HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
Fakultät II
Maschinenbau und
Bioverfahrenstechnik

Console-Pong in C
(Systemprogrammierung 1)

Projektdokumentation

Autor: Mats-Luca Dagott
Betreuer: Prof. Dr.-Ing. von Marcard
5. Juni 2025

Inhaltsverzeichnis

1 Motivation & Ziele

Dieses Projekt realisiert das klassische Arcade-Spiel *Pong* als konsolenbasiertes Programm in C. Pong wurde gewählt, da es wesentliche Bestandteile eines Softwareprojekts vereint: grafische Darstellung, Eingabe, Spiellogik und Physik. Ziel war, das im ersten Projekt erworbene Wissen über die C-Programmierung noch zu vertiefen, indem ein interaktives Spiel nur mithilfe von Text und Zeichen entsteht.

Problemstellung

- **Grafische Darstellung:** Die Konsole soll als Spielfeld dienen. Reine `printf`-Ausgaben können den Bildschirm nicht dynamisch aktualisieren. Deshalb kommt `ncurses` zum Einsatz, um bildschirmweite Aktualisierungen ohne Flackern zu ermöglichen.
- **Tastatureingabe:** Um das Paddle ohne Bestätigungstaste bewegen zu können, sind non-blocking-Tastendrücke erforderlich. `ncurses` stellt `getch()` im „non-blocking"-Modus bereit, sodass kontinuierlich auf Eingaben geprüft wird.
- **Spiel-Physik & Logik:** Für Pong müssen Ball und Paddles gemäß festgelegter Regeln bewegt und Kollisionen berechnet werden. Die Ballposition wird alle 100 ms aktualisiert, Reflexionen am Paddle berücksichtigen Abprallwinkel und Geschwindigkeit, und Punktestände werden erfasst, sobald der Ball eine Spielfeldgrenze überquert.
- **Alternativen zur grafischen Umsetzung:** Anstelle von `ncurses` hätte das Spielfeld auch mit einfachen ANSI-Escape-Sequenzen (z. B. `\033[2J`) aktualisiert werden können. Dies wäre jedoch deutlich umständlicher und weniger portabel gewesen. Bibliotheken wie SDL oder Allegro wären leistungsfähiger, aber für den Einstieg in C unnötig komplex.

Ziele

- Konsolenbasierte Grafikausgabe mit `ncurses` realisieren.
- Schrittweises Vorgehen: zunächst Spielfeld zeichnen, anschließend Eingabe implementieren und schließlich Physik und KI integrieren.
- Testgetriebene Entwicklung: Unit-Tests mit dem `Unity`-Framework für zentrale Funktionen (z. B. Ball-Reflexion, Paddle-Bewegung) schreiben.
- Modularer Quelltext: Jede Komponente (z. B. `physics.c`, `ai.c`, `render.c`) erhält eine eigene Datei.

2 Architektur & Ablauf

Im folgenden Abschnitt wird erläutert, wie das Projekt in Module aufgeteilt ist und wie der Ablauf des Spiels im Loop organisiert wurde. Ziel war es, jede Komponente klar abzugrenzen, um Wartbarkeit und Übersichtlichkeit zu maximieren. Dazu wurden für jede Kernfunktionalität eigene `.c`- und `.h`-Dateien angelegt, sodass Implementierung und Schnittstellen klar getrennt sind.

2.1 Modulübersicht und -Struktur

Jedes Modul repräsentiert einen klar umrissenen Verantwortungsbereich:

- Die `.c`-Datei enthält die Implementierung der eigentlichen Funktionalität.
- Die zugehörige `.h`-Datei deklariert die Schnittstellen (Funktionen, Datentypen), die anderen Modulen zur Verfügung stehen.

Durch diese Trennung lassen sich einzelne Module unabhängig kompilieren und leichter testen. Außerdem wird so die Komplexität verringert, weil jeder Quelltext nur die notwendigen Header einbinden muss und nicht alle globalen Symbole kennt.

