0 || lastAdd <= 0) return;
120
121     char p[8];
122     sprintf(p, "+%d", lastAdd);
123
124     int x = cx + outerR + 25;
125     int y = cy - 20;
126
127     int pad = 6;
128     int w = textwidth(p);
129     int h = textheight(p);
130
131     setfillstyle(SOLID_FILL, WHITE);
132     setcolor(BLACK);
133     bar(x - pad, y - pad, x + w + pad, y + h + pad);
134     rectangle(x - pad, y - pad, x + w + pad, y + h + pad);
135
136     int popupColor = BLACK;
137     if (lastAdd == 2) popupColor = COLOR(60, 60, 60);
138     else if (lastAdd == 4) popupColor = COLOR(255, 180, 0);
139     else if (lastAdd == 6) popupColor = COLOR(0, 170, 60);
140
141     settextstyle(DEFAULT_FONT, HORIZ_DIR, 2);
142     setcolor(popupColor);
143     outtextxy(x, y, p);
144 }
145
```

Explanation of the code:

Scoreboard.cpp provides the implementation of every method declared in Scoreboard.h, defining how each operation behaves during gameplay. The utility method randomFrom() is implemented to randomly select feedback messages from predefined arrays. The constructor initializes all scoreboard variables to their default starting values, while reset() restores these values when a new game begins. The recordAttempt() method increments the attempt counter and updates hit statistics, and getAccuracy() computes the player's accuracy as a percentage while safely handling division by zero. Gameplay logic is implemented in successfulHit(), which increases the score using the current combo value, advances the combo bonus up to a maximum limit, triggers a score popup, and selects a positive feedback message, while missedHit() resets the combo and displays an appropriate miss message. The tickMessage() method updates frame counters to control the lifespan of messages and popups. Finally, rendering methods drawCenter() and drawAddPopup() implement the drawing logic for displaying the score, feedback messages, and colored score popups using graphics functions, ensuring visual feedback is synchronized with the game loop.

Code Class Scoreboard.h:

```
7
8   class Scoreboard {
9   private:
10      // Score
11      int score;
12
13      // Combo system (+2 ? +4 ? +6)
14      int comboAdd;
15      static const int MAX_COMBO = 6;
16
17      // Encouragement message
18      const char* currentMsg;
19      int msgFrames;
20
21      // Popup for "+2/+4/+6"
22      int lastAdd;
23      int popupFrames;
24
25      // Accuracy tracking
26      int attempts;
27      int hits;
28
29      // Utility to pick random encouragement strings
30      static const char* randomFrom(const char* const* arr, int n);
31
32  public:
33      // Constructor
34      Scoreboard();
35
36      // Reset everything
37      void reset();
38
39      // Accuracy tracking
40      void recordAttempt(bool success);
41      int getAttempts() const;
42      int getHits() const;
43      float getAccuracy() const;
44
45      // Gameplay actions
46      void successfulHit();
47      void missedHit();
48
49      // Getters
50      int getScore() const;
51      int getComboAdd() const;
52
53      // Frame update
54      void tickMessage();
55
56      // Drawing
57      void drawCenter(int cx, int cy) const;
58      void drawAddPopup(int cx, int cy, int outerR) const;
59  };
60
61  #endif
```

Explanation of the code:

Scoreboard.h declares the Scoreboard class and defines all the methods that can be accessed by other parts of the game. It specifies the class's responsibility for managing score calculation, combo progression, accuracy tracking, message timing, and score rendering. The header file declares methods such as reset() to reinitialize all scoring data, recordAttempt() to log player actions for accuracy calculation, and getter methods like getScore(), getComboAdd(), getAttempts(), getHits(), and getAccuracy() to provide controlled read-only access to internal data. It also declares gameplay-related methods including successfulHit() and missedHit(), which update the score and combo state based on player performance. In addition, the header file declares frame-based and rendering methods such as tickMessage() for updating message timers each frame, drawCenter() for displaying the score and feedback text at the center of the screen, and drawAddPopup() for rendering animated score increment popups. By listing all method declarations without implementation details, Scoreboard.h clearly defines what operations the Scoreboard class can perform.

Code Class HighlightArea.h:

```
#ifndef HIGHLIGHTAREA_H
#define HIGHLIGHTAREA_H

#include <graphics.h>

class HighlightArea {
private:
    int cx, cy;
    int innerR, outerR;
    float startAngle;
    float sizeDeg; // angular width

public:
    HighlightArea(int x, int y, int innerRadius, int outerRadius);

    void randomize();
    bool contains(float angleDeg) const;
    void draw(int color) const;

    float getStartAngle() const;
    float getSizeDeg() const;
};

#endif
```

Code Class HighlightArea.cpp:

```cpp
#include "HighlightArea.h"
#include <cstdlib>   // rand()

HighlightArea::HighlightArea(int x, int y, int innerRadius, int outerRadius)
    : cx(x), cy(y), innerR(innerRadius), outerR(outerRadius),
      startAngle(0), sizeDeg(40) {
    randomize();
}

void HighlightArea::randomize() {
    startAngle = (float)(rand() % 360);
    // 15° to 70°
    sizeDeg = (float)(15 + rand() % 56);
}

bool HighlightArea::contains(float angleDeg) const {
    float endAngle = startAngle + sizeDeg;

    if (endAngle < 360.0f) {
        return angleDeg >= startAngle && angleDeg <= endAngle;
    }
    // wrap-around case
    return angleDeg >= startAngle || angleDeg <= (endAngle - 360.0f);
}

void HighlightArea::draw(int color) const {
    setcolor(color);
    // draw thick arc by sweeping radii across the donut thickness
    for (int r = innerR; r <= outerR; ++r) {
        arc(cx, cy, (int)startAngle, (int)(startAngle + sizeDeg), r);
    }
}

float HighlightArea::getStartAngle() const {
    return startAngle;
}

float HighlightArea::getSizeDeg() const {
    return sizeDeg;
}
```

Explanation of the code:

The **HighlightArea** class represents the target zone that the player must align the pointer with during gameplay. With **HighlightArea.h** as the header file for the class declaration and **HiglightArea.cpp** as the implementation file for the class definition. This class is responsible for determining hit detection, randomizing target positions, and rendering the highlight area visually.

The class stores positional attributes such as the center coordinates (**cx**, **cy**), **radius**, and **thickness** of the highlight area. These attributes define the circular ring in which the active zone is drawn. Additional attributes such as **startAngle** and **size** are used to represent the angular position and width of the highlighted region, allowing the game to calculate whether the rotating pointer is inside the target area.

The **randomize()** method generates a new random starting angle and angular size for the highlight area. This ensures that the target location changes after each successful hit. The **contains()** method checks whether a given angle lies within the active region of the highlight area. It also correctly handles angle wrapping when the highlighted region crosses the 360-degree boundary.

Overall, the HighlightArea class demonstrates the principle of encapsulation by keeping its internal data private and exposing controlled access through public methods. It also participates in an association relationship with the Pointer class, as the pointer queries the highlight area to determine hit accuracy without owning or controlling it.

Code Class Pointer.h:

```
#ifndef POINTER_H
#define POINTER_H

#include <graphics.h>
#include "HighlightArea.h"

class Pointer {
private:
    float angleDeg;
    float speedDeg;
    bool frozen;
    int cx, cy;
    int radius; // path radius
    int tipR;   // pointer circle size
    int color;

public:
    Pointer(int centerX, int centerY, int pathRadius, int pointerRadius, int c);

    float randomizeSpeed(float minSpeed, float maxSpeed);
    void rotate();

    void freeze();
    void unfreeze();
    bool isFrozen() const;

    float getAngle() const;
    bool isInside(const HighlightArea& zone) const;

    void draw() const;

    void setSpeed(float s);
    float getSpeed() const;
};

