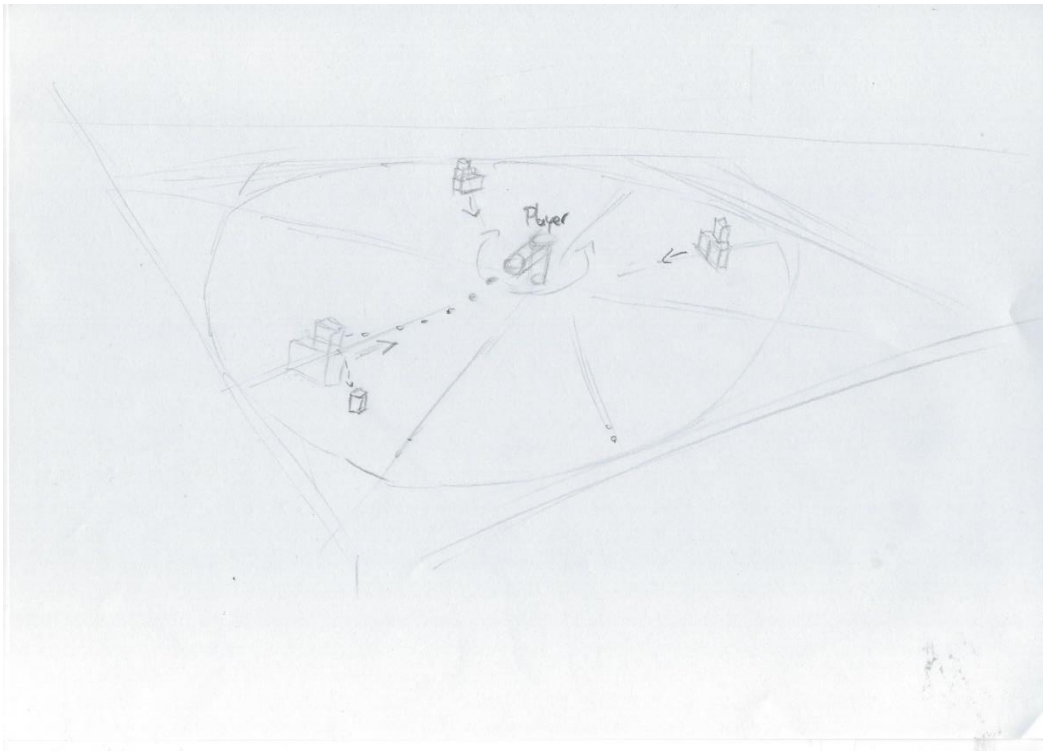
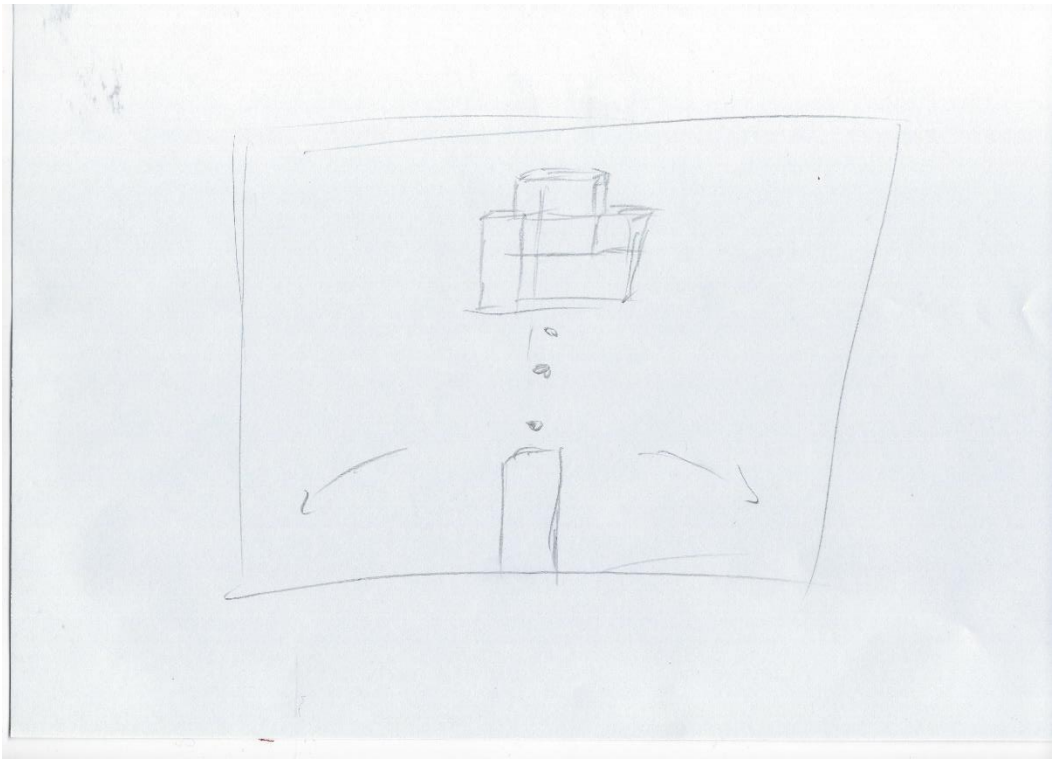
Konzept des 360-Grad-Tower-Defender

