1. Einführung

Modellbildung

- **Modell**: Vereinfachte Darstellung eines Systems
- **Simulation**: Virtuelles Experiment

Mathematische Modellierung

- Features, Größen (Objekte z.B. Staub, Wandposter ; Parameter z.B. Temperatur, Druck)
- Wechselwirkungen (Temperatur auf Personen im Raum; Druck und Staub)
- Eindeutige Lösung: Existiert eine Lösung und ist sie eindeutig?
- Abhängigkeit von Initialwerten: Je nachdem wo man in einem Shooter startet, sind die Überlebenschancen unterschiedlich gut.
- Genauigkeit / Auflösung: Personenströme als Flüssigkeit modellieren, kann nicht eine einzelne Person darstellen.

Modellarten

- **Diskret**: In Schritten Simuliert
- **Kontinuierlich**: In Echtzeit / stetig Simuliert
- **Deterministisch**: Immer das gleiche Ergebnis, mit den gleichen Anfangsbedingungen
- **Stochastisch**: Zufällige Ergebnisse, auch mit den gleichen Anfangsbedingungen (z.B. Würfel)
- **Inverse**: Ausgangsbedingungen aus Ergebnissen bestimmen oder Ergebnisse aus Ausgangsbedingungen bestimmen

Skalen

- 1D, 2D, 3D, so einfach wie möglich, so komplex wie nötig
- Abstraktionsebene: Menschen, Zellen oder Moleküle, oder gar Atome

Diskretitierung

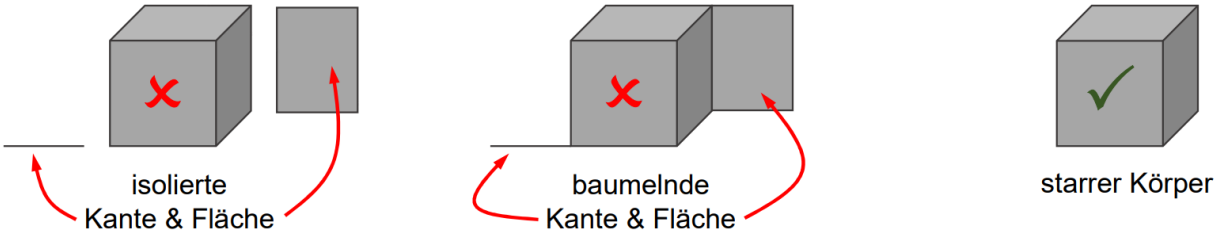
- Gitterbreite h (delta zwischen punkten)

2. Geometrische Modellierung

Rigid Body Anforderungen

- keine losen Teile (Zusammenhängend)
- keine 1D-Teile (z.B Linie, Punkt)
- Es muss ein Innen und Außen geben (Volumen)
- keine Löcher
- keine Überlappungen

- Orientierbar (Außen und Innen unterscheidbar)
- Manigfaltig möglich (Zwei körper teilen sich eine Kante)

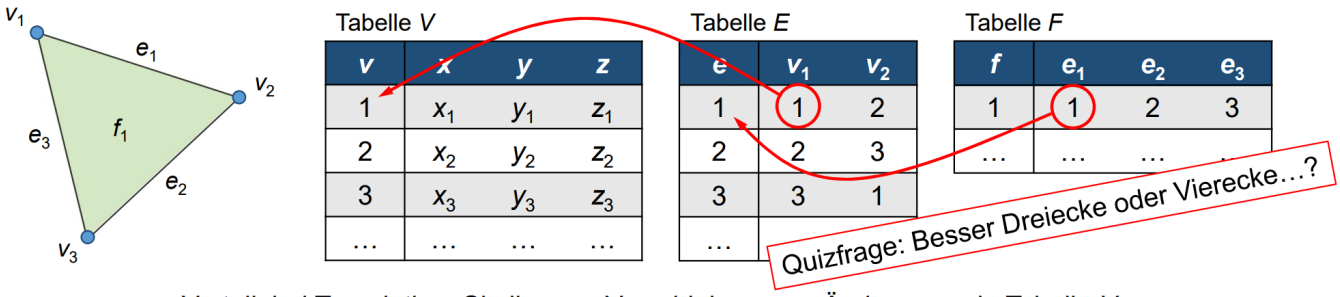


Darstellungsformen (Speicherung)

