GeoCrash

von Aaron Ehrlich, Dennis Natusch,
Felix Kamphues

Idee

Mehrspieler-Geschicklichkeitsspiel

Zwei Spieler benutzen Geometrische Objekte auf dem Spielfeld um sich damit gegenseitig zu bewerfen.

Physikalische Simulation der gewirkten Kräfte, Kollisionen usw.


Lives PLayer1 = 10
Lives PLayer2 = 10

Benutzte Crates

- Piston(verworfen):
  - erste Idee haben uns dann aber schnell dagegen entschieden
- ggez (crates.io/ggez):
  - erlaubt es einfache Formen sowie Bilder und Texte auf dem Bildschirm auszugeben
- nphysics (nphysics.org)
  - 2d und 3d Simulation
  - verwendet nalgebra (nalgebra.org) und ncollide (ncollide.org)

Struktur der Implementierung

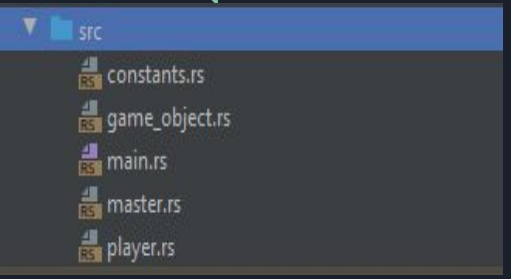
GameObjects und Player bestehen aus RigidBody und Collider

RigidBodies haben eine Masse, aber keine Ausdehnung und werden von Kräften (Gravitation, Beschleunigungen) beeinflusst

Collider haben eine Ausdehnung, aber keine Masse, sie können mit anderen Collidern kollidieren

nPhysics übernimmt Kollisionsberechnung und Anwendung der Kräfte

Struktur der Implementierung



- main:
  - initialisiert das Spiel und das Fenster
- constants:
  - Sammlung von Konstanten
- master:
  - speichert und aktualisiert die simulierte Welt einschließlich der Spieler und der GameObjects
- player:
  - initialisiert, updatet und zeichnet einen Spieler und speichert alle dafür relevanten Daten
- game_objects:
  - initialisiert, updatet und zeichnet ein GameObject und speichert alle dafür relevanten Daten

Gamelogik

```rust
//EventHandler handling events...
impl EventHandler for Master {
    fn update(&mut self, _ctx: &mut Context) -> GameResult<()> {

        if self.over {
            if keyboard::is_key_pressed(_ctx, key: KeyCode::Space) {
                self.over = false;
                self.reset(_ctx);
            }
            return Ok(());
        }

        self.player1.update( context: _ctx, &mut self.bodies, &mut self.force_generators);
        self.player2.update( context: _ctx, &mut self.bodies, &mut self.force_generators);

        self.update_game_objects();

        self.mechanical_world.step(     //move the simulation further one step
             gworld: &mut self.geometrical_world,
            &mut self.bodies,
            &mut self.colliders,
            &mut self.joint_constraints,
             forces: &mut self.force_generators
        );

        self.handle_proximity_events();

        self.handle_contact_events();

        Ok(())
    }
```

Gamelogik
Die mechanical / geometrical world und ihre Sets:

`struct MechanicalWorld<N: RealField, Handle: BodyHandle, CollHandle: ColliderHandle>`:

- simuliert physisches Verhalten gegebener RigidObjekte (Kräfte, Bewegungen/Drehungen, …)

`pub struct GeometricalWorld<N: RealField, Handle: BodyHandle, CollHandle: ColliderHandle>`

- geometrische Berechnungen, wie Kollisionsabfragen, Ray-Tracing

Beide structs verwenden Sets, um indirekt auf Objekte (Rigidbodies, Collider, ForceGenerators) zugreifen zu können.

- BodySet
- ColliderSet
- JointConstraintSet
- ForceGeneratorSet

Gamelogik
Die mechanical / geometrical world und ihre Sets:

MechanicalWorld::step()

- simuliert einen “Schritt” der Welten
- Berechnungen werden ausgeführt und mit den Sets synchronisiert

Ergebnisse werden für unsere weitere Spiellogik verwendet:

- Einwirken von “künstlichen” Kraftvektoren, (z.B. Bewegung des Spielerobjektes)
- VErwendung der Kollisionsabfrage-Ergebnisse, um unsere Spieleffekte auszuführen
- Übertragung von Positions- und Drehungs-Daten der Objekte auf GGEZs Graphik-Funktionalitäten

Gamelogik

handle_proximity_events() - wenn ein Spieler einem GameObject nahe kommt, wird der Spieler beim GameObject registriert. Solange ein Spieler bei einem GameObject registriert ist, wirkt bei jedem Update eine Kraft auf das GameObject, die dieses zum Spieler hinzieht

handle_contact_events() - wenn ein abgeschossenes GameObject und ein Spieler kollidieren, verliert der Spieler Leben

Gamelogik

```
fn handle_proximity_events(&mut self){

    for proximity in self.geometrical_world.proximity_events(){

        let collider1 = self.colliders.get(proximity.collider1).unwrap();
        let collider1 = self.colliders.get(proximity.collider1).unwrap();

        //if collider1 is a player and collider2 is a gameObject:
            //if ProximityEvent == Disjoint:
                //deregister player in gameObject
            //else:
                //register player in gameObject
```

Probleme bei der Implementierung LifeTime-Parameter

Der Struct GameObject sollte die Variable Player (ein anderer struct) enthalten

Rust fordert allerdings einen Lifetime-Parameter, um sicherzustellen, dass das GameObject nicht länger lebt als der Player

Lösung: Wir haben nur eine Kopie der Player-Position als Vektor gespeichert

https://doc.rust-lang.org/reference/items/generics.html

Probleme bei der Implementierung
&dyn Trait vs Box<T>

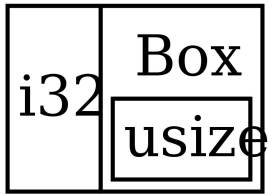
Problem:

- Typsicherheit von Rust benötigt schon zur Compilezeit genaue Größe übergebener Parameter-Typen
- Rekursive Datentypen, welche sich selbst beinhalten können, sind nicht zugelassen.

Lösungen:

- &dyn dynamische, veränderbare Referenz:
  - übergebe nur die Referenz zum Objekt, wird erst zur Laufzeit ausgewertet.
    Laufzeiteinbußen
- Box<T> Smart Pointer auf einen Heap:
  - Speichere T, anstelle von standardmäßig im Stack, auf einem Heap
  - rekursive Typ-Elemente werden stattdessen durch Pointer auf Heap-Speicherplätze “umgelagert”

Bsp.: Typ def. mit Boxed i32



GeoCrash

(von Aaron Ehrlich, Dennis Natusch, Felix Kamphues)

Danke für eure Aufmerksamkeit