17.2.2023

Dino Bomb

Design Dokument

Jan Christmeier
263164
MIB
PRIMA
WS 2022/2023

Kurzbeschreibung

Bei dem Spiel handelt es sich um ein kleines 2D Spiel, welches durch Worms und Konsorten inspiriert wurde. Allerdings handelt sich hierbei um ein in Echtzeit stattfindenden Kampf zwischen den Charakteren. Jeder Charakter kann eine Bombe werfen. Bei Treffer, wird dem Getroffenen Leben abgezogen. Dies geht so lange, bis kein Charakter mehr übrig ist. Charaktere werden durch eine config.json Datei erstellt und konfiguriert.

Bewertungskriterien

Units and Positions

Null ist genau in der Mitte der Welt (0/0/0). Eins ist eine Standarteinheit von Fudge. Ein Charakter und auch eine Bombe sind von Grund auf in der Mitte aufgehängt. Ein Charakter ist 1 groß. Somit muss einmal per Code eine Verschiebung um 0,5 stattfinden.

Hierarchy

- World (Graph)
    - Böl+ackground
    - Ground (Graph)
        - mainland
            - border_left
            - border_right
        - water

Charaktere und Bomben werden als Kinder des World Graphen erzeugt. Somit sind diese komplett unabhängig. Ansonsten hat diese Hierarchie eher nur Vorteile, falls diese im Editor bearbeitet werden müssen. Vom Spiel her oder vom Code her ergibt sich kein nennenswerter Vorteil durch diesen Aufbau

Editor

Alle statischen Elemente sind per Editor erstellt. Alle dynamischen Objekte sind per Code erstellt worden. Da in diesem Prototyp sehr viele unterschiedliche dynamische Objekte gleichzeitig angezeigt werden, dynamisch erzeugt und entfernt werden können, ist hier nur eine Umsetzung per Code sinnvoll. Die anderen Objekte müssen nicht derartig dynamisch sein und können so einfacher im Editor erstellt werden

Scriptcomponents

Jedem vierten erzeugten Charakter wird eine Skripskomponente per Code hinzugefügt, welche den Charakter drehen lässt. In diesem Prototyp hat diese Komponente allerdings keinen wirklichen Mehrwert

Extend

Charakter und Bombe sind eigene Klassen, welche Kinderklassen von ƒ.Node sind. Diese Struktur macht es nicht nur einfacher zu programmieren, sie ist für den Prototypen auch nahezu erforderlich. Nur so lässt sich vernünftig eine beliebige Anzahl an Charakteren erzeugen, welche ihre eigenen Bomben abzuschießen haben.

Sound

Es gibt eine Hintergrundmusik, welche in Dauerschleife läuft. Weiterhin gibt es einen kurzen Soundeffekt, wenn eine Bombe abgefeuert wird. Da es sich um ein 2D Game, bei dem eine Richtung des Sounds nahezu irrelevant ist, sind beide Sounds am Hauptgraphen angeheftet

VUI

Auf der linken Seite neben dem Spielfenster werden die erzeugten Charakter-Namen und deren aktuelle Lebenspunkte angezeigt. Die Anzahl der Anzeigen entspricht der Anzahl der erzeugten Charaktere. Bei einem Treffer von einer Bombe werden 25 Leben des Chars abgezogen. Bei null Leben übrig, wird der entsprechende Charakter aus der Welt entfernt

Event-System

Wenn eine Wand eine Bombe per Physik-Event erkennt, soll das Licht im Hauptgraphen seine Farbe ändern. Da die Wände weiter unten in der Hierarchie sind, als das Licht, wird auf Dieses mit mehreren getParent() Befehlen zugegriffen.

External Data

Die Konfigurationsdatei generiert die Charaktere. In dieser kann ein neuer Eintrag des Charakters-Array eingefügt werden, welcher alle relevanten Daten eines Charakters aufweist.

Dazu gehören: Name, Steuerung. Startkoordinaten, Drehrichtung und Masse.

So können beliebig viele Charaktere erzeugt werden.

Light

Der World-Graph besitzt eine Lichtkomponente, welche dem Boden (Shader Phong) seine Farbe gibt. Weiterhin besitzt jede Bombe eine Point-Light Komponente, welche per Code der Bombe hinzugefügt wird. Eine so erzeugt Bombe erhellt ein wenig ihrer Umgebung, was am Shader-Phong des Bodens gut zu sehen ist. Das Ambient-Light des Grafen ändert sich bei Berührungen einer Bombe mit der Außenwand. Somit Ändert sich für den Spielenden die Farbe des Bodens.

