

Unser Sonnensystem

- eine Vielkörpersimulation -

Dominik Schlothane

Das Ziel

- Interaktive "echtzeit" Simulation der Planeten im Sonnensystem
- Konkret:
 - Vielkörpersimulation mit Gravitationskraft
 - GUI:
 - Simulationsdaten darstellen
 - modifizierung der Simulationsparameter

Theorie

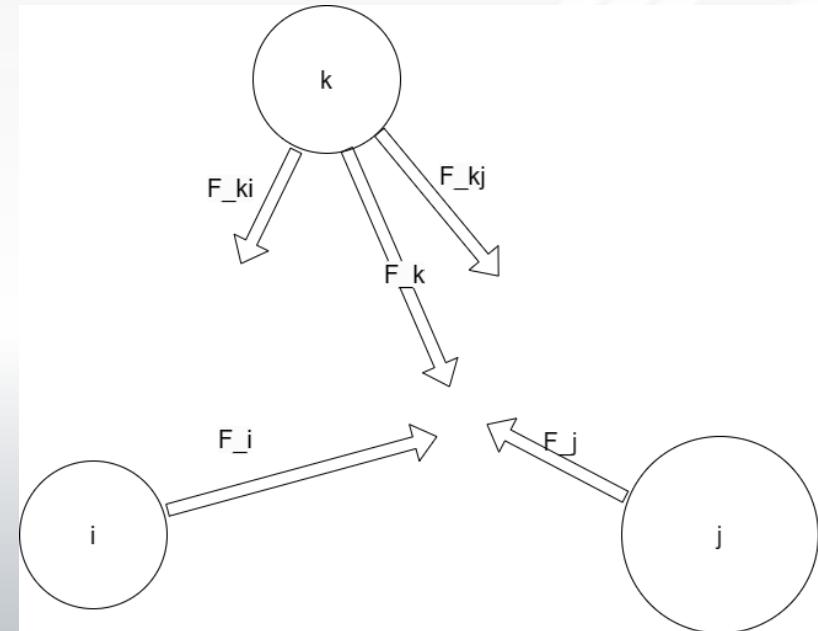
Physik

- klassische Gravitation

$$\vec{F}_{ij} = G \frac{m_i m_j}{(\vec{r}_j - \vec{r}_i)^3} (\vec{r}_j - \vec{r}_i)$$

$$\vec{a}_{ij} = G \frac{(\vec{r}_j - \vec{r}_i)}{(\vec{r}_j - \vec{r}_i)^3} m_j$$

$$\vec{a}_i = \sum_{j \neq i} \vec{a}_{ij}$$



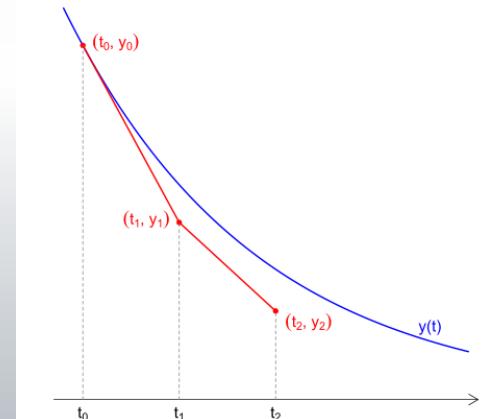
Numerik

- gesucht: $\vec{x}_i(t)$ mit $\vec{a}_i(t) = \frac{d^2}{dt^2} \vec{x}_i(t)$
- DGLs 1. Ordnung: $\vec{a}_i(t) = \frac{d}{dt} \vec{v}_i(t), \quad \vec{v}_i(t) = \frac{d}{dt} \vec{x}_i(t)$
- Eulerverfahren:

