ChickenRoad

1. Analyse

Spielidee

In ChickenRoad steuert der Spieler ein Huhn durch eine dynamisch generierte 3D-Welt, die aus abwechselnden Straßen- und Grasabschnitten besteht. Das Huhn muss stark befahrene Straßen überqueren, ohne von Autos erfasst zu werden, und auf Grasflächen Hindernissen wie Bäumen ausweichen. Ziel ist es, so viele Straßen wie möglich zu überqueren und dabei einen möglichst hohen Score zu erreichen.

Funktionale Anforderungen

- Steuerung: Der Spieler bewegt das Huhn mit den Tasten W (vorwärts), A (links), S (rückwärts), D (rechts)
- Bewegungsmechanik: Das Huhn bewegt sich sprungweise um die festgelegte Tile-Größe
- Ziel: Möglichst viele Punkte sammeln durch Überqueren von Straßen, ohne von Autos erwischt zu werden
- Autos: Bewegen sich auf Straßenabschnitten mit zufälliger Geschwindigkeit und Richtung
- Hindernisse: Bäume auf Grasflächen blockieren den Weg
- Kollisionssystem:
    - Kollision mit Auto → Game Over
    - Kollision mit Baum → Bewegung blockiert
- Score-System: Punkte basierend auf zurückgelegter Distanz
- Level-Generierung: Prozedurale Generierung neuer Abschnitte während des Spiels
- Menü-System: Pause-Menü und Game-Over-Bildschirm mit Restart-Option

Nicht-funktionale Anforderungen

- Performance: Flüssige Darstellung bei mindestens 60 FPS
- Steuerung: Intuitive Tastensteuerung mit Bewegungs-Cooldown
- Grafik: 3D-Darstellung mit isometrischer Kamera
- Assets: Verwendung von 3D-Modellen für alle Spielobjekte
- Plattform: Lauffähig auf Windows, macOS und Linux mit Python 3.13
- Speicher: Persistente Speicherung des Highscores

Abgrenzung

- Kein Mehrspielermodus
- Keine Maussteuerung
- Keine komplexe KI - Auto-Bewegungen sind linear
- Keine Power-Ups oder Spezialfähigkeiten
- Keine unterschiedlichen Charaktere (nur Huhn)

2. Entwurf / Modellierung

Werkzeuge

- Programmiersprache: Python 3.13.0
- Game Engine: Ursina Engine 8.1.1 (basierend auf Panda3D)
- Bibliotheken:
  - ursina - 3D-Grafik, Eingabe, Kollisionserkennung
  - json - Konfigurationsdateien und Highscore-Speicherung
  - random - Zufällige Generierung von Objekten und Geschwindigkeiten
  - math - Berechnungen für Positionierung und Abstände
- **Assets:** GLB-Modelle für Huhn, Autos, Straßen, Gras, Bäume