- Vertecies (Ecken), Edges (Kanten), Faces (Flächen)
- $n_{v} - n_{e} + n_{f} = 2$ (Euler-Formel) -> Gilt für Rechtecke und Dreiecke

vef-Graph

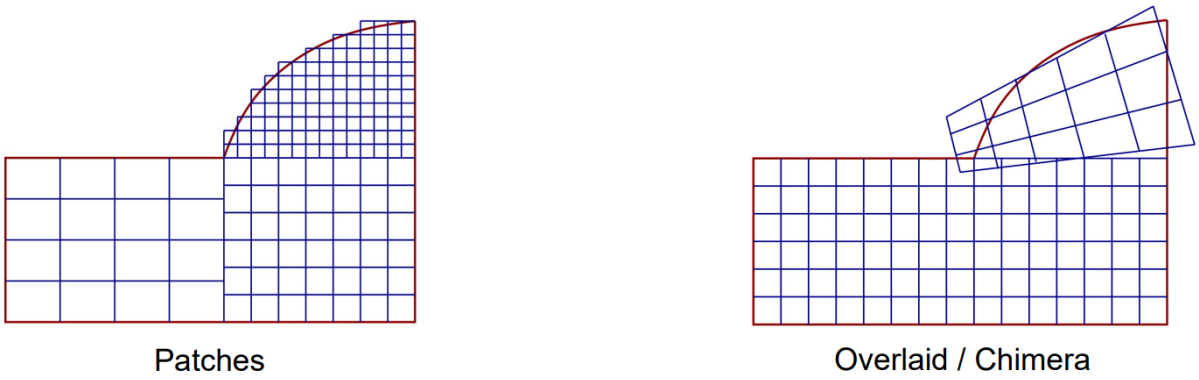
> Es werden nur die Koordinaten der **Eckpunkte** gespeichert, Kanten und Flächen speichern referenzen. Dadurch werden änderungen an den Eckpunkten automatisch in den Kanten und Flächen übernommen.
> Eine **Kante** wird durch zwei Eckpunkte definiert.
> Eine **Fläche** durch drei Kanten. \



⇒ Vorteil: bei Translation, Skalierung, Verschiebung nur Änderungen in Tabelle *V*