#endif
```

Code Class Pointer.cpp:

```cpp
#include "Pointer.h"
#include <cmath>
#include <cstdlib>

Pointer::Pointer(int centerX, int centerY, int pathRadius, int pointerRadius, int c)
    : angleDeg(0.0f),
      speedDeg(8.0f),
      frozen(false),
      cx(centerX),
      cy(centerY),
      radius(pathRadius),
      tipR(pointerRadius),
      color(c) {
}

float Pointer::randomizeSpeed(float minSpeed, float maxSpeed) {
    speedDeg = minSpeed +
               (float)rand() / (float)RAND_MAX * (maxSpeed - minSpeed);
    return speedDeg;
}

void Pointer::rotate() {
    if (frozen) return;

    angleDeg += speedDeg;
    if (angleDeg >= 360.0f) angleDeg -= 360.0f;
    if (angleDeg < 0.0f) angleDeg += 360.0f;
}

void Pointer::freeze() {
    frozen = true;
}

void Pointer::unfreeze() {
    frozen = false;
}

bool Pointer::isFrozen() const {
    return frozen;
}

float Pointer::getAngle() const {
    return angleDeg;
}
```

```
45
46  bool Pointer::isInside(const HighlightArea& zone) const {
47      return zone.contains(angleDeg);
48  }
49
50  void Pointer::draw() const {
51      float rad = angleDeg * 3.14159265f / 180.0f;
52      int px = cx + (int)(radius * cos(rad));
53      int py = cy - (int)(radius * sin(rad));
54
55      setcolor(color);
56      setfillstyle(SOLID_FILL, color);
57      fillellipse(px, py, tipR, tipR);
58  }
59
60  void Pointer::setSpeed(float s) {
61      speedDeg = s;
62  }
63
64  float Pointer::getSpeed() const {
65      return speedDeg;
66  }
```

Explanation of the code:

The **Pointer** class represents the rotating indicator that the player controls during gameplay. It is responsible for managing the pointer's movement, handling hit detection, and rendering the pointer visually on the screen. This class models the core interactive element of the game.

The class maintains several private attributes, including the current rotation angle, rotation speed, and a frozen state. The angle attribute tracks the pointer's current position around the circle, while the speed determines how fast the pointer rotates. The frozen flag is used to temporarily stop movement after a missed hit, enforcing gameplay penalties. Additional attributes such as the center coordinates, pointer length, and color define how the pointer is drawn graphically.

The **rotate()** method updates the pointer's angle based on its speed while ensuring that the value remains within the valid 0–360 degree range. This method only performs rotation when the pointer is not frozen, allowing the game to control player input timing. The **freeze()** and **unfreeze()** methods manage the frozen state, enabling temporary pauses in pointer movement after the player misses.

The **isInside()** method checks whether the pointer's current angle lies within the current highlight area. This method takes a **HighlightArea** object as a parameter and delegates the hit detection logic to the highlight area itself. This design demonstrates an association relationship between the two classes, as the pointer interacts with the highlight area without owning it.

The **draw()** method handles graphical rendering of the pointer. Instead of drawing a full line from the center, the pointer is displayed as a small visual marker positioned at the correct angle around the circle. Trigonometric calculations are used to convert the angle into screen coordinates, ensuring smooth and accurate rotation.

Code Class Timer.h

```
#ifndef TIMER_H
#define TIMER_H

#include <ctime>

class Timer {
private:
    time_t startTime;
    int durationSeconds;
    bool running;

public:
    Timer(int seconds = 60);

    void start();
    void reset(int seconds);

    bool isRunning() const;

    int getElapsed() const;
    int timeLeft() const;
    bool isTimeUp() const;

    int getDuration() const;
};