$$t_{n+1} = t_n + h$$

$$v(t_{n+1}) = v(t_n) + h \cdot a(t_n)$$

$$x(t_{n+1}) = x(t_n) + h \cdot v(t_n)$$



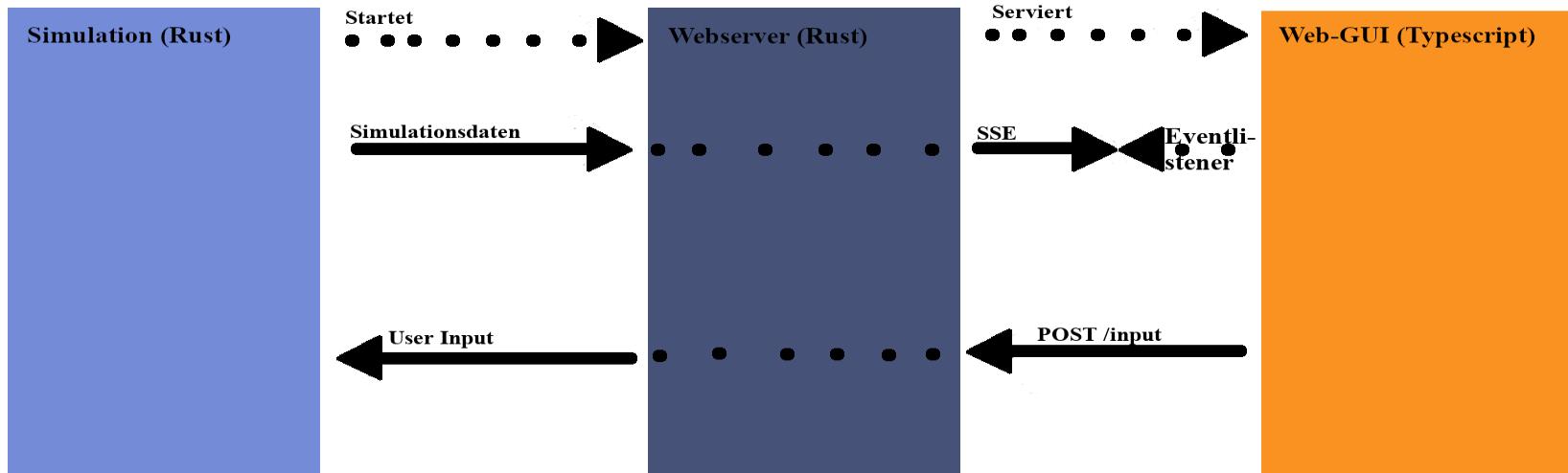
HilberTraum, CC BY-SA 4.0, via Wikimedia Commons

Umsetzung

Umsetzung

- Grafikerfahrung im Web (HTML + CSS + TypeScript)
⇒ Leistung könnte mit Simulation knapp werden
- Simulation in Rust
- Server notwendig (Rust)

