Projektdokumenation

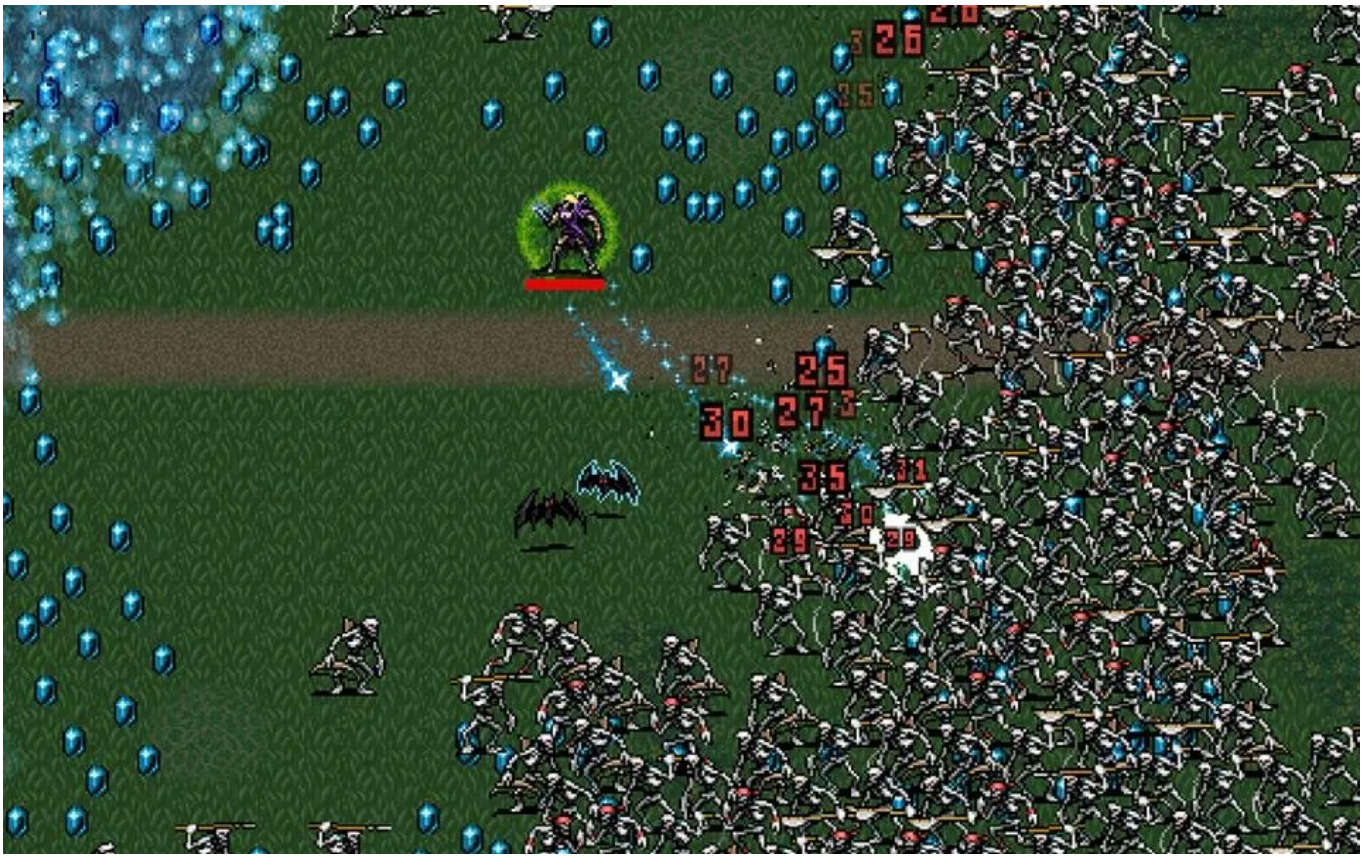
2024

PXIF1 - DR

Cornelius Bonn & Felix Rose

1 - Projektbeschreibung/Zielsetzung/Anforderungen an das Projekt

Das Ziel des Projekts war es, ein einfaches 2D-Spiel zu erstellen, das auf dem Konzept eines Vampire-Suvivors-Like basiert. Das Spiel sollte eine Vielzahl von Funktionen und Mechaniken enthalten, die es dem Spieler ermöglichen, verschiedene Arten von Gegnern zu bekämpfen, verschiedene Waffen und Fähigkeiten zu verwenden und verschiedene Level erreichen.



Vampire Surviviors als Urvater der Vampire-Survivor-Likes

«-<—>-»

Das Ziel des Spiels ist es so weit wie nur möglich in den Dungeon vorzudringen. Ursprünglich war es zwar die Idee einen Vampire-Suvivors-Like zu kreieren, nach einiger Zeit haben wir uns für diese Art von Spiel mehr begeistern können und haben uns deshalb dazu entschieden den Wellenbasierten Ansatz auf einer einzigen Map zu verwerfen und dafür den Ansatz zu wählen, dass man durch verschiedene Räume muss und in jedem eine gewisse Anzahl Gegner zu besiegen hat.

Brotato ein Vampire-Survivor-Like

«-<—>-»

Dass man durch ein Level-Up-System seine Fähigkeiten zu verbessern oder neue erhalten kann, war für uns aber schon von Anfang an im Spiel vorgesehen.
Wir hatten uns das Spiel zuerst farbenfroher vorgestellt mit einer detailliert designten Map allerdings haben wir im verlauf gemerkt, dass wenn wir uns weniger auf die Grafiken konzentrieren mehr noch auf die Spielmechaniken konzentrieren können. Dazu kam, dass wir gefallen an dem "einfachen" Stil unseres Spiel gefunden haben und deshalb uns unter anderem auch dazu entschieden haben die Gegner bei farbigen Vierecken zu belassen. Der einfache Stil war zwar Ursprünglich nicht geplant aber gibt unserer Meinung nach dem Spiel einen eigenen Charm der den Fokus mehr darauf legt, Fortschritt im Spiel zu erzielen, und weniger darauf sich schön designte Aspekte anzusehen.

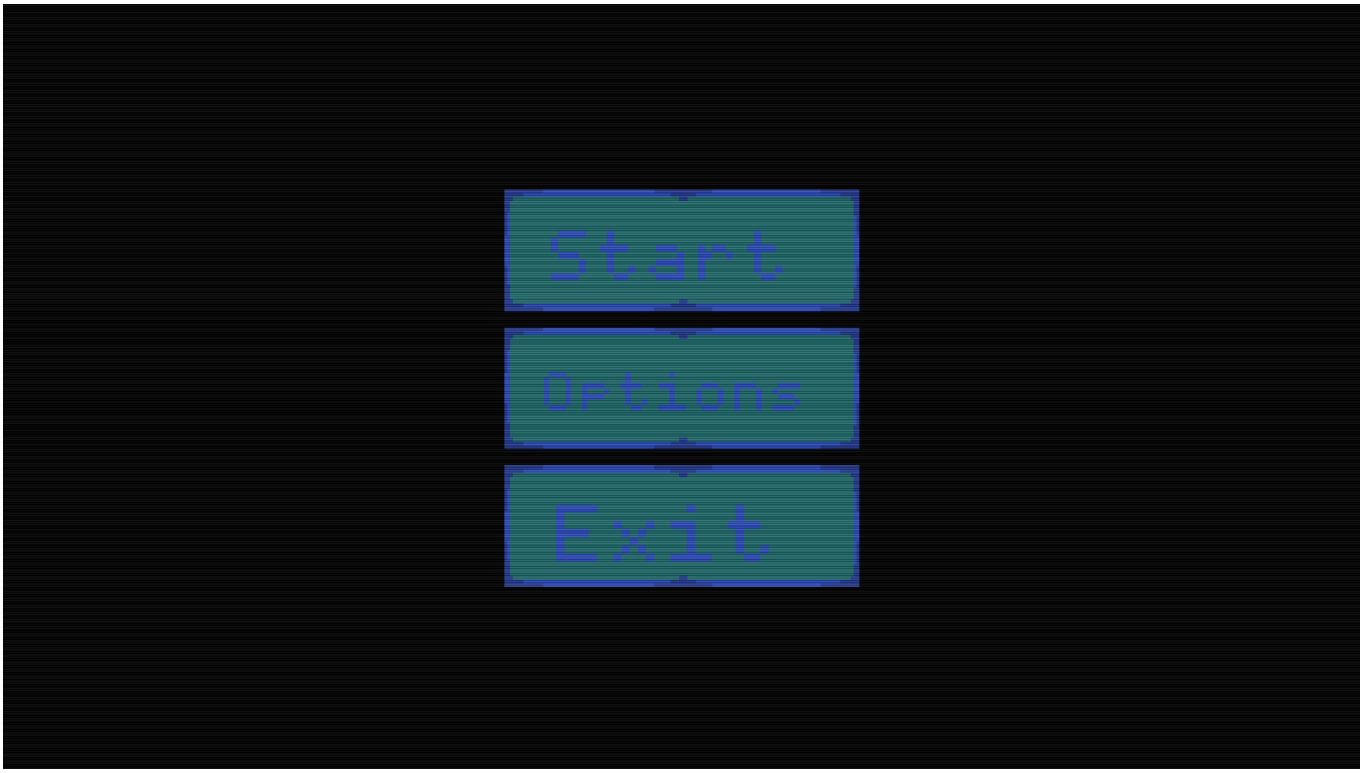
Beispiel Upgrade Auswahl

Außerdem hilft das simple Design dabei, sich mit den Projektilen zurechtzufinden, da diese in unserem Spiel teilweise sehr viele werden könnten, somit wird der Fokus mehr auf kompetitives Spielen gelegt als auf schönes Design.

