CakeProtector

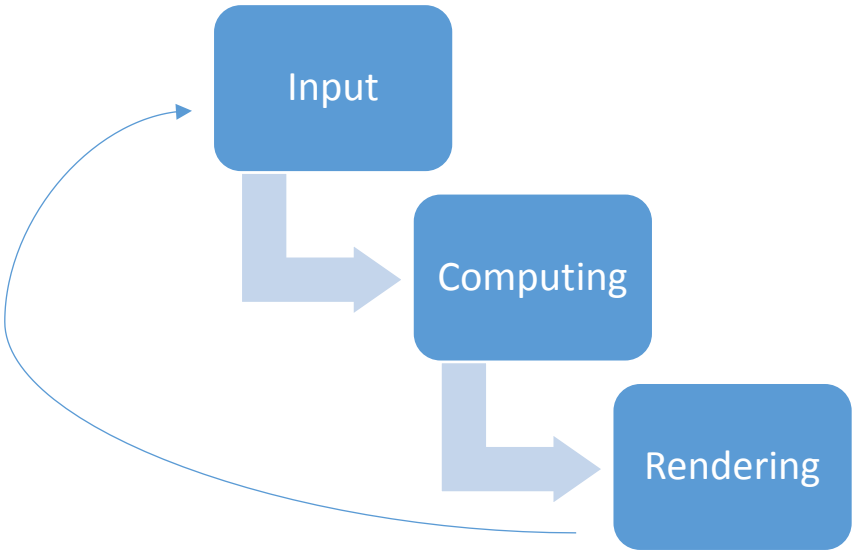
Dokumentation

07.03.2016

INHALTSVERZEICHNIS

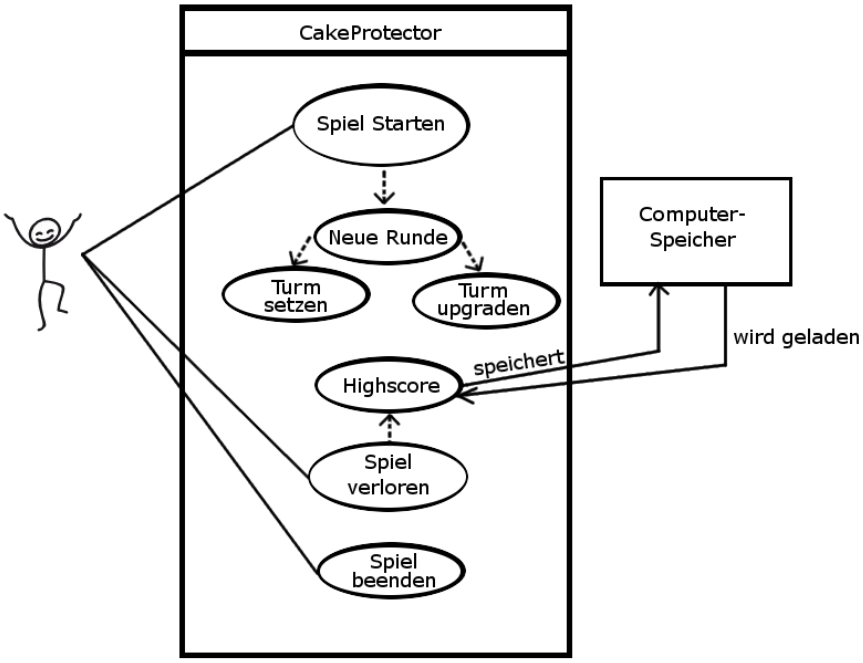
GLIEDERUNG SEITE

GUI ABLAUF



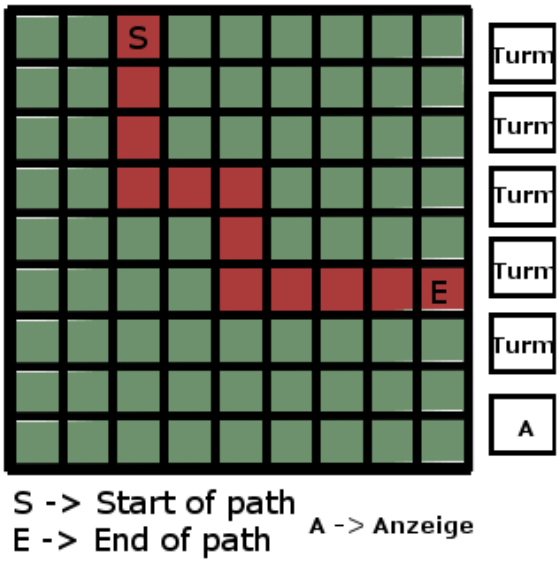
Ein GUI (Grafical User Interface) ist meistens durch eine einfache Loop aufgebaut.
In dieser Loop wird erst auf die Eingabe des Users gewartet, dann werden diese Daten wie z.B. Koordinaten berechnet und weiter verarbeitet. Zum Schluss werden dann die Daten in grafischer Form ausgegeben.

USE CASE DIAGRAMM



GUI SKIZZE

Da wir das Spiel so übersichtlich wie möglich gestalten wollten, haben wir uns bei der Grundskizze des Programms auf die wichtigsten Bestandteile eines Tower Defense-Spiels konzentriert und zwei übersichtliche „Felder“ gezeichnet. Rechts befindet sich die Auswahl der Türme und links das Spielfeld. Die Monster werden bei [S] erscheinen, dem restlichen rot markierten Pfad folgen und bei [E] wieder vom Spielfeld gelöscht werden.
In dieser Skizze waren noch keine Steine vorhanden.



SPIELIDEE

Wir haben uns als Gruppe dazu entschieden „Cake Protector“ zu programmieren, da das Spielprinzip fast jedem geläufig ist und sehr viel Spielraum bietet. Es gibt tausende verschiedene Versionen und Konzepte von „Tower Defense“, wobei das grundlegende Konzept immer erhalten bleibt. Daher kann man sich von vielen verschiedenen „Tower Defense“ Spielen Inspiration beschaffen und damit dann sein ganz eigenes Kunstwerk kreieren und modifizieren. Des Weiteren bietet das Programmieren von „Cake Protector“ Herausforderungen für die Gruppenmitglieder, ist jedoch nicht zu anspruchsvoll um diesen Herausforderungen nicht gewachsen zu sein.

SPIELPRINZIP

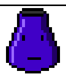
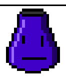
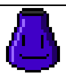
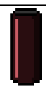
Das Spielprinzip ist relativ einfach: Wenn die Gegner (Gobls) den Kuchen erreichen ist das Spiel verloren. Man muss dabei als Spieler dies solange wie möglich verhindern. Dabei muss man die Türme geschickt auf dem Spielfeld platzieren und upgraden. Die Türme sorgen dafür, dass die Gegner kampfunfähig gemacht werden. Die Gegner folgen der rot markierten Fläche und auf den restlichen freien Feldern (grünen) können Türme gesetzt werden. Sollte ein Gegner doch das Ende des Pfades erreichen und zum Kuchen gelangen, wird dieser ein Stück des Kuchens auffressen und der Spieler verliert ein Leben. Das Spiel ist dann vorbei, wenn die Angreifer den gesamten Kuchen aufgefressen haben. Das Ziel des Spieles ist es die Türme so geschickt wie möglich zu platzieren und die Stufen der Türme dem Spielgeschehen anzupassen, sodass selbst die Monster mit höheren Werten nicht mehr an den Kuchen kommen können.

ANLEITUNG

„Tower Defense“ ist von seinem Spielkonzept so simpel und gleichzeitig komplex wie genial. Wir als Gruppe haben unser Spiel, das dem Prinzip des „Tower Defense“ entspricht, auf den Namen „Cake Protector“ getauft. Dem Spieler stehen verschiedene Türme zur Verfügung, die alle ihre Vor- und Nachteile haben. Durch geschicktes auswählen des zur aktuellen Situation passenden Turmes und einer strategisch durchdachten Positionierung desselben auf dem Spielfeld ist es die Aufgabe des Spielers seinen Kuchen vor feindlich gesinnten Angreifern, in Form von „Gobls“, zu verteidigen und solange wie möglich zu beschützen. Damit dieses Unternehmen nicht zu leicht gestaltet ist, kosten die Türme je nach Art des Turms verschieden viel Geld. Die Preisspanne der Türme reicht von sehr günstig bis extrem teuer, wobei hier teurer nicht immer gleich besser bedeutet. Der Spieler erhält vom Spiel aus jede Sekunde 2 Euro und weiterhin für jeden besiegten Angreifer 2 Euro. Das Spiel ist verloren, wenn die Monster den ganzen Kuchen verschlungen haben.

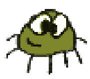
TÜRME

Das Werkzeug des Spielers sind die Türme. Es gibt 5 verschiedene Arten von Türmen, dessen Stufe man 5 Mal erhöhen und somit ihre Werte verbessern kann. Sie können von der In-Game-Währung gekauft werden und variieren vom Preis her sehr stark. Bei den Türmen gibt es zwei verschieden relevante Werte: Die Geschwindigkeit zwischen den Abschuss Intervallen (Cool-Down-Phase) und der Schaden, welchen jedes Projektil anrichtet (Schaden). Der günstigste Turm ist mit dem ersten Monster gleich zu setzen. Seine Werte zeigen nichts Besonderes aber er ist sehr günstig. Dennoch sollten man ihn nicht unterschätzen, denn seine Werte werden durch Upgrades verbessert. Der zweite Turm hat eine besonders niedrige Cool-Down-Phase. Er kann die Projektile sehr schnell abfeuern und die Angriffsgeschwindigkeit erhöht sich bei jedem Upgrade, allerdings richten seine Projektile eher geringen Schaden an. Bei dem dritten Turm ist der Schaden besonders hoch, denn seine Werte für die Projektil-Stärke wurden erhöht, aber im Gegenzug ist seine Cool-Down-Phase sehr hoch. Neben den drei Türmen die eine Äquivalenz zu den Monstern darstellen, haben wir auch zwei besondere. Der vierte Turm hat eine ziemlich geringe Cool-Down-Phase und richtet großen Schaden an. Er steht in Nichts den ersten drei Türmen hinterher, denn all seine Werte sind besser als die der vorigen. Er ist jedoch sehr teuer. Der letzte Turm ist vom Schaden her der Stärkste. Seine Schadenswerte sind sehr hoch aber seine Cool-Down-Phase ist mit Abstand die größte. Er ist der teuerste und stärkste Turm im Spiel.

