Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

Studienprojekt I

Schatzinsel

Entwickeln eines Spiels durch objektorientiertes Programmieren

---

fertiggestellt am TBD

•

Fachbereich Duales Studium Wirtschaft / Technik
Hochschule für Wirtschaft und Recht Berlin

Name:	Jahn, Marko, 612678
Name:	Schenkewitz, Mario, 685593
Name:	Zilius, Sven,
Fachrichtung:	Duales Studium Wirtschaft • Technik
Studiengang:	Informatik
Studienjahrgang:	2020
Betreuer HS:	Zimmermann, Arthur
Anzahl der Wörter:	TDB

Abstract

Development of games isn't for entertainment Purposes only nowadays. It is possible do model different environments and worlds through them and to use those for simulations or predictions. In light of this it is evaluated how well suited older and modern development Environments are in realizing these games. The scenario of a Treasure Island serves as an example. Pirates on this island move in random directions and let each other know when the treasure is located. In this Paper Java, Unreal Engine and Unity Engine are considered for the making of this game.

Kurzfassung

Die Entwicklung von Spielen ist Heutzutage nicht nur für Unterhaltungszwecke interessant. Es ist möglich durch sie auch verschiedene Umgebungen und Welten modellieren und für Simulationen oder vorhersagen verwenden. Vor diesem Hintergrund wird untersucht, wie ältere und moderne Entwicklungsumgebungen sich eignen in der Umsetzung von Spielen. Als Beispiel dient das Szenario einer Schatzinsel, auf der Piraten einen Schatz suchen. Sie Bewegen sich in zufällige Richtungen und lassen sich gegenseitig wissen, wenn der Schatz gefunden wurde. Betrachtet werden in dieser Arbeit die Umsetzung in Java, Unreal Engine und Unity Engine.

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar

Graphical User Interface Graphische Benutzer Oberfläche, beschreibt die graphische Darstellung eines Programms auf einem Bildschirm die für Benutzereingaben und für die Ausgabe vom Programm genutzt werden kann. Sie stellt allgemein Informationen und Möglichkeiten für die Interaktion für den Benutzer dar.

NavMesh Ein NavMesh ist ein Objekt, das aus mehreren zweidimensionalen Polygonen besteht. Diese beschreiben alle Flächen, auf denen ein Charakter stehen kann und nicht durch andere Objekte blockiert werden. Sie werden dafür verwendet, Künstliche Intelligenzen durch komplexe Räume zu lenken.

Akronyme

OOP Objektorientierte Programmierung

OOPL Objektorientierte Programmiersprachen

1 Einleitung

Die Objektorientierte Programmierung (OOP) ist ein heutzutage weit verbreitetes Computerprogrammiermodell oder Programmierparadigma. In der OOP werden Anwendungen um so genannte Objekte und deren Daten herum strukturiert und nicht um Funktionen oder Logik. So fokussiert sich die OOP auf Objekte mit denen eine Anwendung interagiert, diese Objekte können als Datenfeld mit Attributen und eigenem Verhalten angesehen werden.

Einen tieferen Einblick in das Thema der OOP soll der Abschnitt 2 bieten, in dem die Basis für die OOP näher erörtert wird.

Thema dieses Studienprojektes ist es sich dieses Programmierparadigma zunutze zu machen, um ein Spiel zu entwickeln.
Der Titel des Spiels ist „Schatzinsel“, es handelt von drei Piraten die sich auf einer Insel befinden, auf der ebenfalls ein Schatz versteckt ist. Es ist ihnen unbekannt wo sich dieser Schatz befinden. Durch zufällige Bewegung über die Insel wollen sie versuchen diesen Schatz zu finden. Jeder der Piraten hat verschiedene Eigenschaften, wie zum Beispiel Geschwindigkeit.
Sobald einer der Piraten den Schatz entdeckt, ruft er die anderen zu sich. Am Ende versammeln sie sich alle um den Schatz.

Dieser Umstand soll durch das Spiel modelliert und gegebenenfalls erweitert werden.