Beispiel Gegner

«-<—>-»

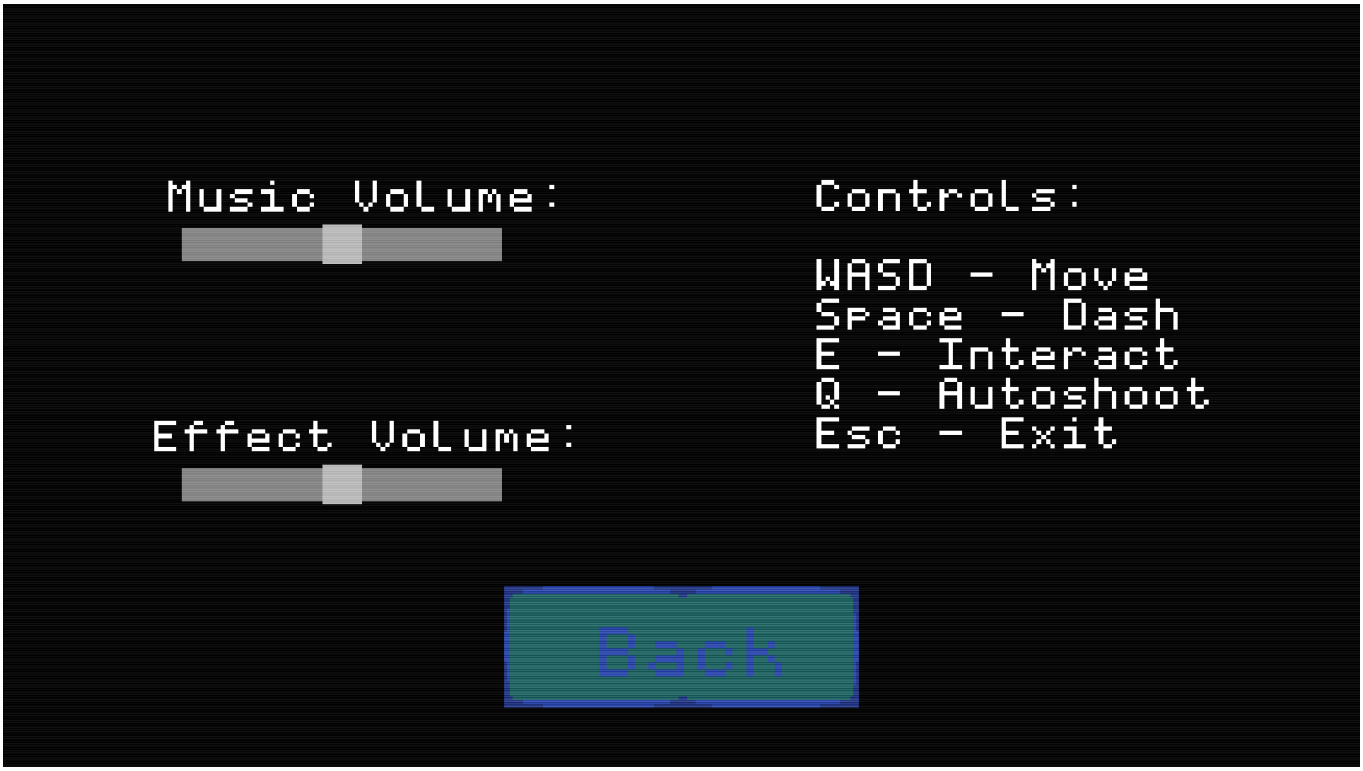
Die Umsetzung mit einem Start und Game Over Screen war auch von Beginn an festgelegt, wobei man sagen muss, dass die Idee zuvor noch ein wenig ausgebauter war als sie jetzt schlussendlich geworden ist. wir hatten überlegt Collectibels zu implementieren, die auch über mehrere Runden verteilt gespeichert werden und einem entweder Vorteile bringen oder nur dem Sammlelaspekt dienen.Auch ein hub war in Überlegung also ein Ort wo man zwischen den Runden zeit ferbringen kann, allerdings mussten wir beide Ideen verwerfen, da wir uns auf das wesentliche konzentriert haben und uns die Zeit fehlte um diese Ideen zu implementieren.



Startscreen

Game Over Bildschirm

Was wir uns aber vor allem von dem Projekt erhofft haben, war es viel ausprobieren zu können und viel zu experimentieren, was man denke ich mal an der riesigen Menge an verschiedenen Tests sieht.



Einstellungen

2 - Struktur

Main.java:
Die Main Methode ist der Einstiegspunkt in das Programm. Es erstellt ein neues Fenster und initialisiert es, worauf es die `Window.run` ausführt welches die Hauptschleife vom Spiel startet.

«-<—>-»

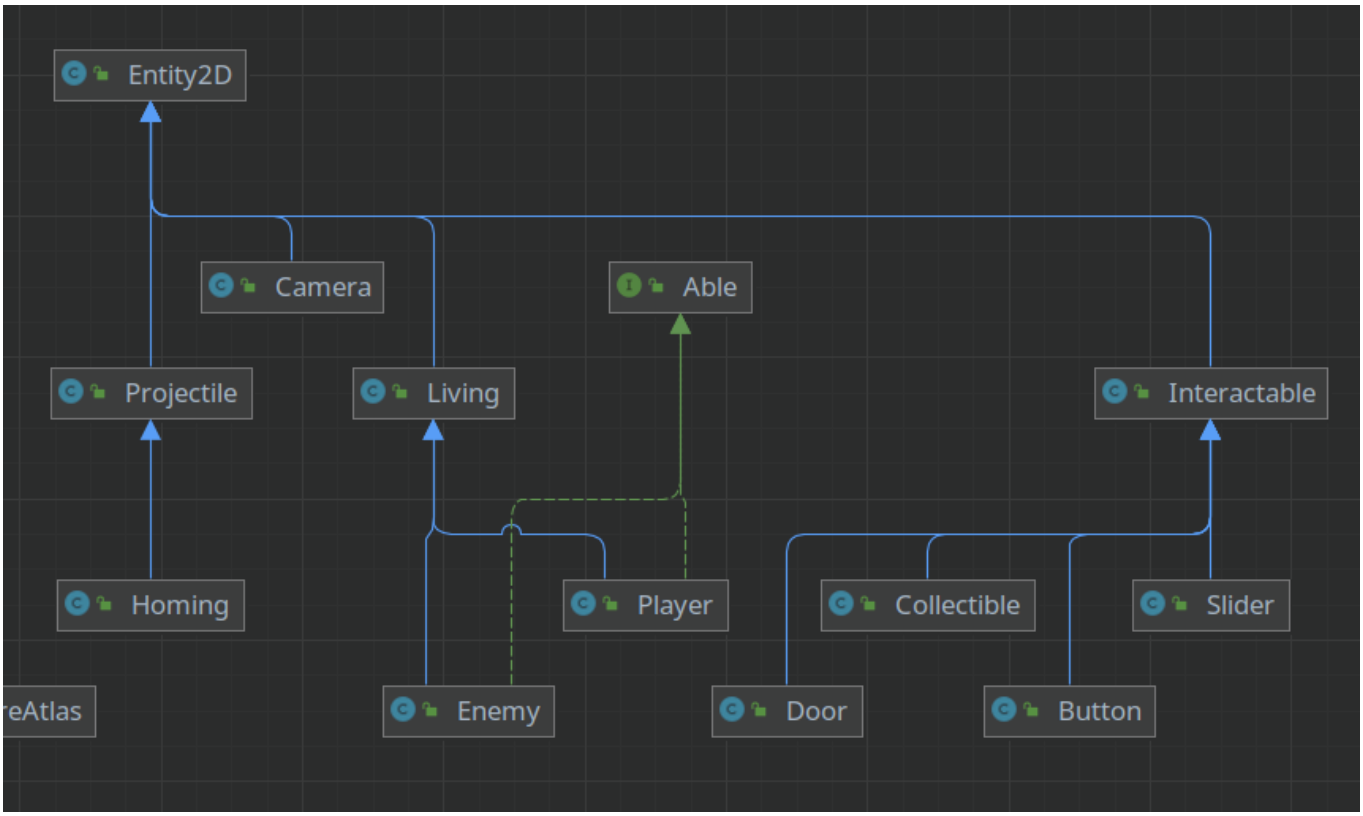
Audio Paket:
Dieses Paket enthält Klassen, die für die Audiofunktionen im Spiel verantwortlich sind. Es verwendet die OpenAL Bibliothek, um Audio im Spiel zu spielen.