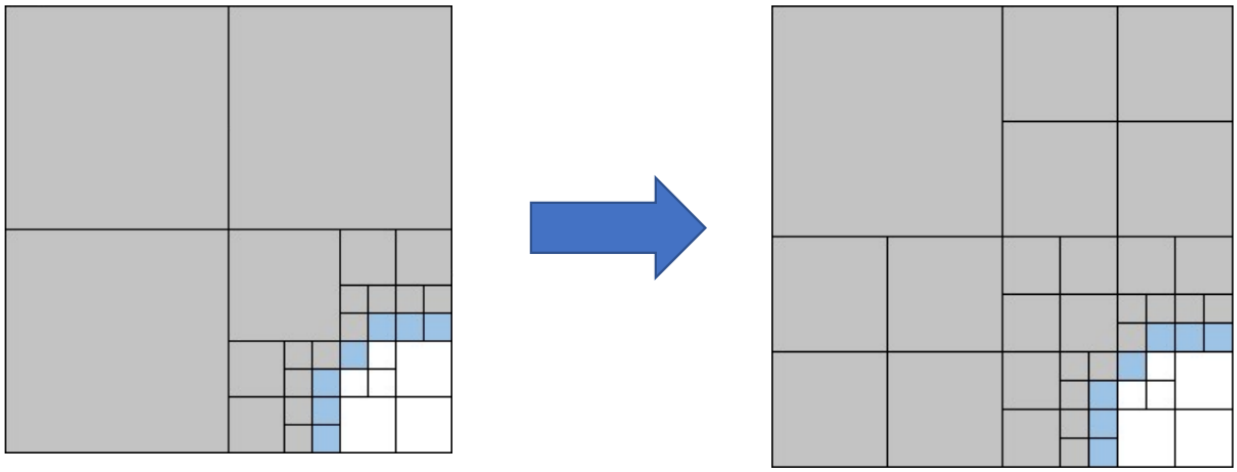
_

Volumenmodell



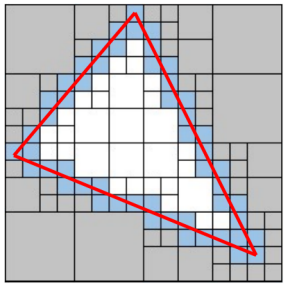
Spacetree

- **Balanzierung**: Jeder Knoten entweder **0** oder **4** Kinder (Eine Fläche 2 hat Angrezende Flächen)

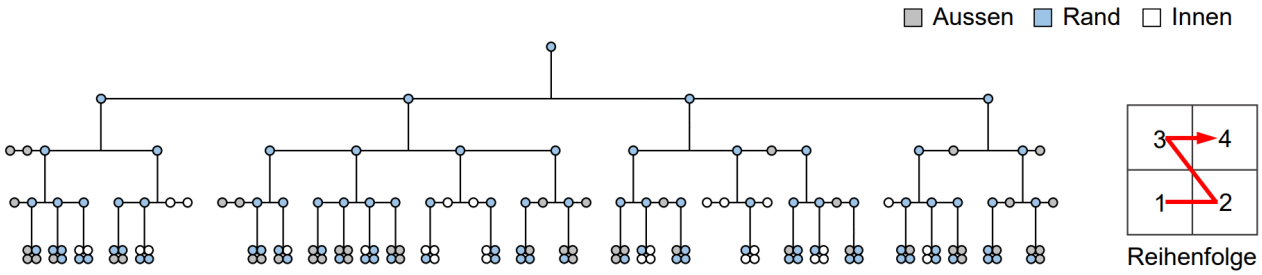


Verschiedene Darstellungsformen

- direkte Darstellung (Fortsetzung)
  - unstrukturierte Ansätze: Spacetrees
    ⇒ auch bekannt als Quadtrees (2D) oder Octrees (3D)
  - Idee: rekursive Substrukturierung
    ⇒ Zerlegung eines Voxels (Zelle) in 2^d kongruente Voxel
  - hierarchische Datenstruktur (⇒ Baum)

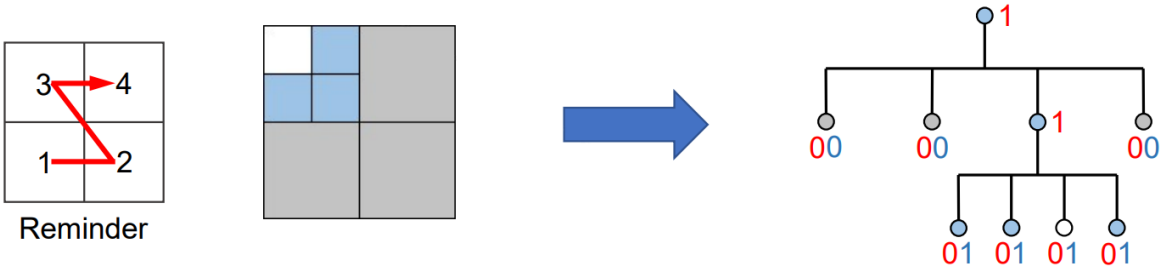


Spacetree (2D)

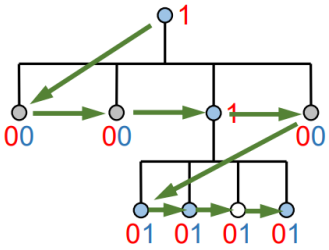


Codierung

- Linearisierung von Spacetrees zur Speicherung
- Codierung Knoten: 1 → Elterknoten (mit Kindern) und 0 → Kindknoten (Blatt)
- Codierung Farbe: 1 → Blatt Rand / Innen und 0 → Blatt Aussen