Einen spielerischen Sinn oder Mehrwert hat Licht in diesem Prototypen nicht. Es wäre alles problemlos mit Shader-Lit umsetzbar.

Physics

Charakter, Bomben, sowie relevante Teile der Welt (um runterfallen zu vermeiden) verwenden Rigidbodys. Die der Welt sind im Editor hinzugefügt worden. Die der Charaktere und Bomben sind per Code generiert. Somit können die verschiedenen Objekte miteinander interagieren und unter anderem über das Physics-Event-System Kollisionen erkennen.

Einige Parameter der Physik können über die Konfigurationsdatei geändert werden. Laufgeschwindigkeit und Sprunghöhe sind jedoch Masseunabhängig.

Bewegung und Schießen funktioniert über Kräfte, welche den Bomben oder den Charakteren gegeben werden.

Net

Wurden nicht verwendet

State Machines

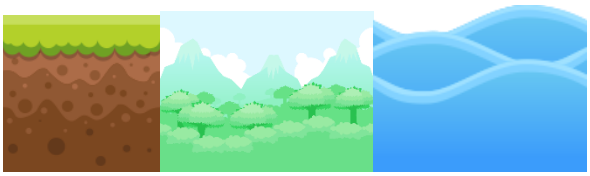
Wurden nicht verwendet

Animation

Allen erzeugten Charakteren werden Lauf- und Idle-Animationen zugewiesen. Dabei werden 4 unterschiedliche Sprite-Sheets verwendet. Wenn die Charaktere in der Schleife erzeugt werden, wird nach dem vierten Sprite-Sheet wieder das erste genommen.
=> erster und fünfter Charakter benutzen dasselbe Sheet. Genauso wie der zweite und sechste.