Aufbau des Projektes

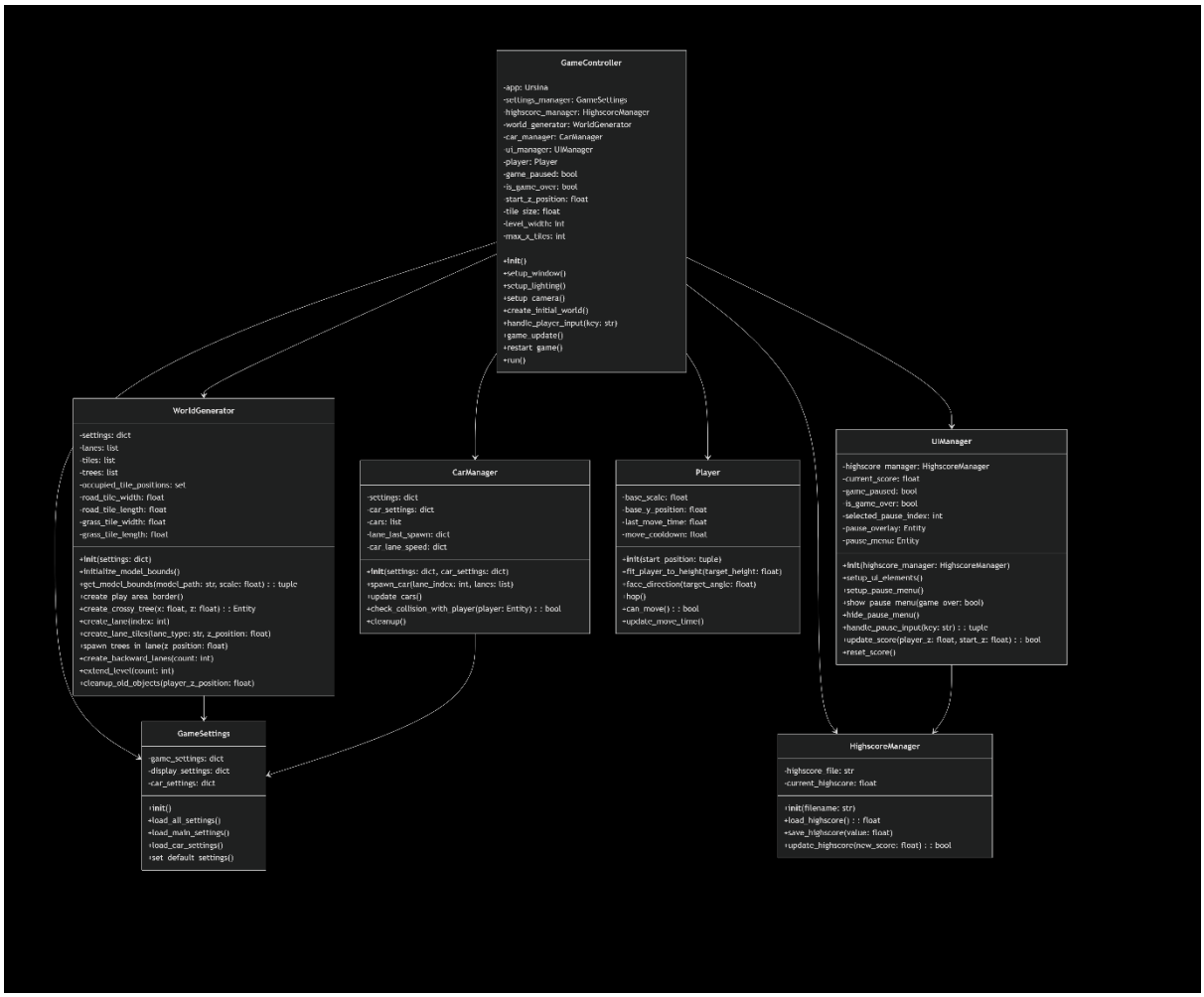
1. **main.py (Hauptmodul)**
   - Initialisiert Ursina Engine und alle Manager
   - Verwaltet den Haupt-Spiel-Loop
   - Verarbeitet Eingaben und Spielzustände
   - Koordiniert alle Spielkomponenten
2. **game_controller.py (Spielsteuerung)**
   - Zentrale Steuerungsklasse für das gesamte Spiel
   - Verwaltet Spielzustand (Pause, Game Over)
   - Koordiniert alle Manager und Komponenten
   - Implementiert Neustart-Funktionalität
3. **game_settings.py (Einstellungsmanagement)**
   - Lädt und verwaltet alle Spielparameter aus JSON-Dateien
   - Bietet Fallback-Werte bei Fehlern
   - Verwaltet Grafikeinstellungen und Spielbalance
4. **world_generator.py (Weltgenerator)**
   - Generiert prozedural Straßen- und Grasabschnitte
   - Platziert Bäume auf Grasflächen
   - Verwaltet Cleanup von nicht sichtbaren Objekten
   - Erstellt Spielbereich-Grenzen

5. **car_manager.py (Auto-Manager)**
    - Verwaltet das Spawnen und Bewegen von Autos
    - Steuert Kollisionserkennung mit Spieler
    - Verwaltet Auto-Geschwindigkeiten und -Modelle
6. **player.py (Spieler-Charakter)**
    - Implementiert die Spieler-Steuerung und -Animationen
    - Verwaltet Bewegungs-Cooldown und Kollisionen
    - Steuert die Hüpf-Animation
7. **ui_manager.py (Benutzeroberfläche)**
    - Zeigt Score und Highscore an
    - Implementiert Pause- und Game-Over-Menüs
    - Verwaltet Button-Interaktionen
8. **highscore_manager.py (Highscore-Verwaltung)**
    - Speichert und lädt Highscore persistent
    - Prüft und aktualisiert Highscores