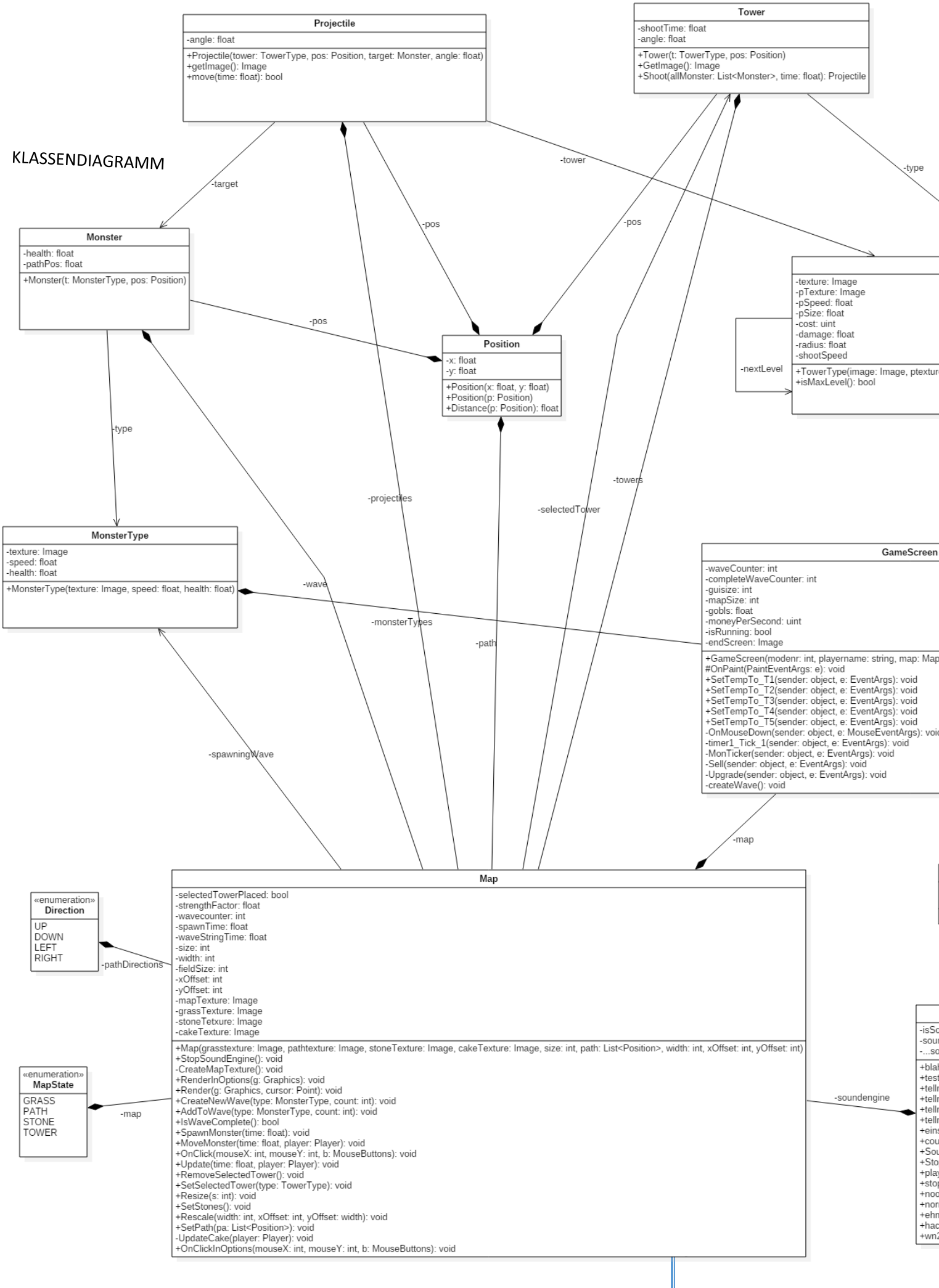
The-Bob	Tripple-Nipple	Power-Bunga	The-Stick	Puke-Nuke
				
				
				
				
				

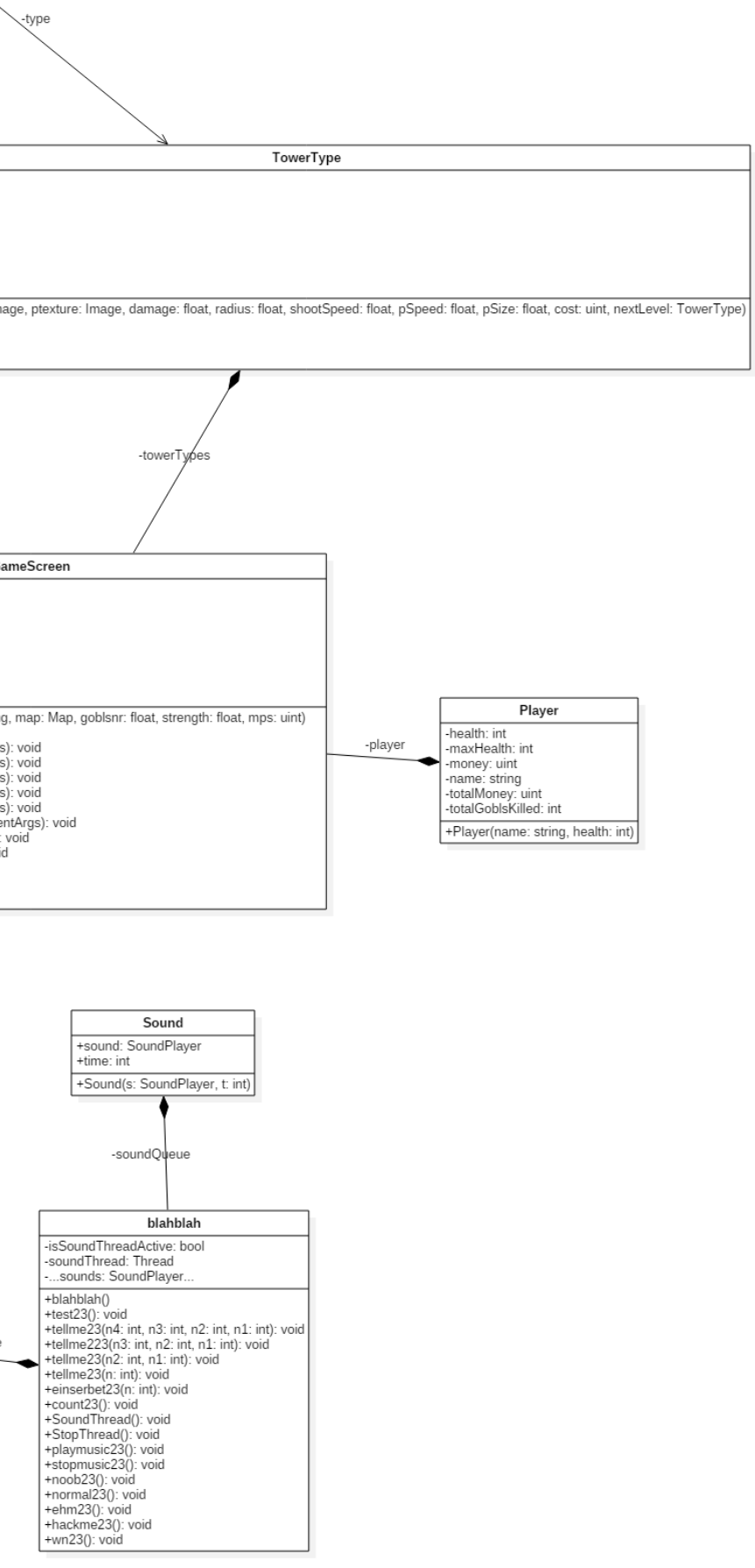
MONSTER

Es gibt drei verschieden Arten von Monstern. Die standard Monster sind in all ihren Werten gewöhnlich. Diese tauchen auch am meisten auch. Es gibt nichts Besonderes an ihnen. Sie haben eine normale Geschwindigkeit und normale Leben. Bei der zweiten Monsterart wurden die Geschwindigkeitswerte erhöht. Sie sind viel schneller als die normalen Monster. Es wäre vorteilhaft einen schnell schießenden Turm gegen diese Art von Monster zu verwenden. Die dritte Art der Monster ist so zu sagen die stärkste. Bei diesen wurden die Leben stark erhöht aber dafür die Geschwindigkeitswerte etwas niedrig gehalten. Dennoch sollte man diese Art von Monstern nicht unterschätzen. Bei diesen Monstern wäre es vorteilhaft Türme zu verwenden, welche viel Schaden anrichten.