#endif
```

Explanation of the code:

Timer.h defines the interface of the Timer class, which is responsible for tracking time in seconds using the system clock. It declares the private data members **startTime (the time when the timer starts**), **durationSeconds (how long the timer should run)**, and

running (whether the timer is active). The public section declares the constructor and all the functions that control and query the timer, such as starting or resetting it, checking if it is running, getting elapsed time, calculating remaining time, and determining whether time is up. By placing only declarations here, the header allows other files to use the Timer class without knowing how it is implemented, promoting clean structure and separation of concerns.

Code Class Timer.cpp:

```cpp
#include "Timer.h"

Timer::Timer(int seconds)
    : startTime(0), durationSeconds(seconds), running(false) {}

void Timer::start() {
    startTime = time(NULL);
    running = true;
}

void Timer::reset(int seconds) {
    durationSeconds = seconds;
    start();
}

bool Timer::isRunning() const {
    return running;
}

int Timer::getElapsed() const {
    if (!running) return 0;
    return (int)difftime(time(NULL), startTime);
}

int Timer::timeLeft() const {
    if (!running) return durationSeconds;
    int remaining = durationSeconds - getElapsed();
    return (remaining > 0) ? remaining : 0;
}

bool Timer::isTimeUp() const {
    return timeLeft() <= 0;
}

int Timer::getDuration() const {
    return durationSeconds;
}
```

Explanation of the code:

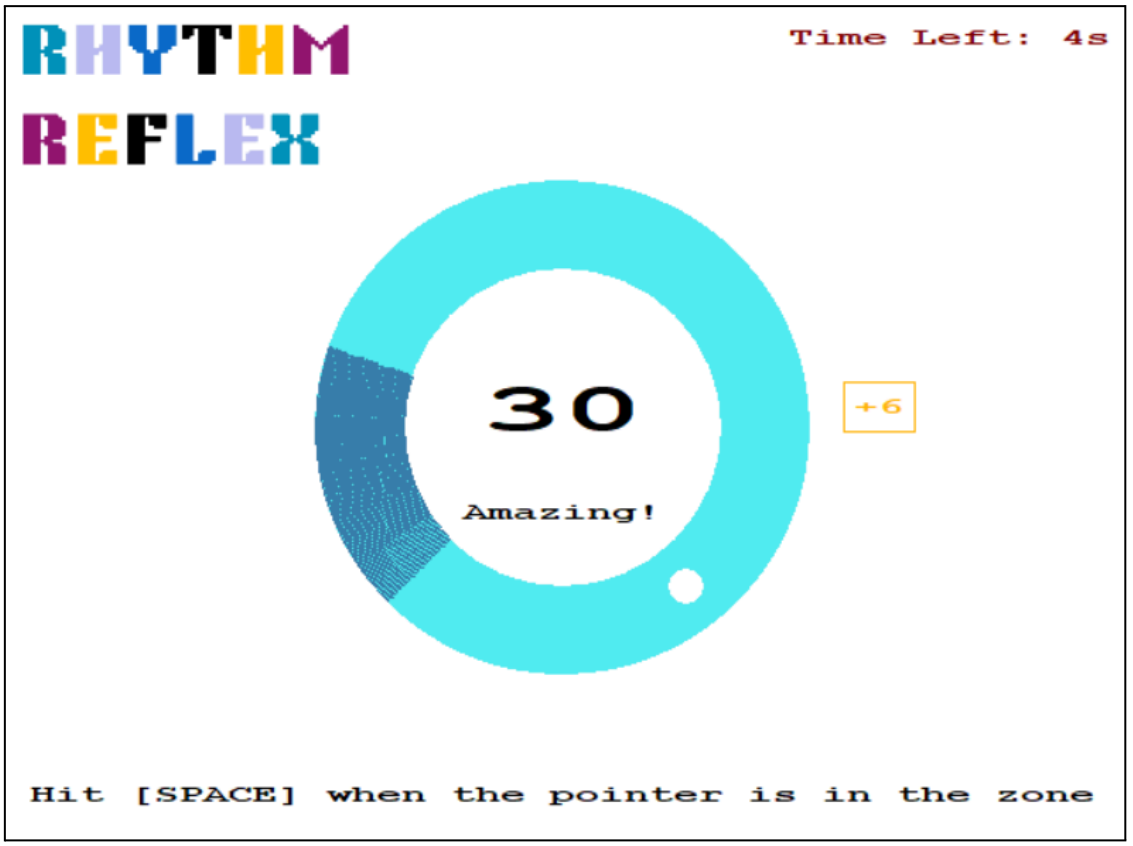
Timer.cpp contains the implementation of all functions declared in **Timer.h**. The constructor initializes the timer duration and sets the timer to a non-running state. The **start()** function records the current system time using **time(NULL)** and marks the timer as running, while **reset()** updates the duration and restarts the timer. The timing-related functions calculate elapsed and remaining time by comparing the current time with **startTime** using **difftime**, ensuring the timer continues accurately even if the game loop pauses. Functions like is**TimeUp()** and **timeLeft()** provide simple checks for game logic, making this file responsible for how the timer actually works internally.