Kommunikation



Simulation

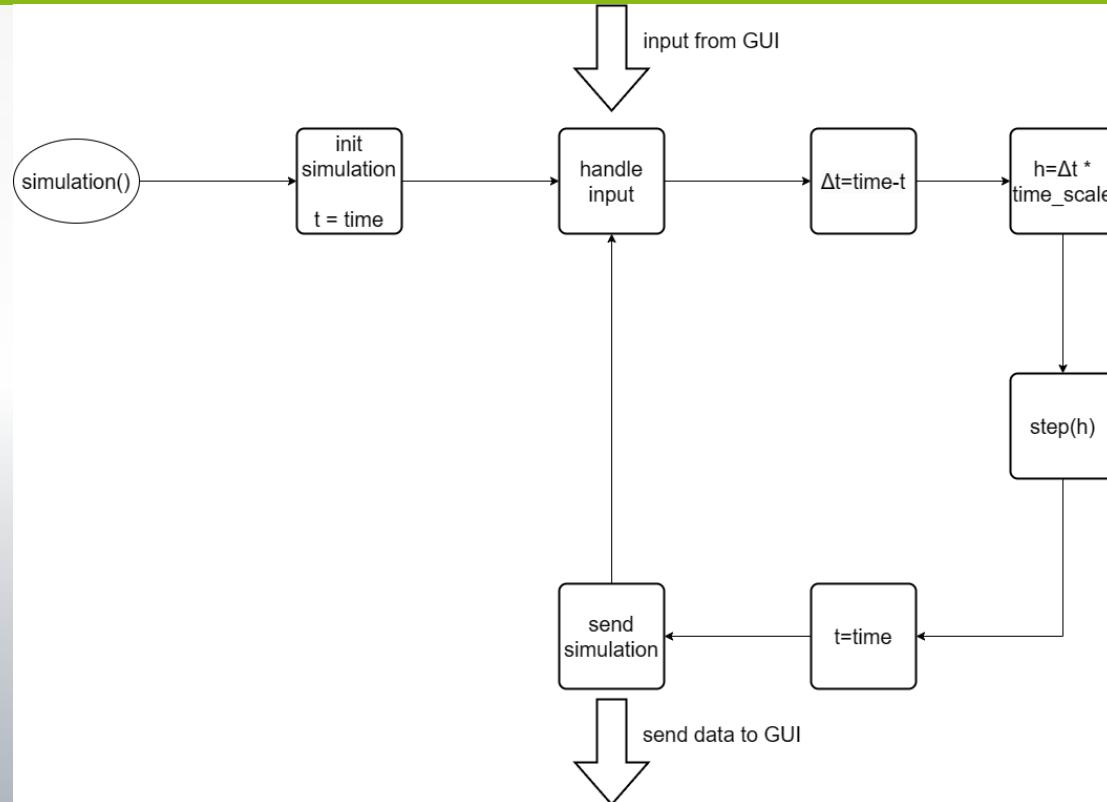
Simulation - Struktur

SimState
+ bodies: Vec<Body>
+ metadata: SimMetaData
+ target_time_per_step_s: f64
+ interact(usize, usize)
+ step(f64, Sender<usize>): Vec<usize>
+ handle_input(InputEvent, Option<Body>, Sender<usize>)
+ handle_meta_input(SimMetaData)

Body
+ mass: f64
+ density: f64
+ radius: f64
+ pos: Vector3<f64>
+ vel: Vector3<f64>
+ acc: Vector3<f64>
+ accelerate(f64): Res<()>
+ movement(f64)

SimMetaData
+ interaction_constant: f64,
+ time_scale: f64

Simulation - Ablauf



Simulation - step

```
step(h: f64) {
    for i in 0..self.bodies.len() {
        for k in i + 1..self.bodies.len() {
            self.interact(i, k);
        }
    }

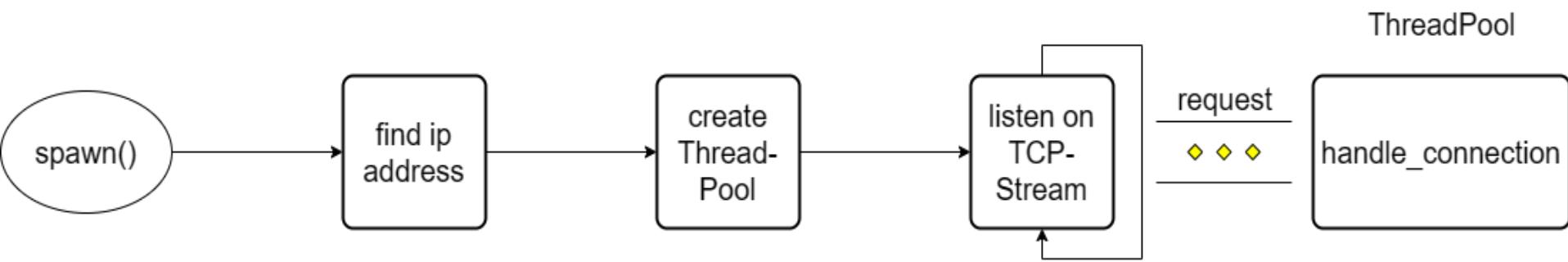
    for i in 0..self.bodies.len() {
        let body = &mut self.bodies[i];
        body.accelerate(h)
        body.movement(h)
    }
}
```