Idee: Physikbeinhaltendes 360 Grad Tower Defense Spiel. Der Spieler spielt eine Kanonenturm in der Mitte des Spielfeldes. Rundherum um den Spieler tauchen auf einzelnen Lanes Geometrieblöcke auf, die auf den Spieler zusteuern. Ziel ist es diese Geometrien zu zerstören und sich somit zu schützen



[Vogelperspektive des Spielfeldes]



[Ego/Spielerperspektive]

Umsetzung:

Zuerst einmal wollte ich die Arena, in der sich das ganze befindet, erstellen. Bevor ich aber die Umgebung erstellte, kümmerte ich mich um eine GameRoot auf der alles aufbaut.

```
let gameRoot: ƒ.Node = new ƒ.Node("GameRoot");
```

Dann war es von Nöten, etwas zu sehen, weswegen ich ein Kamera Setup erstellte, welche später dann um die Spielerausrichtung ergänzt wurde, so konnte ich für den Umgebungsaufbau eine Vogelperspektive und für das Spielen die Spielerausrichtung der Kamera benutzen

```
//Init first Camera Setup
    //cmpCamera.mtxPivot.translateZ(20.5); → Vogelperspektive
    //cmpCamera.mtxPivot.rotateX(0);
    cmpCamera.mtxPivot.translateZ(4.5); →Spielerperspektive
    cmpCamera.mtxPivot.rotateX(75);
    cmpCamera.mtxPivot.rotateY(180);
    console.log(cmpCamera);
```

Danach folgten Boden und Wände, um so eine Basis zu haben, auf der ich alles andere aufbaue. Hierfür gibt es in der Init-Methode eine createBoden() Methode, welche diesen erstellt.

```
 function createBoden():void{
    let material_Boden: ƒ.Material = new ƒ.Material("Boden_Color", ƒ.ShaderUni
Color, new ƒ.CoatColored(new ƒ.Color(1, 1, 1, 1)));
    let cmpMaterialBoden: ƒ.ComponentMaterial = new ƒ.ComponentMaterial(materi
al_Boden);

    let meshBoden: ƒ.MeshCube = new ƒ.MeshCube("Boden");
    let cmpMeshBoden: ƒ.ComponentMesh = new ƒ.ComponentMesh(meshBoden);


    let transformBoden:  ƒ.ComponentTransform = new ƒ.ComponentTransform();
    transformBoden.mtxLocal.scale(new ƒ.Vector3(20, 20, 1));

    boden.addComponent(transformBoden);
    boden.addComponent(cmpMaterialBoden);
    boden.addComponent(cmpMeshBoden);
    boden.addComponent(new ƒ.ComponentRigidbody(1, ƒ.PHYSICS_TYPE.STATIC, ƒ.CO
LLIDER_TYPE.CUBE, ƒ.PHYSICS_GROUP.DEFAULT));
    boden.getComponent(ƒ.ComponentRigidbody).addEventListener(ƒ.EVENT_PHYSICS.
COLLISION_ENTER, handleCollision);
  }
```