2 Grundlagen

2.1 Objektorientierte Programmierung

Objektorientierte Programmiersprachen (OOPL) eignen sich durch die Konzentration auf die Objekte mit denen das Programm interagieren soll dazu, dass Software Wartbar und lesbar bleibt. Objekte und ihre Methoden können getrennt voneinander erstellt und programmiert werden. Diese Unterteilung in Einzelteile ermöglicht des Weiteren die Wiederverwendbarkeit von Code, also dem Quelltext, entweder in dem die Klasse selbst wiederverwendet wird, oder eine andere Klasse von ihr erbt. Dazu mehr im Abschnitt 2.3

OOP ist ein Programmierstil, der mittlerweile in Unternehmen, Industrie und akademischen Kreisen seinen Platz gefunden hat. Es wird ein natürlicheres und dadurch ein mächtigeres Entwurfsparadigma insofern erwartet, dass ein Programmierer produktiver ist, wenn das Anwendungsmodell näher am Problem modelliert werden kann, dass gelöst wird.
Anders gesagt ist eine OOPL mächtiger, weil es einem Programmierer erlaubt ein Problem zu lösen, dass die Struktur einer Welt nachbildet in welcher das Problem entstand. Vertrautheit mit solch einer Sprache eröffnet Wege zu einer klaren und natürlichen Lösung [Yae14].

Es werden also im betrachteten Fall der Schatzinsel die Objekte, die aus der echten Welt bekannt sind, so modelliert, dass die Interaktionen und Verhaltensweisen natürlich in einer Anwendung modelliert werden können.

2.2 Datenabstraktion und Verkapselung

Einer der ersten Schritte bei der OOP ist es alle Objekte zu sammeln, die manipuliert werden müssen. Im Fall des Spiels „Schatzinsel“ gibt es zum Beispiel Piraten, eine Insel, ein Schatz, aber auch eine graphische Benutzeroberfläche die alle als Objekt betrachtet werden können. Sie müssen untereinander aber auch mit dem Benutzer interagieren können.

Einer der großen Durchbrüche bei der Gestaltung von Programmiersprachen geschah, als es ermöglicht wurde, dass Programmierer eigene Datentypen definieren

konnten. C.A.R. Hoare's record class in Simula67 gilt als die Quelle der Idee. [Yae14] Verbreitung des, durch Programmierer definierten, Datentyps fand durch die Sprache Pascal statt. [Wir71]

So kann in Pascal ein Verbund definiert werden, der selbst aus Daten besteht, siehe Listing 2.1.

```
1 TYPE Auto = RECORD
2 KFZ: String[12];
3 Gewicht: Real;
4 AnzPassagiere: Integer;
5 END;
```

Listing 2.1: Verbund Beispiel Definition [Zim20]

Dieses Beispiel (Listing 2.2) definiert einen neuen Datentyp *Auto*, in dem jeweils die Art des KFZ, das Gewischt und die Anzahl der Passagiere angegeben wird. Nun kann man den Typen *Auto* nutzen, um beliebig viele Daten-Entitäten zu erstellen.

```
1 VAR a : Auto;
2 BEGIN
3 a.KFZ := 'B XY 123456';
4 a.Gewicht := 555.77;
5 a.AnzPassagiere := 5;
6 END;
```

Listing 2.2: Verbund Beispiel Instantiierung [Zim20]

Manipulationen der Daten innerhalb des Verbunds werden extern vom Datentyp selbst betrachtet, die Syntax der Sprache indiziert somit nur eine Gruppierung von Daten.

Verbunde in Pascal werden homogen genannt, wenn alle Daten innerhalb des Verbunds denselben Typ haben. Sie sind heterogen, wenn es verschiedene Typen, wie im Beispiel, gibt.

Pascal erlaubt zwar die Erstellung von heterogenen Verbunden, jedoch ist es nicht möglich in diesen Verbunden Operationen einzubauen. Anders als bei Pascal ist die bei Java zum Beispiel möglich.

```
1 class Point {
2 private int x, y;
```

```
3           public int getX() { return x; }
4           public int getY() { return y; }
5           public void setXY(int x, int y) {
6                   if (x >= 0 && y >= 0) {
7                           this.x = x;
8                           this.y = y;
9                   }
10          }
11  }
```

Listing 2.3: Operation innerhalb einer Klasse [Yae14]

Im Listing 2.3 wird zum Beispiel durch die Funktion „setXY()“ ermöglicht, dass x und y gesetzt werden können. Es ist zu bemerken, dass verschiedene Sichtbarkeiten gesetzt werden. Dadurch, dass x und y auf *private* gesetzt sind, können sie also von außen nur durch die Funktion „setXY()“ geändert werden. Dies veranschaulicht auch das Konzept der Verkapselung [Yae14] Seite 4.