Just A Normal Gobl	Mr. Aimgonnafagja	The BOSS
		

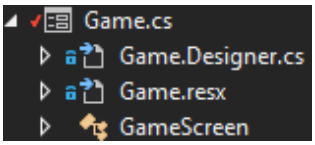
KLASSENDIAGRAMM




-type
TowerType
nage, ptexture: Image, damage: float, radius: float, shootSpeed: float, pSpeed: float, pSize: float, cost: uint, nextLevel: TowerType)
-towerTypes
ameScreen
g, map: Map, goblsnr: float, strength: float, mps: uint)
s): void
s): void
s): void
s): void
s): void
entArgs): void
: void
id
-player
Player
-health: int
-maxHealth: int
-money: uint
-name: string
-totalMoney: uint
-totalGoblsKilled: int
+Player(name: string, health: int)
Sound
+sound: SoundPlayer
+time: int
+Sound(s: SoundPlayer, t: int)
-soundQueue
blahblah
-isSoundThreadActive: bool
-soundThread: Thread
-...sounds: SoundPlayer...
+blahblah()
+test23(): void
+tellme23(n4: int, n3: int, n2: int, n1: int): void
+tellme223(n3: int, n2: int, n1: int): void
+tellme23(n2: int, n1: int): void
+tellme23(n: int): void
+einserbet23(n: int): void
+count23(): void
+SoundThread(): void
+StopThread(): void
+playmusic23(): void
+stopmusic23(): void
+noob23(): void
+normal23(): void
+ehm23(): void
+hackme23(): void
+wn23(): void

Implementierung

Wir haben uns dazu entschieden die objekt- und eventorientierte Programmiersprache C# zu verwenden, da diese zusammen mit Windows Forms einen einfachen und schnellen Weg bietet, Fenster und Grafiken zu erstellen und anzuzeigen. Das Programm besteht aus 18 Klassen, die sich im `namespace Towerdefense` befinden. Eine statische Klasse „Program“, welche die Main-Funktion enthält, 6 Fenster Klassen und 9 Klassen für den Spielmechanismus.



Die Fenster Klassen sind partielle Klassen die von der Windows Forms Klassen Form abgeleitet werden. Sie bestehen aus drei Dateien. Einer C# Datei in der die Eventmethoden und der Konstruktor stehen, einer C# Datei die vom Windows Forms-Designer erstellt wurde und den Code der einzelnen Elemente auf dem Fenster enthält, und noch aus einer Ressource Datei für die Bildverwaltung (welche wir in unserem Falle nicht verwendet haben, da wir die Grafiken direkt aus einem festen Speicherort geladen haben). Die vom Designer erstellte Datei kann man auch manuell bearbeiten, aber man läuft somit Gefahr, dass das Fenster im Designer nicht mehr korrekt bis gar nicht mehr angezeigt wird.

Da wären folgende Klassen, welche später näher beschrieben werden:

<u>Main:</u>

- Program

<u>Fenster Klassen:</u>

- Menu
- Options
- Game
- MSGBox
- YorNO
- Highscore
- HighscoreScreen
- Credits

<u>Klassen:</u>

- Player
- Position
- Monster
- MonsterType
- Tower
- TowerType
- Projectile
- blablah23
- Map

Main

1)

Program Klasse

Zuerst wird das Attribut `[STAThread]` eingebunden welches angibt, dass dieses Threadmodell ein Singlethread-Apartment ist. Dieses MUSS am Anfang einer Kompletten Anwendung stehen, da das Programm nicht richtig funktionieren wird wenn dieses nicht vorhanden ist. Ohne das `[STAThread] wird` ein Multithreaded-Apartmentmodell verwendet, das nicht mit Windows Forms Kompatibel ist.

In der statischen Void Methode `Main()` wird zunächst `Application.EnableVisualStyles();` aufgerufen, welches die visuellen Stile aktiviert.

Daraufhin wird die Methode `Application.SetCompatibleTextRenderingDefault(false)` aufgerufen, welche die Voreinstellung für die „`UseCompatibleTextRendering`“-Eigenschaft aufruft, welche bei bestimmten Steuerelementen festgelegt ist.

Daraufhin wird die Form `Menu` aufgerufen, von der das Programm aus gestartet wird.

Fenster

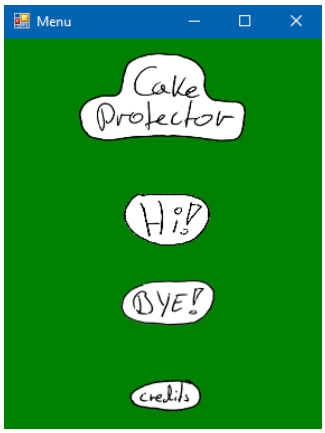
2)

Menu Form

Da der Code von `menu.cs` in der Datei `menu.Designer.cs` vom Visual-Studio-Designer erstellt wird und nur für das visuelle Design zuständig ist, werden wir diese Datei sowohl hier, als auch bei den anderen partiellen Klassen nicht behandeln. Wir beziehen uns hier nur auf die für uns relevanten Teile der Klassen.

Am Anfang der `menu.cs` wird zunächst eine neue Private Instanz der `ComponentResourceManager` mit Standartwerten erzeugt, welche dann später benötigt wird, um bestimmte Grafik-Dateien aus dem Speicher des Computers zu laden.

Im Konstruktor werden mit der Methode `InitializeComponent();` die Komponenten, bzw. die Objekte aus der `menu.Designer.cs` Initialisiert. Da wären zum einen das `logo`, Picturebox `bt_hi`, Picturebox `bt_bye` und die Picturebox `bt_credits`. Diese Pictureboxen werden, um ein wenig vom sturen Windows-Design abzuweichen, von uns quasi als „Buttons“ verwendet.



Da alle Picturebox-„Buttons“ gleich aufgebaut sind, werden alle allgemein erläutert und dann bei Abweichungen in Bezug auf den jeweiligen „Button“ genauer erklärt.

Jedes dieser 3 Objekte besitzt jeweils 3 Methoden, welche bei folgenden Events aufgerufen werden:

1. MouseEnter
2. MouseLeave
3. Click

Bei dem MouseEnter Event wird geprüft, ob der Cursor auf das entsprechende Objekt in einer Form zeigt, das MouseLeave Event schaut, wenn der Cursor dieses verlässt und das Click Event schaut ob in das Objekt hinein geklickt wird.

Bei MouseEnter und MouseLeave wird bei allen drei nur die Textur geändert.
Wenn auf `bt_hi` geklickt wird, wird eine `options` -Form namens „Options“ erstellt, die aktuelle Form wird mit `this.Show()` „versteckt“, daraufhin wird die Form namens „Options“ gezeigt und nachdem diese wieder geschlossen wird, wird sofort das Menu Fenster wieder angezeigt.

Bei `bt_credits` ist es fast so ähnlich, nur wird hier das Fenster „Credits“ erstellt und geöffnet. Bei `bt_hi` wird das Programm beim Klick lediglich geschlossen.

3)
Options Form

Zunächst kann man auf der oberen Seite des Fensters den Spielernamen eingeben, welcher dann in das Gamefenster und schlussendlich in das Highscore-Fenster übergegeben wird.

Daraufhin kann man zwischen dem RadioButton „`ratedmode`“ und „`freemode` auswählen. Im Ratedmode werden die Standartwerte für eine „default-Map“ verwendet und man kann sich später in den Highscore eintragen. Im Freemode kann man seine eigene Map erstellen, den Weg setzen, wieviel Geld man pro Sekunde bekommt, die Mapgröße einstellen und die Anzahl der Monster (Gobls) festlegen.

Wenn man auf den „Okay Let’s Go“ Button klickt, wird ein Gamefenster erstellt und die ausgewählten Werte in das Game-Fenster übergeben.

Das Fenster Options besitzt das Attribut `map` welches ein Objekt der Klasse `Map` ist.
Im Konstruktor werden zunächst die Komponente Initialisiert und daraufhin wird die `mapBox` welches ein Objekt von `GroupBox` ist erstellt. Jetzt wird geschaut ob diese `Groupbox` quadratisch ist. Falls dies nicht der Fall ist, wird ein Error ausgegeben. Nun wird diese Methode als Event registriert. Daraufhin wird ein Container namens `path` des Typs `List` mit Objekten der Klasse `Position erstellt` und ihm werden sofort 7 neue Positionen hinzugefügt.

Danach wird eine „mini“ Version des Spielfeldes auf das Fenster gerendert. Die Methode OnPaint zeichnet diese `map` nach jedem Klick neu.

In der Methode `OKAY_Click` wird zunächst geprüft, ob `ratedmode` oder `freemode` angeklickt wurde. Falls `ratedmode` angeklickt wurde, wird ein Map-Preset ausgewählt und der Spielername und die `map` werden dem jetzt erstellten Fenster `TheGame` vom Typ `GameScreen` übergeben und dieses wird daraufhin angezeigt.

Falls dies nicht der Fall ist werden die ausgefüllten Daten im `freemode` dem Fenster übergeben und angezeigt. Wenn Daten nicht korrekt eingegeben wurden, werden Fenster des Typs `msgbox` erstellt und in ihnen wird der User aufgefordert die Daten Korrekt einzugeben.

In der Methode `OnCLick` wird, wenn der `freemode` ausgewählt wurde die Klickposition herausgefunden und eine Methode namens `OnClickInOptions` aus dem Objekt `map` aufgerufen.

Die Methode `numMapSizeX_OnValueChanged` verändert die Größe der Mini-Map und der späteren `map` mit der Methode `map.Resize`

Die Methode `ratedmode_CheckedChanged` wird aufgerufen wenn sich der Wert von `ratedmode` auf „`Checked`“ verändert. Sie aktiviert die weiteren Optionen die man im Fenster unten ausfüllen kann.