«-<—>-»

Render Paket:
Zusammen mit dem Audio Packet und der Test-Klasse bildet das Render-Paket sämtlichen Hintergrundverwaltung von OpenGL, OpenAL und ihren Abstraktionen in handlichere Basiseinheiten wie `Entity2D`, `Interactable`, `Shader` oder `Texture`.
Dieses Paket alleine stellt also das Pendant zur gemeinsam entwickelten Spieleumgebung bzw. populäreren Game-Engines wie Godot oder Unreal.

Dieses Paket enthält vornehmlich Klassen, die ein funktionales und einfaches Rendern und Verwalten jeglicher Grafiken sichern. Dazu gehört die `Window` Klasse, die das Hauptfenster des Spiels erstellt und verwaltet, sowie die `Renderer` Klasse, die für das Zeichnen von Spielobjekten auf dem Bildschirm verantwortlich ist. Weitere Unterkategorien sind die `Camera`, die neben Kameraverwaltung auch dafür sorgt das der Spieler immer mittig bleibt sowie die `Interactabels`, Eine Ansammlung an Klassen. die `Entity2D`'s um Interagierbarkeit, wie des Klickens, Hoverns, Antippens und Ziehens erweitert.

Darüber hinaus befindet sich im Render-Package die Unterkategorie `Meshdata`, die Klassen, die zur Veränderung von allem visuell wahrnehmbaren da sind, verwaltet.
Eine der wichtigsten Klassen des package ist die `Entity2D` Klasse. Sie gibt durch vererbung an andere Klassen einen Grundbaustein, an dem sich eine sehr große Anzahl an Klassen orientieren, so ist jeder Gegner, der Spieler, jedes Projektil ein Objekt dessen Klasse von Entity2D abstammt.



Verdeutlichung der Vererbungstruktur rund um `Entity2D` herum

«-<—>-»

Game Paket:
Dieses Paket enthält aufbauend auf dem Render-/Audio-Paket spielspezifische Klassen, die die Spiellogik und den Spielzustand auf einer Ebene verwalten, die sich nicht für eine verallgemeinerte Implementation im Renderpaket geeignet haben (`Dungeon` , `Room` oder `Homing`).

Das Game Paket unterteilt sich wiederhohlt in zwei Kategorien: `Action` und die `Entities` .

Die Entities Kategorie verwaltet verschiedene Klassen die meistens von `Entitiy2D` erben, und sich über Vererbung oder Interfaces weiter in z.B. "lebend" und "fähig" unterteilen.

Die Action Kategorie verwaltet Klassen, die auf zurückgreifen auf die Entities leben in das Spiel hauchen. Beispielhaft dafür ist der Erscheinungsprozess von Gegnern über `Wave` und `EnemySpawner` oder `Ability` und `Upgrade` die Gegner wie Spielern modulbasiert Fähigkeiten zuschreiben, um etwa ein `Projectile` zu schießen oder mit der *Dash-Ability* aus der Abilitysammlung `Abilities` einen kleinen Teleport in Laufrichtung auszuführen.

«-<—>-»

Tests Paket:
Dieses Paket enthält verschiedene Testklassen, die vorrangig des Funktionstestens dienten, im späteren Verlauf aber immer mehr zu klasisch verwaltbaren Szenen wurden wie man sie aus

Godot oder Unity kennt. Sie sind also ladbare Bestandteile etwas Größerem die abgeleitet von `Test` die autarken Szenen eines Spiels, wie etwas einen Hauptbildschirm, ein Einstellungsmenü oder einen Game-Over-Screen, darstellen und organisieren.

Wesentliche Test / Szenen unseres Spiels zählen den AbilitySelectionScreen, TestGame, TestStartScreen und TestGameOverScreen. Die Restlichen Szenen waren meistens Experimente oder Tests um Mechaniken auszuprobieren, wie zum Beispiel ein Test um die Kollisionskontrolle zu überprüfen.

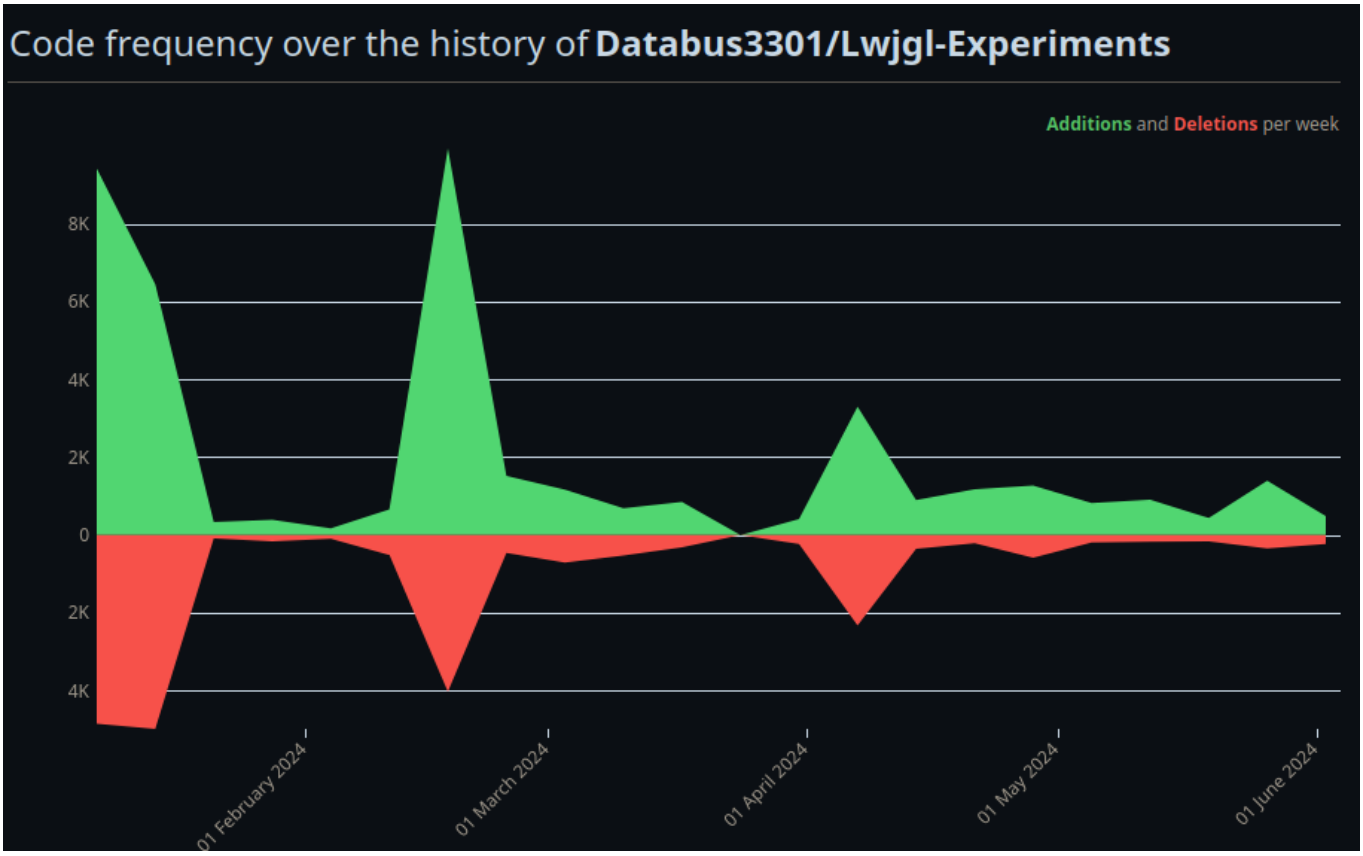
«-<—>-»

"Res"-Ordner:
Der *Res* Ordner enthält alle Ressourcen oder Assets die vom Spiel über Java-Quellcode hinaus benötigt werden. Darunter zählen `textures` für Spieler, Text und Räume, 3D / 2D `models` die aus Blender geladen werden können um Komplexere Formen abzubilden, `audio`-clips für Musik und Effekte, `shaders` für Programme auf Graphikkartenseite zum richtigen Interpretieren von Punktpositionen und RGB-Buffern, sowie die `settings` zum programmausführungsübergreifenden Einstellungsverwalten.