Datenabstraktion in ihrer vollen Kapazität beschreibt also die Möglichkeit für Programmierer ihre eigenen Datentypen zu definieren, diese Typen können eine Heterogene Struktur haben und die Sicht auf diese kann eingeschränkt werden. Außerdem ist es möglich Operationen innerhalb dieser Datentypen zu haben.

In diesem Zusammenhang kann gesagt werden, dass Objekte also aus Attributen und Operationen, auch Funktionen oder Methoden genannt, bestehen und Instanzen von diesen Datentypen sind. Diese Datentypen werden im Allgemeinen auch als Klassen bezeichnet.

2.3 Vererbung

Die Beziehung zwischen Typen in spielt eine wichtige Rolle in den OOPL, so ist die Vergleichbarkeit von Typen wichtig, um Objekte zu vergleichen. Diese Beziehungen werden durch Vererbung erweitert und bereichert. Es stellt sich ein System aus Vererbungs- und Subtypenbeziehungen dar, diese Tragen zur Komplexität einer Sprache bei. Dabei spielen verschiedene Begriffe zentrale Rollen. Zum Beispiel wird gesagt, dass Klasse B von Klasse A *erbt* oder *abgeleitet* ist, wenn Objekte der Klasse B mindestens dieselben Attribute und Verhaltensweisen wie Objekte der Klasse A haben. Die Objekte der Klasse B können auch mit eigenen Attributen und Methoden erweitert werden. B *erbt* diese Attribute von A. Dann ist A die *Basisklasse*,

manchmal auch als *Elternklasse* bezeichnet, von der B abgeleitet wird. Im englischen werden oft auch die Begriffe *Superklass* (Oberklasse) und *Subclass* (Unterklasse) verwendet. Subtypenbeziehungen gestalten sich ähnlich zur Vererbung. Damit eine Subtypenbeziehung besteht, müssen Objekte des Typs B auch als Objekte des Typs A gelten. Nicht nur die Attribute des Objekts vom Typ B stimmen mit denen vom Typ A überein, auch die Zugriffsrechte und Sichtbarkeit auf die verschiedenen Attribute müssen übereinstimmen. Das heißt, dass alle Attribute und Methoden, die in Typ A zur Verfügung stehen, auch in Typ B zur Verfügung stehen. Wenn diese Bedingung erfüllt ist, dann können auch immer Objekte vom Typ B als Argumente an Parameter des Typen A gegeben werden. Im englischen wird gesagt B ist ein *subtype* (Untertyp) von A beziehungsweise A ist ein *supertype* (Obertyp) von Typ B. Im Allgemeinen ist es so, dass es immer, wenn die Funktionalität der Vererbung gegeben ist, auch immer die Funktionalität für Subtypen in Programmiersprachen existiert.

Ein Datentyp, der Vererbung eingearbeitet hat, wird auch als Klasse bezeichnet (s.O.).

```
1  class Disk extends Point {
2          private double radius;
3          private Color color;
4          public double getRadius() { return radius; }
5          public void setRadius(double radius) {
6                  if (radius > 0) this.radius = radius;
7                  }
8          public Color getColor() { return color; }
9          public void setColor(Color color) { this.color =
               color; }
10 }
```

Listing 2.4: Erweiterung der Point Klasse aus Listing 2.3 [Yae14]

Im Listing 2.4 wird die Klasse „Point“ aus 2.3 erweitert. Da eine Scheibe einen Punkt als Basis hat, kann die Klasse auch so erweitert werden, dass man einen Radius und z.B. eine Farbe hinzufügt. Die Vererbung selber ändert nichts an der Verwendung dieser neuen Klasse. Sie muss immer noch Instantiiert werden und über Methoden ihre Parameter gesetzt werden, dafür dienen die Methoden setRadius(), SetColor() und die geerbte Methode setXY(). Es ist also Beispielweise möglich mit den folgenden Befehlen ein entsprechendes Objekt zu erstellen siehe Listing 2.5.

```
1  Disk d = new Disk();
2  d.setXY(100,210);
```

```
3 d.setRadius(25.7);
4 d.setColor(Color.red);
```

Listing 2.5: Instantiierung der Klasse Disk [Yae14]

Es ist zu bemerken, dass die Methoden über das Objekt aufgerufen werden. Es gibt die Möglichkeit auch Methoden auszuführen, die über Klassen ausgeführt werden. Solche Methoden nennt man auch statische Methoden, sie werden über das Schlüsselwort *static* im Quelltext der Sprachen Java, C++ und C# identifiziert.

