## Informatik Projekt Joris Ludwig

Für unsere diesjährige Informatik-Grundkursnote waren wir erneut beauftragt, ein Projekt in einer Objekt orientierten Programmiersprache zu planen und zu entwickeln.

### Grober Ablauf:

Ich habe mich dazu entschieden ein Spiel mit Top-Down-, beziehungsweise Vogel-Perspektive zu erstellen, in dem man gegen verschiedene Monster antreten muss. Die Monster erscheinen in Wellen, welche mit der Zeit immer größer werden. Um mit dem ansteigenden Schwierigkeitsgrad klarzukommen, soll der Spieler nach jeder erfolgreich abgeschlossenen Welle, die Möglichkeit haben Upgrades bei einem Händler zu erwerben. Die hierfür nötige Währung, wird durch das Bestehen der Wellen dem Spieler zugeschrieben. Außerdem sollen verschiedene Menus, sowie Musik und Sound hinzugefügt werden.

### Das Spiel:

In vergangenen Projekten habe ich für die Entwicklung von GUI-Anwendungen die *Simple DirectMedia Layer 2.0 development library* (kurz: SDL2) verwendet. Aufgrund meiner Erfahrungen mit dieser API, habe ich mich auch bei diesem Projekt dazu entschieden, mit ihr zu arbeiten.  
Im Folgenden werde ich die Struktur und den Aufbau meines Programms, über die verschiedenen Klassen erläutern.

class Interface:

Diese Klasse symbolisiert das Fenster meiner Anwendung, dieses wird benötigt, um Inhalte auf meinem PC zu rendern. Ich erstelle das Fenster mit der Hilfe von SDL2, die Verwaltung des Fensters erfolgt anschließend in dieser Klasse.  
Da ich in keinem Fall mehr als nur ein Fenster erstellen möchte, habe ich mich dazu entschieden die Klasse Interface im singleton-pattern zu programmieren. Das einfache Erstellen von Objekten dieser Klasse wird unmöglich gemacht, indem ich alle Konstruktoren privatisiere. Stattdessen füge ich eine Public Methode hinzu, welche die Referenz eines statisch angelegten Objekts der Klasse Interface zurückgibt. Dadurch kann immer nur ein einziges Interface existieren.

class GameHandler:

Die Klasse GameHandler verwaltet den äußeren Ablauf meines Spiels, dazu gehören unter anderem Eingaben des Nutzers, Eingrenzung der Frame rate, errechnen der Delta-Time, Initialisierung der Welt sowie Verwaltung der Texturen und das Öffnen von Menüs.

class SoundHandler:

Der SoundHandler ist, genau wie das Interface, im singleton-pattern programmiert.  
Wie ihr Name impliziert, übernimmt diese Klasse die Aufgabe, Audiodateien abzuspielen.  
Dafür habe ich verschiedene Funktionen, welche von überall in meinem Projekt aufgerufen werden können.

class MenuManager:

Startseite, Pause, Shop und Game Over sind alles Menüs, welche während der Anwendung geöffnet werden können.  
Die Funktionalität und das Rendern dieser Menüs, werden von der Klasse MenuManager übernommen.

class Clock:

Diese simple Klasse soll dabei helfen, bestimmte Aktionen mit einem Cool down zu versehen.  
So wird zum Beispiel verhindert, dass der Spieler jede Sekunde angreifen kann oder dass Gegner nach jedem Frame eine neue Route einschlagen. Die Funktion bool checkClockState() soll hierbei eine Stoppuhr imitieren, die bei abgelaufener Zeit True zurückgibt.

class Entity:

Eine Entity ist ein einfaches Objekt oder ein Gegenstand. Es verfügt über Koordinaten und Größenangaben.

class Body:

Abgeleitet von der Klasse Entity ergibt sich die abstrakte Klasse Body, sie stellt einen Körper dar. Die Klasse Body ist abstrakt, da ein Körper immer zu einem Monster oder Spieler gehören soll. Die Klasse Body verwaltet Animationen, Leben und Schaden von Gegnern und Spielern.

class Player:

Wie der Name vermuten lässt, handelt es sich bei dieser Klasse um den Spieler. Er hat immer dasselbe Aussehen, seine Position ändert sich nie. Der Spieler befindet sich immer in der Mitte des Fensters. Lediglich die Welt bewegt sich um ihn herum, zusammen mit der Animation, gibt dies die Illusion, dass man sich fortbewegt.

class Enemy:

Genau wie der Spieler sind auch die Gegner von der Klasse Body abgeleitet.  
Sie bekommen ihr Aussehen vom GameHandler zugewiesen. Anders als der Spieler bewegen sie sich von alleine. Hierzu habe ich mir einen Algorithmus ausgedacht, solange wie die Gegner den Spieler nicht entdeckt haben, befinden sie sich im „Idle“ Modus. Haben sie eine gerade Sichtlinie zum Player, so starten sie eine Jagd und verfolgen den Spieler.

class Beetle:

Neben den regulären Gegnern soll es auch einen besonderen Käfer geben, welcher über Hindernisse hinweg fliegen kann. Die Klasse Beetle ist von der Klasse Enemy abgeleitet. Die Funktionen für das Pathfinding sind so überschrieben, dass das gewünschte Ergebnis erzielt wird.

