**Let’s break this Win32 “Flappy Bird” clone down into logical sections—what each part does, why it’s there, and what you’d say if someone quizzed you on it.**

**1. Headers & Namespaces**

**#include <windows.h> // Win32 API: window management, GDI drawing, messages**

**#include <vector> // std::vector for dynamic pillar list**

**#include <cstdlib> // rand(), srand()**

**#include <ctime> // time() for seeding rand()**

**#include <string> // std::string**

**#include <fstream> // file I/O for high score**

**#include <memory> // smart pointers (included but not actually used)**

* **windows.h: core Win32 functions/types (HWND, HDC, RECT, messages, GDI).**
* **The C++ STL headers handle containers, randomness, strings, and file streams.**
* **Note: <memory> is included but no unique\_ptr/shared\_ptr is actually employed here.**

**2. Game Constants**

**const int SCREEN\_WIDTH = 800;**

**const int SCREEN\_HEIGHT = 600;**

**const int PILLAR\_GAP = 200;**

**const int PILLAR\_WIDTH = 80;**

**const int BIRD\_SIZE = 30;**

**const int GROUND\_HEIGHT = 50;**

**These define your playfield dimensions and object sizes. Anyone asking you can point to how they control layout without magic numbers sprinkled through the code.**

**3. Bird Structure**

**struct Bird {**

**int x, y;**

**float velocity;**

**const float gravity = 0.4f;**

**const float jumpForce = -8.0f;**

**Bird() : x(100), y(SCREEN\_HEIGHT/2), velocity(0) {}**

**Bird(const Bird& other) : /\* copy ctor initializes all fields \*/ {}**

**Bird& operator=(const Bird& other) { /\* assignment copies mutable fields \*/ }**

**void update() { velocity += gravity; y += velocity; }**

**void jump() { velocity = jumpForce; }**

**RECT getRect();**

**};**

* **Position & Motion: x is fixed (100px); y and velocity model vertical movement.**
* **Gravity vs. Jump: each update() applies gravity, shifting y; jump() blasts you upward.**
* **Copy semantics: they've provided a copy ctor and an assignment operator so you can safely do g\_bird = Bird(); in InitGame().**
* **Collision box: getRect() returns a RECT centered on (x,y) of size BIRD\_SIZE.**

**4. Pillar Structure**

**struct Pillar {**

**int x, gapY;**

**bool passed;**

**Pillar(int x, int gapY) : x(x), gapY(gapY), passed(false) {}**

**void update() { x -= 3; }**

**bool isOffScreen() const { return x < -PILLAR\_WIDTH; }**

**RECT getTopRect() const;**

**RECT getBottomRect() const;**

**};**

* **X-coordinate moves left by 3px each frame.**
* **gapY: vertical center of the gap—randomized on spawn.**
* **passed: tracks if the bird has flown beyond this pillar (to increment score only once).**
* **Geometry: getTopRect()/getBottomRect() carve out the two solid blocks around the gap.**

**5. Global Game State**

**static HWND g\_hwnd = nullptr;**

**static bool g\_gameRunning= true;**

**static int g\_score = 0;**

**static int g\_highScore = 0;**

**static Bird g\_bird;**

**static vector<Pillar> g\_pillars;**

**static int g\_pillarTimer= 0;**

**const int PILLAR\_FREQUENCY = 120;**

* **Window handle (g\_hwnd) and run-flag (g\_gameRunning).**
* **Scores stored in RAM and persisted via highscore.txt.**
* **Bird & pillars are the main game objects.**
* **g\_pillarTimer drives spawn frequency (~every 120 frames).**

**6. Resource Management: GDIResources**

**class GDIResources {**

**HGDIOBJ skyBrush\_, groundBrush\_, pillarBrush\_, birdBrush\_, font\_;**

**public:**

**void initialize(HDC hdc);**

**~GDIResources();**

**HBRUSH getSkyBrush() const;**

**HBRUSH getGroundBrush() const;**

**/\* …etc… \*/**

**};**

* **Encapsulates creation and cleanup of GDI objects: brushes (for color fills) and a font.**
* **Why: avoids leaking GDI handles each frame.**
* **What you’d mention: calling DeleteObject(...) in the destructor ensures cleanup.**

**7. Game Initialization & High Score I/O**

**void InitGame() {**

**g\_bird = Bird(); // reset bird**

**g\_pillars.clear(); // wipe old pillars**

**g\_score = g\_pillarTimer = 0;**

**// load high score**

**ifstream file("highscore.txt");**

**if(file.is\_open()) { file >> g\_highScore; }**

**}**

* **Reset everything to starting state.**
* **High score is read once—later overwritten if beaten.**

