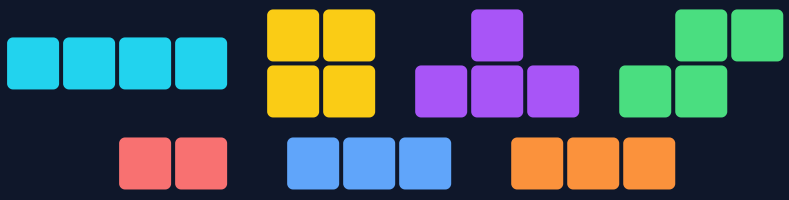
DAS OFFIZIELLE PROJEKT-LERNBUCH

Kiste programmieren lernen mit Kistris

Von der ersten Zeile Code zum fertigen Tetris-Spiel —
Schritt für Schritt, auf Deutsch, von null auf.



Für die Programmiersprache **Kiste** von **Sascha Rust**

1. Auflage 2026

Über dieses Buch

Du hältst ein Versprechen in der Hand: Wenn du dieses Buch durcharbeitest, hast du am Ende mit eigenen Händen ein komplettes Tetris-Spiel programmiert — **Kistris** — als echte, native Windows-Anwendung. Keine Spielzeug-Übungen, kein „Stell dir vor, hier wäre ein Programm". Du baust das echte Ding, mit Sound, Rangliste, Ghost Piece und allem Drum und Dran. Und nebenbei — fast aus Versehen — lernst du die Programmiersprache **Kiste** von Grund auf.

Dieses Buch setzt **nichts** voraus. Nicht eine Zeile Programmiererfahrung. Wenn du einen Computer einschalten und eine Datei speichern kannst, reicht das.

Wie dieses Buch funktioniert

- **Du tippst selbst.** Jedes Code-Beispiel ist zum Nachbauen gedacht. Lesen allein macht keinen Programmierer — so wie das Lesen eines Kochbuchs noch keine Lasagne ergibt.
- **Aufgaben** festigen jedes Kapitel. Schwierigkeit: ★ leicht, ★★ mittel, ★★★ knifflig. Lösungen stehen in Anhang A — aber erst selbst probieren, sonst war's das mit dem Lerneffekt.
- **Teste dich** — kleine Quizfragen am Kapitelende. Antworten ebenfalls in Anhang A.
- **Bau-Stopps** markieren Stellen, an denen dein Programm laufen muss, bevor du weiterliest. Nicht überspringen! Ein Fehler, den du sofort findest, ist ein kleiner Fehler.
- **Jeder Code in diesem Buch wurde real ausgeführt.** Die Konsolen-Beispiele mit `kiste run`, die Spiel-Teile als nativ gebaute `.exe`. Was hier steht, läuft.

Die Hinweis-Boxen

Hinweis
Gut zu wissen — Hintergründe und Zusammenhänge.

Achtung
Häufige Fehler und Stolperfallen. Wer das liest, spart Debugging-Tränen.

Tipp
Tricks, die das Leben leichter machen.

Build-Subset
Kiste hat zwei Ausführungswege: den flexiblen `kiste run`-Modus und den pfeilschnellen nativen Build. Der native Compiler beherrscht (Stand heute) ein *Subset* der Sprache — diese Boxen zeigen dir, wo die Grenzen liegen und welcher dokumentierte Weg drumherum führt. Kistris als GUI-Programm läuft nur nativ, deshalb nehmen wir diese Grenzen ernst — und du lernst nebenbei, wie Profis mit Werkzeug-Grenzen umgehen.

Aufgabe

Jetzt bist du dran.

Teste dich

Kurze Fragen zum Kapitel.

Bau-Stopp

Programm ausführen, Ergebnis prüfen, erst dann weiter.

Was du brauchst

- Einen Windows-PC (Kiste läuft auch anderswo, aber dieses Buch zeigt Windows-Wege)
- Die `kiste.exe` — und idealerweise die **Kiste-IDE** (Editor mit eingebautem Ausführen-Knopf)
- Für den nativen Build: LLVM und MinGW-w64 (Kapitel 2 erklärt die Installation)
- Etwa 20–30 Stunden Neugier, verteilt auf so viele Abende, wie du magst

Noch mehr Kiste

Dieses Buch bringt dich über das Spiel zur Sprache. Das allgemeine Nachschlagewerk **„Kiste — Das Lernbuch"** (KISTE-LERNBUCH.md im Kiste-Repository) behandelt zusätzlich Themen wie Klassen, Fehlerbehandlung, JSON, Netzwerk und vieles mehr. Wenn du nach Kistris weitermachen willst: Dort geht die Reise weiter.

Inhalt

1 Willkommen — die Geschichte von Kiste

1.1 Warum eine deutsche Programmiersprache?

Fast alle Programmiersprachen dieser Welt sprechen Englisch. `if`, `while`, `function`, `return` — wer programmieren lernt, lernt immer zwei Sprachen gleichzeitig: die Programmiersprache *und* Englisch-Vokabeln. Für viele ist genau das die erste Hürde, an der es hakt.

Kiste räumt diese Hürde weg. In Kiste heißt es `wenn`, `solange`, `funktion` und `gib`. Du denkst auf Deutsch, du schreibst auf Deutsch, und der Computer versteht dich trotzdem — denn unter der Haube ist Kiste eine vollwertige, moderne Programmiersprache: mit eigenem Compiler, der echte native `.exe`-Dateien erzeugt, mit Klassen, Modulen, einer umfangreichen Standard-Bibliothek und einer eigenen grafischen Oberfläche. Kiste ist keine „Lernsprache mit Stützrädern" — sie ist eine General-Purpose-Sprache, deren Ökosystem einfach noch jung ist. Das Spiel, das du in diesem Buch baust, ist der Beweis: ein komplettes Tetris als eine einzige native Windows-Anwendung, ohne dass irgendetwas anderes installiert sein muss.

1.2 Wie Kiste entstand — und warum sie so heißt

Kiste ist mein Herzensprojekt — ich bin **Sascha Rust**. Meine Idee: eine deutsche Programmiersprache für fast alle Zwecke — nicht als Spielerei, sondern mit einigen handfesten Verbesserungen gegenüber dem Gewohnten, und so zugänglich, dass wirklich *jeder* den Einstieg schafft.

Und der Name? Das ist die vielleicht ehrlichste Namensgeschichte der Software-Geschichte. Ich hatte zuvor an einem ganz anderen Projekt gearbeitet: einem Onlineshop namens *Kiste* — die Domain war registriert, das Projekt wurde fertig. Doch als es so weit war, merkte ich: Einen Shop zu *betreiben* bedeutet Selbstständigkeit, AHV, Buchhaltung, Kundenreklamationen — kurz: einen Berg Verwaltung, der mit dem eigentlichen Bauen nichts mehr zu tun hat. Also ließ ich den Shop Shop sein. Übrig blieben eine schöne Domain, ein freier Name — und ein Herzensprojekt, das noch keinen hatte. So wurde aus dem Onlineshop-Namen der Name der Programmiersprache: **Kiste**.

> **Und es passt ja auch**
>
> Eine Kiste ist ein ehrliches Ding: Man legt etwas hinein, man holt es wieder heraus, und sie trägt mehr, als man ihr ansieht. Variablen, Listen, Funktionen — am Ende ist Programmieren genau das: Dinge ordentlich in Kisten packen. Der Name hätte kaum besser gewählt sein können, selbst wenn man es geplant hätte. Hat man aber nicht. Umso besser.

1.3 Danke, André

Kein Projekt dieser Größe entsteht allein. Ein besonderer Dank gehört **André Ryser**: Er hat mich durch die gesamte Entwicklungszeit tatkräftig unterstützt, Kiste und die damit gebauten Programme immer wieder geduldig getestet — und er stellt als Sponsor auf **bitpoint.ch** das gesamte Hosting kostenlos zur Verfügung, ohne dafür je eine Gegenleistung zu verlangen. Solche Menschen sind selten. Ich danke dir, André.

1.4 Warum ausgerechnet Tetris?

Weil Tetris das perfekte Lernspiel ist — nicht zum Spielen (na gut, auch zum Spielen), sondern zum *Bauen*:

- Es ist **klein genug**, dass ein Mensch es komplett verstehen kann — jede Zeile.
- Es ist **groß genug**, dass fast alles Wichtige vorkommt: Variablen, Schleifen, Funktionen, Listen, Dateien, Grafik, Sound, Tastatur, Spielzustände.
- Es gibt **sofort sichtbares Feedback**: Jeder Fortschritt landet als fallender, bunter Stein auf deinem Bildschirm.
- Und es ist seit 1984 der unwiderlegte Beweis, dass man kein 3-D-Raytracing braucht, um Menschen die Nacht um die Ohren zu schlagen.

Unser Tetris heißt — natürlich — **Kistris**. So sieht es fertig aus:

Dein Ziel: das fertige Kistris mit Titelbild, Rangliste, Musik — und allem, was dazugehört.

1.5 Der Weg dorthin

Wir gehen in drei Etappen:

Teil	Kapitel	Was passiert
I — Grundlagen	2–8	Du lernst die Sprache in der Konsole: Variablen, Entscheidungen, Schleifen, Funktionen, Listen. Schon hier baust du echte Kistris-Bausteine — die Punkte-Staffel, das Spielfeld, den Zufalls-Beutel — nur eben erst mal als Text.
II — Fenster	9–10	Der Sprung auf den Bildschirm: Fenster, Leinwand, Game-Loop, Tastatur. Am Ende steuerst du einen fallenden Block — das Herz von Kistris schlägt.
III — Kistris	11–16	Stein für Stein entsteht das ganze Spiel: die sieben Tetrominoes, Schwerkraft und Kollision, Reihen löschen, Menü und Pause, Rangliste, Sound. Am Ende: dein fertiges Spiel als `.exe`.

Das Wichtigste im ganzen Buch

Tippe die Beispiele **selbst** ab. Ja, wirklich. Abtippen fühlt sich langsam an — aber deine Finger lernen die Sprache mit, und die kleinen Tippfehler, die dabei passieren, sind die besten Lehrer, die es gibt. Kopieren ist fürs Büro. Hier wird gelernt.

Teste dich (Kapitel 1)

1. Warum heißt Kiste „Kiste"?
2. Ist Kiste eine reine „Lernsprache"? Begründe kurz.
3. Wer stellt das Hosting für Kiste zur Verfügung — und zu welchem Preis?

(Antworten in Anhang A — aber diese drei solltest du jetzt schon können.)

2 Werkzeuge & dein erstes Programm

In diesem Kapitel lernst du

- wie du Kiste-Programme ausführst (`kiste run`) und baust (`kiste build`)
- dein erstes Programm: `sag`
- Eingaben lesen mit `frage`
- Kommentare schreiben

2.1 Die Werkzeugkiste

Du brauchst genau ein Programm: `kiste.exe`. Dazu empfehlenswert: die **Kiste-IDE** — ein Editor, der Kiste versteht, Fehler markiert und deine Programme mit **F5** ausführt. Du kannst aber genauso gut mit einem beliebigen Texteditor und der Eingabeaufforderung arbeiten. Eine `.ki`-Datei ist einfach eine Textdatei.

Kiste hat zwei Ausführungswege, und beide wirst du benutzen:

Befehl	Was passiert	Wofür
`kiste run programm.ki`	Führt dein Programm sofort aus, Zeile für Zeile. Voller Sprachumfang.	Lernen, Entwickeln, Konsolen-Programme — **Teil I dieses Buchs**
`kiste build programm.ki`	Übersetzt dein Programm in eine echte, eigenständige `.exe` — 20- bis 50-mal schneller.	Fertige Programme — und alles mit Fenstern: **Teil II und III dieses Buchs**

Warum zwei Wege?

`kiste run` ist wie Kochen in der eigenen Küche: sofort, flexibel, zum Probieren. `kiste build` ist das Restaurant: Das Ergebnis kannst du jedem servieren — eine `.exe`, die ohne Kiste-Installation auf jedem Windows-PC läuft. Die grafische Oberfläche von Kiste gibt es nur im gebauten Programm, deshalb wechseln wir ab Kapitel 9 zu `kiste build` (in der IDE bleibt es einfach F5).

Für den nativen Build ist alles schon dabei

`kiste build` funktioniert sofort — Kiste bringt seine eigene Build-Werkzeugkiste mit. Wenn du Kiste über den Installer (oder die Kiste-IDE) eingerichtet hast, musst du *nichts* zusätzlich installieren: kein LLVM, kein GCC, kein PATH-Gebastel. Nutzt du nur die blanke `kiste.exe` von der Kommandozeile und meldet `kiste build` beim ersten Versuch fehlende Werkzeuge, holt `kiste toolchain install` sie einmalig nach — danach nie wieder.

2.2 Hallo, Kiste!

Erstelle eine Datei `hallo.ki` mit dieser einen Zeile:

`hallo.ki`

```
sag "Hallo, Kiste!"
```

Und führe sie aus:

```
kiste run hallo.ki
```

```
AUSGABE
Hallo, Kiste!
```

Glückwunsch — du bist jetzt offiziell Programmierer. Was hier passiert:

- `sag` ist ein eingebauter Befehl: „Schreib das auf den Bildschirm."
- `"Hallo, Kiste!"` ist ein **Text** — alles zwischen Anführungszeichen.
- Kein Semikolon, kein Schnickschnack. Eine Zeile, ein Befehl.

2.3 Der Computer fragt zurück: `frage`

begruessung.ki

```
nimm name = frage("Wie heißt du? ")
sag "Hallo {name}! Schön, dass du Kiste lernst."
```

`frage` zeigt den Text an, wartet, bis du etwas tippst und Enter drückst — und liefert das Getippte zurück. Das Ergebnis landet in der Variable `name` (dazu gleich mehr). Die geschweiften Klammern `{name}` setzen den Wert mitten in den Text ein. In der Kiste-IDE erscheint die Eingabezeile automatisch unten im Ausgabe-Bereich.

2.4 Kommentare — Notizen an dich selbst

```
// Alles hinter zwei Schrägstrichen ignoriert Kiste.
sag "Das hier läuft."   // auch hinter Code erlaubt

/*
   Mehrzeilige Kommentare gehen so.
   Praktisch, um Code-Blöcke vorübergehend auszuschalten.
*/
```

Kommentare wirken am Anfang überflüssig — bis du dein eigenes Programm nach zwei Wochen wieder öffnest und dich fragst, welcher Fremde diesen Code geschrieben hat. (Es warst du.) In Kistris werden Kommentare später erklären, *warum* der Code so ist, wie er ist.

Aufgaben

2.1 ★ Schreibe ein Programm, das drei Zeilen ausgibt: deinen Namen, dein Lieblingsspiel und den Satz „Bald kann ich das selbst programmieren."

2.2 ★ Erweitere `begruessung.ki`: Frage zusätzlich nach der Lieblingsfarbe und gib beides in einem Satz aus.

2.3 ★★ Was passiert, wenn du das schließende Anführungszeichen bei `sag "Hallo` vergisst? Probier es absichtlich aus und lies die Fehlermeldung in Ruhe. Fehlermeldungen sind keine Strafe — sie sind eine Wegbeschreibung.

Teste dich (Kapitel 2)

1. Mit welchem Befehl führst du `spiel.ki` direkt aus?
2. Was erzeugt `kiste build spiel.ki`?
3. Warum brauchen wir ab Teil II den nativen Build?

3 Variablen, Zahlen und Texte

In diesem Kapitel lernst du

- Variablen anlegen mit `nimm`, Konstanten mit `fest`
- die wichtigsten Datentypen: `ganz`, `komma`, Text
- Rechnen — und Werte in Texte einsetzen (Interpolation)

3.1 `nimm` — Werte aufbewahren

Eine Variable ist eine beschriftete Kiste (na also, da ist der Name wieder), in die du einen Wert legst. In Kiste erstellst du sie mit `nimm`:

```
nimm punkte = 0
nimm spieler = "Sascha"
nimm tempo = 7.5
```

Lies es wörtlich: „*Nimm* punkte, das ist 0." Danach kannst du den Wert jederzeit benutzen und verändern:

```
punkte = punkte + 100
sag punkte
```

```
AUSGABE
100
```

Die Zeile `punkte = punkte + 100` verwirrt am Anfang jeden. Sie ist keine Mathe-Gleichung, sondern ein Befehl von rechts nach links: „Rechne `punkte + 100` aus und lege das Ergebnis zurück in `punkte`." In Kistris steht später fast dieselbe Zeile — nur mit `800 * level` dahinter, wenn du vier Reihen auf einmal löschst.

Nur beim ersten Mal `nimm`

`nimm` *erstellt* eine Variable. Danach weist du nur noch zu: `punkte = 200`. Ein zweites `nimm punkte = …` im selben Block ist ein Fehler — die Kiste existiert ja schon. Außerdem sind ein paar Wörter reserviert: `zahl`, `text`, `liste`, `karte` und `wahrheit` sind Typ-Namen — nenn deine Variable also lieber `meine_liste` statt `liste`.

3.2 `fest` — Werte, die sich nie ändern

```
fest BREITE = 10
fest HÖHE = 20
sag "Das Spielfeld hat {BREITE * HÖHE} Zellen."
```

```
AUSGABE
Das Spielfeld hat 200 Zellen.
```

`fest` ist das Versprechen: „Dieser Wert bleibt." Per Konvention schreibt man Konstanten GROSS. Kistris beginnt mit genau diesen zwei Zeilen — das Spielfeld ist 10 Zellen breit und 20 hoch, und das soll gefälligst so bleiben, auch wenn um drei Uhr nachts jemand am Code schraubt.

3.3 Zahlen: `ganz` und `komma`

Kiste unterscheidet ganze Zahlen (`42`, `-17`, `1_000_000` — der Unterstrich ist nur Deko zum besseren Lesen) und Kommazahlen (`3.14` — mit Punkt, englische Schreibweise). Für Kistris brauchen wir fast nur ganze Zahlen: Punkte, Level, Zellen-Koordinaten. Rechnen geht, wie du es erwartest:

```
nimm level = 3
nimm punkte = 0
punkte = punkte + 800 * level    // Punkt vor Strich gilt
sag "Nach einem Tetris: {punkte} Punkte!"
```

```
AUSGABE
Nach einem Tetris: 2400 Punkte!
```

Es gibt `+`, `-`, `*`, `/` und `%` (den Rest einer Division: `7 % 3` ist `1`). Klammern setzen die Reihenfolge: `(100 + 200) * 2`.

Build-Subset: Division bei ganzen Zahlen

Im `kiste run`-Modus kannst du mit allem rechnen. Der native Compiler beherrscht für `ganz` aktuell `+ - *` — Division `/` und Rest `%` für ganze Zahlen kommen dort erst noch. Klingt schlimm? Ist es nicht: In Kapitel 13 siehst du, wie Kistris das Level *ohne* Division berechnet — mit einem Trick, der sogar eleganter ist. Merke fürs Spiel: **ganze Zahlen nur mit + − ×**.

3.4 Texte und die Zauberklammern `{ }`

Texte stehen in Anführungszeichen. Das Beste an Kiste-Texten ist die **Interpolation**: In den geschweiften Klammern darf eine Variable oder sogar eine kleine Rechnung stehen — Kiste setzt den Wert ein:

```
nimm spieler = "Sascha"
nimm level = 3
sag "Spieler: {spieler}"
sag "Level: {level}"
sag "Nächstes Level: {level + 1}"
```

```
AUSGABE
Spieler: Sascha
Level: 3
Nächstes Level: 4
```

Texte lassen sich auch zusammenbauen, indem man interpoliert — das nutzt Kistris später ständig, etwa um die Rangliste-Datei zu erzeugen:

```
nimm zeile = ""
zeile = "{zeile}□"
zeile = "{zeile}□"
sag zeile     // □□ – Stück für Stück angebaut
```

Und `länge(text)` verrät, wie viele Zeichen ein Text hat — Kistris prüft damit, ob der Spieler überhaupt einen Namen eingetippt hat (sonst heißt er „Anonym").

Aufgaben

3.1 ★ Lege `fest ZELLE = 30` (Pixelgröße einer Spielfeld-Zelle) an und gib aus, wie breit das Spielfeld in Pixeln ist (`BREITE * ZELLE`).

3.2 ★★ Ein Hard Drop in Kistris gibt 2 Punkte pro übersprungener Zeile. Der Stein fällt 15 Zeilen, danach löschst du 2 Reihen auf Level 4 (2 Reihen = 300 Basispunkte × Level). Berechne den Gesamtgewinn in einem Programm — erwartetes Ergebnis: 1230.

3.3 ★★ Frage mit `frage` nach dem Namen des Spielers und gib aus: `Hallo NAME, dein Name hat X Buchstaben.`

Teste dich (Kapitel 3)

1. Was ist der Unterschied zwischen `nimm` und `fest`?
2. Was gibt `sag "{3 + 4 * 2}"` aus?
3. Warum darfst du deine Variable nicht `liste` nennen?
4. Welche Rechenarten beherrschen ganze Zahlen im nativen Build?

4 Entscheidungen — wenn, sonstwenn, sonst

In diesem Kapitel lernst du

- Verzweigungen mit `wenn` / `sonstwenn` / `sonst`
- Vergleiche: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Bedingungen verknüpfen mit `und` / `oder`
- … und du baust die echte Punkte-Staffel von Kistris

4.1 Programme, die abbiegen

Bis jetzt liefen deine Programme stur von oben nach unten. Spannend wird es, wenn der Code *abbiegen* kann. In Tetris gibt es eine berühmte Stelle, an der genau das passiert: die Punktevergabe. Eine gelöschte Reihe ist nett. Vier auf einmal — ein „Tetris" — ist überproportional viel wert. Das ist eine Entscheidung, und so sieht sie in Kiste aus:

kap04_entscheidungen.ki

```
nimm reihen = 4
nimm level = 2
nimm basis = 0

wenn reihen == 1 {
    basis = 100
} sonstwenn reihen == 2 {
    basis = 300
} sonstwenn reihen == 3 {
    basis = 500
} sonst {
    basis = 800
}

sag "Du löschst {reihen} Reihen auf Level {level}."
sag "Das gibt {basis * level} Punkte."
```

AUSGABE

```
Du löschst 4 Reihen auf Level 2.
Das gibt 1600 Punkte.
```

Die Anatomie:

- `wenn BEDINGUNG { … }` — der Block in den geschweiften Klammern läuft nur, wenn die Bedingung wahr ist.
- `sonstwenn` — „falls nicht, prüf das hier". Beliebig viele davon.
- `sonst` — der Auffangkorb für alles andere. Optional.
- Genau **ein** Block gewinnt. Sobald eine Bedingung zutrifft, werden die restlichen gar nicht mehr angeschaut.

`=` ist nicht `==`

Ein Gleichheitszeichen *weist zu* (`punkte = 100`), zwei Gleichheitszeichen *vergleichen* (`punkte == 100`). Diesen Fehler macht jeder Programmierer mindestens einmal pro Karriere. Heute ist dein Tag vielleicht schon abgehakt.

4.2 Die Vergleiche

Schreibweise	Bedeutung	Beispiel aus Kistris
`==`	gleich	`taste == "leer"` — wurde die Leertaste gedrückt?
`!=`	ungleich	`feld_hole(x, y) != 0` — ist die Zelle belegt?
`<` / `>`	kleiner / größer	`s < 3` — Tempo-Untergrenze erreicht?
`<=` / `>=`	kleiner-gleich / größer-gleich	`x >= BREITE` — aus dem Feld gerutscht?

4.3 `und` / `oder`

```
wenn reihen >= 4 und level >= 2 {
    sag "Ein Tetris auf hohem Level – stark!"
}

wenn taste == "leer" oder taste == "eingabe" {
    sag "Spielstart!"
}
```

`und` verlangt, dass *beide* Seiten stimmen. `oder` ist zufrieden, wenn *mindestens eine* stimmt. Im fertigen Kistris startet das Spiel mit Leertaste *oder* Eingabetaste — exakt die zweite Bedingung von oben.