4)
Game Fenster

Das Fenster Games besitzt die Attribute

```
private Player player;
public Map map {get; set;}
private MonsterType[] monsterTypes;
private TowerType[] towerTypes;
private int waveCounter = 0;
private int completeWaveCounter = 0;
private int guisize;
private int fieldsizex;
private int fieldsizey;
private int gobls;
private uint moneyPerSecond;
private bool isRunning = true;
private Image endScreen = null;
```

`Player player` ist ein Objekt der Klasse `Player` und entspricht dem User. `Map map` ist ein Objekt der Klasse `Map` und steht für die Map auf der Gespielt wird. Diese wird später von dem Fenster gerendert. Das Objekt-Array `monsterTypes` der Klasse `MonsterType[]` steht für die verschiedenen Monstertypen und das Objekt-Array `towerTypes` der Klasse `TowerType[]` sind die Turmtypen. `waveCounter` zählt die Wellen der Monster. Diese Zahl wird dann im Zusammenhang mit der `blablah` Klasse verwendet. `int guisize` dient zum Festlegen für die Größe der Auswahlfläche der Türme und `int fieldsizex` und `int fieldsizex` ist die Größe der Felder auf der Map. `int gobls` sind die Gobls in einer Welle. `uint moneyPerSecond` gibt die Geldanzahl an, welche man pro Sekunde bekommt. `bool isRunning` sagt aus ob das Programm gerade läuft und wird standartmäßig auf `true` gesetzt. `Image endScreen` ist das Bild, welches am Ende angezeigt wird.

Der Konstruktor des Fensters „Game“ bekommt die werte `int modenr, int goblsnr, string playername, Map map` und `uint mps` übergeben. `modenr` legt den Modi fest, `goblsnr` die

gobls pro Welle, `playername` den Spielernamen, `map` übergibt die Map welche zuvor von dem Optionenfenster erstellt wurde.

Jetzt werden die Komponenten aus dem Designer geladen.
Die Größe des Fensters wird nun festgelegt und die Bilder in den Buttons auf der linken Seite werden mittig zentriert. Daraufhin wird der Spielername und die Anzahl der Leben des Spielers in den Konstruktor von `Player` übergeben. Nun wird mit dem Attribut `modenr` die Nummer der Modus herausgefunden und dementsprechend Geld verteilt.

Jetzt werden die 5 verschiedenen Turmtypen erstellt und ihnen werden zudem die einzelnen Bilder und Werte zugewiesen. Das Gleiche passiert auch mit den 3 Monstertypen. Nun wird ein neuer Path erstellt, die Map `this.map` bekommt den übergebenen Wert von `map` zugewiesen. Daraufhin wird dem Objekt `player` der Name übergebeny der Timer wird `timer1` enabled und das Attribut `moneyPerSecond` bekommt den Wert von `mps` zugewiesen.

Die Methode `OnPaint` macht nichts anderes, als das Fenster neu zu Zeichnen.

Die folgenden 5 Methoden sind für die Auswahl der Türme zuständig und werden ausgeführt, wenn man auf das entsprechende Bild drückt. Sie sind vom Prinzip alle gleich aufgebaut. Zunächst stellen sie fest ob das Geld von dem Spieler dem des Preises des Turmes entspricht. Daraufhin wird aus dem Objekt `map` die `SetSelectedTower` Methode aufgerufen und es wird der Turm-Typ, auf den der User geklickt hat der Methode übergeben. Jetzt wird der Preis des ausgewählten Turmes vom Geld abgezogen.
`OnMouseDown` wird in `Game.cs` aufgerufen, wenn auf das Fenster geklickt wird. Daraufhin wird die Methode `OnClick` von dem Objekt `map` aufgerufen. Es werden die `X`-, `Y`-Koordinaten und der Button an der Maus, welcher gedrückt wurde übergeben.
`timer1_Tick_1` startet damit, dass die aktuelle Mausposition dem Attribut `Point mousepos` übergeben wird. Wenn das Programm läuft wird die Map mit der Methode `Update` aus dem Objekt `map` geupdatet wird. Wenn die Leben des Spielers auf `0` fallen, wird das Game-Over Fenster angezeigt und es wird ein Highscore Fenster. Danach wird überprüft, ob eine Welle vorüber ist und falls dies der Fall ist, kreiert es daraufhin eine neue.

Wenn nun die Maus auf einer der 5 Knöpfe zeigt, werden die Labels `labelAttackSpeed` `labelDamage` `labelRange` `labelCost` und die Buttons `buttonUpgrade` `buttonSell` angezeigt und die entsprechenden Werte der Türme eingeblendet.
Falls dies nicht der Fall ist werden diese ausgeblendet. Am Ende der Methode wird das Fenster refresht.

Die Methode `MonTicker` wird jede Sekunde ausgeführt. Sie rechnet dem Spieler jede Sekunde einen Spielgeldwert in der Höhe von `moneyPerSecond` auf sein Spielgeldguthaben.

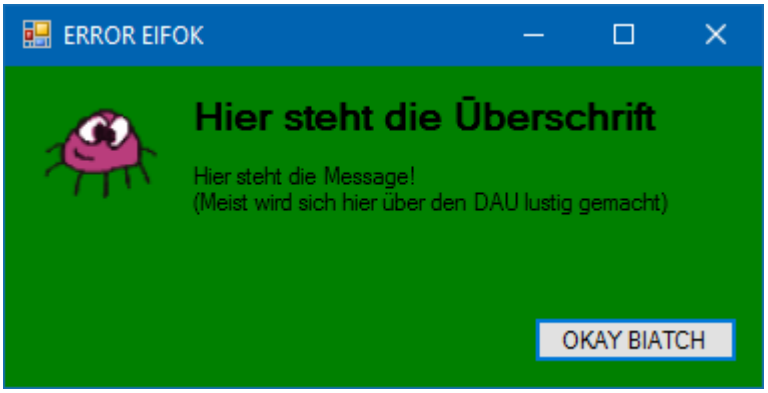
Daraufhin wird das Guthaben im Label `lab_money` zusammen mit den Leben des Spielers im Label `lab_score` angezeigt. Nun wird jeder Preis von jedem Turm auf Level 1 mit dem totalen Geld des Spielers verglichen und dementsprechend wird die Hintergrundfarbe der Buttons von den Türmen gefärbt. Rot wenn das Geld des Spielers kleiner ist als der Wert des Turmes und grün wenn das Geld dem Preis entspricht und er den Turm kaufen kann.

Die Methode `Sell` wird aufgerufen, wenn der Spieler auf den Button „Sell“ drückt. Der Spieler bekommt die Hälfte des Preises erstattet und die Methode `RemoveSelectedTower` wird aus dem Objekt `map` aufgerufen.

Die Methode `Upgrade` wird aufgerufen, wenn auf den Button „Upgrade“ gedrückt wird. Erst wird der Preis des Turmes ein Level höher verglichen und wenn der Turm kaufbar ist wird dem Spieler Geld abgezogen und er wird hochgelevelt.

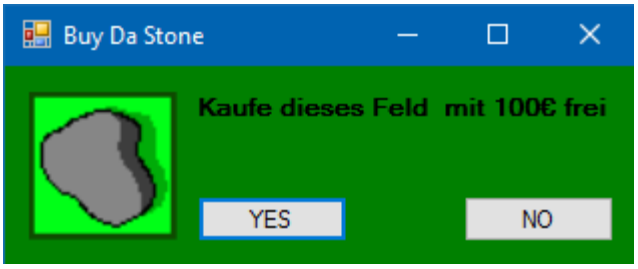
In der Methode `createWave` werden die einzelnen Wellen erstellt. Pro Welle wird ein Entsprechendes Muster ausgeführt. Die Wellen von 1 bis 10 sind festgelegt. Nachdem diese Wellenform ausgeführt wurde wiederholt sich dieses Szenario immer wieder.

5)
MSGBox Fenster



Dieses Fenster dient zur Anzeige der Code- oder der Input-Error Message Fenster.
Im Konstruktor werden dem Objekt Titel und Textinhalt weitergegeben. Die Methode `okay_Click()` beendet dieses Fenster.

6)
YorNO Fenster

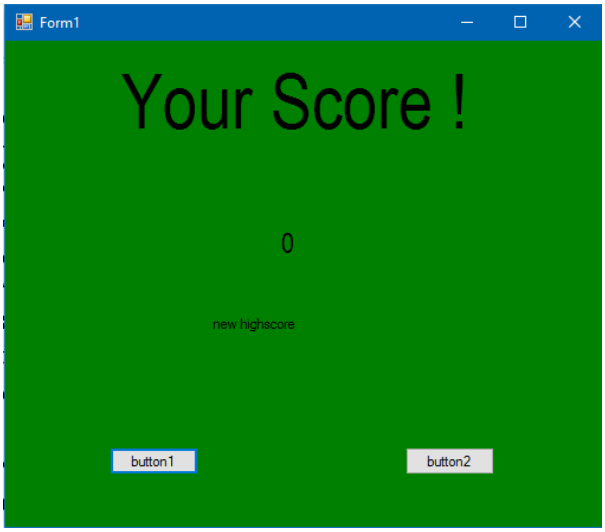


Dieses Fenster dient zum freischalten eines Feldes, welches von einem Stein besetzt wird. Bei dem klicken auf „Yes“ wird die Membervariable `bool yorn` auf `true` gesetzt und das Fenster geschlossen. Bei „No“ wird dieses auf `false` gesetzt und auch hier wird das Programm dann geschlossen. `yorn` kann dann später abgerufen werden, um zu prüfen

7)
HighPopUp Fenster

Das Fenster HigPopUp wird am Ende eines Spiels aufgerufen. Sie dient zur Ausgabe des finalen Scores und der Speicherung des Scores im Falle eines neuen Highscores in eine separate Textdatei. HighPopUp besitzt folgende Attribute:

```
private double score { get; set; }
private string name { get; set; }
private bool k { get; set; }
private SortedList<double, string> pl_list = new SortedList<double, string>();
private bool b { get; set; }
private int index { get; set; }
```



Den Variablen `score` und `name` werden der aktuelle Spielername und dessen Score zugewiesen. Die Variable k dient zur Freigabe des Speicherbuttons, nur wenn `k true` ist, hat der Button eine Funktion. `Bool b` wird `true` gesetzt, wenn der Spieler bereits in der Highscoreliste vorhanden ist. Dies dient zur Prüfung, ob es ein neuer Highscore ist und wie er gespeichert wird. Die Variable `Index` wird später der Index der aktuellen, gegebenenfalls vorhandenen Instanz der `pl_list` zugewiesen. Das Attribut `pl_list` ist ein Container der Sorte `SortedList`. In diesem Container gehört zu einer Instanz immer ein `<Tkey,Tvalue>` Paar mit zwei Werten. In diesem Fall ist der `Tkey` Wert vom Type `double` und der `Tvalue` Wert vom Type `string`. Der Zusatz `Sorted` vor `List` bedeutet, dass alle Instanzen in diesem Container nach der Größe der jeweiligen `Tkey` Werte geordnet werden. Dies ist sehr wichtig für die Ermittlung der Reihenfolge im Highscore-Ranking.