Auch ein Rigidbody wird dem Boden als Komponente angehängt, für die spätere Kollisionsabfrage.

```
function createWalls(): void {
    //Wand Material
    let material_Wand: ƒ.Material = new ƒ.Material("Wand_Color", ƒ.ShaderUniCo
lor, new ƒ.CoatColored(new ƒ.Color(0.5, 1, 0, 1)));
    let cmpMaterialWand_1: ƒ.ComponentMaterial = new ƒ.ComponentMaterial(mater
ial_Wand);
    let cmpMaterialWand_2: ƒ.ComponentMaterial = new ƒ.ComponentMaterial(mater
ial_Wand);
    let cmpMaterialWand_3: ƒ.ComponentMaterial = new ƒ.ComponentMaterial(mater
ial_Wand);
    let cmpMaterialWand_4: ƒ.ComponentMaterial = new ƒ.ComponentMaterial(mater
ial_Wand);

    //Wand Mesh
    let meshWand: ƒ.MeshCube = new ƒ.MeshCube("Wand");
    let cmpMeshWand_1: ƒ.ComponentMesh = new ƒ.ComponentMesh(meshWand);
    let cmpMeshWand_2: ƒ.ComponentMesh = new ƒ.ComponentMesh(meshWand);
    let cmpMeshWand_3: ƒ.ComponentMesh = new ƒ.ComponentMesh(meshWand);
    let cmpMeshWand_4: ƒ.ComponentMesh = new ƒ.ComponentMesh(meshWand);
    //Erstelle Wände

    let transformWand_1:  ƒ.ComponentTransform = new ƒ.ComponentTransform();
    transformWand_1.mtxLocal.scale(new ƒ.Vector3(1, 20, 10));
    transformWand_1.mtxLocal.translateX(10);

    let transformWand_2:  ƒ.ComponentTransform = new ƒ.ComponentTransform();
    transformWand_2.mtxLocal.scale(new ƒ.Vector3(1, 20, 10));
    transformWand_2.mtxLocal.translateX(-10);

    let transformWand_3:  ƒ.ComponentTransform = new ƒ.ComponentTransform();
    transformWand_3.mtxLocal.scale(new ƒ.Vector3(1, 20, 10));
    transformWand_3.mtxLocal.rotation = new ƒ.Vector3(0, 0, 90);
    transformWand_3.mtxLocal.translateX(10);

    let transformWand_4:  ƒ.ComponentTransform = new ƒ.ComponentTransform();
    transformWand_4.mtxLocal.scale(new ƒ.Vector3(1, 20, 10));
    transformWand_4.mtxLocal.rotation = new ƒ.Vector3(0, 0, 90);
    transformWand_4.mtxLocal.translateX(-10);

    let transformWaende: ƒ.ComponentTransform = new ƒ.ComponentTransform();
    waende.addComponent(transformWaende);


    wand_1.addComponent(transformWand_1);
    wand_1.addComponent(cmpMaterialWand_1);
    wand_1.addComponent(cmpMeshWand_1);
    wand_1.addComponent(new ƒ.ComponentRigidbody(1, ƒ.PHYSICS_TYPE.STATIC, ƒ.C
OLLIDER_TYPE.CUBE, ƒ.PHYSICS_GROUP.DEFAULT));
```

```
    wand_2.addComponent(transformWand_2);
    wand_2.addComponent(cmpMaterialWand_2);
    wand_2.addComponent(cmpMeshWand_2);
    wand_2.addComponent(new ƒ.ComponentRigidbody(1, ƒ.PHYSICS_TYPE.STATIC, ƒ.C
OLLIDER_TYPE.CUBE, ƒ.PHYSICS_GROUP.DEFAULT));



    wand_3.addComponent(transformWand_3);
    wand_3.addComponent(cmpMaterialWand_3);
    wand_3.addComponent(cmpMeshWand_3);
    wand_3.addComponent(new ƒ.ComponentRigidbody(1, ƒ.PHYSICS_TYPE.STATIC, ƒ.C
OLLIDER_TYPE.CUBE, ƒ.PHYSICS_GROUP.DEFAULT));



    wand_4.addComponent(transformWand_4);
    wand_4.addComponent(cmpMaterialWand_4);
    wand_4.addComponent(cmpMeshWand_4);
    wand_4.addComponent(new ƒ.ComponentRigidbody(1, ƒ.PHYSICS_TYPE.STATIC, ƒ.C
OLLIDER_TYPE.CUBE, ƒ.PHYSICS_GROUP.DEFAULT));


    waende.addChild(wand_1);
    waende.addChild(wand_2);
    waende.addChild(wand_3);
    waende.addChild(wand_4);

  }
```

Am Ende werden alle Wände zu der Node “waende“ hinzugefügt.

