Belegarbeit Teil II
zur Veranstaltung Computergrafik und Visualisierung II

Günther, Emanuel s76954 - Leopold, Michael s76904 - Fleischer, Kassandra s77282

SoSe 2019

1 Aufgabenstellung

1. **Integration von Schwarmverhalten [20 Punkte]**
   Integrieren Sie die notwendigen Methoden für die Realisierung eines Schwarmverhaltens. Achten Sie dabei auf ein sinnvolles Konzept. Überlegen Sie sich ein geeignetes Projekt, in dem Sie Schwarmverhalten einsetzen wollen. Achten Sie auf die individuellen Bewegungs- und Handlungsmöglichkeiten und adaptieren Sie Ihre Implementierung des Schwarmverhaltens auf das Projekt.
2. **Shader-Visualisierung [10 Punkte]**
   Visualisieren Sie einen Schwarm im FragmentShader. Dazu könnten Sie beispielsweise die Individuen durch einfache OpenGL-Primitive konstruieren.
3. **Shader-Berechnungen[20 Zusatzpunkte und jede Menge Ruhm und Ehre]**
   Als Herausforderung können Sie versuchen, den Code aus der vorhergehenden Aufgabe so anzupassen, dass die update-Methode (die Aktualisierung aller notwendigen Kräfte und damit auch der einzelnen Positionen) durch einen VertexShader realisiert werden. Wer das vorstellen kann, wird in die CG-II-Hall-of-Fame aufgenommen.

2 Lösung

Die vollständigen Lösungen wurden per Email abgegeben. Hier ein Auszug der verwendeten Konzepte und Shader.

Die Lösung wurde in `Eclipse 2019-03` entwickelt.

2.1 Idee/Konzept

Unsere Idee eine Vogelsimulation mit Spielcharakter. Es werden ein Schwarm von Vögeln sowie ein Raubvogel simuliert.

Die Steuerung des Raubvogels erfolgt über die Tastatur ({W, A, S, D} oder Pfeiltasten). Der Vogelschwarm wird mit der Maus gesteuert.

2.1.1 Vogelschwarm

Auf jeden einzelnen Vogel des Schwarms wirken drei Kräfte, die das Schwarmverhalten simulieren. Diese sind Seperation, Angleichung und Zusammenhalt. Seperation bewirkt, dass jeder Vogel seinen eigenen Freiraum besitzt. Durch die Angleichung bewegen sich alle Vögel in eine grundsätzlich ähnliche Richtung. Der Zusammenhalt sorgt dafür, dass die Vögel im Schwarm bleiben.

Zusätzlich zu den Kräften des Schwarmverhaltens werden die Vögel noch anderweitig beeinflusst. Zum einen bewegen sich die Vögel immer zur aktuellen Mausposition hin. Damit lässt sich der Schwarm steuern. Zum anderen fliehen die Vögel vor dem Raubvogel. Dieser stellt einen Fressfeind dar.

Um einen natürlichen Bewegungsfluss zu simulieren, müssen alle einwirkenden Kräfte gewichtet werden. Die Seperationskraft ist wesentlich stärker als die der Angleichung oder die des Zusammenhalts. Die Kräfte der Maus und des Raubvogels besitzen etwa die selbe Gewichtung wie die des Zusammenhaltes. Die Vögel bewegen sich immer mit maximaler Geschwindigkeit.

Die Visualiserung der Vögel erfolgt über Dreiecke. Sie werden entsprechend ihrer jeweiligen Bewegungsrichtung rotiert.

2.1.2 Raubvogel

Auf dem Raubvogel wirken die Kräfte der Steuerungstasten. Außerdem bewegt sich der Raubvogel immer mindestens mit Minimal-, höchstens aber mit Maximalgeschwindigkeit.

Die Visualisierung des Raubvogels erfolgt als geladenes 3D-Modell. Dieses wird, genau wie bei der Darstellung der Vögel, entsprechend der Bewegungsrichtung rotiert.