2.4 Polymorphismus

Subklassen sind oft spezialisierte Klassen daher notwendig , dass ein spezialisiertes Objekt ihre Methoden anders ausführt wir benötigen die Möglichkeit Methoden zu überschreiben *override*

Subklassen sind oft spezialisierte Klassen, die von einer Oberklasse erben. Oft ist es zu erwarten, dass eine solche spezialisierte Klasse sich anders verhält, wenn man eine Methode der Oberklasse aufruft. Man kann zur Veranschaulichung eine Klasse betrachten, die ein beliebiges Tier darstellen soll siehe Listing 2.6.

```
1  class Tier
2  {
3      private String name;
4      private short alter;
5
6      public void gibLaut()
7      {
8          System.out.println("Geraeusche eines allgemeinen
               Tiers");
9      }
10
11 }
```

Listing 2.6: Klasse Tier

Wenn man nun Unterklassen von der Klasse Tier ableitet, könnte man erwarten, dass die Methode *gibLaut()*, je nachdem, um was für ein Tier es sich handelt, eine andere Ausgabe verursacht. Es ist also notwendig eine Möglichkeit zu haben die Verhaltensweisen und Methoden von Oberklassen zu überschreiben. Diese veränderbarkeit der Implementation von Funktionen nennt man auch Polymorphismus.

So kann man in der Programmiersprache Java mit der Deklaration *Override* eine Funktion überschreiben siehe 2.7.

```
class Katze extends Tier
{
    @Override
    public void gibLaut()
    {
        System.out.println("Miau");
    }
}
```

Listing 2.7: Klasse Katze

Je nach Sprache ist diese Funktionalität anders implementiert. In der Sprache C++ werden Funktionen, die überschreibbar sein sollen mit dem Schlüsselwort *virtual* gekennzeichnet siehe 2.8.

```
class Tier {
        public:
        virtual char * gibLaut() { return "Geraeusche eines
            allgemeinen Tiers"; }
        };

class Katze : public Tier {
        public:
        char * gibLaut() { return "Miau"; }
};
```

Listing 2.8: Tier und Katzen Beispiel in C++

2.5 Überladen

Überladen bezieht sich auf Funktionen und Operatoren und bedeutet, dass diese, unter der Verwendung des gleichen Namens, in Abhängigkeit der Typen der übergebenen Parameter verschieden ausgeführt werden. Betrachtet man zum Beispiel die Addition als Operator, so wird bei der Berechnung der Summe von zwei Zahlen je nach Typ der Zahlen die Operation anders ausgeführt. Wenn man Zahlen von verschiedenen Typen addieren möchte, dann wird oft impliziert eine Umwandlung der Typen vollzogen. Zum Beispiel wenn eine ganze Zahl mit einer Fließkommazahl addiert wird, dann wird die ganze Zahl in eine Fließkommazahl umgewandelt, bevor

sie addiert werden. Der + Operator für die Addition wird in einigen Sprachen auch so überladen, dass mit ihm Text verkettet werden kann.

Es ist Programmierern somit auch möglich selber Operatoren oder Funktionen zu überladen.

3 Umgebungen

3.1 Java Swing

Java Swing ist eine Graphical User Interface Bibliothek für Java. Es gehört zu den Java Foundation Classes, eine Sammlung von Bibliotheken für dir Programmierung von graphischen Benutzerschnittstellen. Andere Bibliotheken in dieser Sammlung sind zum Beispiel Java 2d, Java Accessibility API, Drag-and-Drop-API und das Abstract Window Toolkit (AWT) [Ora22]. Swing baut auf AWT auf und ist mit den anderen APIs verwoben. Es gibt auch Konkurrenten zu Swing, z.B. das für Eclipse entwickelte SWT. Auf dieses wird aber nicht eingegangen, da Swing auch mit anderen Entwicklerwerkzeugen und -umgebungen Kompatibel ist. Swing ist, im Gegensatz zu AWT, Plattformunabhängig. Außerdem ist es modular und objektorientiert aufgebaut. Die Plattformunabhängigkeit wird dadurch gewährleistet, dass alle Swing-Komponenten direkt von Java gerendert werden. Nachteil davon ist, dass Swing Anwendungen nicht wie Anwendungen aussehen, die nativ auf dem darunterliegenden Betriebssystem entwickelt wurden. Es gibt so genannte „Look-and-Feels“, die das Aussehen und Verhalten modifizieren.