Danach folgte die Kanone, welche danach als Komponente noch die MainKamera bekommt.

```
//Init Canon / Player
    createCanon();
```

```
function createCanon(): void{
    //Init Canon / Player
    let meshCanon: ƒ.MeshCube = new ƒ.MeshCube("Cube_Player");
    let transformCanon: ƒ.ComponentTransform = new ƒ.ComponentTransform(new ƒ.
Matrix4x4());
    transformCanon.mtxLocal.translateZ(0);
    let cmpMeshCanon = (new ƒ.ComponentMesh(meshCanon));
```

```
    let canon_material: ƒ.Material = new ƒ.Material("Canon_Color", ƒ.ShaderUni
Color, new ƒ.CoatColored(new ƒ.Color(0, 1, 0, 1)));
    let canon_cmpMaterial: ƒ.ComponentMaterial = new ƒ.ComponentMaterial(canon
_material);
    canon.addComponent(canon_cmpMaterial);
    canon.addComponent(cmpMeshCanon);
    canon.addComponent(transformCanon);
    canon.appendChild(kugel_spawner);
    canon.addComponent(cmpCamera);
  }
```

Der hier zu findende Kugelspawner erzeuge ich durch diese Funktion

```
function createKugelSpawner(): void{
    let material_KS: ƒ.Material = new ƒ.Material("KS_Color", ƒ.ShaderUniColor,
 new ƒ.CoatColored(new ƒ.Color(0, 0, 1, 1)));
    let cmpMaterialKS: ƒ.ComponentMaterial = new ƒ.ComponentMaterial(material_
KS);
    let meshKugelSpawner: ƒ.MeshCube = new ƒ.MeshCube("KugelSpawner");
    let cmpMeshKS: ƒ.ComponentMesh = new ƒ.ComponentMesh(meshKugelSpawner);
    let transformKugelSpawner:  ƒ.ComponentTransform = new ƒ.ComponentTransfor
m();
    transformKugelSpawner.mtxLocal.translate(new ƒ.Vector3(0, 3, 3.5));
    transformKugelSpawner.mtxLocal.scale(new ƒ.Vector3(.25, .25, .25));
    kugel_spawner.addComponent(transformKugelSpawner);
    kugel_spawner.addComponent(cmpMaterialKS);
    kugel_spawner.addComponent(cmpMeshKS);
  }
```

Dieser dient dem Spieler als Orientierung für seine Geschosse.

Nun alles noch der GameRoot als Child hinzufügen.

```
//Appending Children to GameRoot
    gameRoot.appendChild(boden);
    gameRoot.appendChild(canon);
    gameRoot.appendChild(waende);
    console.log(gameRoot);
```

Danach wollte ich eine Steuerung einbauen, in der ich die Maus für die Rotation der Kamera benutzen kann.

```
let ctrRotationY: ƒ.Control = new ƒ.Control("AvatarRotationY", -
0.1, ƒ.CONTROL_TYPE.PROPORTIONAL);
  ctrRotationY.setDelay(100);
  let ctrRotationX: ƒ.Control = new ƒ.Control("AvatarRotationX", -
0.1, ƒ.CONTROL_TYPE.PROPORTIONAL);
  ctrRotationX.setDelay(100);
```

```
//Init Mouse Listener
    canvas.addEventListener("mousemove", hndMouse);
    canvas.addEventListener("click", canvas.requestPointerLock);
```

```
 function hndMouse(_event: MouseEvent): void {
    // console.log(_event.movementX, _event.movementY);
    ctrRotationX.setInput(_event.movementX);
    ctrRotationY.setInput(_event.movementY);
  }
```

```
 canon.mtxLocal.rotateZ(ctrRotationX.getOutput()); //Z Achse weil Objekt gedre
ht wurde --> X Achse ist jetzt die Z Achse
      canon.mtxLocal.rotateX(ctrRotationY.getOutput());
      canon.mtxLocal.rotation = new ƒ.Vector3(canon.mtxLocal.rotation.x, 0, ca
non.mtxLocal.rotation.z);
      ctrRotationX.setInput(0);
      ctrRotationY.setInput(0);
```

Ein bisschen "tricky" war hierbei die Rotation in zwei Achsen zur (fast) selben Zeit, was darin resultierte das auch indirekt um die 3.Achse rotiert wird. Um das zu verhindern, setze ich bei jedem Durchgang eben diese 3.Achse auf 0, um ein gutes Spielgefühl zu erzeugen, was die Rotation der Kanone angeht.