Wahr und falsch

Kiste kennt die Wahrheitswerte `wahr` und `falsch`. In den Spiel-Beispielen dieses Buchs benutzen wir stattdessen oft die Zahlen `1` und `0` — das ist das bewährte Muster der Kiste-GUI-Demos (z. B. liefert die Tastatur-Abfrage später `1` für „gedrückt") und funktioniert überall gleich.

Aufgaben

4.1 ★ Schreibe ein Programm mit `nimm level = 7`, das ausgibt: „Anfänger" (Level 1–3), „Fortgeschritten" (4–8) oder „Tetris-Gott" (ab 9).

4.2 ★★ Kistris-Regel: Ein Stein darf nach links, wenn `x > 0` ist *und* die Zielzelle frei ist (tu so, als wäre sie frei, wenn `x != 3`). Schreibe die Prüfung für `nimm x = 5` und für `nimm x = 3` — was passiert jeweils?

4.3 ★★ Frage den Nutzer mit `frage` nach einer Reihenzahl (1–4) und gib die passenden Basispunkte aus. Tipp: `frage` liefert Text — mit `als_ganz(antwort)` machst du eine Zahl daraus.

Teste dich (Kapitel 4)

1. Wie viele Blöcke einer `wenn` / `sonstwenn` / `sonst`-Kette laufen maximal?
2. Was ist falsch an `wenn punkte = 100 { … }`?
3. Wann ist `a oder b` wahr?

5 Schleifen — für und solange

In diesem Kapitel lernst du

- Zähl-Schleifen mit `für … bis` (und `schritt`)
- Bedingungs-Schleifen mit `solange`
- verschachtelte Schleifen — der Schlüssel zum Spielfeld

5.1 `für` — wenn du weißt, wie oft

Ein Tetris-Spielfeld hat 200 Zellen. Niemand schreibt 200-mal denselben Befehl — dafür gibt es Schleifen:

kap05_schleifen.ki (Auszug)

```
sag "Countdown:"
für i = 3 bis 1 schritt -1 {
    sag i
}
sag "Los!"
```

```
AUSGABE
Countdown:
3
2
1
Los!
```

`für i = 3 bis 1 schritt -1` heißt: Starte bei 3, geh in Schritten von −1, hör bei 1 auf — **einschließlich** 1. Ohne `schritt` wird einfach hochgezählt: `für i = 0 bis 9` läuft zehnmal (0, 1, … 9). Dieses „bis einschließlich" ist wichtig: Die Spielfeld-Spalten heißen 0 bis 9, also lautet die Schleife darüber `für x = 0 bis BREITE - 1`.

5.2 Texte in Schleifen aufbauen

```
nimm reihe = ""
für x = 1 bis 10 {
    reihe = "{reihe}□"
}
sag reihe
```

```
AUSGABE
□□□□□□□□□□
```

Pro Durchlauf wird ein Kästchen angehängt. Merk dir dieses Muster — in Kapitel 7 malen wir damit das komplette Spielfeld in die Konsole, und im fertigen Kistris entsteht so die Rangliste-Datei.

5.3 `solange` — wenn du es nicht weißt

Manchmal weißt du nicht, wie oft etwas passieren soll — nur *bis wann*. Das Kistris-Tempo zum Beispiel: Mit jedem Level fällt der Stein schneller, aber irgendwann ist Schluss (sonst wäre das Spiel unspielbar):

```
nimm schwelle = 48
nimm level = 1
solange schwelle > 8 {
    sag "Level {level}: alle {schwelle} Ticks eine Zeile"
    level = level + 1
    schwelle = schwelle - 8
}
```

AUSGABE

```
Level 1: alle 48 Ticks eine Zeile
Level 2: alle 40 Ticks eine Zeile
Level 3: alle 32 Ticks eine Zeile
Level 4: alle 24 Ticks eine Zeile
Level 5: alle 16 Ticks eine Zeile
```

`solange` prüft die Bedingung *vor* jedem Durchlauf und hört auf, sobald sie nicht mehr stimmt. Im fertigen Kistris erledigt `solange` drei Jobs: den Hard Drop (falle, *solange* nichts im Weg ist), das Ghost Piece (gleiche Idee, nur unsichtbar) und den Level-Aufstieg.

Die Endlos-Schleife

Wenn die Bedingung *nie* falsch wird, läuft `solange` für immer — dein Programm hängt. Vergiss also nie die Zeile, die die Bedingung irgendwann kippen lässt (oben: `schwelle = schwelle - 8`). Falls es doch passiert: `Strg+C` im Terminal bzw. Esc in der IDE-Eingabezeile bricht ab. Jeder Programmierer baut gelegentlich eine Endlos-Schleife. Die guten merken es schneller.

5.4 Schleifen in Schleifen

Ein Spielfeld ist zweidimensional: Zeilen *und* Spalten. Die Lösung: eine Schleife *in* einer Schleife. Die äußere geht die Zeilen durch, die innere pro Zeile alle Spalten:

```
für y = 0 bis 2 {
    nimm zeile = ""
    für x = 0 bis 5 {
        zeile = "{zeile}({x},{y}) "
    }
    sag zeile
}
```

AUSGABE

```
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1)
(0,2) (1,2) (2,2) (3,2) (4,2) (5,2)
```

Genau so zeichnet Kistris später jedes Bild: außen `y` von 0 bis 19, innen `x` von 0 bis 9, und für jede belegte Zelle einen Block auf den Bildschirm.

Aufgaben

5.1 ★ Gib mit einer `für`-Schleife die Quadratzahlen von 1 bis 10 aus („1 mal 1 ist 1", …).

5.2 ★★ Die Kistris-Punktestaffel pro Level: Gib für Level 1 bis 5 jeweils aus, wie viel ein Tetris (800 × Level) bringt — mit einer Schleife, versteht sich.

5.3 ★★★ Male mit verschachtelten Schleifen ein Rechteck aus `▣`, 8 breit und 4 hoch. Zusatz: nur den *Rand* aus `▣`, innen `▢` (Tipp: `wenn x == 0 oder x == 7 oder y == 0 oder y == 3`).

Teste dich (Kapitel 5)

1. Wie oft läuft `für i = 0 bis BREITE - 1` bei `BREITE = 10`?
2. Wann prüft `solange` seine Bedingung?
3. Welche Schleife nimmst du für den Hard Drop („falle, bis etwas im Weg ist") — und warum?

6 Funktionen — Code mit Namen

In diesem Kapitel lernst du

- Funktionen definieren mit `funktion`
- Parameter übergeben und Ergebnisse zurückgeben mit `gib`
- das frühe `gib` — Funktionen, die abkürzen

6.1 Einmal schreiben, oft benutzen

Die Punkte-Staffel aus Kapitel 4 wirst du im Spiel immer wieder brauchen. Statt sie jedes Mal hinzuschreiben, gibst du ihr einen Namen — du machst eine **Funktion** daraus. Das hier ist schon der *echte* Kistris-Code (nur dass `level` im Spiel später eine globale Variable ist):

kap06_funktionen.ki

```
funktion punkte_für_reihen(n, level) {
    nimm basis = 100
    wenn n == 2 { basis = 300 }
    wenn n == 3 { basis = 500 }
    wenn n >= 4 { basis = 800 }
    gib basis * level
}

funktion fall_schwelle(level) {
    nimm s = 48 - level * 4
    wenn s < 3 { s = 3 }
    gib s
}

sag "1 Reihe, Level 1:  {punkte_für_reihen(1, 1)} Punkte"
sag "4 Reihen, Level 1: {punkte_für_reihen(4, 1)} Punkte"
sag "4 Reihen, Level 5: {punkte_für_reihen(4, 5)} Punkte"

für level = 1 bis 12 schritt 4 {
    sag "Level {level}: Schwelle {fall_schwelle(level)}"
}
```

AUSGABE

```
1 Reihe, Level 1:  100 Punkte
4 Reihen, Level 1: 800 Punkte
4 Reihen, Level 5: 4000 Punkte
Level 1: Schwelle 44
Level 5: Schwelle 28
Level 9: Schwelle 12
```

Die Bausteine:

- `funktion name(parameter…) { … }` definiert die Funktion. Die Parameter sind Platzhalter — Kisten, die beim Aufruf gefüllt werden.
- `gib wert` liefert das Ergebnis zurück und **beendet die Funktion sofort**.

- Aufgerufen wird mit `name(argumente…)` — auch mitten in einer Interpolation.

Schau dir `fall_schwelle` genau an: Sie ist das Herz des Schwierigkeitsgrads von Kistris. 48 Ticks pro Zeile auf Level 1 (gemütlich), 4 weniger pro Level, Untergrenze 3 (Panik). Eine Funktion, vier Zeilen — und das ganze Spielgefühl steckt darin.

6.2 Das frühe `gib`

Eine Funktion darf an mehreren Stellen `gib` sagen. Die erste, die erreicht wird, gewinnt. Das macht Prüf-Funktionen wunderbar lesbar — hier eine Vorschau auf eine der wichtigsten Funktionen von Kistris:

```
// Ist die Reihe y komplett gefüllt? (Vereinfachte Vorschau)
funktion reihe_voll(y) {
    für x = 0 bis BREITE - 1 {
        wenn feld_hole(x, y) == 0 { gib 0 }   // EINE leere Zelle? Sofort raus: nein!
    }
    gib 1                                     // Schleife überlebt? Dann: ja, voll.
}
```

Lies sie wie einen Türsteher: Er geht die Reihe ab, und beim ersten freien Platz sagt er „nicht voll" und dreht ab. Nur wenn er ganz durchkommt, meldet er „voll". Dieses Muster — *Schleife mit frühem* `gib` — kommt in Kistris fünfmal vor. Wer es einmal verstanden hat, liest den halben Spielcode fließend.

Build-Subset: `gib` immer mit Wert

Im nativen Build muss `gib` immer einen Wert zurückgeben — ein nacktes `gib` (ohne Wert, nur zum Abbrechen) gibt es dort nicht. Kistris strukturiert deshalb seine Handler mit `wenn modus == 1 { … }` statt mit „brich früh ab". Du wirst sehen: Das liest sich sogar besser.

Warum Funktionen die halbe Miete sind

Das fertige Kistris hat rund 40 Funktionen mit Namen wie `spawne`, `kollidiert`, `hard_drop`, `zeichne_titel`. Wer den Code liest, liest fast Prosa: „*wenn kollidiert … fixiere … spawne*". Gute Namen sind keine Kür — sie sind der Unterschied zwischen einem Programm und einem Rätsel.

Aufgaben

6.1 ★ Schreibe `funktion verdopple(x)`, die das Doppelte zurückgibt, und teste sie mit 5 und 21.

6.2 ★★ Schreibe `funktion maximum(a, b)`, die die größere der beiden Zahlen zurückgibt — mit frühem `gib`.

6.3 ★★ Kistris vergibt beim Hard Drop 2 Punkte pro Zeile. Schreibe `funktion drop_bonus(zeilen)` und kombiniere sie mit `punkte_für_reihen`: Was bringt ein 12-Zeilen-Drop mit anschließendem Tetris auf Level 3?

6.4 ★★★ Schreibe `funktion stein_name(t)`, die für 1–7 die Buchstaben I, O, T, S, Z, J, L zurückgibt (frühes `gib`, letzter Fall ohne `wenn`). Diese Funktion brauchst du in Kapitel 8 wirklich!

Teste dich (Kapitel 6)

1. Was passiert mit dem Rest der Funktion, wenn `gib` ausgeführt wird?
2. Was gibt `fall_schwelle(20)` zurück? (Rechne!)
3. Warum steht in `reihe_voll` das `gib 1` *hinter* der Schleife?

7 Listen — das Spielfeld entsteht

In diesem Kapitel lernst du

- Listen anlegen, lesen und mit dem `liste`-Modul verändern
- Module einbinden mit `nutze`
- den wichtigsten Trick des ganzen Buchs: das **flache Spielfeld**
- … und du löschst deine erste volle Reihe — in der Konsole!

7.1 Viele Werte, ein Name

Eine Liste ist eine Kiste mit Fächern. Du legst sie mit eckigen Klammern an und greifst mit einem **Index** auf die Fächer zu — **gezählt ab 0**:

```
nimm farben = ["cyan", "gelb", "violett"]
sag farben[0]      // cyan   – das ERSTE Element hat Index 0!
sag farben[2]      // violett
```

Die Null-Falle

Das erste Element heißt `[0]`, das letzte bei 10 Elementen `[9]`. Daran gewöhnt man sich nach etwa 30 Jahren. Bis dahin hilft die Eselsbrücke: Der Index sagt, *wie viele Schritte vom Anfang* du gehst — zum ersten Element gehst du null Schritte.

7.2 Das `liste`-Modul

Für alles, was über Anlegen und Lesen hinausgeht, bringt Kiste das Modul `liste` mit. Module sind Werkzeugkästen — du holst sie mit `nutze` ins Programm:

```
nutze liste

nimm beutel = [3, 1, 4]
sag liste.länge(beutel)          // 3 – wie viele Elemente?

liste.hänge_an(beutel, 7)        // hinten anhängen → [3, 1, 4, 7]
liste.stelle_voran(beutel, 9)    // vorn einfügen   → [9, 3, 1, 4, 7]
liste.entferne(beutel, 1)        // Index 1 löschen → [9, 1, 4, 7]
liste.füge_ein(beutel, 2, 5)     // an Index 2      → [9, 1, 5, 4, 7]

nimm feld = liste.gefüllt(24, 0) // 24 Nullen auf einen Schlag
```

Diese sechs — `länge`, `hänge_an`, `stelle_voran`, `entferne`, `füge_ein`, `gefüllt` — sind das komplette Listen-Vokabular von Kistris. Mehr braucht ein ganzes Tetris nicht. `gefüllt(n, startwert)` ist der Spezialist fürs Anlegen: *n* gleiche Werte in einer Zeile, und der native Compiler kann eine so gebaute Liste besonders schnell ansprechen.

7.3 Der große Trick: das flache Spielfeld

Jetzt kommt die wichtigste Idee des Buchs. Das Spielfeld ist ein Raster — 10 Spalten, 20 Zeilen. Der naive Gedanke: „eine Liste von Listen". Kistris macht es anders — und besser: **eine einzige flache Liste mit 200 Fächern**, und eine kleine Formel rechnet die Koordinate `(x, y)` in den Index um:

```
index = y * BREITE + x
```

Abbildung 7.1 — Die Index-Formel: Jede Zeile beginnt ein Vielfaches von BREITE weiter.

Build-Subset: Darum flach!

Verschachtelte Listen (`liste<liste<ganz>>`) kann der native Compiler inzwischen — trotzdem wählt Kistris bewusst die flache Liste. Sie ist kein Notnagel: Auch in C, Rust und Spiele-Engines werden 2-D-Raster genau so gespeichert, weil ein zusammenhängender Speicherblock schneller ist. Du lernst hier die Profi-Variante. Bitte beschwer dich nicht.

7.4 Das Übungs-Spielfeld — komplett in der Konsole

Jetzt bauen wir das alles zusammen — ein Mini-Spielfeld (6 × 4 zum Üben), mit allem, was das echte Kistris auch hat: aufbauen, lesen, schreiben, volle Reihe erkennen und löschen. Tipp das vollständig ab, es lohnt sich:

kap07_spielfeld.ki

```
nutze liste

fest BREITE = 6
fest HÖHE = 4

// Feld aufbauen: eine flache Liste, alle Zellen 0 (= leer)
nimm feld = liste.gefüllt(BREITE * HÖHE, 0)

// Lesen und Schreiben über die Index-Formel y * BREITE + x
funktion feld_hole(x, y) {
    gib feld[y * BREITE + x]
}

funktion feld_setze(x, y, w) {
    feld[y * BREITE + x] = w
```

```
}

// Eine Reihe ist voll, wenn keine Zelle 0 ist
funktion reihe_voll(y) {
    für x = 0 bis BREITE - 1 {
        wenn feld_hole(x, y) == 0 { gib 0 }
    }
    gib 1
}

// Das Feld als Text ausgeben - □ leer, ■ belegt
funktion zeige_feld() {
    für y = 0 bis HÖHE - 1 {
        nimm zeile = ""
        für x = 0 bis BREITE - 1 {
            wenn feld_hole(x, y) == 0 {
                zeile = "{zeile}□"
            } sonst {
                zeile = "{zeile}■"
            }
        }
        sag zeile
    }
}

// Ein paar Steine "fallen lassen" …
feld_setze(0, 3, 1)
feld_setze(1, 3, 1)
feld_setze(2, 3, 1)
feld_setze(3, 3, 1)
feld_setze(4, 3, 1)
feld_setze(5, 3, 1)
feld_setze(2, 2, 1)

zeige_feld()
sag "Reihe 3 voll? {reihe_voll(3)}"
sag "Reihe 2 voll? {reihe_voll(2)}"

// Volle Reihe löschen: 6 Zellen herausschneiden, 6 Nullen oben voranstellen
für k = 1 bis BREITE { liste.entferne(feld, 3 * BREITE) }
für k = 1 bis BREITE { liste.stelle_voran(feld, 0) }

sag "Nach dem Löschen:"
zeige_feld()
```

```
AUSGABE
□□□□□□
□□□□□□
□□■□□□
■■■■■■
Reihe 3 voll? 1
Reihe 2 voll? 0
Nach dem Löschen:
□□□□□□
□□□□□□
□□□□□□
□□■□□□
```

Schau dir die Ausgabe genau an: Die volle Reihe ist weg — und der einzelne Block darüber ist **eine Zeile nach unten gerutscht**. Genau das soll Tetris tun! Der Trick steckt in den zwei Schleifen am Ende:

1. `liste.entferne(feld, 3 * BREITE)` — sechsmal: Wir löschen sechsmal das Element am *Anfang von Reihe 3*. Beim Entfernen rückt alles nach — so verschwindet die ganze Reihe.
2. `liste.stelle_voran(feld, 0)` — sechsmal: Wir stopfen sechs leere Zellen *vorn* (also ganz oben) hinein. Dadurch rutscht alles andere um genau eine Zeile nach unten.

Kein Kopieren, keine Verschiebe-Schleifen — die Liste erledigt die Arbeit. Dieser Zwei-Zeilen-Trick ist im echten Kistris exakt derselbe, nur mit `BREITE = 10`.

Bau-Stopp

Dieses Programm muss bei dir laufen und exakt die Ausgabe oben liefern, bevor du weitermachst. Es ist die Generalprobe für das halbe Spiel.

Aufgaben

7.1 ★ Welcher Index gehört bei `BREITE = 10` zur Zelle (x=3, y=5)? Erst rechnen, dann mit `sag` prüfen.

7.2 ★★ Schreibe `funktion zähle_blöcke()`, die mit zwei Schleifen zählt, wie viele Zellen im Feld nicht 0 sind.

7.3 ★★ Fülle Reihe 2 *mit einer Schleife* komplett und lösche danach beide vollen Reihen nacheinander. Stimmt das Feld am Ende?

7.4 ★★★ Schreibe `funktion feld_leeren()`, die alle Zellen wieder auf 0 setzt (Tipp: eine Schleife über alle Indizes 0 bis `BREITE * HÖHE - 1` und in jedem Durchlauf `feld[i] = 0`).

Teste dich (Kapitel 7)

1. Warum ist das Spielfeld eine flache Liste und keine Liste von Listen?
2. Was berechnet `y * BREITE + x` — in einem Satz?
3. Warum lässt das Löschen einer Reihe per `entferne` + `stelle_voran` alles darüber automatisch nach unten rutschen?
4. Warum beginnt der Feld-Aufbau mit `nimm feld = [0]` statt `nimm feld = []`? (Vorgriff: Die Antwort steht in der Subset-Box von Kapitel 11 — rate erst mal.)

8 Zufall — der 7-Bag-Randomizer

In diesem Kapitel lernst du

- Zufallszahlen mit `zufall.ganz`
- warum „echter" Zufall sich unfair anfühlt
- den 7-Bag-Algorithmus — wie ihn auch das offizielle Tetris benutzt

8.1 Würfeln mit Kiste

```
nutze zufall

sag zufall.ganz(1, 6)     // ein Würfelwurf: 1 bis 6, beide einschließlich
sag zufall.ganz(0, 9)     // eine zufällige Spielfeld-Spalte
```

`zufall.ganz(von, bis)` liefert eine ganze Zahl zwischen den Grenzen — **beide einschließlich**. Mehr Zufall braucht ein Tetris nicht.

8.2 Das Problem mit dem reinen Zufall

Der naive Plan für den nächsten Stein wäre `zufall.ganz(1, 7)`. Funktioniert — fühlt sich aber gemein an: Echter Zufall liefert gern fünfmal hintereinander das S-Teil und lässt dich gleichzeitig zwanzig Steine auf das I-Teil warten, während dein schöner Schacht rechts immer tiefer wird. (Jeder Tetris-Spieler kennt diesen Schmerz. Er hat Studien inspiriert.)

Die Lösung des modernen Tetris heißt **7-Bag**: Stell dir eine Tüte vor, in der jeder der 7 Steine genau einmal liegt. Du ziehst blind, bis die Tüte leer ist — dann wird sie neu gefüllt. Ergebnis: Jeder Stein kommt in je 7 Zügen genau einmal. Zufällig, aber fair.

8.3 Der Beutel in Kiste

`kap08_beutel.ki`

```
nutze liste
nutze zufall

nimm beutel = [0]
liste.entferne(beutel, 0)        // leere, aber typisierte Liste

funktion fülle_beutel() {
    nimm kandidaten = [1, 2, 3, 4, 5, 6, 7]
    solange liste.länge(kandidaten) > 0 {
        nimm z = zufall.ganz(0, liste.länge(kandidaten) - 1)
        liste.hänge_an(beutel, kandidaten[z])
        liste.entferne(kandidaten, z)
    }
}
```

```
funktion ziehe_typ() {
    wenn liste.länge(beutel) == 0 { fülle_beutel() }
    nimm t = beutel[0]
    liste.entferne(beutel, 0)
    gib t
}

funktion stein_name(t) {
    wenn t == 1 { gib "I" }
    wenn t == 2 { gib "O" }
    wenn t == 3 { gib "T" }
    wenn t == 4 { gib "S" }
    wenn t == 5 { gib "Z" }
    wenn t == 6 { gib "J" }
    gib "L"
}

sag "14 Steine aus dem 7-Bag (zwei volle Tüten):"
nimm zeile = ""
für i = 1 bis 14 {
    nimm t = ziehe_typ()
    zeile = "{zeile}{stein_name(t)} "
    wenn i == 7 {
        sag zeile
        zeile = ""
    }
}
sag zeile
```

```
AUSGABE
14 Steine aus dem 7-Bag (zwei volle Tüten):
O L S J T I Z
T I J L O S Z
```

(Deine Reihenfolge wird anders aussehen — Zufall eben. Aber zähl nach: In jeder Siebener-Gruppe kommt jeder Buchstabe **genau einmal** vor.)

Der Mischalgorithmus in `fülle_beutel` ist hübsch unaufgeregt: Solange noch Kandidaten da sind, zieh einen *zufälligen* heraus und häng ihn an den Beutel. Kein Vertauschen, kein Spezialwissen — nur `zufall.ganz`, `hänge_an` und `entferne`. Und `ziehe_typ` ist der Service drumherum: Tüte leer? Auffüllen. Dann den vordersten Stein herausgeben.

Die leere, aber typisierte Liste

Die zwei Zeilen `nimm beutel = [0]` + `liste.entferne(beutel, 0)` sehen seltsam aus. Der Grund: Eine Liste, die als *komplett leeres* `[]` startet, kann der native Compiler später nicht zuordnen („Liste wovon denn?"). Wir legen sie deshalb mit einer Probezahl an und werfen die sofort wieder raus — übrig bleibt eine leere Liste, die genau weiß, dass sie Zahlen enthält. Ein klassisches Kiste-Rezept, das du noch öfter sehen wirst.

Aufgaben

8.1 ★ Simuliere 5 Würfelwürfe mit einer Schleife.

8.2 ★★ Ziehe 70 Steine aus dem Beutel und zähle mit, wie oft das I-Teil (Typ 1) kommt. Es müssen exakt 10 sein — egal, wie oft du es laufen lässt. Erkläre, warum.