SECTION C: RESULTS AND OUTPUT

Main Screen:

The main screen shows the main game circle, the pointer inside it, and the game title "Rhythm Reflex." The pointer does not rotate yet. A message saying "Press SPACE to Start" is displayed to tell the player how to begin. This screen is mainly used to introduce the game and wait for the player's input before the timer and scoring system start.

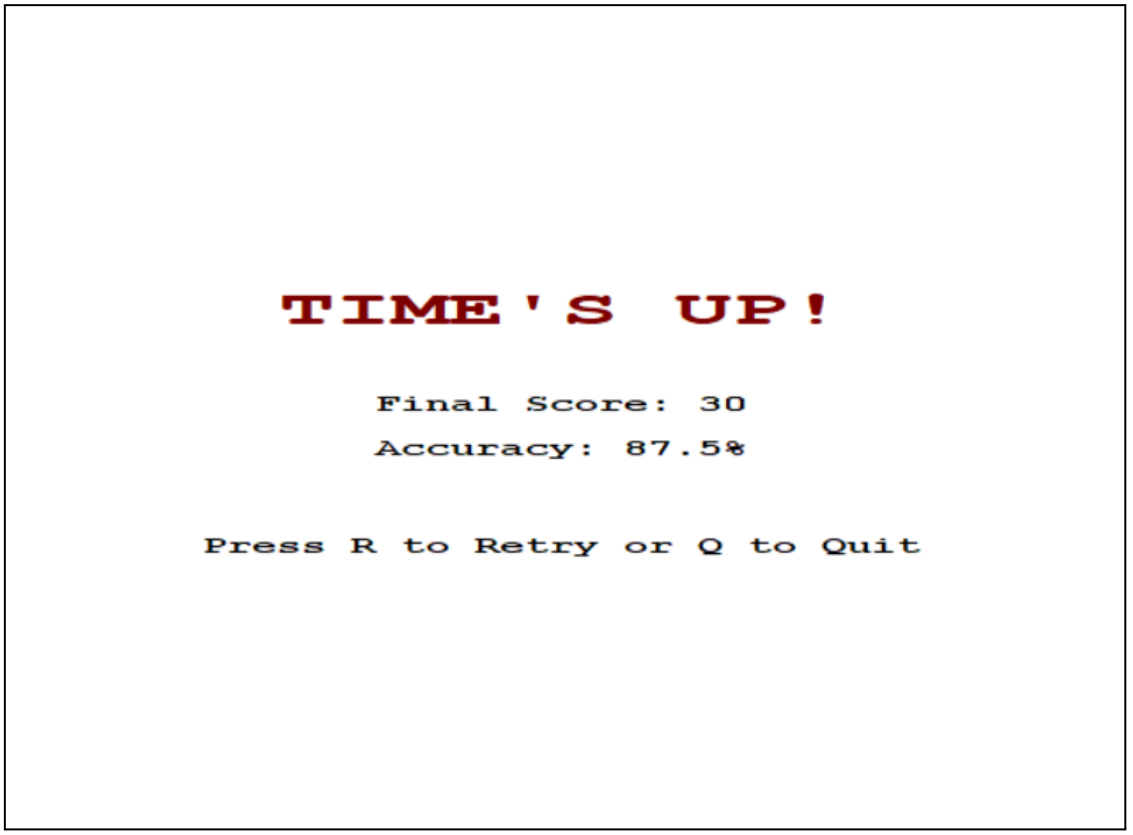
Game Screen:



During gameplay, the pointer rotates inside the circle while a highlighted zone appears randomly. When the player hits the highlighted area at the correct time, the score increases based on the combo (+2, +4, +6), and an encouragement word will appear below the score. This screen shows the active movement of the pointer, the score, the combo progress, and the countdown timer in real time.

If the player presses the key outside the highlighted zone, a wrong hit occurs. The pointer will freeze for about two seconds, but the timer continues running. The highlight disappears, and the combo resets. This screen represents the penalty moment in the game, showing that the pointer has stopped while the countdown continues.

Game End Screen:



When the countdown timer reaches zero, the game switches to the end screen. The pointer stops, and the player's final score is shown in the center. A gratitude message and game hint like "Press SPACE to Play Again" will appear, allowing the player to restart the game. This screen marks the end of a session and lets the player begin a new round easily.

SECTION D: LOG BOOK

Date of Meeting	Time	Attendees	Discussion / Purpose of Meeting / Tasks Done
28 October 2025	10:00 pm	All members	Discussion: Planned overall project structure and assigned responsibilities for Phase 1. Purpose: To organize the team and identify main game components. Tasks Done: Assigned roles for UI design, game logic, audio integration, and documentation. Identified main game objects: Pointer, HighlightArea, Scoreboard, Timer, and UI elements. Planned class structure and interactions.