Danach war es wichtig die Physik mit ins Spiel aufzunehmen und die Gravitation auf die richtige Achse zu mappen, sodass die Kugeln auch nach unten fliegen.

```
//Physic Settings
    ƒ.Physics.settings.defaultRestitution = 0.5;
    ƒ.Physics.settings.defaultFriction = 0.8;
    ƒ.Physics.world.setGravity(new ƒ.Vector3(0, 0, -.55) );
    ƒ.Physics.settings.debugDraw = true;
    ƒ.Physics.adjustTransforms(gameRoot);
```

Auch den Viewport zu initialisieren, darf nicht fehlen.

```
//Init Viewport
    viewport.initialize("Viewport", gameRoot, cmpCamera, canvas);
    viewport.draw();
```

Dann ging es an die Erstellung der einzelnen Objektklassen, angefangen mit der Einzelgeometrie

```
namespace Endabgabe_360_Defender {
  import ƒ = FudgeCore;

  export enum CUBE_TYPE {
    GREEN = "Green",
    RED = "Red",
    BLUE = "Blue",
    YELLOW = "Yellow",
    MAGENTA = "Magenta",
    CYAN = "Cyan"
}
  type Materials = Map<CUBE_TYPE, ƒ.Material>

  export class Einzelgeometrie extends ƒ.Node {
    static _root:ƒ.Node;
    private static materials: Materials = Einzelgeometrie.createMaterials();
    private static mesh: ƒ.Mesh = new ƒ.MeshCube("Cube");
    rigidbody: ƒ.ComponentRigidbody = new ƒ.ComponentRigidbody(1, ƒ.PHYSICS_TY
PE.KINEMATIC, ƒ.COLLIDER_TYPE.CUBE, ƒ.PHYSICS_GROUP.DEFAULT);
    direction: boolean ;

    constructor(_name: string, _pos: ƒ.Vector3, _scale: ƒ.Vector3, _dir: boole
an, _type: CUBE_TYPE, root: ƒ.Node) {
      super(_name);

      Einzelgeometrie._root = root;

      this.addComponent(new ƒ.ComponentTransform());
      this.mtxLocal.translateX(_pos.x);
      this.mtxLocal.translateY(_pos.y);
      this.mtxLocal.translateZ(_pos.z);

      let cmpMesh: ƒ.ComponentMesh = new ƒ.ComponentMesh(Einzelgeometrie.mesh)
;
      cmpMesh.mtxPivot.scaleX(_scale.x);
      cmpMesh.mtxPivot.scaleY(_scale.y);
      cmpMesh.mtxPivot.scaleZ(_scale.z);

      this.direction = _dir;


      this.addComponent(cmpMesh);

      let cmpMaterial: ƒ.ComponentMaterial = new ƒ.ComponentMaterial(Einzelgeo
metrie.materials.get(_type));
      this.addComponent(cmpMaterial);
      this.addComponent(this.rigidbody);
```

```
      this.rigidbody.addEventListener(ƒ.EVENT_PHYSICS.COLLISION_ENTER, this.ha
ndleCollision);
    }

    private static createMaterials(): Materials {
      return new Map([
          [CUBE_TYPE.RED, new ƒ.Material(CUBE_TYPE.RED, ƒ.ShaderUniColor, new
ƒ.CoatColored(ƒ.Color.CSS("RED")))],
          [CUBE_TYPE.GREEN, new ƒ.Material(CUBE_TYPE.GREEN, ƒ.ShaderUniColor,
new ƒ.CoatColored(ƒ.Color.CSS("GREEN")))],
          [CUBE_TYPE.BLUE, new ƒ.Material(CUBE_TYPE.BLUE, ƒ.ShaderUniColor, ne
w ƒ.CoatColored(ƒ.Color.CSS("BLUE")))],
          [CUBE_TYPE.MAGENTA, new ƒ.Material(CUBE_TYPE.MAGENTA, ƒ.ShaderUniCol
or, new ƒ.CoatColored(ƒ.Color.CSS("MAGENTA")))],
          [CUBE_TYPE.YELLOW, new ƒ.Material(CUBE_TYPE.YELLOW, ƒ.ShaderUniColor
, new ƒ.CoatColored(ƒ.Color.CSS("YELLOW")))],
          [CUBE_TYPE.CYAN, new ƒ.Material(CUBE_TYPE.CYAN, ƒ.ShaderUniColor, ne
w ƒ.CoatColored(ƒ.Color.CSS("CYAN")))]
      ]);
  }

    public setTransform(_pos: ƒ.Vector3, _rot: ƒ.Vector3): void {
      this.mtxLocal.translateX(_pos.x);
      this.mtxLocal.translateY(_pos.y);
      this.mtxLocal.translateZ(_pos.z);

      this.mtxLocal.rotateX(_rot.x);
      this.mtxLocal.rotateY(_rot.y);
      this.mtxLocal.rotateZ(_rot.z);

    }



    public move(): void {
      if (!this.direction)
      this.mtxLocal.translateX( - 1 / 4 * ƒ.Loop.timeFrameReal / 1000);
      else
      this.mtxLocal.translateX(  1 / 4 * ƒ.Loop.timeFrameReal / 1000);
    }

    public handleCollision (_event: ƒ.EventPhysics): void
    {
      if(_event.cmpRigidbody.getContainer().name == "Kugel")
      {
        /*let root: ƒ.Node = _event.cmpRigidbody.getContainer().getParent();s
        root.removeChild(_event.cmpRigidbody.getContainer());*/
      }

      if(_event.cmpRigidbody.getContainer().name == "boden")
```

```
      {
        console.log("BodenKoll");
        Einzelgeometrie._root.removeChild(this);
      }
    }


    private activatePhysics(): void {
      //console.log("Col");
      this.rigidbody.physicsType = ƒ.PHYSICS_TYPE.DYNAMIC;
    }

  }
}
```