class Effect:

Unter einem Effekt kann man sich zum Beispiel Explosionen vorstellen, sie haben Koordinaten, Texturen und auch Animationen. Die Klasse Effect wird von der Klasse Entity abgeleitet.

class TradingPost:

Die Klasse TradingPost ist abgeleitet von Entity, ein TradingPost erscheint nach jeder abgeschlossenen Welle, hierfür wird auch ein Effekt verwendet. Der TradingPost verkörpert den Händler, er erscheint an einer zufälligen Stelle in der Nähe des Spielers und durch die Interaktion mit ihm, wird der Shop geöffnet. Der TradingPost speichert, ab welche Upgrades bereits gekauft wurden.

class Vicinity:

Diese Klasse stellt eine optische Schicht des Levels dar, dadurch sieht der Spieler überhaupt erst, dass sich die Welt an ihm vorbeibewegt.

class World:

Die Klasse World ergibt sich abgeleitet von der Klasse Vicinity, die Welt ist das eigentliche Level. Sie hat ein eigenes Aussehen (Der Untergrund, auf dem sich der Spieler bewegt) und ein Objekt Vicinity (was alles repräsentiert, was sich über dem Spieler befindet).

Die Welt hat Pointer auf Spieler, Gegner und Entitys. Sie ist dafür verantwortlich, dass sich Gegner und Spieler ordnungsgemäß bewegen und verhalten. Sie prüft auch, ob Gegner gelöscht werden können oder ob der Spieler besiegt wurde.

class VirtualEnemy:

Mit der Klasse VirtualEnemy ist es möglich das Spiel im Coop zu spielen. Auf dem Server befinden sich „echte“ Enemies, deren Daten werden an den Client gesendet und es wird ein VirtualEnemy erstellt. Ein VirtualEnemy ist also nur ein Repräsentant eines echten Enemy auf einem Server.

class PlayerTwo:

Genau wie VirtualEnemy einen echten Enemy repräsentiert, repräsentiert ein PlayerTwo einen echten Spieler auf einem anderen Computer. Hierbei kann es sich allerding sowohl um den Spieler des Clients handeln als auch um den Spieler des Servers.

class WorkThread:

Meine eigne Thread-Klasse, sie ist rein virtuell. Durch Aufrufen der startThread() Methode wird ein Thread erstellt und die run() Methode der Abgeleiteten Klasse aufgerufen.

class GameClient:

Die Klasse GameClient wird von WorkThread abgeleitet, ein GameClient kommuniziert mit einem GameServer und erhält so die Daten für die VirtualEnemies und den PlayerTwo.  
Außerdem werden andere wichtige Daten übertragen, wie zum Beispiel den Status der aktuellen Welle.

class GameServer:

Die Klasse GameServer wird ebenfalls von WorkThread abgeleitet, der GameServer ist das Gegenstück zum GameClient. Er bietet alle Informationen, welche der Client benötigt.  
Die Kommunikation erfolgt über Sockets.

Das waren alle eigens von mir erstellten Klassen kurz und vereinfacht vorgestellt, eine detailliertere Ansicht erhält man im anbei liegenden Klassen-Diagramm.

## Material 1 [Serverseitiges Netzwerk-Protokoll]

> 2 //Anzahl von Gegnern

**> 1** //ID des Gegners  
> mantis //Typ des Gegners  
> -8099 //Position auf der X-Achse  
> 2053 //Position auf der Y-Achse  
> hit //Modus des Gegners  
> 4 //Aktueller Sprite des Gegners  
> 32 //Y-Position des Sprites auf dem Sheet ( Entscheidet die Richtung des Gegners)

**> 2** //ID des Gegners  
> maggot //Typ des Gegners  
> -45 //Position auf der X-Achse  
> -1003 //Position auf der Y-Achse  
> walk //Modus des Gegners  
> 2 //Aktueller Sprite des Gegners  
> 0 //Y-Position des Sprites auf dem Sheet ( Entscheidet die Richtung des Gegners)

> 770 //Position des Spielers auf der X-Achse  
> 42 //Position des Spielers auf der Y-Achse  
> walk //Modus des Spielers  
> 1 //Aktueller Sprite des Spielers  
> 0 //Richtung des Spielers (Links oder Rechts)

> 100 //Die aktuelle Welle

> false //Ob der zweite Spieler getroffen wurde (true oder false)

< 770 //Position des zweiten Spielers auf der X-Achse  
< 42 //Position des zweiten Spielers auf der Y-Achse  
< idle //Modus des zweiten Spielers  
< 1 //Aktueller Sprite des zweiten Spielers  
< 0 //Richtung des zweiten Spielers (Links oder Rechts)

> true //Ob das Geräusch für einen getroffenen Gegner abgespielt werden muss  
> false //Ob das Geräusch für einen getroffenen Spieler abgespielt werden muss

< 1 //Ob der zweite Spieler einen Angriff startet

> false //True, wenn der Merchant erscheinen soll  
< false //Gibt an, ob Spieler 2 noch mit dem Merchant handelt

< isRunning //Zustand des Spieles auf dem Client  
> isRunning //Zustand des Spieles auf dem Server