8.3 ★★★ Baue eine Variante *ohne* Beutel (reiner `zufall.ganz(1, 7)`), ziehe 70 Steine und zähle wieder die I-Teile. Lass beide Varianten ein paarmal laufen und vergleiche. Spürst du den Unterschied? Das ist Spieldesign.

Teste dich (Kapitel 8)

1. Welche Werte kann `zufall.ganz(0, 6)` liefern — wie viele verschiedene sind das?
2. Was garantiert der 7-Bag, was reiner Zufall nicht garantiert?
3. Wann füllt `ziehe_typ` den Beutel neu?

Halbzeit in Teil I!

Atme kurz durch und schau zurück: Du beherrschst Variablen, Entscheidungen, Schleifen, Funktionen, Listen und Zufall — und du hast schon die echte Punkte-Staffel, das echte Spielfeld samt Reihen-Löschen und den echten Steine-Beutel von Kistris gebaut. Was jetzt noch fehlt, ist im Grunde nur: *ein Bildschirm*. Den holen wir uns im nächsten Kapitel.

9 Dein erstes Fenster — die Leinwand

In diesem Kapitel lernst du

- das `gui`-Modul: Fenster, Leinwand, Formen, Text
- das Koordinatensystem des Bildschirms
- Farben als Hex-Codes
- … und du zeichnest deinen ersten Tetris-Block

Ab jetzt: nativ bauen!

Die grafische Oberfläche von Kiste läuft **nur im nativ gebauten Programm** — `kiste run` kann keine Fenster öffnen. Ab diesem Kapitel heißt der Arbeitsablauf also: speichern → `kiste build datei.ki` → die erzeugte `.exe` starten. In der Kiste-IDE bleibt es bei einem einzigen Tastendruck: **F5** erledigt beides. Falls beim allerersten Build eine Meldung über fehlende Werkzeuge kommt: Kapitel 2.1 hat die Installations-Befehle.

9.1 Fenster + Leinwand = Spiel-Bildschirm

Drei Dinge braucht jedes grafische Kiste-Programm: eine **Leinwand** (die Fläche, auf die du zeichnest), ein **Fenster** (der Rahmen drumherum) und den Startbefehl. Das Minimal-Gerüst:

```
nutze gui

nimm bild = gui.leinwand(400, 300)          // Zeichenfläche: 400 × 300 Pixel
gui.rechteck(bild, 0, 0, 400, 300, "#0f172a")

nimm fenster = gui.fenster_öffne("Mein Fenster", 401, 301)
gui.zeige(fenster, bild)                    // Leinwand ins Fenster legen
gui.starte()                                // Übergabe an die Oberfläche
```

`gui.starte()` ist immer die **letzte** Zeile: Ab hier übernimmt das Fenster die Kontrolle, bis der Benutzer es schließt.

9.2 Das Koordinatensystem — y wächst nach unten!

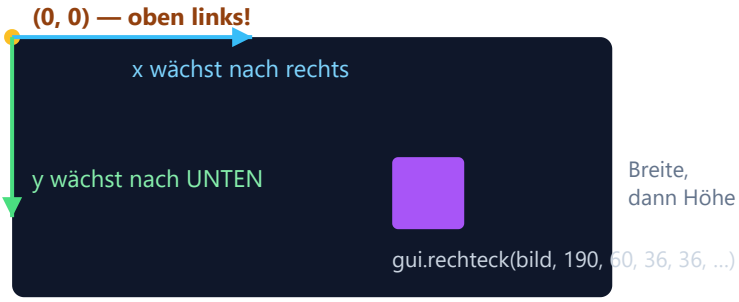


Abbildung 9.1 — Der Ursprung liegt oben links, y wächst nach unten. Für fallende Tetris-Steine ist das ein Geschenk: Fallen heißt einfach `y = y + 1`.

Die wichtigsten Zeichenbefehle — alle wollen zuerst die Leinwand:

Befehl	Zeichnet
`gui.rechteck(bild, x, y, breite, höhe, farbe)`	gefülltes Rechteck — unser Brot-und-Butter-Befehl
`gui.kreis(bild, x, y, radius, farbe)`	gefüllter Kreis (x, y = Mittelpunkt)
`gui.linie(bild, x1, y1, x2, y2, farbe, dicke)`	Linie — für das Spielfeld-Gitter
`gui.text_an(bild, x, y, text, farbe, größe)`	Text an Position (x, y = obere linke Ecke)
`gui.leeren(bild)`	wischt die Leinwand leer — Anfang jedes neuen Bilds

9.3 Farben — das Hex-Format

Farben schreibst du als **Hex-Code**: `"#rrggbb"` — je zwei Stellen für Rot, Grün, Blau, von `00` (nichts) bis `ff` (volle Kraft). `"#ff0000"` ist reines Rot, `"#0f172a"` das dunkle Nachtblau, das in allen Kiste-Demos den Hintergrund macht. Mit **acht** Stellen (`"#rrggbbaa"`) bekommt die Farbe zusätzlich eine Durchsichtigkeit — das hebt sich Kistris für das Ghost Piece und die Pause-Überblendung auf. Du musst Hex-Codes nicht „können": Jeder Online-Farbwähler spuckt sie aus.

9.4 Dein erster Block — und gleich ein ganzer T-Stein

kap09_fenster.ki

```
nutze gui

nimm bild = gui.leinwand(400, 300)

// Ein Tetris-Block: heller Rahmen + satte Füllung
funktion zeichne_block(xp, yp, gr, farbe, hell) {
    gui.rechteck(bild, xp, yp, gr - 2, gr - 2, hell)
    gui.rechteck(bild, xp + 2, yp + 2, gr - 6, gr - 6, farbe)
}

gui.rechteck(bild, 0, 0, 400, 300, "#0f172a")          // dunkler Hintergrund
gui.text_an(bild, 130, 30, "Mein erster Block!", "#e4e4e7", 18)
```

```
zeichne_block(100, 100, 30, "#a855f7", "#d8b4fe")      // violett (T-Stein)
zeichne_block(130, 100, 30, "#a855f7", "#d8b4fe")
zeichne_block(160, 100, 30, "#a855f7", "#d8b4fe")
zeichne_block(130, 70, 30, "#a855f7", "#d8b4fe")       // … ein ganzer T-Stein!

gui.kreis(bild, 320, 220, 25, "#fbbf24")               // und ein Kreis dazu
gui.linie(bild, 20, 270, 380, 270, "#334155", 2)       // … und eine Linie

gui.thema("dunkel")
nimm fenster = gui.fenster_öffne("Kiste – mein erstes Fenster", 401, 301)
gui.zeige(fenster, bild)
gui.starte()
```

Der Trick in `zeichne_block` ist simpel und wirkungsvoll: *zwei* Rechtecke. Erst ein helles in voller Blockgröße, dann das satte zwei Pixel kleiner obendrauf — übrig bleibt ein heller Rand, der den Block plastisch macht. Exakt diese Funktion zeichnet im fertigen Kistris jeden einzelnen Stein. Du hast sie gerade geschrieben.

Bau-Stopp

`kiste build kap09_fenster.ki` (oder F5) — du musst ein Fenster mit violettem T-Stein, gelbem Kreis und einer Trennlinie sehen. Das ist dein erstes gebautes Grafikprogramm: eine echte `.exe`, die du jedem schicken könntest.

Aufgaben

9.1 ★ Ändere die Farben: Mach aus dem T-Stein einen grünen S-Stein-farbenen Klotz (`#4ade80` / `#bbf7d0`).

9.2 ★★ Zeichne mit `zeichne_block` und einer `für`-Schleife eine komplette Reihe aus 10 Blöcken am unteren Fensterrand.

9.3 ★★★ Zeichne einen O-Stein (2×2), einen I-Stein (4 nebeneinander) und einen L-Stein. Tipp: Skizziere die Zell-Koordinaten erst auf Papier — genau so entsteht in Kapitel 11 die Formen-Tabelle.

Teste dich (Kapitel 9)

1. Wo liegt der Punkt (0, 0) auf der Leinwand?
2. Warum ist „y wächst nach unten" für Tetris praktisch?
3. Was bedeutet die Farbangabe `"#ff000080"`?
4. Warum zeichnet `zeichne_block` zwei Rechtecke?

10 Der Game-Loop — Zeit und Tastatur

In diesem Kapitel lernst du

- den Timer `gui.bei_zeit` — den Herzschlag jedes Spiels
- Handler: Funktionen, die „später" laufen
- die zwei Tastatur-Arten: `bei_taste` (diskret) und `taste_gedrückt` (gehalten)
- den Schwerkraft-Akkumulator — wie Kistris sein Tempo steuert

10.1 Spiele sind Filme

Ein Spiel ist ein Film, den dein Programm in Echtzeit malt: viele Bilder pro Sekunde, und zwischen zwei Bildern bewegt sich die Welt ein kleines Stück. Den Takt gibt `gui.bei_zeit` vor:

```
gui.bei_zeit(16, funktion(t) {
    // dieser Block läuft alle 16 Millisekunden – etwa 60-mal pro Sekunde
})
```

Hier passieren zwei neue Dinge auf einmal. Erstens: `bei_zeit` bekommt eine **Funktion als Argument** — ohne Namen, direkt hingeschrieben (`funktion(t) { … }`). Man nennt so etwas einen **Handler**: „Lieber Timer, *wenn* es so weit ist, führe bitte das hier aus." Zweitens: Der Handler läuft nicht jetzt, sondern immer wieder später — er ist der Puls deines Spiels.

Warum 16 Millisekunden?

1000 ms ÷ 60 Bilder ≈ 16,7 ms — der Takt, mit dem auch dein Monitor arbeitet. Ein Tick alle 16 ms ergibt flüssige 60 Bilder pro Sekunde. Kistris zählt einfach Ticks: „Alle 48 Ticks fällt der Stein eine Zeile" heißt: ungefähr alle 0,77 Sekunden. Level rauf → weniger Ticks → schneller. Erinnerst du dich an `fall_schwelle` aus Kapitel 6? Genau dafür war die da.

10.2 Tastatur, Art 1: gehaltene Tasten

Für Bewegung („solange ← gedrückt ist, geh links") fragt der Game-Loop in jedem Tick nach:

```
wenn gui.taste_gedrückt(fenster, "links") == 1 {
    bx = bx - 1
}
```

`taste_gedrückt` liefert `1`, solange die Taste unten ist. Die Tastennamen sind deutsch: `"links"`, `"rechts"`, `"hoch"`, `"runter"`, `"leer"`, `"escape"`, `"eingabe"` — Buchstaben groß: `"P"`.

10.3 Tastatur, Art 2: diskrete Tastendrücke

Für Aktionen, die *pro Druck genau einmal* passieren sollen (Rotation! Hard Drop!), gibt es `gui.bei_taste` — wieder mit einem Handler:

```
gui.bei_taste(fenster, funktion(f) {
    nimm taste = gui.letzte_taste(f)
    wenn taste == "leer" {
        sag "Leertaste!"
    }
})
```

Das Handle-und-Getter-Muster

Der Handler bekommt *nicht* die Taste selbst, sondern das Fenster-Handle `f` — und holt sich die Taste mit `gui.letzte_taste(f)` ab. Alle Kiste-GUI-Handler arbeiten so: erst das Handle, dann der Getter. Wer versucht, die Taste direkt als Parameter zu empfangen, bekommt Kauderwelsch.

10.4 Alles zusammen: der fallende Block

Jetzt baust du das erste „richtige" Stück Kistris: ein Block fällt, du steuerst ihn, und ein Zähler steuert das Tempo. Komplett abtippen — das ist die Blaupause für das ganze Spiel:

kap10_steuern.ki

```
nutze gui

fest ZELLE = 30
fest SPALTEN = 10
fest ZEILEN = 15

nimm bild = gui.leinwand(SPALTEN * ZELLE, ZEILEN * ZELLE)
nimm bx = 4                      // Block-Position in Zellen
nimm by = 0
nimm fall_zaehler = 0
nimm punkte = 0

funktion zeichne() {
    gui.leeren(bild)
    gui.rechteck(bild, 0, 0, SPALTEN * ZELLE, ZEILEN * ZELLE, "#0f172a")
    gui.rechteck(bild, bx * ZELLE, by * ZELLE, ZELLE - 2, ZELLE - 2, "#a5f3fc")
    gui.rechteck(bild, bx * ZELLE + 2, by * ZELLE + 2, ZELLE - 6, ZELLE - 6, "#22d3ee")
    gui.text_an(bild, 8, 6, "Gefangen: {punkte}", "#e4e4e7", 16)
}

zeichne()
gui.thema("dunkel")
nimm fenster = gui.fenster_öffne("Kiste – fang den Block", SPALTEN * ZELLE + 1, ZEILEN * ZELLE + 1)

// Diskrete Taste: Leertaste teleportiert den Block nach oben ("gefangen")
gui.bei_taste(fenster, funktion(f) {
    nimm taste = gui.letzte_taste(f)
    wenn taste == "leer" und by >= ZEILEN - 3 {
        punkte = punkte + 1
        by = 0
    }
})

// Der Game-Loop: 16 ms pro Tick. Gehaltene Tasten + Schwerkraft-Akkumulator.
gui.bei_zeit(16, funktion(t) {
```

```
    wenn gui.taste_gedrückt(fenster, "links") == 1 und bx > 0 {
        bx = bx - 1
    }
    wenn gui.taste_gedrückt(fenster, "rechts") == 1 und bx < SPALTEN - 1 {
        bx = bx + 1
    }
    fall_zaehler = fall_zaehler + 1
    wenn fall_zaehler >= 30 {            // alle 30 Ticks eine Zeile fallen
        fall_zaehler = 0
        by = by + 1
        wenn by >= ZEILEN { by = 0 }     // unten raus? Oben wieder rein.
    }
    zeichne()
})

gui.zeige(fenster, bild)
gui.starte()
```

Das Spiel: Fang den fallenden Block mit der Leertaste, kurz bevor er unten ankommt (in den letzten drei Zeilen). Steuere ihn mit ← und →. Es ist zugegeben kein preisverdächtiges Game-Design — aber schau, was du da alles benutzt:

- **Zell-Koordinaten:** Der Block lebt in Spalte `bx`, Zeile `by` — gezeichnet wird bei `bx * ZELLE` Pixeln. Raster-Denken wie im Spielfeld!
- **Der Schwerkraft-Akkumulator:** Der Loop tickt 60-mal pro Sekunde, aber der Block fällt nur, wenn `fall_zaehler` die Schwelle 30 erreicht. Tempo ändern = Schwelle ändern. Genau so macht es Kistris — nur dass dort `fall_schwelle()` die Schwelle pro Level liefert.
- **Beide Tastatur-Arten parallel:** Bewegung per Polling im Tick, die „Fang"-Aktion diskret per `bei_taste`. Diese Arbeitsteilung übernimmt Kistris eins zu eins: Bewegen/Soft-Drop gehalten, Rotation/Hard-Drop diskret.
- **Zeichnen am Tick-Ende:** Erst denken (bewegen, prüfen), dann malen. Immer.

Bau-Stopp

Bauen, spielen, mindestens 3 Punkte holen. Wenn der Block ruckelfrei fällt und sofort auf ← / → reagiert, hast du soeben einen Game-Loop gemeistert — das schwierigste Konzept des ganzen Buchs. Ab hier wird nichts mehr grundlegend Neues kommen, nur mehr davon.

Aufgaben

10.1 ★ Mach das Spiel schneller: Schwelle 15 statt 30. Und jetzt unspielbar: 3. (Wieder zurückdrehen.)

10.2 ★★ Soft Drop: Solange `"runter"` gehalten wird, soll `fall_zaehler` pro Tick um zusätzliche 8 steigen. Eine einzige `wenn`-Zeile im Loop — und exakt die Kistris-Lösung.

10.3 ★★ Zähle Fehlversuche: Erreicht der Block unten die letzte Zeile, ohne gefangen zu werden, ziehe einen Punkt ab (nicht unter 0!).

10.4 ★★★ Jeder Fang macht das Spiel schneller: Senke die Schwelle pro Punkt um 2, Untergrenze 5. Tipp: Mach aus der festen 30 eine Variable — oder gleich eine Funktion wie `fall_schwelle` aus Kapitel 6.

Teste dich (Kapitel 10)

1. Wie oft läuft ein `bei_zeit(16, …)`-Handler pro Sekunde — ungefähr?
2. Rotation soll pro Tastendruck genau einmal passieren. Welcher Mechanismus — und warum nicht der andere?
3. Was macht der Akkumulator `fall_zaehler`, in einem Satz?
4. Wie kommt der `bei_taste`-Handler an die gedrückte Taste?

11 Die sieben Steine

In diesem Kapitel

- startest du das echte Projekt: `kistris.ki`
- lernst du die sieben Tetrominoes und ihre Rotations-Tabelle kennen
- baust du `form_wert` und die Farb-Funktionen
- verstehst du die `wert.als_ganz`-Brücke des nativen Builds

11.1 Projektstart

Leg einen Ordner `kistris` an und darin die Datei `kistris.ki`. Ab jetzt wächst diese eine Datei Kapitel für Kapitel zum fertigen Spiel — am Ende sind es gut 700 Zeilen, und du wirst jede davon verstehen. Wir beginnen mit dem Kopf: Module und die Konstanten, die Spielfeld und Bildschirm vermessen.

`kistris.ki — Anfang`

```
// kistris.ki — KISTRIS: Tetris in Kiste.
// modus: 0 = Hauptmenü · 1 = Spielt · 2 = Pause · 3 = Game-Over
nutze gui
nutze klang
nutze liste
nutze zufall
nutze wert
nutze datei

fest BREITE = 10
fest HÖHE = 20
fest ZELLE = 30
fest FELD_X = 20                // linke obere Ecke des Spielfelds (Pixel)
fest FELD_Y = 40
fest BILD_B = 580               // Leinwand-Größe
fest BILD_H = 680
fest PANEL_X = 350              // Info-Spalte rechts neben dem Feld
```

Sechs Module auf einen Schlag — alle kennst du schon oder lernst sie in den nächsten Kapiteln: `gui` (Fenster), `klang` (Sound, Kapitel 16), `liste` und `zufall` (Teil I), `datei` (Rangliste, Kapitel 15) und `wert` — der Neuling, den die Box unten erklärt. Die Konstanten trennen *Spielfeld-Logik* (BREITE, HÖHE — in Zellen) von *Bildschirm-Layout* (alles andere — in Pixeln). Eine Zelle ist 30 Pixel groß, das Feld beginnt bei (20, 40), rechts daneben ab x=350 wohnt die Info-Spalte mit Punkten und Vorschau.

11.2 Sieben Steine, ein Format

Jeder Tetris-Stein („Tetromino") besteht aus genau 4 Blöcken. Wir beschreiben ihn relativ zu einem **Drehpunkt**: 4 Paare aus (x, y)-Versätzen. Und weil sich Steine drehen, speichern wir gleich **alle vier Rotationen** — fertig vorgerechnet, als Tabelle. Keine Drehmathematik zur Laufzeit, keine Überraschungen. So sehen die sieben aus:

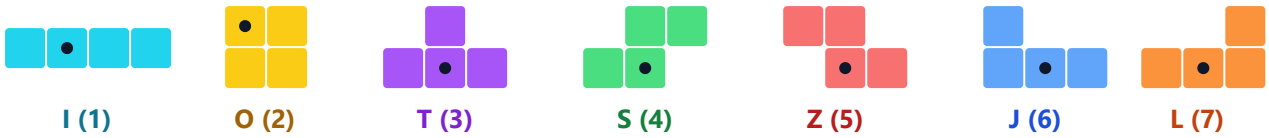


Abbildung 11.1 — Die sieben Tetrominoes in Grundstellung (Rotation 0). Der schwarze Punkt markiert jeweils den Drehpunkt (0, 0); die Typ-Nummer 1–7 ist zugleich der Farbindex im Spielfeld.

Pro Stein eine flache Liste — 4 Rotationen × 4 Blöcke × 2 Zahlen = **32 Werte**. Schau dir zuerst nur den T-Stein an und vergleiche mit der Abbildung: Rotation 0 ist `0,-1` (die Spitze, eine Zeile *über* dem Drehpunkt — y ist ja nach unten!), dann `-1,0`, `0,0`, `1,0` (der Balken):

kistris.ki — die Formen-Tabelle

```
// Pro Steintyp eine flache Liste: 4 Rotationen × 4 Blöcke × (x, y) = 32 Werte,
// relativ zum Drehpunkt. Rotation im Uhrzeigersinn: (x, y) → (−y, x).
nimm FORM_I = [-1,0, 0,0, 1,0, 2,0,   0,-1, 0,0, 0,1, 0,2,
               -1,0, 0,0, 1,0, 2,0,   0,-1, 0,0, 0,1, 0,2]
nimm FORM_O = [0,0, 1,0, 0,1, 1,1,    0,0, 1,0, 0,1, 1,1,
               0,0, 1,0, 0,1, 1,1,    0,0, 1,0, 0,1, 1,1]
nimm FORM_T = [0,-1, -1,0, 0,0, 1,0,  0,-1, 0,0, 1,0, 0,1,
               -1,0, 0,0, 1,0, 0,1,   0,-1, -1,0, 0,0, 0,1]
nimm FORM_S = [0,-1, 1,-1, -1,0, 0,0, 0,-1, 0,0, 1,0, 1,1,
               0,-1, 1,-1, -1,0, 0,0, 0,-1, 0,0, 1,0, 1,1]
nimm FORM_Z = [-1,-1, 0,-1, 0,0, 1,0, 1,-1, 0,0, 1,0, 0,1,
               -1,-1, 0,-1, 0,0, 1,0, 1,-1, 0,0, 1,0, 0,1]
nimm FORM_J = [-1,-1, -1,0, 0,0, 1,0, 1,-1, 0,-1, 0,0, 0,1,
               -1,0, 0,0, 1,0, 1,1,   0,-1, 0,0, 0,1, -1,1]
nimm FORM_L = [1,-1, -1,0, 0,0, 1,0,  0,-1, 0,0, 0,1, 1,1,
               -1,0, 0,0, 1,0, -1,1,  -1,-1, 0,-1, 0,0, 0,1]
```

Woher kommen die Zahlen?
Aus einer einzigen Formel: Eine Drehung im Uhrzeigersinn macht aus jedem Versatz (x, y) den Versatz **(−y, x)**. T-Spitze (0,−1) → gedreht (1,0): Die Spitze zeigt nach rechts. ✓ So wurde Rotation für Rotation vorgerechnet und in die Tabelle geschrieben. Beachte die Faulheits-Tricks: Der O-Stein ist ein Quadrat — alle 4 Rotationen identisch. I, S und Z sehen nach einer halben Drehung wieder gleich aus — Rotation 2 und 3 wiederholen einfach 0 und 1. Nur T, J und L haben vier echte Stellungen. Du musst die Tabelle übrigens nicht nachrechnen — abtippen genügt. Aber prüfe *einen* Stein gegen Abbildung 11.1, damit du dem Format traust.

11.3 Der Nachschlage-Dienst: `form_wert`

Damit der Spielcode nie wieder über 32er-Listen nachdenken muss, kapselt eine Funktion das Nachschlagen. `t` ist der Steintyp (1–7), `r` die Rotation (0–3), `k` die Position innerhalb der Rotation (0–7: vier x,y-Paare):

```
// Einen Tabellen-Wert lesen: k läuft 0..7 (4 Paare x,y) innerhalb der Rotation.
funktion form_wert(t, r, k) {
    nimm idx = r * 8 + k
    wenn t == 1 { gib FORM_I[idx] }
    wenn t == 2 { gib FORM_O[idx] }
    wenn t == 3 { gib FORM_T[idx] }
    wenn t == 4 { gib FORM_S[idx] }
    wenn t == 5 { gib FORM_Z[idx] }
    wenn t == 6 { gib FORM_J[idx] }
    gib FORM_L[idx]
}
```

Wieder die Index-Formel-Idee aus Kapitel 7, nur eine Nummer kleiner: Jede Rotation belegt 8 Plätze, also beginnt Rotation `r` bei `r * 8`. Und wieder das frühe `gib` aus Kapitel 6 — sechs Prüfungen, und wer übrig bleibt, ist der L-Stein. Die x-Koordinate von Block `k` ist `form_wert(t, r, k * 2)`, die y-Koordinate `form_wert(t, r, k * 2 + 1)`.