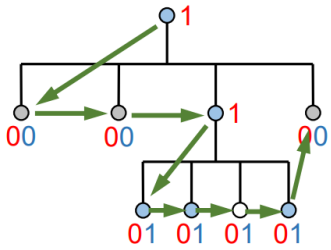


Linearisierung



Breadth-First

⇒ Ausgabe: 1 00 00 1 00 01 01 01 01



Depth-First

⇒ Ausgabe: 1 00 00 1 01 01 01 01 00

3. Spieltheorie

Definition

- **Gewissheit**: Konsequenzen zu Aktionen bekannt
- **Risiko**: Wahrscheinlichkeit von Konsequenz zu Aktion teilweise bekannt
- **Unsicherheit**:keine Konsequenz zu Aktion bekannt

Modellierung

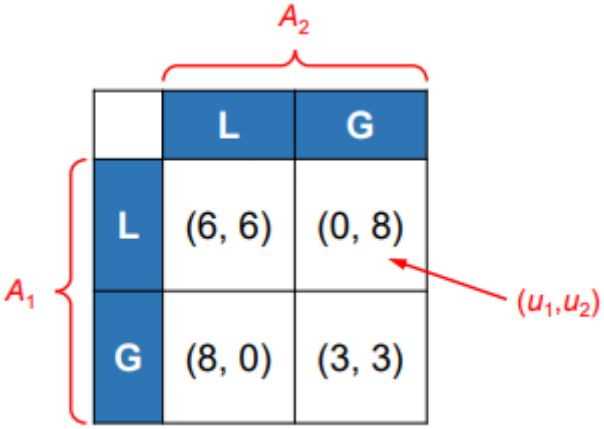
- Numbers of Players $N={1,2, \ldots, n}$
- Anzahl Aktionen (Strategien)$A_i\qquad i\in \mathbb{N}$
- Menge Aktionsprofile (Strategieprofile) $A={\left(a_i \right)_{i\in \mathbb{N}}, a_i\in A_i, i\in\mathbb{N} }$
- Auszahlungsfunktion $u_i: A\rightarrow\mathbb{R}$

Unterteilung

- **Kooperation**
  - **kooperative Spiele**: Spieler kooperieren müssen um zu gewinnen.
  - **nicht-kooperative Spiele**: Spieler konkurrieren um zu gewinnen / spielen gegeneinander. (z.B. nur einer kann gewinnen)
- **Serie**
  - **simultane Spiele**: In einem Durchlauf treffen alle für sich gleichzeitig die Wahl.
  - **sequentielle Spiele**: Regeln wer wann Entscheiden darf.Sequenzielle Abfolge.

Darstellung

- Bimatrixform / Normalform



- Gewinn wird angezeigt (u_i, u_i)

Reine Strategien

Gefangenen-Dilema

- Zwei Subjects $A_1\textrm{ und }A_2$
- Mögliche Aktionen
  - **Leugnen L**: Niedrigere Strafe (2 Jahre)
  - **Gestehen G**: Volle Strafe 5 (Kronzeuge) oder 8 Jahre