3.2 Unreal Engine

Unreal Engine ist eine der am weitesten verbreiteten und fortschrittlichen 3D Entwicklungsumgebungen. Sie wird unter anderem für das Aufnehmen von Animationsfilmen, Präsentationen von Prototypen und zur Erstellung von Computerspielen verwendet. Für das Programmieren können entweder Blueprints, Unreal‘s eigene visuelle Programmiersprache oder C++ verwendet werden. Ein fertiges Projekt wird am Ende für das entsprechende Zielsystem ausgeliefert. Unreal unterstützt hierbei alle verbreiteten Plattformen wie Windows, Mac, Linux und Android.

3.3 Unity Engine

4 Verwendete Werkzeuge

4.1 Git und Github

4.2 Eclipse

Eclipse ist eine integrierte Entwicklungsumgebung die hauptsächlich zum programmieren in der Sprache Java dient. Es bietet ein ausgiebiges Plug-in System mit dem es erweitert werden kann und auch in anderen Programmiersprachen entwickelt werden kann. Es wird heutzutage von der Eclipse Foundation entwickelt. Beliebte alternativen zu Eclipse für die Entwicklung von Programmen in Java sind unter anderen zum Beispiel IntelliJ IDEA von JetBrains und NetBeans von Apachi.

4.3 Unreal Engine Editor

Der Unreal Engine Editor ist die Entwicklungsumgebung für Unreal Engine Projekte. Im Editor sind viele Werkzeuge enthalten, mit denen verschiedenste Datentypen, die bei der Entwicklung verwendet werden können, bearbeiten werden können. Die Auswahl hierbei ist so groß, dass ganze Projekte entwickeln werden können, ohne ein anderes Programm starten zu müssen.

4.4 Unity Engine Editor

5 Realisierung

5.1 Zusammenfassung der Anforderungen

Wie in 1 Beschrieben sind für dieses Projekt einige Anforderungen gegeben, die Umgesetzt werden sollen. Für die Betrachtung der Entwicklung dieses Projekts in den verschiedenen Umgebungen werden also folgende Anforderungen festgelegt. Diese Funktionen werden nach den „MASTeR - Die SOPHIST-Templates für Requirements“ von Sophist formuliert [SG22].

Anforderungen Spielstart

- Wenn das Programm gestartet wird, muss das Spiel 1 Spielfeld, 1 Schatz und 3 Piraten Objekte erstellen.
- Wenn die Piraten erstellt werden, muss das Spiel ihnen verschiedene Geschwindigkeiten zuordnen.
- Wenn das Programm gestartet wird, muss das Spiel die Graphische Oberfläche darstellen.
- Wenn das Programm aktualisiert muss das Spiel die Piraten **zufällig** über das Spielfeld bewegen.

Während des Spiels

- Wenn sich ein Pirat bewegt, sollte das Spiel ein Feld wählen, dass er in den letzten 5 Positionen nicht betreten hat.
- Wenn das Spiel gestartet ist, muss das Spiel eine Option für die Sichtbarkeit des Schatzes anbieten.
- Wenn ein Pirat in der Nähe des Schatzes ist, muss das Spiel den Schatz sichtbar machen, wenn er unsichtbar ist.
- Wenn ein Pirat in der Nähe des Schatzes ist, muss das Spiel ihn Richtung Schatz bewegen. (Nicht mehr zufällig.)
- Wenn ein Pirat in der Nähe des Schatzes ist, muss das Spiel die anderen Piraten in Richtung des Schatzes bewegen.

5.2 Umsetzung in Java

Notiz: Der Quelltext der Umsetzung in Java ist auf https://github.com/MarioSchenkewitz/IT3161-Studienprojekt-Schatzinsel-Java/ zu finden.

Um das Spiel zu entwerfen werden verschiedene Klassen benötigt. Zum einen die Klassen, die die Objekte im Spiel darstellen und zum anderen Klassen um das Spiel selber zu verwalten siehe Abbildung 1.