Die Klasse enthält einen Konstruktor, der den aktuellen Spielername und dessen Score übergeben bekommt. Im Konstruktor wird durch ein paar Verzweigungen ermittelt, ob es einen neuen Higscore gibt und Folge dessen durch den Button „Speichern“ gespeichert werden kann und die Highscoreliste ausgegeben wird. Eine entsprechende Textausgabe „new highscore“ oder „try again“ wird außerdem ausgegeben.

Die Klasse enthält des Weiteren 5 Methoden:

```
public bool search_pl()
```

In dieser Methode wird im Container nach einer Instanz mit dem `Tvalue` des Namens des aktuellen Spielers gesucht. Wird der Name dort gefunden, wird `true` zurückgegeben gibt es schon einen alten Highscore zu diesem Spieler. Wenn der Name nicht gefunden wurde wird der Wert `false` zurückgegeben.

```
public void readfile()
```

In der Methode `readfile` wird die Datei "`high.scores`" mit der Funktion: `System.IO.StreamReader file = new System.IO.StreamReader("high.scores",`

`true)` geöffnet. Danach wird die Datei Zeile für Zeile ausgelesen und die Highscore-Ddaten in den Container `pl_list` geschrieben.

```
public void writefile()
```

Die Methode `writefile` ist analog zur Methode `readfile`. Durch die Funktion `System.IO.StreamWriter file = new System.IO.StreamWriter("high.scores", false)` wird die Dartei mit dem Namen „high.scores“ geöffnet. Falls es sie noch nicht gibt, wird die Datei neu erstellt. Wenn es sie gibt, wird sie überschrieben. Nun wird der Container Zeilenweise in die Datei eingeschrieben.

```
private void button_speichern_Click_1
```

Diese Methode wird aufgerufen, wenn der Button „Speichen“ geklickt wurde. Die Methode ruft die Methode `writefile` auf um den Highscore zu speichern auf und anschließend die Fenster `Highscorescreen` um die Rangliste auszugeben. Dies funktioniert jedoch nur, wenn `k` vorher auf `true` gesetzt wurde. Sprich: wenn es einen neuen Highscore zum Speichern gibt.

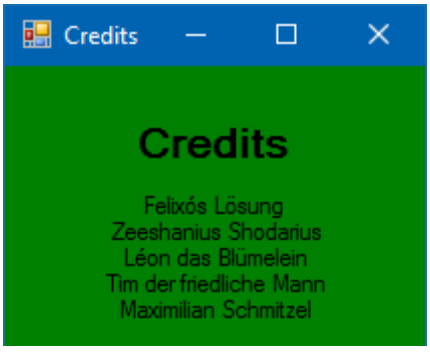
```
private void button_menu_Click
```

Diese Methode wird aufgerufen, wenn der Button „menu“ geklickt wurde. In der Methode wird das Fenster geschlossen. Man gelangt zurück zum Menü.

8)
Highscorescreen Fenster

Die Klasse Highscorescreen wird nach dem speichern in der HighPopUp Klasse aufgerufen und dient zur Ausgabe des Rankings. Der Konstruktor bekommt die SortedList „pl_list“ übergeben, mit den Spielern und dessen Scores. Der Container wird nun in entsprechender Form ausgegeben.

9)
Credits



Hier werden alle Entwickler bei ihrem Künstlernamen erwähnt.

Klassen

1)
Position Klasse

Ein Objekt der Klasse `Position` beschreibt eine Position auf der Map.

```
public float x { get; set; }
public float y { get; set; }
```

Die Klasse besitzt zwei Membervariablen: X- und Y-Koordinate. Sie besitzen beide public Get- und Set-Methoden. Die Klasse besitzt zwei Konstruktoren.

```
public Position(Position p)
public Position(Position p)
```

Einen bei dem zwei Werte für x und y übergeben werden mit den Standartwerden Null und einen bei dem die Werte aus einem bestehenden Objekt kopiert werden. Position besitzt eine Methode:

```
public float Distance(Position p)
```

Die Methode `Distance` berechnet den Abstand zwischen der aktuellen Position und einer übergebenen mithilfe des Satz des Pythagoras.

2)
MonsterType Klasse

Ein Objekt der `MonsterType` Klasse steht für eine Art von Monstern, zum Beispiel die schwachen blauen oder die stärkeren orangen. Sie beinhaltet alle Eigenschaften die zwischen allen Monstern derselben Art geteilt werden, das wären: Textur, Geschwindigkeit und maximale Gesundheit. Hierzu wurden diese Membervariablen erstellt:

```
public Image texture { get; private set; }
public float speed { get; private set; }    //in fields per second
public float health { get; private set; }
```

Alle Membervariablen besitzen eine public Get-Methode und private Set-Methoden, da ihnen nur einmal im Konstruktor ein Wert zugewiesen wird. Der Konstruktor macht nichts außer die Variablen zu initialisieren, dafür bekommt er für jede Variable einen Wert übergeben.

```
public MonsterType(Image texture, float speed, float health)
```

3)
Monster

Ein Objekt der Monster Klasse steht für ein aktives Monster auf dem Feld. Es besitzt die vier Attribute:

```
public MonsterType type { get; private set; }
public float health { get; set; }
public Position pos { get; set; }
public float pathPos {get; set; }
```

`type` beschreibt die Art des Monsters, `health` die aktuelle Gesundheit, `pos` ist die absolute Position auf der Map und beschreibt die Position am Weg entlang, zum Beispiel bei einem Weg mit der Länge 9 ist 0.0f der Anfang, 4.5f die Hälfte und 9.0f das Ziel. `health` alle Membervariablen besitzen public Get- und Set-Methoden außer `type`, da dies nur im Konstruktor initialisiert wird und dann konstant bleibt.

```
public Monster(MonsterType t, Position pos)
```

Der Konstruktor bekommt für type und pos einen Wert übergeben. Health wir auf die maximale Gesundheit gesetzt, die in type gespeichert ist und posPointer wie auf 0.0f gesetzt, da das Monster am Anfang des Weges spawnt.

4)
TowerType Klasse

Ein Objekt der Klasse TowerType beschreibt eine Turmart. Jedes level eines Turms hat seine eigene Turmart mit neuen Eigenschaften. 5 Turmarten * 5 Level macht 25 verschiedene Turmarten im Spiel. `TowerType` besitz folgende Attribute:

```
public Image texture { get; private set; }
public Image pTexture { get; private set; }
public float pSpeed { get; private set; }
public float pSize { get; private set; }
public uint cost { get; private set; }
public float damage { get; private set; }
public float radius { get; private set; }
public TowerType nextLevel { get; private set; }
public float shootSpeed { get; private set; }
```

`texture` beschreibt die Textur des Turms, `pTexture` die Textur des Projektils, das der Turm schießt. `pSpeed` ist die Geschwindigkeit, `pSize` die Größe des Projektils. `cost` ist der Preis des Turms, `damage` ist der Schaden, den der Turm macht, `radius` ist der Schussradius des Turms, `nextLevel` ist eine Referenz zu dem nächstes Level oder null wenn es kein nächstes Level gibt. `shootSpeed` ist die Schussgeschwindigkeit, mit der der Turm schießen kann. Alle Attribute haben eine public Get-Methode und eine private Set-Methode, da sie nur einmal im Konstruktor einen Wert zugewiesen bekommen, außer `nextLevel`, weil diese Variable nur in der Klasse verwendet wird. `TowerType` besitzt einen Konstruktor, der für jedes Attribut einen Wert übergeben bekommt.

```
public TowerType(Image image, Image pTexture, float damage, float radius, float
shootSpeed, float pSpeed, float pSize, uint cost, TowerType nextLevel = null)
```

Der Wert für `nextLevel` ist standardmäßig auf null, damit man nur einen Wert übergeben muss, wenn es auch wirklich ein nächstes Level gibt. Die Klasse `TowerType` enthält eine Methode `isMaxLevel` die einen bool zurückgibt, true wenn es kein nächstes Level gibt.

```
public bool isMaxLevel()
```

5)
Projectile Klasse

Die Klasse `Projectile` stellt ein Geschoss auf dem Spielfeld da, wie zum Beispiel eine Rakete.

```
public Position pos { get; set; }
public Monster target { get; set; }
public TowerType tower { get; set; }
private float angle;
```

Ein Objekt der Klasse `Projectile` besitzt eine Position auf dem Spielfeld namens `pos`, es besitzt ein Ziel zu dem es fliegt namens `target`, einen Turm von dem es abgeschossen wurde namens `tower` und einen Winkel in welche Richtung das Geschoss zeigt namens `angle`. P besitzt einen Konstruktor bei dem für jede Membervariable in Wert übergeben wird.

```
public Projectile(TowerType tower, Position pos, Monster target, float angle)
```

Die Klasse besitzt zwei Methoden:

```
public Image getImage()
public bool move(float time)
```

Die Methode `getImage()` rotiert die Textur des Projektils, welche in `tower` gespeichert ist, mit der Methode `RotateTransform(angle)` der Klasse Graphics und gibt dieses Bild anschließend zurück.