Datei	**Hauptaufgabe**	**Kernfunktionen**
`main.c`	Spiel-Initialisierung und Haupt-Loop	`main()`
`ai.c`	Bot-Logik	`ai_update()`
`ai.h`	Deklaration der Bot-Funktionen	`ai_update()`
`physics.c`	Ball- und Paddle-Physik	`update_paddle()`, `physics_update_ball()`
`physics.h`	Deklaration der Physik-Funktionen	`update_paddle()`, `physics_update_ball()`
`render.c`	Terminal-Ausgabe mit `ncurses`	`render_frame()`
`render.h`	Deklaration der Render-Funktionen	`render_frame()`
`input.c`	Nicht-blockierende Tastatureingabe	`input_poll()`
`input.h`	Deklaration der Eingabe-Funktion	`input_poll()`
`cleanup.c`	Beenden des `ncurses`-Modus	`cleanup_ncurses()`
`cleanup.h`	Deklaration der Cleanup-Funktion	`cleanup_ncurses()`
`config.h`	Zentrale Konfigurationskonstanten	Alle `#define`-Werte

Erläuterung zur Aufteilung:

- `main.c` übernimmt die Initialisierung von `ncurses`, startet den Spielzustand (`physics_create_game`) und verwaltet die Hauptschleife. Dadurch bleibt das Spielgerüst unabhängig von den einzelnen Modulen.
- `ai.c/h` kapseln die gesamte KI-Logik. Andere Module müssen nur `ai_update()` aufrufen, ohne interne Details zu kennen.

- `physics.c/h` enthalten alle Berechnungen zu Bewegungen, Kollisionen und Punkteständen. Die Deklarationen in `physics.h` erlauben das Testen der Physik-Funktionen unabhängig vom Rest.
- `render.c/h` sind ausschließlich für die Darstellung in der Konsole per `ncurses` zuständig. So kann die Physik separat entwickelt werden, ohne direkte Abhängigkeiten zur Ausgabe.
- `input.c/h` kapseln die Eingabelogik (non-blocking), um Tastendrücke zu erkennen. Dadurch bleibt `main.c` übersichtlich.
- `cleanup.c/h` stellt sicher, dass `ncurses` sauber beendet wird, auch wenn das Spiel vorzeitig abbricht.
- `config.h` definiert alle Konstanten (z. B. `PLAYER_ACCELERATION`, `BALL_MAX_SPEED`), um zentrale Werte an einer Stelle zu sammeln und bei Bedarf schnell anzupassen.

2.2 Spiel-Loop und Programmablauf

Der Haupt-Loop des Spiels ist in `main.c` implementiert und gliedert sich in Initialisierung, den wiederkehrenden Zyklus (Loop) und das saubere Beenden. Dabei werden Eingabe, Physik/KI und Rendering strikt voneinander getrennt:

1. **Initialisierung:**
   - `setlocale()`, `srand()` – Unicode-Unterstützung und Zufall.
   - `initscr(), cbreak(), noecho(), keypad(), curs_set()` – `ncurses` konfigurieren (non-blocking I/O, Farben).
   - `game = physics_create_game(max_x, max_y)` – Initialer Spielzustand (Paddles, Ball, Score = 0).
2. **Hauptschleife (`while (running)`):**
   (a) **Eingabe (`input.c`):**
      - `action = input_poll()` – Non-blocking-Tastendrücke (Links/Rechts/'q').
      - Bei `action.quit` Loop verlassen.

   (b) **Physik & KI (`physics.c & ai.c`, alle 100 ms):**
      - `ai_update(&game)` – Ballposition prüfen, Bot-Paddle beschleunigen.
      - `physics_player_update(&game, action.dx)` – Spieler-Paddle bewegen.
      - `physics_update_ball(&game)` – Ball in feinen Schritten bewegen, Kollisionen (Wände, Paddles) prüfen, Score/Spielende aktualisieren.

   (c) **Rendern (`render.c`, $\approx$ 60 Hz):**
      - `render_frame(&game)` – Bildschirm löschen (`erase()`), Spielfeldrahmen, Paddles und Ball zeichnen, Statuszeile (Score, Bot-Beschleunigung, Ball-Geschwindigkeit) anzeigen.

   (d) `sleep_ms(16)` – ca. 16 ms pausieren, um die Anzeige bei etwa 60 Hz zu stabilisieren.

3. **Beenden:**
    - `cleanup_ncurses()` – `ncurses` beenden, Terminal zurücksetzen.