Datenmodell des Spielfelds

- Das Spielfeld besteht aus einer Liste von Lanes (jeweils 1 Tile breit)
- Jede Lane ist ein Dictionary mit den Schlüsseln:
    - type: "road" oder "grass"
    - direction: Bewegungsrichtung (-1, 0, 1)
    - z: Z-Position der Lane
- Autos werden als Entities mit Geschwindigkeit und Richtung verwaltet
- Bäume werden als Entities mit Collidern verwaltet
- Der Spieler wird als Entity mit Position, Rotation und Collider verwaltet

UML-Klassendiagramm:



3. Implementierung

1. Modul: main.py

Funktionen

- show_pause(game_over=False) - Zeigt Pause- oder Game-Over-Menü
- hide_pause() - Versteckt Pause-Menü
- restart_game() - Setzt Spiel komplett zurück
- create_score_text() - Erstellt Score-Anzeige

Zentrale Daten

- player: Spieler-Entity mit Position und Collider
- world_generator: Verwaltet alle Welt-Objekte
- car_manager: Verwaltet alle Auto-Objekte
- current_score: Aktueller Spiel-Score

Game Loop

- Verarbeitet Eingaben → bewegt Spieler um tile_size
- Aktualisiert Welt (neue Abschnitte generieren)
- Bewegt Autos und prüft Kollisionen
- Aktualisiert Score und Highscore
- Zeichnet 3D-Szene und UI-Elemente

2. Modul: world_generator.py

Konstanten

- tile_size = 1.0 (Standard-Tile-Größe)

Funktionen

- create_lane(index) - Erstellt neue Lane an Position
- spawn_trees_in_lane(z_position) - Platziert Bäume auf Gras-Lanes
- extend_level(count) - Erweitert Level um neue Lanes
- cleanup_old_objects(player_z_position) - Räumt entfernte Objekte auf

3. Modul: game_settings.py

Funktionen

- load_main_settings() - Lädt Haupteinstellungen aus settings.json
- load_car_settings() - Lädt Auto-Einstellungen aus car_settings.json
- set_default_settings() - Fallback-Werte bei Fehlern

Einstellungsbereiche

- Auto-Geschwindigkeit: 2.0 - 5.0 Einheiten/Sekunde
- Auto-Spawn-Chance: 18% pro Frame
- Baum-Spawn-Chance: 70% pro Gras-Lane
- Cleanup-Distanz: 120 Einheiten hinter Spieler

4. Modul: car_manager.py

Funktionen

- spawn_car(lane_index, lanes) - Erstellt neues Auto in Lane
- update_cars() - Bewegt alle Autos
- check_collision_with_player(player) - Prüft Auto-Spieler-Kollision

Auto-Typen

- 3 verschiedene Auto-Modelle mit individuellen Skalierungen
- Zufällige Geschwindigkeit pro Lane
- Richtungsabhängige Rotation

5. Modul: player.py

Funktionen

- face_direction(target_angle) - Dreht Spieler in Bewegungsrichtung
- hop() - Spielt Hüpf-Animation ab
- can_move() - Prüft Bewegungs-Cooldown

Animationen

- Hüpf-Animation mit curve.out_quad und curve.in_quad
- Drehanimation bei Richtungswechsel
- Kollisions-Feedback (roter Blitz)

4. Tests und Wartung

Manuelle Tests

Steuerung

- Überprüfung, dass das Huhn auf W,A,S,D-Eingaben in die richtige Richtung springt
- Test der Bewegungs-Begrenzung an den Spielfeldrändern
- Verifikation des Bewegungs-Cooldowns

Kollisionen

- Huhn kollidiert mit Auto → Game Over
- Huhn kollidiert mit Baum → Bewegung blockiert
- Korrekte Kollisionsrückstellung bei Bäumen

Weltgenerierung

- Prozedurale Generierung neuer Abschnitte funktioniert kontinuierlich
- Cleanup von entfernten Objekten verhindert Speicherlecks
- Bäume spawnen mit angemessenen Abständen

UI-System

- Score-Anzeige aktualisiert sich korrekt
- Highscore wird bei Überschreitung gespeichert
- Pause-Menü reagiert auf Tasteneingaben
- Game-Over-Bildschirm ermöglicht Neustart

Auto-System

- Autos spawnen mit angemessenen Abständen
- Geschwindigkeitsvariation zwischen verschiedenen Lanes
- Kollisionserkennung mit Spieler funktioniert zuverlässig

