Simulation von Boids
nach Craig Reynolds

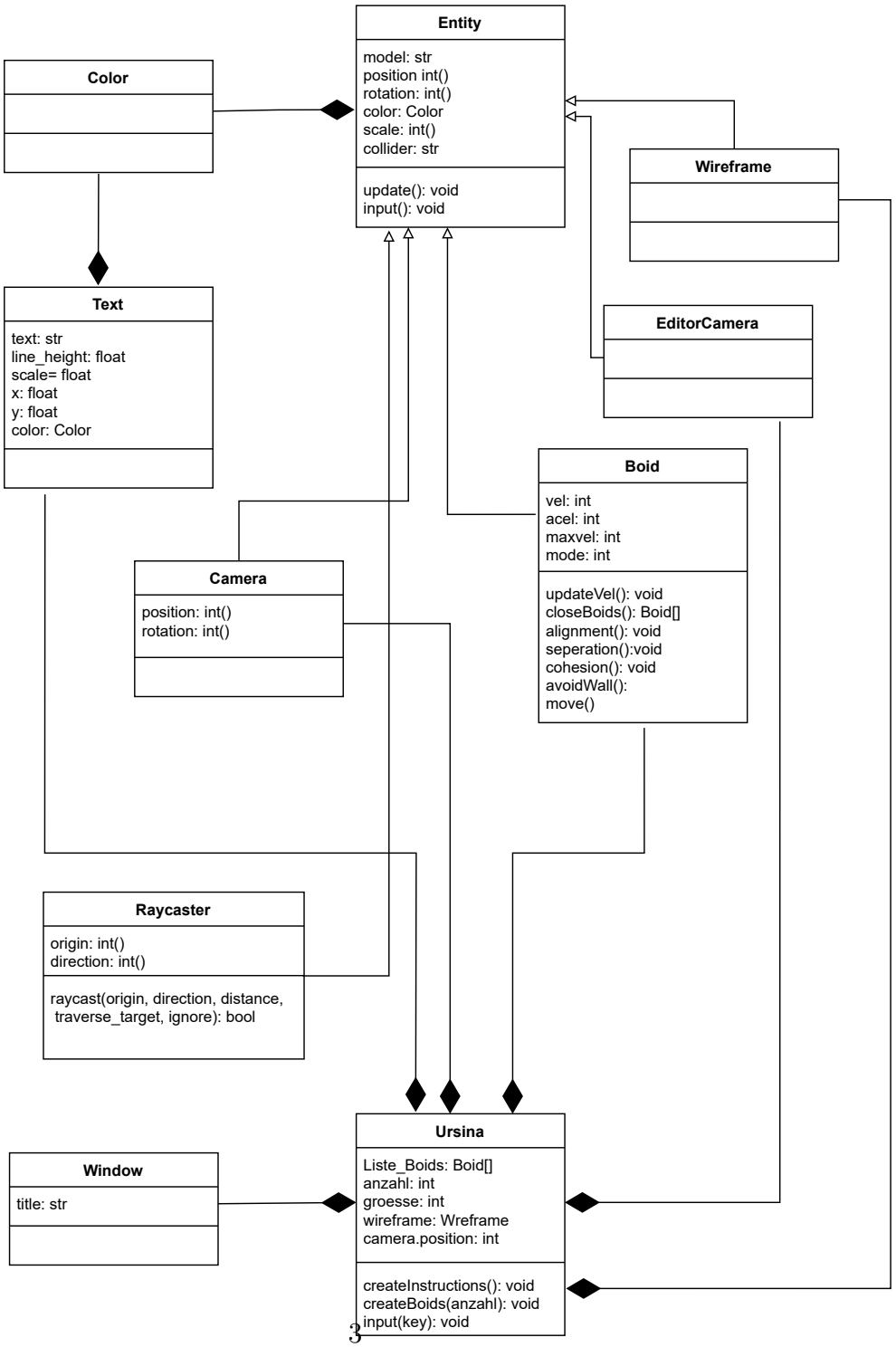
Oliver Fritzler

February 6, 2022

Inhaltsverzeichnis

Contents

1 UML-Diagramm



2 Grundidee

2.1 Umgebung

Die Programmiersprache ***Python*** war vorgegeben, die Bibliothek zur grafischen Darstellung war frei überlassen. Aufgrund der einfachen und verständlichen Syntax, fiel die Auswahl auf die ***ursina engine***, welche auf der ***panda3D engine*** basiert. Ursina ermöglicht es neben vorgegebenen Modellen auch blender-Dateien als Modelle für die einzelnen Objekte zu verwenden. Diese Objekte werden dargestellt und durch spezifische Funktionen (**update()**, **move()**, **input()**) jedes Bild aktualisiert. Dadurch kann man sowohl die Position durch Addition verändern, als auch Eingaben des Benutzers verarbeiten.

2.2 Darstellung des Raumes

Zur Darstellung des Raumes werden 12 Objekte in Form eines Würfels erstellt, jedoch werden diese dann so skaliert, dass sie die Form eines Rechteckes annehmen. Diese werden dann so angeordnet, dass sie die Kanten eines Würfels bilden. Des Weiteren werden die Seitenflächen erstellt. Dabei werden wie davor 6 Objekte in Form eines Würfels erstellt. Nur werden diese jetzt so skaliert, dass sie eine quadratische Fläche bilden. Um diese unsichtbar zu machen wird der Sichtbarkeitswert auf Null gesetzt.

2.3 Allgemeines zu den Boids

Boids sind Körper, die sich Bewegen. Zur Veranschaulichung der Richtung sollten die Boids einen Körper mit einer Spitze haben, wie zum Beispiel ein Kegel. Da ein Kegel sehr viele Seiten hat, die alle jedes Bild berechnet werden müssen und dadurch die Performance senken, sollte man auf einen einfacheren Körper mit Spitze zurückgreifen, wie eine Pyramide. Diese Körper bewegen sind jedoch nicht chaotisch, da sie sich an ***drei Regeln*** handeln, die später erklärt werden.

Verhalten an Wänden Das Verhalten der Boids an den Wänden kann man auf verschiedene Weise gestalten. In dieser Simulation wurden drei davon versucht umzusetzen.

1. **Warp**
   Bei dieser Umsetzung werden die Boids bei Kontakt mit einer Wand auf die gegenüberliegende Seite "teleportiert".

2. **Wände vermeiden**
   Dabei drehen die Boids vor Kontakt mit einer Wand ab.

3. **PONG**
   Hier verhalten sich die Boids, wie der Ball im Spiel **"PONG"**, d.h. sie bewegen sich auf die Wand in einem bestimmten Winkel zu und verlassen diese Wand dann in dem gleichen Winkel, nach dem *Einfallswinkel-gleich-Ausfallswinkel-Prinzip*

2.4 Regeln

Boids sind Objekte die sich in einem Raum bewegen. Dabei verfolgen sie drei Grundregeln:

- **Seperation**
  Diese Regel besagt, dass jeder einzelne Boid versucht, keinen anderen Boid zu treffen.

- **Alignment**
  Diese Regel besagt, dass jeder Boid versucht, in die selbe Richtung wie ein anderer sich zu bewegen. Dadurch entsteht ein sogenannter **"Flock"** also ein Schwarm von Boids.

- **Coheseion**
  Diese Regel besagt, dass die Boids in die Mitte des Schwarms steuern.

2.5 Eingabe

Der Benutzer soll durch Bewegen der Maus die Kamera rotieren können und durch Tastendruck die Kamera bewegen können. Außerdem soll es ihm ermöglicht werden die Anzahl der Boids und deren Verhalten zu ändern.

3 Umsetzung

3.1 Allgemein

Aufrgund der Umsetzung der Eigesnchaften ist es nicht möglich die Darstellungsebene und die Informationsebene in verschiedene Dateien aufzuteilen. Es würde ein Import-Fehler auftreten, da die Liste aus der Darstellungsebene in die Informationsebene importiert werden müsste und die Klasse aus der Informationsebene in die Darstellungsebene.

3.2 Fenster

Mit dem Aufrufen der Klasse ***Ursina*** wird ein Fenster erschaffen, welches am Ende ausgeführt wird, mit all den benötigen Grafiken. Diesem Fenster kann man einen Titel vergeben, welche dann als Task angezeigt wird. Diese Klasse hat wie jede andere Klasse in Ursina die Funktion ***input()***, mit deren Hilfe man die Eingabe des Benutzers verarbeiten kann, die in dieser Umsetzung ***hier*** nachzulesen sind. Um dem Benutzer die Nutzung der Simulation zu vereinfachen, erstellt man in der Funktion ***createInstruction()***, mit der Klasse ***Text*** eine Beschreibung und skaliert und positioniert sie so, dass sie nicht im Vordergrund liegt.[Code]

3.3 Raum

Die Klasse ***Wireframe*** wird erstellt, welche von der ***Entity***-Klasse erbt. Diese Klasse beinhaltet insgesamt 18 andere Entitys. 12 davon sind Würfel die so skaliert und positioniert werden, dass sie die Kanten eines großen Würfels bilden. Die anderen 6 sind ebenfalls Würfel, welche jedoch so skaliert werden, dass sie quadratischen Flächen formen. Diese Flächen werden dann so positioniert, dass die Seitenflächen des großen Würfels bilden. Da diese Seitenflächen nicht nötig zu sehen sind, kann man entweder den α-Wert der Farbe oder den α-Wert einzeln auf 0 setzen, beide Werte sind die Sichtbarkeitswerte. Bei dieser Umsetzung wird letzteres angewandt. Sowohl den Kanten, als auch den Seitenflächen wird eine Box als Kollisionskörper unter *collider* zugeteilt, damit die Boids daran abprallen können[Code]
Warum diese Klasse erstellt werden muss, wird im Teil ***seperation()*** erklärt.

3.4 Boids

Die Darstellung der Boids erfolgt durch die Implementierung der gleichnamigen Klasse ***Boid***. Diese Klasse erbt von der Klasse ***Entity***, welche

laut den Entwicklern von Ursina die *"god class"* ist. Den Boids wird eine selbsterstellte .blender-Datei als Model gegeben. Dieses Model gleicht einer Pyramide. Die Farbe dieser Pyramide wird mithilfe des *random_color()* Befehls der Klasse ***color*** festgelegt, welche innerhalb *self.color* gespeichert wird. Des Weiteren speichert der Boid seine Position sowohl in einem Tuple unter *self.position* und als auch in drei einzelnen Variablen *self.x*, *self.y* und *self.z*. Um die Bewegung zu ermöglichen wird dem Boid eine Ausrichtung gegeben, die er unter *self.rotation* speichert. Um die Geschwindigkeit zu variieren wird die Geschwindigkeit, die Beschleunigung und die zu maximal erreichende Geschwindigkeit unter *self.vel*, *self.acel* und *self.maxVel* gespeichert. Um das Verhalten an den Wänden zu ändern, speichert der Boid den aktuellen Modus unter *self.mode*. Damit der Boid die **Raycasts** der anderen Boids auslösen kann, wird dem Boid ein Körper zugeteilt der dies ermöglicht, welcher unter *self.collider* gespeichert wurde. Den Boids wird eine Box zugeteilt, da die von den vorgegebenen Beispielen am passendsten sind. Nach dem Konstruktor werden die Getter, die die Werte der Eigenschaften zurückgeben und die Setter, die die Werte der Eigenschaften überschreiben, deklariert. Im folgenden werden die weiteren Funktionen erklärt:

updateVel() Damit sich die Boids nicht mit der selben Geschwindigkeit bewegen, wird hier die Geschwindigkeit mithilfe der Addition der Beschleunigung *self.acel* erhöht.

closeBoids() Damit der Boid sich an die vorgegebenen Regeln halten kann, muss er wissen welche Boids sich in seiner Nähe befinden. Dafür wird in dieser Funktion die Liste in der alle Boids gespeichert wurden (siehe ***createBoids()***) durchgegangen und überprüft ob die Distanz in zwischen dem Boid und dem zu vergleichenden Boid unter einem bestimmten Wert liegt. Falls dies zu trifft wird der zu vergleichende Boid in eine Liste hinzugefügt, die alle Boids in der Nähe beinhaltet. Diese Liste wird am Ende zurückgegeben. Mithilfe dieser Funktion wird dem Boid ein gewisser Sichtbereich gegeben, da er nicht weiter als der festgelegt Wert "sehen" kann.[Code]

alignment() Diese Funktion ist dazu da um die 2. Regel ***"Alignment"*** zu befolgen. Dafür werden anfangs die Boids in der Nähe mithilfe der ***closeBoids()***-Funktion bestimmt, und in der *proximity*-Liste gespeichert. Darauf werden alle Werte die etwas mit der Bewegung zu tun haben, d.h. sowohl die Rotationswerte als auch alle Werte der Geschwindigkeit des Boids gespeichert. Das Selbe wird nun mit jedem Boid gemacht der in der *proximity*-Liste gespeichert ist. Für jeden gespeicherten Wert wird der Durchschnitt berechnet. Nach der Berechnung werden die Eigenschaften mit diesen Durch-

schnittswerten überschrieben.