2 December 2025	9:00 pm	All members	Discussion: Started implementing core classes (Pointer and HighlightArea). Purpose: To develop the movement and hit detection logic. Tasks Done: Implemented pointer rotation and speed randomization. Implemented highlight zone generation, angle detection, and drawing functions. Verified interactions between pointer and highlight zone.
10 December 2025	10:00 pm	All members	Discussion: Focused on scoring and combo mechanics. Purpose: To implement the game's scoring system and feedback for player actions. Tasks Done: Implemented Scoreboard class with combo tracking, pop-up feedback, and accuracy calculations. Integrated successful hit and miss handling, including visual feedback and sound triggers.
12 December 2025	8:30 pm	All members	Discussion: UI design and helper functions. Purpose: To improve game interface and text rendering. Tasks Done: Implemented UIHelper namespace with functions for drawing titles, timer, and centered/right-aligned text. Added large and small title images, timer formatting, and message display. Tested drawing functions on screen.

27 December 2025	9:00 pm	All members	Discussion: Integrated all modules and tested gameplay loop. Purpose: To ensure smooth interaction between pointer, highlight zones, scoreboard, timer, and UI. Tasks Done: Integrated Pointer, HighlightArea, Scoreboard, Timer, and UI elements in main.cpp. Implemented main game loop with input handling (SPACE for hit, Q to quit), double buffering for smooth animation, and frame delay for 60 FPS. Tested zone detection, score updates, and UI responsiveness.
9 January 2026	10:00 pm	All members	Discussion: Sound effects and visual effects. Purpose: To enhance player experience with audio-visual feedback. Tasks Done: Added sound effects for successful and missed hits (PlaySound). Implemented donut shake effect for missed hits. Verified timing, animations, and proper frame updates with shake offsets.
12 January 2026	7:00 pm	All members	Discussion: Final debugging and performance testing. Purpose: To finalize the project and prepare for submission. Tasks Done: Fixed minor bugs in pointer rotation, highlight zone detection, and scoring logic. Optimized game performance. Tested on multiple devices and screen resolutions. Prepared final submission with all source files (.h and .cpp) and required assets (images, sounds).