11.4 Farben — zwei Schwester-Funktionen

```
funktion farbe_von(t) {
    wenn t == 1 { gib "#22d3ee" }
    wenn t == 2 { gib "#facc15" }
    wenn t == 3 { gib "#a855f7" }
    wenn t == 4 { gib "#4ade80" }
    wenn t == 5 { gib "#f87171" }
    wenn t == 6 { gib "#60a5fa" }
    gib "#fb923c"
}

funktion farbe_hell_von(t) {
    wenn t == 1 { gib "#a5f3fc" }
    wenn t == 2 { gib "#fef08a" }
    wenn t == 3 { gib "#d8b4fe" }
    wenn t == 4 { gib "#bbf7d0" }
    wenn t == 5 { gib "#fecaca" }
    wenn t == 6 { gib "#bfdbfe" }
    gib "#fed7aa"
}
```

Pro Typ eine satte Füllfarbe und eine helle Randfarbe — du erinnerst dich an den Zwei-Rechtecke-Trick aus Kapitel 9. Die klassische Zuordnung: I cyan, O gelb, T violett, S grün, Z rot, J blau, L orange. Seit 40 Jahren weltweit gleich — daran rüttelt man nicht.

11.5 Zustand, Spielfeld — und die `wert`-Brücke

Jetzt kommen die globalen Variablen des Spiels und das Spielfeld — Letzteres kennst du wortwörtlich aus Kapitel 7, nur mit einer kleinen, wichtigen Änderung:

```
nimm modus = 0
nimm bild = gui.leinwand(BILD_B, BILD_H)

// Spielfeld: flache Liste, 200 Zellen, 0 = leer, 1–7 = Farbindex.
```

```
nimm feld = [0]
für i = 2 bis BREITE * HÖHE {
    liste.hänge_an(feld, 0)
}

// Aktueller Stein
nimm typ = 1
nimm rot = 0
nimm px = 4                      // Position des Drehpunkts im Feld
nimm py = 1
nimm naechster_typ = 1

// Zellwert lesen – wert.als_ganz, weil mutierte Listen geboxte Werte tragen.
funktion feld_hole(x, y) {
    gib wert.als_ganz(feld[y * BREITE + x])
}

// Zellwert schreiben – direkte Index-Zuweisung, in-place.
funktion feld_setze(x, y, w) {
    feld[y * BREITE + x] = w
}

funktion feld_leeren() {
    für i = 0 bis BREITE * HÖHE - 1 {
        feld[i] = 0
    }
}
```

Build-Subset: drei Rezepte auf einen Blick

1. `wert.als_ganz(…)`**:** Im nativen Build speichern Listen, die wachsen und schrumpfen, ihre Werte „verpackt" (geboxt). Beim Herauslesen packt `wert.als_ganz` die Zahl wieder aus. Im `kiste run`-Modus war das unnötig (und das Modul `wert` existiert dort gar nicht!) — nativ ist es Pflicht. Deshalb kapseln wir jeden Feld-Zugriff in `feld_hole`: *eine* Stelle, an der die Brücke wohnt, statt fünfzig.

2. `feld[idx] = w` **schreibt direkt:** Die Index-Zuweisung auf Listen ist eine native In-Place-Operation (seit 2026-06-12 — ältere Kistris-Versionen mussten hier mit `entferne` + `füge_ein` tricksen). Auch `feld[idx] += 1` und Co. funktionieren.

3. `[0]` **+ Aufbau-Schleife:** die „leere, aber typisierte Liste" aus Kapitel 8 — ein leeres `[]` kann der Build nicht zuordnen.

11.6 Zeichnen und erster Build

Zum Abschluss des Kapitels bringen wir einen Stein auf den Bildschirm — mit dem Block-Zeichner aus Kapitel 9 (jetzt mit den Farb-Funktionen verheiratet) und einem Rahmen für das Feld:

```
// Ein Block: heller Rahmen + satte Füllung – flacher Look mit Kontur.
funktion zeichne_block(xp, yp, gr, t) {
    gui.rechteck(bild, xp, yp, gr - 2, gr - 2, farbe_hell_von(t))
    gui.rechteck(bild, xp + 2, yp + 2, gr - 6, gr - 6, farbe_von(t))
}

funktion zeichne_feld_rahmen() {
```

```
    gui.rechteck(bild, FELD_X - 4, FELD_Y - 4, BREITE * ZELLE + 8, HÖHE * ZELLE + 8, "#334155")
    gui.rechteck(bild, FELD_X - 2, FELD_Y - 2, BREITE * ZELLE + 4, HÖHE * ZELLE + 4, "#0b1120")
}

funktion zeichne_stein() {
    für k = 0 bis 3 {
        nimm x = px + form_wert(typ, rot, k * 2)
        nimm y = py + form_wert(typ, rot, k * 2 + 1)
        wenn y >= 0 {
            zeichne_block(FELD_X + x * ZELLE + 1, FELD_Y + y * ZELLE + 1, ZELLE, typ)
        }
    }
}

funktion zeichne() {
    gui.leeren(bild)
    gui.rechteck(bild, 0, 0, BILD_B, BILD_H, "#0f172a")
    zeichne_feld_rahmen()
    zeichne_stein()
}

zeichne()
gui.thema("dunkel")
nimm fenster = gui.fenster_öffne("Kistris", BILD_B + 1, BILD_H + 1)
gui.zeige(fenster, bild)
gui.starte()
```

`zeichne_stein` ist die Formen-Tabelle in Aktion: Für jeden der 4 Blöcke holt sie den Versatz, addiert die Stein-Position `(px, py)` dazu und rechnet die Zelle in Pixel um — Zelle mal `ZELLE` plus Feld-Ecke. (Das `wenn y >= 0` ignoriert Blöcke oberhalb des Felds — frisch gespawnte Steine ragen kurz hinaus.)

Bau-Stopp

Nativ bauen! Du musst das dunkle Fenster mit Feldrahmen sehen und oben einen cyanfarbenen I-Stein (Typ 1, Rotation 0, an Position 4/1). Setze testweise `nimm typ = 3` und `nimm rot = 1` — jetzt muss ein violettes T mit Spitze nach rechts dastehen. Wenn ja: Deine Tabelle stimmt. Zurückstellen auf Typ 1, weiter!

Aufgaben

11.1 ★ Probiere alle 7 Typen und alle 4 Rotationen des L-Steins durch (Variablen ändern, neu bauen). Vergleiche mit Abbildung 11.1.

11.2 ★★ Berechne von Hand: Welche 4 Feld-Zellen belegt ein T-Stein (Typ 3) in Rotation 2 an Position px=5, py=10? Dann im Programm nachschauen.

11.3 ★★★ Schreibe (auf Papier!) die Tabellen-Zeile für einen erfundenen „Punkt-Stein" aus nur 1 Block — was wäre seine 32-Werte-Liste? (Ja, viermal dasselbe. Genau wie beim O.)

Teste dich (Kapitel 11)

1. Wie viele Zahlen speichert die Tabelle pro Stein — und wie setzt sich die Zahl zusammen?
2. Was macht die Dreh-Formel (x, y) → (−y, x) anschaulich?
3. Warum kapseln wir Feld-Zugriffe in `feld_hole` / `feld_setze`, statt überall die Index-Formel auszuschreiben?
4. Wozu dient `wert.als_ganz` — und warum brauchte Kapitel 7 es noch nicht?

12 Schwerkraft, Kollision und Rotation

In diesem Kapitel

- schreibst du `kollidiert` — die wichtigste Funktion des Spiels
- lässt du den Stein fallen und steuerst ihn mit DAS-Wiederholung
- baust du Rotation mit Wall-Kick

12.1 Eine Frage entscheidet alles: `kollidiert`

Darf der Stein hierhin? Diese eine Frage steckt hinter *allem*: Bewegen, Fallen, Drehen, Spawnen, Game Over. Deshalb bauen wir sie als eine Funktion, die einen *hypothetischen* Zustand prüft — Typ `t`, Rotation `r`, Position `(bx, by)` — ohne den echten Stein anzufassen:

```
// Kollidiert Stein t in Rotation r an Basisposition (bx, by)?
// Oberhalb des Felds (y < 0) ist frei – dort darf rotiert/gespawnt werden.
funktion kollidiert(t, r, bx, by) {
    für k = 0 bis 3 {
        nimm x = bx + form_wert(t, r, k * 2)
        nimm y = by + form_wert(t, r, k * 2 + 1)
        wenn x < 0 oder x >= BREITE { gib 1 }
        wenn y >= HÖHE { gib 1 }
        wenn y >= 0 {
            wenn feld_hole(x, y) != 0 { gib 1 }
        }
    }
    gib 0
}
```

Für jeden der 4 Blöcke drei Prüfungen: seitlich raus? unten raus? Zelle schon belegt? Beim ersten Treffer: `gib 1` — kollidiert. Überleben alle vier: `gib 0` — Platz frei. (Das Türsteher-Muster aus Kapitel 6, schon wieder.) Und weil die Funktion nur *prüft*, kannst du fragen „dürfte ich einen Schritt nach links?", bevor du ihn machst:

```
funktion versuche_bewege(dx) {
    wenn kollidiert(typ, rot, px + dx, py) == 0 {
        px = px + dx
        neu_zeichnen = 1
    }
}
```

Falls erlaubt: Position ändern und `neu_zeichnen = 1` setzen — ein neues globales Flag (lege oben `nimm neu_zeichnen = 0` und `nimm fall_zaehler = 0`, `nimm links_seit = 0`, `nimm rechts_seit = 0` an). Die Idee: Funktionen melden nur „hier hat sich was geändert" — gezeichnet wird einmal zentral am Tick-Ende. So malt das Spiel nicht 60-mal pro Sekunde umsonst, sondern nur, wenn es etwas Neues gibt.

12.2 Fallen — vorerst ohne Landung

```
// Eine Zeile fallen; geht es nicht, … (die Landung baut Kapitel 13)
funktion versuche_fallen() {
    wenn kollidiert(typ, rot, px, py + 1) == 0 {
        py = py + 1
        neu_zeichnen = 1
    }
}

// Tempo pro Level: Ticks (à 16 ms) pro Fall-Zeile – kleiner = schneller.
funktion fall_schwelle() {
    nimm s = 48 - level * 4
    wenn s < 3 { s = 3 }
    gib s
}
```

(`fall_schwelle` kennst du aus Kapitel 6 — im Spiel liest sie das globale `level`; lege dafür schon mal `nimm level = 1` an, zusammen mit `nimm punkte = 0` und `nimm reihen_gesamt = 0` für Kapitel 13.)

12.3 Der Game-Loop — mit Profi-Steuerung (DAS)

Jetzt der Loop. Eine Feinheit macht den Unterschied zwischen „fühlt sich billig an" und „fühlt sich nach Tetris an": Wenn du ← *hältst*, soll der Stein *einmal sofort* ziehen, dann kurz warten, dann schnell wiederholen — wie eine Schreibmaschinen-Taste. Die Spielebranche nennt das DAS (Delayed Auto Shift). Wir zählen einfach, wie lange die Taste schon gehalten wird:

```
gui.bei_zeit(16, funktion(t) {
    wenn modus == 1 {
        // Links/Rechts: erster Druck sofort, nach 12 Ticks Auto-Wiederholung alle 3.
        wenn gui.taste_gedrückt(fenster, "links") == 1 {
            links_seit = links_seit + 1
            wenn links_seit == 1 { versuche_bewege(-1) }
            wenn links_seit >= 12 {
                versuche_bewege(-1)
                links_seit = 9
            }
        } sonst {
            links_seit = 0
        }
        wenn gui.taste_gedrückt(fenster, "rechts") == 1 {
            rechts_seit = rechts_seit + 1
            wenn rechts_seit == 1 { versuche_bewege(1) }
            wenn rechts_seit >= 12 {
                versuche_bewege(1)
                rechts_seit = 9
            }
        } sonst {
            rechts_seit = 0
        }
        // Soft Drop: Schwerkraft kräftig beschleunigen, solange "runter" gehalten.
        wenn gui.taste_gedrückt(fenster, "runter") == 1 {
            fall_zaehler = fall_zaehler + 8
        }
        // Schwerkraft per Akkumulator – Tempo steuert fall_schwelle() (Level).
```

```
        fall_zaehler = fall_zaehler + 1
        wenn fall_zaehler >= fall_schwelle() {
            fall_zaehler = 0
            versuche_fallen()
        }
    }
    wenn neu_zeichnen == 1 {
        zeichne()
        neu_zeichnen = 0
    }
})
```

Lies die DAS-Logik einmal laut: Tick 1 → ein Schritt. Tick 2 bis 11 → nichts (die Wartezeit). Ab Tick 12 → Schritt, und der Zähler springt auf 9 zurück, sodass er drei Ticks später wieder 12 erreicht — Dauerfeuer alle 3 Ticks. Loslassen → Zähler auf 0, alles beginnt von vorn. Der Soft Drop ist dagegen herrlich simpel: `runter` schaufelt pro Tick 8 Extra-Punkte in den Schwerkraft-Akkumulator — der Stein sinkt rund neunmal schneller. Und beachte das `wenn modus == 1` um alles herum: Der Loop arbeitet nur, wenn wirklich gespielt wird — die Zustandsmaschine aus Kapitel 14 wirft hier ihren Schatten voraus. (Setze für den Moment oben testweise `nimm modus = 1`.)

12.4 Rotation mit Wall-Kick

Drehen heißt: Rotations-Nummer hochzählen (nach 3 kommt wieder 0) — *wenn* die neue Stellung passt. Aber was, wenn der Stein direkt an der Wand lehnt und die gedrehte Form um ein Feld hinausragen würde? Echte Tetris-Spiele schubsen ihn dann höflich von der Wand weg — der **Wall-Kick**. Wir probieren der Reihe nach: an Ort und Stelle, eins links, eins rechts — und für den langen I-Stein sogar zwei:

```
// Rotation mit Wall-Kick: erst direkt, dann ±1, für den I-Stein auch ±2.
funktion rotiere() {
    nimm nr = rot + 1
    wenn nr > 3 { nr = 0 }
    wenn kollidiert(typ, nr, px, py) == 0 {
        rot = nr
        neu_zeichnen = 1
    } sonstwenn kollidiert(typ, nr, px - 1, py) == 0 {
        px = px - 1
        rot = nr
        neu_zeichnen = 1
    } sonstwenn kollidiert(typ, nr, px + 1, py) == 0 {
        px = px + 1
        rot = nr
        neu_zeichnen = 1
    } sonstwenn kollidiert(typ, nr, px - 2, py) == 0 {
        px = px - 2
        rot = nr
        neu_zeichnen = 1
    } sonstwenn kollidiert(typ, nr, px + 2, py) == 0 {
        px = px + 2
        rot = nr
        neu_zeichnen = 1
    }
}
```

Passt gar nichts, passiert gar nichts — die Drehung wird verworfen, der Spieler merkt nur, dass „es gerade nicht geht". Fünf `sonstwenn`-Kandidaten, ein Gewinner: Das ist dieselbe Ketten-Logik wie die Punkte-Staffel aus Kapitel 4 — nur dass hier Positionen statt Punktzahlen zur Wahl stehen.

Angeschlossen wird die Rotation diskret (einmal pro Druck — Kapitel 10 lässt grüßen):

```
gui.bei_taste(fenster, funktion(f) {
    nimm taste = gui.letzte_taste(f)
    wenn modus == 1 {
        wenn taste == "hoch" { rotiere() }
        zeichne()
    }
})
```

Bau-Stopp

Bauen und ausgiebig spielen: Der Stein fällt, ← → steuern mit spürbarer DAS-Wiederholung, ↓ beschleunigt, ↑ dreht — auch direkt an der Wand (Wall-Kick!). Der Stein bleibt unten einfach liegen-ohne-liegenzubleiben (er stoppt, nichts passiert weiter) — genau richtig: Die Landung ist Kapitel 13. Teste bewusst: I-Stein längs an die linke Wand, dann drehen — er muss wegschubsen statt zu klemmen.

Aufgaben

12.1 ★ Ändere die DAS-Werte (12 und 9) auf 20/17 und auf 6/5. Was fühlt sich träge an, was nervös? Zurück zu 12/9 — oder bleib bei deinem Favoriten, es ist dein Spiel.

12.2 ★★ Erkläre in zwei Sätzen, warum `kollidiert` Parameter hat (t, r, bx, by), statt einfach die globalen `typ/rot/px/py` zu lesen. (Tipp: Schau, was `rotiere` ihr übergibt.)

12.3 ★★★ Baue testweise eine Gegen-Uhrzeiger-Rotation (`nr = rot - 1`, unter 0 → 3) auf eine zweite Taste, z. B. `"W"`. Vergiss den Wall-Kick nicht!

Teste dich (Kapitel 12)

1. Welche drei Dinge prüft `kollidiert` pro Block?
2. Warum zählt `links_seit` nach dem Wiederholungs-Schritt auf 9 zurück statt auf 0?
3. Was tut `rotiere`, wenn alle fünf Positionen kollidieren?
4. Warum darf y < 0 in `kollidiert` nicht als Kollision gelten?

13 Fixieren, Reihen löschen, Punkte

In diesem Kapitel

- landet der Stein endlich — mit Lock-Delay wie bei den Profis
- löschst du volle Reihen mit dem Trick aus Kapitel 7
- zählst du Punkte und Level — Letzteres mit Ganzzahl-Division (`div`)
- kommen 7-Bag, Hard Drop und Game Over ins Spiel

13.1 Den Steinhaufen zeichnen

Bisher zeichnest du nur den fallenden Stein. Gelandete Steine leben im `feld` — ihre Typ-Nummer ist ja zugleich ihr Farbindex. Die Doppelschleife aus Kapitel 5/7, jetzt mit Pixeln statt Kästchen-Symbolen (hänge sie in `zeichne()` direkt hinter den Rahmen):

```
funktion zeichne_feld_inhalt() {
    für y = 0 bis HÖHE - 1 {
        für x = 0 bis BREITE - 1 {
            nimm z = feld_hole(x, y)
            wenn z != 0 {
                zeichne_block(FELD_X + x * ZELLE + 1, FELD_Y + y * ZELLE + 1, ZELLE, z)
            }
        }
    }
}
```

13.2 Nachschub: der 7-Bag zieht ein

Den Beutel hast du in Kapitel 8 fertig gebaut — er zieht um, mit der inzwischen vertrauten `wert.als_ganz`-Brücke an den zwei Lese-Stellen. Dazu kommt `spawne`: Der vorgemerkte „nächste" Stein wird zum aktuellen, ein neuer wird gezogen (so kann die Vorschau in Kapitel 14 ihn anzeigen), Startposition oben Mitte:

```
nimm beutel = [0]
liste.entferne(beutel, 0)        // leere, aber typisierte Liste

funktion fülle_beutel() {
    nimm kandidaten = [1, 2, 3, 4, 5, 6, 7]
    solange liste.länge(kandidaten) > 0 {
        nimm z = zufall.ganz(0, liste.länge(kandidaten) - 1)
        liste.hänge_an(beutel, wert.als_ganz(kandidaten[z]))
        liste.entferne(kandidaten, z)
    }
}

funktion ziehe_typ() {
    wenn liste.länge(beutel) == 0 { fülle_beutel() }
    nimm t = wert.als_ganz(beutel[0])
    liste.entferne(beutel, 0)
```

```
    gib t
}

funktion spawne() {
    typ = naechster_typ
    naechster_typ = ziehe_typ()
    rot = 0
    px = 4
    py = 1
    wenn typ == 1 { py = 0 }     // I-Stein liegt flach – eine Zeile höher starten
    wenn kollidiert(typ, rot, px, py) == 1 {
        game_over()              // sofort blockiert → Partie vorbei
    }
    neu_zeichnen = 1
}
```

Die letzte Prüfung ist bemerkenswert: Wenn schon der *frisch gespawnte* Stein kollidiert, ist der Stapel bis oben voll — das ist die Game-Over-Erkennung von Tetris. Eine Funktion `game_over` gibt es noch nicht; für dieses Kapitel reicht ein Platzhalter, den Kapitel 15 dann ausbaut:

```
funktion game_over() {
    modus = 3
    neu_zeichnen = 1
}
```

13.3 Reihen prüfen, löschen, verbuchen

`reihe_voll` kennst du auswendig (Kapitel 6). Das Löschen ist der Konsolen-Trick aus Kapitel 7 — herausschneiden und oben Nullen nachschieben — verpackt in eine Schleife, die von unten nach oben wandert:

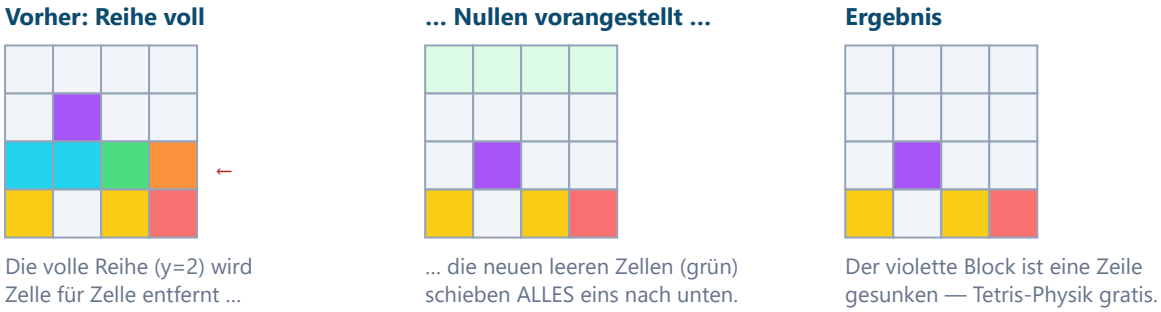


Abbildung 13.1 — Reihen löschen in der flachen Liste: entfernen + voranstellen, fertig.

```
funktion reihe_voll(y) {
    für x = 0 bis BREITE - 1 {
        wenn feld_hole(x, y) == 0 { gib 0 }
    }
    gib 1
}

// Volle Reihen entfernen: die 10 Zellen der Reihe aus der flachen Liste
// herausschneiden und 10 Nullen VORANSTELLEN – alles darüber rutscht damit
// automatisch eine Zeile nach unten. Dieselbe Zeile danach erneut prüfen.
funktion entferne_volle_reihen() {
```

```
    nimm geloescht = 0
    nimm y = HÖHE - 1
    solange y >= 0 {
        wenn reihe_voll(y) == 1 {
            für k = 1 bis BREITE { liste.entferne(feld, y * BREITE) }
            für k = 1 bis BREITE { liste.stelle_voran(feld, 0) }
            geloescht = geloescht + 1
        } sonst {
            y = y - 1
        }
    }
    gib geloescht
}
```

Die feine Stelle

Nach dem Löschen wandert `y` **nicht** weiter nach oben! Denn alles ist ja gerade eine Zeile nach unten gerutscht — in Zeile `y` liegt jetzt das, was vorher darüber war, und das könnte *auch* voll sein (4 Reihen auf einmal!). Deshalb prüft die Schleife dieselbe Zeile erneut und geht nur im `sonst`-Fall hoch. Solche Index-Feinheiten sind der Grund, warum Programmierer beim Wort „off-by-one" zusammenzucken.