Warum 10 Hz für Physik & KI?

- Ein Update-Intervall von 100 ms liefert eine flüssige Ballbewegung, ohne dass man einzelne Ticks wahrnimmt.
- Höhere Frequenzen (z. B. 30 Hz) beanspruchen mehr CPU-Zeit, bringen in einer reinen Konsolenumgebung jedoch kaum Verbesserung.

3 Dokumentation von Funktionen und wichtigen Variablen/Strukturen

In diesem Kapitel werden die wichtigsten Datenstrukturen und die zentralen Funktionen beschrieben. Dabei steht das Verstehen der Programmabläufe im Vordergrund – ausführliche Codezeilen wurden zugunsten klarer Beschreibungen weggelassen.

3.1 Wichtige Datenstrukturen

`paddle_t` Beschreibt ein Paddle, egal ob Spieler oder Bot.

- `float x` – Die horizontale Position (linke Kante) auf dem Spielfeld (Gleitkommawert für feine Bewegungen).
- `int y` – Die feste vertikale Zeile des Paddles (oben oder unten).
- `int width` – Die Breite in Zeichen, also wie viele Spalten das Paddle einnimmt.
- `float vx` – Die aktuelle Geschwindigkeit in horizontaler Richtung (positiv nach rechts, negativ nach links).
- `float ax` – Die aktuelle Beschleunigung in horizontaler Richtung.

`ball_t` Beschreibt den Ball im Spiel.

- `float x, y` – Die aktuelle Position in Gleitkomma, damit der Ball fein über das Spielfeld gleitet.
- `float vx, vy` – Die Geschwindigkeit in x- und y-Richtung, steuert, wie schnell und in welche Richtung sich der Ball bewegt.

`game_state_t` Beinhaltet alles, was zum aktuellen Spielgeschehen gehört.

- `int field_width, field_height` – Größe des Spielfelds, entspricht der Terminalbreite und -höhe.
- `paddle_t player, bot` – Strukturen für das Spieler- und Bot-Paddle.
- `ball_t ball` – Die Daten zum Ball.
- `int score` – Der aktuelle Punktestand des Spielers.
- `int paddle_hits` – Zähler, wie oft der Ball bereits am Paddle abgeprallt ist (eingesetzt, um den Ball nach mehreren Treffern schrittweise zu beschleunigen).

3.2 Konstanten (`config.h`)

Um zentrale Werte nicht im Code zu verteilen, wurden alle relevanten Konstanten in `config.h` gesammelt:

`PLAYER_ACCELERATION` Legt fest, wie stark das Paddle bei Tastendruck beschleunigt.

`PLAYER_MAX_SPEED` Höchstgeschwindigkeit des Spieler-Paddles, damit es nicht unkontrolliert schnell wird.

`BOT_BASE_ACCELERATION` Grundbeschleunigung des Bot-Paddles, auf die im Laufe des Spiels weiterer Bonus addiert werden kann.

`BOT_ACCEL_PER_POINT` Zusätzliche Beschleunigung pro erreichtem Punkt, damit der Bot mitwächst.

`BALL_INITIAL_SPEED` Startgeschwindigkeit des Balls, direkt nach dem Aufschlag.

`BALL_BOUNCE_MULTIPLIER` Multiplikator, um den Ball bei jedem Abprall etwas schneller zu machen.

`BALL_MAX_SPEED` Obere Grenze für die Geschwindigkeit des Balls, verhindert, dass er unberechenbar wird.

`BALL_MIN_SPEED` Untere Grenze, damit der Ball nicht stehen bleibt.

`BALL_MIN_VY_FRAC` Minimaler Anteil der vertikalen Komponente, um zu verhindern, dass der Ball nur waagerecht prallt.

`PADDLE_DAMPING` Dämpfungsfaktor, wenn kein Input erfolgt (das Paddle verlangsamt sich allmählich).

`PADDLE_STOP_EPS` Grenzwert, unterhalb dessen die Geschwindigkeit auf exakt 0 gesetzt wird.

3.3 Wichtige Funktionen – in Worten erklärt

Im Folgenden werden die zentralen Funktionen beschrieben.