Hierbei ist der angesprochene ENUM für die Farbe enthalten. Auch eine Kollisionsabfrage um zu checken, mit was gerade kollidiert wird ist enthalten.

Danach ging es an die Gegnergeometrie, welche aus einzelnen Einzelgeometrien besteht.

```
namespace Endabgabe_360_Defender {
  import ƒ = FudgeCore;

  export class Gegnergeometrie extends ƒ.Node  {

    direction: boolean;

    constructor(_name: string, _pos: ƒ.Vector3, _scale: ƒ.Vector3, _dir: boole
an, _count: number) {
      super(_name);

      this.direction = _dir;

      let rand_1 : number;
      let rand_2 : number;

      this.addComponent(new ƒ.ComponentTransform());
      this.mtxLocal.translateX(_pos.x);
      this.mtxLocal.translateY(_pos.y);
      this.mtxLocal.translateZ(_pos.z);
      //Setup der Gegnerstruktur
      for ( let i: number = 1; i < _count + 1; i++) {
        if (_count == 3)
        i++;

        // tslint:disable-next-line: typedef
        let positions = new Map();

        for (let j: number = 1; j < _count + 1; j++) {
```

```
          rand_1  = Gegnergeometrie.createRandomNumber(j);
          rand_2  = Gegnergeometrie.createRandomNumber(j);


          if (positions.has(rand_1 + "|" + rand_2)) {

            j--;
          }

          else {
            positions.set(rand_1 + "|" + rand_2 , true);
            let pos: ƒ.Vector3 = new ƒ.Vector3(0, rand_1, rand_2);
            let _type: CUBE_TYPE =  Gegnergeometrie.getRandomEnum(CUBE_TYPE);
            this.appendChild(new Einzelgeometrie(_name, pos, _scale, _dir, _ty
pe, this));

          }
        }
      }
      //this.direction = _dir;
    }

    private static getRandomEnum<T>(_enum: {[key: string]: T}): T {
      let randomKey: string = Object.keys(_enum)[Math.floor(Math.random() * Ob
ject.keys(_enum).length)];
      console.log(randomKey);
      return _enum[randomKey];
  }

    private static createRandomNumber(j: number): number {
      let rand: number = Math.round(Math.random() * j);
      return rand;
    }

    public move(): void {
      if (!this.direction)
      this.mtxLocal.translateX( - 1 / 6 * ƒ.Loop.timeFrameReal / 1000);
      else
      this.mtxLocal.translateX(  1 / 6 * ƒ.Loop.timeFrameReal / 1000);
    }


    public getPosX(): number {
      let pos: number = 3;

      if (!this.direction)
      return pos + this.mtxLocal.translation.x;
      else {
        if (this.mtxLocal.translation.x > 0) {
        return pos - this.mtxLocal.translation.x;
```

```
      }
      else {
        return pos + Math.abs(this.mtxLocal.translation.x);
      }
      }
    }









  }
}
```

Hierbei wird eine Node erstellt, die an zufälligen Positionen in einem vorgegebenen Raster = Anzahl von Einzelgeometrie pro Gegnergeometrie, Geometrie erzeugt, welche der Node als Childs angehängt werden. getPosX() ist für die Abfrage der Position im Verhältnis zur Mitte(Spieler) zuständig. Move() dafür, dass sich die Geometrie auf den Spieler zubewegt.