Die Methode `move(float time)` kriegt die vergangene Zeit übergeben, mit dieser und der Geschwindigkeit in `tower` wird die zurückgelegte Distanz berechnet. Dann wird der Abstand zwischen dem Zielmonster und dem Projektil mithilfe des Satz des Pythagoras berechnet. Wenn die Distanz kleiner als 0,3 ist, heißt das, dass das Monster getroffen wird. Dann wird dem Monster Gesundheit in Höhe des Schadens in `tower` abgezogen und die Methode gibt `true` zurück. Wenn das Monster noch nicht getroffen ist wird das Projektil in Richtung des Monsters bewegt und der Winkel so erneuert, dass das Projektil auf das Monster zeigt, die Methode gibt dann `false` zurück.

6)
Tower Klasse

Ein Objekt der Klasse `Tower` steht für einen aktiven Turm auf der Map. Es besitz die Attribute:

```
public TowerType type { get; set; }
public Position pos { get; set; }
private float shootTime;
private float angle;
```

Die Membervariable `type` ist der Typ des Turms, `pos` ist die Position des Turms auf der Map, `shootTime` ist die Zeit seit dem letzten Schuss und `angle` ist die aktuelle Rotation des Turms. `type` und `pos` haben public Get-und Set-Methoden, `shootTime` und `angle` sind private Variablen. Die Klasse `Tower` besitzt einen Konstruktor:

```
public Tower(TowerType t, Position pos) {
```

Werte für `type` und `pos` werden übergeben, `shootTime` und `angle` sind standartmäßig auf null gesetzt, da der Turm nach oben guckt beim Spawnen und noch keinen Schuss abgegeben hat. `Tower` besitzt zwei Methoden:

```
public Image GetImage()
public Projectile Shoot(List<Monster> allMonster, float time)
```

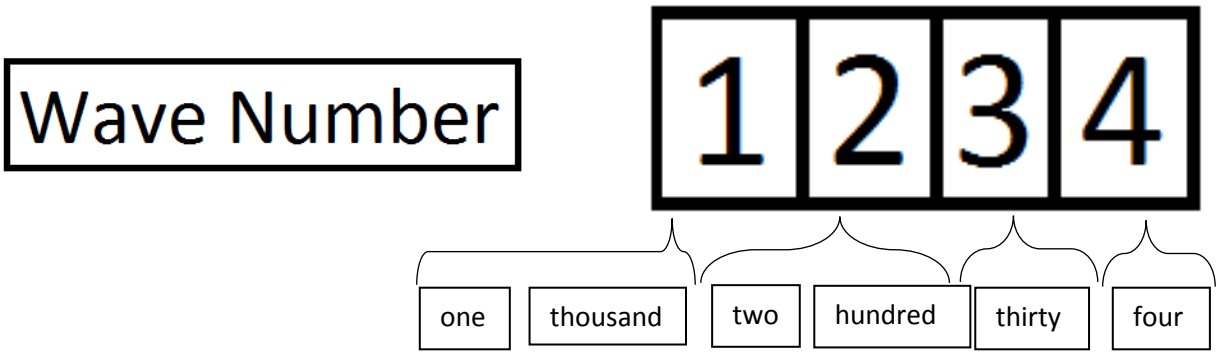
`GetImage` gibt die rotierte Textur des Turmes zurück;
Die Methode `Shoot` gibt ein neues Projektil zurück, das der Turm schießt, dafür bekommt sie alle Monster, die sich auf der Map befinden, und die vergangene Zeit übergeben. Die Methode überprüft welches Monster am dichtesten am Turm ist und ob es in Schussreichweite ist, dafür wird der Abstand zwischen jedem Monster aus der Liste `allMonster` berechnet und dabei das Monster mit dem kleinsten Abstand zwischengespeichert. Wenn es ein Monster in Reichweite gibt wird der Winkel berechnet, sodass der Turm zu diesem Monster zeigt, dann wird überprüft ob der Turm schießen kann, hierfür wird `shootTime` mit `shootSpeed` in `type` verglichen. Wenn der Turm nicht schießen kann gibt die Methode `null` zurück, sonst wird ein neues Objekt der Klasse `Projectile` erstellt mit der Position, dem Typ und dem Winkel des Turmes und mit dem Monster, auf das der Turm zielt.

7)
Sound Struktur

In dieser Struktur sind folgender Membervariablen
```
public SoundPlayer sound;
public int time;
```
In dem Konstruktor der Struktur werden die Attribute der Struktur eingestellt.

8)
Blablah Klasse

Ein Objekt der `blablah` Klasse stellt die Soundengine dar, welche zur Sprach-Ausgabe der Wellennummern gedacht ist. Inspiriert wurde diese Klasse durch die Funktionsweise eines Navigationssystems. Einzelne Wort oder Satzfetzen werden einzeln aufgenommen und können somit später für mehrere Worte oder Sätze verwendet werden. So muss man nicht jeden einzelnen Satz einzeln aufnehmen und in das Programm einbinden. Dies spart und Speicher.
Es wurde folgendes aufgenommen: Eine Testaufnahme, Hintergrundmusik, 4 Aufnahmen für 4 verschiedene Modi, die aber nicht eingebaut wurden, die Ansage „Wave number…“ um die Welle anzusagen, die Zahlen 1 bis 9 und 11, 12, 13, 15, die Zahlen 1-9 betont, die Zahlen 10-90, 100 und 1000, zudem noch „teen“.
Die komplette Soundengine läuft in dem Gamefenster in einem parallelen Thread.



Die Klasse besitzt folgende Attribute

```
private volatile bool isSoundThreadActive;
private volatile List<Sound> soundQueue;
private Thread soundThread;
private static string pathPrefix = "../../resources/sounds/";
```

Die Variable `isSoundThreadActive` sagt aus, ob der Soundengine-Thread aktiv ist. In `List<Sound> soundQueue` werden vom Spielthread die abzuspielenden Sounds abgespeichert. Der Soundthread liest diese dann der Reihe nach aus und löscht sie danach. Daraufhin folgt der `soundThread` Thread, in dem die Soundengine laufen wird. `pathPrefix` ist einfach nur ein Kürzel für den Dateipfad in dem die Sounds gespeichert sind.

Im Konstruktor wird `isSoundThreadActive` auf `true` gesetzt, `soundQueue` und `soundThread`, werden einen Wert zugeordnet und der Thread `soundThread` wird gestartet. Mit den Methoden `test23()noob23()normal23()ehm23()hackme23()wn23()` werden nur einzelne aufnahmen ausgegeben. Mit `playmusic23()` wird die Hintergrundmusik gestartet und mit `stopmusic23()` wird die Musik gestoppt.

In der Methode `tellme23(int n4, int n3, int n2, int n1)` werden die Zahlen per Sprachausgabe als eine Zahl mithilfe der Methoden `tellme23(int n3, int n2, int n1)` `tellme23(int n2, int n1)` und `tellme23(int n)` zusammengesetzt und ausgegeben. Dort werden folgende Werte übergeben:
`int n4` stellen die Tausender dar, `int n3` die Hunderter, `int n2` stellt die Zehner dar und `int n1` sind die Einser. Wenn `n4` den Wert Null hat, leitet das Programm die Werte `n3`, `n2` und `n1` in die Methode `tellme23(int n3, int n2, int n1)` weiter und setzt hier die Zahl die Zahlen per Sprachausgabe als eine Zahl mithilfe der Methoden `tellme23(int n2, int n1)` und `tellme23(int n)` zusammen und gibt diese mit den Aufnahmen aus.

Wenn `n3` Null, ist werden `n2` und `n1` in die Methode `tellme23(int n2, int n1)` weitergeleitet und mithilfe der Methoden `tellme23(int n2, int n1)` und `tellme23(int n)` zusammengesetzt und ausgegeben. Wenn `n2` Null ist wird `n1` zu `tellme23(int n)` weitergeleitet. Die Methode `einserbet23(int n)`gibt die Einser betont aus.

`count23()` ist zum Testen der `tellme23` Methoden und gibt alle Zahlen aus. Mit `StopThread()` wird `isSoundThreadActive` auf `false` gesetzt und der Thread beendet.