3.3.1 Bewegen und Dämpfen eines Paddles: `update_paddle(paddle_t *p, float dir, float accel, float v_max, int field_w)`

Diese Funktion sorgt dafür, dass sich ein Paddle (Spieler oder Bot) horizontal über das Spielfeld bewegen kann und niemals über den Rand hinausrutscht.

- Befindet sich `dir` auf -1 oder $+1$, wird `p->ax = accel * dir` gesetzt und `p->vx` um diesen Wert erhöht. So beschleunigt das Paddle in die entsprechende Richtung.
- Ist `dir = 0`, wird das Paddle nicht aktiv beschleunigt, sondern gedämpft: `p->vx` wird mit `PADDLE_DAMPING` multipliziert. Liegt $p- > vx$ unter `PADDLE_STOP_EPS`, wird die Geschwindigkeit auf 0 gesetzt.
- Anschließend wird `p->vx` auf $\pm v\_max$ begrenzt, falls dieser Wert überschritten wird.
- Die Position wird durch `p->x += p->vx` aktualisiert.

- Falls `p->x` unter 0 oder über $field\_w - p- > width - 1$ rutscht, wird `p->x` auf den Rand gesetzt und `p->vx` auf 0 zurückgesetzt, damit das Paddle nicht aus dem Spielfeld verschwindet.

3.3.2 Einfache Bot-Logik: `ai_update(game_state_t *g)`

Der Bot folgt dem Ball, ohne komplizierte Vorhersagen. So funktioniert die Logik:

- Berechne `ball_mid = g->ball.x` und `bot_mid = g->bot.x + g->bot.width/2`.
- Liegt der Ball weiter als 0,5 Zeichen entfernt, setzt `dir = +1`, wenn sich der Ball rechts befindet, sonst `dir = -1`. Andernfalls bleibt `dir = 0`.
- Die Beschleunigung berechnet sich aus `BOT_BASE_ACCELERATION` plus einem Bonus `BOT_ACCEL_PER_POINT * g->score`, sodass der Bot mit steigendem Punktestand des Spielers schneller agiert.
- Rufe `update_paddle(&g->bot, dir, accel, BOT_MAX_SPEED, g->field_width)` auf, um das Bot-Paddle entsprechend zu bewegen oder zu dämpfen.

3.3.3 Ballbewegung, Kollisionsprüfung und Punkte: `physics_update_ball(game_state_t *g)`

Diese zentrale Funktion bewegt den Ball in kleinen Teilstücken und prüft dabei, ob er mit Wänden oder Paddles kollidiert. Anschließend werden Punkte vergeben oder Game Over ausgelöst.

- Bestimme die Anzahl der Sub-Schritte (`sub_steps`) anhand der aktuellen Geschwindigkeit: $\max(g- > ball.vx,\ g- > ball.vy)$ aufgerundet.
- Berechne `step_x = g->ball.vx / sub_steps` und `step_y = g->ball.vy / sub_steps`.
- Für jeden Sub-Schritt:
  - Aktualisiere `g->ball.x += step_x` und `g->ball.y += step_y`.
  - **Seitenwand-Kollision:** Ist `g->ball.x` 0 oder `g->field_width` 1, kehrt `g->ball.vx` das Vorzeichen um und korrigiert `g->ball.x` auf den Rand.
  - **Bot-Paddle oben:** Wenn `g->ball.vy < 0` und `g->ball.y` in der Zeile direkt unterhalb von `g->bot.y` liegt, prüfe, ob `g->ball.x` innerhalb der X-Grenzen des Bot-Paddles ist. Trifft das zu, rufe `reflect_paddle(&g->ball, &g->bot, ++g->paddle_hits)` auf, um den Ball zurückzuschicken.
  - **Spieler-Paddle unten:** Analog: Wenn `g->ball.vy > 0` und `g->ball.y` eine Zeile oberhalb von `g->player.y` erreicht, sowie `g->ball.x` im horizontalen Bereich des Spieler-Paddles liegt, ebenfalls `reflect_paddle(&g->ball, &g->player, ++g->paddle_hits)` aufrufen.
  - **Punkt oder Game Over:**
    - Fällt der Ball über den oberen Rand (`g->ball.y < 0`), erhält der Spieler einen Punkt (`g->score++`), dann wird der Ball mit `reset_ball(g)` in die Mitte gesetzt und ein Countdown mit `show_countdown()` gestartet. Die Sub-Schleife wird abgebrochen, damit der neue Aufschlag beginnt.