2.2 Details zur Implementierung

Nachfolgend werden wesentliche Auszüge aus den wichtigsten Klassen des Projektes vorgestellt.

2.2.1 Klasse `Vogelsimulation`

Die Klasse `Vogelsimulation` stellt den Einstiegspunkt in das Projekt dar. In ihr werden das LWJGL-Fenster initialisiert, die Shader geladen, kompiliert und verlinkt, der Vogelschwarm und der Raubvogel angelegt sowie der Renderloop samt Transformationen ausgeführt.

2.2.2 Klasse `Vogelschwarm`

In der Vogelschwarm-Klasse werden alle Vögel eines Schwarmes verwaltet. Sie ist mithilfe des Singleton-Patterns implementiert, d.h. es gibt nur eine Instanz dieser Klasse. Folgende Methoden implementieren das Pattern:

```
1  private Vogelschwarm() {
2      schwarm = new ArrayList<Vogel>();
3  }
4  public static Vogelschwarm getInstance() {
5      return exemplar;
6  }
7  public Object clone() throws CloneNotSupportedException {
8      throw new CloneNotSupportedException("Es gibt nur EINEN Schwarm.");
9  }
```

Zur Verwaltung der Vögel existieren die Methoden `update()` und `render()`. Sie leiten den jeweiligen Befehl an alle Vögel des Schwarms weiter.

2.2.3 Klasse `Vogel`

Jeder Vogel erbt von der Klasse BeweglichesObjekt. Unter anderem besitzt er eine boolsche Membervariable `gefressen`, die auf `true` gesetzt wird, wenn der Vogel vom Raubvogel gefressen wurde. Außerdem sind die Methoden `update()` und `render()` implementiert, die jeweils nur ausgeführt werden, wenn der Vogel noch nicht gefressen wurde. Die Methode `update()` leitet den Update-Befehl in diesem Fall an das entsprechende Schwarmverhalten weiter.

2.2.4 Klasse `Schwarmverhalten`

Im Schwarmverhalten werden alle Kräfte berechnet, die zur Berechnung der nächsten Position notwendig sind. Dazu gehören zum einen Methoden für das Schwarmverhalten (`seperation()`, `angleichung()`, `zusammenhalt()`). Zum anderen sind das Methoden zur Mausverfolgung und zur Flucht vor dem Raubvogel.

Bei der Kraft zur Flucht vor dem Raubvogel wird der Vektor zwischen Raubvogel und Vogel berechnet und normiert.

```
1  public Vektor3D flieheVorRaubvogel() {
2      Vektor3D force = new Vektor3D();
3      int max = 15 * (int)dist;
4      try {
5          int dis = (int)LineareAlgebra.euklDistance(vogel.pos, raubvogel.pos);
6          if (dis < max) {
7              force.add(LineareAlgebra.sub(vogel.pos, raubvogel.pos));
8              force.normalize();
9          }
10     } catch (Exception e) {
11     }
12     return force;
13 }
```

In der Methode `update()` werden alle diese Kräfte übernommen, gewichtet und auf die Geschwindigkeit und Position des Vogels angewendet.

2.2.5 Klasse `Raubvogel`

Beispielhaft für die Steuerung mit der Tastatur soll hier ein Ausschnitt aus der Methode `tastatursteuerung()` gezeigt werden. Dabei wird überprüft, ob bestimmte Tasten ({W, A, S, D} oder Pfeiltasten) gedrückt sind. Entsprechend der gedrückten Tasten werden dann geeignete Richtungsvektoren auf die Steuerungskraft addiert.

```
if (Keyboard.isKeyDown(Keyboard.KEY_LEFT) || Keyboard.isKeyDown(Keyboard.KEY_A)) {
    try {
        Vektor3D sp = raubvogel.speed;
        double a = - Math.PI / 2;
        double x = sp.x * Math.cos(a) - sp.y * Math.sin(a);
        double y = sp.x * Math.sin(a) + sp.y * Math.cos(a);
        force.add(new Vektor3D(x, y, 0));
    } catch (Exception e) {
    }
}
```

Bei jedem Update des Raubvogels wird auch überprüft, ob der Raubvogel andere Vögel in Reichweite frisst. Dabei wird für alle Vögel aus dem Schwarm die Distanz zum Raubvogel berechnet. Ist diese kleiner als 10, gelten sie als vom Raubvogel gefressen und werden in den kommenden Berechnungen nicht mehr berücksichtigt. Gleichzeitig wird pro gefressenem Vogel die Variable `mageninhalt` des Raubvogels inkrementiert.

```
public void frissDenVogel() {
    Vektor3D pos = raubvogel.pos;
    int schwarmgroesse = schwarm.getSchwarmgroesse();
    for (int i = 0; i < schwarmgroesse; i++) {
        Vogel v = schwarm.getVogel(i);
        if (v.gefressen) continue;
        try {
            double dist = LineareAlgebra.euklDistance(pos, v.pos);
            if (dist < 10) {
                v.setGefressen(true);
                i--;
                schwarmgroesse--;
                raubvogel.mageninhalt++;
            }
        } catch (Exception e) {
        }
    }
}
```

2.2.6 Klasse `OBJLoader`

Diese Klasse sorgt für das Laden und Bereitstellen eines 3D-Modells aus einer OBJ-Datei. Zur Visualisierung wird bei der Instanziierung ein Objekt der Klasse `Model` an den Raubvogel übergeben. Dieses wird in der Funktion `render()` ausgelesen und in OpenGL-Primitive überführt.

Der `OBJLoader` stellt eine statische Methode zum Laden von OBJ-Dateien zur Verfügung. Dabei liest er die Datei und parst nach den Symbolen für Vertices, Normalen-Vertices und Faces. Nachfolgend wird exemplarisch das Auslesen der Daten für einen Vertex gezeigt.

```
if (line.startsWith("v ")) {
    float x = Float.valueOf(line.split(" ")[1]);
    float y = Float.valueOf(line.split(" ")[2]);
    float z = Float.valueOf(line.split(" ")[3]);
    m.vertices.add(new Vector3f(x, y, z));
}
```

2.3 Shader

2.3.1 Klasse `LoadShader`

Die Klasse LoadShader besitzt einen privaten Konstruktor und stellt eine statische Klasse zur Verfügung, die den Shadercode aus einer Datei liest und diesen als String bereitstellt.

```
public static String load(String src) {
    File f = new File(src);
    FileInputStream fis;
    ByteArrayOutputStream bas = new ByteArrayOutputStream();
```

```
    int len; //number of read bytes
    byte[] buf = new byte[1024];
    try {
        fis = new FileInputStream(f);
        while ((len = fis.read(buf)) > -1) {
            bas.write(buf, 0, len);
        }
    } catch (Exception e) {
        System.out.println(e);
    }
    String code = new String(bas.toByteArray());
    System.out.println(code);
    return code;
}
```

2.3.2 Vertexshader

Im Vertexshader wird mithilfe der ModelViewProjectionMatrix der jeweilige Vertex an seine Zielposition verschoben.

```
void main() {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Dabei wird eine orthografische Projektion angewendet, sodass mit ganzzahligen Pixelwerten in der Größe der Fenstermaße gearbeitet werden kann.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, WIDTH, HEIGHT, 0, 0, 1);
glMatrixMode(GL_MODELVIEW);
```

2.3.3 Fragmentshader

Der Fragmentshader zeichnet alle betroffenen Pixel schwarz.

```
void main() {
    gl_FragColor = vec4(0, 0, 0, 1.0);
}
```

Literatur

[1] Block-Berlitz M.: "*Warum sich der Dino furchtbar erschreckte - Lehrbuch zu Beleuchtung und Rendering mit Java, LWJGL, OpenGL und GLSL*", vividus Wissenschaftsverlag, 2019 (to appear :))