Server

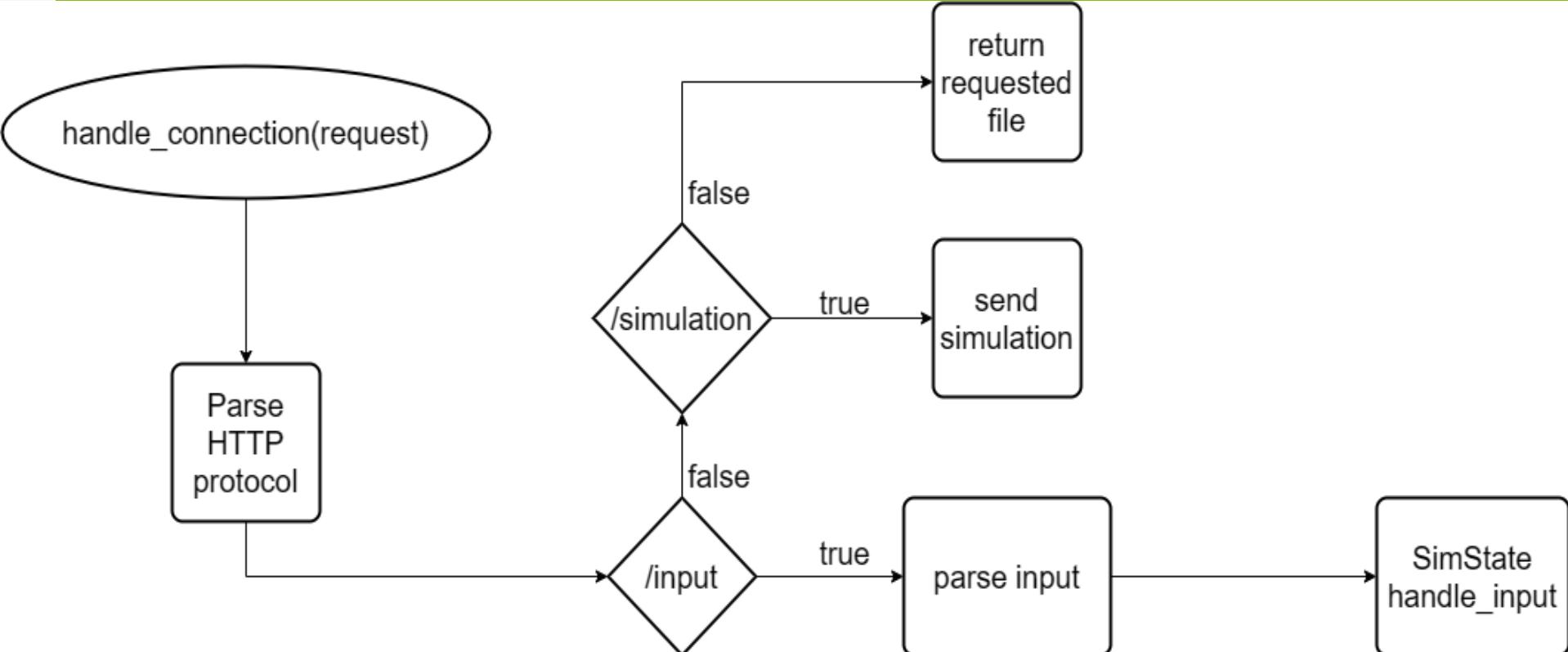
Server - Endpoints

- GET /simulation:
öffnet Verbindung, um SSE zu senden
- POST /input {eventtype, event}:
 - Add: erzeuge neuen Body, event = new Body
 - Remove: entferne einen Body, event = {index: index}
 - Update: modifiziere eine Body, event = {index: Body}
 - Meta: modifiziere MetaData, event = {time_scale, interaction_constant}