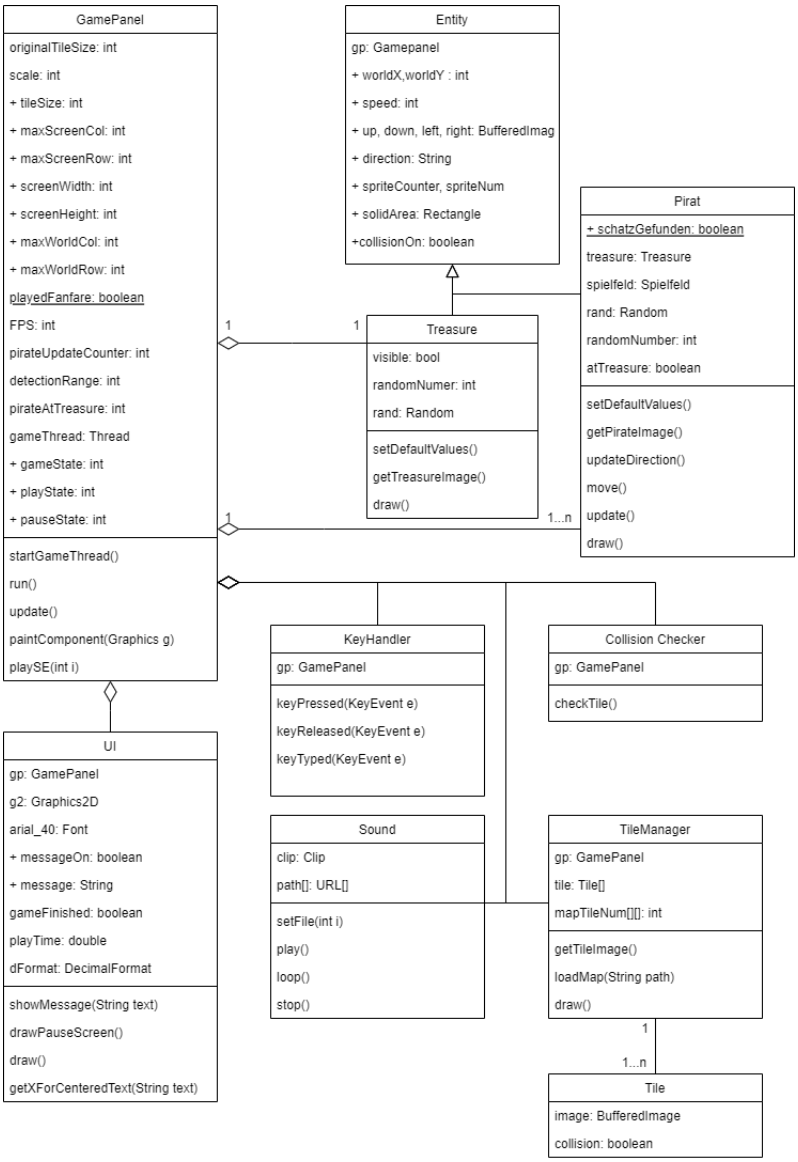

Abbildung 1: UML-Klassendiagramm für die Umsetzung in Java

Die zentrale Klasse des Spiels ist die Klasse GamePanel, sie ist das Fenster in dem sich alles abspielt. In der Klasse Main wird ein Fenster erzeugt, in dem sich das

GamePanel befindet. In dieser Klasse werden auch alle Einstellungen festgelegt, die Frames die pro Sekunde dargestellt und berechnet werden sollen, Fenstergröße, Skalierung, wie viele Piraten vorhanden sein sollen und weitere Einstellungen. Für die Entitäten im Spiel selber gibt es eine Überklasse, Entity, von der die anderen Klassen Pirat und Schatz abgeleitet werden. Zur Verwaltung des Spiels sind noch weitere Klassen notwendig.

Verwaltungsklassen

- GamePanel: Wie oben erwähnt bestimmt sie das Spielfenster und die Einstellungen.
- KeyHandler: Implementiert den KeyListener aus der AWT-Bibliothek, der Tastatureingaben verarbeitet.
- TileManager und Tiles: Tiles sind die Spielfeldkacheln, sie stellen die Insel dar und verwalten das Laden der Kacheln aus den verschiedenen Bilddateien und der Kartendatei.
- UI (User Interface): Sie ist für den Text auf dem Bildschirm zuständig,
- Sound: Nutzt die Java Sound Bibliotheken um Tondateien abzuspielen. In unserem Fall die Fanfare, wenn der Schatz gefunden wird.
- CollisionChecker: Überprüft ob die Kacheln, auf die sich eine Entität zubewegt Kollision auslöst.

Es werden also alle Verwaltungsklassen in entsprechenden Objekten instanziiert, die Funktion zum starten des Threads definiert. Hier wird auch der Status des Spiels festgelegt, um pausieren zu erlauben.