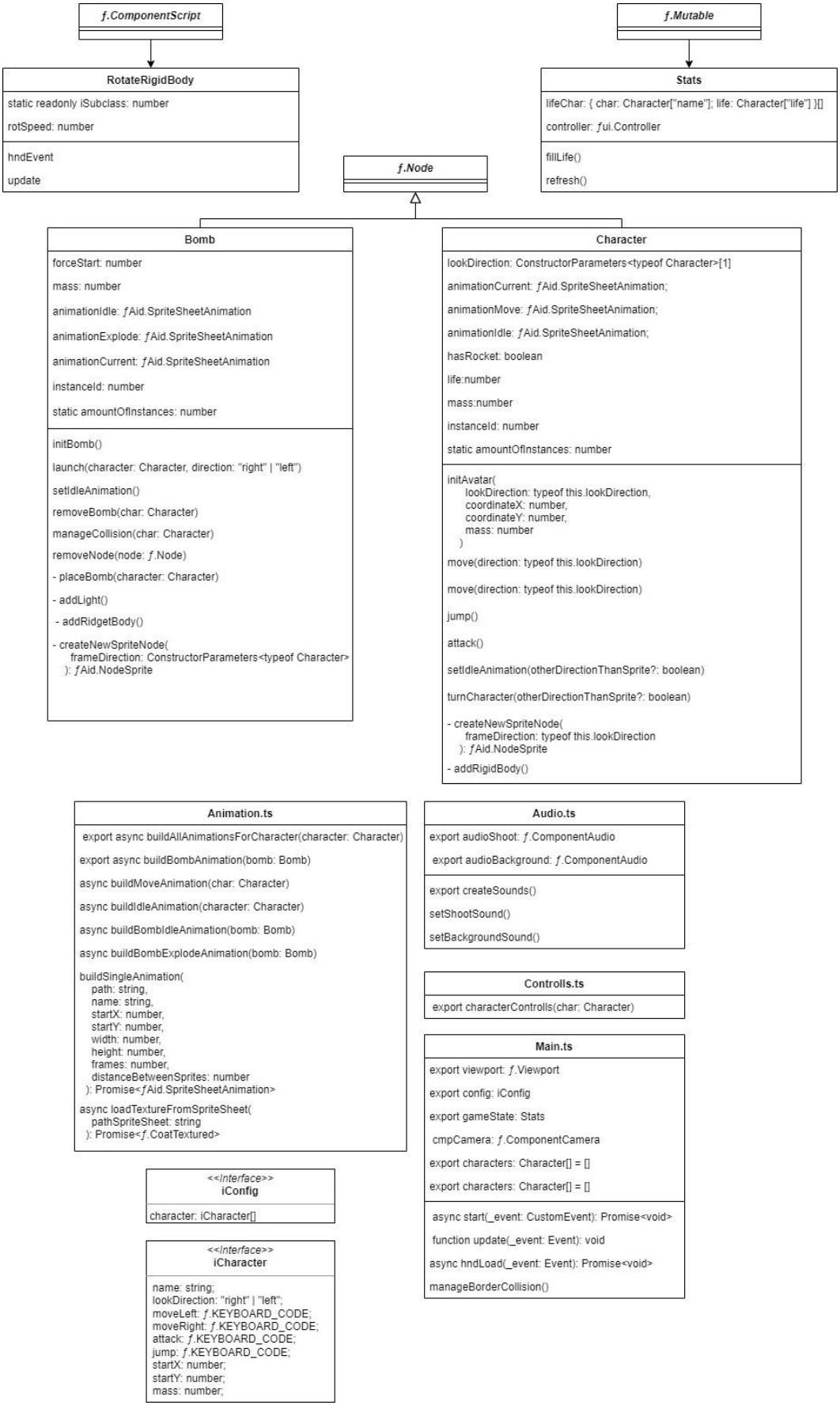
Assets

World



Sprites

Aufbau Code


ƒ.ComponentScript
RotateRigidBody
static readonly iSubclass: number
rotSpeed: number
hndEvent
update
ƒ.Mutable
Stats
lifeChar: { char: Character["name"]; life: Character["life"] }[]
controller: ƒui.Controller
fillLife()
refresh()
ƒ.Node
Bomb
forceStart: number
mass: number
animationIdle: ƒAid.SpriteSheetAnimation
animationExplode: ƒAid.SpriteSheetAnimation
animationCurrent: ƒAid.SpriteSheetAnimation
instanceId: number
static amountOfInstances: number
initBomb()
launch(character: Character, direction: "right" | "left")
setIdleAnimation()
removeBomb(char: Character)
manageCollision(char: Character)
removeNode(node: ƒ.Node)
- placeBomb(character: Character)
- addLight()
- addRidgetBody()
- createNewSpriteNode(
frameDirection: ConstructorParameters<typeof Character>
): ƒAid.NodeSprite
Character
lookDirection: ConstructorParameters<typeof Character>[1]
animationCurrent: ƒAid.SpriteSheetAnimation;
animationMove: ƒAid.SpriteSheetAnimation;
animationIdle: ƒAid.SpriteSheetAnimation;
hasRocket: boolean
life:number
mass:number
instanceId: number
static amountOfInstances: number
initAvatar(
lookDirection: typeof this.lookDirection,
coordinateX: number,
coordinateY: number,
mass: number
)
move(direction: typeof this.lookDirection)
move(direction: typeof this.lookDirection)
jump()
attack()
setIdleAnimation(otherDirectionThanSprite?: boolean)
turnCharacter(otherDirectionThanSprite?: boolean)
- createNewSpriteNode(
frameDirection: typeof this.lookDirection
): ƒAid.NodeSprite
- addRigidBody()
Animation.ts
export async buildAllAnimationsForCharacter(character: Character)
export async buildBombAnimation(bomb: Bomb)
async buildMoveAnimation(char: Character)
async buildIdleAnimation(character: Character)
async buildBombIdleAnimation(bomb: Bomb)
async buildBombExplodeAnimation(bomb: Bomb)
buildSingleAnimation(
path: string,
name: string,
startX: number,
startY: number,
width: number,
height: number,
frames: number,
distanceBetweenSprites: number
): Promise<ƒAid.SpriteSheetAnimation>
async loadTextureFromSpriteSheet(
pathSpriteSheet: string
): Promise<ƒ.CoatTextured>
Audio.ts
export audioShoot: ƒ.ComponentAudio
export audioBackground: ƒ.ComponentAudio
export createSounds()
setShootSound()
setBackgroundSound()
Controlls.ts
export characterControlls(char: Character)
Main.ts
export viewport: ƒ.Viewport
export config: iConfig
export gameState: Stats
cmpCamera: ƒ.ComponentCamera
export characters: Character[] = []
export characters: Character[] = []
async start(_event: CustomEvent): Promise<void>
function update(_event: Event): void
async hndLoad(_event: Event): Promise<void>
manageBorderCollision()
<<Interface>>
iConfig
character: iCharacter[]
<<Interface>>
iCharacter
name: string;
lookDirection: "right" | "left";
moveLeft: ƒ.KEYBOARD_CODE;
moveRight: ƒ.KEYBOARD_CODE;
attack: ƒ.KEYBOARD_CODE;
jump: ƒ.KEYBOARD_CODE;
startX: number;
startY: number;
mass: number;