**8. Spawning & Updating Logic**

**void SpawnPillar() { /\* random gapY, push\_back new Pillar \*/ }**

**void UpdateGame() {**

**g\_bird.update();**

**if (++g\_pillarTimer >= PILLAR\_FREQUENCY) SpawnPillar(), g\_pillarTimer=0;**

**for(auto& p : g\_pillars) {**

**p.update();**

**if(!p.passed && p.x + PILLAR\_WIDTH < g\_bird.x) {**

**p.passed = true; ++g\_score; /\* update high score file if needed \*/**

**}**

**}**

**// remove off-screen pillars**

**if(!g\_pillars.empty() && g\_pillars.front().isOffScreen())**

**g\_pillars.erase(g\_pillars.begin());**

**// collision checks:**

**RECT b = g\_bird.getRect();**

**// ground or ceiling → restart**

**if(b.bottom > SCREEN\_HEIGHT - GROUND\_HEIGHT || b.top < 0) { InitGame(); return; }**

**// pillar collisions → restart**

**for(auto& p : g\_pillars) {**

**if(b.right > p.getTopRect().left && b.left < p.getTopRect().right) {**

**if(b.top < p.getTopRect().bottom || b.bottom > p.getBottomRect().top)**

**{ InitGame(); return; }**

**}**

**}**

**}**

* **Score increments when the bird fully passes a pillar.**
* **Collision: rectangle overlap tests against ground, ceiling, and pillars, all resetting the game on hit.**

**9. Rendering Frame by Frame**

**void DrawGame(HDC hdc) {**

**GDIResources res;**

**res.initialize(hdc);**

**// 1) sky background**

**FillRect(hdc, &skyRect, res.getSkyBrush());**

**// 2) ground**

**FillRect(hdc, &groundRect, res.getGroundBrush());**

**// 3) pillars**

**for(auto& p : g\_pillars) {**

**FillRect(hdc, &p.getTopRect(), res.getPillarBrush());**

**FillRect(hdc, &p.getBottomRect(), res.getPillarBrush());**

**}**

**// 4) bird**

**FillRect(hdc, &g\_bird.getRect(), res.getBirdBrush());**

**// 5) text**

**SelectObject(hdc, res.getFont());**

**TextOut(hdc, 10, 10, scoreText.c\_str(), scoreText.size());**

**TextOut(hdc, 10, 50, highScoreText.c\_str(), highScoreText.size());**

**}**

* **A fresh GDIResources object each frame ensures brushes/fonts are always valid and cleaned up at end of scope.**
* **Order matters: sky → ground → pillars → bird → UI text.**

**10. Win32 Boilerplate & Message Loop**

**LRESULT CALLBACK WindowProc(...) {**

**switch(uMsg) {**

**case WM\_DESTROY: g\_gameRunning = false; PostQuitMessage(0); return 0;**

**case WM\_KEYDOWN: if(wParam==VK\_SPACE) g\_bird.jump(); return 0;**

**case WM\_LBUTTONDOWN: g\_bird.jump(); return 0;**

**}**

**return DefWindowProc(...);**

**}**

**int WINAPI WinMain(...) {**

**// 1) RegisterClass + CreateWindowEx**

**// 2) ShowWindow**

**InitGame(); srand(time(NULL));**

**// 3) Game loop:**

**while(g\_gameRunning) {**

**// a) Process all pending messages**

**// b) UpdateGame()**

**// c) DrawGame()**

**// d) Sleep(16) → ~60 FPS**

**}**

**return 0;**

**}**

* **WindowProc handles quit, space-bar, or mouse clicks to make the bird jump.**
* **Main loop: collects Windows messages, steps simulation, renders, and throttles to ~60 Hz.**

**Potential Quiz Questions & Pitfalls**

1. **Why GDIResources per-frame?**
   * **Ensures brushes/fonts match current HDC and get cleaned up immediately, avoids leaks.**
2. **Why copy ctor & assignment in Bird?**
   * **So g\_bird = Bird(); safely resets even though Bird has const members.**
3. **Collision logic nuance:**
   * **The pillar‐collision test checks horizontal overlap first, then vertical; resetting on either top‐or‐bottom collision.**
4. **File I/O for high score:**
   * **Opens/writes highscore.txt only when beaten, preventing constant disk churn.**
5. **Frame rate control:**
   * **Sleep(16) approximates 60 FPS but isn’t precise—Windows may wake later. In a production game you’d use a high‐precision timer.**

**With this roadmap, you can confidently explain both “how” it works and “why” it’s structured this way if any interviewer or peer asks you to walk through it.**

**Simple Pseudocode**

BEGIN

Initialize game state

WHILE game is running DO

Handle input (space‐bar or click → make bird jump)