Moderne Prozessoren sind sehr schnell und können die Aktualisierung der Spielzustände und der Darstellung sehr schnell berechnen. Um das Spiel verfolgen zu können muss die Rate mit der das Spiel dargestellt wird begrenzt werden, siehe Listing 5.1.

```
1 public void run() {
2         double drawInterval = 1000000000/FPS;
3         double delta = 0;
4         long lastTime = System.nanoTime();
5         long currentTime;
6 
7         while(gameThread!= null) {
8                 currentTime = System.nanoTime();
9                 delta += (currentTime - lastTime) /
                      drawInterval;
```

```
10                      lastTime = currentTime;
11
12                      if(delta >=1) {
13                              update();
14                              repaint();
15                              delta--;
16                      }
17              }
18      }
```

Listing 5.1: Beschränkung der Aktualisierungen

In der Funktionen update() befindet sich die Logik des Spiels, dessen Auswirkungen im Anschluss mit der Funktion repaint() erneut gezeichnet werden. Die Details zu der Logik und allen weiteren Klassen sind im GitHub Repository zu sehen, siehe Notiz am Anfang des Abschnitts 5.2.

Die Funktion update() überprüft zuerst in welchem Zustand sich das Spiel befindet. Mit der selben Logik wie in Listing 5.1 wird auch in der update() Funktion sicher gestellt, dass die Piraten nur in bestimmten Zeitabständen ihre Richtung wechseln, dies geschieht in der update() Funktion der Klasse Pirat. Wenn der Schatz jedoch gefunden wurde, bewegen sie sich kontinuierlich auf den Schatz zu. Der Rest der Logik in dieser Funktion überprüft den Abstand jedes Piraten zum Schatz und setzt den Status der Piraten dem entsprechend, so dass sie sich nicht mehr bewegen, wenn sie am Schatz sind bzw. dass das Spiel zu Ende ist, wenn alle am Schatz sind und eine kleine Fanfare abgespielt wird.

Die repaint() Funktion ist eine Funktion der AWT-Bibliothek und sorgt dafür, dass die paintComponent() Funktion aufgerufen wird. Diese zeichnet abhängig vom Spielzustand die Entsprechenden Darstellungen auf den Bildschirm. Dafür wird Hauptsächlich die Verwaltungsklasse UI in Anspruch genommen.

5.3 Umsetzung in Unreal Engine

Notiz: Der Quelltext der Umsetzung in Unreal ist auf https://github.com/TheidenHD/Studienprojekt/ zu finden.
Das fertige Projekt ist auf https://github.com/TheidenHD/Studienprojekt/releases/tag/1.0/ zu finden.

Aufgrund der Komplexität von Unreal werden einige Standard-Klassen benötigt, die keinen großen Einfluss auf das Endprodukt haben. Beispiele sind hier der GameMode, welcher unter anderem zur Definition von Standard Klassen und die GameInstance, welche hier zum Übermitteln von Variablen zwischen bestimmten Abschnitten im Projektablauf.

5.3.1 UI

Die Benutzeroberflächen werden aus einzelnen Bausteinen zusammengesetzt, die am Ende über das laufende Projekt gerendert werden. Wir verwenden dies im Hauptmenü, wo wir ein laufendes Level haben, damit wir einen animierten Hintergrund haben und im Pausemenü, wo wir die Ausführung des Level pausieren, die Bewegungskontrolle deaktivieren und einen Unschärfe-Effekt über den Hintergrund legen. Das Klicken der Knöpfe erzeugt Events, die wir behandeln müssen, um den erwünschten Effekt zu erhalten.

5.3.2 Level

Das Projekt hat zwei Level, das Erste wird als Hintergrund für das Hauptmenü benutzt. In ihm befinden sich lediglich einige animierte Objekt und eine statische Kamera. Das Zweite ist das Spielfeld, in dem die Piraten nach dem Schatz suchen. Das Spielfeld ist statisch designt, das heißt, die Piraten starten immer an der gleichen Stell und der Schatz wird beim Start an einer von 6 vordefinierten Positionen zufällig platziert. Der Schatz kann hierbei unsichtbar sein, dies ändert allerdings nur die Sichtbarkeit für den Spieler und hat keinen Einfluss auf die Erkennbarkeit durch die Piraten und auf die Kollision. Die Piraten haben unterschiedliche Bewegungsgeschwindigkeiten, Blickwinkel und Sichtweiten. Diese werden in jeder Instanz durch Variablen definiert und nach dem Platzieren selbständig eingelesen und in den jeweiligen Komponenten hineingeschrieben. Außerdem verwendet eine Instanz das weibliche Modell und alle anderen das standard-männliche Modell.