9)
Map Klasse

Die Klasse `Map` ist die wichtigste Klasse im Spiel. Sie beinhaltet das Spielfeld und alle Monster, Türme und Projektile. Auch ist `Map` für das Anzeigen des Spielfelds zuständig sowie für die Sound Ausgabe. Sie beinhaltet einige Membervariablen, wir beginnen hier mit den grundlegenden:

```
private int size;
public int width { get; private set; }
private int fieldSize;
public int xOffset { get; private set; }
public int yOffset { get; private set; }
```

`size` ist die Anzahl Felder die die Map auf einer Achse besitzt, `width` ist die Breite und Höhe der Map auf dem Bildschirm in Pixel, `fieldSize` ist die Anzahl Pixel pro Feld, diese wird im Konstruktor berechnet, `xOffset` und `yOffset` sind die Position der linken oberen Ecke auf dem Bildschirm in Pixel. Wenn beide 0 wären, würde die Map oben links im Fenster angezeigt werden. Auf `width, xOffset und xOffset` kann auch von außen zugegriffen werden. Für das Spielfeld gibt es dann drei weitere Variablen:

```
private MapState[,] map
private List<Position> path;
private List<Direction> pathDirections;
```

`map` ist ein zweidimensionales Array des Enums `MapState`. Dieses Enum zeigt einfach an was sich an diesem Feld befindet, der Weg, ein Stein, ein Turm oder nur Grass.

```
public enum MapState
{
    PATH,
    GRASS,
    STONE,
    TOWER
}
```

`path` ist eine Liste von sortierten Koordinaten der einzelnen Weg-Felder. `pathDirections` ist eine Liste des Enums `Direction`. Das Enum beschreibt die Richtung in welcher der Weg fortgesetzt wird.

```
public enum Direction
{
    UP,
    DOWN,
    LEFT,
    RIGHT
}
```

`pathDirections` ist in der gleichen Reihenfolge sortiert wie `path`. Das Erste Objekt ist der Anfang, das letzte das Ziel. Hierbei zu beachten ist, dass `pathDirections` immer um eins kleiner ist als `path`, da das letzte Feld logischer Weise keine Richtung besitzt.

```
private List<Monster> wave;
private List<Projectile> projectiles;
private List<Tower> towers;
```

`wave`, `projectiles` und `towers` sind einfache Container die alle Monster, Türme und Projektile auf dem Spielfeld beinhalten. Sie sind nicht sortiert, die Objekte befinden sich in der Reihenfolge in der sie gespawnt werden. Für das Wellensystem beinhaltet die Klasse `Map` fünf Variablen und eine Konstante:

```
private List<MonsterType> spawningWave;
public int wavecounter { get; private set; }
public float strengthFactor { get; set; }
```

```
private float spawnTime;
private const float SPAWN_TIME_DISTANCE = 2.0f;
private const float STRENGTH_BASE = 1.1f;
```

`spawningWave` ist eine Liste mit den Monsterarten, die als nächstes spawnen werden. Sie ist so sortiert, dass das erste Objekt das nächste Monster darstellt. `wavecounter` ist die aktuelle Wellenzahl und wird bei jeder neuen Welle um eins erhöht. Sie ist von außen aufrufbar damit man feststellen kann, bei welcher Welle sich der Spieler befindet. `strengthFactor` ist der Faktor, mit dem die Leben eines Monsters multipliziert werden wenn es spawnt und `STRENGTH_BASE` ist die Basis für die Formel, mit der die Stärke der Monster berechnet wird, $strenghtFactor * STRENGTH\_BASE^{wavecount}$. `spawnTime` ist die Zeit, die es noch braucht bis das nächste Monster spawnt. Wenn die Variable Null erreicht wird ein Monster gespawnt und die Variable auf den Wert von `SPAWN_TIME_DISTANCE` gesetzt, da diese Konstante den zeitlichen Abstand zweier Monster definiert.
Es gibt zwei Variablen für den ausgewählten Turm:

```
public Tower selectedTower { get; set; }
public bool selectedTowerPlaced { get; set; }
```

`selectedTower` ist der Ausgewählte Turm. Wenn der Turm auf der Map platziert ist, ist dies eine Referenz auf den Turm, wenn der Turm noch nicht platziert ist, enthält `selectedTower` die aktuelle Position auf dem Feld und den `TowerType` des gekauften Turmes. Wenn kein Turm ausgewählt ist, ist `selectedTower` `null`. `selectedTowerPlaced` ist ein `bool` der angibt ob der Tower auf der Map platziert ist oder nur die Vorschau bevor man ihn platziert.

Um das Spielfeld auf dem Bildschirm anzuzeigen sind weitere sechs Variablen und drei Konstanten erforderlich:

```
private Image mapTexture, grassTexture, pathTexture, stoneTexture, cakeTexture;
public float waveStringTime { get; set; }
public const float WAVE_STRING_TIME = 2.0f;
private const float WAVE_STRING_TIME_HIGHLIGHT = 0.5f;
private const float CAKE_RADIUS = 0.35f;
```

`grassTexture`, `pathTexture`, `stoneTexture` und `cakeTexture` sind die gespeicherten Bilder der in den Optionen geladenen Texturen. `cakeTexture` wird jedes Mal bearbeitet wenn der Spieler ein Leben verliert, damit der Kuchen langsam aufgegessen wird. `mapTexture` ist ein Bild der ganzen Map ohne Türme, Monstern und Projektilen. Statt immer mehrmals die Grass-Textur anzuzeigen wir die Map einmal auf dieses Bild gezeichnet, sodass dann immer `mapTexture` angezeigt werden kann. Dies erhöht deutlich die Laufzeitgeschwindigkeit. Immer wenn man eine neue Welle erreicht, wird es auf dem Bildschirm angezeigt. `waveStringTime` ist die Zeit, die die Schrift schon angezeigt wird, `WAVE_STRING_TIME` ist die gesamte Zeit in der die Schrift angezeigt wird und `WAVE_STRING_TIME_HIGHLIGHT` ist der Zeitpunkt an dem die Schrift komplett rot ist und dann wieder verblasst. Dieser Zeitpunkt muss kleiner sein als `WAVE_STRING_TIME`. `CAKE_RADIUS` ist der Radius des Kuchens, der benötigt wird um den Kuchen korrekt anzuzeigen.

```
private blablah soundengine;
```

`soundengine` ist ein Objekt der `blablah` Klasse und für die Ansage bei jeder neuen Welle zuständig.
Die Klasse `Map` besitzt einen Konstruktor:

```
public Map(Image grassTexture, Image pathTexture, Image stoneTexture,
Image cakeTexture, int size, List<Position> path, int width, int xOffset = 0,
 int yOffset = 0)
```

Es werden die Texturen übergeben, sowie die Größe der Map, die Breite in Pixeln und die Position auf dem Bildschirm. `wave`, `projectiles`, `towers`, `spawningWave`, `path`, und `pathDirections` werden mit dem Standardkonstruktor initialisiert.

```
wave = new List<Monster>(); usw..
```

`map` wird mit der übergebenen Mapgröße initialisiert:
`map = new MapState[size, size];`
`spawnTime` erhält den Standardwert 1,5, damit nach dem das Spiel gestartet wird eine kurze Pause ist. `wavecounter` wird auf 0 gesetzt und `selectedTowerPlaced` auf `false`. `strengthFactor` wird auf 1,0 gestezt und ändert sich nur wenn man in den Optionen etwas anderes einstellt. `soundengine` wird mit dem Standardkonstruktor initialisiert. `fieldSize` wird mit den übergebenen Werten berechnet: `fieldSize = width / size`
Jedes Feld von `map` auf `MapState.GRASS` gesetzt. Danach wird für jedes Objekt in `path` an der entsprechenden Position `map` auf `MapState.PATH` gesetzt. Anschließend wird die Liste `pathDirections` ausgefüllt indem jede Position in `path` mit der nächsten Position verglichen wird. Als letztes wird `mapTexture` initialisiert mit der Methode `CreateMapTexture()`.
Die Klasse `Map` hat 19 Methoden, davon sind 7 Methoden einfache Get-und Set-Methoden:

```
public void SetPath(List<Position> pa)

public void Rescale(int width, int xOffset = 0, int yOffset = 0)

public void SetStones()

public void Resize(int s)

public void SetSelectedTower(TowerType type)

public void RemoveSelectedTower()

public bool IsWaveComplete()
```

`SetPath(List<Position> pa)` setzt einen neuen Weg, dazu bekommt sie den Weg in Form einer Liste übergeben. Der alte Weg und die Richtungen werden gelöscht, danach wird `path` zu `pa` gesetzt und die Richtungen neu berechnet.

`Rescale(…)` skaliert die Map auf dem Bildschirm neu. Dazu wird die neue Breite in Pixeln und die Position auf dem Bildschirm übergeben. Die Position ist standardmäßig auf die linke obere Ecke gesetzt (0, 0). Anschließend werden die entsprechenden Attribute geändert und `fieldSize` neu berechnet. Danach muss noch die Maptextur mit der Methode `CreateMapTexture()` neu erstellt werden.

`SetStones()` setzt die Steine auf der Map zu den Standard Steinen im Rated Mode.

`Resize(int s)` ändert die Größe der Map, also die Anzahl Felder in X- und Y-Richtung. Hierzu wird die neue Größe übergeben. Die alte Map wird gespeichert und `map` dann neu initialisiert mit der neuen Größe: `map = new MapState[s, s];` Danach wird die alte Map auf die neue kopiert und wenn die Map größer geworden ist, werden die neuen Felder gleich `MapState.GRASS` gesetzt. Dann muss `fieldSize` neu berechnet werden und die Maptextur neu erstellt werden.

`SetSelectedTower(TowerType type)` wird aufgerufen wenn man einen neuen Turm kauft, dabei wird der gekaufte Turm übergeben. Die Methode setzt dann den ausgewählten Turm zum gekauften und setzt `selectedTowerPlaced` auf `false`.

`RemoveSelectedTower()` wird aufgerufen wenn ein Turm verkauft wird und setzt den ausgewählten Turm auf Null. Dafür wird die Stelle in `map`, wo sich der Turm befindet, auf `MapState.GRASS` gesetzt. Anschließend wird der Turm aus der Liste `towers` entfernt und `selectedTower` wird zu `null` gesetzt.

`IsWaveComplete()` gibt einen `bool` zurück ob die Welle zuende ist oder nicht. Hierfür wird einfach `true` zurückgegeben wenn keine Monster mehr auf dem Feld sind und es keine Monster mehr gibt, die gespawnt werden müssen.