3 - Implementation und Techniken

Ausmaße:

Die abertausende Zeilen an Quellcode einzeln zu analysieren würde ein Ausmaß von mindestens genauso vielen Seiten verlangen.



Code-Frequency Analyse vonGitHub

Und da es zum Glück auch nur um Schlüsselstellen geht ist das auch nicht weiter schlimm. Es ist allerdings ganz hilfreich sich trotz fehlender Dokumentation von jeder Nische vor die Augen zu rufen um welches Ausmaß es sich hier handelt.

Was gegeben war, waren die 115 Zeilen minimalstem Startercodes aus der LWJGL Dokumentation. Darin enthalten:

- Betriebssystemunabhängiges erschaffen eines Fensters
- Weiterleiten von Tastatur- und Mauseingaben an Programmschnittstellen
- Und Aufsetzen des *OpenGL* Graphik-Standarts der es unter anderem ermöglicht Buffer an eine Graphikkarte zu Schicken und dieser von Grund auf zu beschreiben wie dieser

Interpretiert werden soll

Dazu kommt die Mathe-Bibliothek *JOML* die lineare Algebra auf Vektoren und Matrizen ermöglicht und *OpenAL* als Standart für die Schnittstelle zu den Lautsprechern.

ALLES Andere hinweg über `Shader`, `Texture`, `VertexBuffer`, `IndexBuffer`, `ObjModelParser`, `Font`, `TextureAtlas` und `Button`, `Slider` oder `Entity2D` und jeder anderen `src` Datei ist auf der oben beschrieben Basis unter Anwendung von Computer-Graphik Methodik und ganz viel Mathematik entstanden und sprengt damit schon jeglichen Rahmen.

«-<—>-»

Test:

```
/**
Serves as a template for all tests.
Tests are used to test the functionality of the engine.
 **/
 public class Test {

    /**
     * Called after Window/OpenGL/OpenAL initialization.
     * Called on the first frame of the test.
     */
     public void OnStart() {}

    /**
     * Called every frame.
     * @param dt Delta time.
     */
      public void OnUpdate(float dt) {}

    /**
     * Called every frame before OnUpdate.
     * */
     public void OnRender() {}
    /**
     * Called when the window is closed.
     * */
     public void OnClose() {}
    /**
     * Called when a key is pressed/released/held
     */
         public void OnKeyInput(long window, int key, int scancode, int
```

```
action, int mods) {}

    /**
     * Called when the window is resized.
     */
        public void OnResize(int width, int height) {
    }
```

Die Kommentare sind hier Selbsterklärend. `Test` abstrahiert die einzelnen relevanten Stellen aus `Window`.

```
…
private final ArrayList<BiConsumer<Float, Vector2f>> updateCallbacks;
…
public void OnUpdate(float dt) {
    for(int i = 0; i < updateCallbacks.size(); i++) {
        updateCallbacks.get(i).accept(dt, mousePos);
    }
}
…
public void addUpdateListener(BiConsumer<Float, Vector2f> callback) {
    updateCallbacks.add(callback);
}
```

Und bietet über `addUpdateListener` jeder anderen Klasse die Möglichkeit ohne explizites Ausschreiben in etwa `OnUpdate` eine anonyme Funktion, eine Callback oder einen Consumer, je nachdem wie man es nennen mag, jeden Frame aufzurufen.

Diese Funktionalität benutzten dann beispielsweise `Interactable`'s um automatisch ihren Status in Relation zu der sich ständig ändernden Cursorposition zu aktualisieren und die relevanten Callbacks aufzurufen.

```
public <T extends Test> Interactable(T scene) {
    super();
    init(scene);
}
public <T extends Test> void init(T scene) {
    scene.addKeyListener(this::onKeyInput);
    scene.addUpdateListener(this::onUpdate);
}

public void onUpdate(float dt, Vector2f mousePos) {
switch (state) {
    case DEFAULT -> defaultCallback.accept(this);
```

```
    case HOVER -> hoverCallback.accept(this);
    case PRESSED -> pressedCallback.accept(this);
    case RELEASED -> releasedCallback.accept(this);
    case DRAGGED -> draggedCallback.accept(this);
}

updateStates(mousePos);
…
}
```

`Interactable` Schmied

`Test` bildet die Grundlage für alle anderen Szenen die, die Funktionalität erben und erweitern.

«-<—>-»

TestGame:

Die dabei wohl relevanteste Szene bildet TestGame in der die Kernlogik des Spiels zusammenkommt. Die `OnUpdate` Methode fasst den Aufbau des Spiels gut zusammen. Hier wird:

- zunächst der Spieler
  - in Anpassung an die Performance des Endnutzer-PCs über `dt`

    (delta time, also die Zeit zwischen 2 Frames, die bei langsameren PCs höher sein wird als bei schnellen)

- mit seiner Geschwindigkeit `player.getSpeed()`
- in Richtung des Kraftvektors `player.getVelocity` bewegt.

```
// move player
player.translate(player.getVelocity().mul(player.getSpeed() * dt, new
Vector2f()));
```

- Dann die Kamera über lineare Interpolation auf den Spieler zentriert.

```
camera.centerOn(player);
```

- Der Spieler unter nutzen verschiedener Kollisionsalgorithmen (AABB, SAT, MBR) mit seiner Umwelt kollidiert.

```
// collide player and its fields
player.collide(enemies);
player.collide(room);
player.collide(props);
```

- Die `enemies` ArrayList nach Regeln des Gegnererscheinens in `EnemySpawner` bevölkert

```
// spawn enemies
spawner.update(dt, enemies);
```

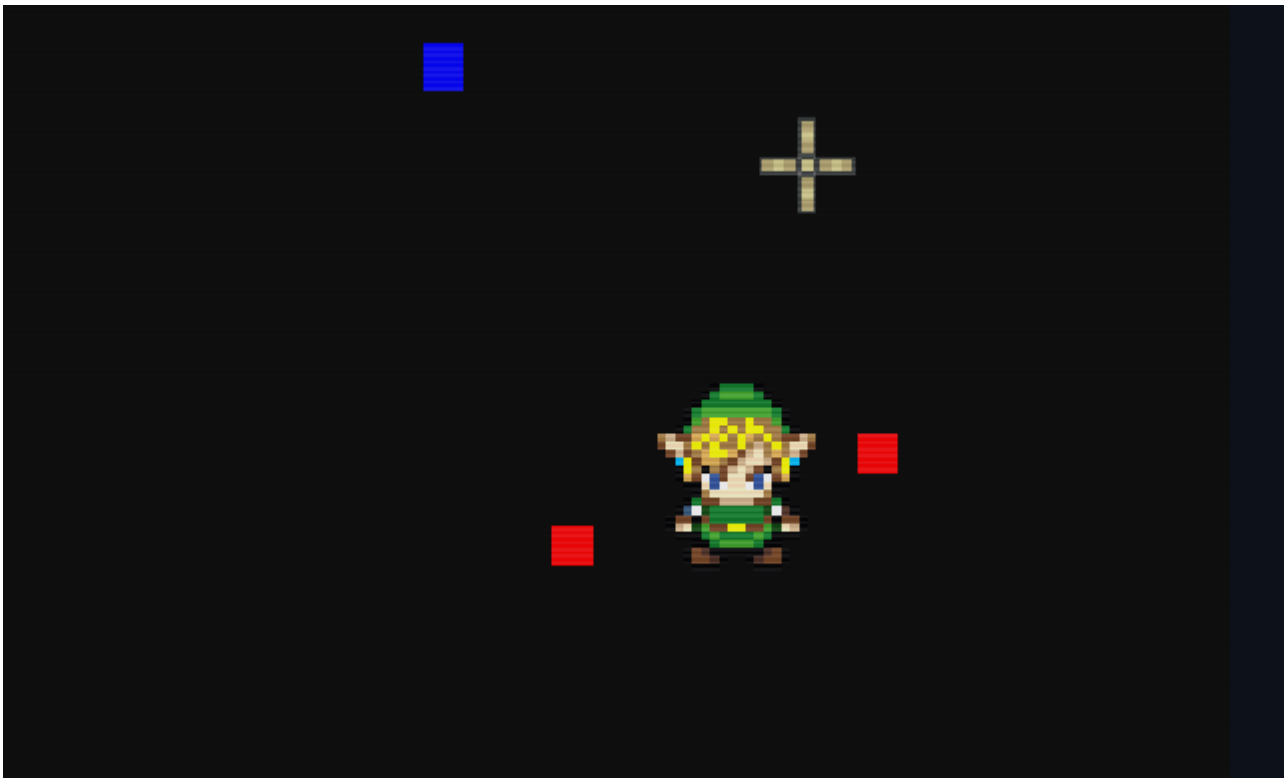
- Potentiell der Raum gewechselt sollte `room.update(…)` einen anderen als den aktuellen Raum zurückgeben

```
// change room
room = room.update(dt, spawner, player, enemies, projectiles, props);
```