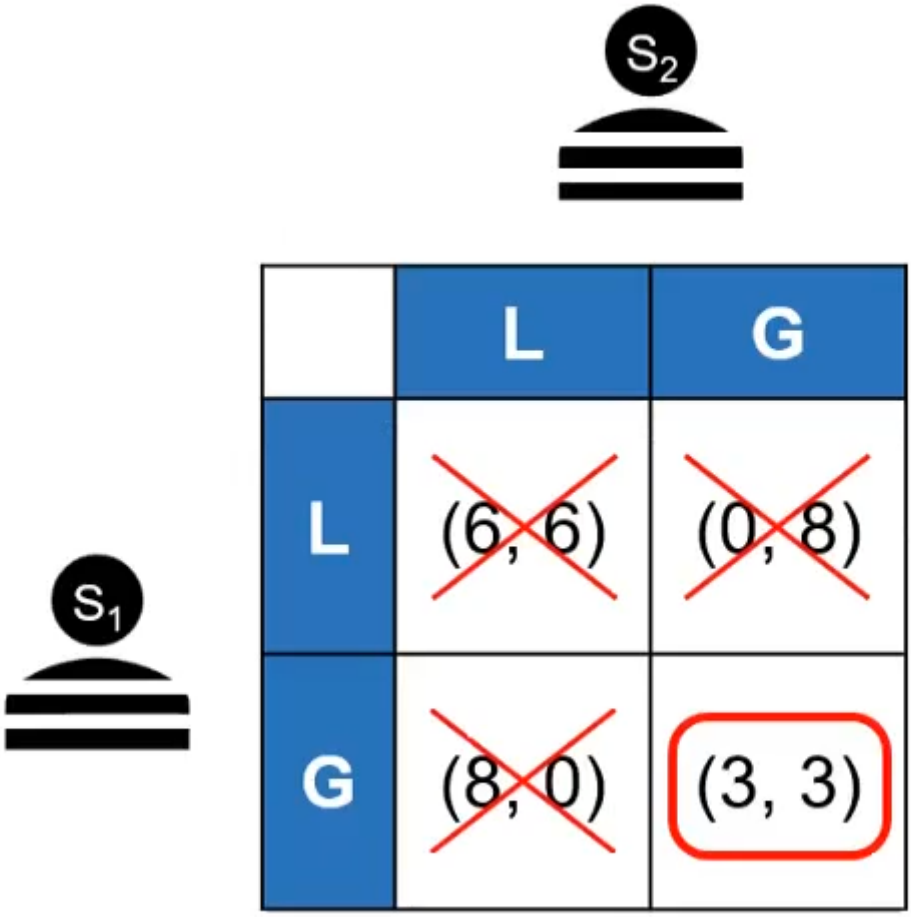
Nash-Gleichgewicht

> Ist das Optimale Ergebnis für **alle** Spieler

- Wenn ein Spieler durch Ändern seiner Strategie, seinen Gewinn erhöhen kann, ist der aktuelle Zustand nicht im Nash-Gleichgewicht.

Vorsicht!:

- Jede Zelle muss einzeln betrachtet werden
- WICHTIG: Nicht mit der gleichen Strategie vergleichen
- Nash-Gleichgewicht muss für BEIDE Spieler gelten



Dominante Strategien

> Eine Strategie die besser oder gleich gut ist wie alle anderen Strategien (>=) **Beispiel:** Sicht von S1:

- S2 nimmt Leugnen: Nimmt S1 Gestehen = 8, nimmt S1 Leugnen = 6. 8 >= 6
- S2 nimmt Gestehen: Nimmt S1 Gestehen = 3, nimmt S1 Leugnen = 0. 3 >= 0
- Kreuzen sich Dominante Strategien, ist dort das Nash-Gleichgewicht (aber nicht umgekehrt)

Stark Dominante Strategien

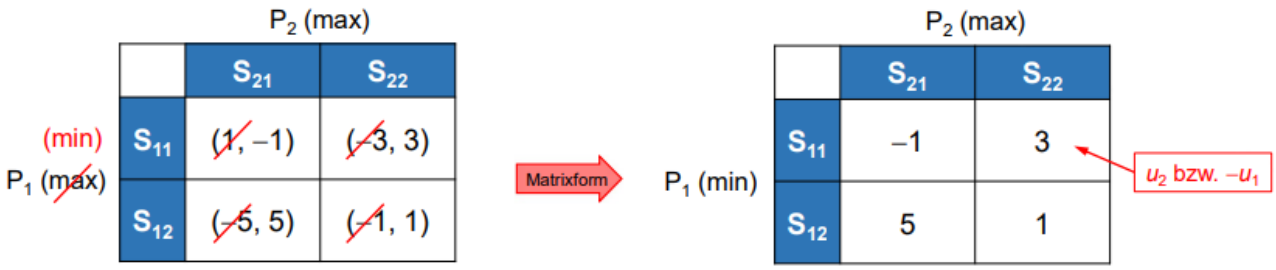
> Eine Strategie die echt besser ist als alle anderen Strategien (>)

Falke-Taube-Spiel

Zwei-Personen-Nullsummenspiel

- Spieler 2 will gewinn maximieren, Spieler 1 will verlust minimieren
- nicht jedes 2PNS in reinen Strategien hat ein Nash-Gleichgewicht.
- **Gewinnfunktion**: $u_2 = -u_1$

- Matrixform



Konservative Strategien

- Wenn die Konservate Strategie beider Spieler gleich ist, ist dort ein Nash-Gleichgewicht.