    - Überschreitet der Ball den unteren Rand (`g->ball.y > g->field_height`), endet das Spiel (`return false`).

- Wenn kein Game Over eintritt, gibt die Funktion `return true` zurück, damit der nächste Zyklus starten kann.

Hilfsfunktion: `reflect_paddle(ball_t *ball, const paddle_t *p, int hits)` Diese Funktion bestimmt, wie der Ball nach einem Treffen am Paddle zurückprallt, in einfachen Worten:

- Zunächst wird berechnet, wie weit der Ball vom Mittelpunkt des Paddles entfernt ist:
  `offset = (ball->x - (p->x + p->width/2)) / (p->width/2)`.
  Ein `offset` von -1 bedeutet links außen, 0 genau in der Mitte, $+1$ rechts außen.
- Die momentane Geschwindigkeit des Balls (`speed`) bestimmt zusammen mit `offset` den neuen Abprallwinkel: Liegt der Ball seitlich, erhält er einen stärkeren horizontalen Anteil, liegt er mittig, prallt er fast senkrecht ab.
- Ein Teil des Paddle-Spins (die aktuelle Bewegung `p->vx`) wird dem horizontalen Anteil hinzugefügt, damit ein bewegtes Paddle den Ball etwas mitnimmt.
- Mit jedem weiteren Paddle-Treffer wird der Ball etwas schneller, indem man eine feste Basis (`BALL_BOUNCE_MULTIPLIER`) plus einen Bonus (abhängig von `hits`) addiert.
- Abschließend wird geprüft, dass die Geschwindigkeit weder über `BALL_MAX_SPEED` noch unter ein berechnetes Minimum fällt. Das sorgt dafür, dass der Ball nicht unkontrolliert übermäßig schnell wird oder stehen bleibt.
- Das Ergebnis sind die neuen Geschwindigkeiten `ball->vx` und `ball->vy`, die den Ball in einer angemessenen Richtung und Geschwindigkeit ins Spielfeld zurückschicken.

3.3.4 Rendern des Frames: `render_frame(const game_state_t *g)`

Diese Funktion zeichnet in jedem Loop-Durchgang den kompletten Spielzustand in der Konsole neu. Die Vorgehensweise ist:

1. `erase()` – löscht den vollständigen Bildschirm, damit beim Neuzeichnen kein Flackern entsteht.
2. **Spielfeldrahmen:**
   - Obere Kante über dem Bot-Paddle: Mit `ACS_ULCORNER`, `ACS_HLINE`, `ACS_URCORNER`.
   - Untere Kante unter dem Spieler-Paddle: Mit `ACS_LLCORNER`, `ACS_HLINE`, `ACS_LRCORNER`.
   - Seitliche Linien: Zwei vertikale Linien jeweils links und rechts, die über die gesamte Feldhöhe reichen, gezeichnet mit `mvvline()`.
3. **Statuszeile:** `mvprintw(0, 2, SScore: %d (q = quit)", g->score);` – zeigt den Punktestand sowie eine Quit-Anweisung. Bot-Beschleunigung und Ball-Geschwindigkeit werden farblich hervorgehoben, um die Spielmechanik zu verdeutlichen.
4. **Paddles zeichnen:**

- Spieler-Paddle: `draw_paddle(&g->player, COLOR_PAIR(3), player_flash > 0)` – zeichnet ein farbig hinterlegtes Rechteck, das bei Treffer kurz blinkt.
- Bot-Paddle: `draw_paddle(&g->bot, COLOR_PAIR(4), bot_flash > 0)` – analog, um visuelle Rückmeldung zu geben.

5. **Ball zeichnen:** `mvaddch((int)g->ball.y, (int)g->ball.x, ACS_DIAMOND)` – zeichnet ein kleines Rauten-Symbol an der aktuellen Ballposition.
6. `refresh()` – alles Gestrichene auf den Bildschirm übertragen.

So entsteht ein flüssiges Bild in der Konsole, das mit etwa 60Hz aktualisiert wird und sowohl Spielstände, Paddles als auch Ballbewegung übersichtlich anzeigt.