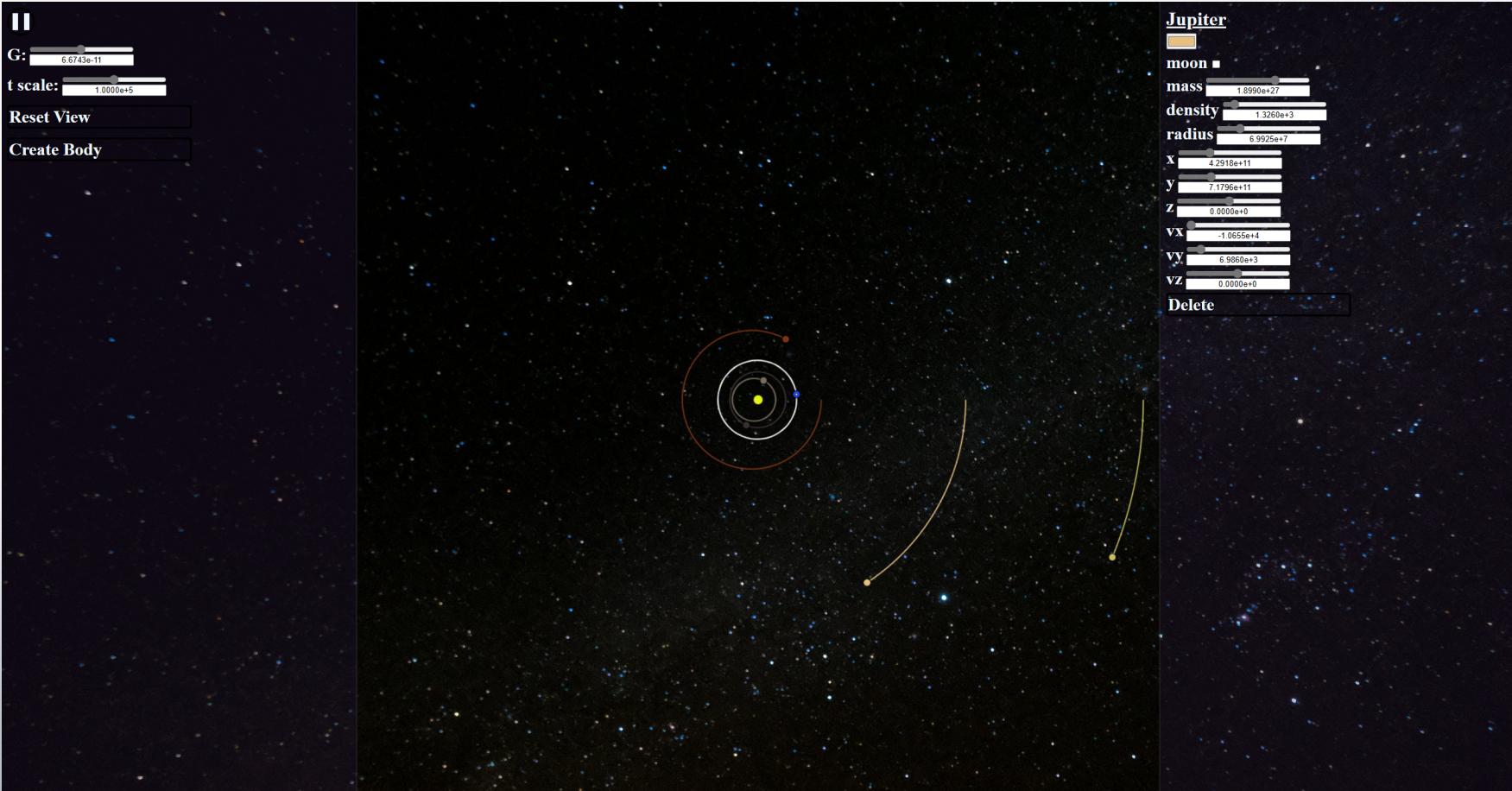
Server - Ablauf

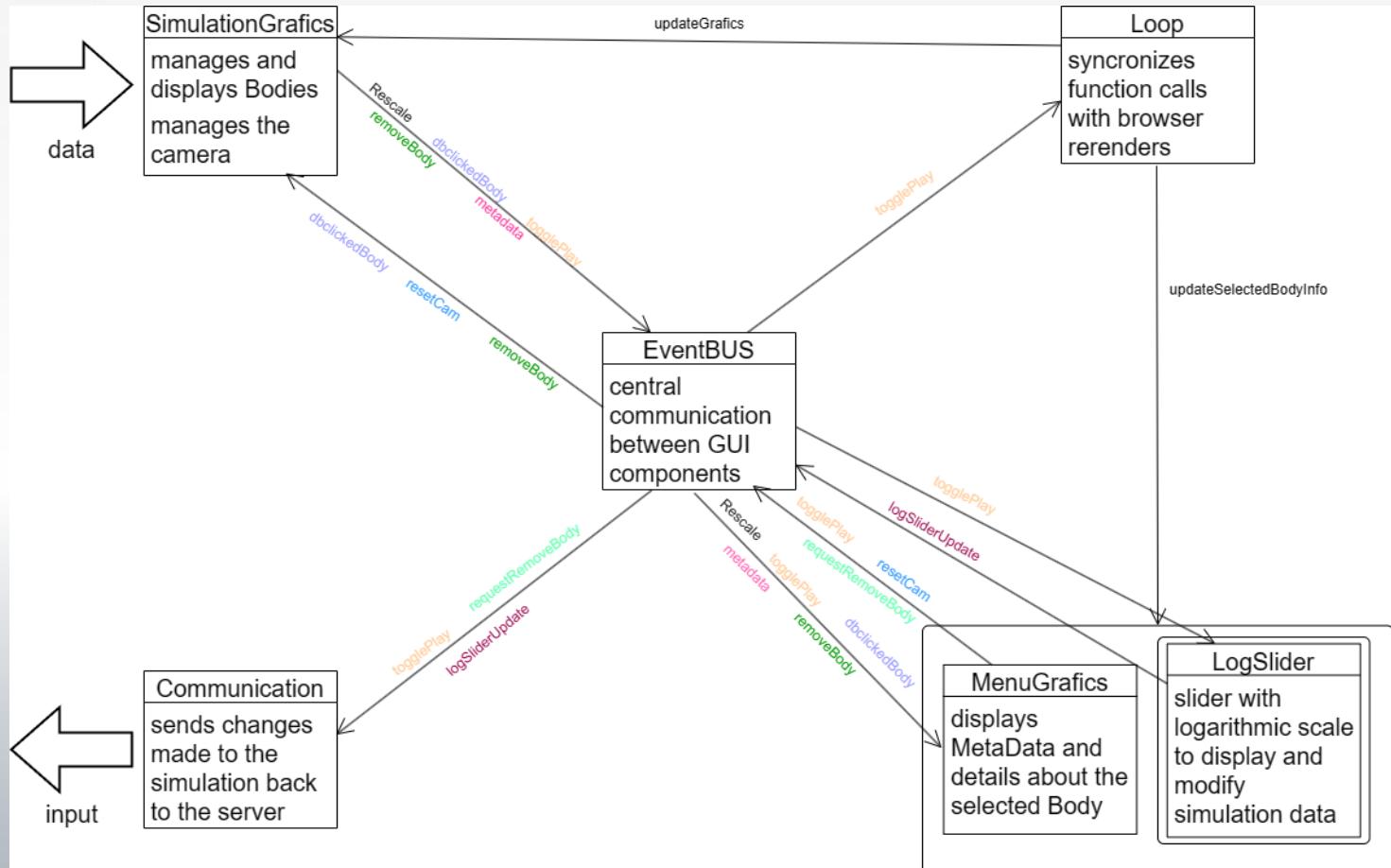


Server - handle_connection



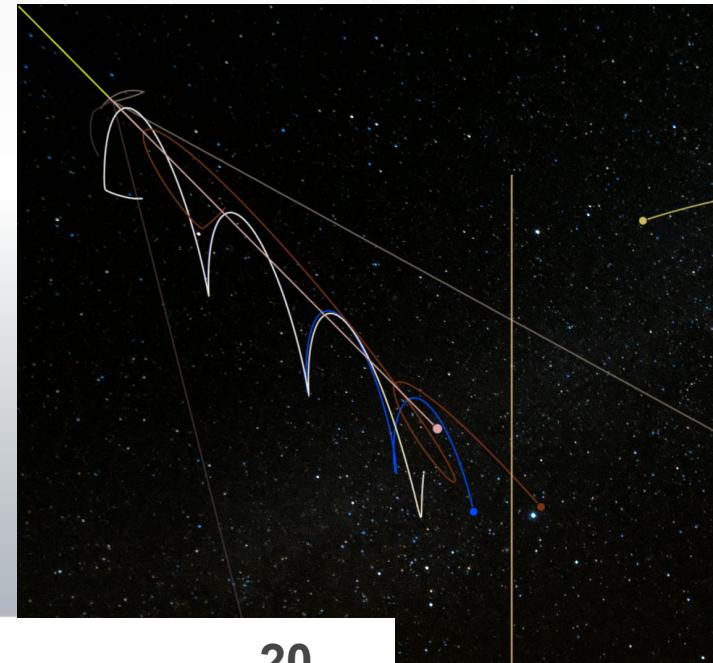
GUI





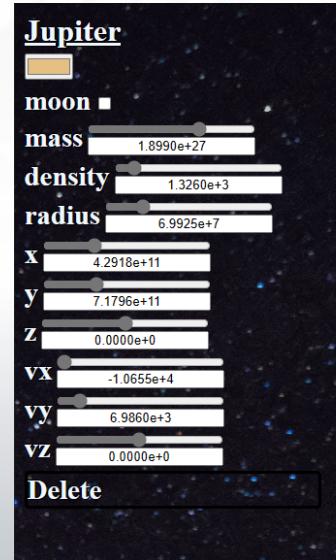
Kollision

- keine Kollision implementiert
- Punktmassen kommen sich sehr nahe
 $\Rightarrow a > 10^6 \frac{\text{m}}{\text{s}^2}$ möglich
- typisch: `time_scale = 1e6, h = 1e3`
 $\Rightarrow \Delta x > 10^{12} \text{ m}$
- 2 Lösungen:
 - lösche Bodies mit $a > 10^6 \frac{\text{m}}{\text{s}^2}$
 - implementiere Kollision