MinMax-Strategie Was sollte ich höchstens verlieren?

> 1. Mache eine liste mit dem jeweils größten Verlust
> 2. Daraus wähle den kleinsten Verlust

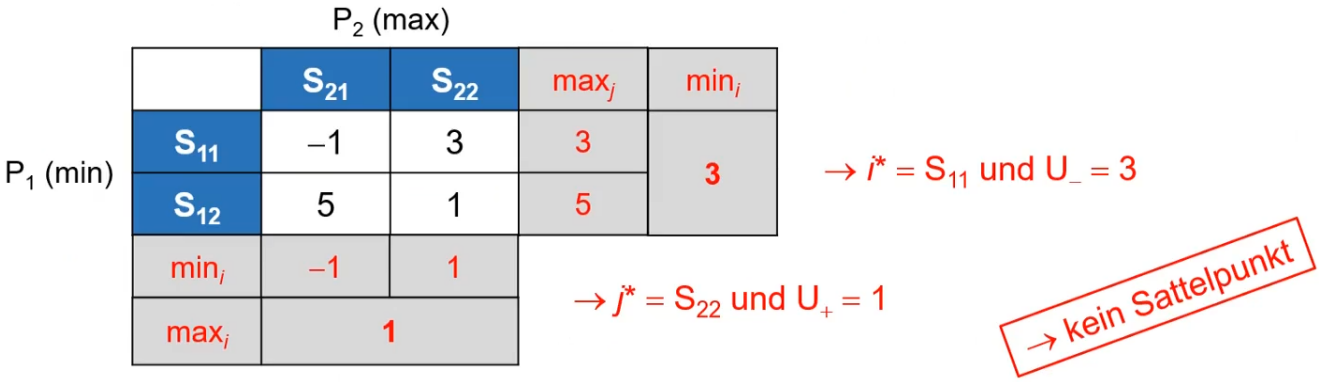
$$U_{-} = \min_{i}(\max_{j}(a_{ij}))$$

MaxMin-Strategie Was sollte ich mindestens gewinnen?

> 1. Mache eine liste mit dem minimalen gewinn
> 2. Daraus wähle den größten Gewinn

$$U_{+} = \max_{j}(\min_{i}(a_{ij}))$$

Sattelpunkt $$U_{-} = a_{i^* j^*} = U_{+}$$



Gemischte Strategien

Gruppenentscheidungen

4. Automaten

- **Forderung**: Simulation schneller als Echtzeit

Bestandteile

- **Zellraum**: Spielfeld, i.d.R. eine Matrix in der jedes Element einer Zelle entspricht
- **Zustandsmenge**: Menge aller möglichen (diskrete) Zustände einer Zelle -> State Machine
- **Nachbarschaftsbeziehung**: Zustände der Nachbarzellen werden berücksichtigt
- **diskrete Zeit**: Zustandsänderung eines ZA erfolgt in diskreten Zeitschritten δt (z.B. 1s)
- **lokale übergangsfunktion**: Berechnet den Zustand $Z_{t+\delta t}$ einer Zelle aus dem aktuellen Zustand Z_t und den Zuständen der Nachbarzellen

Nachbarzellen (im 2D und Schachbrett)

- **Moore-Nachbarschaft**: 8 Zellen um die Zelle
- **Von-Neumann-Nachbarschaft**: 4 Zellen um die Zelle

Übergangsfunktion (Beispiel Conways Game of Life)

```
def transition_function(cell, neighbors):
    if cell == 0 and sum(neighbors) == 3: # Wenn Zelle tot und 3 Nachbarn lebendig
        return 1                          # Dann wird die Zelle wiederbelebt
    elif cell == 1 and sum(neighbors) in [2, 3]: # Wenn Zelle lebendig und 2 oder
3 Nachbarn lebendig
        return 1
    else:
        return 0
```

Game of Life

Start mit 2D-Gitter. Jede Zelle ist entweder lebendig oder tot und ändert ihren Zustand gemäß der Nachbarzellen:
Eine lebende Zelle mit 2 oder 3 lebenden Nachbarn bleibt am Leben, sonst stirbt sie. Eine tote Zelle mit genau 3 lebenden Nachbarn wird lebendig. Es werden ALLE Zellen mit dem aktuellen Zustand und den Nachbarzellen betrachtet, daraus wird der nächste Zustand berechnet, damit die Zellen parallel aktualisiert werden.