3.3.5 Nicht-blockierende Eingabe: `input_action_t input_poll(void)`

Für die nicht-blockierende Eingabe wird `getch()` von `ncurses` in den „nodelay"-Modus versetzt. Die Funktion liest daraufhin sofort den zuletzt gedrückten Tastencode ein, ohne auf Enter zu warten. Der Ablauf:

- `int ch = getch();` liest den Tastencode oder liefert `ERR`, wenn keine Taste gedrückt wurde.
- In einem `switch(ch)` wird ausgewertet:
  - `KEY_LEFT` → `action.dx = -1`, Paddle Move nach links.
  - `KEY_RIGHT` → `action.dx = +1`, Paddle Move nach rechts.
  - `'q'` oder `'Q'` → `action.quit = 1`, Abbruchsignal.
  - Standardfall (kein Key oder andere Taste): `action.dx = 0; action.quit = 0`.
- Rückgabe: `action` mit den beiden Feldern `dx` und `quit`.

3.3.6 Cleanup von `ncurses`: `void cleanup_ncurses(void)`

Zum sauberen Beenden des Spiels muss `ncurses` wieder zurückgesetzt werden. Diese Funktion ruft nur `endwin()` auf, damit der Terminal in den normalen Modus zurückkehrt.

3.3.7 Initialisierung des Spiels: `game_state_t physics_create_game(int width, int height)`

Beim Programmstart legt diese Funktion den kompletten Spielzustand an. Schrittweise:

- Zunächst wird geprüft, ob `width` oder `height` unter den Mindestmaßen (`MIN_TERMINAL_WIDTH`, `MIN_TERMINAL_HEIGHT`) liegen. Falls ja, werden diese festgelegten Mindestwerte verwendet, damit Spielfeld und Elemente nicht zu klein ausfallen.
- Anschließend wird `g.field_width = width` und `g.field_height = height` gespeichert.
- Die Zähler `g.score = 0` und `g.paddle_hits = 0` werden initialisiert.
- Das Spieler-Paddle wird in der Mitte horizontal und ganz unten positioniert:

  - Berechne `player.width = width / PADDLE_WIDTH_RATIO`, damit das Paddle ausreichend breit, aber nie zu schmal wird.
  - Setze `player.x = (width - player.width) / 2`, also horizontal genau mittig.
  - `player.y = height - 2` positioniert das Paddle zwei Zeilen über dem unteren Rand, damit noch Platz für den Rahmen bleibt.
  - Setze `player.vx = player.ax = 0.0f`.
- Das Bot-Paddle wird identisch angelegt, aber eine Zeile unterhalb des oberen Rahmens: `bot.y = 1`, sowie `bot.vx = bot.ax = 0.0f`.
- Der Ball wird in der Spielfeldmitte platziert:
  - `ball.x = width/2.0f`, `ball.y = height/2.0f`.
  - Die horizontale Startgeschwindigkeit `ball.vx` wird zufällig auf $\pm BALL\_INITIAL\_SPEED$ gesetzt, damit der Aufschlag mal nach links, mal nach rechts fliegt.
  - Die vertikale Geschwindigkeit `ball.vy = -BALL_INITIAL_SPEED`, damit der Ball zuerst nach oben in Richtung Bot fliegt.
- Rückgabe: Die komplett initialisierte Struktur `game_state_t g`.