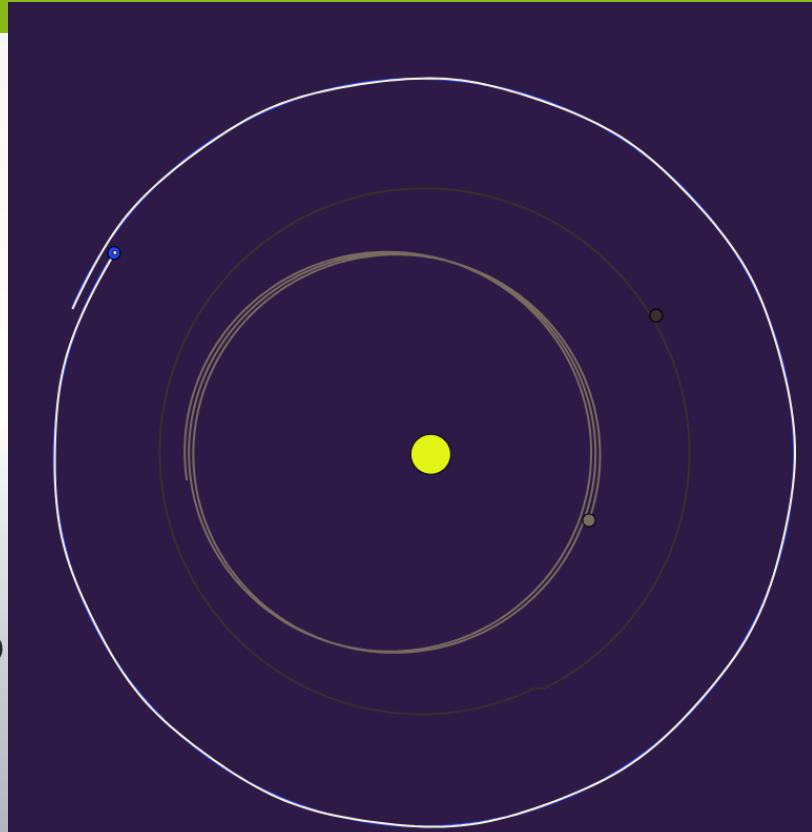
Simulationsparameter modifizieren

- Slider zeigen aktuellen Wert der Simulationsparameter
⇒ pausiere GUI, um parameter zu ändern
- Problem: 1 Loop
⇒ zeichnen wird auch pausiert
⇒ unschöner Sprung
- 2 Lösungen:
 - Simulation pausieren
 - 2 Loops: einer zum Zeichnen, der andere für Parameter



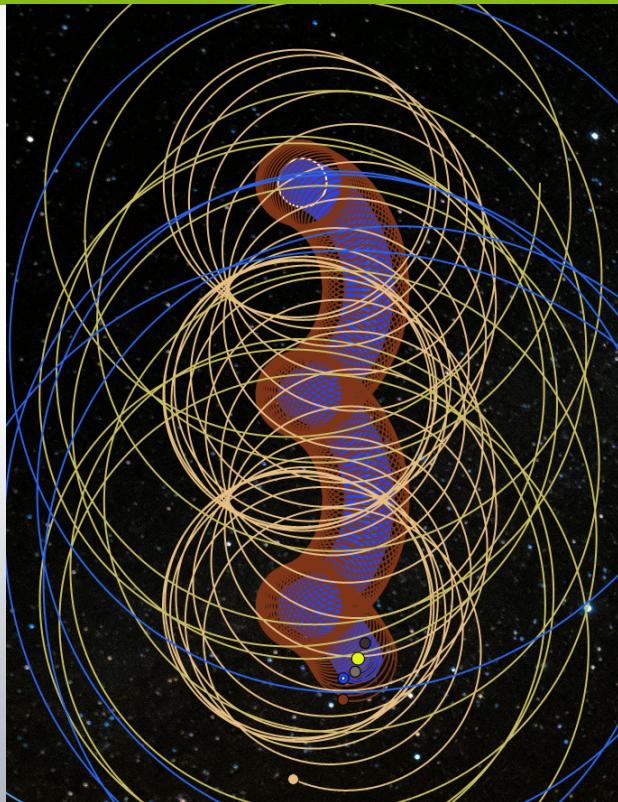
Umlaufbahnen

- Speichere alle Position um die Flugbahn zu zeichnen
- Problem:
 - ewige Datenspeicherung
 - Speicher läuft voll
 - Performance nimmt drastisch ab



Umlaufbahnen Optimisation

- Pollingrate proportional zum Radius
⇒ Weniger Datenpunkte
- Pfade schließen
⇒ Datenmenge beschränkt
- Beschleunigte Bewegung des Systems
⇒ Pfade nur selten schließbar



GUI - Loop

```
const loop = (time: number) => {
    const delta = time - lastRender;
    if (delta >= 1000 / FPSTARGET) {
        updateFunctions.forEach((f) => f(delta));
        lastRender = time;
    }
    if (!end) window.requestAnimationFrame(loop);
};

const registerOnUpdate = (...onUpdate: ((delta: number) => any)[]) => {
    updateFunctions.push(...onUpdate);
};
```

GUI - EventBus

```
const eventTypes = ["togglePlay", "resetCam"] as const;
type Events = (typeof eventTypes)[number];
type EventDefinitions = {
    togglePlay: { play: boolean };
    resetCam: {};
};

const fireEvent = <K extends Events>(
    eventType: K,
    event: EventDefinitions[K]
) => {
    registeredFunctions[eventType].forEach((l) => l(event));
};
```