Wenn auf die Map geklickt wird gibt es zwei Methoden:

```
public void OnClick(int mouseX, int mouseY, MouseButtons b)
```

```
public bool OnClickInOptions(int mouseX, int mouseY, MouseButtons b)
```

`OnClick(…)` wird während dem Spiel aufgerufen und `OnClickInOptions(…)` wenn man in den Optionen ist. Beide Methoden sind sehr ähnlich. Sie bekommen die Mausposition auf dem Bildschirm übergeben, sowie die Maustaste die gedrückt wurde. Als erstes wird überprüft ob die Maus wirklich in die Map geklickt hat, wenn nicht wird die Methode sofort beendet. In der `OnClick(…)` Methode wird noch überprüft ob der ausgewählte Turm platziert ist und wenn ja wird er deselektiert. Wenn die Maus in der Map ist wird das Feld berechnet auf dem sie Liegt. Hier unterscheiden sich die Methoden jetzt.

`OnClick(…)` prüft on man einen unplatzierten Turm ausgewählt hat und auf Grass geklickt hat, dann wird der Turm an diese Stelle platziert, indem der Wert in `map` auf `MapState.TOWER` gesetzt wird und der Turm der Liste `towers` hinzugefügt wird. Dann wird der Turm deselektiert. Wenn das aber nicht der Fall war, wird überprüft ob man auf Stein geklickt hat, wenn ja wird das Fenster `YorNO` geöffnet. Wenn der Spieler dann auf JA geklickt hat wird der Stein in der Map gelöscht und die Maptextur neu erstellt. Wenn der Spieler auf einen Turm klickt und aktuell keinen ausgewählt hat, wird der angeklickte Turm zum Selektierten. Dafür wird der Turm in `towers` gesucht, der die Richtige Position hat, und dem Attribut `selectedTower` zugewiesen.

`OnClickInOptions(…)` prüft erst ob die linke oder die rechte Maustaste gedrückt wurde. Bei der Linken überprüft sie dann ob das Feld ein Weg ist und ob es das letzte Feld des Weges ist, wenn ja wird das Feld aus dem Weg gelöscht und die Maptetextur neu erstellt. Wenn der Spieler auf Grass geklickt hat und das Feld neben dem letzten Feld des Weges liegt, sofern ein Weg existiert, wird der Weg um dieses Feld erweitert. Wenn der Spieler die rechte Maustaste gedrückt hat und dabei auf Grass geklickt hat wird an dieser Stelle ein Stein gesetzt, hat er auf einen Stein gesetzt wird der Stein gelöscht. Danach wird die Maptextur neu erstellt.

Für die Texturen in Map gibt es zwei Methoden:

```
private void CreateMapTexture()
```

```
private void UpdateCake(Player player)
```

`CreateMapTexture()` erstellt die Maptextur neu. Dafür wird `mapTexture` zu einer neuen `Bitmap` gestezt mir der Größe der Map auf dem Bildschirm. Dann wird für jedes Feld im Array `map` die Grass- Pfad- oder Stein-Textur an die entsprechende Stelle auf `mapTexture` gezeichnet. Dies erfolgt mit der Methode `Graphics.DrawImage(…)`. Das `Graphics` Objekt hier wird von der der Maptexture erstellt `Graphics g = Graphics.FromImage(mapTexture)`.

Die Methode `UpdateCake(Player player)` aktualisiert den Kuchen am Ende des Weges. Sie bekommt den Spieler übergeben und errechent daraus erstmal den Winkel, dem der Kuchen

jetzt fehlt mit der Formel: $2\pi * \left(1 - \frac{\text{player.health}}{\text{player.maxHealth}}\right)$. Dann wird bei diesem Winkel die Textur mit der Farbe Transparent übermalt. Dafür wird erst ein `Graphics` Objekt aus `cakeTexture` erstellt und die Eigenschaft `g.CompositingMode` auf `System.Drawing.Drawing2D.CompositingMode.SourceCopy` gesetzt. Die zu übermalende Fläche wird in Dreiecke unterteilt und dann mit der Methode `Graphics.FillPolygon(…)` übermalt.

Um die Map dann tatsächlich auf dem Bildschirm anzuzeigen gibt es zwei verschiedene Methoden, je nachdem ob man im Spiel oder in den Optionen ist.

```
public void Render(Graphics g, Point cursor)
```

```
public void RenderInOptions(Graphics g)
```

Übergeben wird das `Graphics` Objekt aus der OnPaint(…) Methode des Fensters. `Render(Graphics g, Point cursor)` bekommt zusätzlich die Position des Cursors übergeben.

`Render(Graphics g, Point cursor)` zeichnet erstmal die Maptextur und die Kuchentextur mit der Methode `Graphics.DrawImage(…)`. Danach wird für jedes Monster, das `Image monster.type.texture` gezeichnet, und wenn das Monster schon mal Schaden bekommen hat, wird ein grünes Viereck für die Lebensanzeige gezeichnet mit der Methode `Graphics.FillRectangle(…)`.

Für jeden Turm wird ein `Image` gezeichnet, welches man aus der Methode `tower.GetImage()` bekommt. Für den ausgewählten Turm wird erst wenn er nicht platziert ist die Position aktualisiert mit `cursor`, dann wird die Textur gezeichnet und ein Kreis mit dem Radius des Turmes. Hierfür wird die Methode `Graphics.DrawEllipse(…)` verwendet. Der Kreis ist blau wenn der Turm platziert ist, grün wenn er platziert werden kann und rot wenn er nicht platziert werden kann. Anschließend wird für jedes Projektil die Textur aus `projectile.getImage()` gezeichnet. Wenn die Schrift bei einer neuen Welle gezeichnet werden soll wird zunächst die Farbe dafür berechnet. Dafür wird die Zeit `waveStringTime` und die Konstanten `WAVE_STRING_TIME` und `WAVE_STRING_TIME_HIGHLIGHT` verwendet. Diese berechnete Zahl wird dann als Alpha und Rot Wert der Farbe genommen. Der Text wird mit der Methode `Graphics.DrawString(…)` gezeichnet.

Die Methode `RenderInOptions(Graphics g)` wird in den Optionen verwendet und zeichnet die Maptextur und danach für jedes Feld des Weges die jeweilige Indexnummer. Wenn die Indexnummern mehrstellig werden, wird die Schriftgröße verringert.

Für den Spielmechanismus sind in `Map` drei Methoden:

```
public void MoveMonster(float time, Player player)
```

```
public void SpawnMonster(float time)
```

```
public void Update(float time, Player player)
```

Es wird die vergangene Zeit seit dem letzten Update übergeben und der aktuelle Spieler.

Die Methode `MoveMonster(float time, Player player)` geht durch jedes Objekt von `wave` und prüft ob das Monster gestorben ist, wenn ja wird es aus der Liste entfernt, der Spieler kriegt Geld und die Schleife geht weiter. Es wird die Distanz berechnet, die das Monster in der Zeit zurückgelegt hat. Dann wird `pathPos` des Monsters um diesen Wert erhöht und es wird die

absolute Position auf dem Spielfeld mithilfe von `pathDirections` berechnet, welche dann `pos` des Monsters zugewiesen wird. Am Ende wird getestet, ob das Monster am Ende des Weges ist und wenn dies zutrifft wird es aus der Liste entfernt, der Spieler verliert ein Leben und der Kuchen wird aktualisiert.

Die Methode `SpawnMonster(float time)` verringert `spawnTime` um die vergangene Zeit. Wenn `spawnTime` dann 0 erreicht, heißt das ein neues Monster kann gespawnt werden. Dazu wird ein neues Objekt der Klasse `Monster` der Liste `wave` hinzugefügt. Dieses Monster hat den `MonsterType` der als erstes in der Liste `spawningWave` steht. Anschließend wird das erste Element aus `spawningWave` entfernt. Die Leben des Monsters werden mit der oben genannten Formel: $monster.health \mathrel{*}= strenghtFactor * STRENGTH\_BASE^{wavecount}$. Als letztes wird dann `spawnTime` wieder auf `SPAWN_TIME_DISTANCE` gesetzt, damit das nächste Monster wieder im richtigen Zeitabstand spawnt.

In `Update(…)` wird als erstes die Methode `SpawnMonster(float time)` aufgerufen und danach die Methode `MoveMonster(float time, Player player)`. Danach wird jedes `Projectile` in `projectiles` bewegt mit der Methode `projectile.move(time)`. Wenn das Projektil auftrifft gibt die Methode `true` zurück, wodurch das Objekt aus der Liste entfernt wird. Dann wird für jeden `Tower` in `towers` die Methode `tower.Shoot(wave, time)` aufgerufen. Die Methode gibt ein neues `Projectile` zurück welches, sofern es existiert, zu `projectiles` hinzugefügt wird. Als letztes wird die Zeit `waveStringTime` um `time` erhöht.

`Map` besitzt noch eine letzte Methode:

```
public void StopSoundEngine()
```

Diese wird aufgerufen wenn das Spiel vorbei ist damit der zusätzliche Soundthread beendet wird.

10)

Player Klasse

Die Klasse `Player` enthält alle Daten die den Spieler betreffen:

```
public int health { get; set; }
public int maxHealth { get; set; }
public uint money { get; set; }
public string name { get;  set; }
public uint totalMoney { get; set; }
public int totalGoblsKilled { get; set; }
```

`health` ist die aktuelle Gesundheit des Spielers und `maxHealth` die Startgesundheit. `money` ist das aktuelle Geld des Spielers, `totalMoney` ist das gesamte Geld, das der Spieler während dem Spiel gesammelt hat. `name` ist der Name des Spielers, der für den Highscore benötigt wird und `totalGoblsKilled` ist die Anzahl an getöteten Monstern.

Die Klasse `Player` hat einen Konstruktor, bei dem der Name und die maximale Gesundheit übergeben werden. Das Startgeld beträgt 30 und die Gesundheit ist am Anfang gleich der maximalen Gesundheit.

Quellen:

https://de.wikipedia.org/wiki/Tower_Defense

https://msdn.microsoft.com/

https://de.wikipedia.org/wiki/Grafische_Benutzeroberfl%C3%A4che

credits

Felix

Zeeshan

Tim

León

Max