Jetzt war es noch wichtig die Lanes zu definieren, auf der sich die Gegnergeometrie bewegt.

```
namespace Endabgabe_360_Defender {
  import ƒ = FudgeCore;

  export class QuadLane extends ƒ.Node {
    static mesh: ƒ.Mesh = new ƒ.MeshCube("Lane");
    static material: ƒ.Material = new ƒ.Material("Black", ƒ.ShaderUniColor, ne
w ƒ.CoatColored(new ƒ.Color(1, 0, 0, 1)));

    constructor(_name: string, _pos: ƒ.Vector3, _scale: ƒ.Vector3 ) {
      super(_name);

      this.addComponent(new ƒ.ComponentTransform());
      this.mtxLocal.translateX(_pos.x);
      this.mtxLocal.translateY(_pos.y);
      this.mtxLocal.translateZ(_pos.z);

      let cmpMesh: ƒ.ComponentMesh = new ƒ.ComponentMesh(QuadLane.mesh);
      cmpMesh.mtxPivot.scaleX(_scale.x);
      cmpMesh.mtxPivot.scaleY(_scale.y);
      cmpMesh.mtxPivot.scaleZ(_scale.z);


      this.addComponent(cmpMesh);

      this.addComponent(new ƒ.ComponentMaterial(QuadLane.material));
      this.addComponent(new ƒ.ComponentRigidbody(1, ƒ.PHYSICS_TYPE.KINEMATIC,
ƒ.COLLIDER_TYPE.CUBE, ƒ.PHYSICS_GROUP.DEFAULT));
    }
```

```
    public setTransform(_pos: ƒ.Vector3, _rot: ƒ.Vector3 ): void {
      this.mtxLocal.translateX(_pos.x);
      this.mtxLocal.translateY(_pos.y);
      this.mtxLocal.translateZ(_pos.z);

      this.mtxLocal.rotateX(_rot.x);
      this.mtxLocal.rotateY(_rot.y);
      this.mtxLocal.rotateZ(_rot.z);
    }

  }
}
```