Die Verbuchung kombiniert die Punkte-Staffel (Kapitel 4) mit dem Level-Aufstieg: alle 10 Reihen eine Stufe höher. Das ist genau eine **Ganzzahl-Division** — und dafür hat Kiste den `div`-Operator. (Das gewöhnliche `/` gäbe hier das präzise Ergebnis `25 / 10` → `2.5`; wir wollen aber die ganze Stufe, also `25 div 10` → `2`, plus 1 für den Start bei Level 1.)

```
// Punkte gestaffelt (Tetris = überproportional), Level alle 10 Reihen.
funktion verbuche_reihen(n) {
    wenn n > 0 {
        nimm basis = 100
        wenn n == 2 { basis = 300 }
        wenn n == 3 { basis = 500 }
        wenn n >= 4 { basis = 800 }
        punkte = punkte + basis * level
        reihen_gesamt = reihen_gesamt + n
        level = (reihen_gesamt div 10) + 1
    }
}
```

13.4 Die Landung: fixieren mit Lock-Delay

Jetzt das Finale des Kapitels. `fixiere` schreibt die 4 Blöcke ins Feld (Typ-Nummer = Farbe), lässt Reihen prüfen und verbuchen und ruft den Nachschub. Ragt der Stein beim Festschreiben oben aus dem Feld (`y < 0`), ist die Partie vorbei:

```
// Stein ins Feld schreiben; ragt er oben hinaus → Game-Over.
funktion fixiere() {
    nimm oben_raus = 0
    für k = 0 bis 3 {
        nimm x = px + form_wert(typ, rot, k * 2)
        nimm y = py + form_wert(typ, rot, k * 2 + 1)
        wenn y < 0 {
```

```
            oben_raus = 1
        } sonst {
            feld_setze(x, y, typ)
        }
    }
    nimm n = entferne_volle_reihen()
    verbuche_reihen(n)
    wenn oben_raus == 1 {
        game_over()
    } sonst {
        spawne()
    }
    neu_zeichnen = 1
}
```

Und `versuche_fallen` bekommt seinen fehlenden `sonst`-Zweig — aber nicht sofort zuschnappend! Gutes Tetris gönnt dir beim Aufsetzen einen Wimpernschlag, um den Stein noch unter eine Kante zu schieben — den **Lock-Delay**. Wir zählen, wie oft die Schwerkraft den Stein schon „am Boden" angetroffen hat (oben anlegen: `nimm boden_seit = 0`):

```
// Eine Zeile fallen; geht es nicht, erst nach kurzem Lock-Delay fixieren –
// der klassische Moment, um den Stein noch unter eine Kante zu schieben.
funktion versuche_fallen() {
    wenn kollidiert(typ, rot, px, py + 1) == 0 {
        py = py + 1
        boden_seit = 0
        neu_zeichnen = 1
    } sonst {
        boden_seit = boden_seit + 1
        wenn boden_seit >= 2 {       // ~2 Schwerkraft-Schläge Gnadenfrist
            boden_seit = 0
            fixiere()
        }
    }
}
```

13.5 Hard Drop — das Ausrufezeichen

Der Hard Drop (Leertaste) lässt den Stein sofort ganz nach unten knallen — `solange` Platz ist, eine Zeile tiefer, 2 Bonuspunkte pro Zeile, dann *ohne* Gnadenfrist fixieren. Häng ihn in den `bei_taste`-Handler (`wenn taste == "leer" { hard_drop() }`):

```
funktion hard_drop() {
    solange kollidiert(typ, rot, px, py + 1) == 0 {
        py = py + 1
        punkte = punkte + 2      // kleiner Bonus pro übersprungener Zeile
    }
    fixiere()
}
```

Bau-Stopp — das ist schon Tetris!

Bauen und spielen. Steine fallen, landen, stapeln sich; volle Reihen verschwinden; die Leertaste donnert. Es gibt noch keine Anzeige (Punkte zählen unsichtbar mit) und kein Menü — aber das Spielgefühl ist da. Wenn du jetzt „nur noch kurz eine Reihe" sagst und 20 Minuten später wieder auftauchst: vollkommen normal, ärztlich unbedenklich, willkommen im Klub.

Aufgaben

13.1 ★ Stell den Lock-Delay auf 0 (sofort fixieren) und auf 5. Spüre den Unterschied beim Schieben unter Kanten. Zurück auf 2.

13.2 ★★ Rechne nach: Du löschst nacheinander 2 + 1 + 4 + 3 Reihen, alle auf Level 1. Wie viele Punkte? Und auf welchem Level bist du danach (Schwelle beachten!)?

13.3 ★★★ Warum prüft `fixiere` auf `y < 0`, obwohl `spawne` doch schon Kollision prüft? Konstruiere die Situation, die nur die `fixiere`-Prüfung erwischt. (Tipp: Lock-Delay + ganz oben drehen.)

Teste dich (Kapitel 13)

1. Warum bleibt `y` nach einem Reihen-Löschen stehen, statt weiterzuwandern?
2. Wie steigt das Level — und worin unterscheidet sich `div` von `/`?
3. Was unterscheidet die Landung per Schwerkraft von der per Hard Drop?

14 Die Zustandsmaschine — Menü, Pause, Game Over

In diesem Kapitel

- verstehst du Zustandsmaschinen — das Rückgrat jedes Spiels
- baust du Titelbild, Pause- und Game-Over-Schirm
- kommen Info-Panel, Next-Vorschau und Ghost Piece dazu

14.1 Ein Spiel ist mehr als Spielen

Drück mal in einem echten Spiel die Leertaste: Im Menü startet sie das Spiel, mitten in der Partie ist sie der Hard Drop, im Game-Over-Schirm startet sie neu. *Dieselbe Taste, drei Bedeutungen.* Die Lösung ist ehrwürdig und simpel: eine Variable, die sagt, in welchem **Zustand** sich das Spiel befindet — unser `modus` aus Kapitel 11:

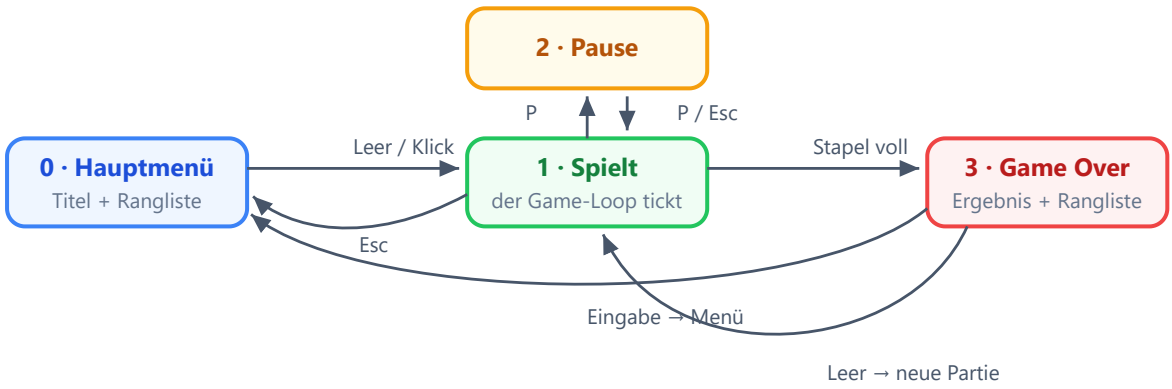


Abbildung 14.1 — Die Zustandsmaschine von Kistris: vier Zustände, klare Übergänge.

Zwei eiserne Regeln machen daraus stabilen Code: **Jeder Tasten-Handler prüft zuerst den Modus.** Und: **Gezeichnet wird, was der Modus sagt.** Hier der vollständige Tasten-Handler — ersetze deinen kleinen aus Kapitel 12:

```
// Diskrete Tasten: Rotation, Hard Drop, Pause, Zustandswechsel.
gui.bei_taste(fenster, funktion(f) {
    nimm taste = gui.letzte_taste(f)
    wenn modus == 1 {
        wenn taste == "hoch" { rotiere() }
        wenn taste == "leer" { hard_drop() }
        wenn taste == "p" oder taste == "P" {
            modus = 2
        }
        wenn taste == "escape" {
            modus = 0
        }
        zeichne()
    } sonstwenn modus == 2 {
        wenn taste == "p" oder taste == "P" oder taste == "escape" oder taste == "leer" {
```

```
            modus = 1
            zeichne()
        }
    } sonstwenn modus == 3 {
        wenn taste == "leer" { starte_spiel() }
        wenn taste == "eingabe" {
            modus = 0
            zeichne()
        }
    } sonstwenn modus == 0 {
        wenn taste == "leer" oder taste == "eingabe" { starte_spiel() }
    }
})
```

Dazu der saubere Spielstart — alles zurück auf Anfang, frische Tüte, erster Stein:

```
funktion starte_spiel() {
    feld_leeren()
    punkte = 0
    level = 1
    reihen_gesamt = 0
    fall_zaehler = 0
    boden_seit = 0
    solange liste.länge(beutel) > 0 {    // frische Tüte für die neue Partie
        liste.entferne(beutel, 0)
    }
    naechster_typ = ziehe_typ()
    spawne()
    modus = 1
    zeichne()
}
```

Wichtig: Oben in der Datei steht jetzt wieder `nimm modus = 0` — das Spiel beginnt im Menü, nicht mitten in der Partie.

14.2 Drei Bildschirme, eine Leinwand

Wir zeichnen alle Zustände auf *dieselbe* Leinwand — `zeichne()` wird zur Weiche. Die Overlays nutzen den Alpha-Trick aus Kapitel 9: `"#0f172acc"` ist unser Nachtblau mit Durchsichtigkeit — das Spielfeld schimmert unter der Pause durch:

```
funktion zeichne_pause() {
    gui.rechteck(bild, FELD_X, FELD_Y, BREITE * ZELLE, HÖHE * ZELLE, "#0f172acc")
    gui.text_an(bild, FELD_X + 95, FELD_Y + 250, "Pause", "#fbbf24", 32)
    gui.text_an(bild, FELD_X + 93, FELD_Y + 302, "P = weiter", "#94a3b8", 18)
}

funktion zeichne_vorbei() {
    gui.rechteck(bild, FELD_X, FELD_Y, BREITE * ZELLE, HÖHE * ZELLE, "#0f172add")
    gui.rechteck(bild, FELD_X + 10, FELD_Y + 90, BREITE * ZELLE - 20, 400, "#1e293b")
    gui.text_an(bild, FELD_X + 58, FELD_Y + 110, "Game Over", "#f87171", 34)
    gui.text_an(bild, FELD_X + 95, FELD_Y + 165, "Punkte: {punkte}", "#e4e4e7", 20)
    gui.text_an(bild, FELD_X + 48, FELD_Y + 440, "Leertaste = neue Partie", "#e4e4e7", 17)
    gui.text_an(bild, FELD_X + 73, FELD_Y + 464, "Eingabe = Hauptmenü", "#94a3b8", 15)
}
```

```
funktion zeichne_titel() {
    gui.text_an(bild, 153, 63, "KISTRIS", "#0c4a6e", 64)                  // Schatten
    gui.text_an(bild, 148, 58, "KISTRIS", "#38bdf8", 64)
    gui.text_an(bild, 175, 140, "Tetris, gebaut in Kiste", "#94a3b8", 18)
    // Deko: die sieben Steine in ihren Farben, einmal quer übers Bild.
    nimm deko_x = 62
    für t = 1 bis 7 {
        zeichne_form(t, 0, deko_x, 215, 17)
        deko_x = deko_x + 70
    }
    gui.text_an(bild, 130, 530, "links/rechts bewegen  ·  hoch drehen  ·  runter sinken", "#94a3b8", 15)
    gui.text_an(bild, 152, 554, "Leertaste Hard Drop  ·  P Pause  ·  Esc Menü", "#94a3b8", 15)
    gui.text_an(bild, 173, 632, "Leertaste oder Klick = Start", "#fbbf24", 22)
}

funktion zeichne() {
    gui.leeren(bild)
    gui.rechteck(bild, 0, 0, BILD_B, BILD_H, "#0f172a")
    wenn modus == 0 {
        zeichne_titel()
    } sonst {
        zeichne_feld_rahmen()
        zeichne_feld_inhalt()
        wenn modus == 1 { zeichne_ghost() }
        zeichne_stein()
        zeichne_panel()
        wenn modus == 2 { zeichne_pause() }
        wenn modus == 3 { zeichne_vorbei() }
    }
}
```

Der Doppel-Text im Titel ist ein Mini-Designtrick: erst dunkel, 5 Pixel versetzt, dann hell obendrauf — fertig ist der Schlagschatten. Und die Steine-Parade nutzt eine kleine neue Hilfsfunktion, die einen Stein *frei* auf der Leinwand platziert (sie zeichnet gleich auch die Vorschau):

```
// Einen Steintyp frei auf der Leinwand zeichnen (Vorschau, Titel-Deko).
// (pvx, pvy) = Pixel-Position des Drehpunkts, gr = Blockgröße.
funktion zeichne_form(t, r, pvx, pvy, gr) {
    für k = 0 bis 3 {
        nimm x = pvx + form_wert(t, r, k * 2) * gr
        nimm y = pvy + form_wert(t, r, k * 2 + 1) * gr
        zeichne_block(x, y, gr, t)
    }
}
```

14.3 Panel, Vorschau, Ghost

Rechts neben dem Feld wohnen Logo, Zahlen und die Vorschau auf den nächsten Stein (`naechster_typ` wartet ja schon seit Kapitel 13 darauf, angezeigt zu werden):

```
funktion zeichne_vorschau() {
    gui.text_an(bild, PANEL_X, FELD_Y + 310, "Nächster", "#94a3b8", 16)
    gui.rechteck(bild, PANEL_X, FELD_Y + 338, 130, 96, "#1e293b")
    // Drehpunkt so versetzen, dass jede Form optisch mittig sitzt (gr = 22):
```

```
    nimm ox = PANEL_X + 54          // 3 Zellen breit (T/S/Z/J/L)
    nimm oy = FELD_Y + 338 + 48
    wenn naechster_typ == 1 {       // I: 4 breit, 1 hoch
        ox = PANEL_X + 43
        oy = FELD_Y + 338 + 37
    }
    wenn naechster_typ == 2 {       // O: 2 × 2 um den Drehpunkt rechts-unten
        ox = PANEL_X + 43
        oy = FELD_Y + 338 + 26
    }
    zeichne_form(naechster_typ, 0, ox, oy, 22)
}

funktion zeichne_panel() {
    gui.text_an(bild, PANEL_X + 2, FELD_Y + 2, "KISTRIS", "#0c4a6e", 34)   // Schatten
    gui.text_an(bild, PANEL_X, FELD_Y, "KISTRIS", "#38bdf8", 34)
    gui.text_an(bild, PANEL_X, FELD_Y + 70, "Punkte", "#94a3b8", 16)
    gui.text_an(bild, PANEL_X, FELD_Y + 92, "{punkte}", "#e4e4e7", 26)
    gui.text_an(bild, PANEL_X, FELD_Y + 150, "Level", "#94a3b8", 16)
    gui.text_an(bild, PANEL_X, FELD_Y + 172, "{level}", "#e4e4e7", 26)
    gui.text_an(bild, PANEL_X, FELD_Y + 230, "Reihen", "#94a3b8", 16)
    gui.text_an(bild, PANEL_X, FELD_Y + 252, "{reihen_gesamt}", "#e4e4e7", 26)
    zeichne_vorschau()
    gui.text_an(bild, PANEL_X, FELD_Y + 540, "P  Pause", "#64748b", 14)
    gui.text_an(bild, PANEL_X, FELD_Y + 560, "Esc  Menü", "#64748b", 14)
}
```

Bleibt das **Ghost Piece** — der Schatten am Boden, der zeigt, wo der Stein landen würde. Die Logik ist ein gedanklicher Hard Drop (gleiche `solange`-Schleife, nur ohne Folgen), die Optik eine Alpha-Farbe: Wir hängen einfach `40` an den Hex-Code des Steins — 25 % Deckkraft:

```
// Wo käme der Stein bei einem Hard Drop an? (Für das Ghost Piece.)
funktion ghost_y() {
    nimm gy = py
    solange kollidiert(typ, rot, px, gy + 1) == 0 {
        gy = gy + 1
    }
    gib gy
}

// Ghost Piece: halbtransparente Landeposition (Hex mit Alpha) – rein visuell.
funktion zeichne_ghost() {
    nimm gy = ghost_y()
    wenn gy > py {
        nimm f = farbe_von(typ)
        nimm geist = "{f}40"
        für k = 0 bis 3 {
            nimm x = px + form_wert(typ, rot, k * 2)
            nimm y = gy + form_wert(typ, rot, k * 2 + 1)
            wenn y >= 0 {
                gui.rechteck(bild, FELD_X + x * ZELLE + 2, FELD_Y + y * ZELLE + 2, ZELLE - 4, ZELLE - 4,
geist)
            }
        }
    }
}
```

Noch zwei Handgriffe: Erstens, am Dateiende vor `gui.zeige` ein Maus-Handler — ein Klick aufs Spielfeld startet vom Titel und nach dem Game Over (warum das später noch wichtig wird, verrät Kapitel 15):

```
gui.bei_maus_klick(bild, funktion(b) {
    wenn modus == 0 oder modus == 3 { starte_spiel() }
})
```

Zweitens: Vor `fenster_öffne` gehört jetzt `lade_rangliste()` … nein, halt — die gibt es ja noch gar nicht. Die bekommt das nächste Kapitel. Stattdessen nur: direkt vor dem Fenster-Öffnen einmal `zeichne()` aufrufen, damit das Titelbild beim Start schon dasteht.

Bau-Stopp

Der volle Durchlauf muss sitzen: Titelbild mit Steine-Parade → Leertaste → Spiel mit Panel, Vorschau und Ghost → P pausiert (Feld schimmert durch!) → P weiter → absichtlich verlieren → Game-Over-Schirm mit Punktzahl → Leertaste → neue Partie → Esc → Menü. Jede Kante dieses Diagramms einmal ablaufen — du testest gerade systematisch eine Zustandsmaschine, wie es Profis tun.

Aufgaben

14.1 ★ Ändere den Ghost von `40` auf `80` Alpha. Zu aufdringlich? Dein Spiel, dein Geschmack.

14.2 ★★ Warum prüft `zeichne_ghost` `wenn gy > py`? Was würde ohne diese Zeile passieren, wenn der Stein direkt auf dem Stapel aufliegt?

14.3 ★★★ Füge einen fünften Zustand hinzu: `modus 4` = „Countdown" — nach dem Start zählt das Spiel 3-2-1 herunter (nutze einen Tick-Zähler im Game-Loop), erst dann fällt der erste Stein. Alle nötigen Werkzeuge hast du.

Teste dich (Kapitel 14)

1. Warum prüft jeder Tasten-Handler *zuerst* den Modus?
2. Wie entsteht der „durchscheinende" Pause-Effekt?
3. Worin unterscheiden sich `zeichne_stein` und `zeichne_form`?
4. Welche Funktion teilen sich Hard Drop und Ghost Piece — und warum ist das kein Zufall?

15 Die Rangliste — Dateien und Widgets

In diesem Kapitel

- liest und schreibst du Dateien mit dem `datei`-Modul
- tippst du Text direkt auf die Leinwand — Namenseingabe ganz ohne Widget
- sortierst du ohne Sortier-Funktion — per Einfüge-Sortierung
- überlebt dein Highscore endlich den Neustart

15.1 Der Plan

Eine Rangliste braucht drei Dinge: einen **Namen** (den der Spieler nach dem Spiel eintippt), eine **sortierte Liste** der besten Ergebnisse (im Speicher) und eine **Datei**, damit alles den Neustart überlebt. Wir gehen sie rückwärts an — erst die Daten, dann die Datei, dann die Namenseingabe.

Den Namen tippst du **nach** der Partie direkt auf den Game-Over-Schirm — so wie in den Arcade-Automaten früher. Das ist nicht nur stimmiger, es erspart uns auch ein Eingabefeld-Widget, das während des Spiels heikel wäre (dazu gleich in 15.4 mehr).

15.2 Sortieren per Einfügen — parallele Listen

Wir halten zwei Listen nebeneinander: `rang_punkte` und `rang_namen` — Index `i` gehört in beiden zum selben Eintrag. Und statt je zu „sortieren", halten wir die Listen *von Anfang an* sortiert: Jeder neue Eintrag wird gleich an der richtigen Stelle *eingefügt*. Lege oben an: `nimm bestwert = 0`, `fest RANGDATEI = "kistris_rang.txt"`, `nimm rang_punkte = [0]`, `nimm rang_namen = [""]` (je ein Platzhalter — die typisierte-Liste-Nummer kennst du ja).

```
funktion rang(i) { gib wert.als_ganz(rang_punkte[i]) }
funktion rang_name(i) { gib wert.als_text(rang_namen[i]) }

funktion sortiert_einfügen(p, name) {
    nimm pos = 0
    für i = 0 bis liste.länge(rang_punkte) - 1 {
        wenn rang(i) < p { pos = i + 1 }
    }
    liste.füge_ein(rang_punkte, pos, p)
    liste.füge_ein(rang_namen, pos, name)
}

funktion aktualisiere_bestwert() {
    nimm n = liste.länge(rang_punkte)
    bestwert = rang(n - 1)
}
```

`sortiert_einfügen` sucht die letzte Position, deren Punkte noch kleiner sind, und fügt *dahinter* ein — die Listen bleiben dadurch immer aufsteigend sortiert, der beste Eintrag steht hinten. (`wert.als_text` ist die Text-Schwester von `als_ganz` — dieselbe Auspack-Brücke, nur für Namen.)

Build-Subset: Warum nicht `liste.sortiere`?
Kiste hat durchaus Sortier-Funktionen — aber das Sortieren *zweier zusammengehöriger* Listen bräuchte eine Vergleichs-Funktion, und genau die ist im nativen Build noch begrenzt. Die Einfüge-Sortierung umgeht das komplett — und ist obendrein ein Klassiker, den jeder Informatik-Studienanfänger lernen muss. Du kriegst ihn gratis mit Tetris.

15.3 Die Datei: ein Format mit Köpfchen

Das `datei`-Modul macht Lesen und Schreiben einfach: `datei.existiert(pfad)`, `datei.zeilen(pfad)` (liefert eine Liste der Zeilen) und `datei.schreibe(pfad, text)`. Spannender ist das *Format*. Unseres: Zeile 1 ist ein **Marker** (`kistris-v1`), danach **zwei Zeilen pro Eintrag** — Punkte, dann Name:

```
AUSGABE
kistris-v1
3682
Sascha
408
Sascha
```

Warum der Marker? Falls je eine fremde oder uralte Datei dort liegt, erkennen wir sie und ignorieren sie, statt Müll zu laden. Warum zwei Zeilen pro Eintrag? Weil Namen alles enthalten dürfen — auch Leerzeichen und Kommas. Eine eigene Zeile ist die sicherste Schublade. Solche Format-Entscheidungen unterscheiden Programme, die funktionieren, von Programmen, die *weiter* funktionieren.

```
funktion lade_rangliste() {
    wenn datei.existiert(RANGDATEI) {
        nimm zeilen = datei.zeilen(RANGDATEI)
        nimm n = liste.länge(zeilen)
        wenn n > 0 und zeilen[0] == "kistris-v1" {
            für i = 1 bis n - 2 schritt 2 {
                wenn länge(zeilen[i]) > 0 {
                    sortiert_einfügen(als_ganz(zeilen[i]), zeilen[i + 1])
                }
            }
        }
    }
    aktualisiere_bestwert()
}

// Ergebnis eintragen, beste 8 behalten, Datei komplett neu schreiben.
funktion trage_ein(p, name) {
    sortiert_einfügen(p, name)
    solange liste.länge(rang_punkte) > 8 {     // beste 8 behalten - der 0-Platz-
        liste.entferne(rang_punkte, 0)         // halter ist der kleinste und fliegt
        liste.entferne(rang_namen, 0)          // als Erster vorne raus
    }
    nimm s = "kistris-v1\n"
    nimm n = liste.länge(rang_punkte)
    für i = 0 bis n - 1 {
        nimm r = rang(n - 1 - i)
        wenn r > 0 { s = "{s}{r}\n{rang_name(n - 1 - i)}\n" }
```

```
    }
    datei.schreibe(RANGDATEI, s)
    aktualisiere_bestwert()
}
```

Drei Lese-Details, alle schon mal dagewesen: Die Lade-Schleife springt mit `schritt 2` durch die Zeilen-*Paare*. `als_ganz(text)` (Kapitel 4, Aufgabe 4.3!) macht aus der Zeile `"3682"` die Zahl. Und beim Schreiben läuft `n - 1 - i` die Liste *rückwärts* ab — der beste Eintrag steht ja hinten, soll in der Datei aber vorn stehen. `\n` im Text ist übrigens der Zeilenumbruch.

15.4 Der Name — nach dem Spiel getippt

Jetzt der Name. Wir könnten ein **Eingabefeld-Widget** in eine Kopfzeile setzen (das große Lernbuch zeigt das im Detail) — aber ein Tippfeld, das den **Fokus** hat, frisst *alle* Tasten, auch Pfeile und Leertaste. Mitten im Spiel wäre das ein Ärgernis. Deshalb machen wir es wie die alten Arcade-Automaten: Der Name wird **erst nach dem Spiel** eingetippt, direkt auf den Game-Over-Schirm. Kein Widget, kein Fokus-Problem — wir lesen die Tasten einfach selbst, so wie du es aus Kapitel 10 längst kannst.

Dafür genügen zwei neue globale Variablen (zu den anderen oben) und ein `nutze text` ganz oben bei den Modulen (für das Kürzen beim Löschen):

```
nutze text                      // für text.erste_n (Backspace)

nimm eingabe_name = ""          // der nach dem Spiel getippte Name
nimm gespeichert = 0            // 1 = Ergebnis dieser Partie schon eingetragen

funktion spieler_name() {
    wenn länge(eingabe_name) == 0 { gib "Anonym" }
    gib eingabe_name
}
```

Wer nichts eintippt, heißt eben „Anonym" (das Internet ist voll von dieser Person, sie ist erstaunlich gut in allem). Das eigentliche Tippen passiert im Tasten-Handler — im Game-Over-Zustand (`modus == 3`), solange noch nicht gespeichert wurde. Ersetze den kleinen `modus == 3`-Zweig aus Kapitel 14 durch diesen:

```
    } sonstwenn modus == 3 {
        wenn gespeichert == 0 {
            // Namenseingabe: Buchstaben tippen, Enter speichert, Esc = Menü.
            wenn taste == "eingabe" {
                speichern()
            } sonstwenn taste == "escape" {
                modus = 0
            } sonstwenn taste == "BackSpace" {
                wenn länge(eingabe_name) > 0 {
                    eingabe_name = text.erste_n(eingabe_name, länge(eingabe_name) - 1)
                }
            } sonstwenn taste == "leer" {
                wenn länge(eingabe_name) < 14 { eingabe_name = "{eingabe_name} " }
            } sonstwenn länge(taste) == 1 {
                wenn länge(eingabe_name) < 14 { eingabe_name = "{eingabe_name}{taste}" }
            }
        } sonst {
```

```
            // Schon gespeichert: neue Partie oder zurück ins Menü.
            wenn taste == "leer" { starte_spiel() }
            sonstwenn taste == "eingabe" oder taste == "escape" { modus = 0 }
        }
        zeichne()
    }
```

Der Trick steckt in `gui.letzte_taste`: Normale Buchstaben kommen als **ein-Zeichen-Text** an (`länge(taste) == 1`) — den hängen wir einfach mit den Zauberklammern an `eingabe_name` an. Sondertasten haben lange Namen: `"eingabe"` (Enter), `"escape"`, `"BackSpace"`, `"leer"` (Leertaste). `text.erste_n(text, n)` gibt die ersten `n` Zeichen zurück — mit `länge - 1` wird daraus ein Rückschritt. Die `< 14`-Grenze hält den Namen kurz genug für die Anzeige.

Warum kein Eingabefeld?
Weil wir den Fokus-Ärger komplett umgehen: Es gibt kein Widget, das dem Spiel die Tasten wegschnappen könnte. Während des Spiels landet *jede* Taste dort, wo sie hingehört; erst im Game-Over-Schirm werden Buchstaben zu einem Namen. Das `gespeichert`-Flag ist die Weiche: `0` = du tippst noch, `1` = fertig, jetzt zählt Leertaste wieder als „neue Partie".

Ein Detail noch: Der **Maus-Klick** aus Kapitel 14 startete bei `modus == 3` sofort neu — das würde jetzt mitten in die Namenseingabe platzen. Er darf erst wieder starten, wenn schon gespeichert wurde. Ersetze den Handler:

```
gui.bei_maus_klick(bild, funktion(b) {
    wenn modus == 0 { starte_spiel() }
    sonstwenn modus == 3 und gespeichert == 1 { starte_spiel() }
})
```

15.5 Alles verbinden

Vier Handgriffe. **Erstens:** `game_over` speichert jetzt *nicht* sofort — es macht nur den Schirm für die Namenseingabe frei (ersetze den Platzhalter aus Kapitel 13). Das eigentliche Eintragen erledigt `speichern`, wenn der Spieler Enter drückt:

```
// Partie beendet: Namenseingabe vorbereiten (Eintragen kommt bei Enter).
funktion game_over() {
    modus = 3
    gespeichert = 0
    eingabe_name = ""
    neu_zeichnen = 1
}

// Ergebnis erst NACH der Namenseingabe eintragen (Enter im Game-Over-Bild).
funktion speichern() {
    wenn modus == 3 und gespeichert == 0 {
        wenn punkte > 0 { trage_ein(punkte, spieler_name()) }
        gespeichert = 1
        neu_zeichnen = 1
    }
}
```

Zweitens: eine Anzeige-Funktion, die die besten Einträge an beliebiger Stelle auflistet — Aufrufe davon im Titel (`zeichne_rangliste(195, 280, 8)` nach der Steine-Parade) und im Game-Over-Schirm (gleich unten):

```
// Die besten Einträge der Rangliste an Position (x, y) – anzahl Zeilen.
funktion zeichne_rangliste(x, y, anzahl) {
    gui.text_an(bild, x + 40, y, "Rangliste", "#fbbf24", 22)
    nimm n = liste.länge(rang_punkte)
    nimm zeile = 0
    für i = 0 bis n - 1 {
        nimm p = rang(n - 1 - i)
        wenn p > 0 und zeile < anzahl {
            zeile = zeile + 1
            gui.text_an(bild, x, y + 12 + zeile * 26, "{zeile}.  {rang_name(n - 1 - i)} – {p}", "#e4e4e7",
16)
        }
    }
    wenn zeile == 0 {
        gui.text_an(bild, x, y + 40, "noch keine Einträge", "#94a3b8", 15)
    }
}
```

Drittens: der Game-Over-Schirm aus Kapitel 14 bekommt seine zwei Gesichter. Solange `gespeichert == 0` ist, zeigt er die **Namenseingabe** mit einem blinkenden Cursor (`{eingabe_name}|` — der senkrechte Strich ist einfach ein getipptes Zeichen). Danach die **Rangliste**. Ersetze `zeichne_vorbei` aus Kapitel 14:

```
funktion zeichne_vorbei() {
    gui.rechteck(bild, FELD_X, FELD_Y, BREITE * ZELLE, HÖHE * ZELLE, "#0f172add")
    gui.rechteck(bild, FELD_X + 10, FELD_Y + 90, BREITE * ZELLE - 20, 400, "#1e293b")
    gui.text_an(bild, FELD_X + 58, FELD_Y + 110, "Game Over", "#f87171", 34)
    gui.text_an(bild, FELD_X + 95, FELD_Y + 165, "Punkte: {punkte}", "#e4e4e7", 20)
    wenn gespeichert == 1 {
        zeichne_rangliste(FELD_X + 55, FELD_Y + 215, 6)
        gui.text_an(bild, FELD_X + 48, FELD_Y + 440, "Leertaste = neue Partie", "#e4e4e7", 17)
        gui.text_an(bild, FELD_X + 73, FELD_Y + 464, "Eingabe = Hauptmenü", "#94a3b8", 15)
    } sonst {
        gui.text_an(bild, FELD_X + 55, FELD_Y + 225, "Gib deinen Namen ein:", "#fbbf24", 18)
        gui.rechteck(bild, FELD_X + 40, FELD_Y + 260, 220, 40, "#0b1120")
        gui.text_an(bild, FELD_X + 52, FELD_Y + 270, "{eingabe_name}|", "#e4e4e7", 20)
        gui.text_an(bild, FELD_X + 58, FELD_Y + 332, "Enter = Name speichern", "#94a3b8", 15)
        gui.text_an(bild, FELD_X + 92, FELD_Y + 358, "Esc = Hauptmenü", "#64748b", 14)
    }
}
```

Viertens: der Bestwert ins Panel (in `zeichne_panel`, nach der Vorschau) — und beim Programmstart einmal laden (direkt vor dem ersten `zeichne()` am Dateiende ein `lade_rangliste()`):

```
    gui.text_an(bild, PANEL_X, FELD_Y + 460, "Bestwert", "#94a3b8", 16)
    gui.text_an(bild, PANEL_X, FELD_Y + 482, "{bestwert}", "#fbbf24", 26)
```

Bau-Stopp

Bauen, eine Partie verlieren: Der Game-Over-Schirm fragt nach deinem Namen. Tippen, mit **Enter** speichern — dein Name muss in der Rangliste erscheinen, auch auf dem Titel. Jetzt das Programm **komplett schließen und neu starten** — die Rangliste muss noch da sein. Wirf auch einen Blick in die Datei `kistris_rang.txt` (liegt neben der .exe): Da steht dein Format, Marker und alles. Du hast soeben Persistenz gebaut.

Aufgaben

15.1 ★ Ändere die Rangliste auf „beste 5". Welche Zahl(en) musst du anfassen?

15.2 ★★ Öffne `kistris_rang.txt` in einem Editor und ändere die Marker-Zeile in `kistris-v2`. Starte das Spiel. Was passiert — und warum ist genau das das gewünschte Verhalten?

15.3 ★★★ Erkläre Schritt für Schritt, wo `sortiert_einfügen(350, "Mia")` den Eintrag platziert, wenn `rang_punkte` gerade `[0, 198, 408, 3682]` enthält. (Papier reicht — aber nachprüfen darfst du.)

Teste dich (Kapitel 15)

1. Wozu dient die Marker-Zeile in der Datei?
2. Warum zwei Zeilen pro Eintrag statt „Punkte,Name" in einer?
3. Warum bleibt die Liste durch `sortiert_einfügen` dauerhaft sortiert?
4. Warum tippt der Spieler seinen Namen erst nach dem Spiel — und nicht während des Spiels in ein Eingabefeld?

16 Sound, Feinschliff — und dein fertiges Spiel

In diesem Kapitel

- spielst du Klänge ab mit dem `klang`-Modul
- baust du Checkbox und Lautstärke-Regler in die Kopfzeile
- polierst du das Spielfeld mit einem Gitter
- … und dann bist du fertig. Wirklich.

16.1 Vier Geräusche und eine Melodie

Sound in Kiste ist erfreulich unkompliziert: WAV-Datei laden, abspielen, fertig. Die Dateien liegen in einem Ordner `klaenge/` neben deinem Programm. (Im Kistris-Projektordner des Kiste-Repos liegen fünf fertige Chiptune-Klänge bereit — samt dem Python-Skript, das sie erzeugt hat. Die Hintergrundmusik ist Korobeiniki, das russische Volkslied, das die Welt als „die Tetris-Melodie" kennt — gemeinfrei und unsterblich. Eigene WAVs gehen natürlich genauso.)

```
// Klänge (WAV neben der .exe in klaenge/) – zu den globalen Variablen:
nimm s_landung = klang.lade("klaenge/landung.wav")
nimm s_reihe = klang.lade("klaenge/reihe.wav")
nimm s_tetris = klang.lade("klaenge/tetris.wav")
nimm s_gameover = klang.lade("klaenge/gameover.wav")
nimm s_musik = klang.lade("klaenge/musik.wav")
```

Abgespielt wird an den Spiel-Ereignissen — drei kleine Ergänzungen in Funktionen, die du längst hast. In `verbuche_reihen`, ans Ende des `wenn n > 0`-Blocks:

```
        wenn n >= 4 {
            klang.spiele(s_tetris)            // „Tetris!" – die große Fanfare
        } sonst {
            klang.spiele(s_reihe)
        }
```

In `fixiere`, direkt nach `verbuche_reihen(n)` — der Aufprall-Sound, aber nur, wenn keine Reihe klingt (sonst überlagern sich die Effekte unschön):

```
    wenn n == 0 { klang.spiele(s_landung) }    // sonst klingt schon die Reihe
```

Und in `game_over`, gleich nach `modus = 3`: `klang.spiele(s_gameover)`.

16.2 Musik an, Musik aus — Checkbox und Regler

Hintergrundmusik soll man abschalten können (frag deine Mitbewohner). Zwei neue Widgets — ein **Kontrollkästchen** und ein **Schieberegler** — plus ihre Handler. Zu den Widget-Definitionen oben:

```
nimm musik_haken = gui.kontrollkästchen("Musik")
```

```
nimm lautstärke_regler = gui.schieberegler(0, 100)
```

Die Handler (zu den anderen Funktionen — beachte wieder das Handle-und-Getter-Muster: `gui.angehakt(k)` bzw. `gui.schieber_wert(s)`):

```
// Hintergrundmusik an/aus – Kontrollkästchen in der Kopfzeile.
funktion bei_musik(k) {
    wenn gui.angehakt(k) == 1 {
        klang.spiele_schleife(s_musik)
    } sonst {
        klang.stoppe(s_musik)
    }
}

// Master-Lautstärke für alle Spielklänge (Regler in der Kopfzeile).
funktion bei_lautstärke(s) {
    nimm w = gui.schieber_wert(s)
    klang.lautstärke(s_musik, w)
    klang.lautstärke(s_landung, w)
    klang.lautstärke(s_reihe, w)
    klang.lautstärke(s_tetris, w)
    klang.lautstärke(s_gameover, w)
}
```

Verdrahtet wird nach `fenster_öffne` (Handler anmelden, Startwert setzen) — und jetzt bekommt Kistris seine **Kopfzeile**. Bisher war das Fenster nur die Leinwand; die beiden Musik-Widgets wohnen in einer schmalen Leiste darüber. Ersetze am Dateiende dein `gui.zeige(fenster, bild)` durch:

```
gui.bei_umschaltung(musik_haken, bei_musik)
gui.setze_schieber_wert(lautstärke_regler, 80)
gui.bei_schieben(lautstärke_regler, bei_lautstärke)

// Kopfzeile: links der Musik-Haken, daneben dehnt sich der Lautstärke-Regler
// (die MITTE eines Rand-Layouts bekommt die restliche Breite).
nimm kopf = gui.rand_layout(lautstärke_regler, [0, 0, musik_haken, 0])
nimm inhalt = gui.vertikal([kopf, bild])
gui.zeige(fenster, inhalt)
gui.starte()
```

Damit die Leiste Platz hat, gib dem Fenster etwas Kopfhöhe — ändere die `fenster_öffne`-Zeile aus Kapitel 11 auf `gui.fenster_öffne("Kistris", BILD_B + 1, BILD_H + 60)`. `gui.vertikal([…])` stapelt Kopfzeile und Leinwand untereinander; `gui.rand_layout(mitte, [oben, unten, links, rechts])` legt ein Element in die Mitte (wo es sich dehnen darf) und die anderen an die Ränder — die Nullen heißen „nichts". `klang.spiele_schleife` spielt die Musik in Endlos-Schleife, `klang.stoppe` beendet sie, und `klang.lautstärke(klang, 0–100)` regelt — der Regler ist die Master-Lautstärke für alles.

16.3 Der letzte Schliff: das Gitter

Profi-Tetris hat ein dezentes Gitter im Feld — es gibt den Augen Halt beim Zielen. Zwei Schleifen mit `gui.linie` in `zeichne_feld_rahmen`, Farbe so dunkel, dass sie nur flüstert:

```
    // dezentes Gitter – gibt dem Feld Tiefe, ohne von den Steinen abzulenken
    für x = 1 bis BREITE - 1 {
        gui.linie(bild, FELD_X + x * ZELLE, FELD_Y, FELD_X + x * ZELLE, FELD_Y + HÖHE * ZELLE, "#1e293b",
1)
    }
    für y = 1 bis HÖHE - 1 {
        gui.linie(bild, FELD_X, FELD_Y + y * ZELLE, FELD_X + BREITE * ZELLE, FELD_Y + y * ZELLE, "#1e293b",
1)
    }
```

Letzter Bau-Stopp — die Abnahme

Bauen. Musik anhaken. Eine ernsthafte Partie spielen und dabei die Checkliste abhaken: Titelbild ✓ Steine fallen mit Gitter ✓ DAS-Steuerung ✓ Rotation samt Wall-Kick ✓ Ghost ✓ Soft + Hard Drop ✓ Reihen verschwinden mit Sound ✓ Vierfach räumt mit Fanfare ✓ Level beschleunigt ✓ Vorschau stimmt ✓ Pause ✓ Game Over mit Klang ✓ Rangliste gespeichert ✓ Musik-Schalter & Regler ✓. Alles grün? Dann lies bitte den nächsten Absatz im Stehen.

16.4 Du hast ein Spiel programmiert.

Kein Tutorial-Abklatsch, kein zusammenkopiertes Etwas: ein vollständiges, flüssiges, klingendes Tetris mit gut 700 Zeilen Kiste — jede davon von dir getippt, jede davon verstanden. Unterwegs hast du gelernt: Variablen, Konstanten, Entscheidungen, Schleifen, Funktionen, Listen, Module, Zufall, Dateien, Strings, ein Koordinatensystem, einen Game-Loop, zwei Tastatur-Modelle, eine Zustandsmaschine, ein Datei-Format, einen Sortier-Algorithmus, Widgets, Layouts, Sound — und den professionellen Umgang mit den Grenzen eines Werkzeugs. Das ist keine Anfängerliste. Das ist das Fundament, auf dem alles Weitere steht.

Wie geht's weiter?

Drei Wege, alle gut:

Hold-Funktion (Stein zwischenparken), ein zweiter Spielmodus, Lösch-Animationen, eigene Sounds, andere Farb-Themes. Du hast jetzt alle Werkzeuge — und die Aufgaben 14.3 war schon ein Vorgeschmack.

Snake? Breakout? Vier gewinnt? Mit Leinwand, Game-Loop, Zustandsmaschine und flachem Raster hast du den kompletten Werkzeugkasten. Die Beispiele im Kiste-Repo (`beispiele/gui_*.ki`) zeigen weitere Tricks.

Das große „Kiste — Das Lernbuch" behandelt, was Kistris nicht brauchte: Klassen und Objekte, Fehlerbehandlung, Karten, JSON, Netzwerk, Regex und die ganze Standard-Bibliothek.

Das Abschluss-Quiz — einmal quer durchs Buch

1. Welcher Index gehört zur Zelle (x=7, y=12) bei BREITE=10?
2. Warum laufen Bewegung und Rotation über verschiedene Tastatur-Mechanismen?
3. Nenne die drei Build-Subset-Rezepte, die Kistris beim Spielfeld benutzt.
4. Was garantiert der 7-Bag — und welche Kiste-Bausteine stecken in `fülle_beutel`?
5. Der Spieler drückt P mitten in der Partie. Beschreibe den Weg durch den Code bis zum „Pause"-Schriftzug auf dem Bildschirm.

A Lösungen zu Aufgaben und Quiz

A.1 Quiz-Antworten

Kapitel 1 — 1. Der Name stammt von meinem früheren Onlineshop-Projekt „Kiste"; der Shop wurde nie betrieben (zu viel Verwaltung), der schöne Name samt Domain blieb — und ging an das Herzensprojekt. 2. Nein: Kiste ist eine vollwertige General-Purpose-Sprache mit nativem Compiler, GUI, Klassen und Standard-Bibliothek — nur ihr Ökosystem ist jung. 3. André Ryser auf bitpoint.ch — komplett kostenlos, als Sponsor ohne Gegenleistung.

Kapitel 2 — 1. `kiste run spiel.ki`. 2. Eine eigenständige native `.exe`. 3. Die grafische Oberfläche läuft nur im nativ gebauten Programm.

Kapitel 3 — 1. `nimm` erstellt eine veränderbare Variable, `fest` eine Konstante. 2. `11` (Punkt vor Strich: 3 + 8). 3. `liste` ist ein reservierter Typ-Name. 4. Nur `+ - ×`.

Kapitel 4 — 1. Höchstens einer — mit `sonst` genau einer. 2. `=` weist zu statt zu vergleichen; richtig ist `==`. 3. Wenn mindestens eine der beiden Seiten wahr ist.

Kapitel 5 — 1. Zehnmal (0 bis 9, einschließlich). 2. *Vor* jedem Durchlauf. 3. `solange` — du weißt nicht, wie viele Zeilen frei sind; die Bedingung „noch keine Kollision" entscheidet.

Kapitel 6 — 1. Er wird nie ausgeführt — `gib` beendet die Funktion sofort. 2. `48 - 80 = -32`, das ist kleiner als 3 → Untergrenze: **3**. 3. Nur wenn die Schleife *keine* leere Zelle gefunden hat (und darum nie früh zurückkehrte), ist die Reihe wirklich voll.

Kapitel 7 — 1. Die flache Liste ist die übliche Profi-Lösung für ein 2-D-Raster (ein zusammenhängender Speicherblock, schneller Zugriff über `y · BREITE + x`) — so machen es auch C, Rust und Spiele-Engines. 2. Den Listen-Index der Zelle (x, y). 3. `entferne` lässt alle späteren Elemente nachrücken; das Voranstellen von BREITE Nullen schiebt alles um genau eine Zeilen-Länge nach hinten — optisch: eine Zeile nach unten. 4. Ein leeres `[]` kann der native Compiler keinem Element-Typ zuordnen.

Kapitel 8 — 1. 0, 1, 2, 3, 4, 5, 6 — sieben Werte. 2. Dass jeder Stein in je 7 Zügen genau einmal kommt — keine Dürren, keine Fluten. 3. Nur wenn er leer ist.

Kapitel 9 — 1. Oben links. 2. Fallen ist einfach `y = y + 1`. 3. Reines Rot mit halber Deckkraft (Alpha `80` hex ≈ 50 %). 4. Das größere helle Rechteck bleibt als Rand sichtbar — der Block bekommt Kontur.

Kapitel 10 — 1. Rund 60-mal. 2. `bei_taste` (diskret) — mit `taste_gedrückt` würde der Stein 60-mal pro Sekunde rotieren. 3. Er sammelt Ticks und löst erst beim Erreichen der Schwelle aus — so steuert eine Zahl das Tempo. 4. Über das Fenster-Handle: `gui.letzte_taste(f)`.

Kapitel 11 — 1. 32 = 4 Rotationen × 4 Blöcke × 2 Koordinaten. 2. Sie dreht einen Versatz um 90° im Uhrzeigersinn um den Drehpunkt. 3. Damit die Index-Formel und die `wert.als_ganz`-Lesebrücke an genau *einer* Stelle wohnen — ändert sich etwas (wie 2026-06-12, als `feld[idx] = w` nativ wurde), passt man eine Funktion an statt fünfzig Aufrufstellen. 4. Sie packt „geboxte" Werte aus mutierten Listen wieder aus; im `kiste run`-Modus von Kapitel 7 gibt es kein Boxing (und kein Modul `wert`).

Kapitel 12 — 1. Seitlich außerhalb? Unten außerhalb? Zielzelle belegt? 2. Damit der Zähler nach genau 3 Ticks wieder die 12 erreicht — das ergibt die schnelle Wiederholrate. 3. Nichts — die Drehung wird

verworfen. 4. Frisch gespawnte oder ganz oben gedrehte Steine ragen über den Feldrand; das muss erlaubt sein.

Kapitel 13 — 1. Nach dem Löschen ist die Zeile darüber nachgerutscht — sie könnte selbst voll sein. 2. `level = (reihen_gesamt div 10) + 1`: `div` ist die Ganzzahl-Division (ganze Stufen), während `/` das präzise Dezimal-Ergebnis (`2.5`) gäbe. 3. Schwerkraft gönnt den Lock-Delay (2 Schläge Gnadenfrist), der Hard Drop fixiert sofort.

Kapitel 14 — 1. Weil dieselbe Taste je Zustand etwas anderes bedeutet. 2. Ein halbtransparentes Rechteck (Acht-Stellen-Hex) über dem Feld — darunter bleibt alles sichtbar. 3. `zeichne_stein` zeichnet den aktuellen Stein an seiner Feld-Position; `zeichne_form` zeichnet einen beliebigen Typ frei in Pixeln (Vorschau, Titel-Deko). 4. `ghost_y` benutzt dieselbe `solange kollidiert…`-Idee wie `hard_drop` — das Ghost *ist* ein simulierter Hard Drop.

Kapitel 15 — 1. Fremde und veraltete Dateien werden erkannt und ignoriert statt als Datenmüll geladen. 2. Namen dürfen Leerzeichen und Kommas enthalten — in einer eigenen Zeile können sie nichts kaputt machen. 3. Jeder neue Eintrag landet von vornherein an der richtigen Stelle — die Sortierung wird nie verletzt, also muss nie „sortiert" werden. 4. Ein Eingabefeld mit Fokus würde während des Spiels alle Tasten fressen (Pfeile, Leertaste); die Namenseingabe nach dem Spiel umgeht das komplett — kein Widget, kein Fokus-Problem.

Kapitel 16 (Abschluss-Quiz) — 1. `12 · 10 + 7 = 127`. 2. Bewegung soll beim Halten fließend wiederholen (Polling + DAS im Tick), Rotation exakt einmal pro Druck feuern (diskreter Handler). 3. Flache Liste statt 2-D; Zell-Mutation per direkter Index-Zuweisung `feld[i] = w`; `[0]`-Aufbau für leere Listen — plus die `wert.als_ganz`-Lesebrücke. 4. Jeder Stein genau einmal pro Tüte; Bausteine: `solange`, `zufall.ganz`, `hänge_an`, `entferne`. 5. `bei_taste`-Handler feuert → `letzte_taste` liefert „P" → Zweig `modus == 1` → `modus = 2` → `zeichne()` → die Weiche sieht Modus 2 → zeichnet Feld + `zeichne_pause` → halbtransparentes Rechteck und Schriftzug liegen über dem Spielfeld.

A.2 Ausgewählte Aufgaben-Lösungen

3.2 — `15 · 2 + 300 · 4 = 30 + 1200 = 1230`:

```
nimm bonus = 15 * 2
nimm reihen_punkte = 300 * 4
sag "Gesamt: {bonus + reihen_punkte}"
```

4.1 —

```
nimm level = 7
wenn level <= 3 {
    sag "Anfänger"
} sonstwenn level <= 8 {
    sag "Fortgeschritten"
} sonst {
    sag "Tetris-Gott"
}
```

4.3 —

```
nimm antwort = frage("Wie viele Reihen (1-4)? ")
nimm n = als_ganz(antwort)
wenn n == 1 { sag "100 Punkte" }
sonstwenn n == 2 { sag "300 Punkte" }
sonstwenn n == 3 { sag "500 Punkte" }
sonst { sag "800 Punkte — Tetris!" }
```

5.3 (Rand-Variante) —

```
für y = 0 bis 3 {
    nimm zeile = ""
    für x = 0 bis 7 {
        wenn x == 0 oder x == 7 oder y == 0 oder y == 3 {
            zeile = "{zeile}▣"
        } sonst {
            zeile = "{zeile}□"
        }
    }
    sag zeile
}
```

6.2 —

```
funktion maximum(a, b) {
    wenn a >= b { gib a }
    gib b
}
```

6.4 — steht komplett in Kapitel 8 (`stein_name`). Wer hier spickt, hat technisch gesehen in die Zukunft gespickt. Respekt.

7.2 —

```
funktion zähle_blöcke() {
    nimm anzahl = 0
    für y = 0 bis HÖHE - 1 {
        für x = 0 bis BREITE - 1 {
            wenn feld_hole(x, y) != 0 { anzahl = anzahl + 1 }
        }
    }
    gib anzahl
}
```

7.4 —

```
funktion feld_leeren() {
    für i = 0 bis BREITE * HÖHE - 1 {
        feld[i] = 0
    }
}
```

8.2 — 70 Züge = exakt 10 volle Tüten, und jede Tüte enthält das I-Teil genau einmal. Zähl-Code:

```
nimm i_anzahl = 0
für i = 1 bis 70 {
    wenn ziehe_typ() == 1 { i_anzahl = i_anzahl + 1 }
}
sag "I-Teile: {i_anzahl}"     // immer 10
```

10.2 — eine Zeile im Game-Loop, vor der Schwerkraft:

```
wenn gui.taste_gedrückt(fenster, "runter") == 1 {
    fall_zaehler = fall_zaehler + 8
}
```

13.2 — Reihenfolge zählt! 2 Reihen → 300 · 1 = 300 (gesamt 2). 1 Reihe → 100 (gesamt 3). 4 Reihen → 800 (gesamt 7). 3 Reihen → 500 — *noch* mit Level 1 verbucht, denn der Aufstieg passiert nach der Punktevergabe (gesamt 10 → Level 2). Summe: **1700 Punkte, Level 2**.

15.3 — Die Schleife vergleicht: 0 < 350 → pos = 1; 198 < 350 → pos = 2; 408 < 350? Nein. 3682 < 350? Nein. Eingefügt wird an Position 2: `[0, 198, 350, 408, 3682]` — zwischen 198 und 408. Sortierung intakt.

B Fehlersuche — Fehlermeldungen verstehen

Fehlermeldungen sind keine Anklage, sondern eine Wegbeschreibung: *Datei, Zeile, Spalte, Problem*. Hier die Klassiker aus diesem Buch — was sie bedeuten und was zu tun ist.

Meldung (sinngemäß)	Ursache	Abhilfe
Parse-Fehler nahe `…` / unerwartetes Zeichen	Vergessene schließende Klammer `}`, fehlendes Anführungszeichen, Tippfehler in einem Schlüsselwort (`wen` statt `wenn`)	Genau an der angegebenen Zeile schauen — und eine Zeile *darüber*: oft fehlt dort das `}`.
`Modul "wert" kann nicht gefunden werden`	Du führst ein Programm mit `nutze wert` per `kiste run` aus — `wert` existiert nur im nativen Build (dort braucht man es; im Tree-Walker nicht).	Für Konsolen-Tests die `wert.als_…`-Aufrufe weglassen — oder nativ bauen.
`Listen-Element-Typ stimmt nicht: … Zuweisung liefert …`	Du schreibst einen fremden Typ in eine typisierte Liste, z. B. `zahlen[0] = "text"` (seit 2026-06-12 ist `l[i] = w` selbst nativ erlaubt).	Passenden Typ schreiben — oder bewusst eine `liste<beliebig>` verwenden (entsteht z. B. durch `liste.hänge_an`-Aufbau).
`10 / 3` ergibt `3.333…` statt `3`	`/` gibt immer das präzise Ergebnis, nie Ganzzahl-Truncation.	Ganze Stufen gewünscht? `div` nutzen (`10 div 3` → `3`).
Kein Fenster bei `kiste run`	GUI-Programme laufen nur nativ.	`kiste build` bzw. F5 in der IDE.
`Index … außerhalb` zur Laufzeit	Zugriff auf `liste[n]`, obwohl der letzte Index `n - 1` ist — die Null-Falle aus Kapitel 7.	Schleifengrenzen prüfen: `bis länge - 1`, nicht `bis länge`.
Programm hängt / Fenster friert ein	Endlos-Schleife — eine `solange`-Bedingung wird nie falsch.	Prüfen: Verändert der Schleifenrumpf wirklich die Bedingung? (Kapitel 5)
`kiste build` meldet fehlende Werkzeuge	Die blanke `kiste.exe` ohne die mitgelieferte Build-Werkzeugkiste (bei Installer/IDE ist sie dabei).	Einmal `kiste toolchain install` ausführen — siehe Kapitel 2.1.

Die universelle Fehlersuch-Strategie

1. Meldung *lesen* (wirklich lesen — Zeile und Spalte stehen drin). 2. Die letzte Änderung verdächtigen: Was hast du seit dem letzten funktionierenden Stand angefasst? 3. Mit `sag` Zwischenwerte ausgeben — „Was steht denn *wirklich* in `px`?" beantwortet eine Zeile `sag px` schneller als zehn Minuten Grübeln. 4. Im Zweifel: kleiner machen. Halbiere das Programm, bis der Fehler verschwindet — dann war er in der anderen Hälfte.

C Spickzettel — Kiste auf zwei Seiten

Sprachkern

```
// Variablen & Konstanten
nimm punkte = 0                  // veränderbar
fest BREITE = 10                 // Konstante
punkte = punkte + 100            // Zuweisung (ohne nimm)

// Texte
nimm name = "Sascha"
sag "Hallo {name}, {punkte} Punkte!"   // Interpolation
nimm n = länge(name)             // Textlänge
nimm zahl = als_ganz("42")       // Text → Zahl
nimm s = "Zeile 1\nZeile 2"      // \n = Zeilenumbruch

// Entscheidungen
wenn a == b { … }
sonstwenn a < b { … }
sonst { … }
// Vergleiche: == != < > <= >=   Verknüpfung: und, oder

// Schleifen
für i = 0 bis 9 { … }            // 10 Durchläufe, Ende einschließlich
für i = 10 bis 2 schritt -2 { … }
solange bedingung { … }          // prüft vor jedem Durchlauf

// Funktionen
funktion quadrat(x) {
    gib x * x                    // gib beendet sofort (frühes gib!)
}

// Konsole
sag wert                         // ausgeben
nimm e = frage("Prompt? ")       // einlesen
```

Listen (`nutze liste`)

```
nimm l = [1, 2, 3]               // Literal; Zugriff l[0] (ab 0!)
liste.länge(l)                   // Anzahl
liste.hänge_an(l, 4)             // hinten anhängen
liste.stelle_voran(l, 0)         // vorn einfügen
liste.entferne(l, i)             // Index i entfernen (alles rückt nach)
liste.füge_ein(l, i, w)          // an Index i einfügen
l[i] = w                         // Zell-Mutation: direkte Index-Zuweisung
l[i] += 1                        // … auch mit Rechen-Zuweisung
// 2-D-Raster: flache Liste, index = y * BREITE + x
```

Module für Kistris

```
nutze zufall
zufall.ganz(1, 7)                // Zufallszahl, Grenzen einschließlich

nutze datei
datei.existiert(pfad)            // 1/0
datei.zeilen(pfad)               // Liste der Zeilen
datei.schreibe(pfad, text)       // Datei (über)schreiben

nutze klang                      // (nur nativ)
nimm s = klang.lade("klaenge/x.wav")
klang.spiele(s)                  // einmal
klang.spiele_schleife(s)         // Endlos-Schleife
klang.stoppe(s)
klang.lautstärke(s, 80)          // 0–100

nutze wert                       // (nur nativ!)
wert.als_ganz(x)                 // geboxten Wert → Zahl
wert.als_text(x)                 // geboxten Wert → Text
```

GUI (`nutze gui` — baut nur nativ)

```
nimm bild = gui.leinwand(b, h)
gui.rechteck(bild, x, y, b, h, "#rrggbb")     // 8 Stellen = mit Alpha
gui.kreis(bild, x, y, r, farbe)
gui.linie(bild, x1, y1, x2, y2, farbe, dicke)
gui.text_an(bild, x, y, text, farbe, größe)
gui.leeren(bild)

gui.thema("dunkel")
nimm fenster = gui.fenster_öffne(titel, b, h)
gui.zeige(fenster, inhalt)
gui.starte()                                  // immer die letzte Zeile

// Zeit & Eingabe
gui.bei_zeit(16, funktion(t) { … })           // ~60×/Sekunde
gui.bei_taste(fenster, funktion(f) {
    nimm taste = gui.letzte_taste(f)          // Handle + Getter!
})
gui.taste_gedrückt(fenster, "links")          // 1 solange gehalten
gui.bei_maus_klick(bild, funktion(b) { … })
// Tastennamen: "links" "rechts" "hoch" "runter" "leer" "escape" "eingabe" "P"

// Widgets & Layout
nimm e = gui.eingabefeld("Platzhalter …")
gui.wert(e)                                   // Inhalt lesen
nimm et = gui.etikett_neu("Text")
nimm k = gui.kontrollkästchen("Musik")
gui.bei_umschaltung(k, handler)               // gui.angehakt(k) → 1/0
nimm r = gui.schieberegler(0, 100)
gui.bei_schieben(r, handler)                  // gui.schieber_wert(r)
```

```
gui.setze_schieber_wert(r, 80)
gui.vertikal([a, b])                          // untereinander
gui.raster(2, [a, b])                         // 2 Spalten
gui.rand_layout(mitte, [oben, unten, links, rechts])   // 0 = leer
```

Die Build-Subset-Rezepte (nativ)

Der native Build (`kiste build`) ist stark gewachsen — ganz-Division (`div` / `%`), Listen-Index-Zuweisung (`l[i] = w`), verschachtelte Listen und leeres `gib` sind inzwischen direkt dabei. Diese wenigen Grenzen bleiben — mit dem passenden Rezept:

Grenze	Rezept
leeres `[]` inferiert nicht	`[0]` anlegen, Platzhalter entfernen
mutierte Listen boxen Werte	`wert.als_ganz` / `wert.als_text` beim Lesen
Comparator-Sortierung begrenzt	Einfüge-Sortierung mit parallelen Listen

D Das komplette kistris.ki

Zum Nachschlagen und Abgleichen: das vollständige Spiel, exakt der Stand, den dieses Buch Schritt für Schritt aufgebaut hat — nativ gebaut und durchgespielt. (Auch zu finden im Kiste-Repository unter `beispiele/kistris/`, zusammen mit den Klang-Dateien.)

```
// kistris.ki — KISTRIS: Tetris in Kiste. Die Vorzeige-Demo zum Sprach-Launch.
//
// Ein vollständiger Tetris-Klon als eine einzige native .exe: 7 Steine mit
// Rotations-Tabelle, 7-Bag-Randomizer, Ghost Piece, Level-Tempo, Rangliste
// mit Namen (Datei-Persistenz), Sound und Zustandsmaschine.
//
// GUI baut nur nativ (F5 in der Kiste-IDE bzw. `kiste build`).
//
// Architektur-Notizen (verifizierte Build-Subset-Wege):
//   - Spielfeld = EINE flache Liste, Index y * BREITE + x (kein 2D im Subset).
//   - Zell-Mutation = direkte Index-Zuweisung `feld[i] = w` (seit Härtung
//     T.3; vorher der entferne+füge_ein-Umweg, O(n) pro Zelle).
//   - Reihen löschen = 10 Zellen der Reihe entfernen + 10 Nullen voranstellen.
//   - Level-Tempo = `reihen_gesamt div 10` — ganz-Division ist seit dem
//     Bug-014-Fix (2026-06-13) nativ im Build.
//   - Rangliste: Einfüge-Sortierung mit parallelen Listen (Snake-Rezept).
//
// modus: 0 = Hauptmenü · 1 = Spielt · 2 = Pause · 3 = Game-Over
nutze gui
nutze klang
nutze liste
nutze zufall
nutze wert
nutze datei
nutze text

// ---- Konstanten ------------------------------------------------------------

fest BREITE = 10
fest HÖHE = 20
fest ZELLE = 30
fest FELD_X = 20                 // linke obere Ecke des Spielfelds (Pixel)
fest FELD_Y = 40
fest BILD_B = 580                // Leinwand-Größe
fest BILD_H = 680
fest PANEL_X = 350               // Info-Spalte rechts neben dem Feld

// ---- Formen-Tabelle --------------------------------------------------------
// Pro Steintyp eine flache Liste: 4 Rotationen × 4 Blöcke × (x, y) = 32 Werte,
// relativ zum Drehpunkt. Rotation im Uhrzeigersinn: (x, y) → (-y, x).
// Typ → Farbe: 1=I cyan · 2=O gelb · 3=T violett · 4=S grün · 5=Z rot ·
//              6=J blau · 7=L orange

nimm FORM_I = [-1,0, 0,0, 1,0, 2,0,   0,-1, 0,0, 0,1, 0,2,
               -1,0, 0,0, 1,0, 2,0,   0,-1, 0,0, 0,1, 0,2]
nimm FORM_O = [0,0, 1,0, 0,1, 1,1,    0,0, 1,0, 0,1, 1,1,
               0,0, 1,0, 0,1, 1,1,    0,0, 1,0, 0,1, 1,1]
nimm FORM_T = [0,-1, -1,0, 0,0, 1,0,  0,-1, 0,0, 1,0, 0,1,
               -1,0, 0,0, 1,0, 0,1,   0,-1, -1,0, 0,0, 0,1]
nimm FORM_S = [0,-1, 1,-1, -1,0, 0,0, 0,-1, 0,0, 1,0, 1,1,
               0,-1, 1,-1, -1,0, 0,0, 0,-1, 0,0, 1,0, 1,1]
nimm FORM_Z = [-1,-1, 0,-1, 0,0, 1,0, 1,-1, 0,0, 1,0, 0,1,
               -1,-1, 0,-1, 0,0, 1,0, 1,-1, 0,0, 1,0, 0,1]
nimm FORM_J = [-1,-1, -1,0, 0,0, 1,0, 1,-1, 0,-1, 0,0, 0,1,
               -1,0, 0,0, 1,0, 1,1,   0,-1, 0,0, 0,1, -1,1]
nimm FORM_L = [1,-1, -1,0, 0,0, 1,0,  0,-1, 0,0, 0,1, 1,1,
               -1,0, 0,0, 1,0, -1,1,  -1,-1, 0,-1, 0,0, 0,1]

// Einen Tabellen-Wert lesen: k läuft 0..7 (4 Paare x,y) innerhalb der Rotation.
funktion form_wert(t, r, k) {
    nimm idx = r * 8 + k
    wenn t == 1 { gib FORM_I[idx] }
    wenn t == 2 { gib FORM_O[idx] }
    wenn t == 3 { gib FORM_T[idx] }
    wenn t == 4 { gib FORM_S[idx] }
    wenn t == 5 { gib FORM_Z[idx] }
```

```
    wenn t == 6 { gib FORM_J[idx] }
    gib FORM_L[idx]
}

funktion farbe_von(t) {
    wenn t == 1 { gib "#22d3ee" }
    wenn t == 2 { gib "#facc15" }
    wenn t == 3 { gib "#a855f7" }
    wenn t == 4 { gib "#4ade80" }
    wenn t == 5 { gib "#f87171" }
    wenn t == 6 { gib "#60a5fa" }
    gib "#fb923c"
}

funktion farbe_hell_von(t) {
    wenn t == 1 { gib "#a5f3fc" }
    wenn t == 2 { gib "#fef08a" }
    wenn t == 3 { gib "#d8b4fe" }
    wenn t == 4 { gib "#bbf7d0" }
    wenn t == 5 { gib "#fecaca" }
    wenn t == 6 { gib "#bfdbfe" }
    gib "#fed7aa"
}

// ---- Zustand ------------------------------------------------------------

nimm modus = 0
nimm bild = gui.leinwand(BILD_B, BILD_H)

// Klänge (WAV neben der .exe in klaenge/ – erzeugt von _erzeuge_klaenge.py)
nimm s_landung = klang.lade("klaenge/landung.wav")
nimm s_reihe = klang.lade("klaenge/reihe.wav")
nimm s_tetris = klang.lade("klaenge/tetris.wav")
nimm s_gameover = klang.lade("klaenge/gameover.wav")
nimm s_musik = klang.lade("klaenge/musik.wav")

// Kopfzeilen-Widgets – nur noch Musik + Lautstärke. Der Name wird NICHT mehr
// vorab getippt, sondern nach dem Spiel direkt auf dem Bildschirm (wie im
// Sternjäger): kein Eingabefeld, das dem Spiel die Tasten wegfrisst.
nimm musik_haken = gui.kontrollkästchen("Musik")
nimm lautstärke_regler = gui.schieberegler(0, 100)

// Spielfeld: flache Liste, 200 Zellen, 0 = leer, 1–7 = Farbindex.
// Leeres []-Literal inferiert im Build nicht – Aufbau über [0] + Schleife.
nimm feld = [0]
für i = 2 bis BREITE * HÖHE {
    liste.hänge_an(feld, 0)
}

// Aktueller Stein
nimm typ = 1
nimm rot = 0
nimm px = 4
nimm py = 1
nimm naechster_typ = 1

// Game-Loop-Zähler
nimm fall_zaehler = 0
nimm links_seit = 0              // Ticks, die "links" schon gehalten wird (DAS)
nimm rechts_seit = 0
nimm boden_seit = 0              // Lock-Delay: Schläge, die der Stein schon aufliegt
nimm neu_zeichnen = 0

// Punkte & Level
nimm punkte = 0
nimm level = 1
nimm reihen_gesamt = 0
nimm bestwert = 0

// Namenseingabe nach Game-Over (auf dem Bildschirm getippt).
nimm eingabe_name = ""
nimm gespeichert = 0             // 1 = Ergebnis dieser Partie schon eingetragen

// Rangliste: zwei parallele Listen, AUFSTEIGEND nach Punkten gehalten –
// Einfüge-Sortierung statt Comparator (Build-Subset). 0 = Platzhalter.
fest RANGDATEI = "kistris_rang.txt"
nimm rang_punkte = [0]
nimm rang_namen = [""]
```

```
// ---- Spielfeld-Zugriff (flache Liste) ---------------------------------------

// Zellwert lesen — wert.als_ganz, weil mutierte Listen geboxte Werte tragen.
funktion feld_hole(x, y) {
    gib wert.als_ganz(feld[y * BREITE + x])
}

// Zellwert schreiben — seit Härtung T.3 eine echte Index-Zuweisung
// (vorher: liste.entferne + liste.füge_ein, O(n) pro Zelle).
funktion feld_setze(x, y, w) {
    feld[y * BREITE + x] = w
}

funktion feld_leeren() {
    für i = 0 bis BREITE * HÖHE - 1 {
        feld[i] = 0
    }
}

// ---- Rangliste (mit Namen) --------------------------------------------------
// Datei-Format: Marker-Zeile "kistris-v1", dann zwei Zeilen pro Eintrag
// (Punkte, Name) — robust gegen Sonderzeichen im Namen, Fremddateien werden
// ignoriert. Die Listen bleiben durch Einfüge-Sortierung immer aufsteigend.

funktion rang(i) { gib wert.als_ganz(rang_punkte[i]) }
funktion rang_name(i) { gib wert.als_text(rang_namen[i]) }

funktion sortiert_einfügen(p, name) {
    nimm pos = 0
    für i = 0 bis liste.länge(rang_punkte) - 1 {
        wenn rang(i) < p { pos = i + 1 }
    }
    liste.füge_ein(rang_punkte, pos, p)
    liste.füge_ein(rang_namen, pos, name)
}

funktion aktualisiere_bestwert() {
    nimm n = liste.länge(rang_punkte)
    bestwert = rang(n - 1)
}

funktion lade_rangliste() {
    wenn datei.existiert(RANGDATEI) {
        nimm zeilen = datei.zeilen(RANGDATEI)
        nimm n = liste.länge(zeilen)
        wenn n > 0 und zeilen[0] == "kistris-v1" {
            für i = 1 bis n - 2 schritt 2 {
                wenn länge(zeilen[i]) > 0 {
                    sortiert_einfügen(als_ganz(zeilen[i]), zeilen[i + 1])
                }
            }
        }
    }
    aktualisiere_bestwert()
}

// Ergebnis eintragen, beste 8 behalten, Datei komplett neu schreiben.
funktion trage_ein(p, name) {
    sortiert_einfügen(p, name)
    solange liste.länge(rang_punkte) > 8 {     // beste 8 behalten — der 0-Platz-
        liste.entferne(rang_punkte, 0)         // halter ist der kleinste und fliegt
        liste.entferne(rang_namen, 0)          // als Erster vorne raus
    }
    nimm s = "kistris-v1\n"
    nimm n = liste.länge(rang_punkte)
    für i = 0 bis n - 1 {
        nimm r = rang(n - 1 - i)
        wenn r > 0 { s = "{s}{r}\n{rang_name(n - 1 - i)}\n" }
    }
    datei.schreibe(RANGDATEI, s)
    aktualisiere_bestwert()
}

funktion spieler_name() {
    wenn länge(eingabe_name) == 0 { gib "Anonym" }
    gib eingabe_name
}
```

```
// ---- Kollision & Bewegung ------------------------------------------------

// Kollidiert Stein t in Rotation r an Basisposition (bx, by)?
// Oberhalb des Felds (y < 0) ist frei – dort darf rotiert/gespawnt werden.
funktion kollidiert(t, r, bx, by) {
    für k = 0 bis 3 {
        nimm x = bx + form_wert(t, r, k * 2)
        nimm y = by + form_wert(t, r, k * 2 + 1)
        wenn x < 0 oder x >= BREITE { gib 1 }
        wenn y >= HÖHE { gib 1 }
        wenn y >= 0 {
            wenn feld_hole(x, y) != 0 { gib 1 }
        }
    }
    gib 0
}

funktion versuche_bewege(dx) {
    wenn kollidiert(typ, rot, px + dx, py) == 0 {
        px = px + dx
        neu_zeichnen = 1
    }
}

// Rotation mit Wall-Kick: erst direkt, dann ±1, für den I-Stein auch ±2.
funktion rotiere() {
    nimm nr = rot + 1
    wenn nr > 3 { nr = 0 }
    wenn kollidiert(typ, nr, px, py) == 0 {
        rot = nr
        neu_zeichnen = 1
    } sonstwenn kollidiert(typ, nr, px - 1, py) == 0 {
        px = px - 1
        rot = nr
        neu_zeichnen = 1
    } sonstwenn kollidiert(typ, nr, px + 1, py) == 0 {
        px = px + 1
        rot = nr
        neu_zeichnen = 1
    } sonstwenn kollidiert(typ, nr, px - 2, py) == 0 {
        px = px - 2
        rot = nr
        neu_zeichnen = 1
    } sonstwenn kollidiert(typ, nr, px + 2, py) == 0 {
        px = px + 2
        rot = nr
        neu_zeichnen = 1
    }
}

// ---- Reihen löschen & Verbuchen ------------------------------------------

funktion reihe_voll(y) {
    für x = 0 bis BREITE - 1 {
        wenn feld_hole(x, y) == 0 { gib 0 }
    }
    gib 1
}

// Volle Reihen entfernen: die 10 Zellen der Reihe aus der flachen Liste
// herausschneiden und 10 Nullen VORANSTELLEN – alles darüber rutscht damit
// automatisch eine Zeile nach unten. Dieselbe Zeile danach erneut prüfen.
funktion entferne_volle_reihen() {
    nimm geloescht = 0
    nimm y = HÖHE - 1
    solange y >= 0 {
        wenn reihe_voll(y) == 1 {
            für k = 1 bis BREITE { liste.entferne(feld, y * BREITE) }
            für k = 1 bis BREITE { liste.stelle_voran(feld, 0) }
            geloescht = geloescht + 1
        } sonst {
            y = y - 1
        }
    }
    gib geloescht
}
```

```
// Punkte gestaffelt (Tetris = überproportional), Level alle 10 Reihen.
funktion verbuche_reihen(n) {
    wenn n > 0 {
        nimm basis = 100
        wenn n == 2 { basis = 300 }
        wenn n == 3 { basis = 500 }
        wenn n >= 4 { basis = 800 }
        punkte = punkte + basis * level
        reihen_gesamt = reihen_gesamt + n
        level = (reihen_gesamt div 10) + 1
        wenn n >= 4 {
            klang.spiele(s_tetris)            // „Tetris!" – die große Fanfare
        } sonst {
            klang.spiele(s_reihe)
        }
    }
}

// Partie beendet: Klang, Ergebnis in die Rangliste, Overlay zeigen.
funktion game_over() {
    modus = 3
    gespeichert = 0
    eingabe_name = ""
    klang.spiele(s_gameover)
    neu_zeichnen = 1
}

// Ergebnis erst NACH der Namenseingabe eintragen (Enter im Game-Over-Bild).
funktion speichern() {
    wenn modus == 3 und gespeichert == 0 {
        wenn punkte > 0 { trage_ein(punkte, spieler_name()) }
        gespeichert = 1
        neu_zeichnen = 1
    }
}

// ---- 7-Bag-Randomizer ------------------------------------------------------
// Eine "Tüte" mit den Typen 1–7 in zufälliger Reihenfolge; ist sie leer, wird
// neu gefüllt. Fairer als reiner Zufall: nie 10× dasselbe Teil, nie ewige I-Dürre.

nimm beutel = [0]
liste.entferne(beutel, 0)        // leere, aber typisierte Liste

funktion fülle_beutel() {
    nimm kandidaten = [1, 2, 3, 4, 5, 6, 7]
    solange liste.länge(kandidaten) > 0 {
        nimm z = zufall.ganz(0, liste.länge(kandidaten) - 1)
        liste.hänge_an(beutel, wert.als_ganz(kandidaten[z]))
        liste.entferne(kandidaten, z)
    }
}

funktion ziehe_typ() {
    wenn liste.länge(beutel) == 0 { fülle_beutel() }
    nimm t = wert.als_ganz(beutel[0])
    liste.entferne(beutel, 0)
    gib t
}

// ---- Spawnen & Fixieren ----------------------------------------------------

funktion spawne() {
    typ = naechster_typ
    naechster_typ = ziehe_typ()
    rot = 0
    px = 4
    py = 1
    wenn typ == 1 { py = 0 }     // I-Stein liegt flach – eine Zeile höher starten
    wenn kollidiert(typ, rot, px, py) == 1 {
        game_over()              // sofort blockiert → Partie vorbei
    }
    neu_zeichnen = 1
}

// Stein ins Feld schreiben; ragt er oben hinaus → Game-Over.
funktion fixiere() {
    nimm oben_raus = 0
    für k = 0 bis 3 {
```

```
        nimm x = px + form_wert(typ, rot, k * 2)
        nimm y = py + form_wert(typ, rot, k * 2 + 1)
        wenn y < 0 {
            oben_raus = 1
        } sonst {
            feld_setze(x, y, typ)
        }
    }
    nimm n = entferne_volle_reihen()
    verbuche_reihen(n)
    wenn n == 0 { klang.spiele(s_landung) }    // sonst klingt schon die Reihe
    wenn oben_raus == 1 {
        game_over()
    } sonst {
        spawne()
    }
    neu_zeichnen = 1
}

// Wo käme der Stein bei einem Hard Drop an? (Für das Ghost Piece.)
funktion ghost_y() {
    nimm gy = py
    solange kollidiert(typ, rot, px, gy + 1) == 0 {
        gy = gy + 1
    }
    gib gy
}

// Eine Zeile fallen; geht es nicht, erst nach kurzem Lock-Delay fixieren –
// der klassische Moment, um den Stein noch unter eine Kante zu schieben.
funktion versuche_fallen() {
    wenn kollidiert(typ, rot, px, py + 1) == 0 {
        py = py + 1
        boden_seit = 0
        neu_zeichnen = 1
    } sonst {
        boden_seit = boden_seit + 1
        wenn boden_seit >= 2 {       // ~2 Schwerkraft-Schläge Gnadenfrist
            boden_seit = 0
            fixiere()
        }
    }
}

funktion hard_drop() {
    solange kollidiert(typ, rot, px, py + 1) == 0 {
        py = py + 1
        punkte = punkte + 2      // kleiner Bonus pro übersprungener Zeile
    }
    fixiere()
}

// Tempo pro Level: Ticks (à 16 ms) pro Fall-Zeile – kleiner = schneller.
funktion fall_schwelle() {
    nimm s = 48 - level * 4
    wenn s < 3 { s = 3 }
    gib s
}

// ---- Zeichnen ----------------------------------------------------------

// Ein Block: heller Rahmen + satte Füllung – flacher Look mit Kontur.
funktion zeichne_block(xp, yp, gr, t) {
    gui.rechteck(bild, xp, yp, gr - 2, gr - 2, farbe_hell_von(t))
    gui.rechteck(bild, xp + 2, yp + 2, gr - 6, gr - 6, farbe_von(t))
}

funktion zeichne_feld_rahmen() {
    gui.rechteck(bild, FELD_X - 4, FELD_Y - 4, BREITE * ZELLE + 8, HÖHE * ZELLE + 8, "#334155")
    gui.rechteck(bild, FELD_X - 2, FELD_Y - 2, BREITE * ZELLE + 4, HÖHE * ZELLE + 4, "#0b1120")
    // dezentes Gitter – gibt dem Feld Tiefe, ohne von den Steinen abzulenken
    für x = 1 bis BREITE - 1 {
        gui.linie(bild, FELD_X + x * ZELLE, FELD_Y, FELD_X + x * ZELLE, FELD_Y + HÖHE * ZELLE, "#1e293b", 1)
    }
    für y = 1 bis HÖHE - 1 {
        gui.linie(bild, FELD_X, FELD_Y + y * ZELLE, FELD_X + BREITE * ZELLE, FELD_Y + y * ZELLE, "#1e293b", 1)
    }
}
```

```
funktion zeichne_feld_inhalt() {
    für y = 0 bis HÖHE - 1 {
        für x = 0 bis BREITE - 1 {
            nimm z = feld_hole(x, y)
            wenn z != 0 {
                zeichne_block(FELD_X + x * ZELLE + 1, FELD_Y + y * ZELLE + 1, ZELLE, z)
            }
        }
    }
}

funktion zeichne_stein() {
    für k = 0 bis 3 {
        nimm x = px + form_wert(typ, rot, k * 2)
        nimm y = py + form_wert(typ, rot, k * 2 + 1)
        wenn y >= 0 {
            zeichne_block(FELD_X + x * ZELLE + 1, FELD_Y + y * ZELLE + 1, ZELLE, typ)
        }
    }
}

// Ghost Piece: halbtransparente Landeposition (Hex mit Alpha) - rein visuell.
funktion zeichne_ghost() {
    nimm gy = ghost_y()
    wenn gy > py {
        nimm f = farbe_von(typ)
        nimm geist = "{f}40"
        für k = 0 bis 3 {
            nimm x = px + form_wert(typ, rot, k * 2)
            nimm y = gy + form_wert(typ, rot, k * 2 + 1)
            wenn y >= 0 {
                gui.rechteck(bild, FELD_X + x * ZELLE + 2, FELD_Y + y * ZELLE + 2, ZELLE - 4, ZELLE - 4, geist)
            }
        }
    }
}

// Einen Steintyp frei auf der Leinwand zeichnen (Vorschau, Titel-Deko).
// (pvx, pvy) = Pixel-Position des Drehpunkts, gr = Blockgröße.
funktion zeichne_form(t, r, pvx, pvy, gr) {
    für k = 0 bis 3 {
        nimm x = pvx + form_wert(t, r, k * 2) * gr
        nimm y = pvy + form_wert(t, r, k * 2 + 1) * gr
        zeichne_block(x, y, gr, t)
    }
}

funktion zeichne_vorschau() {
    gui.text_an(bild, PANEL_X, FELD_Y + 310, "Nächster", "#94a3b8", 16)
    gui.rechteck(bild, PANEL_X, FELD_Y + 338, 130, 96, "#1e293b")
    // Drehpunkt so versetzen, dass jede Form optisch mittig sitzt (gr = 22):
    nimm ox = PANEL_X + 54          // 3 Zellen breit (T/S/Z/J/L)
    nimm oy = FELD_Y + 338 + 48
    wenn naechster_typ == 1 {       // I: 4 breit, 1 hoch
        ox = PANEL_X + 43
        oy = FELD_Y + 338 + 37
    }
    wenn naechster_typ == 2 {       // O: 2 × 2 um den Drehpunkt rechts-unten
        ox = PANEL_X + 43
        oy = FELD_Y + 338 + 26
    }
    zeichne_form(naechster_typ, 0, ox, oy, 22)
}

funktion zeichne_panel() {
    gui.text_an(bild, PANEL_X + 2, FELD_Y + 2, "KISTRIS", "#0c4a6e", 34)   // Schatten
    gui.text_an(bild, PANEL_X, FELD_Y, "KISTRIS", "#38bdf8", 34)
    gui.text_an(bild, PANEL_X, FELD_Y + 70, "Punkte", "#94a3b8", 16)
    gui.text_an(bild, PANEL_X, FELD_Y + 92, "{punkte}", "#e4e4e7", 26)
    gui.text_an(bild, PANEL_X, FELD_Y + 150, "Level", "#94a3b8", 16)
    gui.text_an(bild, PANEL_X, FELD_Y + 172, "{level}", "#e4e4e7", 26)
    gui.text_an(bild, PANEL_X, FELD_Y + 230, "Reihen", "#94a3b8", 16)
    gui.text_an(bild, PANEL_X, FELD_Y + 252, "{reihen_gesamt}", "#e4e4e7", 26)
    zeichne_vorschau()
    gui.text_an(bild, PANEL_X, FELD_Y + 460, "Bestwert", "#94a3b8", 16)
    gui.text_an(bild, PANEL_X, FELD_Y + 482, "{bestwert}", "#fbbf24", 26)
    gui.text_an(bild, PANEL_X, FELD_Y + 540, "P  Pause", "#64748b", 14)
```

```
    gui.text_an(bild, PANEL_X, FELD_Y + 560, "Esc  Menü", "#64748b", 14)
}

// Die besten Einträge der Rangliste an Position (x, y) – anzahl Zeilen.
funktion zeichne_rangliste(x, y, anzahl) {
    gui.text_an(bild, x + 40, y, "Rangliste", "#fbbf24", 22)
    nimm n = liste.länge(rang_punkte)
    nimm zeile = 0
    für i = 0 bis n - 1 {
        nimm p = rang(n - 1 - i)
        wenn p > 0 und zeile < anzahl {
            zeile = zeile + 1
            gui.text_an(bild, x, y + 12 + zeile * 26, "{zeile}.  {rang_name(n - 1 - i)} – {p}", "#e4e4e7", 16)
        }
    }
    wenn zeile == 0 {
        gui.text_an(bild, x, y + 40, "noch keine Einträge", "#94a3b8", 15)
    }
}

// Hauptmenü: Logo, alle 7 Steine als Farbband, Rangliste, Steuerung.
funktion zeichne_titel() {
    gui.text_an(bild, 153, 63, "KISTRIS", "#0c4a6e", 64)                    // Schatten
    gui.text_an(bild, 148, 58, "KISTRIS", "#38bdf8", 64)
    gui.text_an(bild, 175, 140, "Tetris, gebaut in Kiste", "#94a3b8", 18)
    // Deko: die sieben Steine in ihren Farben, einmal quer übers Bild.
    nimm deko_x = 62
    für t = 1 bis 7 {
        zeichne_form(t, 0, deko_x, 215, 17)
        deko_x = deko_x + 70
    }
    zeichne_rangliste(195, 280, 8)
    gui.text_an(bild, 130, 530, "links/rechts bewegen  ·  hoch drehen  ·  runter sinken", "#94a3b8", 15)
    gui.text_an(bild, 152, 554, "Leertaste Hard Drop  ·  P Pause  ·  Esc Menü", "#94a3b8", 15)
    gui.text_an(bild, 74, 592, "Tipp: Nach dem Spiel tippst du deinen Namen für die Rangliste.", "#64748b", 14)
    gui.text_an(bild, 173, 632, "Leertaste oder Klick = Start", "#fbbf24", 22)
}

funktion zeichne_pause() {
    gui.rechteck(bild, FELD_X, FELD_Y, BREITE * ZELLE, HÖHE * ZELLE, "#0f172acc")
    gui.text_an(bild, FELD_X + 95, FELD_Y + 250, "Pause", "#fbbf24", 32)
    gui.text_an(bild, FELD_X + 93, FELD_Y + 302, "P = weiter", "#94a3b8", 18)
}

funktion zeichne_vorbei() {
    gui.rechteck(bild, FELD_X, FELD_Y, BREITE * ZELLE, HÖHE * ZELLE, "#0f172add")
    gui.rechteck(bild, FELD_X + 10, FELD_Y + 90, BREITE * ZELLE - 20, 400, "#1e293b")
    gui.text_an(bild, FELD_X + 58, FELD_Y + 110, "Game Over", "#f87171", 34)
    gui.text_an(bild, FELD_X + 95, FELD_Y + 165, "Punkte: {punkte}", "#e4e4e7", 20)
    wenn gespeichert == 1 {
        zeichne_rangliste(FELD_X + 55, FELD_Y + 215, 6)
        gui.text_an(bild, FELD_X + 48, FELD_Y + 440, "Leertaste = neue Partie", "#e4e4e7", 17)
        gui.text_an(bild, FELD_X + 73, FELD_Y + 464, "Eingabe = Hauptmenü", "#94a3b8", 15)
    } sonst {
        // Namenseingabe – direkt auf dem Bildschirm getippt (kein Widget).
        gui.text_an(bild, FELD_X + 55, FELD_Y + 225, "Gib deinen Namen ein:", "#fbbf24", 18)
        gui.rechteck(bild, FELD_X + 40, FELD_Y + 260, 220, 40, "#0b1120")
        gui.text_an(bild, FELD_X + 52, FELD_Y + 270, "{eingabe_name}|", "#e4e4e7", 20)
        gui.text_an(bild, FELD_X + 58, FELD_Y + 332, "Enter = Name speichern", "#94a3b8", 15)
        gui.text_an(bild, FELD_X + 92, FELD_Y + 358, "Esc = Hauptmenü", "#64748b", 14)
    }
}

funktion zeichne() {
    gui.leeren(bild)
    gui.rechteck(bild, 0, 0, BILD_B, BILD_H, "#0f172a")
    wenn modus == 0 {
        zeichne_titel()
    } sonst {
        zeichne_feld_rahmen()
        zeichne_feld_inhalt()
        wenn modus == 1 { zeichne_ghost() }
        zeichne_stein()
        zeichne_panel()
        wenn modus == 2 { zeichne_pause() }
        wenn modus == 3 { zeichne_vorbei() }
    }
}
```

```
// ---- Spielsteuerung ------------------------------------------------------

funktion starte_spiel() {
    feld_leeren()
    punkte = 0
    level = 1
    reihen_gesamt = 0
    fall_zaehler = 0
    boden_seit = 0
    solange liste.länge(beutel) > 0 {    // frische Tüte für die neue Partie
        liste.entferne(beutel, 0)
    }
    naechster_typ = ziehe_typ()
    spawne()
    modus = 1
    zeichne()
}

// Hintergrundmusik an/aus – Kontrollkästchen in der Kopfzeile.
funktion bei_musik(k) {
    wenn gui.angehakt(k) == 1 {
        klang.spiele_schleife(s_musik)
    } sonst {
        klang.stoppe(s_musik)
    }
}

// Master-Lautstärke für alle Spielklänge (Regler in der Kopfzeile).
funktion bei_lautstärke(s) {
    nimm w = gui.schieber_wert(s)
    klang.lautstärke(s_musik, w)
    klang.lautstärke(s_landung, w)
    klang.lautstärke(s_reihe, w)
    klang.lautstärke(s_tetris, w)
    klang.lautstärke(s_gameover, w)
}

lade_rangliste()
zeichne()

gui.thema("dunkel")
nimm fenster = gui.fenster_öffne("Kistris", BILD_B + 1, BILD_H + 60)

// Klick aufs Spielfeld: startet vom Titel / nach Game-Over – und holt nebenbei
// den Fokus vom Namensfeld zurück (Eingabefeld-Fokus frisst sonst Spieltasten).
gui.bei_maus_klick(bild, funktion(b) {
    wenn modus == 0 { starte_spiel() }
    sonstwenn modus == 3 und gespeichert == 1 { starte_spiel() }
})

gui.bei_umschaltung(musik_haken, bei_musik)
gui.setze_schieber_wert(lautstärke_regler, 80)
gui.bei_schieben(lautstärke_regler, bei_lautstärke)

// Diskrete Tasten: Rotation, Hard Drop, Pause, Zustandswechsel.
gui.bei_taste(fenster, funktion(f) {
    nimm taste = gui.letzte_taste(f)
    wenn modus == 1 {
        wenn taste == "hoch" { rotiere() }
        wenn taste == "leer" { hard_drop() }
        wenn taste == "p" oder taste == "P" {
            modus = 2
        }
        wenn taste == "escape" {
            modus = 0
        }
        zeichne()
    } sonstwenn modus == 2 {
        wenn taste == "p" oder taste == "P" oder taste == "escape" oder taste == "leer" {
            modus = 1
            zeichne()
        }
    } sonstwenn modus == 3 {
        wenn gespeichert == 0 {
            // Namenseingabe: Buchstaben tippen, Enter speichert, Esc = Menü.
            wenn taste == "eingabe" {
                speichern()
```

```
            } sonstwenn taste == "escape" {
                modus = 0
            } sonstwenn taste == "BackSpace" {
                wenn länge(eingabe_name) > 0 {
                    eingabe_name = text.erste_n(eingabe_name, länge(eingabe_name) - 1)
                }
            } sonstwenn taste == "leer" {
                wenn länge(eingabe_name) < 14 { eingabe_name = "{eingabe_name} " }
            } sonstwenn länge(taste) == 1 {
                wenn länge(eingabe_name) < 14 { eingabe_name = "{eingabe_name}{taste}" }
            }
        } sonst {
            // Schon gespeichert: neue Partie oder zurück ins Menü.
            wenn taste == "leer" { starte_spiel() }
            sonstwenn taste == "eingabe" oder taste == "escape" { modus = 0 }
        }
        zeichne()
    } sonstwenn modus == 0 {
        wenn taste == "leer" oder taste == "eingabe" { starte_spiel() }
    }
})

// Game-Loop: 16 ms ≈ 60 fps. Gehaltene Tasten (DAS) + Schwerkraft-Akkumulator.
gui.bei_zeit(16, funktion(t) {
    wenn modus == 1 {
        // Links/Rechts: erster Druck sofort, nach 12 Ticks Auto-Wiederholung alle 3.
        wenn gui.taste_gedrückt(fenster, "links") == 1 {
            links_seit = links_seit + 1
            wenn links_seit == 1 { versuche_bewege(-1) }
            wenn links_seit >= 12 {
                versuche_bewege(-1)
                links_seit = 9
            }
        } sonst {
            links_seit = 0
        }
        wenn gui.taste_gedrückt(fenster, "rechts") == 1 {
            rechts_seit = rechts_seit + 1
            wenn rechts_seit == 1 { versuche_bewege(1) }
            wenn rechts_seit >= 12 {
                versuche_bewege(1)
                rechts_seit = 9
            }
        } sonst {
            rechts_seit = 0
        }
        // Soft Drop: Schwerkraft kräftig beschleunigen, solange "runter" gehalten.
        wenn gui.taste_gedrückt(fenster, "runter") == 1 {
            fall_zaehler = fall_zaehler + 8
        }
        // Schwerkraft per Akkumulator — Tempo steuert fall_schwelle() (Level).
        fall_zaehler = fall_zaehler + 1
        wenn fall_zaehler >= fall_schwelle() {
            fall_zaehler = 0
            versuche_fallen()
        }
    }
    // Auch Zustandswechsel aus dem Loop heraus (z. B. Game-Over beim Fixieren)
    // sollen sofort sichtbar werden — deshalb außerhalb von modus == 1.
    wenn neu_zeichnen == 1 {
        zeichne()
        neu_zeichnen = 0
    }
})

// Kopfzeile: links der Musik-Haken, daneben der Lautstärke-Regler, der sich in
// die Breite dehnt (Mitte eines Rand-Layouts). Kein Namensfeld mehr — der Name
// wird nach dem Spiel auf dem Bildschirm getippt.
nimm kopf = gui.rand_layout(lautstärke_regler, [0, 0, musik_haken, 0])
nimm inhalt = gui.vertikal([kopf, bild])
gui.zeige(fenster, inhalt)
gui.starte()
```

Ende. Und jetzt: nur noch eine Reihe.

Kiste programmieren lernen mit Kistris · 1. Auflage 2026
Für die Programmiersprache Kiste von Sascha Rust · Mit Dank an André Ryser (bitpoint.ch)
Geschrieben von Sascha Rust · Jeder Code real ausgeführt & nativ gebaut