- Und ein ähnlicher Prozess des Bewegens, Kollidierens und Entfernens für Gegner Durchlaufen

```
  Iterator<Enemy> enemyIterator = enemies.iterator();
  while (enemyIterator.hasNext()) {
  Enemy enemy = enemyIterator.next();
  // movement, abilities, iFrames, etc..
  enemy.update(dt, mousePos, player.getPosition(), room);
  // push away from each other
  enemy.collide(dt, enemies, player);
```

```
    // kill enemies
    if (enemy.getLP() <= 0) {
        enemy.spawnXp(this, props, player, room);
        enemyIterator.remove();
    }
    // print debug info if on cursor
    if (cursor.collideRect(enemy))
                renderer.drawText("LivePoints: " + enemy.getLP(),
                new Vector2f(enemy.getPosition().x - enemy.getScale().x
/ 2f,
                    enemy.getPosition().y + 15), 5);
    }
```

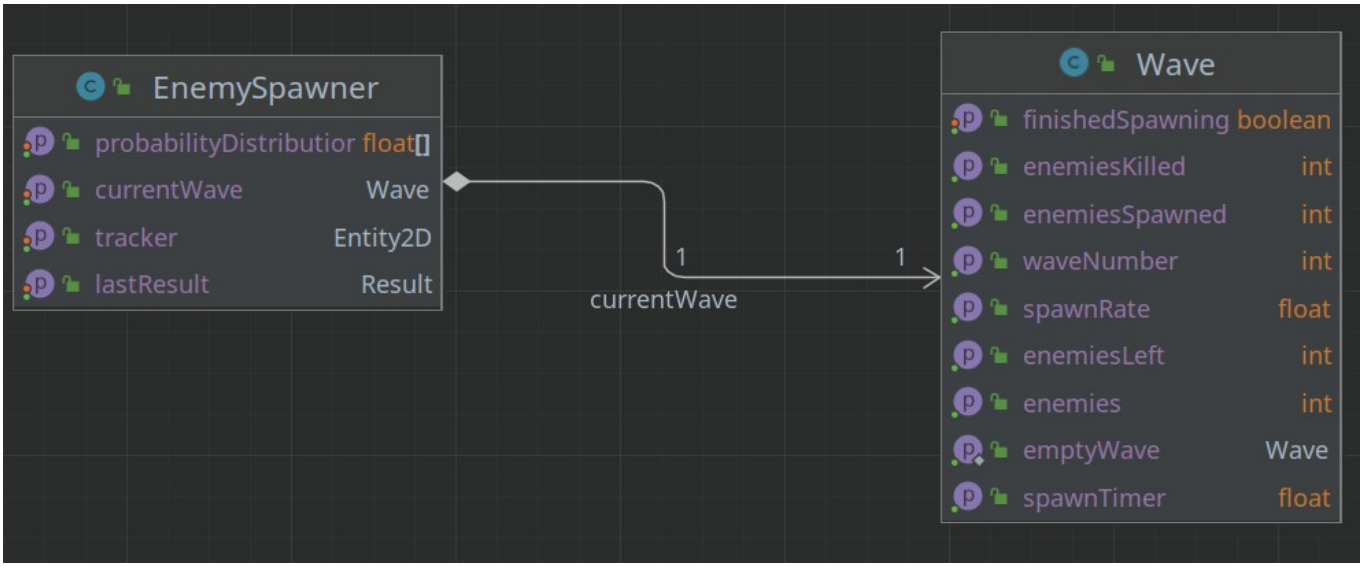


Gegnergruppe um einen Spieler gesammelt, *Kollidiert, Bewegt, Zentriert*

«-<—>-»

EnemySpawner:
Erschaffen werden die Gegner immer dann wenn genug Zeit verstrichen ist um eine Diskrepanz zwischen erschaffenen und zu erschaffen gehabt zu habenen Gegnern herscht.

Die Frequenz und Ereigniszahl werden dabei durch die `currentWave` beeinflusst.



Zusammenhang `EnemySpawner` // `Wave`

Je nach eingetretenem Fall wird eine andere `Result` enum zurückgemeldet, um beispielsweise das ende einer Welle zur Anzeige von neuen Fähigkeiten zu nutzen.

```
public Result update(float dt, ArrayList<Enemy> enemyCollection) {
    int enemiesLeft = currentWave.getEnemiesLeft();
    currentWave.update(dt);

    if (currentWave.getEnemiesLeft() < enemiesLeft) {
        for(int i = 0; i < enemiesLeft - currentWave.getEnemiesLeft(); i++)
{
            enemyCollection.add(spawn());
        }
        return lastResult = Result.SPAWNED;
    }

    if (currentWave.isFinishedSpawning()) {
        if(enemyCollection.isEmpty()) // if there are no enemies left alive
            return lastResult = Result.WAVE_OVER;
        return lastResult = Result.FINISHED_SPAWNING;
    }

    return lastResult = Result.NOTHING;
}
```

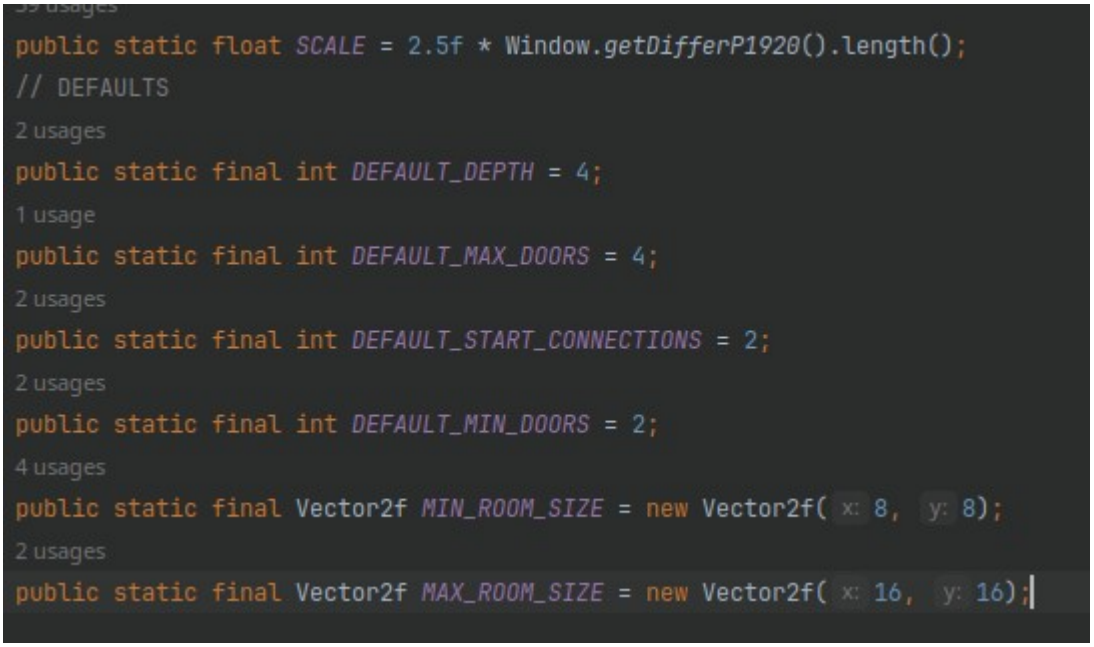
Erschaffen wird nach einer *Cumulative Distribution Function* die sich die Addierbarkeit von Wahrscheinlichkeiten zu nutze macht. Um Basierend auf der Wahrscheinlichkeitsverteilung für unterschiedliche Gegnertypen eine Auswahl zu treffen

```
public Enemy spawn() {
        float rand = (float) Math.random();
        float sum = 0;
        int index = 0;
        for (int i = 0; i < probabilityDistribution.length; i++) {
                sum += probabilityDistribution[i];
                if (rand < sum) {
                        index = i;
                        break;
                }
        }
…
}
```

«-<—>-»

Dungeon:
Die Dungeon Klasse verwaltet die Strukturen denen der Spieler oder die Spielerin auf Gegner treffen kann.

Unter einfach zu justierenden Standarts:



generiert die`generate` Methode mit hilfe von *Rekursion* einen zufälligen Dungeon.

```
private Room[] generate(int depth, int maxDoors, int connections) {
    Vector2f dim = new Vector2f(
    (int) (Math.random() * (MAX_ROOM_SIZE.x - MIN_ROOM_SIZE.x)
    + MIN_ROOM_SIZE.x),
    + (int) (Math.random() * (MAX_ROOM_SIZE.y - MIN_ROOM_SIZE.y)
```

```
    + MIN_ROOM_SIZE.y)
    );

    if(depth <= 0) {
        rc++;
        return new Room[]{
        new Room(
         player, RoomType.BOSS,  "Boss", 0,
         RoomDesign.values()[floor % RoomDesign.values().length],
         this, dim.add(2, 2), floor)
        };
    }

    int newDoors = (int) (Math.random() * (maxDoors-(DEFAULT_MIN_DOORS-1))
    + DEFAULT_MIN_DOORS);

    if(depth == 1)
        newDoors = 1;

    Room[] rooms = new Room[connections];
    for (int j = 0; j < connections; j++) {
        // generate random design
        RoomDesign design = RoomDesign.values()[
                floor % RoomDesign.values().length
        ];
        // generate random type
        RoomType type = rndmRoomType();
        //RoomType type = RoomType.SMITH;
        String name = "Room " + (max_depth-depth + 1) + "-" + floor;

        rooms[j] = new Room(player, type, name, newDoors,
                                                design, this, dim, floor);
        rooms[j].setDepth(depth);
        rc++;
    }

    for (Room room : rooms) {
        room.setConnectedRooms(generate(depth - 1, maxDoors, newDoors));
    }

    return rooms;
}
```

Sie generiert ein Array von Room Objekten, die ein Dungeon repräsentieren. Die Tiefe der Rekursion wird durch den `depth` Parameter bestimmt. Die Methode beginnt mit der Erstellung eines zufälligen `Vector2f Objekts`, das die Dimensionen des Raums repräsentiert. Wenn die

Tiefe 0 oder weniger ist, wird ein Boss-Raum erstellt und zurückgegeben. Ansonsten wird die Anzahl der Türen für den nächsten Raum zufällig bestimmt und ein Array von Room Objekten erstellt. Für jede Verbindung wird ein neuer Raum mit zufälligem Design und Typ erstellt. Schließlich wird für jeden erstellten Raum die generate Methode erneut aufgerufen, um die verbundenen Räume zu erstellen, wobei die Tiefe um 1 reduziert wird. Die Methode gibt schließlich das Array von erstellten Räumen zurück.

«-<—>-»

Room:
Und diese Räume verwalten wiederum ihre Wände, Türen, Verbindungen und auch Titel.

- so wird nicht nur auf jedem update überprüft ob der Spieler mit einer offenen Tür kollidiert und demnach der Raum gewechselt werden soll

```
        Room room = this;
        for(int i = 0; i< doors.length; i++) {
            // short circuit if doors are closed to better performance
            if(!doors[i].isOpen()) continue;
            // if player entered door → switch room
            if(doors[i].collideRect(player.getCollider())) {
                // switch room
                room = doors[i].getConnectedRoom();

player.setPosition(doors[i].getConnectedRoom().getPosition());
                …
                }
        }
        …
``` ``````
```

- sondern auch ob und mit welcher Transparenz in Abhängigkeit von der Zeit seit dem in den Raum gewechselt wurde, der Titel angezeigt werden soll.
  Wobei `ColorReplacement` dabei eine Abstraktion zu der 4×4 Matrix die, die GPU erhält, um im `Shader` die durch die erste Reihe definierte Farbe durch die in der zweiten Reihe und die in der Dritten mit der der Vierten zu ersetzen. Was wie hier zum beispiel für das selektive Abblenden von Farben genutzt werden kann

```
timeSinceLoad += dt;
if(timeSinceLoad < 2 && type != Dungeon.RoomType.START) {
    ColorReplacement cr = new ColorReplacement();
```
```

```
    Vector4f t_color = new Vector4f(1, 1, 1, 1 * (1 - timeSinceLoad/2));
    if (Objects.equals(title, "Boss")) t_color.set(1, 0, 0, t_color.w);
    cr.swap(new Vector4f(1, 1, 1, 1), t_color);

    renderer.drawText(title, new Vector2f(position.x, position.y + 64),
45,
            Font.RETRO_TRANSPARENT_WHITE, Shader.TEXTURING_CRA,
            Font::centerFirstLine_UI, cr, null
         );
}
```

«-<—>-»

Renderer:

Die Zentrale Renderinginstanz der Engine die die vereinfachenden Abstraktionen zurück in für den Datentransfer zur GPU geeignete Packete verwandelt.

- Dazu Zählt das Übertragen Variabeln, wie diese Transformationsmatrizen

  ```
          shader.setUniformMat4f("uModel", modelMatrix);
          shader.setUniformMat4f("uView", camera.calcViewMatrix());
          shader.setUniformMat4f("uProj", camera.getProjectionMatrix());
  ```

  oder Zeit und Auflösung für dynamische Shadereffekte wie das Wabern am Anfang des Spiels

  ```
          if (shader.hasUniform("uResolution"))
              shader.setUniform2f("uResolution", Window.dim.x,
Window.dim.y);
          if (shader.hasUniform("uTime"))
              shader.setUniform1f(
                      "uTime",
                      ((System.currentTimeMillis()) % 100000) / 1000f
              );
  ```

- aber auch das interpretieren und übergeben sämtlicher Entitätsdaten in `draw` Methoden wie dieser

  ```
        public <T extends Entity2D> void draw(T entity) {
              assert entity != null :
              "[ERROR] (Render.Renderer.DrawEntity2D) Entity2D is null";
  ```

```
            if (entity.isHidden()) return;

            chooseShader(entity);
            SetUniforms(currentShader, entity);

            // choose Texture
            if (entity.getTexture() != null)
                entity.getTexture().bind();
            if (entity.getAnimation() != null) {

entity.getAnimation().getAtlas().getTexture().bind();
                        entity.getModel().replaceTextureCoords(
                        entity.getAnimation().getTexCoords()
                        );
            }

            // choose Model
            ObjModel model = entity.getModel();
            assert model != null :
                "[ERROR] (Render.Renderer.DrawEntity2D) Entity2D has no
model";

            // choose VertexArray and IndexBuffer
            VertexArray va = new VertexArray();
            va.addBuffer(model.getVertexBuffer(), Vertex.getLayout());
            IndexBuffer ib = model.getIndexBuffer();

            va.bind();
            ib.bind();

            glDrawElements(GL_TRIANGLES, ib.getCount(), GL_UNSIGNED_INT,
0);

            //draw abilities
            if (entity instanceof Able able) {
                if (able.getAbilities() == null) return;
                for (Ability ability : able.getAbilities()) {
                    for (Projectile projectile :
ability.getProjectiles()) {
                        draw(projectile);
                    }
                }
            }
        }
```

- Dabei ist jeder Aufruf von `glDrawElements` oder `SetUniforms` mit langen Übertragungsraten zwischen CPU und GPU verbunden und zieht deswegen viel Leistung. Um die Aufrufe zu diesen Methoden zu vermindern gibt es Computer-Graphik Methoden wie das Batch-Rendering bei dem alle Daten in einem `glDrawElements` Aufruf übergeben werden. Dafür müssen diese erst speziell zusammengefügt werden und die GPU muss über diese Zusammensetzung informiert werden. Eine solche Implementation findet sich in der `drawText` Methode um nicht für jeden Buchstaben einen einzelnen Methodenaufruf zu generieren.

```
    // Create combined vertex and index buffers
        VertexBuffer vb = new VertexBuffer(totalVertices);
        IndexBuffer ib = new IndexBuffer(totalIndices);

        long vertexOffset = 0;
        long indexOffset = 0;

        // Append data from each entity's buffers to the combined
buffers
        short[][][] faces = model.getFaces();
        float[][] positions = model.getPositions();
        int xOffset = 0;

        for (int i = 0; i < texCoordArr.length; i++) {
            float[] data = new float[model.getVertexCount() *
Vertex.SIZE];
            int[] indices = new int[model.getIndexCount()];

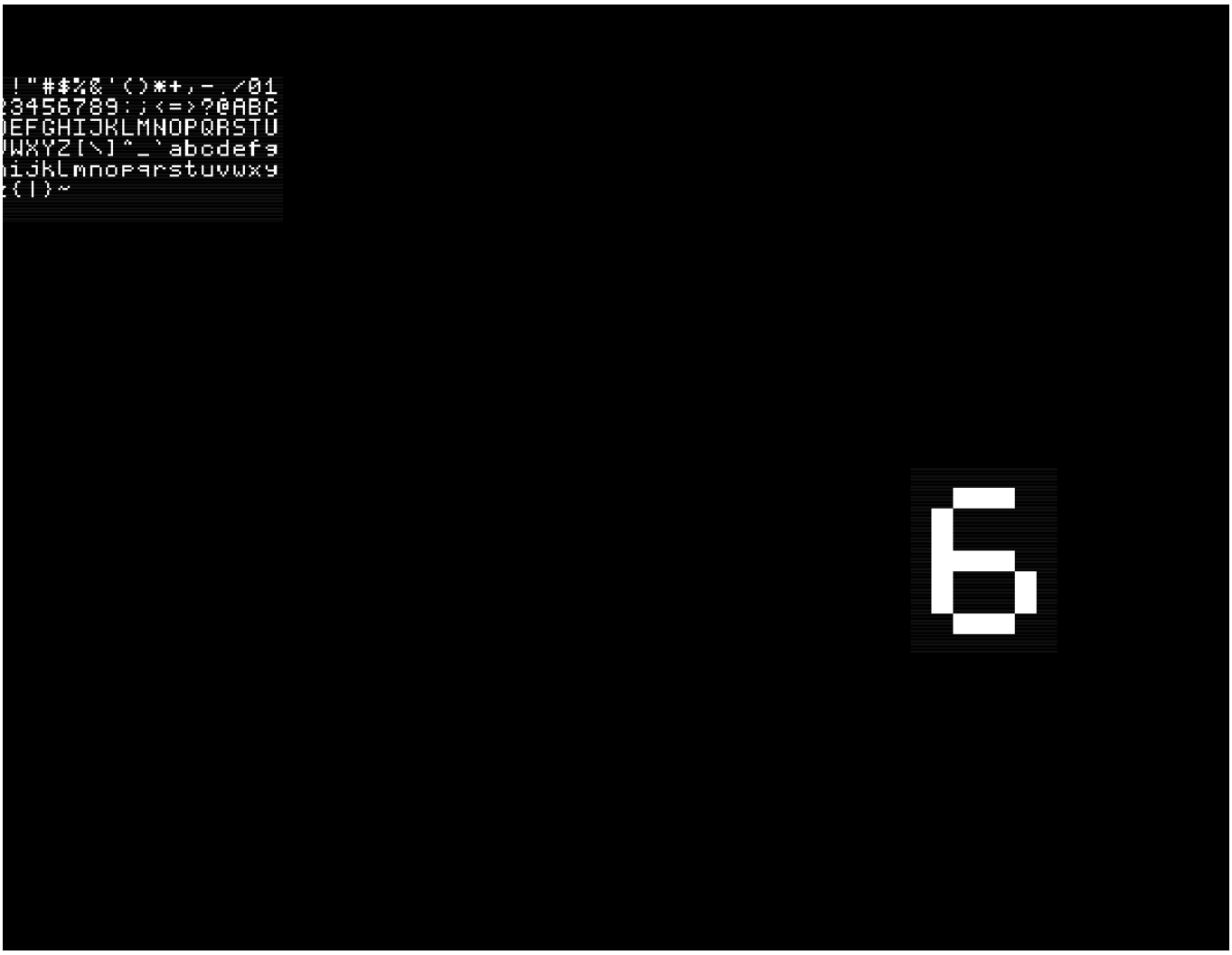
                …

            short dataIndex = 0;
            for (short[][] face : faces) {
                for (short k = 0; k < face.length; k++) {
 // die Positionen werden aus der geladenen OBJ Datei
 // gesucht und Skaliert/Verschoben da nicht jeder Buchstabe
 // auf der GPU verschoben werden kann wenn wir entlang des
 // Batch Buffers nur ein Verschiebepaket schicken
 // (Aufrufe Sparen)
                    float[] position = positions[face[k][0] - 1];
                    data[dataIndex++] = position[0]
                    * scale.x + pos.x + scale.x * 2 * i;
                    data[dataIndex++] = position[1]
                    * scale.y + pos.y + -scale.y / characterAspect * 2;
                    data[dataIndex++] = position[2];
// Da alle Buchstaben in einem Bild sind müssen
// hier auch die Texturkoordinaten für jeden
```

```
// Buchstaben mitverpackt werden
                    if (Vertex.SIZE > 3) {
                        if (face[k].length > 1) {
                            float[] texture = texCoordArr[i][face[k][1]
- 1];
                            data[dataIndex++] = texture[0];
                            data[dataIndex++] = texture[1];
                        } else {
                            dataIndex += 2;
                        }
                    }
                    indices[dataIndex / Vertex.SIZE - 1] =
                    (short) (dataIndex / Vertex.SIZE - 1)
                    + (int) indexOffset / 4;
                }
            }
            xOffset++;

            vb.update(data, vertexOffset);
            ib.update(indices, indexOffset);

            vertexOffset += data.length * 4L;
            indexOffset += indices.length * 4L;
        }
```

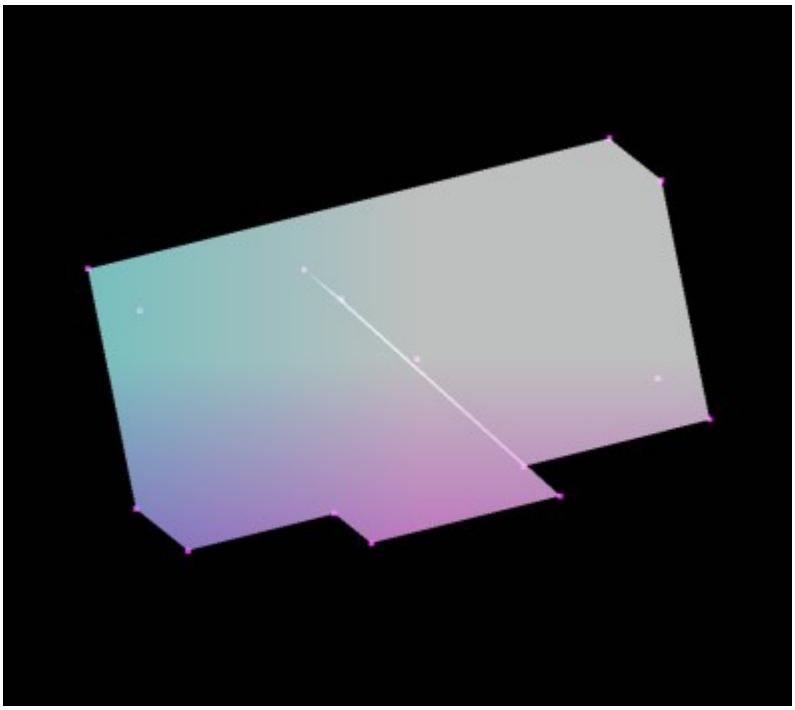


Texturenatlas oben links, Texturabschnitt rechts → Modifizierte Texturkoordinaten → TestTextureAtlas // "ta"

«-<—>-»

ObjModel:

Die Engine benutzt, weil ich einen Interpretierer für 3D Dateien schreiben wollte, intern das leicht einzulesende *Wavefront OBJ* Format.



Aus Blender exportierte .obj Datei drehend im Test3Dspin / "3d"

Da das Format als ASCII-Text vorliegt ist es einzulesen recht trivial:

```
String line;
while ((line = reader.readLine()) != null) {
    // splits line on whitespace
    String[] parts = line.split("\\s+");
    switch (parts[0]) {
        case "v":
            model._positions.add(parseFloatArray(parts));
                break;
                case "vn":
            model._normals.add(parseFloatArray(parts));
                break;
                case "vt":
            model._textures.add(parseFloatArray(parts));
                break;
                case "f":
            model._faces.add(parseFace(parts));
            model._materialIDs.add(currentMaterialID);
                break;
                case "mtllib":
                        model._materials.addAll(parseMTL(parts[1]));
                break;
                case "usemtl":
                    …
                break;
```

```
            }
        }
```

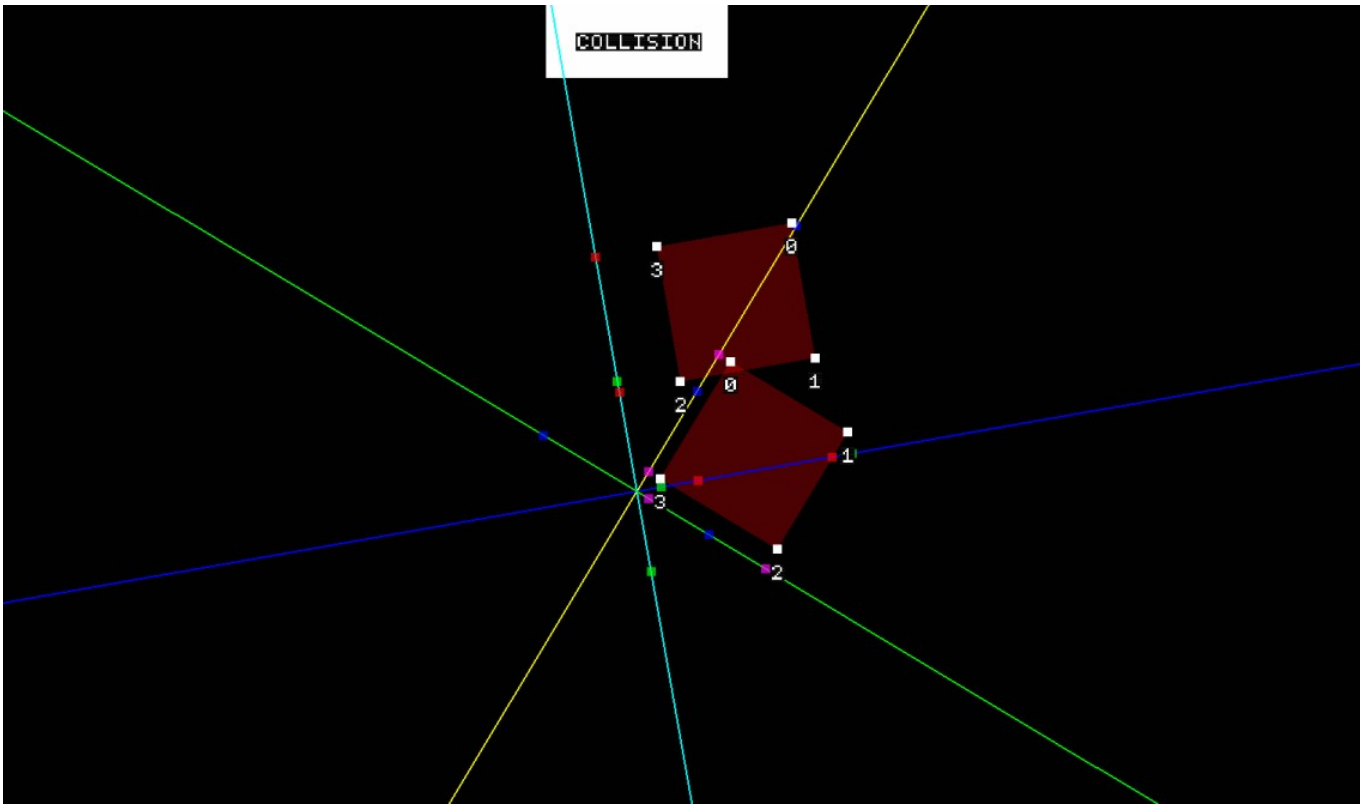
Bemerkenswert ist eventuell das nach dem Einleseprozess über ArrayLists, die das stetige einlesen von Dateien unbekannter Länge erlauben, alle internen Operationen auf Arrays weiterlaufen da diese deutlich kürzere Zugriffszeiten und Arbeitsspeicherfußabdrücke haben.

```
        public void castToArrays() {
                    positions = toArray(_positions, float[].class);
                    normals = toArray(_normals, float[].class);
                    textures = toArray(_textures, float[].class);
                    faces = toArray(_faces, short[][].class);
                    materials = toArray(_materials, ObjMaterial.class);
                    short[][] IDs = toArray(_materialIDs, short[].class);
                    materialIDs = new short[IDs.length];
                    for(int i = 0; i < IDs.length; i++) {
                        materialIDs[i] = IDs[i][0];
                    }

                    calcIndexCount();
                    calcVertexCount();

                    _positions.clear();
                    _normals.clear();
                    _textures.clear();
                    _faces.clear();
                    _materials.clear();
                    _materialIDs.clear();
        }
        private static <T> T[] toArray(ArrayList<T> list, Class<T> c) {
            @SuppressWarnings("unchecked")
            T[] array = (T[]) Array.newInstance(c, list.size());
            list.toArray(array);
            return array;
        }
```

SAT:
Das *Separating Axis Theorem* ist der Grundstein der meisten Kollisionsalgorithmen unserer Zeit. Graphikkarten werden speziell zur Berechnung seiner Rechenschritte optimiert, große Game engines nutzen ihn, Robotik wäre ohne ihn nicht denkbar.



SAT visualsiert im TestSAT // "sat

Der Algorithmus über den ich ein Mathe-Referat gehalten hab, basiert auf Projektion, dem Skalarprodukt, Normalenvektoren und auf der einfachen Grundlage das sollten sich 2 konvexe Formen nicht überlappen es möglich sein muss eine Linie zwischen ihnen zu ziehen die beide nicht berührt und das diese Linie wenn es sie gibt orthogonal zu einer der Kanten aller beteilitgten Formen ist.

Er lässt sich auf beliebig viele Dimensionen erweitern und führte zwischenzeitig unsere Kollisionserkennung an. Über Optimierungen wie das zusammenfassen eines Raumes als ein Kollisionsrechteck statt Kollisionen mit jeder Wand zu überprüfen ist er allerdings größtenteils überfällig geworden.

4-Reflexion

---

3 - Reflexion

Habe wir unsere Ziele erreicht?
Mit dem obersten Ziel der Selbstweiterbildung abseits von Noten kann man diese Frage nur bejahen. Abgeglichen mit dem Modulhandbuch Informatik des KIT's entspricht der Umfang und die Methodik bis auf Bildsyntheseverfahren dem behandeltem Stoff des Wahlmodules Computergraphik

Mein Persöhnliches Highlight des 3D Modelle Einlesens funktioniert nach wie vor Astrein (auch wenn es nie zur Anwendung gefunden hat) und wird außerhalb des Projektrahmens hoffentlich noch ausgebaut sollte ich Zeit finden.

Was hat sich als schwierig/unlösbar erwiesen?
Probleme in kleinere Aufzuteilen und diese nach und nach abzuhaken hat auch hier super funktioniert. Jedoch brauchte es gerade für den grundlegenden OpenGL kram einen Überblick der sich schwer zu erarbeiten war (viel neue lineare Algebra, wenig einsteiger Anleitungen (mit Java bezug)), wobei sich ein Verständnis von C-Pointern definitiv bewährt hat.

Unlösbar bis zum Schluss war das *Instatiating* eine Computer-Graphik-Methode die ähnlich wie Batch-Rendering an Draw Calls spart aber nur vielfache vom gleichen Objekt mahlen kann. Nützlich für etwa Partikel Systeme.

Welchen Erkenntniszuwachs hat mir das Projekt gebracht?
Wenn nicht die bereits erwähnten Kenntnisse aus dem Wahlmodul, dann die Dimensionen in denen Software Projekte wie Unity, Godot und Unreal arbeiten. Die Vielfalt an simulatenen Baustellen die praktisch sauberen Code erzwingt und Versionsmanagement im Arbeiten von (zweier) Teams über GitHub

Was würde ich bei einem ähnlichen Projekt anders machen?
Spezifischer für mein Ziel entwickeln statt alle Wege zu erkunden. Auch wenn mir letzteres auch viel Spaß bereitet ist der Zeitliche Rahmen irgendwo dann doch der limitierende Faktor.

Wie zufrieden bin ich mit dem Endergebnis?
Alles in allem sind wir sehr zufrieden mit dem Ergebnis. Es ist am Ende doch noch alles zusammengekommen und der Weg dahin war ein langer und lehrreicher.