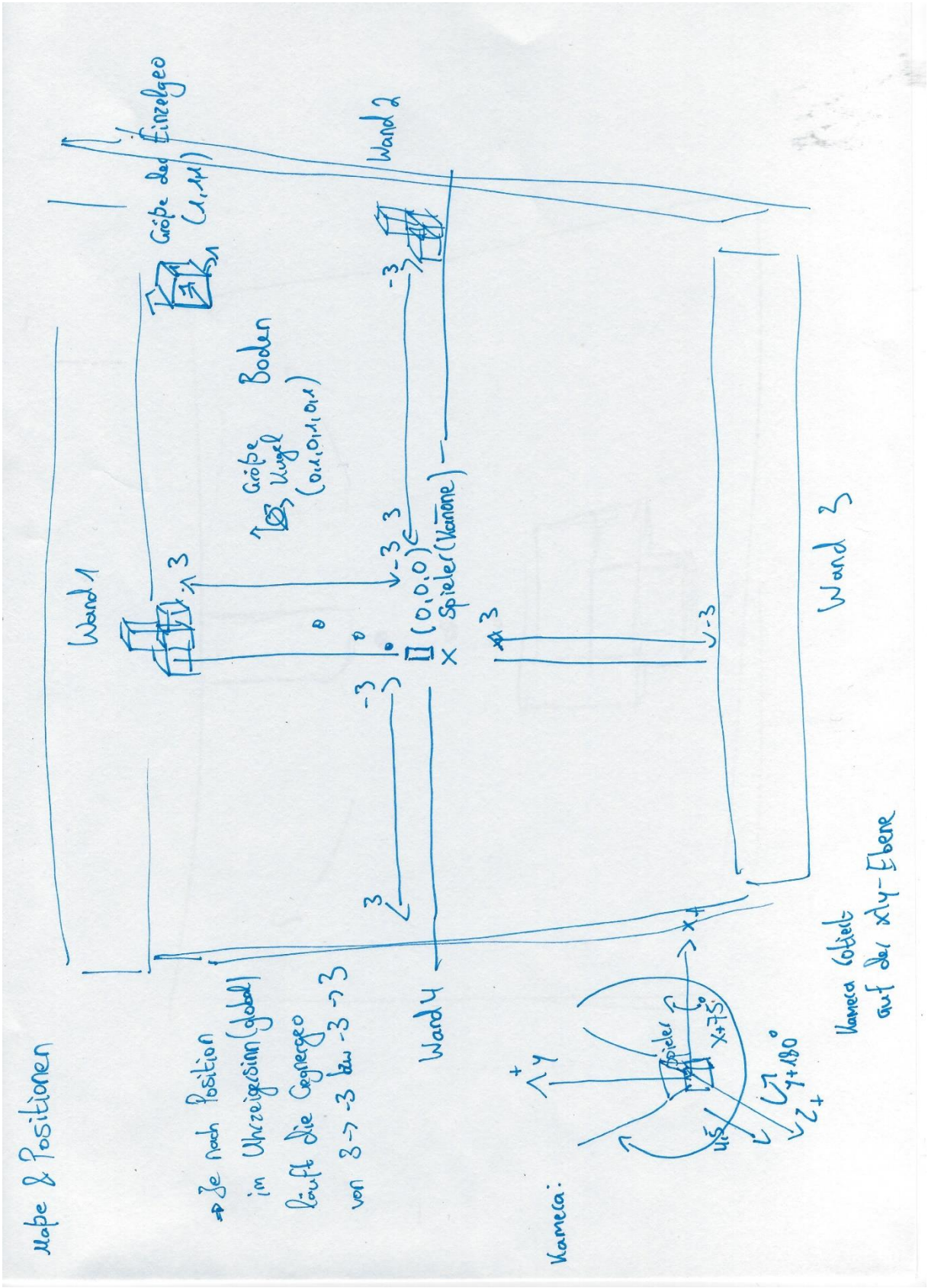
Zum Abschluss musste noch die Kugelklasse erstellt werden, welche das Verhalten der Geschosse enthält, welche der Spieler abschießt.

```
namespace Endabgabe_360_Defender {

  import ƒ = FudgeCore;

  export class Kugeln extends ƒ.Node {
    static mesh: ƒ.Mesh = new ƒ.MeshSphere("Sphere");
    static material: ƒ.Material = new ƒ.Material("Black", ƒ.ShaderUniColor, ne
w ƒ.CoatColored(new ƒ.Color(0, 0, 0, 1)));
    rigidbody: ƒ.ComponentRigidbody = new ƒ.ComponentRigidbody(2, ƒ.PHYSICS_TY
PE.DYNAMIC, ƒ.COLLIDER_TYPE.SPHERE, ƒ.PHYSICS_GROUP.DEFAULT);

    velocity: ƒ.Vector3;


    constructor(_name: string, _pos: ƒ.Vector3, _scale: ƒ.Vector3, _rot: ƒ.Vec
tor3) {
      super(_name);

      let forward : ƒ.Vector3 = new ƒ.Vector3();

      console.log(_rot.x);

      this.velocity = new ƒ.Vector3(_pos.x * 2.5 , _pos.y * 2.5 , .7);

      this.addComponent(new ƒ.ComponentTransform());
      this.mtxLocal.translate(_pos);
      this.mtxLocal.scale(_scale);
      this.mtxLocal.rotate(_rot);

      this.addComponent(this.rigidbody);
      this.rigidbody.restitution = 2;
```

```
      this.rigidbody.addEventListener(ƒ.EVENT_PHYSICS.COLLISION_ENTER, this.ha
ndleCollision);

      let cmpMesh: ƒ.ComponentMesh = new ƒ.ComponentMesh(Kugeln.mesh);
      this.addComponent(cmpMesh);

      this.addComponent(new ƒ.ComponentMaterial(Kugeln.material));

      this.getComponent(ƒ.ComponentRigidbody).setVelocity(this.velocity);
      this.getComponent(ƒ.ComponentRigidbody).setPosition(new ƒ.Vector3(0, 0,
0));

    }


    private handleCollision(_event: ƒ.EventPhysics) : void
    {
      let name: string = _event.cmpRigidbody.getContainer().name;

      //console.log(_event.cmpRigidbody.getContainer().name);

      switch (name)
      {
        case "boden":
        break;

        case "enemy":
        _event.cmpRigidbody.physicsType = ƒ.PHYSICS_TYPE.DYNAMIC;
        _event.cmpRigidbody.applyImpulseAtPoint(new ƒ.Vector3
          (_event.collisionNormal.x * _event.normalImpulse/3, _event.collision
Normal.y * _event.normalImpulse/3,
            _event.collisionNormal.z * _event.normalImpulse/3), _event.collisi
onPoint);

        break;
      }

    }

  }

}
```

Diese bekommt die Velocity entsprechend der globalen Ausrichtung des Kugelspawners, welcher dem Spieler als Orientierung des Abschusspunktes dient. Trifft die Kugel einen enemy = Einzelgeometrie wird dessen Rigidbody Dynamic und bekommt den Impuls der Kugel, um auch in eine logische Richtung, je nach Treffpunkt der Kugel, zu fliegen.

Maße und Positionen:



Szenenhierarchie: