

Dokumentation:

YELLOW TALE

Benjamin Simikic, Emil Hopf, Matthias Wohland

Inhaltsverzeichnis

1. Allgemeiner Hinweis der Dokumentation

2. Aufgabenverteilung

I. YellowTale

1. Projektbeschreibung.....	7
1.1 Spielbeschreibung und Motivation	
2. Grafiken.....	9
2.1 Spielbeschreibung und Motivation	
2.2 Planung des Mapdesigns	
2.3 Umsetzung der Grafik	
2.4 Umsetzung und Implementierung der Maps	
3. Klassen (Klassendiagramm).....	14
3.1 Game1	
3.2 Abilities Manager	
3.2.1 AblityManager	
3.2.2 ShotEngine	
3.3 Animation Manager	
3.3.1 AnimationManager	
3.3.2 Animations	
3.4 Gegner Manager	
3.4.1 EnemyManager	
3.4.2 Enemy	
3.5 Kollision Manager	
3.5.1 BulletCollision	
3.6 Map	
3.6.1 Collision	
3.6.2 Layer	
3.6.3 Map_Changer	
3.7 Menu	
3.7.1 Button	
3.7.2 GameOver	
3.7.3 Menus	

3.8	NPC Manager
3.8.1	NPCManager
3.8.2	NPC
3.8.3	Textbox
3.9	Objekt Manager
3.9.1	ObjectManager
3.9.2	InteractiveObjects
3.9.3	PickUpItems
3.10	Partikel Manager
3.10.1	ParticleManager
3.10.2	Particle
3.11	Player
3.11.1	Movement
3.11.2	Player
3.12	Waffen Manager
3.12.1	WeaponManager
3.13	HUD
3.14	SaveLoad

II. MapEditor

1. Programmbeschreibung.....	107
2. Programmablauf.....	108
3. Klassen.....	110
3.1 MainWindows	
3.2 StaticMembers	
3.3 Draw	
3.3.1 DrawImage	
3.4 Layers	
3.4.1 Background	
3.4.2 Collision	
3.4.3 Layer1	
3.4.4 Layer2	
3.4.5 Layer3	
3.5 Map	
3.5.1 InputMap	
3.5.2 OutputMap	
3.6 Tiles	
3.6.1 TileMap	

III. Tiles aus TileSet

1. Programmbeschreibung.....	139
2. Programmablauf.....	140
3. Klassen.....	141
3.1 Form1	

1. Allgemeiner Hinweis der Dokumentation:

Alle Attribute wurden mit einem kleinen Anfangsbuchstabe geschrieben. Demnach wurden die Eigenschaften mit einem großen Anfangsbuchstabe geschrieben. In der Dokumentation wird nicht auf die Eigenschaften hingewiesen!

Zur leichteren Verständnis wird vorausgesetzt das man neben der Dokumentation auch das Projekt offen hat.

Ordner(Orange 2)

Klassen(Rot 1)

Attribute(Himmelblau)

Methoden(Himmelblau 3)

Schriftart Dokumentation: Bahnschrift

Schriftart Projekt: manaspc.ttf (<https://www.dafont.com/manaspace.font>)

Monogame 3.6 (<http://www.monogame.net/>)

2. Aufgabenverteilung:

Benjamin Simikic	Emil Hopf	Matthias Wohland
I. 1. Projektbeschreibung	I. 3.2 Abilities Manager	I. 3.1 Game1
I. 2. Grafiken	I. 3.3.1 AnimationManager	I. 3.3.2 Animations
	I. 3.4 Gegner Manager	I. 3.6 Map
	I. 3.5 Kollision Manager	I. 3.7 Menu
	I. 3.8.1 NPCManager I. 3.8.2 NPC I. 3.8.2 NPC : Tutorial I. 3.8.3 Textbox	I. 3.8.2 NPC : Shop
	I. 3.9 Objekt Manager	I. 3.11.1 Movement
	I. 3.10 Partikel Manager	I. 3.14 SaveLoad
	I. 3.11.2 Player	
	I. 3.12 Waffen Manager	
	I. 3.13 HUD	
	II. MapEditor	
	III. TilesAusTileSet	

I. YellowTale

1. Projektbeschreibung

1.1 Spielbeschreibung und Motivation

Bevor das Projekt überhaupt angefangen hat, haben wir in der Gruppe diskutiert was für eine Art von Projekt wir anfertigen sollen.

Wir haben uns schnell auf ein Spiel geeinigt, jedoch wussten wir nicht direkt welche Richtung dieses annehmen sollte. Nach einiger Diskussion haben wir uns schlussendlich auf ein sogenanntes Bullet-Hell-RPG geeinigt.

Dabei handelt es sich um ein klassisches Rollenspielkonzept, jedoch macht das Kampfsystem den entscheidenden Unterschied zum normalen RPG: Sowohl Spieler als auch Gegner (CPU) schießen Projektile, die ausgewichen und manipuliert werden können.

Der Spieler hat dabei durch 3 verschiedene Klassen, mehrere Möglichkeiten diese abzufeuern. Ebenfalls besitzt jede Klasse 3 verschiedene Fähigkeiten, welche das Spielerlebnis erweitern.

Bei der Auswahl der Klassen haben wir uns auf ein Menu-System entschieden, dass bei einem sogenannten NPC (Non-Player-Character) zur Verfügung steht.

Über dem NPC befinden sich 3 Köpfe, welche als Auswahlknöpfe der Klassen agieren.

Dieser NPC befindet sich im Haus in dem man sich direkt zu Spielbeginn befindet, so kann man als erstes entscheiden mit welcher Klasse man sich durchs Spiel begeben will.

Dieses Haus befindet sich wiederum im Zentrum der Karte („Stadt“), und man kann dieses nach jedem Durchlauf eines Bioms erreichen.

Jedes der drei Biome – Wald, Eis und Lava – besteht immer aus 3 Weltkarten (Standard-Maps) und 3 Dungeonkarten.

Die Weltkarten sind dabei der Prolog der dem Spieler die Umgebung und die Bedingungen der späteren Dungeonkarten zeigen soll.

Beispiel: Die Standardkarten des Lava-Bioms enthalten Lavaseen in der Gegner problemlos überleben können, der Spieler diese jedoch nicht betreten kann. Der Spieler kann allerdings trotzdem die Gegner in der Lava angreifen.

Die letzte Dungeonkarte jedes Bioms enthält einen Bossgegner, bei dessen Erlegung der Spieler mit Erfahrungspunkten und Talies (Währung) belohnt wird. Wenn der Spieler genug Erfahrung gesammelt hat, schaltet er weitere Fähigkeiten frei, die er im Kampf benutzen kann. Der Spieler kann mit Talies daraufhin im Klassen-Auswahlmenu seine Fähigkeiten verbessern, jede Fähigkeit besitzt 3 Stufen die verbessert werden können.

2. Grafiken

2.1 Planung des Grafikdesigns

Bereits als wir uns auf die Spielart entschieden haben, haben wir uns ebenfalls auf den Stil des Spiels geeinigt, eine Pixelbasierte Darstellung.

Die grafische Darstellung des Spiels erfolgt über 5 Layer/Schichten:

- 1. Background Layer

Auf diesem Layer wird die Grundform der Karte dargestellt, sprich die Basis der Karte.

Im Regelfall sind das Boden- und Wandtiles.

- 2. Layer 1

Dieser Layer beinhaltet Decals und weitere Maßnahmen die die Umgebung erweitern.

Alles was sich in diesem Layer befindet wird hinter dem Spieler gezeichnet.

z.B.: Ränder für Gras-/Schnee-/Schlammtexturen, Fackeln, Blut, Risse im Boden, etc...

- 3. Layer 2

Dieser Layer beinhaltet den Großteil der Objekte die sich auf der Karte befinden.

Alles was sich in diesem Layer befindet wird vor dem Spieler gezeichnet.

z.B.: Bäume, Büsche, Fässer, Häuser, Zäune, Fallen, Schilder, etc...

- 4. Layer 3

In diesem Layer befinden sich die Gegner und NPCs der Map.

Ohne dieses Layer, würden sich auf der Karte keine Gegner befinden und man hätte keine Möglichkeit mit NPCs zu interagieren.

- 5. Collision Layer

In diesem Layer wird die Kollision der Karte bestimmt.

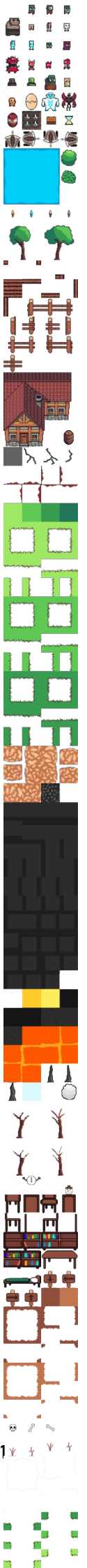
Dabei unterscheidet man von 7 verschiedenen Kollisionsarten.

1. (Rotes X): Charaktere und Geschosse können nicht durchdringen.

2. (Grünes X): Charaktere können nicht durchdringen, Geschosse dagegen schon.

3 – 7. (Nummeriert): Portale für die Fortbewegung zwischen den verschiedenen Maps. Ohne diese Portale kann man sich nicht zwischen Maps bewegen und ist statisch nur auf einer Map vertreten.

Charaktere (inklusive dem Spieler), Gegner und Projektile werden ZWISCHEN Layer 1 und Layer 2 gezeichnet.



Daher erscheint der Spieler zum Beispiel hinter Büschen die sich auf Layer 2 befinden und nicht davor, da die Büsche vor dem Spieler gezeichnet werden und sie keine Kollision besitzen.

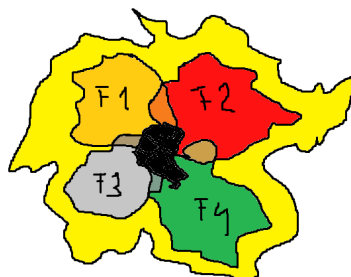
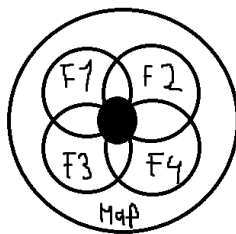
2.2 Planung des Mapdesigns

(Beachte dass es im Endprodukt nur 3 statt 4 Biome gibt)

Zuerst war geplant die gesamte Spielkarte in 5 Sektoren einzuteilen:

F1 – F4 sind die 4 verschiedenen Biome der Karte, in der Mitte befindet sich der Zugangspunkt zu jedem Biom, auch „Stadt“ genannt.

Entwicklungsphase der Karte:



Dabei ist die Karte programmtechnisch so nicht existent, der Spieler wird lediglich von Karte zu Karte transportiert.

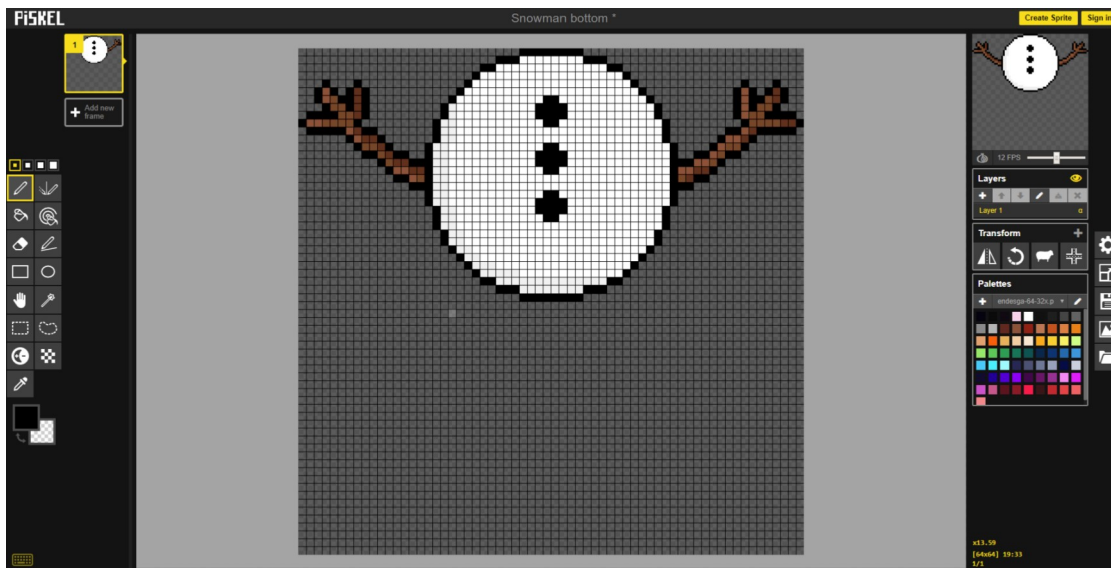
Die einzelnen Maps an sich bestehen aus 64x64 großen Tiles, die aus dem TileSet entzogen werden (Das TileSet ist Rechts am Rand zu sehen).

Das TileSet enthält alle Tiles mit denen die Maps dargestellt werden.

2.3 Umsetzung der Grafik

Alle Sprites und Tiles wurden mithilfe des Piskelapp-Editors auf Rastern erstellt.

(<http://www.piskelapp.com>)



Es wurde für die Kreierung der Grafiken ein bestimmtes Farbschema festgelegt, wir haben uns für die Endesga 32 Palette entschieden die genau 32 verschiedene Farbtöne enthält.



Um den „Pixelstil“ des Spiels zu erhalten wurden *die meisten* Grafiken auf einem Raster gezeichnet, welches halb so groß ist wie die Grafik selbst.

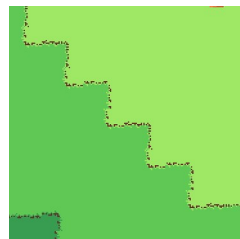
z.B.: Grafik der Größe 64x64 → gezeichnet auf Raster 32x32

Bei Decalgrafiken – wie zum Beispiel die Grasränder der Waldmaps

wurde lediglich die selbe Grafik in 4 verschiedenen Ausrichtungen verwendet.

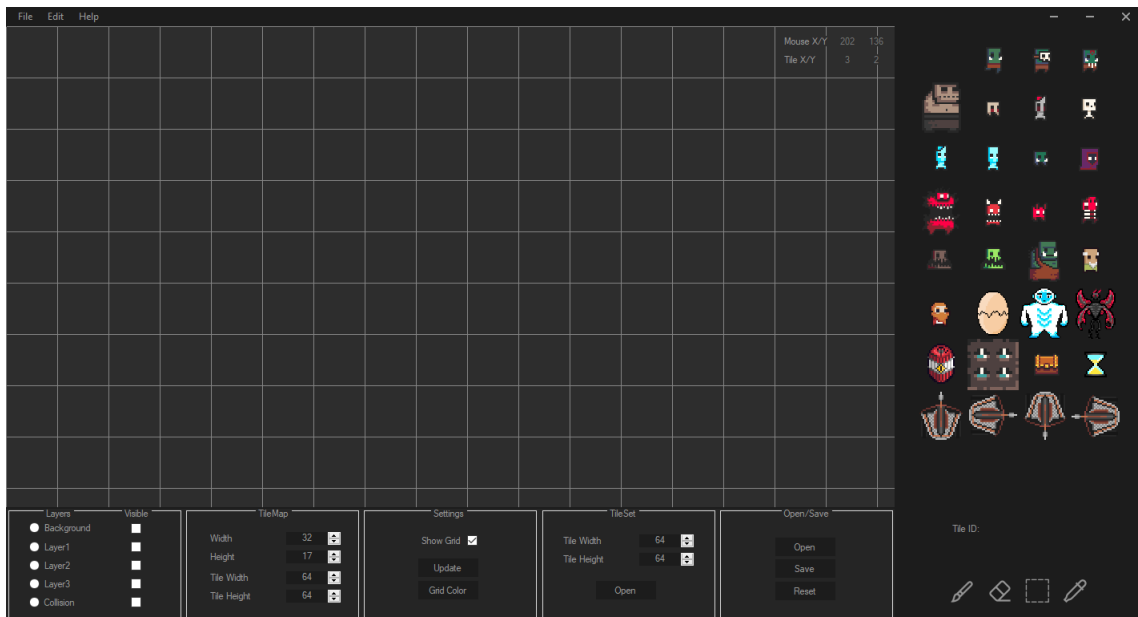
Objekte auf der Map die größer als 64x64 sind, werden in mehrere Tiles aufgeteilt.

Die Bäume sind zum Beispiel 128x128 groß, was eine Aufteilung in 4 verschiedene Tiles erfordert die jeweils 64x64 groß sind.



2.4 Umsetzung und Implementierung der Maps

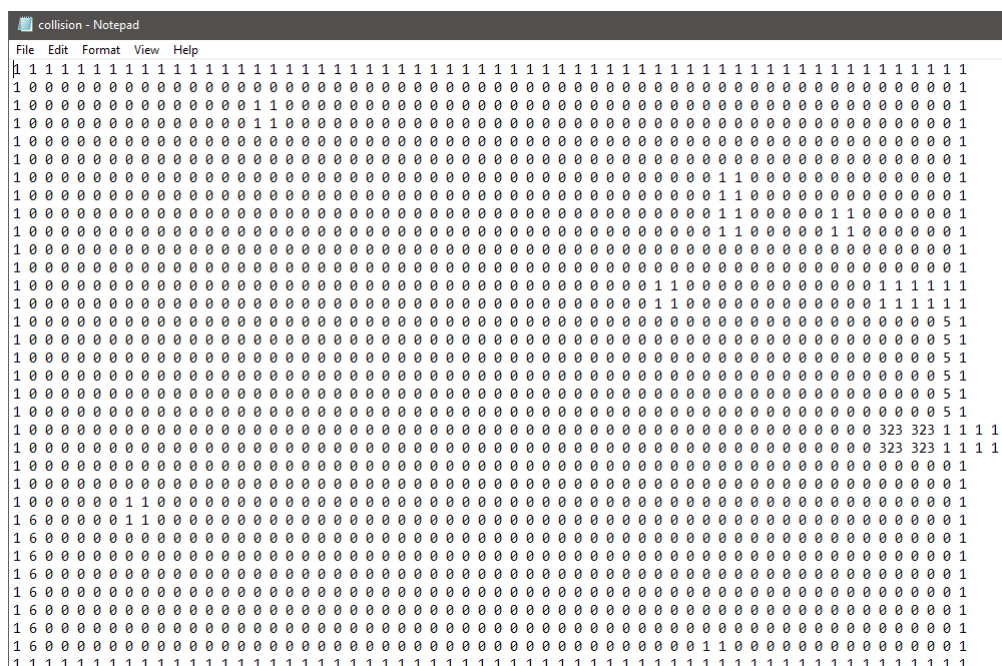
Die Implementierung der Maps erfolgte über den extra entwickelten Map Editor, der Dateien ausliest und diese neu beschriftet.



Nachdem man für jedes Layer eine Datei ausgewählt und ins Programm geladen hat, kann man mithilfe des Brush-Tools ein Tile aus dem TileSet nehmen und anfangen diese Layer für Layer zu platzieren.

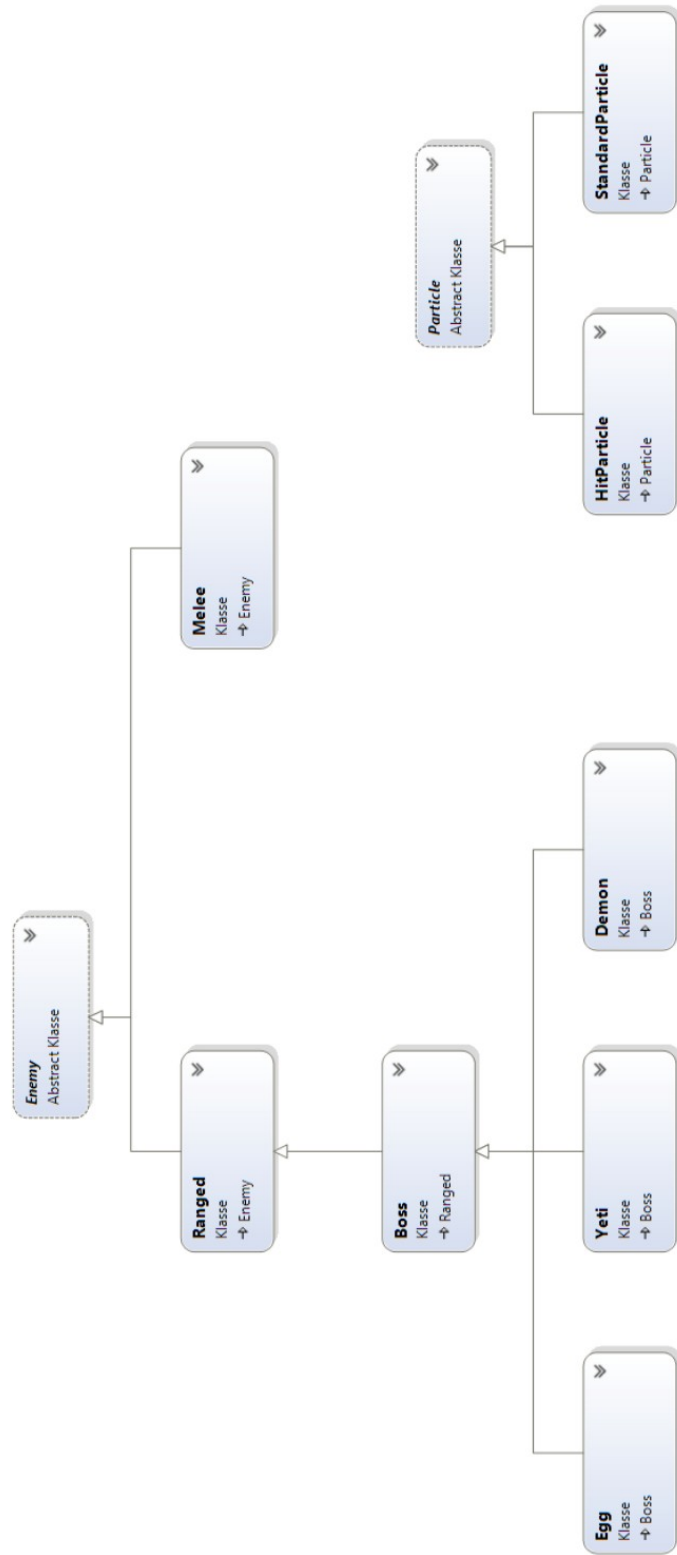
Da jedes Tile eine bestimmte ID hat, können sie eindeutig identifiziert werden. Bevor eine Datei beschrieben wird (wenn sie „leer“ ist), besteht der Inhalt aus einem Raster aus Nullen. Diese Nullen werden dann nach dem Speichern durch die IDs der Tiles ersetzt, sodass sich ein klares Muster widerspiegelt.

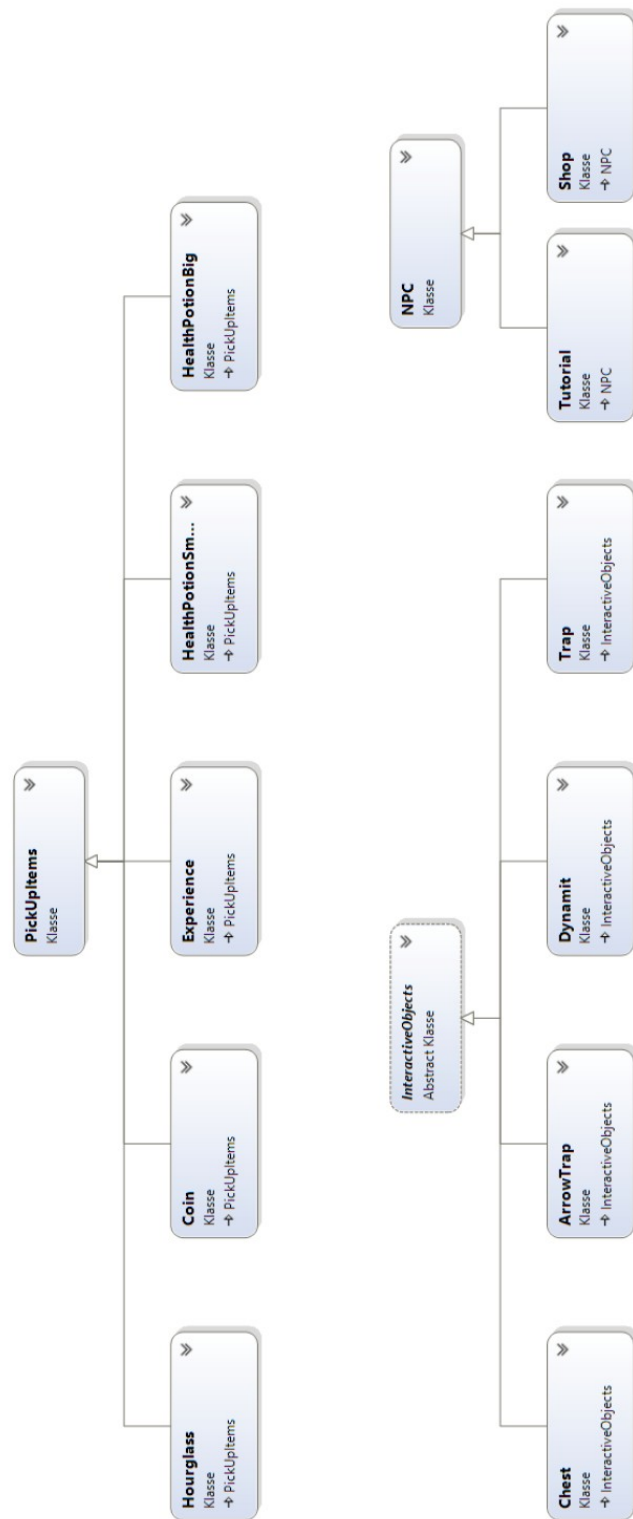
Beispiel einer Datei des Collision-Layers

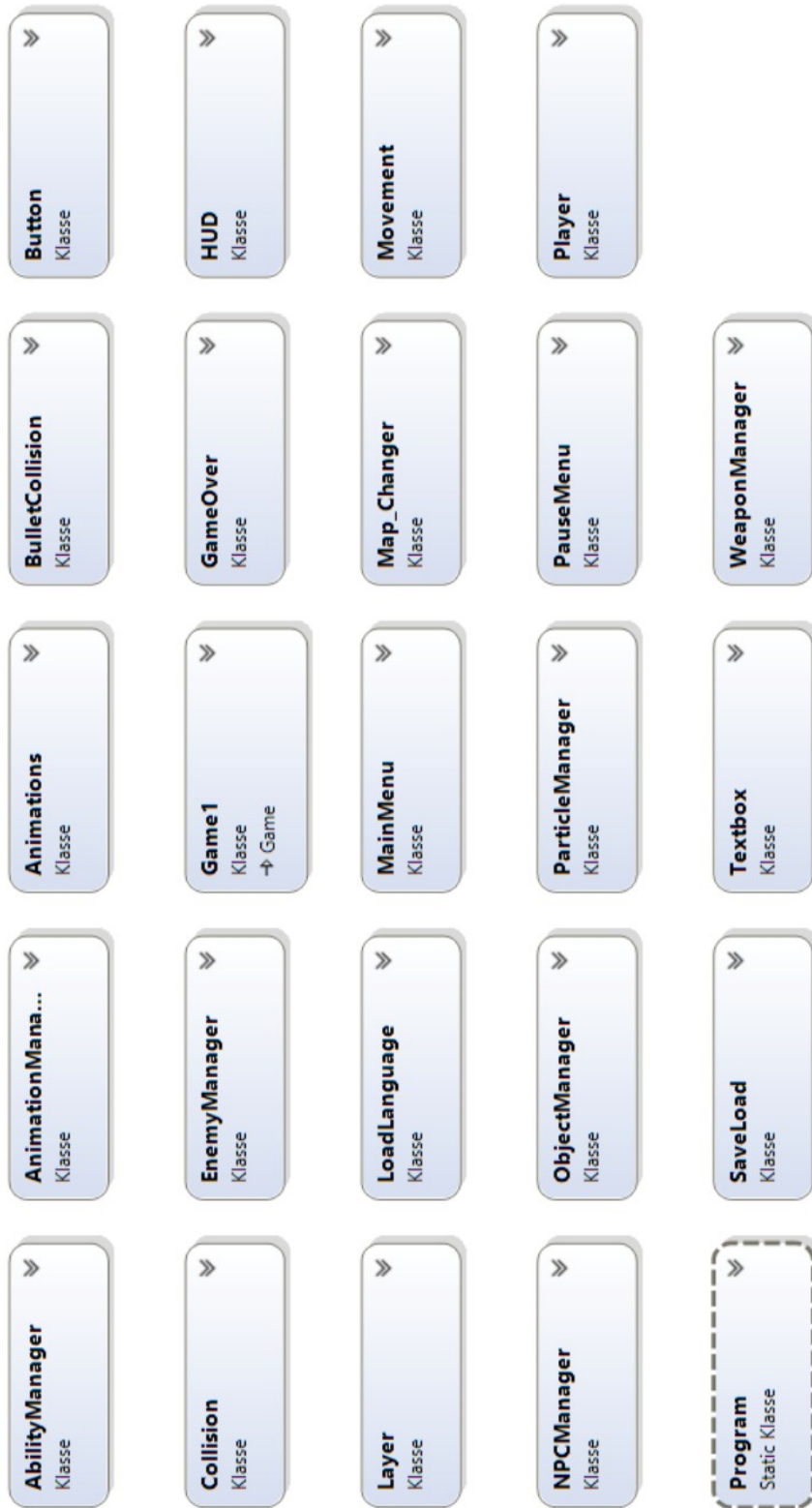


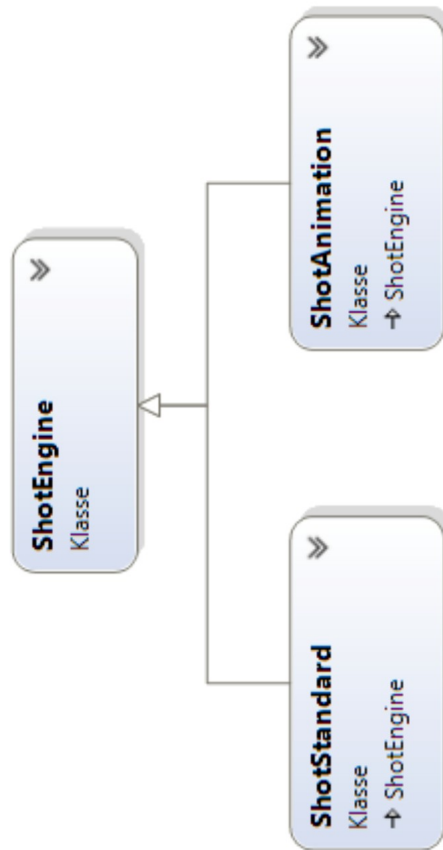
3.

Klassen









Klasse:
3.1 Game1

3.1 GAME1

Die Klasse **Game1** ist die Hauptsteuerklasse. Alle Manager Klassen werden in **Game1** aufgerufen.

Attribute

Game1 besitzt 43 Attribute. (Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- GraphicsDeviceManager	graphicsDeviseManager	Verwaltet Einstellungen vom Bildschirm und ähnlichem.
- SpriteBatch	spriteBatch	Wird für alle Zeichenaufgaben benötigt.
- MouseState	mouseState	Wird für die Position der Maus benutzt.
- MouseState	lastmouseState	Wird für bestimmte Eingaben der Maus gebraucht und um die letzte Mausektion zu überprüfen.
- KeyboardState	keyState	Durch dieses Objekt können Tasteneinschläfe überprüft werden.
- Vector2	scaledMausPosition	Da unser Spiel skaliert wird sobald sich die Auflösung ändert muss auch die Mausposition verändert werden.
- Matrix	bildschirmScaling	Die Skalierungsrate der Auflösung.
- Matrix	mausScaling	Die Skalierungsrate der Mausposition.
- Player	_player	_player verwaltet den Spieler selbst.
- EnemyManager	enemyManager	Der enemyManager verwaltet die Gegner.
- ObjectManager	objectManager	Der objectManager verwaltet die Gegner.
- NPCManager	nPCManager	Der nPCManager verwaltet die Gegner.
- ParticleManager	particleManager	Der particleManager verwaltet die Gegner.
- AnimationManager	animationManger	Der animationManager verwaltet die Gegner.
- AbilityManager	abilityManager	Der abilityManager verwaltet die Gegner.
- WeaponManager	weaponManager	Der enemyManager verwaltet die Gegner.
- BulletCollision	bulletCollision	BulletCollision verwaltet die Collision der Projektile mit Gegner und Spielern.
- MainMenu	_mainmenu	_mainMenu zeichnet und verwaltet das Hauptmenu
- PauseMenu	_pausemenu	_pauseMenu zeichnet und verwaltet das Pause-Menu
- GameOver	_gameover	_gameover verwaltet das ganze Spielaus.
- Map_Changer	_mapChanger	_mapChanger ist für das Wechseln der Maps zuständig.
- Movement	movement	Movement sorgt für die Spielerbewegung.
- Collision	collision	Collision ist für alle Arten von Kollisionen mit der Welt verantwortlich.
- HUD	hud	Hud ist für das Spieleroverlay verantwortlich.
- SaveLoad	_saveLoad	_saveLoad verwaltet Speichern und Laden.

- Layer	drawlayer1, 2, 3 und background	Sie zeichnen und laden die Kartenschichten.
- enum	GameState	Speichert die einzelnen Stadien in denen das Spiel sein kann (Hauptmenu, Pause und im Spiel)
- GameState	currentState	GameState als Variable.
- enum	Spieler	Speichert die Klassen des Spielers.
- Spieler	currentSpieler	Spieler als Variable.
- string	CurrentSpielerString	CurrentSpieler als string für Speicherung.
- int	savegame	Der momentane Spielstand als Nummer.
- Song	songMenu, songBoss, songStandart	Die Musik die während dem Spiel läuft.
- bool	ignoreCooldown	Ist dafür da ob der Klickcooldown ignoriert werden soll.
+ float	musicVolume	Die Lautstärke der spielenden Musik.
+ int	ResolutionWidth	Die Breite der Auflösung.
+ int	ResolutionHeight	Die Höhe der Auflösung.
+ string	windowMode	Der momentane Fenstermodus als string.
- bool	newMap	Ob eine neue Map geladen werden muss.
- float	clickCooldown	Die Verzögerung zwischen Klicks.
- float	pauseCooldown	Die Verzögerung zwischen den Pausenmenu-Aufrufen.
- SpriteFont	font	Die Schrift welche im Spiel benutzt werden.
- Texture2D	pause	Pauseschrift welche im Pausenmenu kommt.

Methoden

Game1 besitzt 18 Methoden([Initialize](#), [LoadContent](#), [UnloadContent](#), [Update](#), [Draw](#), [UpdateGameScreen](#), [UpdateMapChange](#), [DrawGameScreen](#), [DrawMapChange](#), [Load](#), [Save](#), [newGame](#), [resetClickCooldown](#), [graphicChange](#), [autoResolution](#), [BackMainMenu](#), [changeClass](#), [changeState](#)). **Game1** besitzt ebenfalls auch einen Konstruktor welche sich selbst an den GraphicsDeviceManager übergibt und einen Contentpfad setzt. Initialize wird einmal aufgerufen und setzt alle Objekte und Anfangswerte. LoadContent lädt alle wichtigen Dateien aus dem Content-Ordner und macht sie benutzfähig wie z.B. Texturen. UnloadContent entfernt diese wieder. Update aktualisiert unser Programm und durch Update sind Zeitaufgaben möglich. Draw zeichnet wird benutzt um alles auf dem Bildschirm darzustellen. UpdateGameScreen ist ein Teil von Update da unser Spiel in Spielstadien aufgeteilt ist wird diese Methode benötigt (Wir brauchen 3 UpdateMethoden und diese ist eine davon). UpdateMapChange aktualisiert den Kartenwechsel und führt alle nötigen Aktionen für den Wechsel aus. DrawGameScreen wird aus dem gleichen Grund wie UpdateGameScreen benötigt. DrawMapChange zeichnet den Mapübergang. Load und Save sind für Laden und Speichern da damit im Hauptmenu geladen und im Spiel gespeichert werden kann (Die externen Klassen können diese Aktionen nur durch diese Methoden durchführen). NewGame ist dafür da um ein neues Spiel zu erstellen sie wird im MainMenu aufgerufen. ResetClickCooldown ist dafür da das die Klickverzögerung bei Ziehleisten ignoriert wird. GraphicChange und autoResolution werden dafür benötigt das in den Optionen die Grafikeinstellungen übernommen werden. BackMainMenu wird benutzt um unser Spiel an den Anfang zu bringen. ChangeClass ermöglicht das Wechseln der Klassen durhc andere Klassen (.cs). ChangeState ermöglicht das Wechseln des Spielstadium durch andere Klassen.

+ `Initialize()`

`Initialize()` Wird aufgerufen um alle Objekte und Anfangswerte zu setzen.

Hier werden alle Einstellungen geladen und übernommen. Ebenfalls wird auch die Musik eingestellt und die Grafik angepasst.

+ `LoadContent()`

`LoadContent()` wird aufgerufen um alle Texturen und Bilder zu laden.

Eine der wichtigsten Klassen da sie die Texturen reinlädt und fertig macht.

+ `UnloadContent()`

`UnloadContent()` wird von uns nicht benutzt und hat deswegen in unserem Spiel keinen Nutzen.

+ `Update(GameTime gameTime)`

`Update(..)` wird immer in einem bestimmten Zeit in der Spielschleife aufgerufen.

`GameTime` ermöglicht arbeiten mit der Zeit und Zietabläufe. `Update` aktualisiert alle Position sorgt für Bewegung und ermöglicht das Updaten der Gegner (schießen, laufen). Die `Update` Methode wird jedoch unterteilt da wir mehrere Spielstadien haben.

+ `Draw(GameTime gameTime)`

`Draw(..)` zeichnet alles was angezeigt wird und wird in der Spielschleife nach `Update` aufgerufen.

`Draw` zeichnet alle Dinge die angezeigt werden. Die Methode wird jedoch auch wegen der Spielstadien unterteilt da ja nicht das Menu und das Spiel gezeichnet werden soll.

+ `UpdateGameScreen(GameTime gameTime)`

`UpdateGameScreen(..)` ist ein Teil der `Update`-Methode.

`UpdateGameScreen` wird aufgerufen wenn der Spielstatus auf `Spiel` steht. Die Methode aktualisiert alles rund um das Spiel und das spielen.

+ `DrawGameScreen(GameTime gameTime)`

`DrawGameScreen(..)` ist ein Teil der `Update`-Methode.

`DrawGameScreen` wird aufgerufen wenn der Spielstatus auf `Spiel` steht. Die Methode zeichnet alles rund um das Spiel wie z.B. Karten, Gegner usw..

+ `DrawMapChange(GameTime gameTime)`

`DrawMapChange(..)` ist ein Teil der `Update`-Methode.

`DrawMapChange` wird aufgerufen wenn die Karte wechselt. Sie zeichnet die Karten im Hintergrund und ruft die `Draw`-Methode aus `Map_Change` auf damit die Schwärze gezeichnet wird.

+ **Load**(int sg)

Load(..) sorgt dafür das man im Spiel einen Spielstand laden kann.

Sg sagt welcher Speicherstand geladen werden soll. Load setzt alle wichtigen Werte des Spielers und lädt den Kartennamen damit die Karte geladen werden kann.

+ **Save**()

Save() sorgt dafür das das Spiel speichert.

Save speichert alle wichtigen Werte des Spielers und auch seine momentane Karte.

+ **newGame**()

newGame() sorgt dafür das man ein neues Spiel anfangen kann.

NewGame erstellt einen neuen Spielstand und setzt die Werte vom Anfang des Spiels.

+ **resetClickCoolDown**()

resetClickCoolDown() setzt alle Verzögerungen zurück.

ResetClickCoolDown ermöglicht die Benutzung von Ziehleisten da die Klickverzögerung das normalerweise nicht zulassen würde.

+ **graphicChange**(int window, string language)

GraphicChange(..) übernimmt alle Grafikeinstellungen.

Durch graphicChange kann man alle eingestellten Grafikeinstellungen wirksam machen. Es speichert alle Einstellungen ebenso wie die Sprache. Window gibt an in welchem Fenstermodus das Spiel laufen soll.

+ **autoResolution**()

autoResolution() übernimmt alle Grafikeinstellungen.

AutoResolution setzt die Spielauflösung auf die Auflösung des Bildschirmes.

+ **BackMainMenu**()

BackMainMenu() setzt das Spiel auf den Anfang vom Hauptmenu.

BackMainMenu verändert der Status auf Hauptmenu und setzt den Map_Changer zum Anfang.

+ **ChangeClass**(int classNumber)

ChangeClass(..) kann die Klasse des Spielers ändern.

Durch die Angabe von classNumber kann man sagen in welche Klasse die Klasse des Spielers verändert werden soll. Wird benutzt vom ShopNPC.

+ **ChangeState**(int state)

ChangeState(..) kann den Status des Spiels verändern.

ChangeState kann durch die Angabe von state das Spiel in den gewünschten Status setzen. Die Methode wird vom Hauptmene, Pausemenu und vom Todesscreen unter anderem verwendet.

Ordner:

3.2 Abilities Manager

Klassen:

3.2.1 AbilityManager

3.2.2 ShotEngine

3.2.1 AbilityManager

Die Klasse **AbilityManager** verwaltet alle Abilitys des Spielers(Archer, Wizard, Knight).

Public Attribute

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
+ List<ShotEngine>	lstAbility	Liste in der alle Schuss Objekte des Spielers gespeichert wird.
+ List<ShotEngine>	lstNoCollision	Schuss Objekte die keine Kollision haben(Archer Ability 3 einzelner Schuss).
+ Rectangle[.]	knightRectangles = new Rectangle[3,3]	2D Rechteck Array für die Knight Abilites, damit die Schüsse abprallen.
+ Texture2D	arrow	Pfeil Textur.
+ Texture2D	arrowice	Eis Pfeil Textur.
+ Texture2D	arrowtoxic	Gift Pfeil Textur.
+ Texture2D	arrowpierce	Pierce Textur.
+ Texture2D	fireball	Feuerball Textur.
+ Texture2D	fireball2	Feuerball2 Textur.
+ Texture2D	fireball3	Feuerball3 Textur.
+ Texture2D	fireball4	Feuerball4 Textur.
+ Texture2D	arrowbig	Großer Pfeil Textur.

Private Attribute

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- float	maxCooldownAutoAttackArcher	Maximaler Cooldown der Standard Attacke des Bogenschützen.
- float	maxCooldownAbility1Archer	Maximaler Cooldown der Ability 1 des Bogenschützen.
- float	maxCooldownAbility2Archer	Maximaler Cooldown der Ability 2 des Bogenschützen.
- float	maxCooldownAbility3Archer	Maximaler Cooldown der Ability 3 des Bogenschützen.
- float	maxCooldownSpecialArcher	Maximaler Cooldown der Dash Ability des Bogenschützen.
- float	maxCooldownAutoAttackWizard	Maximaler Cooldown der Standard Attacke des Magiers.
- float	maxCooldownAbility1Wizard	Maximaler Cooldown der Ability 1 des Magiers.
- float	maxCooldownAbility2Wizard	Maximaler Cooldown der Ability 2 des Magiers.
- float	maxCooldownAbility3Wizard	Maximaler Cooldown der Ability 3 des Magiers.
- float	maxCooldownSpecialWizard	Maximaler Cooldown der Dash Ability des Magier.
- float	maxCooldownAutoAttackKnight	Maximaler Cooldown der Standard Attacke des Ritters.
- float	maxCooldownAbility1Knight	Maximaler Cooldown der Ability 1 des Ritters.
- float	maxCooldownAbility2Knight	Maximaler Cooldown der Ability 2 des Ritters.
- float	maxCooldownAbility3Knight	Maximaler Cooldown der Ability 3 des Ritters.
- float	maxCooldownSpecialKnight	Maximaler Cooldown der Dash Ability des Ritters.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- float	cooldownAutoAttackArcher	Cooldown Standard Attacke Bogenschützen.
- float	cooldownAbility1Archer	Cooldown Ability 1 Bogenschützen.
- float	cooldownAbility2Archer	Cooldown Ability 2 Bogenschützen.
- float	cooldownAbility3Archer	Cooldown Ability 3 Bogenschützen.
- float	cooldownSpecialArcher	Cooldown Dash Ability Bogenschützen.
- float	cooldownAutoAttackWizard	Cooldown Standard Attacke Magier.
- float	cooldownAbility1Wizard	Cooldown Ability 1 Magiers.
- float	cooldownAbility2Wizard	Cooldown Ability 2 Magier.
- float	cooldownAbility3Wizard	Cooldown Ability 3 Magier.
- float	cooldownSpecialWizard	Cooldown Dash Ability des Magier.
- float	cooldownAutoAttackKnight	Cooldown Standard Attacke Ritter.
- float	cooldownAbility1Knight	Cooldown Ability 1 Ritter.
- float	cooldownAbility2Knight	Cooldown Ability 2 Ritter.
- float	cooldownAbility3Knight	Cooldown Ability 3 Ritter.
- float	cooldownSpecialKnight	Cooldown Dash Ability Ritter.
- int	WizardReihenfolge	Wizard Reihenfolge welche Feuerball Textur als nächstes übergeben wird.

Methoden

Alle **Ability..**/**AutoAttack** Methoden werden in **UpdateAbilities()** aufgerufen.

Methodenname	Beschreibung
+ AbilityManager()	Es werden allen cooldown Attribute maxCooldown zugewiesen. Z.B. cooldownAutoAttackArcher = maxCooldownAutoAttackArcher..
+ AutoAttackArcher (Vector2 mouse , Vector2 player , Texture2D texture , int lvl)	Es wird lstAbility ein neues ShotStandard Objekt mit den gegebenen Parametern erzeugt.
+ Ability1Archer (Vector2 mouse , Vector2 player , Texture2D texture , int lvl)	Es wird lstAbility ein neues ShotStandard Objekt mit den gegebenen Parametern erzeugt. Allerdings wird Ability1Archer 10 Mal in UpdateAbilities() aufgerufen.
+ Ability2Archer (Vector2 mouse , Vector2 player , Texture2D texture , int lvl)	Es werden 72 ShotStandard Objekte (als Kreis angeordnet) in lstAbility hinzugefügt.
+ Ability3Archer (bool auswahl , Vector2 mouse , Vector2 player , Texture2D texture1 , Texture2D texture2 , Texture2D texture3 , int lvl)	<p>Es wird ein Pfeil erzeugt der beim erneuten Tastendruck(E) explodiert.</p> <p>Auswahl = true → ShotStandard Objekt wird lstNoCollision hinzugefügt.</p> <p>Auswahl = false → Explosion. Wird auf false gesetzt wenn cooldownAbility3Archer = 0.5 ist oder der einzelne Pfeil (mit dem Rand oder einem Boss) kollidiert.</p> <p>Um eine Explosion zu erzeugen wird mithilfe von Sinus und Cosinus um den Pfeil (lstNoCollision[0].ShotXY) in jede Richtung 3 weitere Pfeile (mit unterschiedlicher Geschwindigkeit (2.5, 3, 3.5)) erzeugt. Als letztes wird lstNoCollision[0] entfernt.</p>
+ AutoAttackWizard (Vector2 mouse , Vector2 player , Texture2D texture , int lvl)	Es wird lstAbility ein neues ShotAnimation Objekt mit den gegebenen Parametern hinzugefügt.
+ Ability1Wizard (Vector2 mouse , Vector2 player , Texture2D texture , int lvl)	<p>Erzeugt ein Ring aus Feuerbällen in die Richtung des Mauszeigers.</p> <p>Es werden 18 Feuerbälle (als Kreis angeordnet) in lstAbility hinzugefügt. x/y ist die Start Positon (Feuerbälle um den Spieler). x1/y1 ist die Ziel Position.</p>
+ Ability2Wizard (Vector2 mouse , Vector2 player , Texture2D texture1 , Texture2D texture2 , Texture2D texture3 , Texture2D texture4 , int lvl)	<p>Kleine Explosion mit verschiedenen Feuerbällen an der Position des Mauszeigers.</p> <p>Es werden 4 verschiedene Feuerball Texturen als Kreis angeordnet (WizardReihenfolge). Der Liste lstAbility werden demnach 18 ShotAnimation Objekte hinzugefügt.</p>
+ Ability3Wizard (Vector2 mouse , Vector2 player , Texture2D texture1 , Texture2D texture2 , Texture2D texture3 , int lvl)	<p>Explosion als Stern angeordnet an der Position des Spielers.</p> <p>float j → Geschwindigkeit wie schnell sich die Projektile ausbreiten.</p> <p>bool anordnung → Anordnung der Projektile als Stern.</p> <p>Es werden 3 ShotAnimation Objekte mit jeweils unterschiedlichen j Werten erstellt. anordnung wird true, wenn der Winkel(w) % 40 gleich 20 ist. Somit wird j bei jedem Durchgang um 0.25 erhöht. Wenn der Winkel(w) % 40 gleich 0 ist, wird anordnung auf false gesetzt und somit j bei jedem Durchgang um 0.25 vermindert.</p>

+ <code>AutoAttackKnight(Vector2 mouse, Vector2 player, Texture2D texture, int lvl)</code>	Es werden 6 <code>ShotAnimation</code> Objekte erstellt. Je nach dem wo sich der Mauszeiger befindet, spreizen sich die Objekte unterschiedlich.
+ <code>Ability1Knight(Vector2 mouse, Vector2 player, Texture2D texture, int lvl)</code>	Es werden 6 <code>ShotAnimation</code> Objekte erstellt. Je nach dem wo sich der Mauszeiger befindet, spreizen sich die Objekte unterschiedlich.
+ <code>Ability2Knight(Vector2 mouse, Vector2 player, Texture2D texture, int lvl)</code>	Es werden 6 <code>ShotAnimation</code> Objekte erstellt. Je nach dem wo sich der Mauszeiger befindet, spreizen sich die Objekte unterschiedlich.
+ <code>Ability3Knight(Vector2 mouse, Vector2 player, Texture2D texture, int lvl)</code>	Es werden 6 <code>ShotAnimation</code> Objekte erstellt. Je nach dem wo sich der Mauszeiger befindet, spreizen sich die Objekte unterschiedlich.
+ <code>UpdateCooldown(GameTime gameTime, int currentPlayer)</code>	Je nach dem welcher Spieler ausgewählt ist(<code>currentPlayer</code>), wird überprüft ob <code>cooldown..</code> kleiner als <code>maxCooldown..</code> ist, und somit die vergangene Zeit(seit dem letzten Update) <code>cooldown..</code> dazu addiert.
+ <code>DrawAbilities(SpriteBatch spriteBatch, GameTime gameTime)</code>	Eine For-Schleife die alle <code>IstAbility[i].Draw()</code> aufruft und somit zeichnet.

+ `UpdateAbilities`(GameTime `gameTime`, MouseState `mouseState`, KeyboardState `keyState`, Movement `movement`, Vector2 `mouseposition`, Player `_player`, int `currentplayer`, EnemyManager `enemyManager`, WeaponManager `weapon`)

In der `UpdateAbilities` Methode werden alle `Ability../AutoAttack..` Methoden aufgerufen.

Archer (currentplayer = 0):

Links Klick:

Es wird überprüft ob `CooldownAutoAttackAcher` größer gleich `maxCooldownAutoAttackAcher` ist, und somit `AutoAttackArcher(..)` aufgerufen und `CooldownAutoAttackArcher` = 0 gesetzt.

Taste Shift (Freigeschaltet ab `_player.LvlArcher` >= 2):

Es wird überprüft ob `CooldownAbility1Acher` größer gleich `maxCooldownAbility1Acher` ist, und somit `CooldownAbility1Archer` = 0 gesetzt.

Nun wird in einer For-Schleife abgefragt ob `CooldownAbility1Archer` >= i und <= i + 0.01 ist. Somit wird `Ability1Archer(..)` 10 Mal aufgerufen.

Taste Q (Freigeschaltet ab `_player.LvlArcher` >= 4):

Es wird überprüft ob `CooldownAbility2Acher` größer gleich `maxCooldownAbility2Acher` ist, und somit `Ability2Archer(..)` aufgerufen und `CooldownAbility2Archer` = 0 gesetzt.

Taste E (Freigeschaltet ab `_player.LvlArcher` >= 6):

Es wird überprüft ob `CooldownAbility3Acher` größer gleich `maxCooldownAbility3Acher` ist, und somit `Ability3Archer(true,..)` (`Auswahl` = true d.h. einzelner Pfeil in `IstNoCollision`) aufgerufen und `CooldownAbility3Archer` = 0 gesetzt.

In der For-Schleife wird jedes `IstNoCollision` Objekt aktualisiert. Nun wird überprüft ob sich das Objekt mit dem Rand kollidiert (-/+ 45 weil sonst die Schüsse wegen der Collision sofort verschwinden). Wenn dies zutrifft wird `IstNoCollision[i].CurrentTime` auf 0.9 gesetzt. Wenn es nicht zutrifft wird überprüft ob `IstNoCollision[i].CurrentTime` >= 0.2 ist somit man 0.2 Sekunden warten muss bevor die Explosion ausgelöst werden kann. Als letztes wird überprüft ob `IstNoCollision[i].CurrentTime` >= 0.9 ist, und somit `Ability3Archer(false, ..)` aufgerufen.

Rechts Klick:

Es wird überprüft ob `CooldownSpecialArcher` >= `maxCooldownSpecialArcher` && `CooldownSpecialArcher` >= `CooldwonSpecialArcher` + 0.01 ist. Demnach wird `movement.speed` geändert und `CooldownSpecialArcher` 0 gesetzt.

Nun wird geschaut ob `CooldwonSpecialArcher` >= 0.01 ist, und somit `movement.speed` wieder auf Standard(5) gesetzt.

Wizard (currentplayer = 1):

Links Klick:

Es wird überprüft ob `CooldownAutoAttackWizard` größer gleich `maxCooldownAutoAttackWizard` ist, und somit `AutoAttackWizard(..)` aufgerufen und `CooldownAutoAttackArcher` = 0 gesetzt.

Taste Shift (Freigeschaltet ab `player.LvlWizard` >= 2):

Es wird überprüft ob `CooldownAbility1Wizard` größer gleich `maxCooldownAbility1Wizard` ist, und somit `Ability1Wizard(..)` aufgerufen und `CooldownAbility1Wizard` = 0 gesetzt.

Taste Q (Freigeschaltet ab `player.LvlWizard` >= 4):

Es wird überprüft ob `CooldownAbility2Wizard` größer gleich `maxCooldownAbility2Wizard` ist, und somit `Ability2Wizard(..)` aufgerufen und `CooldownAbility2Wizard` = 0 gesetzt.

Taste E (Freigeschaltet ab `player.LvlWizard` >= 6):

Es wird überprüft ob `CooldownAbility3Wizard` größer gleich `maxCooldownAbility3Wizard` ist, und somit `Ability3Wizard(..)` aufgerufen und `CooldownAbility3Wizard` = 0 gesetzt.

Rechts Klick:

Es wird überprüft ob `CooldownSepecialWizard` >= `max CooldownSepecialWizard` && `CooldownSepecialWizard` >= `CooldownSepecialWizard` + 0.01 ist. Demnach wird `movement.speed` geändert und `CooldownSepecialWizard` 0 gesetzt.

Nun wird geschaut ob `CooldwonSpecialWizard` >= 0.01 ist, und somit `movement.speed` wieder auf Standard(5) gesetzt.

Knight (currentplayer = 2):

Links Klick:

Es wird überprüft ob `CooldownAutoAttackKnight` größer gleich `max CooldownAutoAttackKnight` ist, und somit `AutoAttackKnight(..)` aufgerufen und `CooldownAutoAttackKnight` = 0 gesetzt.

Taste Shift (Freigeschaltet ab `player.LvlKnight` >= 2):

Es wird überprüft ob `CooldownAbility1Knight` größer gleich `maxCooldownAbility1Knight` ist, und somit `Ability1Knight(..)` aufgerufen und `CooldownAbility1Knight` = 0 gesetzt.

Taste Q (Freigeschaltet ab `player.LvlKnight` >= 4):

Es wird überprüft ob `CooldownAbility2Knight` größer gleich `maxCooldownAbility2Knight` ist, und somit `Ability2Knight(..)` aufgerufen und `CooldownAbility2Knight` = 0 gesetzt.

Taste E (Freigeschaltet ab `player.LvlKnight` >= 6):

Es wird überprüft ob `CooldownAbility3Knight` größer gleich `maxCooldownAbility3Knight` ist, und somit `Ability3Knight(..)` aufgerufen und `CooldownAbility3Knight` = 0 gesetzt.

Rechts Klick:

Es wird überprüft ob `CooldownSepecialKnight` >= `max CooldownSepecialKnight` && `CooldownSepecialKnight` >= `CooldownSepecialKnight` + 0.01 ist. Demnach wird `movement.speed` geändert und `CooldownSepecialKnight` 0 gesetzt.

Nun wird geschaut ob `CooldwonSpecialKnight` >= 0.01 ist, und somit `movement.speed` wieder auf Standard(5) gesetzt.

3.2.2 ShotEngine

Die Klasse **ShotEngine** ist die Basisklasse von **ShotAnimation** und **ShotStandard**.

Attribute

Datentyp	Attributname	Beschreibung
- Vector2	shotXY	Schuss Position.
- Vector2	mouseXY	Maus Position.
- Vector2	playerXY	Spieler Position.
- Vector2	directionXY	Schuss Richtung.
- Texture2D	shotTexture	Schuss Textur.
- float	shotSpeed	Schuss Geschwindigkeit.
- float	oldSpeed	Schuss Geschwindigkeit vor der Berechnung.
- float	Rotation = 0	Schuss Rotation.
- float	currentTime = 0	Schuss Lebenszeit.
- Vector2	shotSize	Größe des Projektils.
- int	shotDamage	Schaden des Projektils.
- string	shotName	Name des Projektils.
+ Animations	animation	Objekt der Animations Klasse.

Methoden

Methodenname	Beschreibung
+ ShotEngine (Vector2 target, Vector2 start, float speed, Texture2D texture, Vector2 size, int damage, string name)	Zuweisen von shotName. OldSpeed, shotDamage, shotSize, shotSpeed, MouseXY, ShotTexture. DirectionXY ist die Richtung in der sich das Projektil bewegt(MouseXY - start). Die rotation (also die Neigung) wird mit Math.Atan2(DirectionXY.Y, DirectionXY.X) berechnet. Um eine von der Maus Position(MouseXY) unabhängige Geschwindigkeit(shotSpeed) zu erhalten, wird float speed1 der Betrag (Vektor2.Length()) von DirectionXY zugewiesen. Nun wird speed durch speed1 dividiert und mit DirectionXY multipliziert.
+ Update (Vector2 mouse, Vector2 player, Texture2D texture, int lvl)	DirectionXY wird mit der vergangenen Zeit(seit dem letzten Update Aufruf) multipliziert und zu ShotXY addiert. CurrentTime wird ebenfalls mit die vergangenen Zeit addiert.
+ virtual Draw (SpriteBatch spriteBatch, GameTime gameTime)	

ShotStandard : ShotEngine

Die Klasse **ShotEngine** wird von **ShotEngine** geerbt. Das Objekt der Klasse ist ein Projektil ohne Animation.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ ShotStandard (Vector2 target , Vector2 start , float speed , Texture2D texture , Vector2 size , int damage , string name) : base(target , start , speed , texture , _size , _damage , _name)	Es werden alle Parameter an die Basis Klasse übergeben.
+ override Draw (SpriteBatch spriteBatch , GameTime gameTime)	Mithilfe von spriteBatch.Draw(..) wird ShotTexture an der Position von ShotXY sowie mit der angegebenen Rotation gezeichnet.

ShotAnimation : ShotEngine

Die Klasse **ShotAnimation** wird von **ShotEngine** geerbt. Das Objekt der Klasse ist ein Projektil mit Animation.

Attribute

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- int	width	Breite der Textur
- int	height	Höhe der Textur
- int	fps	Bilder pro Sekunde.
- int	frames	Anzahl der Bilder.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ ShotAnimation (Vector2 target , Vector2 start , float speed , Texture2D texture , int _fps , int _frames , int _width , int _height , Vector2 size , int damage , string name) : base(target , start , speed , texture , _size , _damage , _name)	Bis auf _width , _height , _fps und _frames , werden alle Parameter an die Basis Klasse übergeben. _width , _height , _fps und _frames werden den gegebenen Attributen zugeteilt.
+ override Draw (SpriteBatch spriteBatch , GameTime gameTime)	Es wird animations.drawanimation(..) aufgerufen und mit den gegebenen Parametern gezeichnet.

Ordner:

3.3 Animation Manager

Klassen:

3.3.1 AnimationManager

3.3.2 Animations

3.3.1 AnimationManager

Die Klasse **AnimationManager** verwaltet die Spieler Animationen.

Attribute

AnimationManager besitzt 8 Attribute.(Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Animations	animations	Objekt der Animations Klasse
- Vector2	playerpositionoffset	Spieler Position + Spieler Mitte (Vector2(16, 24))
+ Texture2D	ArcherMove	Bogenschütze Bewegung Animation
+ Texture2D	ArcherIdle	Bogenschütze untätig Animation
+ Texture2D	WizardMove	Magier Bewegung Animation
+ Texture2D	WizardIdle	Magier untätig Animation
+ Texture2D	KnightMove	Ritter Bewegung Animation
+ Texture2D	KnightIdle	Ritter untätig Animation

Methoden

AnimationManager besitzt eine Methode(**PlayerAnimation**). Je nach Tastendruck(**keyState**) sowie Position des Mauszeigers(**mouseState**) wird eine Animation gezeichnet.

+ **PlayerAnimation**(Vector2 **playerposition**, Vector2 **mouseposition**, int **currentplayer**, KeyboardState **keyState**, MouseState **mouseState**, SpriteBatch **spriteBatch**,GameTime **gameTime**)

PlayerAnimation(..) wird in **Game1.DrawGameScreen**(..) aufgerufen.

Zu Beginn wird **playerposition** mit dem Offset des Spielers(16, 24) addiert und in **playerpositionoffset** gespeichert. Danach wird geschaut an welcher Stelle sich der Mauszeiger befindet(**mouseposition**), wenn der Zeiger rechts vom Spieler(**playerpositionoffset**) ist, soll der Spieler nach rechts schauen und wenn der Zeiger links ist soll der Spieler nach links schauen. Als nächstes wird mit **keyState** geprüft ob eine Taste gedrückt wird demnach wird ausgewählt ob eine ..**Idle** oder ..**Move** Animation gezeichnet wird. Mit **currentplayer**(0 = Archer, 1 = Wizard, 2 = Knight) wird dann geprüft, welcher Spieler(**Archer**.., **Wizard**.., **Knight**..) gezeichnet wird.

3.3.2 Animations

Die Klasse **Animations** verwaltet die allgemeinen Animationen.

Attribute

Animations besitzt 2 Attribute. (Tabelle)

Datentyp	Attributname	Beschreibung
- int	animation	Die Bildzahl in der Animation
- TimeSpan	zeit	Zeitspanne für die Verzögerung

Methoden

Animations besitzt sechs Methoden (**getanimation**, **drawanimation**, **drawanimationrotation**, **drawanimationrotationknight**, **drawanimationGespiegelt**, **drawanimationGespiegeltVertically**). **getanimation** ist dafür verantwortlich den richtigen Frame zurückzugeben. Die anderen fünf Methoden sind dafür da die animation zu zeichnen je nachdem ob man sie gespiegelt, gedreht oder normal will.

+ **getanimation**(GameTime gameTime, int fps, int frames)

getanimation(..) wird überall aufgerufen wo eine Animation angefordert wird.

Durch die Angabe der fps wird angegeben wie schnell die frames pro Sekunde durchlaufen. Durch die angegebene Anzahl der Frames wird angegeben wie viele Bilder die Bilderfolge hat. Die Variable **animation** wird alle X-Millisekunden/Sekunden hochgezählt (Angabe durch fps. Bsp: 60Fps = 60 Bilder pro Sekunden = Alle 16,6ms wird **animation** hochgezählt und die Bilderfolge weitergeführt). Die vergangene Zeit wird mit gameTime.TotalGameTime berechnet. Bsp: **zeit** + TimeSpan.FromMilliseconds(Angegebene Fps-Zeit) < gameTime.TotalGameTime (gesamtvergangene Zeit) Solange die **zeit**, welche auf die gesamt vergangene Zeit gesetzt wird, plus die angegebene Wechselzeit kleiner ist als die vergangene Zeit dann soll die Bildfolge nicht weiter gehen. Wenn die Bildzahl noch nicht die letzte erreicht hat wird sie hochgezählt ansonsten wird sie zurück auf 0 gesetzt und das erste Bild der Folge wird geladen. Am Ende der Methode wird die momentane Bildzahl ausgegeben.

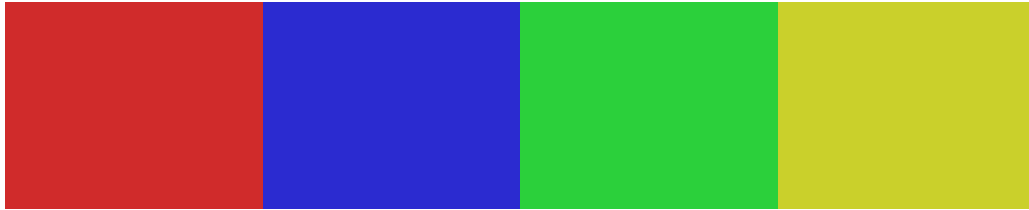
+ **drawanimation**(Texture2D **textur**, Vector2D **Position**, SpriteBatch **spriteBatch**, int **frame**, int **width**, int **height**)
+ **drawanimationrotation**(Texture2D **textur**, Vector2D **Position**, SpriteBatch **spriteBatch**, int **frame**, int **width**, int **height**, float **rotation**)
+ **drawanimationrotationknight**(Texture2D **textur**, Vector2D **Position**, SpriteBatch **spriteBatch**, int **frame**, int **width**, int **height**, float **rotation**)
+ **drawanimationGespiegelt**(Texture2D **textur**, Vector2D **Position**, SpriteBatch **spriteBatch**, int **frame**, int **width**, int **height**, float **rotation**)
+ **drawanimationGespiegeltVertically**(Texture2D **textur**, Vector2D **Position**, SpriteBatch **spriteBatch**, int **frame**, int **width**, int **height**, float **rotation**)

Die drawanimation-Methoden werden überall benutzt wo eine Animation gezeichnet werden soll.

In jeder Methode wird spriteBatch.Draw aufgerufen. Die **textur** gibt die Bildfolge an. Die **Position** gibt die Position der Animation an. **Frame**, **width** und **height** werden benutzt um das Quellenrechteck anzugeben was das Bild aus der Bildfolge lädt. **rotation** gibt bei den anderen Methoden die Rotation oder die Spiegelung der Animation an wie z.B. bei Projektilen oder bei dem Ritter das Schwert.

Animations Beispiel

Bilderfolge Beispiel:



Frame 0 (1)

Frame 1 (2)

Frame 2 (3)

Frame 3 (4)

Da der Framezähler mit 0 anfängt, wird bei 0 angefangen zu zählen.

Aus der Bilderfolge mit 4 Bilder (Jeweils 40x32) werden durch Angabe von **frame**, **width** und **height** die jeweiligen Bilder ausgeschnitten.

Wenn **frame** = 2, **width** = 40 und **height** = 32 dann wird der grüne Frame gezeichnet.

Sollte man falsche werte eingeben und width wird z.B. verdoppelt dann werden bei **frame** = 2 der grüne und gelbe Frame gezeichnet.

Ordner:










3.4 Gegner Manager











Klassen:



3.4.1 EnemyManager

3.4.2 Enemy

3.4.1 EnemyManager

<u>Texture</u>	<u>Name</u>	<u>Typ</u>	<u>Leben</u>	<u>Größe</u>	<u>Schaden</u>	<u>Speed</u>	<u>Radius (Pixel)</u>	<u>Cooldown (Sekunden)</u>
	Ogre	Nahkampf	15	64x64	5	3	300	0.5
	Orc	Fernkampf	2	32x40	2	2	400	2
	OrcMasked	Fernkampf	5	32x40	3	1	400	1
	Shamen	Nahkampf	8	32x40	5	2	500	2
	Egg	Nahkampf Fernkampf	100	176x176	3	2	2000	0.2
	IceZombie1	Fernkampf	9	32x40	6	2	400	2
	IceZombie2	Nahkampf	12	32x40	6	3	500	0.5
	Yeti	Nahkampf Fernkampf	200	176x176	6	10	2000	0.125
	DemonBig	Nahkampf	30	64x64	10	2	400	0.5

	Chort	Fernkampf	15	32x40	7	2	200	0.75
	IMP	Fernkampf	10	32x32	5	3	300	0.5
	Wogol	Fernkampf	15	32x40	8	1	400	1
	Demon	Nahkampf Fernkampf	400	176x176	8	5	2000	0.358
	ZombieBig	Nahkampf	20	64x64	1	1	500	0.4
	Skeleton	Fernkampf	5	32x40	2	1	300	2
	Zombie	Nahkampf	5	32x40	5	2	500	0.5
	ZombieTiny	Nahkampf	1	32x32	1	3	200	0.5
	Muddy	Nahkampf	15	40x40	5	1	300	0.5
	Swampy	Nahkampf	10	40x40	3	1	300	0.5

	Necromancer	Fernkampf	6	32x40	3	1	400	1
	Goblin	Nahkampf	3	32x32	1	4	800	0.5

Eis, Lava, Wald, Dungeon, Sonstige

3.4.2 Enemy

Die abstrakte Klasse **Enemy** ist die Basis Klasse von **Ranged**, **Melee** und **Boss**.

Attribute

Enemy besitzt 18 Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
# Animations	animations	Animations Objekt.
# Texture2D	enemyTextureIdle	Gegner Still Animation.
# Texture2D	enemyTextureRun	Gegner Lauf Animation.
# float	enemyspeed	Gegner Laufgeschwindigkeit.
# Vector2	playerXY	Spieler Position → Gegner Ziel Position.
# Vector2	enemySize	Größe der Collision.
# int	enemyHP	Gegner aktuelles Leben.
# int	enemyMaxHP	Gegner Maximales Leben.
# Texture2D	enemyAnzeige	Gegner Lebensanzeige
# Texture2D	enemyFrame	Gegner Lebensframe
# int	enemyRadius	Gegner Fokussiert den Spieler erst wenn der Spieler sich in sein Radius befindet.
# bool	enemyTarget	True = Gegner verfolgt Spieler.
# float	enemyCooldown	Zeit bis zum nächsten Angriff.
# float	enemyCooldownMax	Zeit bis zum nächsten Angriff.
# string	enemyName	Zuteilung des Gegner Namen.
# Random	enemyRandom	Random Objekt das übergeben wurde.
# int	enemyDamage	Schaden des Gegners

Methoden

Methodenname	Beschreibung
+ Enemy (Vector2 enemy , Texture2D textureIdle , Texture2D textureRun , Texture2D anzeige , Texture2D frame , float speed , Vector2 collision , int hp , int radius , float cooldownMax , string name , Random random , int dmg)	Standard Konstruktor der alle Variablen zuweist.
+ abstract Update (GameTime gameTime , Vector2 player , Collision collision)	
+ virtual Draw (SpriteBatch spriteBatch , GameTime gameTime)	Je nach dem wo sich der Spieler befindet(links/rechts neben dem Gegner) wird entweder animations.drawanimation oder animations.drawanimationGespiegelt aufgerufen. Ebenso wird geschaut ob der Gegner den Spieler fokussiert(enemyTarget) und somit eine andere Textur übergeben. Als letztes wird die Lebensanzeige gezeichnet.

Ranged : Enemy

Die Klasse **Ranged** erbt von **Enemy**, das Objekt wird für Gegner die Projektile Schießen verwendet.

Attribute

Ranged besitzt 5 Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- float	enemyShotSpeed	Schuss Geschwindigkeit.
- int	height = 0	Laufweite.
- bool	up	Laufweite.
- int	richtung	Richtung in der der Gegner läuft.
- Random	random	Random Objekt.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Ranged (Vector2 enemy , Texture2D textureIdle , Texture2D textureRun , Texture2D anzeige , Texture2D frame , float speed , Vector2 collision , int hp , int radius , float cooldownMax , float shotSpeed , string name , Random random , int dmg)	Zuweisen von EnemyShotSpeed .
+ override Update (GameTime gameTime , Vector2 player , Collision collision)	Als erstes wird die Spieler Position aktualisiert. Dann wird überprüft ob sich der Spieler im Radius von Gegner befindet und somit enemyTarget auf true gesetzt. Erst wenn enemyTarget auf true ist, wird der Spieler fokussiert.
+ override Draw (SpriteBatch spriteBatch , GameTime gameTime)	base. Draw wird aufgerufen.

Melee : Enemy

Die Klasse **Melee** erbt von **Enemy**, das Objekt wird für Nahkampf Gegner verwendet.

Attribute

Ranged besitzt 5 Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- int	height = 0	Laufweite.
- bool	up	Laufweite.
- int	richtung	Richtung in der der Gegner läuft.
- Random	random	Random Objekt.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Melee (Vector2 enemy , Texture2D textureIdle , Texture2D textureRun , Texture2D anzeige , Texture2D frame , float speed , Vector2 collision , int hp , int radius , float cooldownMax , float shotSpeed , string name , Random random , int dmg)	Zuweisen von crack und stateCooldown = 1.
+ override Update (GameTime gameTime , Vector2 player , Collision collision)	Als erstes wird die Spieler Position aktualisiert. Dann wird überprüft ob sich der Spieler im Radius von Gegner befindet und somit enemyTarget auf true gesetzt. Erst wenn enemyTarget auf true ist, wird der Spieler fokussiert.
+ override Draw (SpriteBatch spriteBatch , GameTime gameTime)	base. Draw wird aufgerufen.

Boss : Enemy

Die Klasse **Boss** erbt von **Ranged**. **Boss** ist die Basisklasse von

Attribute

Boss besitzt 3 Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Texture2D	crack	Egg crack textur.
- int	state = 0	Verschieden Boss Zustände.
- float	stateCooldown	Länge eines Zustandes.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Boss (Vector2 enemy , Texture2D textureIdle , Texture2D textureRun , Texture2D anzeige , Texture2D frame , float speed , Vector2 collision , int hp , int radius , float cooldownMax , float shotSpeed , string name , Random random , int dmg)	Zuweisen von EnemyShotSpeed .
+ override Update (GameTime gameTime , Vector2 player , Collision collision)	Als Erstes wird die Spieler Position aktualisiert. Dann wird überprüft ob sich der Spieler im Radius von Gegner befindet und somit enemyTarget auf true gesetzt. Erst wenn enemyTarget auf true ist, wird der Spieler fokussiert.
+ override Draw (SpriteBatch spriteBatch , GameTime gameTime)	Je nach dem welchen Wert stateCooldown hat, wird eine andere Textur/Animation gezeichnet und state verändert. Ebenso wird enemyFrame sowie enemyAnzeige gezeichnet.

Egg: Boss

Die Klasse **Egg** erbt von **Boss**. Das Objekt lässt ein Ei erscheinen.

Attribute

Egg besitzt 1 Attribut.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Texture2D	crack	Egg crack textur.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Egg (Vector2 enemy , Texture2D textureIdle , Texture2D textureRun , Texture2D anzeige , Texture2D frame , float speed , Vector2 collision , int hp , int radius , float cooldownMax , float shotSpeed , string name , Random random , int dmg)	Zuweisen von EnemyShotSpeed .
+ override Update (GameTime gameTime , Vector2 player , Collision collision)	Wenn State = 0 ist wird base. Update aufgerufen.
+ override Draw (SpriteBatch spriteBatch , GameTime gameTime)	Je nach dem welchen Wert stateCooldown hat, wird eine andere Textur/Animation gezeichnet und state verändert. Ebenso wird enemyFrame sowie enemyAnzeige gezeichnet.

Yeti : Boss

Die Klasse **Yeti** erbt von **Boss**. Das Objekt lässt ein Yeti erscheinen.

Attribute

Yeti besitzt 1 Attribut.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- bool	target	Spieler Fokus.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Yeti (Vector2 enemy , Texture2D textureIdle , Texture2D textureRun , Texture2D anzeige , Texture2D frame , float speed , Vector2 collision , int hp , int radius , float cooldownMax , float shotSpeed , string name , Random random , int dmg)	Zuweisen von StateCooldown = 0 .
+ override Update (GameTime gameTime , Vector2 player , Collision collision)	Gleich aufgebaut wie base. Update nur das die Spieler Position ein mal aktualisiert wird.
+ override Draw (SpriteBatch spriteBatch , GameTime gameTime)	Je nach dem welchen Wert stateCooldown hat, wird eine andere Textur/Animation gezeichnet und state verändert. Ebenso wird enemyFrame sowie enemyAnzeige gezeichnet.

Demon : Boss

Die Klasse **Yeti** erbt von **Boss**. Das Objekt lässt ein Yeti erscheinen.

Attribute

Yeti besitzt 1 Attribut.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- float	rotation	Schwert Rotation.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Demon (Vector2 position , Texture2D textureIdle , Texture2D textureCharge , Texture2D fill , Texture2D frame , float speed , Vector2 collision , int hp , int radius , float cooldownMax , float shotSpeed , string name , Random random , int dmg)	Zuweisen von StateCooldown = 0 .
+ override Update (GameTime gameTime , Vector2 player , Collision collision)	base. Update wird aufgerufen wenn State = 2 ist. Es wird ebenso rotation + 0.2 gezählt sodass sich das Schwert rotiert.
+ override Draw (SpriteBatch spriteBatch , GameTime gameTime)	Je nach dem welchen Wert stateCooldown hat, wird eine andere Textur/Animation gezeichnet und state verändert. Ebenso wird enemyFrame sowie enemyAnzeige gezeichnet.

Ordner:

3.5 Kollision Manager

Klassen:

3.5.1 BulletCollision

3.5.1 BulletCollision

Die Klasse **BulletCollision**, überprüft ob sich Gegner Projektile mit dem Spieler kollidieren, Spieler Projektile mit den Gegner kollidieren sowie die Dynamit Schüsse mit dem Gegner oder Spieler kollidieren.

Attribute

BulletCollision besitzt keine Attribute.

Methoden

+ **UpdateBulletCollision**(GameTime gameTime, AbilityManager abilityManager, ObjectManager objectManager, Collision collision, ParticleManager particleManager, EnemyManager enemyManager, Movement movement, int currentPlayer, EnemyManager Player_player, Game1 game)

In der **UpdateBulletCollision** Methode werden alle **ShotEngine** Objekte aktualisiert und überprüft ob z.B. mit dem Rand Kollidiert wird (**lstAbility**, **lstDynamitArrowTrapBullets**, **lstGegnerShots**).

In der ersten For-Schleife wird überprüft ob sich ein **ShotEngine** Objekt in einer der Listen befindet. Nun wird überprüft ob i kleiner **abilityManager.lstAbility.Count** ist und somit ein Objekt in der Liste befindet. Das **ShotEngine** Objekt wird aktualisiert. Es wird geschaut ob sich das Objekt mit dem Rand beziehungsweise mit der **Collision.BulletCollision** kollidiert oder ob das Objekt, eine Lifetime(**CurrentTime**) länger als 10 Sekunden besitzt. Wenn dies geschieht wird **particleManager.SpawnStandardParticle** aufgerufen und das **ShotEngine** Objekt entfernt. In der nächsten For-Schleife wird überprüft ob sich ein Objekt der **lstAbility** mit eines der **objectManager.lstDynamit** kollidiert und somit Explodiert (**objectManager.lstDynamit.Explosion** = true). Das Objekt wird entfernt und die aktuelle Schleife wird abgebrochen, da das Objekt entfernt wurde.

Das selbe wird mit **enemyManager.lstGegnerShot** gemacht, nur das hier überprüft wird ob sich das Objekt mit dem Spieler kollidiert. Es wird **_player.LebenAbziehen** aufgerufen und der Schaden (**enemyManager.lstGegnerShot.ShotDamage**) übergeben. Ebenfalls wird das **ShotEngine** Objekt entfernt.

Wie auch bei **lstGegnerShot** als auch bei **lstAbility** wird bei **lstDynamitArrowTrapBullets** zuerst überprüft sich das **ShotEngine** Objekt mit etwas Kollidiert. Da die Objekte mit dem Gegner als auch mit dem Spieler kollidieren können, müssen beide Listen überprüft werden. Da sich ein Objekt auch mit weiteren **lstDynamit** Objekten kollidieren kann, wird dies ebenfalls überprüft. Allerdings muss überprüft werden, ob das **Dynamit** Objekt bereits Explodiert ist (**objectManager.lstDynamit.Exploded**).

Die nächsten drei for-Schleifen sind bis auf den Listennamen alle ähnlich aufgebaut. Bei allen wird geschaut ob sich ein **ShotEngine** Objekt (**lstAbility** und **lstDynamitArrowTrapBullets**) mit dem Gegner kollidiert. Wenn sich ein Objekt von **lstAbility** mit einem Gegner (**lstGegnerRanged**, **lstGegnerMelee**, **lstBoss**) kollidiert, soll ein der Gegner Schaden(**lstAbility.ShotDamage**) bekommen sowie ein Rückstoß simuliert werden.

Ordner:

3.6 Map

Klassen:

3.6.1 Collision

3.6.2 Layer

3.6.3 Map_Changer

3.6.1 Collision

Die Klasse **Collision** behandelt die Kollision des Spielers mit seiner Umwelt (z.B gegen eine Wand laufen).

Attribute

Collision besitzt 5 Attribute. (Tabelle)

Datentyp	Attributname	Beschreibung
- Int[60,34]	CollisionLayer	CollisionLayer speichert die Werte der Collision der Maps. (Z.B. 0, 1, 2 usw.)e
+ Vector2	topleft, topright	Sie geben die Positionen der oberen zwei Ecken des Spielers an.
+ Vector2	bottomleft, bottomright	Sie geben die Positionen der unteren zwei Ecken des Spielers an.
+ Vector2	midright, midleft, midbottom, midtop	Sie geben die Positionen der Mittleren Punkte bei einem Objekt größer als 32x32 an.
+ Vector2	bulletEdge1, bulletEdge2, VectorEdges	Sie geben die Positionen der Ecken eines Projektils an.
+ string	MapPath	Gibt den Pfad der jeweiligen Collisiondatei einer Map an.

Methoden

Collision besitzt sechs Methode (**Read_Collision**, **CollisionCheck**, **BulletCollision**, **KnockbackCollision**, **MapChangeCheck**, **Clear_Layer**). **Read_Collision** liest durch einen StreamReader und einem Algorithmus die Kollisionsdatei ein und speichert diese in **CollisionLayer**. **CollisionCheck** überprüft durch die Koordinaten des Spielers und **CollisionLayer** ob der Spieler dort laufen kann. **BulletCollision** macht dasselbe mit Projektilen. **KnockbackCollision** überprüft die Kollision beim erhalten von Rückstoß durch Projektilen. **MapChangeCheck** gibt zurück an welcher Seite der Spieler die Map verlässt falls er dies tut. **Clear_Layer** leert **CollisionLayer** damit es neu beschrieben werden kann.

+ **Read_Collision()**

Read_Collision() Wird in **Game1** beim Kartenwechsel aufgerufen damit die Karte eingelesen wird.

readerchar ist die Stelle welche eingelesen wird. Die Variable wird je nach Zeichen hochgezählt. Leerzeichen werden übersprungen ebenso Zeilenwechsel. Der StreamReader bringt die Kollisionsdatei in einen Array voraus die Zahlen gelesen werden. **Read_Collision funktioniert nur bei Dateien mit 34 Zeilen und 60 durch Leerzeichen getrennte Zahlen.**

0 = Keine Kollision 1 = Kollision mit etwas festem (Wand, Stein) 2 = Kollision mit Wasser (Für Projektilen)

+ **CollisionCheck**(Vector2 position, char cSide, int height, int width)

CollisionCheck(..) Wird in überall verwendet wo sich ein Gegner oder Spieler bewegt damit die Kollision mit der Umwelt abgefragt werden kann.

Durch die Angabe der Position des Spielers können alle Eckpunkte berechnet werden. Die mittleren Punkte werden durch Schleifen bei größeren Objekten öfters berechnet. Durch die Angabe von cSide kann man angeben welche Seite überprüft werden soll (links, rechts, oben, unten). Die Angaben von der Höhe (height) und Breite (width) werden benötigt um die Eckpunkte zu berechnen und um die Kollision abzufragen. Bei einer Kollision gibt die Methode false zurück falls keine Kollision zu stande kommt gibt sie true zurück.

+ **BulletCollision**(Vector2 position, int height, int width, float rotation)

BulletCollision(..) Wird bei allen Projektilen aufgerufen um zu überprüfen ob es mit der Welt kollidiert.

BulletCollision dreht zwei Punkte jeweils der rotation, damit abgefragt werden kann ob das Projektil zerstört wird oder nicht. BulletCollision gibt nur bei festen Objekten (1) den Wert false aus da die Projektile über Wasser (2) fliegen.

Diese Art der Kollisionsabfrage konnten ich nicht richtig überprüfen!

+ **KnockbackCollision**(Vector position, int height, int width)

KnockbackCollision(..) Wird bei Objekten aufgerufen welche durch Schaden zurückgeworfen werden.

KnockbackCollision ist eine leicht veränderte Form von CollisionCheck(). Der einzige Unterschied liegt darin das hier alle Punkte abgefragt werden und nicht nur eine bestimmte Seite.

+ **MapChangeCheck**(ref Vector2 position, int height, int width, Vector2 rightChange = default(Vector2), Vector2 leftChange = default(Vector2), Vector2 topChange = default(Vector2), Vector2 botChange = default(Vector2))

MapChangeCheck(..) Wird in der Klasse **Map_Changer** aufgerufen um zu überprüfen ob der Spieler die Map verlässt. Es kann ebenfalls auch die Position des Spielers ändern.

Durch Abfragen durch die Eckpunkte wird überprüft ob der Spieler in ein Feld läuft welches die Werte 3,4, 5 oder 6 hat. Sollte dies der Fall sein wird der Spieler entweder durch Standard-Werte teleportiert oder durch selbstangegebene Werte an bestimmte Punkte teleportiert (Angaben erfolgen über die Vektoren).

3 = links raus 4 = rechts raus 5 = oben raus 6 = unten raus

+ **Clear_Layer**()

Clear_Layer() Wird in am Anfang von Read_Collision aufgerufen damit das Layer wieder leer ist und neu beschrieben werden kann.

Clear_Layer setzt durch zwei Schleifen CollisionLayer auf null.

Collision-Beispiel

1	0	0	0	0
1	0	1	1	0
1	0	1	1	0
1	0	0	0	0

Eine Datei mit dem Inhalt:

```
10000
10110
10110
10000
```

Würde einen 64x64 soliden Block in der Spielwelt darstellen durch den der Spieler nicht passieren kann und über/durch den keine Projektile fliegen.
Sollte man die inneren Einsen durch Zweien ersetzen könnte der Spieler immer noch nicht passieren doch Projektile könnten durch diesen Block durch und würde an der Wand dahinter zerspringen. Da unser Spiel einen Schirm von 1920x1080 simuliert ist bei einer 32x32 feinen Kollisionsabfrage eine Dateigröße von 60 getrennten Zeichen und 34 Zeilen notwendig (34 da 1080 nicht durch 32 Teilbar ist).

3.6.2 Layer

Die Klasse **Layer** liest die Kartendateien ein und zeichnet die einzelnen Schichten einer Karte.

Attribute

Layer besitzt 9 normale Attribute, 13 Animationsobjekte und 335 Texturattribute als Array.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
+ Texture2D[335]	texturen	Alle Texturen für die Kartenzeichnung.
- int	vtileWidth	Die Breite einer Textur
- int	vtileHeight	Die Höhe einer Textur
- int	vWidth	Die Breite der Karte
- int	vHeight	Die Höhe der Karte
+ Matrix	scaling	Eine Matrix für die Skalierung der Karte auf andere Auflösungen.
- int[30,17]	MapLayer	Speichert die einzelnen Dateien für die Schichten zwischen.
+ string	path	Der Pfad für die eingelesenen Dateien.
+ bool	SpawnObject	Damit Gegner und Objekte nur einmal gespawnt werden.

Methoden

Layer besitzt 3 Methode(**draw**, **ReadLayer**, **Clear_Layer**). Draw Zeichnet eine Schicht der Karte. ReadLayer liest die Schichten ein und Clear_Layer löscht alles auf MapLayer.

+ **draw**(SpriteBatch spriteBatch, GameTime gameTime, ObjectManager objectManager, EnemyManager enemyManager, NPCManager nPCManager, HUD hud, Player _player)

draw(..) Wird in **Game1** bei Draw aufgerufen damit die Schichten der Karte gezeichnet werden und damit die Objekte/Gegner erscheinen.

Draw zeichnet durch Schleifen und der Angabe der Kartenhöhe und Breite die einzelnen eingelesenen Schichten. Jede Zahl hat eine Textur welche dann gezeichnet wird. Durch die Animationsobjekte können auch bewegende Objekte gezeichnet werden (Wasser, Fackeln). Bestimmte Zahlen im Layer lassen auch Gegner oder andere Objekte erscheinen (z.B. Dynamit oder Kisten).

+ ReadLayer()

ReadLayer() Wird in **Game1** beim Kartenwechsel aufgerufen damit die Karte eingelesen wird.

readerchar ist die Stelle welche eingelesen wird. Die Variable wird je nach Zeichen hochgezählt. Leerzeichen werden übersprungen ebenso Zeilenwechsel. Der Streamreader bringt die Schichten-Datei in einen Array voraus die Zahlen gelesen werden. ReadLayer funktioniert nur bei Dateien mit 34 Zeilen und 60 durch Leerzeichen getrennte Zahlen.

0 = Keine Kollision 1 = Kollision mit etwas festem (Wand, Stein) 2 = Kollision mit Wasser (Für Projektile)

+ Clear_Layer()

Clear_Layer() Wird in am Anfang von ReadLayer aufgerufen damit das Layer wieder leer ist und neu beschrieben werden kann.

Clear_Layer setzt durch zwei Schleifen MapLayer auf null.

3.6.3 Map_Changer

Die Klasse **Map_Changer** wechselt die Karte und ändert die Position des Spielers.

Attribute

Map_Changer besitzt 8 Attribute(Tabelle).

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
+ string	mapName	Der Name der momentanen Karte
- Vector2	newPosition	Die neue Position des Spielers bei einem MapWechsel
+ Texture2D	blur	Das schwarze was kommt wenn die Karte wechselt.
- enum	ChangeState	Welcher Status der Wechsel hat (Starten, Verlassen, Eingehend).
- ChangeState	changeState	ChangeState als Variable.
- float	time	Wird benötigt um das Schwarz werden des Bildschirms zu verzögern.
- int	alpha	Die Sichtbarkeit der Schwärze.
+ bool	mapLeft	Gibt an ob die Map verlassen wurde oder nicht.

Methoden

Map_Changer besitzt 6 Methode(**checkMapChange**, **changePosition**, **Update**, **DrawBlur**, **setStart**, **mapLoad**). **CheckMapChange** überprüft den Mapwechsel und **changePosition** wechselt die Position im richtigen Moment beim Übergang zur anderen Map. **Update** aktualisiert den Kartenübergang und **DrawBlur** zeichnet den Übergang. **SetStart** setzt alles zum Anfang zurück. **MapLoad** wird für das Laden benutzt.

+ **checkMapChange**(Vector2 playerPosition, int height, int width, Collision _collision, ref bool mapChanged, ObjectManager objectManager, Game1 game)

checkMapChange(..) Wird in **Game1** aufgerufen um zu überprüfen ob ein Mapwechsel stattfindet.

Durch die Angaben der Position und der Größe des Spielers wird der Wechsel überprüft. Dazu wird _collision benötigt. MapChanged gibt an ob die Map gewechselt wird und game wird für Aktionen in **Game1** benötigt. Wenn die Map wechselt wird der Name der momentanen Karte gewechselt und somit auch alle Pfade zu den einzelnen Schichtendateien der Karten.

+ **changePosition**(ref Vector2 posi)

changePosition(..) Wird aufgerufen um den Positionswechsel im richtigen Moment durchzuführen.

Da ein Kartenübergang stattfindet muss die Position des Spielers erst gewechselt werden wenn die neue Karte geladen ist. Dazu wird diese Methode benötigt. Durch posi wird die Position des Spielers verändert.

+ **Update**(GameTime gameTime, Game1 game)

Update(..) aktualisiert die Schwärze beim Übergang.

Update zählt je nach Wechselstatus alpha hoch oder runter. Game wird wieder für aktionen in **Game1** benötigt.

+ **DrawBlur**(SpriteBatch spriteBatch)

DrawBlur(..) zeichnet die Schwärze beim, Übergang.

DrawBlur zeichnet mit Hilfe von alpha die Schwärze und wie stark sie gezeichnet wird.

+ **setStart**()

setStart () setzt alles zum Anfang.

SetStart setzt alle Variablen zum Anfang zurück damit der Mapwechsel beim Laden richtig durchgeführt wird.

+ **mapLoad**(Vector2 loadedPos)

mapLoad () wird beim Laden benötigt.

MapLoad setzt alles so damit beim Laden von Spielständen keine Probleme auftreten.

Ordner:
3.7 Menu

Klassen:
3.7.1 Button
3.7.2 GameOver
3.7.3 Menus
(MainMenu/PauseMenu)

3.7.1 Button

Die Klasse **Button** kann benutzt werden um Buttons im Spiel zu realisieren.

Attribute

Button besitzt 3 Attribute. (Tabelle)

Datentyp	Attributname	Beschreibung
- Rectangle	ButtonRectangle	Der Button selber als Rectangle.
+ Rectangle	TrackRectangle	Nur benötigt für Trackbars wie z.B. Lautstärke.
+ bool	enabled	Ob der Button anklickbar/sichtbar ist oder nicht.

Methoden

Button besitzt drei Methoden (**ButtonClick**, **ButtonHover**, **TrackButton**). **ButtonClick** ist dafür verantwortlich zurückzugeben ob der Button angeklickt wurde oder nicht. **ButtonHover** gibt zurück ob die Maus über dem Button liegt oder nicht (Nützlich für z.B. Texturänderungen). **TrackButton** wird benutzt um das **TrackRectangle** zu verändern und auch dafür zurückzugeben ob die Trackbar verändert wurde. Der Konstruktor setzt das **ButtonRectangle**, also Position und Größe.

+ **ButtonClick**(**Point mousePosition**)

ButtonClick(..) wird überall aufgerufen wo überprüft werden soll ob der jeweilige Button angeklickt wurde oder nicht.

Durch die Angabe der **mousePosition** wird angegeben an welchem Punkt sich die Maus befindet. Durch die Angabe der Position der Maus kann man dann mithilfe des **ButtonRectangle**'s überprüfen ob sich die Maus im jeweiligen Button beim klicken befindet. Sollte sich die Maus beim Klicken auf dem Button befinden gibt die Methode **true** aus ansonsten gibt sie **false** aus.

+ **ButtonHover**(**Point mousePosition**)

ButtonHover(..) wird aufgerufen wenn man überprüfen will ob der Mauszeiger über dem Button liegt. Nützlich für Texturänderungen wie im Hauptmenu.

Durch die Angabe der **mousePosition** wird angegeben an welchem Punkt sich die Maus befindet. Durch die Angabe der Position der Maus kann man dann mithilfe des **ButtonRectangle**'s überprüfen ob sich die Maus über dem jeweiligen Button befindet. Sollte sich die Maus über dem Button befinden gibt die Methode **true** aus ansonsten gibt sie **false** aus.

+ **TrackButton**(**Point mousePosition**)

TrackButton(..) wird aufgerufen um das **TrackRectangle** beim ziehen über die Trackbar zu verändern und um zu überprüfen ob die Trackbar überhaupt angeklickt wird.

Durch die Angabe der **mousePosition** wird angegeben an welchem Punkt sich die Maus befindet. Durch diese Angabe wird das **TrackRectangle** verändert wenn sich die Maus darüber befindet. Nach Veränderung des **TrackRectangle** gibt die Methode **true** aus. Sollte die Maus sich nicht über der Trackbar befinden bleibt das **Rectangle** gleich und die Methode gibt **false** aus.

Rectangle = Rechteck

Button = Knopf

Trackbar = Ziehleiste

3.7.2 Game Over

Die Klasse **GameOver** verwaltet den GameOver-Screen beim sterben im Spiel.

Attribute

GameOver besitzt 10 Attribute. (Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
+ Texture2D	Button	Textur der Buttons
- Button	Load	Objekt der Button -Klasse
- Button	Quit	Objekt der Button -Klasse
- Button	BackMainMenu	Objekt der Button -Klasse
- string[18]	lines	Übersetzungen der Buttonbeschriftung.
- enum	Language	Speichert die momentan ausgewählte Sprache.
- Language	language	Language als Variable.
- LoadLanguage	LanguageLoader	Objekt der Klasse LanguageLoader. Es lädt die Übersetzungen.
- string	oldLanguage	Speichert die alte Sprache zwischen.
- int	mitte	Die Mitte des Spielebildes.

Methoden

GameOver besitzt vier Methoden(**Update**, **ClickCheck**, **Draw**, **disableButtons**). **Update** ist dafür verantwortlich den GameOver-Screen zu aktualisieren. **Draw** zeichnet den ganzen Screen. **ClickCheck** wird benutzt um zu überprüfen welche Buttons angeklickt werden und was dann passieren soll. **DisableButtons** deaktiviert ganz simpel alle Buttons.

+ **Update**(string lan)

Update(..) wird in **Game1** aufgerufen sobald der Spieler stirbt.

Update aktualisiert nur die Sprache falls diese sich ändern sollte und aktiviert ebenfalls auch die Buttons.

+ **Draw**(SpriteBatch spriteBatch, Point mousePosition, SpriteFont font)

Draw(..) wird in **Game1** aufgerufen sobald der Spieler stirbt.

Draw zeichnet das Menu des GameOverscreens. Es aktualisiert ebenfalls auch die Texturen beim drüberziehen über die Button.

+ `ClickCheck`(int sv, Point mousePosition, Game1 game)

`ClickCheck(..)` wird in `Game1` ebenfalls aufgerufen sobald der Spieler stirbt,

`ClickCheck` bekommt durch `sv` die Nummer der `SaveFile` falls man neu laden will. `MousePosition` gibt die Position der Maus an damit der Klick überprüft werden kann. `Game` wird benutzt um bestimmte Methoden aus der `MainKlasse` aufzurufen. `ClickCheck` verwalten im Endeffekt die Klicks im `GameOver-Screen`.

+ `disableButtons`()

`disableButtons()` wird in `ClickCheck` aufgerufen damit falls der `GameOver-Screen` beendet werden soll die Buttons aus sind und nichtmehr aufgerufen werden können.

Screen = Bildschirm

SaveFile = Speicherstand

3.7.3 Menus (PauseMenu/MainMenu)

Die Klasse **MainMenu** beinhaltet alle Buttons und Aktionen im MainMenu.

Attribute

MainMenu besitzt 41 Attribute.(Tabelle)

Datentyp	Attributname	Beschreibung
- Button	Load, newGame,exitGame, options, back, save1, save2, save3, load1, load2, load3, Resright, Resleft, Winright, Winleft, graphicChanges, volume, Yes, No, german, english	Alles Objekte der Button -Klasse.
- int	saveFile	Nummer der SaveFile.
- enum	MenuState	Speichert den Menustatus.
- MenuState	menuState	MenuState als Variable
- enum	Resolution	Speichert die Auflösungen.
- Resolution	resolution	Resolution als Variable
- enum	WindowState	Speichert den Status des Fensters.
- WindowState	windowState	WindowState als Variable
+ Texture2D	ButtonTexture, arrowLeft, arrowRight, menuBackground, de, en, Trackbar, Logo, messagebox	Die Texturen der einzelnen Buttons.
- int	mitte	Die Mitte des Spielebildschirms.
- string[18]	lines	Übersetzungen der Buttonbeschriftung.
- enum	Language	Speichert die momentan ausgewählte Sprache.
- Language	language	Language als Variable.
- LoadLanguage	LanguageLoader	Objekt der Klasse LanguageLoader. Es lädt die Übersetzungen.

Methoden

Button besitzt zehn Methoden(**draw**, **update**, **clickCheck**, **backMainMenu**, **backClick**, **disableButtons**, **config**, **confirm**, **windowString**, **resolutionString**). Draw zeichnet das momentane Menu und update aktualisiert dieses. ClickCheck wird benutzt um zu überprüfen welche Buttons angeklickt werden und was dann passieren soll. BackMainMenu verändert den Menustatus zum Hauptmenu. BackClick aktualisiert die Buttons zum Hauptmenu. DisableButtons deaktiviert alle Buttons. Config verändert durch die geladenen Einstellungen die Schrift auf den Buttons in den Einstellungen. Confirm wird aufgerufen wenn etwas bestätigt werden soll. WindowString und ResolutionString geben ihre Einstellung als string zurück damit dieser string in den Einstellungen gezeichnet werden kann.

+ `Update`(string lan)

`Update`(..) wird in `Game1` aufgerufen sobald der Spieler stirbt.

`Update` aktualisiert nur die Sprache falls diese sich ändern sollte und aktiviert ebenfalls auch die Buttons.

+ `Draw`(SpriteBatch spriteBatch, Point mousePosition, SpriteFont font)

`Draw`(..) wird in `Game1` aufgerufen sobald der Spieler stirbt.

`Draw` zeichnet das Menu des GameOverscreens. Es aktualisiert ebenfalls auch die Texturen beim drüberziehen über die Button.

+ `ClickCheck`(int sv, Point mousePosition, Game1 game)

`ClickCheck`(..) wird in `Game1` ebenfalls aufgerufen sobald der Spieler stirbt,

`ClickCheck` bekommt durch `sv` die Nummer der SaveFile falls man neu laden will. `MousePosition` gibt die Position der Maus an damit der Klick überprüft werden kann. `Game` wird benutzt um bestimmte Methoden aus der MainKlasse aufzurufen. `ClickCheck` verwaltet im Endeffekt die Klicks im GameOver-Screen.

+ `backMainMenu`()

`backMainMenu`(..) wird benutzt um den Menustate auf den Hauptmenu-State zu setzen.

+ `backClick`()

`backClick`() wird aufgerufen um die Buttons zu aktivieren welche im Hauptmenu benötigt werden.

+ `disableButtons`()

`disableButtons`() wird aufgerufen um alle Buttons zu deaktivieren.

+ `config`(string loadedWidth, string windowMode)

`config`(..) wird aufgerufen um die States des Fensters und der Auflösung zu setzen.

+ `confirm`(string loadedWidth, string windowMode)

`config`(..) wird aufgerufen um abzufragen ob die Aktion wirklich durchgeführt werden soll.

+ `windowString()`

`windowString()` gibt anhand des Fensterzustandes den jeweiligen Begriff zurück.

+ `resolutionString()`

`resolutionString()` gibt anhand des Auflösungszustandes die jeweilige Auflösung zurück.

State = Zustand/Status

3.7.3 Menus (PauseMenu/MainMenu)

Die Klasse `PauseMenu` ist eine leichte Veränderung des Mainmenus und wird deswegen nicht aufgeführt. `PauseMenu` dazu da das Menu beim Pausieren darzustellen.

Die einzig neue Methode in dieser Klasse ist `lang(string lan)` welche die Sprache bei Veränderung einliest.

+ `lang(string lan)`

`Lang(..)` liest Sprache ein und setzt den Sprachenstatus zu dieser.

Ordner:
3.8 NPC Manager

Klassen:
3.8.1 NPCManager
3.8.2 NPC(Shop/Tutorial)
3.8.3 Textbox

3.8.1 NPCManager

Die Klasse **NPCManager** verwaltet alle **NPC**(non-player charakter) Objekte sowie Texturen.

Attribute

NPCManager besitzt für jede Klasse(**Tutorial** und **Shop**) eine **List**.(Tabelle)

<u>Klassenname</u>	<u>Listenname</u>
NPC	
Tutorial : NPC	+ lstNPCTutorial
Shop : NPC	+ lstNPCShop

Methoden

Um ein NPC zu erstellen hat **NPCManager** für jede Liste eine **Spawn...** Methoden. Alle haben als erst Parameter ein **Vector2 Position**. Ebenso wird im Methodenrumpf mindestens einmal die **List.Add(new Klasse())** aufgerufen um ein neues Objekt in die dazugehörige Liste hinzufügen.

<u>Methodenname</u>	<u>Beschreibung</u>
+ SpawnShopNPC (Vector2 position)	Es wird ein neues Shop Objekt der lstNPCShop hinzugefügt.
+ SpawnTutorialNPC (Vector2 position)	Es wird ein neues Tutorial Objekt der lstNPCTutorial hinzugefügt.

+ **Update**(GameTime **gameTime**)

In der **Update** Methode werden alle Objekte die in den Listen enthalten sind aktualisiert.(gilt für alle)

lstNPCTutorial

Jedes Objekt in **lstNPCTutorial** wird aktualisiert.

lstNPCShop

Jedes Objekt in **lstNPCShop** wird aktualisiert.

+ **Draw**(SpriteBatch **spriteBatch**, GameTime **gameTime**)

In der **Draw** Methode werden alle Objekte die in den Listen enthalten sind gezeichnet.

Dazu bekommt jede Liste eine eigene For-Schleife die durch alle List Objekte durchgeht und die dazugehörigen **List.Draw()** aufruft. Somit wird jedes Objekt mithilfe von **spriteBatch** gezeichnet.

3.8.2 NPC

Die Klasse **NPC** ist die Basisklasse von allen NPCs, sie besitzt die Subklasse: **Shop** und **Tutorial**.

Attribute

NPC besitzt 3 Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Texture2D	texture	Speichert die NPC Textur.
- SpriteFont	font	Schriftart.
- Vector2	position	Koordinaten(X,Y) für die Position.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Textbox (Texture2D _texture , Vector2 _position , SpriteFont _font)	Standard Konstruktor der Textbox Klasse. Als Parameter wird _text , _position und _coinSize _font .
+ Update (GameTime gameTime)	Zuerst wird time aktualisiert. Dann wird geprüft ob time größer als 0.1 ist. Wenn es zutrifft, wird time auf 0 gesetzt. Ebenso wird überprüft ob i kleiner als text.Length (Text Länge) ist, demnach wird zu temp der char, der an Stelle i von text ist, addiert.
+ virtual Draw (SpriteBatch spriteBatch, GameTime gameTime)	Zeichnet temp auf position

3.8.2 ShopNPC : NPC

Die Klasse **ShopNPC** stellt einen Shop dar in dem man seine Klasse wechseln kann und in dem man seine Klasse wechseln kann.

Attribute

Klassenname besitzt 5Attribute.(Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- HUD	hud	Wird für bestimmte Texturen aus der Klasse benötigt.
- Button	upgrade1, upgrade2, upgrade3, switchClass1, switchClass2, switchClass3	Alle Buttons vom Shop.
- Player	player	Die Spielerklasse für die Werte
- Animations	animations	Wird für die Münzbewegung gebraucht.
- enum	Spieler	Die Spielerklassen wie aus Game1
- Spieler	playerClass	Spieler als Variable
- int	priceArcher1, priceArcher2, priceArcher3	Preise der einzelnen Fähigkeiten des Bogenschützen.
- int	priceWizard1, priceWizard2, priceWizard3	Preise der einzelnen Fähigkeiten des Zauberers.
- int	priceKnight1, priceKnight2, priceKnight3	Preise der einzelnen Fähigkeiten des Ritters.

Methoden

ShopNPC besitzt drei Methode(**Update**, **clickCheck**, **Draw**). Update aktualisiert die Preise und die Buttons. ClickCheck überprüft ob einer der Buttons angeklickt wurde. Draw zeichnet den NPC, die Buttons und die Preise.

+ **Update**(GameTime gameTime, int _playerClass, Player _player)

Update(..)wird im NPC Manager aufgerufen und aktualisiert den NPC.

Update überprüft welche Knöpfe aktiv sind wieviel die Upgrades kosten und welche Klasse ausgewählt ist. _playerClass gibt die Klasse an und _player wird für die Fähigkeiten benötigt.

+ **clickCheck**(Point mousePosition, ref Player player, Game1 game)

clickCheck(..)wird im NPC Manager aufgerufen und überprüft ob Buttons angeklickt wurden und was dann geschehen soll.

Durch die Angabe der Mausposition kann der Knopfdruck ermittelt werden. Player wird für die Fähigkeiten benutzt und game wird gebraucht um die Klasse zu ändern.

+ **Draw**(SpriteBatch spriteBatch, GameTime gameTime, Point mousePosition, SpriteFont font)

Draw(..) wird im NPC Manager aufgerufen und sie zeichnet den ganzen Shop-NPC.

MousePosition wird gebraucht um zu überprüfen ob die Maus über einem Knopf liegt. Font wird für die Preise gebraucht und spriteBatch um alles zeichnen zu können. Draw zeichnet alle Knöpfe, die Preise und den NPC selbst.

3.8.2 TutorialNPC : NPC

Das Objekt der Klasse **TutorialNPC** erzeugt ein NPC der mithilfe der Klasse **Textbox** eine kurze Anleitung des Spiels erzeugt.

Attribute

Klassenname besitzt 5 Attribute. (Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- float	time	Zeit wie lange die Nachricht bleibt.
- int	auswahl = 1	Auswahl welcher Text ausgegeben wird.
- List< Textbox >	lstTextbox	Liste aus Textboxen

Methoden

ShopNPC besitzt drei Methode (**Update**, **clickCheck**, **Draw**). Update aktualisiert die Preise und die Buttons. ClickCheck überprüft ob einer der Buttons angeklickt wurde. Draw zeichnet den NPC, die Buttons und die Preise.

+ **Update**(GameTime gameTime, int language)

Update wird im NPC Manager aufgerufen und aktualisiert den NPC.

Zuerst wird **time** mit der vergangenen Zeit hochgezählt. Danach wird überprüft welchen Wert **auswahl** hat. Nun wird überprüft ob **lstTextbox.Count = 0** ist, so dass immer nur eine Textbox gezeichnet wird. Ein Textbox Objekt wird mit den gegebenen Parametern in **lstTextbox** hinzugefügt. Nach 3.5 Sekunden wird das Objekt entfernt, **time** auf 0 gesetzt sowie **auswahl** auf 2 gesetzt sodass die nächste Textbox geladen wird. Nach der letzten Textbox(7) wird **auswahl** auf 1 gesetzt, sodass eine Endlosschleife entsteht.

+ **override Draw**(SpriteBatch spriteBatch, GameTime gameTime)

In der Draw Methode wird der NPC gezeichnet sowie die Textbox.

3.8.3 Textbox

Das Objekt der Klasse **Textbox** zeichnet einen definierten Text. Wird von **Shop** und **Tutorial** aufgerufen.

Attribute

Textbox besitzt 6 Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- string	text	Speichert den ganzen Text.
- string	Temp = „“	Speichert ein Teil des Textes.
- Vector2	position	Koordinaten(X,Y) für die Position.
- SpriteFont	font	Schriftart
- float	time = 0f	Mithilfe von gameTime wird die vergangene Zeit (seit dem letztem Update Aufruf) addiert
- int	i = 0	Zähler

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Textbox (string _text , Vector2 _position , SpriteFont _font)	Standard Konstruktor der Textbox Klasse. Als Parameter wird _text , _position und _font übergeben.
+ Update (GameTime gameTime)	Zuerst wird time aktualisiert. Dann wird geprüft ob time größer als 0.1 ist. Wenn es zutrifft, wird time auf 0 gesetzt. Ebenso wird überprüft ob i kleiner als text.Length (Text Länge) ist, demnach wird zu temp der char, der an Stelle i von text ist, addiert.
+ virtual Draw (SpriteBatch spriteBatch, GameTime gameTime)	Zeichnet temp auf position

Ordner:

3.9 Objekt Manager

Klassen:

3.9.1 ObjectManager

3.9.2 InteractiveObjects

3.9.3 PickUpItems

3.9.1 ObjectManager

Die Klasse **ObjectManager** verwaltet alle **InteractiveObjects** und **PickUpItems** Objekte sowie Texturen.

Attribute

ObjectManager besitzt für jede Klasse eine **List** der dazugehörigen Klasse.(Tabelle)

Klassenname	Listenname
abstract InteractiveObjects	
Dynamit : InteractiveObjects	+ lstDynamit
Trap : InteractiveObjects	+ lstTrap
Chest : InteractiveObjects	+ lstChest
ArrowTrap : InteractiveObjects	+ lstArrowTrap
PickUpItems	
Coin : PickUpItems	+ lstCoin
Experience : PickUpItems	+ lstExperience
HealthPotionSmall : PickUpItems	+ lstHealthPotionSmall
HealthPotionBig : PickUpItems	+ lstHealthPotionBig
Hourglass : PickUpItems	+ lstHourglass
ShotEngine	+ lstDynamitArrowTrapBullets

Methoden

Um ein bestimmtes Objekt zu erstellen hat **ObjectManager** für jede Liste eine **Spawn...** Methoden. Bis auf ein paar Ausnahmen, sind die Methoden gleich aufgebaut. Alle haben als erst Parameter ein **Vector2 position**. Ebenso wird im Methodenrumpf mindestens einmal die **List.Add(new Klasse())** aufgerufen um ein neues Objekt in die dazugehörige Liste hinzufügen.

Methodenname	Beschreibung
+ SpawnCoinXP (Vector2 position)	Mit einer Wahrscheinlichkeit von 20% wird ein Coin Object mit der dazugehörigen Texture sowie Position und Größe in lstCoin hinzugefügt. Das gleiche gilt für die HealthPotionSmall nur das hier eine Wahrscheinlichkeit von 10% anliegt und der lstHealthPotionSmall hinzugefügt wird.
+ SpawnChestItem (Vector2 position)	Es wird 10 mal ein Coin Object der lstCoin hinzugefügt, ebenso wird der einsammle Radius verkleinert, damit der Spieler nicht direkt interagiert. Außerdem wird ein HealthPotionBig Object der lstHealthPotionBig hinzugefügt.
+ SpawnHourglass (Vector2 position)	Hinzufügen eines Hourglass Object in lstHourglass .
+ SpawnChest (Vector2 position)	Hinzufügen eines Chest Object in lstChest .

+ <code>SpawnTrap(Vector2 position)</code>	Hinzufügen eines <code>Trap</code> Object in <code>lstTrap</code>
+ <code>SpawnDynamit(Vector2 position)</code>	Hinzufügen eines <code>Dynamit</code> Object in <code>lstDynamit</code> .
+ <code>SpawnArrowTrap(Vector2 position, int direction)</code>	Hinzufügen eines <code>Hourglass</code> Object in <code>lstHourglass</code> . Der extra Parameter <code>direction</code> gibt an in welche Richtung das Objekt zeigt. 0 → oben, 1 → rechts, 2 → unten und 3 → links.(Uhrzeigersinn)
+ <code>ObjectDelete()</code>	Entfernt alle Objekte die in <code>lstDynamit</code> , <code>lstChest</code> , <code>lstTrap</code> , <code>lstArrowTrap</code> und <code>lstHourglass</code> enthalten sind.

+ `Update(GameTime gameTime, Vector2 playerposition, Player player, int currentSpieler, ParticleManager particleManager, Abilities abilities)`

In der `Update` Methode werden alle Objekte die in den Listen enthalten sind aktualisiert.(gilt für alle)

`lstDynamit`

Für jedes `Dynamit` Objekt wird geschaut ob `Explosion`(true) und `Exploded`(false) ist. Wenn dies der Fall ist, werden 36 `ShotAnimation` Objekte der `lstDynamitArrowTrapBullets` hinzugefügt. Mithilfe von `Math.Sin` und `Math.Cos` können die `ShotAnimation` Objekte in einem Kreis angeordnet werden und somit in allen Richtungen verbreitet werden. Wenn alle 36 Objekte erstellt wurden, wird `Exploded` auf true gesetzt und `Explosion` auf false damit sich das `Dynamit` Objekt nicht mehrmals explodiert.

`lstArrowTrap`

Für jedes `ArrowTrap` Objekt wird geschaut ob `Cooldown` größer gleich 1 ist. Wenn dies der Fall ist, wird `Direction` abgefragt und es wird in die bestimmte Richtung ein neues `ShotAnimation` Objekt der `lstDynamitArrowTrapBullets` hinzugefügt. Somit wird jede Sekunde ein neues `ShotAnimation` Objekt erstellt.

`lstChest`

Für jedes `Chest` Objekt wird mithilfe von `Intersects` geprüft ob sich der Spieler mit dem `Chest` Objekt kollidiert und `Opened` auf false ist. Wenn es zutrifft wird `Opened` auf true gesetzt damit sich die aktuelle Chest nur einmal öffnen lässt. Ebenso wird `SpawnChestItem`(aktuelle Chest Position) aufgerufen.

`lstTrap`

Für jedes `Trap` Objekt wird mithilfe von `Intersects` geprüft ob sich der Spieler mit dem `Trap` Objekt kollidiert und `Cooldown` größer gleich 0,5 ist. Wenn es zutrifft wird `player.LebenAbziehen(10, currentSpieler)` aufgerufen. Ebenso wird `Cooldown` auf 0 gesetzt somit wieder 0,5 Sekunden gewarten werden muss, damit der Spieler Schaden bekommt.

`lstDynamitArrowTrapBullets`

Jedes Objekt in `lstDynamitArrowTrapBullets` wird aktualisiert.

+ `Draw(SpriteBatch spriteBatch, GameTime gameTime)`

In der `Draw` Methode werden alle Objekte die in den Listen enthalten sind gezeichnet.

Dazu bekommt jede Liste eine eigene For-Schleife die durch alle List Objekte durchgeht und die dazugehörigen List.`Draw()` aufruft. Somit wird jedes Objekt mithilfe von `spriteBatch` gezeichnet.

3.9.2 InteractiveObjects

Die abstrakte Klasse **InteractiveObjects** ist die Basisklasse von allen Interaktiven Objekten, sie besitzt die Subklassen: **Dynamit**, **Trap**, **Chest** und **ArrowTrap**.

Attribute

InteractiveObjects besitzt 2 Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
# Texture2D	texture	Speichert die Textur.
# Vector2	position	Koordinaten(X,Y) für die Position.

Methoden

InteractiveObjects hat wie jede andere Klasse eine **Update** sowie **Draw** Methode mit der Besonderheit das beide abstrakt sind. Ebenso besitzt die Klasse einen Konstruktor für die Übergabe der beiden Attribute(**texture**, **position**).

<u>Methodenname</u>	<u>Beschreibung</u>
+ InteractiveObjects (Texture 2D _texture , Vector2 _position)	Standard Konstruktor der InteractiveObjects Klasse. Als Parameter wird _texture und _position übergeben.
+ abstract Update (GameTime gameTime)	Die Update Methode ist abstrakt.
+ abstract Draw (SpriteBatch spriteBatch, GameTime gameTime)	Die Draw Methode ist abstrakt.

Dynamit : InteractiveObjects

Die Klasse **Dynamit** erzeugt eine Explosion wenn der Spieler damit Interagiert(Schuss).

Attribute

Dynamit besitzt 6 extra Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- bool	exploded	Wird true wenn das Dynamit Objekt ausgelöst <u>wurde</u> . Für die Texturen.
- bool	explosion	Wird true wenn das Dynamit Objekt ausgelöst <u>wird</u> . Für die Explosion.
- Texture2D	dynamitexplode1	Textur 1 nach der Explosion.
- Texture2D	dynamitexplode2	Textur 2 nach der Explosion.
- Texture2D	dynamitexplode3	Textur 3 nach der Explosion.
- int	auswahl	Gibt an welche Explosion Textur gewählt wird.(1, 2, 3).

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Dynamit (Texture2D dynamit , Texture2D _dynamitexplode1 , Texture2D _dynamitexplode2 , Texture2D _dynamitexplode3 , Vector2 position , int Auswahl) base: (texture , position)	Übergibt der Basisklasse texture sowie position .
+ override Update (GameTime gameTime)	
+ override Draw (SpriteBatch spriteBatch, GameTime gameTime)	Die Draw Methode wird Überschrieben. Je nach auswahl wird eine andere Textur gezeichnet. Wenn auswahl gleich 0 ist wird base. texture gezeichnet.

Trap : InteractiveObjects

Die Klasse **Trap** fügt dem Spieler Schaden zu, wenn der Spieler damit Interagiert(Position, Größe)

Attribute

Trap besitzt 2 extra Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Animations	animations	Zeichnet die Animation.
- float	cooldown	Mithilfe von gameTime wird die vergangene Zeit (seit dem letztem Update Aufruf) addiert

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Trap (Texture 2D texture , Vector2 position) base: (texture , position)	Übergibt der Basisklasse texture sowie position .
+ override Update (GameTime gameTime)	Die Update Methode wird Überschrieben. Solange cooldown kleiner gleich 0.5 ist, wird die vergangene Zeit(des letzten Update Aufrufs) mithilfe von gameTime addiert.
+ override Draw (SpriteBatch spriteBatch, gameTime gameTime)	Die Draw Methode wird Überschrieben. Mithilfe von animations wird die drawanimation Methode aufgerufen, welche jede Sekunde 12 mal das Frame ändert.

Chest : InteractiveObjects

Die Klasse **Chest** lässt eine Kiste erscheinen. Wenn der Spieler sich der Kiste nähert öffnet sich die Kiste und lässt Gegenstände(**Coin**, **HealthPotionBig**) fallen.

Attribute

Chest besitzt 2 extra Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- bool	opened	Gibt an ob das Chest Objekt offen bzw. zu ist.
- Texture2D	close	Textur Kiste zu.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Chest (Texture2D texture , Texture2D textureclose , Vector2 position) base: (texture , position)	Übergibt der Basisklasse texture sowie position . Ebenso wird close gesetzt.
+ override Update (GameTime gameTime)	
+ override Draw (SpriteBatch spriteBatch, GameTime gameTime)	Die Draw Methode wird Überschrieben. Wenn opened auf true ist, wird mit spriteBatch.Draw base. Texture und base. Position , die offene Kiste gezeichnet. Wenn opened false ist wird close als Texture verwendet.

ArrowTrap : InteractiveObjects

Die Klasse **ArrowTrap** lässt eine Schuss Falle erscheinen. Sekündlich wird ein Pfeil in einer der angegebenen Richtung geschossen.

Attribute

ArrowTrap besitzt 6 extra Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- int	direction	Gibt an in welche Richtung das ArrowTrap Objekt gerichtet ist.(0 = oben, 1 = rechts, 2 = unten, 3 = links)
- float	cooldown	Mithilfe von GameTime wird die vergangene Zeit (seit dem letztem Update Aufruf) addiert
- Texture2D	textureUpLoaded	Textur mit Pfeil nach oben
- Texture2D	textureUpUnloaded	Textur ohne Pfeil nach oben
- Texture2D	textureRightLoaded	Textur mit Pfeil nach rechts
- Texture2D	textureRightUnloaded	Textur ohne Pfeil nach rechts

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ ArrowTrap (Texture2D _textureUpLoaded , Texture2D _textureUpUnloaded , Texture2D _textureRightLoaded , Texture2D _textureRightUnloaded , Vector2 position , int _direction) base: (_textureUpLoaded , position)	Übergibt der Basisklasse _textureUpLoaded sowie position . Ebenso werden alle anderen Texturen sowie die direction gesetzt.
+ override Update (GameTime gameTime)	Die Update Methode wird Überschrieben. Solange cooldown kleiner gleich 1 ist, wird die vergangene Zeit(des letzten Update Aufrufs) mithilfe von gameTime addiert.
+ override Draw (SpriteBatch spriteBatch, GameTime gameTime)	Die Draw Methode wird Überschrieben. Wenn cooldown zwischen 0 und 0.5 ist wird die Textur ...Unloaded verwendet. Mithilfe von direction kann nun bestimmt werden in welche Richtung gezeichnet wird. Mit SpriteEffects.FlipVertically/FlipHorizontally kann die Textur gespiegelt(gedreht) werde. Das selbe gilt bei einem cooldown Wert zwischen 0.5 und 1, nur das nun ...Loaded als Textur verwendet wird.

3.9.3 PickupItems

Die Klasse **PickUpItems** ist die Basisklasse von allen Gegenstände die man einsammeln kann, sie besitzt die Subklassen: **Coin**, **Experience**, **HealthPotionSmall**, **HealthPotionBig** und **Hourglass**.

Attribute

PickUpItems besitzt 10 Attribute.

Datentyp	Attributname	Beschreibung
- Animations	animations	Zeichnet die Animation.
- Texture2D	itemTexture	Item Texture.
- Vector2	itemPosition	Koordinaten(X,Y) für die Item Position.
- Vector2	itemSize	Größe des Items(Width, Height).
- Vector2	directionXY	Richtung in der sich das Item bei der Interaktion mit dem Spieler bewegt.
- int	Radius = 100	Einsammle Radius.
- int	speed = 8	Item einsammle Geschwindigkeit.
- int	height = 0	Item Hoch und Runter Bewegung
- bool	collected	Auf true wenn Item eingesammelt
- bool	down	true = runter, false = hoch

Methoden

PickUpItems hat wie jede andere Klasse eine **Update** sowie **Draw** Methode, mit der Besonderheit das die **Draw** Methode als virtual gekennzeichnet ist und der Rumpf leer ist.

Methodenname	Beschreibung
+ PickUpItems (Texture 2D _coinTexture , Vector2 _coinPosition , Vector2 _coinSize)	Standard Konstruktor der PickUpItems Klasse. Als Parameter wird _coinTexture , _coinPosition und _coinSize übergeben.
+ Update (GameTime gameTime)	Als erstes wird directionXY aktualisiert indem man die Spieler Position(_playerPosition) mit itemPosition subtrahiert, dadurch hat man nun die Richtung in dem sich das Item bewegt, wenn sich der Spieler in dem bestimmten „Radius“(Rechteck) befindet. Demnach wird die directionXY mit speed multipliziert und die itemPosition aktualisiert. Wenn der Spieler sich allerdings nicht im „Radius“ befindet, wird das Item Hoch und Runter bewegt. Dies wird mit height und down realisiert.
+ virtual Draw (SpriteBatch spriteBatch, GameTime gameTime)	

Coin : PickUpItems

Das Objekt der Klasse **Coin** lässt eine Münze erscheinen.

Attribute

PickUpItems besitzt kein extra Attribute.

Methoden

Coin überschreibt die **Draw** Methode und **base.Update** wird verwendet.

Methodenname	Beschreibung
+ Coin (Texture 2D _coinTexture , Vector2 _coinPosition , Vector2 _coinSize) : base(_coinTexture, _coinPosition, _coinSize)	Standard Konstruktor der Coin Klasse. Als Parameter wird _coinTexture , _coinPosition und _coinSize übergeben.
+ override Draw (SpriteBatch spriteBatch, gameTime gameTime)	Mithilfe von base.animations.drawanimation() wird base.ItemTexture als Animation gezeichnet.

Experience : PickUpItems

Das Objekt der Klasse **Experience** lässt eine Erfahrungskugel erscheinen, die beim wiederholten Einsammeln das Level des Spielers erhöht.

Attribute

Experience besitzt kein extra Attribute.

Methoden

Experience überschreibt die **Draw** Methode und **base.Update** wird verwendet.

Methodenname	Beschreibung
+ Experience (Texture 2D _EXPTTexture , Vector2 _EXPPosition , Vector2 _EXPSize) : base(_coinTexture, _coinPosition, _coinSize)	Standard Konstruktor der Experience Klasse. Als Parameter wird _EXPTTexture , _EXPPosition und _EXPSize übergeben.
+ override Draw (SpriteBatch spriteBatch, gameTime gameTime)	Mithilfe von base.animations.drawanimation() wird base.ItemTexture als Animation gezeichnet.

HealthPotionSmall : PickupItems

Das Objekt der Klasse **HealthPotionSmall** lässt kleines Essen erscheinen, die dem Spieler Lebenspunkte hinzufügt.

Attribute

HealthPotionSmall besitzt 3 extra Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Random	random	Random Objekts
- Texture2D	potionTexture2	Item Texture2 .
- int	zufall	Speichert die Zufallszahl

Methoden

HealthPotionSmall überschreibt die **Draw** Methode und base.**Update** wird verwendet.

<u>Methodenname</u>	<u>Beschreibung</u>
+ HealthPotionSmall (Texture 2D _PotionTexture1 , Texture 2D _PotionTexture2 , Vector2 _PotionPosition , Vector2 _PotionSize): base(_PotionTexture , _PotionPosition , _PotionSize)	Standard Konstruktor der HealthPotionSmall Klasse. Als Parameter der Basisklasse wird _PotionTexture , _PotionPosition und _PotionSize übergeben. Mithilfe von random.Next() wird zufall einen Wert zugewiesen.
+ override Draw (SpriteBatch spriteBatch, GameTime gameTime)	Wenn zufall gleich 0 ist wird mithilfe von spriteBatch. Draw() base. ItemTexture auf base. ItemPosition gezeichnet. Wenn zufall nicht 0 ist, wird potionTexture2 gezeichnet.

HealthPotionBig : PickupItems

Das Objekt der Klasse **HealthPotionBig** lässt großes Essen erscheinen, die dem Spieler Lebenspunkte hinzufügt.

Attribute

HealthPotionBig besitzt 3 extra Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Random	random	Random Objekts
- Texture2D	PotionTexture2	Item Texture2 .
- int	zufall	Speichert die Zufallszahl

Methoden

HealthPotionBig überschreibt die **Draw** Methode und **base.Update** wird verwendet.

<u>Methodenname</u>	<u>Beschreibung</u>
+ HealthPotionBig (Texture 2D _PotionTexture , Vector2 _PotionPosition , Vector2 _PotionSize) : base (_PotionTexture , _PotionPosition , _PotionSize)	Standard Konstruktor der HealthPotionBig Klasse. Als Parameter wird _PotionTexture , _PotionPosition und _PotionSize übergeben. Mithilfe von random.Next() wird zufall einen Wert zugewiesen.
+ override Draw (SpriteBatch spriteBatch, GameTime gameTime)	Wenn zufall gleich 0 ist wird mithilfe von spriteBatch. Draw() base.ItemTexture auf base.ItemPosition gezeichnet. Wenn zufall nicht 0 ist, wird potionTexture2 gezeichnet.

Hourglass : PickupItems

Das Objekt der Klasse **Hourglass** lässt ein Item erscheinen, das dem Spieler die Cooldown der Abilities auf Maximal setzt.

Attribute

Hourglass besitzt keine extra Attribute.

Methoden

Hourglass überschreibt die **Draw** Methode und **base.Update** wird verwendet.

<u>Methodenname</u>	<u>Beschreibung</u>
+ Hourglass (Texture 2D _hourglass , Vector2 _position , Vector2 _size) : base (_hourglass , _position , _size)	Standard Konstruktor der Hourglass Klasse. Als Parameter wird _hourglass , _position und _size übergeben.
+ override Draw (SpriteBatch spriteBatch, GameTime gameTime)	Mithilfe von spriteBatch. Draw() wird base.ItemTexture auf base.ItemPosition gezeichnet.

Ordner:

3.10 Partikel Manager

Klassen:

3.10.1 ParticleManager

3.10.2 Particle

3.10.1 ParticleManager

Die Klasse **ParticleManager** verwaltet alle **Particle** Objekte sowie Texturen und Fonts.

Attribute

ParticleManager besitzt für jede **Particle** Klasse eine **List** .(Tabelle)

<u>Klassenname</u>	<u>Listenname</u>
abstract Particle	
HitParticle : Particle	+ lstHitParticle
StandardParticle : Particle	+ lstStandardParticle

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Texture2D	standardParticle	Speichert die Textur.
- SpriteFont	font	Schriftart
- Random	random	Random Objekt

Methoden

Um ein bestimmtes Partikel zu erstellen hat **ParticleManager** zwei ähnliche **Spawn...()** Methoden. Beide haben als erst Parameter ein **Vector2 Position**. Ebenso wird im Methodenrumpf mindestens einmal die **Listenname.Add(new Klasse())** Methode aufgerufen um ein neues Objekt in die dazugehörige Liste hinzuzufügen.

<u>Methodenname</u>	<u>Beschreibung</u>
+ SpawnHitParticle (Vector2 position , int dmg , Color color)	Wird verwendet wenn der Spieler oder Gegner Schaden bekommt. Als Parameter wird die Position(position), der Anzeigeschaden(dmg) und die Farbe(color) übergeben. Mit random.Next(0, 3) wird dann Zufällig die Richtung bestimmt in der sich das HitParticle Objekt bewegt.
+ SpawnStandardParticle (Vector2 position)	Wird verwendet wenn sich ein Schuss(ShotEngine Objekt) mit der Collision interagiert. Es wird die Animation Texture übergeben sowie die Position(position).

+ Update(GameTime gameTime)

In der **Update** Methode werden alle Partikel Objekte die in den Listen enthalten sind aktualisiert.(gilt für beide)

lstHitParticle

Für jedes **HitParticle** Objekt wird geschaut ob **Cooldown** zwischen 0,25 und 0,26 ist. Wenn dies der Fall ist, bekommt **HitParticle.zufall** einen neuen **random.Next(0, 3)** Wert. Das Objekt wird erst entfernt wenn **Cooldown** größer als 0,5 ist. Wenn allerdings **Cooldown** nicht größer als 0,5 ist, wird geschaut welcher Wert **zufall** hat und somit in einer bestimmten Richtung Verschieben(**Position**) und Rotiert(**Rotation**).

lstStandardParticle

Für jedes **StandardParticle** Objekt wird geschaut ob **Cooldown** größer als 0,2 ist. Wenn dies der Fall ist, wird das Objekt entfernt.

+ Draw(SpriteBatch spriteBatch, GameTime gameTime)

In der **Draw** Methode werden alle Objekte die in den Listen(**lstHitParticle** und **lstStandardParticle**) enthalten sind gezeichnet.

Dazu bekommt jede Liste eine eigene For-Schleife die durch alle List Objekte durchgeht und die dazugehörigen List.**Draw()** aufruft. Somit wird jedes Partikel Objekt mithilfe von **spriteBatch** gezeichnet.

3.10.2 Particle

Die abstrakte Klasse **Particle** ist die Basisklasse von allen Partikel, sie besitzt die Subklassen: **HitParticle** und **StandardParticle**.

Attribute

Particle besitzt 2 Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
# Vector2	position	Koordinaten(X,Y) für die Position.
# float	cooldown	Mithilfe von gameTime wird die vergangene Zeit (seit dem letztem Update Aufruf) addiert

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ Particle (Vector2 _position)	Standard Konstruktor der Particle Klasse. Als Parameter wird _position übergeben.
+ abstract Update (GameTime gameTime)	
+ abstract Draw (SpriteBatch spriteBatch, gameTime gameTime)	

HitParticle : Particle

Das Objekt der geerbte Klasse **HitParticle** wird erstellt wenn der Gegner oder Spieler Schaden bekommt.

Attribute

Particle besitzt 7 extra Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Random	random	Zufalls Objekt.
- SpriteFont	font	In font wird die Schriftart gespeichert.
- int	zufall	Mithilfe von random wird eine Zufalls Zahl gespeichert.
- float	scale = 1.5f	Größe der Schrift.
- int	damage	Schaden der als Schrift ausgegeben wird.
- float	Rotation = 0	Neigung der Schrift.
- Color	color	Farbe der Schrift.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ HitParticle (SpriteFont _font , Vector2 _position , int _damage , Color _color) : base(_position)	Standard Konstruktor der HitParticle Klasse. Als base Parameter wird _position übergeben.
+ override Update (GameTime gameTime)	Cooldown wird aktualisiert (Mit der Vergangenen Zeit addiert) und Scale (Größe der Schrift) wird mit 0.033 subtrahiert.
+ override Draw (SpriteBatch spriteBatch, GameTime gameTime)	Mithilfe von spriteBatch.DrawString(..) kann nun das Partikel (mit den angegebenen Attributen) gezeichnet werden.

StandardParticle : Particle

Das Objekt der geerbte Klasse **StandardParticle** wird erstellt wenn ein Schuss(**ShotEngine** Objekt) mit der **Collision** kollidiert oder nach einer angegebene Zeit verschwindet.

Attribute

StandardParticle besitzt 2 extra Attribute.

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Animations	animations	Animations Objekt.
- Texture2D	standardTexture	Speichert die Partikel Animation Textur.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ StandardParticle (Texture _texture , Vector2 _position) : base(_position)	Standard Konstruktor der Particle Klasse. Als base Parameter wird _position übergeben.
+ override Update (GameTime gameTime)	Cooldown wird aktualisiert(Mit der Vergangenen Zeit addiert).
+ override Draw (SpriteBatch spriteBatch, GameTime gameTime)	Mithilfe von animations.drawanimation(..) kann nun standardTexture gezeichnet werden. Mit animations.getanimation(..) kann dann genau bestimmt werden wie schnell sich die Animation bewegen soll.(12 Frames per Second: 60 * 0,2(Sekunden bis das Objekt verschwindet)).

Ordner:
3.11 Player

Klassen:
3.11.1 Movement
3.11.2 Player

3.11.1 Movement

Die Klasse **Movement** verwaltet das Bewegen des Spielers.

Attribute

Movement besitzt 5 Attribute. (Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
+ Vector2D	position	Gibt die Position des Spielers an.
+ int	speed	Gibt die Bewegungsgeschwindigkeit des Spielers an.
+ int	playerheight	Gibt die Höhe des Spielers an.
+ int	playerwidth	Gibt die Breite des Spielers an.
- int	pixel	Wird in den Schleifen der Bewegung benutzt um die Bewegung bei der Collision so nah wie möglich an die Wand zu bringen.

Methoden

GameOver besitzt eine Methode(**moving**). Moving bewegt den Spieler.

+ **moving**(**Collision** collisionproof)

moving(..) wird in **Game1** aufgerufen damit der Spieler sich bewegen kann.

Moving kann durch das übergebene Objekt collisionproof überprüfen ob der Spieler beim bewegen gegen eine Wand läuft oder nicht. Der Spieler wird so nah wie möglich an die Wand gebracht.

3.11.2 Player

Die Klasse **Player** verwaltet Level und Lebenspunkte des Spielers.

Attribute

Player besitzt 5 Attribute. (Tabelle)

Datentyp	Attributname	Beschreibung
- int	hpArcher = 100	Bogenschütze aktuelles Leben.
- int	hpWizard = 100	Magier aktuelles Leben.
- int	hpKnight = 150	Ritter aktuelles Leben.
- int	hpArcherMax = 100	Bogenschütze maximales Leben.
- int	hpWizardMax = 100	Magier maximales Leben.
- int	hpKnightMax = 150	Ritter maximales Leben.
- int	xpArcher = 0	Bogenschütze aktuelle Erfahrung.
- int	xpWizard = 0	Magier aktuelle Erfahrung.
- int	xpKnight = 0	Ritter aktuelle Erfahrung.
- int	xpArcherMax = 100	Bogenschütze maximale Erfahrung.
- int	xpWizardMax = 100	Magier maximale Erfahrung.
- int	xpKnightMax = 100	Ritter maximale Erfahrung.
- int	lvlArcher = 100	Aktuelles Level Archer.
- int	lvlWizard = 100	Aktuelles Level Wizard.
- int	lvlKnight = 150	Aktuelles Level Knight.
- int	lvlArcher = 100	Aktuelles Level Archer.
- int	lvlWizard = 100	Aktuelles Level Wizard.
- int	lvlKnight = 150	Aktuelles Level Knight.
- int	archerAbility1lvl	Aktuelles Level der Archer Ability1.
- int	archerAbility2lvl	Aktuelles Level der Archer Ability2.
- int	archerAbility3lvl	Aktuelles Level der Archer Ability3.
- int	wizardAbility1lvl	Aktuelles Level der Wizard Ability1.
- int	wizardAbility2lvl	Aktuelles Level der Wizard Ability2.
- int	wizardAbility3lvl	Aktuelles Level der Wizard Ability3.
- int	knightAbility1lvl	Aktuelles Level der Knight Ability1.
- int	knightAbility2lvl	Aktuelles Level der Knight Ability2.
- int	knightAbility3lvl	Aktuelles Level der Knight Ability3.

Methoden

Methodenname	Beschreibung
+ <code>LebenAbziehen</code> (int <code>dmg</code> , int <code>currentSpieler</code> , <code>ParticleManager</code> <code>particleManager</code> , Vector2 <code>position</code> , Game1 <code>game</code>)	Überprüft welcher Spieler ausgewählt ist(<code>currentPlayer</code>). Dann wird vom aktuellen Spieler Leben abgezogen und geschaut ob der Spieler mehr als 0 Lebenspunkte hat. Somit wird zum GameOver Screen gewechselt.(<code>game.changeState()</code>). Als letztes wird <code>SpawnHitParticle</code> aufgerufen.
+ <code>LebenHinzufügen</code> (int <code>wert</code> , int <code>currentSpieler</code>)	Wie auch bei <code>LebenAbziehen</code> wird zuerst geprüft welcher Spieler ausgewählt ist. Danach wird geprüft ob <code>hp.. + wert</code> größer als der <code>MaximalWert</code> ist und somit wird <code>hp..</code> mit <code>hp..Max</code> gleichgesetzt. Wenn dies nicht der Fall ist, wird überprüft ob <code>hp..</code> kleiner als <code>hp..Max</code> ist. Es wird zu <code>hp..</code> , <code>wert</code> dazu addiert,
+ <code>XpHinzufügen</code> (int <code>exp</code> , int <code>currentSpieler</code>)	Auch hier wird <code>currentPlayer</code> geprüft. Danach wird die Erfahrung zum <code>xp..</code> addiert. Wenn <code>xp..</code> größer gleich <code>xp..Max</code> ist, wird das Level (lvl..) erhöht, die Erfahrung zurückgesetzt und 20 Erfahrung zu <code>xp..Max</code> addiert.
+ <code>loadPlayer</code> (string[] <code>content</code>)	Alle Attribute bekommen den zuvor gespeicherten Wert.

Ordner:

3.12 Waffen Manager

Klassen:

3.12.1 WeaponManager

3.12.1 WeaponManager

Die Klasse **WeaponManager** zeichnet die Waffe des Spieles (Bogen, Zauberstab, Schwert).

Public Attribute

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
+ Texture2D	bow	Bogen Textur.
+ Texture2D	magicstaff	Zauberstab Textur.
+ Texture2D	sword	Schwert Textur.
+ Texture2D	swordburn	Schwert Feuer Animation.
- Vector2	positionPlayer	Spieler Position.
- Vector2	positionWeapon	Waffen Position.
- Vector2	directionXY	Maus Richtung.
- float	rotation = 0	Waffe Rotation.
- float	richtung = 1	Waffe Spiegelung.
- bool	burn = false	True wenn das (Ritter) Schwert brennt.
- Animations	animations	Animations Objekt.

Methoden

+ **Update**(Vector2 **mouse**, Vector2 **player**, int **currentPlayer**)

In der Update Methode wird die Waffen Position sowie die Richtung aktualisiert.

Wenn der Bogenschütze ausgewählt ist (currentPlayer = 0), wird richtung auf 1 gesetzt. Ist der Magier oder Ritter ausgewählt, wird richtung auf -1 gesetzt und somit gespiegelt. Als nächstes wird die Spieler Position (positionPlayer) mit dem Offset addiert. Danach wird positionPlayer mit mouse subtrahiert und mit richtung multipliziert (gespeichert in directionXY). Mithilfe von Math.Atan2 wird somit die Drehung in rotation gespeichert.

+ **DrawWeapon**(SpriteBatch **spriteBatch**, int **currentPlayer**, GameTime **gameTime**)

Zeichnet die Waffe des Spielers

Es wird geprüft welcher Spieler ausgewählt ist (**currentPlayer**), demnach wird die zugehörige Texture gezeichnet.

Klasse:

3.13 HUD

3.13 HUD

Die Klasse **HUD** zeichnet die Lebensanzeige, die Erfahrungsanzeige sowie weitere Overlays.

Attribute

HUD besitzt 21 Texture2D Attribute.(Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
+ Texture2D	xpFill	Erfahrungsanzeige gefüllt
+ Texture2D	xpFrame	Erfahrungsanzeige Rahmen
+ Texture2D	hpFill	Lebensanzeige gefüllt
+ Texture2D	hpFrame	Lebensanzeige Rahmen
+ Texture2D	coin	Münze
+ Texture2D	abilityload	Ability Cooldown Umrahmung
+ Texture2D	abilitylocked	Ability Anzeige geschlossen
+ Texture2D	ability1Archer	Archer Ability 1
+ Texture2D	ability2Archer	Archer Ability 2
+ Texture2D	ability3Archer	Archer Ability 3
+ Texture2D	ability1Wizard	Wizard Ability 1
+ Texture2D	ability2Wizard	Wizard Ability 2
+ Texture2D	ability3Wizard	Wizard Ability 3
+ Texture2D	ability1Knight	Knight Ability 1
+ Texture2D	ability2Knight	Knight Ability 2
+ Texture2D	ability3Knight	Knight Ability 3
+ Texture2D	abilityLevel	Gelbes Rechteck welches das Level der Ability anzeigt.
+ Texture2D	headArcher	Archer Kopf
+ Texture2D	headWizard	Wizard Kopf
+ Texture2D	headKnight	Knight Kopf
+ Texture2D	crosshair	Fadenkreuz

Methoden

HUD besitzt die `drawxpbar()`-, die `drawhpBar()`- und die `drawAbility()` Methode, welche innerhalb der `DrawHUD` Methode aufgerufen werden.

Methodenname	Beschreibung
- <code>drawxpbar()</code>	
- <code>drawhpBar()</code>	
- <code>drawAbility()</code>	
+ <code>DrawHUD(SpriteBatch spriteBatch, Gametime gameTime, Vector2 mouseposition, int currentSpieler, AbilityManager abilityManager, Player _player)</code>	<p>Zuerst wird überprüft welcher Spieler ausgewählt ist. Wenn zum Beispiel <code>currentSpieler = 0</code> (Archer) ist, wird geschaut welches Level der Spieler hat (<code>_player.LvlArcher</code>). Demnach wird <code>drawAbility(..)</code> mit den dazugehörigen Parametern aufgerufen. Als nächstes wird das Level der Ability (<code>_player.ArcherAbility1..2..3</code>) überprüft und je nach Wert werden bis zu 3 <code>abilityLevel</code> Texturen nebeneinander gezeichnet. Nun wird <code>drawhpBar(..)</code> sowie <code>drawxpbar(..)</code> aufgerufen. Als letztes wird dann noch das aktuelle Level, die aktuelle Erfahrung sowie der Kopf gezeichnet (<code>headArcher</code>).</p> <p>Der selbe Durchgang wird für <code>currentSpieler = 1</code> (Wizard) und <code>currentSpieler = 2</code> (Knight) wiederholt.</p>

Klassen:

1.14 SaveLoad
LoadLanguage

1.14 SaveLoad

Die Klasse **SaveLoad** verwaltet das Speichern und Laden von Speicherständen und auch das Laden und Speichern von Einstellungen.

Attribute

SaveLoad besitzt 3 Attribute. (Tabelle)

Datentyp	Attributname	Beschreibung
- string[29]	saveContent	Speichert was gespeichert werden soll damit per File alles in die Datei geschrieben werden kann.
- string[29]	loadContent	Speichert den Inhalt der der File aus der Datei gelesen wird.
- string[5]	configContent	Speichert die Einstellungen die per File ausgelesen werden.

Methoden

SaveLoad besitzt 6 Methode(**save**, **saveDead**, **load**, **loadHealth**, **configSave**, **configLoad**). **Save** speichert alle Werte des Spielers und auf welcher Map sich dieser befindet. **SaveDead** ist ein bestimmtes Speichern was nur beim Tod aufgerufen wird da hier nur das Leben verändert wird. **Load** lädt alle Werte aus der File und gibt diese zurück. **LoadHealth** wird benötigt um nur die Leben der einzelnen Klassen aus den Dateien rauszulesen. **ConfigSave** speichert alle Einstellungen welche in den Optionen vorgenommen wurden. **ConfigLoad** liest alle Einstellungen und gibt diese Zurück.

+ **save**(string mapname, string klasse, float positionX, float positionY, int healthA, int maxhealthA, int xpA, int xpMaxA, int levelA, int healthK, int maxhealthK, int xpK, int xpMaxK, int levelK, int healthW, int maxhealthW, int xpW, int xpMaxW, int levelW, int coins, int archer1lvl, int archer2lvl, int archer3lvl, int wizard1lvl, int wizard2lvl, int wizard3lvl, int knight1lvl, int knight2lvl, int knight3lvl, int saveFile)

save(..) wird in **Game1** aufgerufen damit der Spielstand gespeichert wird.

SaveFile gibt an in welcher der 3 Files der Speicherstand gespeichert werden soll (1-3). Alle anderen Parameter werden in den String-Array **saveContent** geschrieben welcher dann per **File.WriteAllLines** in die Datei geschrieben wird.

+ **load**(ref string mapname, ref string klasse, ref float positionX, ref float positionY, int loadFile)

load(..) wird in **Game1** aufgerufen damit der Spielstand geladen wird.

LoadFile gibt an in welcher Speicherstand geladen werden soll (1-3). Einige Werte werden schon in der Methode beschrieben der Rest wird per Array zurück gegeben. Gelesen wird per **File.ReadAllLines**.

+ **saveDead**(int saveFile, int klasse, int health)

saveDead(..) wird aufgerufen wenn der Spieler stirbt.

Bei dieser Art von Speicherung wird nur das Leben gespeichert da man nach jedem Tod beim Neustarten 50% des Lebens wieder bekommt. Diese Methode ermöglicht das man nur angeben muss welche Klasse (int klasse) und wie viel Leben (int health) gespeichert werden muss.

+ `loadHealth`(int klasse, int loadFile)

`loadHealth`(..) wird benutzt um das Maximale Leben einer Klasse auszulesen.

Da man es nicht ausnutzen soll das man Leben beim Neustart bekommt, kann das Leben nie über das maximale Leben gehen. Jedoch muss man das maximale Leben auslesen können und genau das macht diese Methode. Klasse definiert welche der drei Klassen gespielt wurde und loadFile gibt an aus welchem Speicherstand gelesen werden soll.

+ `configSave`(string width, string height, string windowMode, float volume, string language)

`configSave`(..) wird in `Game1` aufgerufen um die Einstellungen zu speichern.

Die oben angegebenen Parameter sind alle Einstellungen welche dann in ConfigContent geschrieben werden. ConfigContent wird dann mithilfe von File.WriteAllLines in die config.cfg geschrieben.

+ `configLoad`()

`configLoad`() wird in `Game1` aufgerufen um die Einstellungen auszulesen

`configLoad` lädt die Einstellungen als Array aus der config.cfg Datei und gibt den ganzen Array dann zurück.

LoadLanguage

Die Klasse **LoadLanguage** verwaltet das Speichern und Laden von Sprachdateien.

Attribute

LoadLanguage besitzt keine Attribute.

Methoden

LoadLanguage besitzt eine Methode(**loadLanguage**). **LoadLanguage** liest alle Zeilen aus der Sprachdatei und gibt diese aus.

+ **loadLanguage**()

loadLanguage(..) wird in den Menüs aufgerufen um dort die Wörter auf den Knöpfen zu übersetzen.

LoadLanguage liest mithilfe der angegebenen Sprache (string lan) die Zeilen aus der Datei und setzt den Spracharray gleich dem was **File.ReadAllLines** aus der Datei liest.

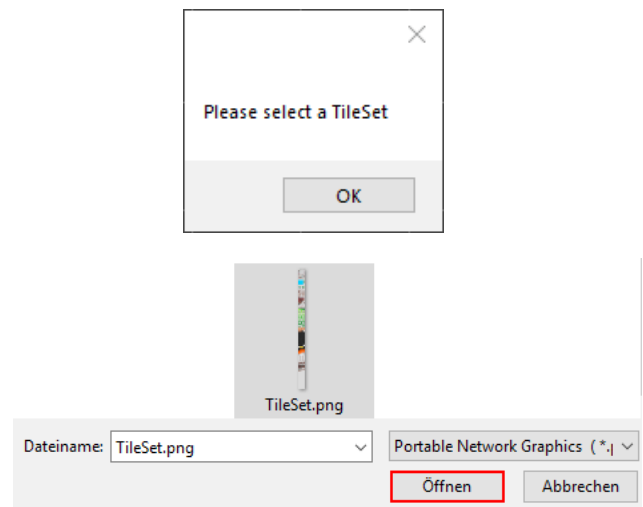
II. MapEditor

1. Programmbeschreibung:

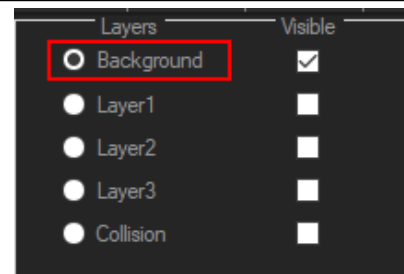
Der MapEditor wurde erstellt, um schnell viele Maps für YellowTale zu erstellen. Alle Maps haben die selbe Größe (x:32*64, y:17*64) und die Größe kann während der Laufzeit nicht geändert werden (änderbar nur im Code). Eine zugehöriges TileSet muss man selber erstellen und als .png speichern (Vorlage unter MapEditor/TileSet.png). Jede Map besteht aus 5 txt Dateien. Background und Layer1, welche hinter dem Spieler gezeichnet werden. Layer2 und Layer3, welche vor dem Spieler gezeichnet werden. Die Collision txt Datei ist doppelt so groß sodass jedes Kollision-Tile 32x32 groß ist.

2. Programmablauf (einfacher Ablauf mit einen Layer):

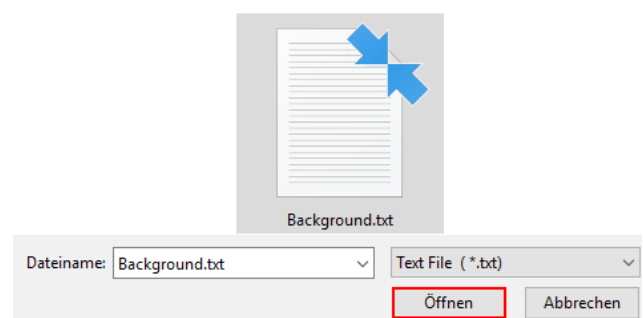
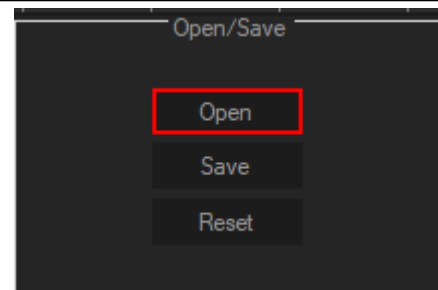
1. Als erstes wird der Benutzer aufgefordert ein TileSet auszuwählen (MapEditor/Tileset.png).



2. Nun kann die Map ausgewählt werden (MapEditor/VorlageMap/..). Hier im Beispiel wird background.txt verwendet.



3. Als nächstes muss Open angeklickt werden. Es erscheint der Windows Explorer und es wird unter (MapEditor/VorlageMap) die Background.txt Datei ausgewählt.



4. Mithilfe des Pinsels kann ein Tile angeklickt werden und es wird rechts neben Tile ID: angezeigt.



5. Nun kann man auf der Großen Fläche das ausgewählte Tile platzieren. Mit dem Radierer (Rechts neben dem Pinsel) kann man das Tile entfernen.



6. Um die Map zu speichern, kann man entweder auf Save drücken (Es wird nur das ausgewählte Layer gespeichert) oder Strg+s (Es werden alle Layer gespeichert). Weitere Hotkeys unter MenuStrip/Help/Controls.



3 .Klassen:

3.1 MainWindow

3.1 MainWindow

Die Klasse **MainWindow** ist die Steuerklasse aller Klassen. Ebenso gehen von ihr alle Events aus.

Attribute

MainWindow besitzt 14 Attribute. (Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- int	XOffsetMap	Bewegung der Map in X Richtung.
- int	YOffsetMap	Bewegung der Map in Y Richtung.
- int	XOffsetTileMap	Bewegen des TileSet(Scrolling).
- int	YOffsetTileMap	Bewegen des TileSet(Scrolling).
- bool	UpdateTileSet = true	Wird in pbTileMap_Paint abgefragt sowie gesetzt.
- bool	UpdateSelectedTile = false	Wird auf True gesetzt sobald eine TileMap vorhanden ist.
- bool	LoadGrid = true	Damit das Grid nur 1x geladen wird.
- DrawImage	drawImage	Objekt der DrawImage Klasse.
- TileMap	tileMap	Objekt der TileMap Klasse.
- Background	background	Objekt der Background Klasse.
- Layer1	layer1	Objekt der Layer1 Klasse.
- Layer2	layer2	Objekt der Layer2 Klasse.
- Layer3	layer3	Objekt der Layer3 Klasse.
- Collision	collision	Objekt der Collision Klasse.

Methoden

Alle Methoden wurden in #region Codeblöcke eingeteilt.

#region MainWindow: Load ,Close, Minimize, Maximize, Move, KeyDown

Methodenname	Beschreibung
- MainWindow_Load (object sender , EventArgs e)	pBTileMap wird das MouseWheel Event hinzugefügt.
- btnClose_Click (object sender , EventArgs e)	Schließt den MapEditor.
- btnMinimize_Click (object sender , EventArgs e)	Minimiert den MapEditor.
- btnMaximize_Click (object sender , EventArgs e)	Maximiert den MapEditor bzw. wenn der MapEditor bereits Maximiert ist, wird ResetMainWindow() aufgerufen und der MapEditor auf die Standard Größe zurückgesetzt.
- menuStrip1_MouseDown (object sender , EventArgs e)	Wenn die Linke Maus Taste gedrückt wird, wird ResetMainWindow() aufgerufen sowie den beiden splitContainer eine neuen SplitterDistance gesetzt. ReleaseCapture() sowie SendMessage(Handle, WM_NCLBUTTONDOWN, HT_CAPTION, 0) wurde von https://stackoverflow.com/questions/1607841/c-how-to-drag-a-from-by-the-form-and-its-controls rauskopiert.
- MainWindow_KeyDown (object sender , EventArgs e)	Wird aufgerufen wenn eine Taste gedrückt wird. Wenn z.B. STRG + S gedrückt wird, wird saveAllToolStripMenuItem. PerformClick() ausgeführt und somit ein Click simuliert. Wenn W,A,S oder D (Richtungstasten) gedrückt wird, wird geschaut ob rBtnCollision.Checked auf false ist und somit drawImage.MoveBitmap (Richtung) aufgerufen. Ausserdem wird je nach Richtung X/YOffsetMap mit drawImage.TileWidth addiert bzw. subtrahiert. Alle weiteren Hotkeys findet man im MapEditor unter MenuStrip/Help/Controls

#region pBMap: MouseClick, MouseMove, Paint

<u>Methodenname</u>	<u>Beschreibung</u>
- pBMap_MouseClick (object sender , MouseEventArgs e)	Wird aufgerufen wenn auf pBMap geklickt wird. drawImage.CurrentTileX/Y wird den Inhalt von tbTileX/Y übergeben. Als nächstes wird geprüft welcher Layer/Background/Collision ausgewählt ist. Wenn z.B. rBtnBackground.Checked = true ist, wird background.TileX/Y den Wert von drawImage.CurrentTileX/Y zugewiesen. Nun wird noch geprüft welches Tool ausgewählt ist (Pen oder Eraser) und demnach background.WriteArray()/EraseArray() sowie drawImage.DrawBackground(true/false) aufgerufen. Als letztes wird MapUpdate() sowie SelectedTileUpdate() aufgerufen.
- pBMap_MouseMove (object sender , EventArgs e)	In tbMouseX/Y.Text wird die Maus Anzeige Position geschrieben, ebenso wird in tbTileX/Y.Text das zugehörige Tile geschrieben. Wenn die Linke Maus Taste gedrückt wird, wird exakt der selbe Code wie bei pBMap_MouseClick() ausgeführt.
- pBMap_Paint (object sender , EventArgs e)	In pBMap_Paint() werden alle Bitmaps gezeichnet. Gezeichnet wird in dieser Reihenfolge (Wenn alle Checkboxes auf True sind): 1. Background (Wird zuerst gezeichnet) 2. Layer1 3. Layer2 4. Layer3 5. Collision 6. Grid (Wird als letztes gezeichnet) Alle Bitmaps werden über die e.Graphics.DrawImage() Methode gezeichnet. Als Offset wird XOffsetMap sowie YOffsetMap übergeben. Bei cBShowGrid wird ebenso geprüft ob LoadGrid auf true ist und somit das Grid bereits geladen wurde.

#region pBTileMap: MouseClick, MouseMove, Paint, ScrollBar , MouseWheel

Methodenname	Beschreibung
- <code>pBTileMap_MouseClick</code> (object <code>sender</code> , MouseEventArgs <code>e</code>)	In <code>tBTileID.Text</code> wird <code>tileMap.TileID()</code> übergeben. Außerdem wird die <code>TileID</code> von <code>background</code> , <code>layer1</code> , <code>layer2</code> , <code>layer3</code> und <code>collision</code> gesetzt. Zuletzt wird noch <code>SelectedTileUpdate()</code> aufgerufen.
- <code>pBTileMap_MouseMove</code> (object <code>sender</code> , EventArgs <code>e</code>)	<code>TileMap.MouseX/Y</code> wird mithilfe der aktuelle Maus Position(<code>e.X/Y</code>) sowie <code>X/YOffsetTileMap</code> aktualisiert. Selbe gilt für <code>tileMap.TileX/Y</code> nur das hier noch durch die Tile Breite (<code>tilemap.TileWidth</code>) bzw. Tile Höhe (<code>tileMap.TileHeight</code>) geteilt wird.
- <code>pBTileMap_Paint</code> (object <code>sender</code> , EventArgs <code>e</code>)	Zuerst wird geprüft ob bereits eine TileSet ausgewählt wurde(<code>UpdateTileSet</code>). Wenn kein TileSet ausgewählt wurde, wird aufgefordert ein TileSet(.png) auszuwählen. Es wird <code>tileMap.DrawTileMap()</code> ausgeführt und <code>UpdateTileSet</code> auf false gesetzt. Nun wird je nach größe des TileSet das <code>Maximum</code> der ScrollBarTileMap verändert. Wenn <code>rBtnCollision.Checked</code> auf true ist, wird <code>bmCollisionTileMap</code> gezeichnet. Wenn <code>rBtnCollision.Checked</code> auf false ist, wird die Standard Bitmap gezeichnet(<code>StaticMembers.bmTileMap</code>).
- <code>ScrollBarTileMap_MouseWheel</code> (object <code>sender</code> , MouseEventArgs <code>e</code>)	<code>YOffsetTileMap</code> wird ScrollBarTileMap. <code>Value</code> zugewiesen. <code>TileSetUpdate()</code> wird aufgerufen.
- <code>pBTileMap_MouseWheel</code> (object <code>sender</code> , EventArgs <code>e</code>)	Mithilfe von <code>e.Delta</code> wird geschaut ob man das Mause rad nach oben(Positiv) oder nach unten(negativ) dreht. Ist also <code>e.Delta</code> größer als 0 wird <code>nUDTileSetHeight.Value</code> von ScrollBarTileMap. <code>Value</code> subtrahiert. Wenn <code>e.Delta</code> kleiner als 0 ist wird <code>nUDTileSetHeight.Value</code> von ScrollBarTileMap. <code>Value</code> addiert.

#region pBSelectedTile: Paint

Methodenname	Beschreibung
- <code>pBSelectedTile_Paint</code> (object <code>sender</code> , MouseEventArgs <code>e</code>)	Überprüft ob <code>UpdateSelectedTile</code> true ist, und führt dann <code>tileMap.DrawSelectedTile()</code> aus. Zeichnet (wenn vorhanden) <code>bmSelectedTile</code> auf <code>pBSelectedTile</code> .

#region Groupbox Layers: Object, Collision, Animation, Layer1, Layer2

Methodenname	Beschreibung
- <code>rBtnBackground_CheckChanged(object sender, EventArgs e)</code>	cBBackgroundVisible.Checked auf true setzen.
- <code>rBtnLayer1_CheckChanged(object sender, EventArgs e)</code>	cBLayer1Visible.Checked auf true setzen.
- <code>rBtnLayer2_CheckChanged(object sender, EventArgs e)</code>	cBLayer2Visible.Checked auf true setzen.
- <code>rBtnLayer3_CheckChanged(object sender, EventArgs e)</code>	cBLayer3Visible.Checked auf true setzen.
- <code>rBtnCollision_CheckChanged(object sender, EventArgs e)</code>	Wenn rBtnCollision.Checked auf true ist, wird tileMap.collisionChecked auf true gesetzt sowie ScrollBarTileMap.Value auf 0 gesetzt. Wenn rBtnCollision.Checked false ist, wird tileMap.collisionChecked auf false gesetzt. Als letztes wird cBCollisionVisible.Checked auf true gesetzt sowie TileSetUpdate() wird aufgerufen.
- <code>cBBackgroundVisible_CheckChanged(object sender, EventArgs e)</code>	An btnUpdate wird ein Button Klick simuliert.
- <code>cBLayer1Visible_CheckChanged(object sender, EventArgs e)</code>	An btnUpdate wird ein Button Klick simuliert.
- <code>cBLayer2Visible_CheckChanged(object sender, EventArgs e)</code>	An btnUpdate wird ein Button Klick simuliert.
- <code>cBLayer3Visible_CheckChanged(object sender, EventArgs e)</code>	An btnUpdate wird ein Button Klick simuliert.
- <code>cBCollisionVisible_CheckChanged(object sender, EventArgs e)</code>	An btnUpdate wird ein Button Klick simuliert.

#region Groupbox TileMap: MapWidth, MapHeight, TileWidth, TileHeight

Methodenname	Beschreibung
+ <code>nUDWidth_ValueChanged(object sender, EventArgs e)</code>	Wird aufgerufen wenn sich der Wert verändert. drawImage.Width bekommt den Wert von nUDWidth.Value. Grid wird aktualisiert und an btnUpdate wird ein Klick simuliert.
+ <code>nUDHeight_ValueChanged(object sender, EventArgs e)</code>	drawImage.Height bekommt den Wert von nUDHeight.Value. Grid wird aktualisiert und an btnUpdate wird ein Klick simuliert.
+ <code>nUDTileWidth_ValueChanged(object sender, EventArgs e)</code>	drawImage.TileWidth bekommt den Wert von nUDTileWidth.Value. Grid wird aktualisiert und an btnUpdate wird ein Klick simuliert.
+ <code>nUDTileHeight_ValueChanged(object sender, EventArgs e)</code>	drawImage.TileHeight bekommt den Wert von nUDTileHeight.Value. Grid wird aktualisiert und an btnUpdate wird ein Klick simuliert.

#region Groupbox Settings: ShowGrid, GridColor, Update

Methodenname	Beschreibung
- <code>cBShowGrid_CheckedChanged</code> (object <code>sender</code> , EventArgs <code>e</code>)	Wenn <code>cBShowGrid.Checked</code> auf <code>true</code> ist wird <code>drawImage.ShowGrid()</code> ausgeführt. Außerdem wird <code>btnUpdate.PerformClick()</code> ausgeführt.
- <code>btnGridColor_Click</code> (object <code>sender</code> , EventArgs <code>e</code>)	Es wird <code>colorDialogGridColor.ShowDialog()</code> ausgeführt und somit erscheint ein neues Fenster in dem man eine Farbe auswählen kann. <code>drawImage.GridColor</code> wird <code>colorDialogGridColor.Color</code> übergeben, <code>drawImage.ShowGrid()</code> und <code>MapUpdate()</code> werden ausgeführt.
- <code>btnUpdate_Click</code> (object <code>sender</code> , EventArgs <code>e</code>)	Alle Update Methoden werden ausgeführt(<code>MapUpdate()</code> , <code>TileSetUpdate()</code> , <code>SelectedTileUpdate()</code>).

#region Groupbox TileSet: Open

Methodenname	Beschreibung
- <code>btnOpenTileSet_Click</code> (object <code>sender</code> , EventArgs <code>e</code>)	<code>tileMap.DrawTileMap()</code> und <code>TileSetUpdate()</code> werden ausgeführt. Je nach größe der Bitmap (<code>StaticMember.bmTileMap.Height</code>) wird das Maximum verändert. <code>ScrollBarTileMap.Value</code> wird <code>ScrollBarTileMap.Minimum</code> übergeben.

#region Groupbox Open/Save: Open, Save, Reset

Methodenname	Beschreibung
- <code>btnOpenMap_Click</code> (object <code>sender</code> , EventArgs <code>e</code>)	Es wird ein neues <code>Map.InputMap</code> Objekt erstellt(<code>inputMap</code>). Es wird geprüft ob <code>rBtn[*].Checked</code> auf <code>true</code> ist sowie <code>cB[*].Visible.Checked</code> auf <code>true</code> ist. Wenn dies der Fall ist wird <code>inputMap.Input[*]()</code> und <code>drawImage.Import[*](true)</code> ausgegeben. Wenn allerdings <code>cB[*].Visible.Checked</code> auf <code>false</code> ist, wird eine Meldung ausgegeben („[*] Visible muss angekreuzt sein“).
- <code>btnSaveMap_Click</code> (object <code>sender</code> , EventArgs <code>e</code>)	Es wird ein neues <code>Map.InputMap</code> Objekt erstellt(<code>inputMap</code>). Es wird geprüft welcher Layer ausgewählt ist, und somit <code>outputMap.Output[*]()</code> ausgeführt.
- <code>btnReset_Click</code> (object <code>sender</code> , EventArgs <code>e</code>)	Es wird eine Warnung ausgegeben ob man wirklich den Layer löschen möchte. Wenn sich der Nutzer für <code>DialogResult.Yes</code> entschieden hat wird überprüft welcher Layer ausgewählt ist. Demnach wird <code>[*].ResetArray()</code> und <code>drawImage.Import[*](false)</code> ausgeführt. Als letztes wird dann noch die Map aktualisiert.

*Background, Layer1, Layer2, Layer3, Collision

#region Tools: Pen, Eraser, Select, TileSelect

Methodenname	Beschreibung
+ <code>rBtnPen_CheckedChanged(object sender, EventArgs e)</code>	Wenn <code>rBtnPen.Checked</code> true ist, wird ein dunkles Icon als Image verwendet. Ist <code>rBtnPen.Checked</code> false, wird ein helles Icon verwendet.
+ <code>rBtnEraser_CheckedChanged(object sender, EventArgs e)</code>	Wenn <code>rBtnEraser.Checked</code> true ist, wird ein dunkles Icon als Image verwendet. Ist <code>rBtnEraser.Checked</code> false, wird ein helles Icon verwendet.
+ <code>rBtnSelect_CheckedChanged(object sender, EventArgs e)</code>	Wenn <code>rBtnSelect.Checked</code> true ist, wird ein dunkles Icon als Image verwendet. Ist <code>rBtnSelect.Checked</code> false, wird ein helles Icon verwendet.
+ <code>rBtnTileSelect_CheckedChanged(object sender, EventArgs e)</code>	Wenn <code>rBtnTileSelect.Checked</code> true ist, wird ein dunkles Icon als Image verwendet. Ist <code>rBtnTileSelect.Checked</code> false, wird ein helles Icon verwendet.

#region MenuStrip

Methodenname	Beschreibung
- <code>saveAllToolStripMenuItem_Click(object sender, EventArgs e)</code>	Erstellt ein neues <code>Map.OutputMap</code> Objekt(<code>outputMap</code>). Überprüft ob <code>StaticMembers.[*]Path</code> verfügbar ist, und führt dann <code>outputMap.Output[*]()</code> aus.
- <code>openToolStripMenuItem_Click(object sender, EventArgs e)</code>	Simuliert ein Klick an <code>btnOpenMap</code> .
- <code>fillAllTilesTollStripMenuItem_Click(object sender, EventArgs e)</code>	Überprüft welcher Layer ausgewählt wird und führt dann <code>tileMap.FillAllTiles</code> (0→ background, 1/2/3 → layer1/2/3, 4 → collision) sowie <code>drawImage.Import[*](true)</code> aus. Als letztes wird <code>MapUpdate()</code> aufgerufen.
- <code>resetAllTilesToolStripMenuItem_Click(object sender, EventArgs e)</code>	Simuliert ein Klick an <code>btnReset</code> .
- <code>controlsToolStripMenuItem_Click(object sender, EventArgs e)</code>	Erstellt eine neue Form welche die Hotkeys anzeigt.
- <code>contaToolStripMenuItem_Click(object sender, EventArgs e)</code>	Öffnet den Standard Webbrowser mit der angebenen URL (http://jaemil.de). Website ist noch in Bearbeitung.

*Background, Layer1, Layer2, Layer3, Collision

#region Update

Methodenname	Beschreibung
- <code>MapUpdate()</code>	Führt <code>pBMap.Refresh()</code> aus.
- <code>TileSetUpdate()</code>	Führt <code>pBTileMap.Refresh()</code> aus.
- <code>SelectedTileUpdate()</code>	Führt <code>pBSelectedTile.Refresh()</code> aus.

Klasse:

3.2 StaticMembers

3.2 StaticMembers

Statische Klasse **StaticMembers**

Attribute

StaticMembers besitzt 20 statische Attribute. (Tabelle)

Datentyp	Attributname	Beschreibung
+ int	DrawImageWidth = 30	Breite der Map
+ int	DrawImageHeight = 17	Höhe der Map
+ int	tileWidth = 64	Tile Breite
+ int	tileHeight = 64	Tile Höhe
+ int[,]	backgroundArray = new int [DrawImageWidth, DrawImageHeight]	2D Array von background.
+ string	backgroundPath	Speichert den Path von background.txt
+ Bitmap	BackgroundBitmap = new Bitmap [DrawImageWidth * tileWidth, DrawImageHeight * tileHeight]	Bitmap von background
+ int[,]	layer1Array = new int [DrawImageWidth, DrawImageHeight]	2D Array von layer1.
+ string	layer1Path	Speichert den Path von layer1.txt
+ Bitmap	layer1Bitmap = new Bitmap [DrawImageWidth * tileWidth, DrawImageHeight * tileHeight]	Bitmap von layer1.
+ int[,]	layer2Array = new int [DrawImageWidth, DrawImageHeight]	2D Array von layer2.
+ string	layer2Path	Speichert den Path von layer2.txt
+ Bitmap	layer2Bitmap = new Bitmap [DrawImageWidth * tileWidth, DrawImageHeight * tileHeight]	Bitmap von layer2.
+ int[,]	layer3Array = new int [DrawImageWidth, DrawImageHeight]	2D Array von layer3.
+ string	layer3Path	Speichert den Path von layer3.txt
+ Bitmap	layer3Bitmap = new Bitmap [DrawImageWidth * tileWidth, DrawImageHeight * tileHeight]	Bitmap von layer3.
+ int[,]	collisionArray = new int [DrawImageWidth * 2, DrawImageHeight * 2]	2D Array von collision. Doppelt so groß weil die Kollision 32x32 groß ist.
+ string	collisionPath	Speichert den Path von collision.txt
+ Bitmap	collisionBitmap = new Bitmap [DrawImageWidth * tileWidth, DrawImageHeight * tileHeight]	Bitmap von collision.
+ int[,]	TileMapArray = new int[4, 200]	4 Tiles Breit und 200 Tiles Hoch.
+ Bitmap	bmSelectedTile	Bitmap von SelectedTile
+ Bitmap	bmTileMap	Bitmap von der TileMap.

Ordner:
3.3 Draw

Klassen:
3.3.1 DrawImage

3.3.1 DrawImage

Die Klasse **DrawImage** zeichnet das Grid sowie die Layer.

Attribute

MainWindow besitzt 17 Attribute.(Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
+ Bitmap	bmGrid	Bitmap für das Grid(Gitter).
- Graphics	gBackground	Graphics für den Hintergrund.
- Graphics	gLayer1	Graphics für Layer1.
- Graphics	gLayer2	Graphics für Layer2.
- Graphics	gLayer3	Graphics für Layer3.
- Graphics	gCollision	Graphics für Collision.
- int	mouseX = 0	Maus Position X.
- int	mouseY = 0	Maus Position Y.
- int	width = StaticMembers.DrawImageWidth	Breite der Bitmap.
- int	height = StaticMembers.DrawImageWidth	Höhe der Bitmap.
- int	tileWidth = StaticMembers.tileWidth	Breite eines Tiles.
- int	tileHeight = StaticMembers.tileHeight	Höhe eines Tiles.
- int	currentTileX	Aktuelles Tile X.
- int	currentTileY	Aktuelles Tile Y.
- int	currentTileCollisionX	Aktuelles Tile X der Collision.
- int	currentTileCollisionY	Aktuelles Tile Y der Collision.
- Color	gridColor	Farbe von dem Grid.

Methoden

Methodenname	Beschreibung
- <code>ShowGrid()</code>	<p>Zeichnet das Grid.</p> <p>int <code>_width</code> und <code>_height</code> ist die Größe der <code>bmGrid</code> Bitmap. Zwei verschachtelte Schleifen die jeweils abfragen ob <code>x/y % TileWidth/TileHeight</code> gleich 0 ist. Wenn dies der Fall ist wird ein Punkt gezeichnet und somit entsteht ein Gitter.</p>
- <code>MoveBitmap(int direction)</code>	<p>Wird aufgerufen wenn man die Bitmap bewegt (WASD-Tasten).</p> <p><code>direction</code>: 0(links),1(rechts),2(hoch),3(runter) → background, layer1, layer2, layer3 <code>MouseX/Y</code> wird mit <code>TileWidth/TileHeight</code> addiert bzw. subtrahiert.</p> <p><code>direction</code>: 4(links),5(rechts),6(hoch),7(runter) → collision <code>MouseX/Y</code> wird mit <code>TileWidth : 2/TileHeight : 2</code> addiert bzw. subtrahiert. Durch 2 teilen weil die Größe der Collision 32x32 beträgt.</p>

Methodenname	Beschreibung
- <code>DrawAlgorithm</code> (bool <code>mode</code> , Bitmap <code>bmLayer</code> , Graphics <code>gLayer</code>)	<p>bool <code>mode</code> (gilt für alle Draw.. Methoden) → true = zeichnen → false = radieren</p> <p>Rectangle <code>rect</code> wird die aktuelle Position sowie die Größe des Tiles übergeben.</p> <p>Wenn der Zeichnen Modus aktiviert ist, wird <code>gLayer</code> Graphics.<code>FromImage</code>(<code>bmLayer</code>) übergeben. Nun werden mithilfe von zwei verschachtelten Schleifen das aktuelle Tile „gelöscht“(<code>Color.Clear</code>). Nun wird das ausgewählte Tile(<code>StaticMembers.bmSelectedTile</code>) an die gewünschte Tile Position gezeichnet.</p> <p>Ist der Radier Modus ausgewählt(<code>mode</code> = false), wird wie beim Zeichnen Modus das aktuelle Tile „gelöscht“ (<code>Color.Clear</code>).</p>
+ <code>DrawBackground</code> (bool <code>mode</code>)	<p>Es wird abgefragt ob ein Dateipfad (<code>StaticMembers.backgroundPath</code>) ausgewählt wurde. Wenn ein Pfad bereits ausgewählt wurde, wird <code>DrawAlgoritm</code>(<code>mode</code>, <code>StaticMembers.backgroundBitmap</code>, <code>gBackground</code>) ausgeführt. Wenn der Nutzer keine Datei ausgewählt hat, wird eine Anforderung ausgegeben die bittet den Nutzer eine Datei auszuwählen.</p>
+ <code>DrawLayer1</code> (bool <code>mode</code>)	<p>Gleich aufgebaut wie <code>DrawBackground</code>(bool <code>mode</code>), nur das hier <code>StaticMembers.layer1Path</code>, <code>StaticMembers.layer1Bitmap</code> und <code>gLayer1</code> verwendet wird.</p>
+ <code>DrawLayer2</code> (bool <code>mode</code>)	<p>Gleich aufgebaut wie <code>DrawBackground</code>(bool <code>mode</code>), nur das hier <code>StaticMembers.layer2Path</code>, <code>StaticMembers.layer2Bitmap</code> und <code>gLayer2</code> verwendet wird.</p>
+ <code>DrawLayer3</code> (bool <code>mode</code>)	<p>Gleich aufgebaut wie <code>DrawBackground</code>(bool <code>mode</code>), nur das hier <code>StaticMembers.layer3Path</code>, <code>StaticMembers.layer3Bitmap</code> und <code>gLayer3</code> verwendet wird.</p>
+ <code>DrawCollision</code> (bool <code>mode</code>)	<p>Gleich aufgebaut wie <code>DrawBackground</code>(bool <code>mode</code>), nur das hier <code>StaticMembers.collisionPath</code>, <code>StaticMembers.collisionBitmap</code> und <code>gCollision</code> verwendet wird.</p>

Methodenname	Beschreibung
- <code>ImportAlgorithm</code> (Bitmap <code>bmLayer</code> , Graphics <code>gLayer</code> , int[,] <code>arrLayer</code>)	<p>bool <code>mode</code> (gilt für alle Import.. Methoden) → true = zeichnen → false = reset</p> <p>Graphics <code>g</code> wird Graphics.<code>FromImage</code>(<code>StaticMembers.bmSelectedTile</code>) zugewiesen. Rectangle <code>rectTile</code> ist das Rechteck aus dem TileSet. Rectangle <code>rectMap</code> ist das Rechteck wo <code>rectTile</code> hingezeichnet wird.</p> <p><code>gLayer.Clear()</code> wird aufgerufen und löscht somit <code>bmLayer</code>. Durch eine verschachtelte Schleife wird jedes Tile von dem TileSet übergeben. <code>rectMap</code> wird das aktuelle Tile zugeordnet (<code>x/y * TileWidth/TileHeight</code>). Nun wird überprüft welchen Wert <code>arrLayer[x,y]</code> hat, mit % 4 kann der x-Wert geprüft werden sowie mit / 4 der y-Wert.</p> <p>Mithilfe von <code>g.DrawImage</code>(<code>StaticMembers.bmTileMap</code>, <code>rectTile</code>) wird aus <code>StaticMembers.bmTileMap</code> das zugehörige Rechteck ausgeschnitten und in <code>StaticMembers.bmSelectedTile</code> eingesetzt. Als letztes wird <code>StaticMembers.bmSelectedTile</code> an der Position von <code>rectMap</code> auf <code>gLayer</code> gesetzt.</p>
+ <code>ImportBackground</code> (bool <code>mode</code>)	<p>Mithilfe von Graphics.<code>FromImage</code>() wird <code>gBackground</code> <code>StaticMembers.backgroundBitmap</code> zugewiesen. Nun wird gefragt ob gezeichnet oder gelöscht werden soll. Wenn gezeichnet werden soll, wird <code>ImportAlgoritm</code>(<code>StaticMembers.backgroundBitmap</code>, <code>gBackground</code>, <code>StaticMembers.backgroundArray</code>) ausgeführt. Wenn gelöscht werden soll, wird <code>gBackground.Clear</code>(Color.Transparent) ausgeführt.</p>
+ <code>ImportLayer1</code> (bool <code>mode</code>)	<p>Gleich aufgebaut wie <code>ImportBackground</code>(bool <code>mode</code>), nur das hier <code>gLayer1</code>, <code>StaticMembers.layer1Bitmap</code> und <code>StaticMembers.layer1Array</code> verwendet wird.</p>
+ <code>ImportLayer2</code> (bool <code>mode</code>)	<p>Gleich aufgebaut wie <code>ImportBackground</code>(bool <code>mode</code>), nur das hier <code>gLayer2</code>, <code>StaticMembers.layer2Bitmap</code> und <code>StaticMembers.layer2Array</code> verwendet wird.</p>
+ <code>ImportLayer3</code> (bool <code>mode</code>)	<p>Gleich aufgebaut wie <code>ImportBackground</code>(bool <code>mode</code>), nur das hier <code>gLayer3</code>, <code>StaticMembers.layer3Bitmap</code> und <code>StaticMembers.layer3Array</code> verwendet wird.</p>
+ <code>ImportCollision</code> (bool <code>mode</code>)	<p>Gleich aufgebaut wie <code>ImportBackground</code>(bool <code>mode</code>), nur das hier <code>gCollision</code> <code>StaticMembers.collisionBitmap</code> und <code>StaticMembers.collisionArray</code> verwendet wird.</p>

Ordner:

3.4 Layers

Klassen:

- 3.4.1 Background*
- 3.4.2 Layer1*
- 3.4.3 Layer2*
- 3.4.4 Layer3*
- 3.4.5 Collision*

*Die Klassen Background, Layer1, Layer2, Layer3 und Collision sind alle sehr ähnlich aufgebaut.
Es wird nur immer ein anderes Array verwendet.

3.4.1 Background

Attribute

Background besitzt 5 Attribute.(Tabelle)

Datentyp	Attributname	Beschreibung
- Draw.DrawImage	drawImage	Objekt der DrawImage Klasse.
- Tiles.TileMap	tileMap	Objekt der TileMap Klasse.
- int	tileX	Tile Position X.
- int	tileY	Tile Position Y.
- int	tileID	Wert des Tiles.

Methoden

Methodenname	Beschreibung
+ WriteArray()	Wird aufgerufen wenn der Nutzer ein Tile ausgewählt hat und auf die Map Klickt. Wird in einer Try-Catch geschrieben, da TileX/TileY größer als das Array sein könnten. In StaticMembers.backgroundArray[TileX, TileY] wird TileID geschrieben.
+ EraseArray()	Selbe Prozedur wie bei der WriteArray() Methode, nur das hier nicht die TileID übergeben wird, sondern 0.
+ ResetArray()	Setzt StaticMembers.backgroundArray auf 0.

3.4.2 Layer1

Attribute

Layer1 besitzt 5 Attribute.(Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Draw.DrawImage	drawImage	Objekt der DrawImage Klasse.
- Tiles.TileMap	tileMap	Objekt der TileMap Klasse.
- int	tileX	Tile Position X.
- int	tileY	Tile Position Y.
- int	tileID	Wert des Tiles.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ WriteArray()	Wird aufgerufen wenn der Nutzer ein Tile ausgewählt hat und auf die Map Klickt. Wird in einer Try-Catch geschrieben, da TileX/TileY größer als das Array sein könnten. In StaticMembers.layer1Array[TileX, TileY] wird TileID geschrieben.
+ EraseArray()	Selbe Prozedur wie bei der WriteArray() Methode, nur das hier nicht die TileID übergeben wird, sondern 0.
+ ResetArray()	Setzt StaticMembers.layer1Array auf 0.

3.4.3 Layer2

Attribute

Layer2 besitzt 5 Attribute. (Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Draw.DrawImage	drawImage	Objekt der DrawImage Klasse.
- Tiles.TileMap	tileMap	Objekt der TileMap Klasse.
- int	tileX	Tile Position X.
- int	tileY	Tile Position Y.
- int	tileID	Wert des Tiles.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ WriteArray()	Wird aufgerufen wenn der Nutzer ein Tile ausgewählt hat und auf die Map Klickt. Wird in einer Try-Catch geschrieben, da TileX/TileY größer als das Array sein könnten. In StaticMembers.layer2Array[TileX, TileY] wird TileID geschrieben.
+ EraseArray()	Selbe Prozedur wie bei der WriteArray() Methode, nur das hier nicht die TileID übergeben wird, sondern 0.
+ ResetArray()	Setzt StaticMembers.layer2Array auf 0.

3.4.4 Layer3

Attribute

Layer3 besitzt 5 Attribute. (Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Draw.DrawImage	drawImage	Objekt der DrawImage Klasse.
- Tiles.TileMap	tileMap	Objekt der TileMap Klasse.
- int	tileX	Tile Position X.
- int	tileY	Tile Position Y.
- int	tileID	Wert des Tiles.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ WriteArray()	Wird aufgerufen wenn der Nutzer ein Tile ausgewählt hat und auf die Map Klickt. Wird in einer Try-Catch geschrieben, da TileX/TileY größer als das Array sein könnten. In StaticMembers.layer3Array[TileX, TileY] wird TileID geschrieben.
+ EraseArray()	Selbe Prozedur wie bei der WriteArray() Methode, nur das hier nicht die TileID übergeben wird, sondern 0.
+ ResetArray()	Setzt StaticMembers.layer3Array auf 0.

3.4.5 Collision

Attribute

Collision besitzt 5 Attribute.(Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Draw.DrawImage	drawImage	Objekt der DrawImage Klasse.
- Tiles.TileMap	tileMap	Objekt der TileMap Klasse.
- int	tileX	Tile Position X.
- int	tileY	Tile Position Y.
- int	tileID	Wert des Tiles.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ WriteArray()	Wird aufgerufen wenn der Nutzer ein Tile ausgewählt hat und auf die Map Klickt. Wird in einer Try-Catch geschrieben, da tileX/tileY größer als das Array sein könnten. In StaticMembers.collisionArray[tileX, tileY] wird tileID geschrieben.
+ EraseArray()	Selbe Prozedur wie bei der WriteArray() Methode, nur das hier nicht die tileID übergeben wird, sondern 0.
+ ResetArray()	Setzt StaticMembers.collisionArray auf 0. drawImage.Height * 2 / drawImage.Width * 2 da die Kollision vier mal so groß ist.

Ordner:

3.5 Map

Klassen:

3.5.1 InputMap

3.5.2 OutputMap

3.5.1 InputMap

Die Klasse **InputMap** besitzt den Algorithmus zum einlesen der txt Datei.

Attribute

InputMap besitzt 3 Attribute.(Tabelle)

Datentyp	Attributname	Beschreibung
- MainWindow	mainWindow	Objekt der MainWindow Klasse.
- Draw.DrawImage	drawImage	Objekt der DrawImage Klasse.
- StreamReader	reader	Zum einlesen der txt Datei.

Methoden

Methodenname	Beschreibung
+ InputAlgorithm (int width , int height , string content , int[,] arr)	<p>In content wird die ganze txt als string gespeichert. In storage wird die ID(Tile) eingetragen und später in arr übergeben.</p> <p>Zuerst wird überprüft ob content nicht leer ist. Nun kommen zwei for-Schleifen(in x- sowie y-Richtung) zum Einsatz. Es wird in content[i] geprüft ob der char den Wert 13(linefeed), 10(linefeed) oder 32(\n) hat.</p> <p>Nun wird geprüft ob ein Absatz vorliegt(content[i] = 13 und conten[i+1] = 10) und demnach i 2 dazu addiert, x 1 subtrahiert sowie continue ausgeführt sodass der jetzige Durchlauf übersprungen wird.</p> <p>Wenn kein Absatz anliegt wird geschaut ob der temporäre string(storage) leer ist und somit zu storage content[i] hinzugefügt. Ebenso wird storage in arr[x,y] reingeschrieben, storage = „“ gesetzt sowie i hoch gezählt.</p> <p>Wenn kein Leerzeichen oder Absatz an content[i] ist, wird content[i] zu storage hinzugefügt, x 1 subtrahiert und i hochgezählt.</p>

<u>Methodenname</u>	<u>Beschreibung</u>
+ <code>InputBackground()</code>	<p>Wird aufgerufen wenn rBtnBackground ausgewählt ist und man auf btnOpen klickt.</p> <p>Es wird ein OpenFileDialog geöffnet und <code>reader</code> den Datei Pfad übergeben. In <code>objecttxt</code> wird die ganze txt Datei eingelesen. Als letztes wird <code>InputAlgorithm(drawImage.Width, drawImage.Height, objecttxt, StaticMembers.backgroundArray)</code> aufgerufen.</p>
+ <code>InputLayer1()</code>	<p>Wird aufgerufen wenn rBtnLayer1 ausgewählt ist und man auf btnOpen klickt.</p> <p>Selbe Prozedur wie bei <code>InputBackground()</code> nur das hier <code>StaticMembers.layer1Array</code> und <code>StaticMembers.layer1Path</code> verwendet wird.</p>
+ <code>InputLayer2()</code>	<p>Wird aufgerufen wenn rBtnLayer2 ausgewählt ist und man auf btnOpen klickt.</p> <p>Selbe Prozedur wie bei <code>InputBackground()</code> nur das hier <code>StaticMembers.layer2Array</code> und <code>StaticMembers.layer2Path</code> verwendet wird.</p>
+ <code>InputLayer3()</code>	<p>Wird aufgerufen wenn rBtnLayer3 ausgewählt ist und man auf btnOpen klickt.</p> <p>Selbe Prozedur wie bei <code>InputBackground()</code> nur das hier <code>StaticMembers.layer3Array</code> und <code>StaticMembers.layer3Path</code> verwendet wird.</p>
+ <code>InputCollision()</code>	<p>Wird aufgerufen wenn rBtnCollision ausgewählt ist und man auf btnOpen klickt.</p> <p>Selbe Prozedur wie bei <code>InputBackground()</code> nur das hier <code>StaticMembers.collisionArray</code> und <code>StaticMembers.collisionPath</code> verwendet wird.</p>

3.5.2 OutputMap

Die Klasse **OutputMap** besitzt den Algorithmus zum einlesen der txt Datei.

Attribute

OutputMap besitzt 4 Attribute.(Tabelle)

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- MainWindow	mainWindow	Objekt der MainWindow Klasse.
- Draw.DrawImage	drawImage	Objekt der DrawImage Klasse.
- StreamWriter	writer	Zum schreiben der txt Datei.
- string	content	Speichert die txt Datei.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
+ OutputAlgorithm (string path , int[,.] arr)	content wird „“ gesetzt und writer wird path zugeordnet. Es werden zwei For-Schleifen für das beschreiben der txt verwendet. Für jeden X Durchlauf wird der Wert von arr[x,y] zu content hinzugefügt, sowie ein Leerzeichen hinzugefügt. Für jeden Y Durchlauf wird zu content Environment.NewLine (neuer Absatz) hinzugefügt. Nun wird writer.Write(content) aufgerufen und somit die txt Datei verändert.

<u>Methodenname</u>	<u>Beschreibung</u>
+ <code>OutputBackground()</code>	<p>Wird aufgerufen wenn rBtnBackground ausgewählt ist und man auf btnSave klickt.</p> <p>Es wird versucht OutputAlgorithm(<code>StaticMemeber.backgroundPath</code>, <code>StaticMembers.backgroundArray</code>) aufzurufen. Wenn es nicht aufgerufen werden kann, wird eine Meldung ausgegeben die bittet den Nutzer eine Datei für den Background auszuwählen.</p>
+ <code>OutputLayer1()</code>	<p>Wird aufgerufen wenn rBtnLayer1 ausgewählt ist und man auf btnSave klickt.</p> <p>Es wird versucht OutputAlgorithm(<code>StaticMemeber.layer1Path</code>, <code>StaticMembers.layer1Array</code>) aufzurufen. Wenn es nicht aufgerufen werden kann, wird eine Meldung ausgegeben die bittet den Nutzer eine Datei für den Background Layer1.</p>
+ <code>OutputLayer2()</code>	<p>Wird aufgerufen wenn rBtnLayer2 ausgewählt ist und man auf btnSave klickt.</p> <p>Es wird versucht OutputAlgorithm(<code>StaticMemeber.layer2Path</code>, <code>StaticMembers.layer2Array</code>) aufzurufen. Wenn es nicht aufgerufen werden kann, wird eine Meldung ausgegeben die bittet den Nutzer eine Datei für den Background Layer2.</p>
+ <code>OutputLayer3()</code>	<p>Wird aufgerufen wenn rBtnLayer3 ausgewählt ist und man auf btnSave klickt.</p> <p>Es wird versucht OutputAlgorithm(<code>StaticMemeber.layer3Path</code>, <code>StaticMembers.layer3Array</code>) aufzurufen. Wenn es nicht aufgerufen werden kann, wird eine Meldung ausgegeben die bittet den Nutzer eine Datei für den Background Layer3.</p>
+ <code>OutputCollision()</code>	<p>Wird aufgerufen wenn rBtnCollision ausgewählt ist und man auf btnSave klickt.</p> <p>Es wird versucht OutputAlgorithm(<code>StaticMemeber.collisionPath</code>, <code>StaticMembers.collisionArray</code>) aufzurufen. Wenn es nicht aufgerufen werden kann, wird eine Meldung ausgegeben die bittet den Nutzer eine Datei für die Collision auszuwählen.</p>

Ordner:

3.6 Tiles

Klassen:

3.6.1 TileMap

3.6.1 TileMap

Attribute

TileMap besitzt 5 Attribute.(Tabelle)

Datentyp	Attributname	Beschreibung
+ Bitmap	bmCollisionTileSet	Bitmap des Collision TileSet.
- int	mouseX	Maus Position X.
- int	mouseY	Maus Position Y.
- int	tileX	Tile Position X.
- int	tileY	Tile Position Y.
- int	id	Wert des Tiles.
- int	currentTileX	Ausgewähltes Tile X.
- int	currentTileY	Ausgewähltes Tile Y.
- int	width	Breite des TileSet.
- int	height	Höhe des TileSet.
- int	tileWidth = 64	Breite einzelnes Tile.
- int	tileHeight = 64	Höhe einzelnes Tile.
+ bool	collisionChecked	Überprüft ob Collision ausgewählt ist.

Methoden

Methodenname	Beschreibung
+ DrawTileMap()	Wird aufgerufen wenn der Nutzer ein TileSet auswählt. Zuerst wird ein OpenFileDialog angezeigt. Nun wird von Nutzer erwartet ein TileSet auszuwählen. Hat der Nutzer ein TileSet ausgewählt so wird es StaticMembers.bmTileMap übergeben. bmCollisionTileSet wird ein TileSet zugewiesen. Als letztes bekommt jedes einzelnes Tile eine eigene ID(i).
+ int TileID()	Es wird die id von dem aktuellen Tile zurückgegeben sowie CurrentTileX/Y gesetzt.
+ DrawSelectedTile()	Aktualisiert das ausgewählte Tile und überprüft ob die Collision oder Standard TileSet ausgewählt wird(collisionChecked).
+ FillAllTiles(int mode)	Füllt alle Tiles mit der ausgewählten ID. mode prüft welcher Layer ausgewählt ist und entscheidet so welches Array ausgefüllt wird.

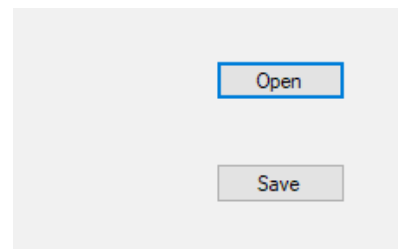
III. Tiles aus TileSet

1. Programmbeschreibung:

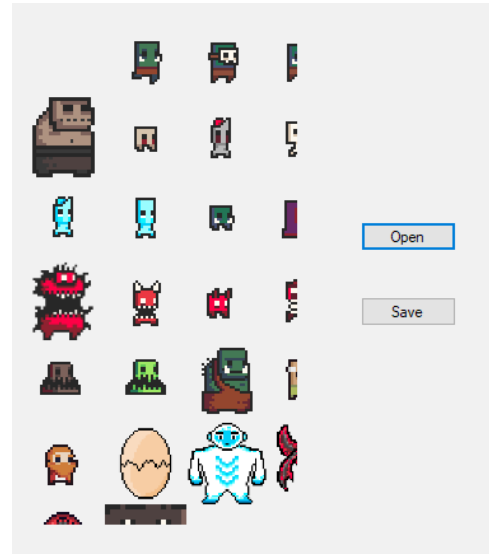
Das Programm TilesAusTileSet wurde gemacht um aus einen TileSet die Tiles in einzelnen png Dateien zu kopieren. Die einzelnen Tiles werden für die Maps von YellowTale benötigt.

2. Programmablauf:

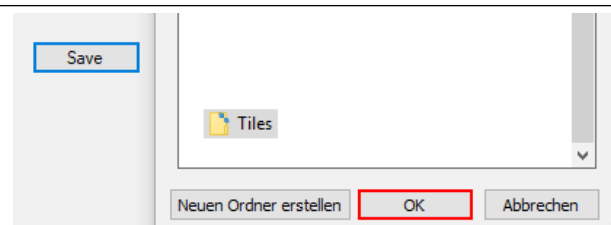
1. Beim Programmstart erscheint eine Form mit 2 Buttons.



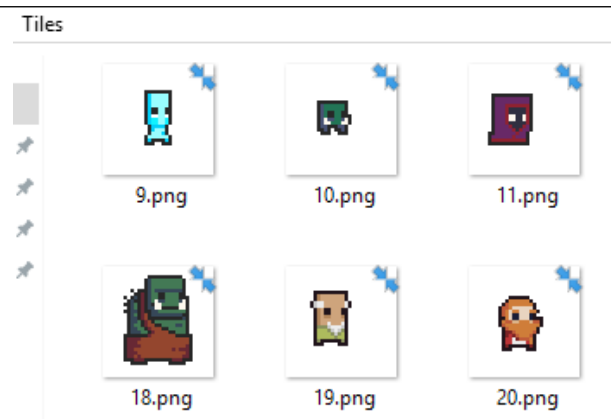
2. Als erstes wird mit Open ein TileSet ausgewählt. Es erscheint in einer PictureBox.



3. Danach soll gespeichert werden, demnach wird auf Save gedrückt. Jetzt wird der Abgefragt in welchen Ordner die einzelnen Dateien gespeichert werden.



4. Fertig. Alle einzelne Tiles wurden mit einer Nummerierung abgespeichert.



3. Klassen:

3.1 Form1

Attribute

<u>Datentyp</u>	<u>Attributname</u>	<u>Beschreibung</u>
- Bitmap	<code>bmTileMap</code>	Speichert die ganze TileMap.
- Bitmap	<code>bmSpeicher</code>	Speichert das ausgewählte Tile.

Methoden

<u>Methodenname</u>	<u>Beschreibung</u>
- <code>btnOpen_Click</code> (object <code>sender</code> , EventArgs <code>e</code>)	Es erscheint ein OpenFileDialog womit man die TileMap auswählen kann. Die ausgewählte TileMap wird dann in <code>pictureBox1.Image</code> geschrieben.
- <code>btnSave_Click</code> (object <code>sender</code> , EventArgs <code>e</code>)	<p>int <code>i</code> wird als Zähler verwendet damit jedes Tile eine eigene Nummer bekommt. string <code>ordnerPath</code> speichert den Ordner Pfad. In string <code>path</code> wird der Datei Pfad gespeichert.</p> <p>Als erstes wird abgefragt in welchen Ordner seine Tiles speichern möchte. Somit wird <code>ordnerPath</code> verändert. Nun wird mit einer verschachtelten Schleife die einzelnen Tiles in <code>bmSpeicher</code> gespeichert. Mit <code>bmSpeicher.Save</code> wird somit das Tile mit der zugehörigen Nummer gespeichert. Als letztes wird <code>i</code> hochgezählt. Dieser Durchgang wird solange wiederholt, bis <code>y</code> größer als <code>bmTileMap.Height</code> ist.</p>