5.3.3 Künstliche Intelligenz

Für die Künstliche werden 5 verschiedene Objekte verwendet. Vier davon sind für den Piraten und das fünfte ist der Schatz, den die Piraten suchen. Als erstes haben wir den Charakter, welcher alle physischen Komponenten beinhaltet, wie das Bewegungsmodul, die Sensorik und das Modell, welches für den Spieler sichtbar ist. Außerdem ist er für die Animation des Modells zuständig. Als zweites haben wir den Controller, welcher als Gehirn für den Piraten fungiert und alle Entscheidun-

gen trifft. Als drittens haben wir einen Entscheidungsbaum, welcher ein Flowchart ist, für alle Entscheidungen des Controllers. Und zu guter letzt haben wir noch ein Blackboard, welches Variablen für den Entscheidungsbaum bereithält. Der Schatz hat keine besonderen Bestandteile und wird lediglich im Controller als Ziel für die Sensorik definiert.

5.3.4 Sensorik

Die Sensorik kann Ziele auf zwei unterschiedliche Methoden wahrnehmen, visuell und auditiv. Da wir hier nicht mit Geräuschen arbeiten, können wir den auditiven Teil einfach ignorieren. Beim visuellen teil wird ein Kegel vom Kopf des Charakters aufgespannt und für alle Objekte der Klasse Pawn und deren Unterklassen die sich in diesem Kegel befinden und nicht durch ein anderes undurchsichtiges Objekt verdeckt werden, wird ein Event ausgeworfen, in dem man nun überprüfen muss ob dieses Objekt das Gesuchte ist.

5.3.5 Bewegung

Damit sich die Piraten bewegen können, benötigen wir zwei Komponenten, das Level in dem der Pirat sich bewegt und das Bewegungsmodul im Piraten selbst. Damit er sich durch das Level bewegen kann, benötigt er mindestens eine Fläche und eine Definition, welche der Flächen begehbar sind. Diese Definition ist das NavMesh, welches dynamisch durch das NavMeshBoundsVolume im laufenden Betrieb erstellt wird. Wenn sich der Pirat nun im NavMesh befindet und einen Punkt bekommt, der sich ebenfalls im NavMesh befindet, wird er sich auf den kürzesten Weg zu diesem Punkt bewegen.

5.3.6 Ablauf

Wenn die Piraten auf dem Level erscheinen, überprüfen sie zuerst, ob der Schatz bereits gefunden wurde. Wenn dies der Fall ist, gehen sie zum Schatz und ignorieren alles andere. Wenn dies nicht der Fall ist, suchen sie sich eine zufälligen Punkt in einem bestimmten Radius, zu dem sie sich bewegen können und gehen dann zu ihm. Während der Bewegung halten die Piraten Ausschau nach dem Schatz und wenn sie ihn finden, bleiben sie sofort stehen, holen sich vom Level eine Liste von allen Instanzen derselben Klasse und benachrichtigen diese, dass der Schatz gefunden wurde und wo er sich befindet. Diese Funktion befindet sich in einer Endlosschleife, die entweder nach dem Erreichen des Punktes oder dem Finden des Schatzes von vorne beginnt.

5.4 Umsetzung in Unity Engine

6 Ergebnisse

7 Fazit

8 Ausblick

Literaturverzeichnis

[Ora22] Oracle (2022). Java Foundation Classes (JFC). https://www.oracle.com/java/technologies/foundation-classes-faq.html.

[SG22] SOPHIST-GmbH (2022). MASTeR - Die SOPHIST-Templates für Requirements. https://www.sophist.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/Wissen_for_free/MASTeR_-Kern_RE-Plakat2.pdf.

[Wir71] Wirth, N. (1971). *The Programming Language Pascal.* Acta Informatica 1. pp. 35–63.

[Yae14] Yaeger, D. P. (2014). *Object-Oriented Programming Languages And Event-Driven Programming.* Mercury Learning and Information, Dulles, Virginia; Boston, Massachusetts; New Delhi.

[Zim20] Zimmermann, A. (2020). Einführung in die Programmierung, Aggregierte Datentypen Übung. http://azdom.de/hwr/prog/.

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich:

1. dass ich meine Bachelor-Thesis selbstständig verfasst habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe,

3. dass ich meine Bachelor-Thesis bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum	Jahn, Marko
Ort, Datum	Schenkewitz, Mario
Ort, Datum	Zilius, Sven