Kurzfassung:

- Grid anlegen (2D-Liste)
- Jede Runde Nachbarn zählen
- Nach obigen Regeln neue Zustände berechnen
- Endlos aktualisieren und ausgeben.

```
import pygame
import sys
import numpy as np

# Pygame initialisieren
pygame.init()

# Fenstergröße und Gitterparameter
WIDTH, HEIGHT = 800, 800
```

```
CELL_SIZE = 10
GRID_WIDTH = WIDTH // CELL_SIZE
GRID_HEIGHT = HEIGHT // CELL_SIZE

# Farben definieren
BG_COLOR = (10, 10, 10)
GRID_COLOR = (40, 40, 40)
ALIVE_COLOR = (255, 255, 255)

# Fenster erstellen
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Game of Life")

# Uhr für die Framerate
clock = pygame.time.Clock()
FPS = 10  # Geschwindigkeit des Spiels

# Gitter initialisieren (zufällige Zustände)
grid = np.random.choice([0, 1], size=(GRID_HEIGHT, GRID_WIDTH))


def draw_grid(surface, grid):
    surface.fill(BG_COLOR)
    for y in range(GRID_HEIGHT):
        for x in range(GRID_WIDTH):
            if grid[y, x] == 1:
                pygame.draw.rect(
                    surface,
                    ALIVE_COLOR,
                    (x * CELL_SIZE, y * CELL_SIZE, CELL_SIZE - 1, CELL_SIZE - 1),
                )
    # Optionale Gitternetzlinien
    for x in range(0, WIDTH, CELL_SIZE):
        pygame.draw.line(surface, GRID_COLOR, (x, 0), (x, HEIGHT))
    for y in range(0, HEIGHT, CELL_SIZE):
        pygame.draw.line(surface, GRID_COLOR, (0, y), (WIDTH, y))


def update_grid(current_grid):
    new_grid = current_grid.copy()
    for y in range(GRID_HEIGHT):
        for x in range(GRID_WIDTH):
            # Anzahl der lebenden Nachbarn
            total = (
                current_grid[y, (x - 1) % GRID_WIDTH]
                + current_grid[y, (x + 1) % GRID_WIDTH]
                + current_grid[(y - 1) % GRID_HEIGHT, x]
                + current_grid[(y + 1) % GRID_HEIGHT, x]
                + current_grid[(y - 1) % GRID_HEIGHT, (x - 1) % GRID_WIDTH]
                + current_grid[(y - 1) % GRID_HEIGHT, (x + 1) % GRID_WIDTH]
                + current_grid[(y + 1) % GRID_HEIGHT, (x - 1) % GRID_WIDTH]
                + current_grid[(y + 1) % GRID_HEIGHT, (x + 1) % GRID_WIDTH]
            )
```

```
            # Spielregeln
            if current_grid[y, x] == 1:
                if total < 2 or total > 3:
                    new_grid[y, x] = 0
            else:
                if total == 3:
                    new_grid[y, x] = 1
    return new_grid


def main():
    global grid
    running = True
    paused = False

    while running:
        clock.tick(FPS)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            # Pause/Weiter mit Leertaste
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_SPACE:
                    paused = not paused
            # Klick zum Setzen von Zellen
            if event.type == pygame.MOUSEBUTTONDOWN:
                x, y = pygame.mouse.get_pos()
                grid[y // CELL_SIZE, x // CELL_SIZE] = 1

        if not paused:
            grid = update_grid(grid)

        draw_grid(screen, grid)
        pygame.display.flip()

    pygame.quit()
    sys.exit()


if __name__ == "__main__":
    main()
```