seperation() Mithilfe dieser Funktion versucht der Boid die 1. Regel ***"Seperation"*** zu befolgen. Bei dieser Funktion gibt es 2 Umsetzungsversuche:

- **Versuch 1**
  Wiedermal werden die Boids in der Nähe in der *proximity*-Liste durch die ***closeBoids()***-Funtktion gespeichert. Nun wird die Distanz zwischen dem Boid und dem zu vergleichenden Boid überprüft und falls diese in einem kleinen Bereich liegt, wird die Differenz des Abstands und der eigenen Position multipliziert mit 0.5, der eigenen Position hinzuaddiert damit die beiden Boids an Abstand gewinnen.

- **Versuch 2**
  Bei diesem Versuch werden 5 ***Raycasts*** erstellt, die nach vorne, hinten, oben, rechts und links gerichtet sind. Diese haben eine sehr kurze Distanz und ignorieren die ***Wireframe***-Klasse, da sie nur von Boids ausgelöst werden können. Falls einer dieser ***Raycasts*** ausgelöst wird, wird die Rotation in die entgegengesetzte Richtung durch Addieren des Produkts aus Richtung und Geschwindigkeit. Diese Umsetzung ist der Grund für die Erstellung der ***Wireframe***-Klasse

Anmerkung: Der Versuch 1 scheint nicht zu funktionieren, da es oft vorkommt, dass Boids ineinander liegen. Um die zu umgehen, habe ich sowohl versucht den Vorfaktor zu ändern, was im chaotischen Rotieren resultierte, als auch durch Veränderung der Differenz, welches nichts änderte Der Versuch 2 ist aufgrund der Raycasts sehr ressourcenfressend, weswegen ich empfehle den Versuch 2 auszukommentieren und den Versuch 1 zu entkommentieren. Des Weiteren kann das nicht Funktionieren an der ***cohesion()***-Funktion liegen, die als Gegenspieler agiert.

cohesion() Diese Funktion dient dazu, dass der Boid die 3. Regel ***"Cohesion"*** befolgt. Sie funktioniert ähnlich wie ***seperation()***, nur leicht abgewandelt. Anstatt die Distanz mit jedem Boid abzugleichen, wird hier zuerst ein Mittelpunkt aller Boids eines *Flocks* berechnet. Falls die Distanz zwischen diesem Mittelpunkt und der Position des Boids über einem bestimmten Wert liegt, wird die Differenz des Mittelpunkts und der eigenen Position multipliziert mit 0.5, der eigenen Position abgezogen. Dadurch bewegt sich der

Boid weiter in die Mitte des **Flocks**.

avoidWall() Diese Funktion verändert das Verhalten der Boids in der Nähe der Wände. Dafür erstellt der Boid einen **Raycast**, der in Bewegungsrichtung gerichtet ist. Falls dieser **Raycast** etwas berührt werden vier weitere erstellt, nur dieses Mal nach nach vorne, hinten , rechts und links gerichtet. Danach wird nach dem Herausfinden der *kleinsten* Differenz entschieden, in welche Richtung die Rotation durch Multiplikation verändert wird. **Anmerkung:** Bei dem Versuch es zu implementieren, ist mir aufgefallen, dass die Boids durch die Wände schreiten und nie umdrehen. Die einzige logische Möglichkeit, womit ich dies erklären, kann ist das durch die ***move()***-Funktion die Boids die Wände überspringen und dadurch der **Raycast** nicht aktiviert wird. Man könnte es versuchen zu lösen, indem man bei Überschreitung die Position überschreibt, damit die **Raycasts** eventuell auslösen.

move() Diese Funktion ermöglicht die Fortbewegung des Boids durch Addition des Produkts aus der Bewegungsrichtung und der Geschwindigkeit dividiert durch 1000. Des Weiteren wird die Geschwindigkeit erhöht, durch den Aufruf der ***updateVel()***-Funktion. In dieser Funktion wird neben der Bewegung auch das Verhalten an den Wänden bestimmt durch Abgleichung des Moduswertes.

- Falls dieser Wert auf 1 liegt, wird der ***"Warp"*** verwendet, welcher die aktuellen Positionswerte mit den Grenzwerten abgleicht. Falls die Positionswerte diese Grenzwerte überschreiten, wird die Position auf die gegenüberliegende Seite gelegt.
- Falls der Moduswert auf 2 liegt, wird das ***"Wände vermeiden"***-Verhalten aufgerufen und die ***avoidWall()***-Funktion aufgerufen.
- Falls der Wert auf 3 liegt, wird der ***"PONG"***-Modus aufgerufen. Dabei wird versucht bei Überschreitung der Grenzwerte bestimmte Rotationswerte nach dem "Einfallswinkel entspricht Ausgleichswinkel"-Prinzip zu ändern, dies hat jedoch mithilfe der Rechnung *-1-180 oder anderen Methoden nicht funktioniert, da es zu viele Spezialfälle gibt um es allgemein zu verfassen, und außerdem vermute ich, dass es ebenfalls aufgrund des Gimble-Locks ebenfalls nicht funktioniert.

update() Da diese Funktion jedes Bild automatisch aufgerufen wird, ruft sie alle nötigen Funtkonen auf. D.h. ***move()***, ***cohesion()***, ***seperation()*** und ***alignment()***.

3.5 Funktionen im Fenster

createBoids() Diese Funktion wird vom ***Fenster*** aufgerufen und erstellt für die festgelegt Anzahl ***Boids*** mit einer festgelegten Größe und speichert diese in *Liste_Boids*. Dabei wird dem Boid immer der Modus **1** gegebenen. Bei dem Versuch es mit einer Variable zu lösen, die den aktuellen Modus beinhalten, kam kein brauchbares Ergebnis heraus

3.6 Eingabe

input() Wie auch die ***update()***-Funtkion wird die input()-Funktion jedes Bild automatisch aufgerufen. Diese gleicht ab welche Taste gedrückt wurde und führt Fälle aus.
Gedrückte Taste:

1. **+/-** Beim Drück dieser Tasten, wird entweder ein ***Boid*** erstellt und der Liste angehängt oder der letzte Boid wird deaktiviert und aus der Liste gelöscht. Er muss deaktiviert werden, da er sonst immer noch dargestellt werden würde.

2. **1,2,3** Beim Druck einer dieser Tasten wird der das ***Verhalten*** jedes einzelnen Boids durch Änderung der Modus-Eigenschaft in einer for-Schleife geändert.

4 wichtigste Funktionen

4.1 Fenster

```
#Fenster wird erstellt
app = Ursina()
#Fenstertitel wird festgelegt
window.title = "Boids Simulation"
#Fenster wird ausgefuehrt
app.run()
```

4.2 Kamera

```
#Kameraposition wird veraendert um den Bereich von Anfang
    an zu sehen
camera.position = (0,10,-350)
#EditorCamera ermoeglicht die Veraenderung der Kamera
EditorCamera()
```

4.3 Wireframe

```
#Erzeugung des Wuerfels
wf1 = Entity(model = "cube", collider = 'box', position =
    (0, -51, -51), scale_x = 102)
...
#Erzeugung der Waende
w1 = Entity(model = 'cube', collider = 'box', position =
    (0,0,-52),scale=(110,110,0), color = color.red, alpha =
    0)
```

4.4 closeBoids()

```
    closeBoids(self):
    close = []
        i    Liste_Boids:
           i != self:
        #wenn die Distanz zwischen dem self.Boid und dem zu
            vergleichenden Boid unter 5 liegt wird der
           vergleichende Boid in die Liste mit den Boids in
            der Naehe hinzugefuegt
```

```
6                       distanceBoids = distance(self.position,
                            i.position)
7                       distanceBoids < 5:
8                           close.append(i)
9               close
```

4.5 createBoids()

```
1       createBoids(anzahl):
2   #Boids werden erstellt und in der Liste gespeichert
3       i           (anzahl):
4   temp = Boid(randint(-30,30), randint(-30,30), randint(-30,
        30), randint(0,360), randint(0,360), randint(0,360),
        uniform(0.0, 100.0), uniform(0.0, 10.0), 300.0, 1,
        groesse)
5   Liste_Boids.append(temp)
```

5 Quellennachweis