Update bird position (apply gravity, move by velocity)

Spawn new pillars at regular intervals

Move pillars left; remove off‐screen ones

Check for collisions (ground, ceiling, pillars)

IF collision THEN

Reset game state (score, bird, pillars)

Draw frame:

Clear screen

Draw background, ground

Draw all pillars

Draw bird

Draw score and high score

Wait ~16 ms (to cap at ~60 FPS)

END WHILE

END

**Detailed Pseudocode**

// --- Game Constants & Globals ---

CONST SCREEN\_WIDTH = 800

CONST SCREEN\_HEIGHT = 600

CONST PILLAR\_GAP = 200

CONST PILLAR\_WIDTH = 80

CONST BIRD\_SIZE = 30

CONST GROUND\_HEIGHT = 50

VAR score, highScore, pillarTimer, gameRunning

VAR bird(x, y, velocity)

LIST pillars

// --- Initialization ---

FUNCTION InitGame()

bird ← (x=100, y=SCREEN\_HEIGHT/2, velocity=0)

CLEAR pillars

score ← 0

pillarTimer ← 0

highScore ← READ from "highscore.txt" (if exists)

END FUNCTION

// --- Main Loop ---

InitGame()

gameRunning ← TRUE

WHILE gameRunning DO

// 1) Process Input

FOR each pending Windows message DO

IF message == WM\_KEYDOWN and key == SPACE OR message == MOUSE\_CLICK THEN

bird.velocity ← jumpForce

IF message == WM\_DESTROY THEN

gameRunning ← FALSE

END IF

END FOR

// 2) Update Simulation

// 2a) Bird physics

bird.velocity ← bird.velocity + gravity

bird.y ← bird.y + bird.velocity

// 2b) Pillar spawning

pillarTimer ← pillarTimer + 1

IF pillarTimer ≥ PILLAR\_FREQUENCY THEN

gapY ← random between [minGap, maxGap]

ADD new Pillar(x=SCREEN\_WIDTH, gapY) to pillars

pillarTimer ← 0

END IF

// 2c) Move pillars and score

FOR each pillar IN pillars DO

pillar.x ← pillar.x - scrollSpeed

IF NOT pillar.passed AND pillar.x + PILLAR\_WIDTH < bird.x THEN

pillar.passed ← TRUE

score ← score + 1

IF score > highScore THEN

highScore ← score

WRITE highScore to "highscore.txt"

END IF

END IF

END FOR

// 2d) Remove off‐screen pillars

IF first pillar.x < -PILLAR\_WIDTH THEN

REMOVE first pillar from pillars

END IF

// 2e) Collision detection

birdRect ← RECT around bird center of size BIRD\_SIZE

IF birdRect.bottom > SCREEN\_HEIGHT - GROUND\_HEIGHT OR birdRect.top < 0 THEN

InitGame(); CONTINUE main loop

END IF

FOR each pillar IN pillars DO

topRect ← pillar top block rectangle

bottomRect ← pillar bottom block rectangle

IF birdRect overlaps topRect OR birdRect overlaps bottomRect THEN

InitGame(); BREAK out of pillar loop

END IF

END FOR

// 3) Render Frame

hdc ← Get device context for window

// 3a) Prepare GDI brushes/fonts

CREATE skyBrush, groundBrush, pillarBrush, birdBrush, font

// 3b) Draw background and ground

FillRect(hdc, full screen, skyBrush)

FillRect(hdc, ground area, groundBrush)

// 3c) Draw pillars

FOR each pillar IN pillars DO

FillRect(hdc, topRect, pillarBrush)

FillRect(hdc, bottomRect, pillarBrush)

END FOR

// 3d) Draw bird

FillRect(hdc, birdRect, birdBrush)

// 3e) Draw text

SelectObject(hdc, font)

TextOut(hdc, 10, 10, "Score: " + score)

TextOut(hdc, 10, 50, "High Score: " + highScore)

Release device context

// 4) Frame rate cap

Sleep(16) // approx. 60 FPS

END WHILE

Let’s walk through that detailed pseudocode section by section, explaining **why** each part exists and **what purpose** it serves in the overall game.

**1. Game Constants & Globals**

CONST SCREEN\_WIDTH = 800

CONST SCREEN\_HEIGHT = 600

CONST PILLAR\_GAP = 200

…

VAR score, highScore, pillarTimer, gameRunning

VAR bird(x, y, velocity)

LIST pillars

* **Purpose**: Centralize all “magic numbers” so you can tweak screen size, bird size, pillar speed, etc., in one place.
* **Reasoning**: Makes the code easier to maintain and balance—if you want a narrower gap or a taller window, update the constant rather than hunt through the code.
* **Globals** for score, highScore, etc., let every function read or modify the current game state without awkward parameter passing.