Modultests

world_generator.create_lane(index)

- Erzeugt bei jedem Aufruf eine neue Lane mit zufälligem Typ
- Korrekte Positionierung von Tiles und Bäumen
- Vermeidung von überlappenden Objekten

car_manager.spawn_car(lane_index, lanes)

- Autos spawnen nur auf Straßen-Lanes
- Berücksichtigung von Mindestabständen zwischen Autos
- Korrekte Geschwindigkeitszuweisung

highscore_manager.update_highscore(new_score)

- Highscore wird nur bei Überschreitung aktualisiert
- Persistente Speicherung in JSON-Datei
- Korrekte Rückgabewerte bei Erfolg/Misserfolg

Fehlerbehebung (Debugging)

Grafik-Flackern bei Gras-Texturen

- Problem: Gras-Tiles zeigten gelegentliches Flackern
- Lösung: Anpassung der Render-Reihenfolge und Double-Sided-Flag

Auto-Kollisionserkennung unzuverlässig

- Problem: Kollisionen wurden manchmal nicht erkannt
- Lösung: Anpassung der Collider-Größen und -Positionen

Performance-Einbrüche bei vielen Objekten

- Problem: Framerate-Drop bei vielen generierten Objekten
- Lösung: Implementierung von Cleanup-System für entfernte Objekte

Score-Text wurde nicht angezeigt

- Problem: Score-Text erschien erst nach erster Bewegung
- Lösung: Erstellung des Text-Objekts bei Spielstart mit Standardwert

Wartung und Erweiterbarkeit

Assets austauschen

- Einfacher Austausch von 3D-Modellen durch Anpassung der Model-Pfade
- Skalierungswerte in car_settings.json konfigurierbar

Neue Objekt-Typen

- Einfache Erweiterung um neue Lane-Typen (z.B. Wasser mit Baumstämmen)
- Hinzufügen neuer Auto-Modelle durch Erweiterung der car_models-Liste

Erweiterungen

- Punktesystem mit Bonussen für spezielle Aktionen
- Mehrere Leben oder Gesundheitsleiste
- Power-Ups (z.B. temporäre Unverwundbarkeit, Geschwindigkeitsboost)
- Verschiedene Spielcharaktere mit unterschiedlichen Fähigkeiten
- Tageszeit-Zyklus mit dynamischer Beleuchtung
- Wettereffekte (Regen, Schnee) die das Gameplay beeinflussen

Optimierungen

- Level of Detail (LOD) für weit entfernte Objekte
- Objekt-Pooling für häufig erstellte Entities
- Asynchrones Laden von Assets im Hintergrund

5. Technische Besonderheiten

- 3D-Engine Integration
- Das Spiel verwendet die Ursina Engine, die auf Panda3D aufbaut und eine pythonische Schnittstelle für 3D-Spieleentwicklung bietet. Dies ermöglicht:
- Einfache Entity-basierte Objektverwaltung
- Integrierte Kollisionserkennung
- Automatisches Rendering und Beleuchtung
- Kamera-Steuerung mit Lerp für sanfte Bewegungen
  Prozedurale Generierung
  Die Welt wird dynamisch generiert während der Spieler sich vorwärts bewegt:
- Unendliche Level-Länge durch kontinuierliche Generierung
- Zufällige Verteilung von Straßen- und Grasabschnitten
- Intelligentes Spawning von Hindernissen mit Mindestabständen
  Performance-Optimierung
- Automatisches Cleanup: Entfernte Objekte werden nach 120 Einheiten zerstört
- Effiziente Kollisionserkennung: Box-Collider für alle interaktiven Objekte
- Optimierte Render-Pipeline: Ursina verwaltet automatisch Render-Reihenfolge
  Modulare Architektur
- Klare Trennung der Verantwortlichkeiten zwischen Klassen
- Einfache Erweiterbarkeit durch klare Schnittstellen
- Konfiguration aller Spielparameter in JSON-Dateien
- Robustes Fehlerhandling mit Fallback-Mechanismen

6. Weiterhin bekannte Fehler (BUGS)

- Manche Hitboxen könnten noch besser angepasst werden
- Animation in schnellem Springen wird ausgesetzt
- Wiese flackert weiterhin
- Man kann in einem Baum spawnen, da der Spawn als „Wiese“ Lane deklariert ist