4 Fazit & kritischer Rückblick

Erreichtes

- **Grafik:** Mit `ncurses` lässt sich das Spielfeld sauber und flüssig (ca. 60 Hz) darstellen, ohne dass das Terminal störend flackert. Rahmen, Paddles und Ball werden in jeder Iteration präzise neu gezeichnet.
- **Eingabe:** Dank non-blocking-`getch()` konnten Tastendrücke unmittelbar ausgewertet werden. Die Funktion `input_poll()` liefert eine kompakte Struktur (`dx` und `quit`) und hält die Hauptschleife schlank.
- **Physik & KI:**
  - Die Physik-Funktion `physics_update_ball()` bewegt den Ball in kleinen Teilstücken und berücksichtigt Kollisionsabpraller, Punktvergabe und Game Over sauber.
  - Das Paddle-Update (`update_paddle()`) bricht Bewegung in Beschleunigung, Dämpfung und Randbegrenzung herunter, sodass ein realistischer Bewegungsfluss entsteht.
  - Die Bot-Logik (`ai_update()`) reagiert dynamisch auf die Ballposition und steigende Punktzahl des Spielers, ohne komplexe Vorhersagen – das Ergebnis ist ein angemessen anspruchsvoller Gegner für den Einsteiger.
- **Modularität:** Durch separate `.c`- und `.h`-Dateien für jede Kernfunktion (Eingabe, Rendering, Physik, Bot, Cleanup) bleibt der Code übersichtlich. Änderungen in einem Modul wirken sich nicht auf andere Module aus, und Unit-Tests können gezielt Teilfunktionen prüfen.

- **Qualität:** Kommentare und einheitliche Konstantendefinitionen über `config.h` sorgen dafür, dass der Code leicht verständlich und wartbar ist.

Testgetriebene Entwicklung Das `Unity`-Framework wurde eingesetzt, um zentrale Funktionen wie `update_paddle()`, `reflect_paddle()` und `physics_update_ball()` im Vorfeld zu verifizieren und erlaubte es, Fehler frühzeitig zu erkennen, bevor sie im Spielablauf sichtbar wurden.

Verbesserungsideen

- **KI-Verbesserung:** Der Bot sollte nicht nur stur dem Ball folgen, sondern etwas menschlicher wirken. Mögliche Ansätze:
  - Reaktionsverzögerung: Berechne die nächste Bewegung nur alle X ms, statt bei jedem Physik-Update, um das Ruckeln zu reduzieren.
  - Unterschiedliche Schwierigkeitsgrade: Leicht, Mittel, Schwer mit variierenden Parametern wie `BOT_ACCEL_PER_POINT` oder künstlicher Verzögerung.
  - Prognose der Ballbahn: Statt sofortiger 1:1-Bewegung könnte der Bot den letzten Ballwinkel extrapolieren und so versuchen, den Ball schon etwas früher „abzufangen“.
- **Menüs und Highscore:** Aktuell startet das Spiel direkt und endet abrupt. Ein kleines Menüsystem würde das Nutzererlebnis deutlich aufwerten:
  - Startbildschirm: Noch bevor die Schleife beginnt, eine kurze Anleitung (z. B. „←/→ steuern, q beenden“) und Auswahl des Schwierigkeitsgrads.
  - Pause- und Game-Over-Menü: Mit der Möglichkeit, das Spiel zu pausieren (`p` zum Pausieren, `r` zum Fortsetzen) und nach Spielende den Highscore anzuzeigen.
  - Highscore-Liste: Nach Spielende den Bestwert in einer lokalen Datei speichern und anzeigen, damit man sehen kann, wie viele Punkte bisher maximal erreicht wurden.
- **Multiplayer-Modus:** Ein abgewandelter Modus, in dem zwei Spieler über getrennte Eingabetasten (z. B. W/S vs. ←/→) gegeneinander antreten.
  - Implementiere eine zweite `input_poll()`-Funktion, die neben dem linken Spieler-Paddle auch den rechten Spieler steuert.
  - Passe die Spielfeldlogik so an, dass der Ball bei einem Treffer eines der beiden Paddles jeweils zurückprallt und nur dann Punkte erhält, wenn er hinter einem Paddle ins Aus geht.
  - Nach jedem Punkt wechselt das Aufschlagsrecht, und das Spiel beginnt an der Ballmitte mit zufälliger Richtung.

Insgesamt hat dieses Projekt gezeigt, dass ein textbasiertes Spiel wie Pong alle relevanten Aspekte der Systemprogrammierung (I/O, Timing, Modularität, Testbarkeit) vereint und sich auch hervorragend als Lernprojekt eignet. Die Struktur und die einzelnen Module bilden eine gute Basis für einen weiteren Ausbau, ohne dabei den Überblick zu verlieren.

5 Quellen & Hilfsmittel

Literatur

[1] *ncurses Manual*, `https://invisible-island.net/ncurses/`

[2] *OpenAI ChatGPT*, eingesetzt für Analyse und Code-Optimierung (Verwendung: 2025).