**2. Initialization (InitGame)**

FUNCTION InitGame()

bird ← (x=100, y=SCREEN\_HEIGHT/2, velocity=0)

CLEAR pillars

score ← 0

pillarTimer ← 0

highScore ← READ from "highscore.txt"

END FUNCTION

* **Why Reset Everything?** After the player crashes, you need a clean slate: bird back in mid-air, no pipes on screen, score reset.
* **High Score Loading**: By reading from disk here, you ensure the latest saved best score persists across sessions. Future writes only happen when that score is beaten, minimizing file I/O.

**3. Main Game Loop**

InitGame()

gameRunning ← TRUE

WHILE gameRunning DO

[1) Process Input]

[2) Update Simulation]

[3) Render Frame]

[4) Frame Rate Cap]

END WHILE

**3.1 Process Input**

FOR each pending Windows message DO

IF message == KEYDOWN(SPACE) OR MOUSE\_CLICK THEN

bird.velocity ← jumpForce

IF message == WM\_DESTROY THEN

gameRunning ← FALSE

END FOR

* **Purpose**: Catch user actions (spacebar or click) to make the bird jump, and detect window-close events to stop the loop.
* **Reasoning**: By polling all pending messages, you handle multiple quick inputs and system events before each frame—keeps controls responsive and ensures clean shutdown.

**3.2 Update Simulation**

// Bird physics

bird.velocity += gravity

bird.y += bird.velocity

// Pillar spawning

pillarTimer++

IF pillarTimer >= PILLAR\_FREQUENCY THEN

ADD new pillar at right edge with random gapY

pillarTimer = 0

END IF

// Move pillars & score

FOR each pillar IN pillars DO

pillar.x -= scrollSpeed

IF not passed AND pillar.x + width < bird.x THEN

passed = TRUE; score++

IF score > highScore THEN write to file

END IF

END FOR

// Remove off-screen pillars

IF first pillar.x < -width THEN REMOVE it

// Collision detection

IF bird hits ground OR ceiling THEN InitGame()

FOR each pillar DO

IF birdRect overlaps pillarRect THEN InitGame()

END FOR

* **Bird Physics**: Applying gravity each frame creates smooth acceleration downward; jumpForce negates that velocity for an instant rise.
* **Pillar Spawning**: pillarTimer insures a fixed interval between new pipes, controlling difficulty and spacing. Randomizing gapY makes each obstacle unique.
* **Scoring**: Marking a pillar “passed” once its right edge crosses the bird’s x prevents double-counting. Immediate high-score persistence gives instant feedback.
* **Cleanup**: Removing pillars once off-screen keeps the list small and avoids unnecessary collision checks.
* **Collision**: Central game-over condition. Checking both vertical bounds and pillar overlaps resets everything on any hit.

**3.3 Render Frame**

hdc = GetDC(window)

CREATE or initialize GDI brushes & font

// Draw order matters:

FillRect(sky)

FillRect(ground)

FOR each pillar: FillRect(top & bottom)

FillRect(bird)

Select font; TextOut(score & highScore)

ReleaseDC(hdc)

* **Why This Order?** You paint background first, then ground, then obstacles, then the bird, then UI text—so each layer appears on top of the previous.
* **GDIResources**: Encapsulating brush/font creation + cleanup ensures you don’t leak GDI handles every frame. It also groups “all drawing assets” in one place for clarity.

**3.4 Frame Rate Cap**

Sleep(16) // ~16 ms → ~60 FPS

* **Purpose**: Prevents the loop from running flat-out, which would make the bird fall too fast and hammer the CPU.
* **Reasoning**: A fixed delay approximates a steady frame rate. (In a production engine you’d use a high-precision timer to adapt the delay dynamically.)

**4. Why This Structure Matters**

1. **Separation of Concerns**
   * **Input**, **Update**, and **Render** are distinct phases. This clean separation avoids “spaghetti” loops where logic and drawing intermingle unpredictably.
2. **Deterministic Timing**
   * Consistent update intervals (gravity, pillar moves) and a frame cap make gameplay feel fair and predictable.
3. **Resource Safety**
   * By scoping GDI objects per frame and cleaning them in a destructor, you avoid slow leaks that would eventually crash the program under Windows.
4. **Easy Tuning**
   * All speeds, dimensions, and frequencies are in named constants—tweaking difficulty is a matter of changing one number.
5. **Persistent State**
   * Saving high scores to disk only when needed preserves user progress across runs without excessive file writes.

If someone asks “why did you design it this way?”, you can point to these principles—modularity, performance, maintainability, and a responsive user experience.