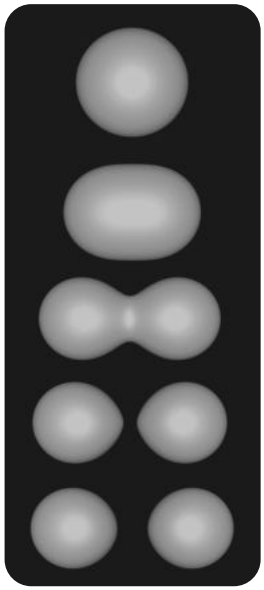
BELEGIDEE

Computergrafik/Visualisierung II

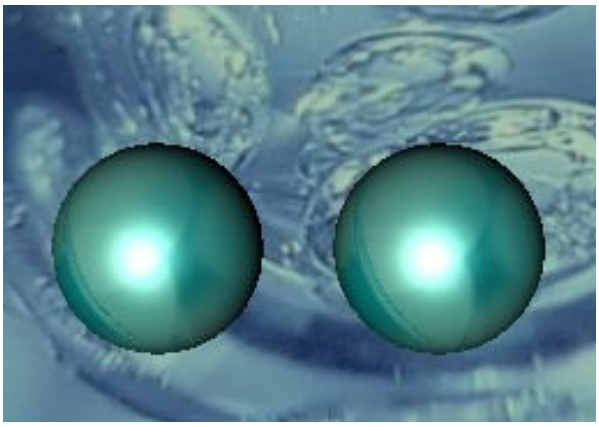
Mohammed Hijazi (s82822)
Benno Ulitzka (s82853)
Oliver Kupplmayr (s82829)

Seifenblasen Simulation Projektbeschreibung

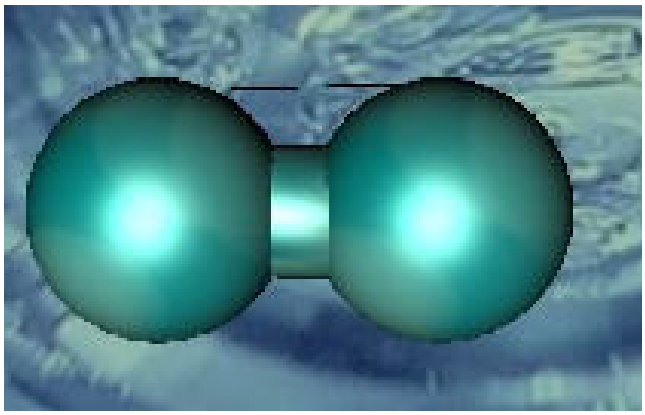


Wir wollen eine Seifenblasen Simulation umsetzen und das eventuell mit Hilfe von Metaballs.

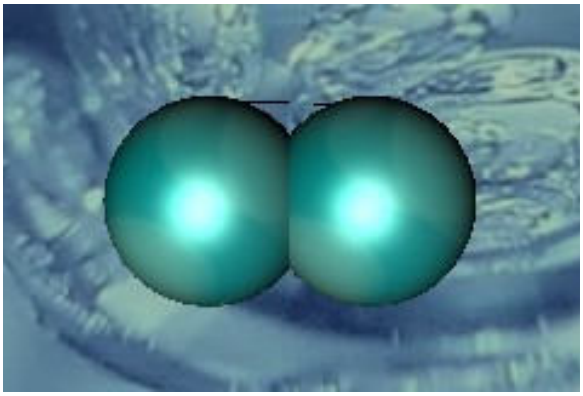
Seifenblasen begeistern wegen ihrer Zerbrechlichkeit und ihres farbenfrohen Aussehens. Die ProjektIdee ist es, dass wir uns auf die Simulation von diesen Seifenblasen konzentrieren. Diese könnte man in 4 Hauptphasen darstellen, die Nähe zueinander, aneinander stoßen, sich verbinden und schließlich zerplatzen. Beim Projekt könnte man sich ebenfalls noch auf die besonderen Farbeffekte konzentrieren.



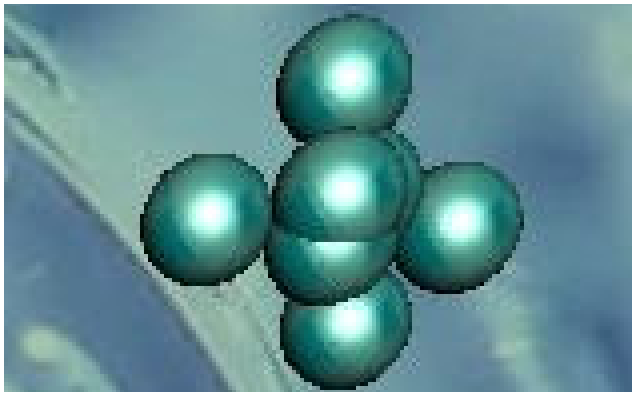
Nähe zueinander



aneinanderstoßen



verbinden

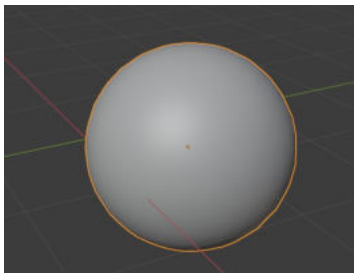
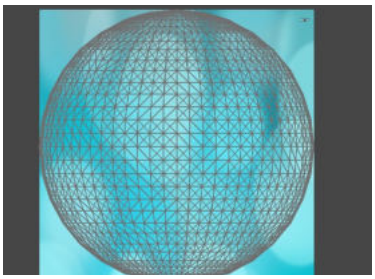


zerplatzen

Projektdokumentation

Blender:

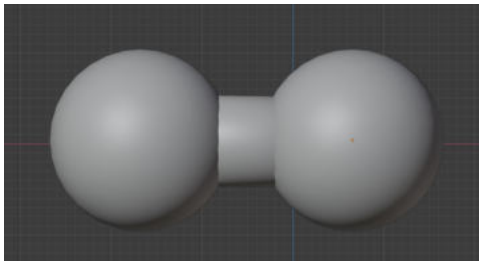
Wir haben eine Textur auf ein Objekt in Blender angewand, welches dann unsere Seifenblasen ergeben soll.


+

=


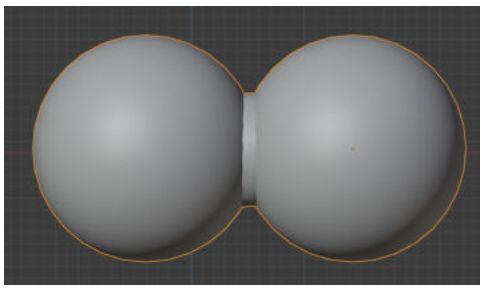
Das Modell der Seifenblase besteht im übrigen nur aus Dreiecken

Der Kollision wurde in 3 Phasen in Blender modelliert:

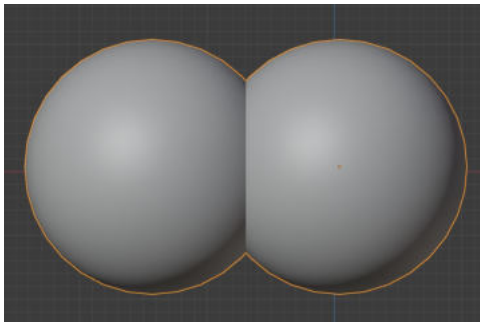
Kollision Phase 1



Kollision Phase 2



Kollision Phase 3



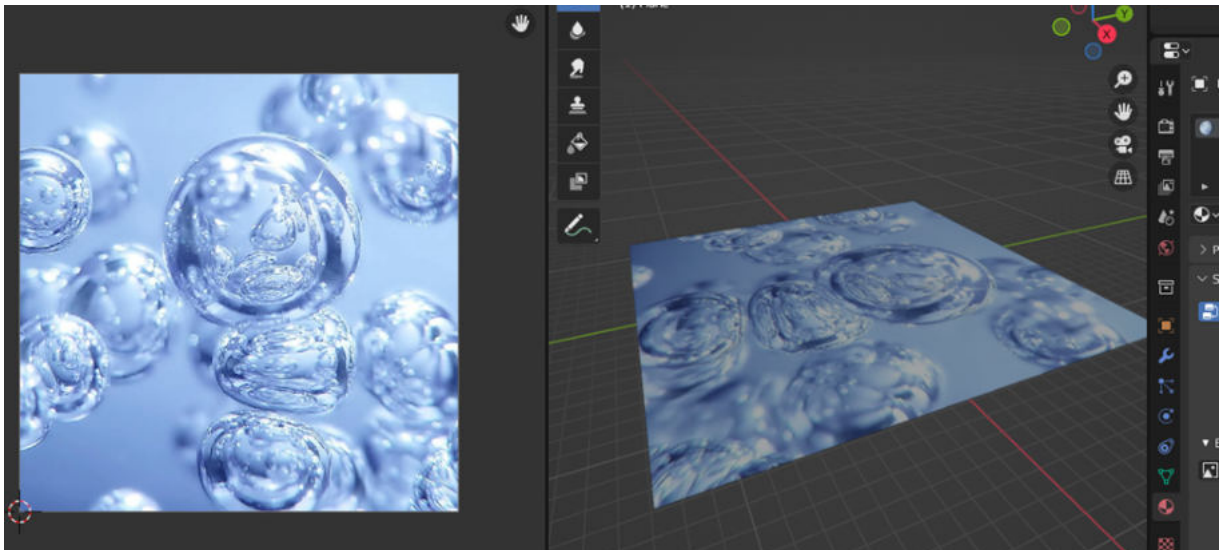


Bild wird als Hintergrund in der Simulation verwendet

Die wichtigsten Infos zum Code:

Klasse OBJLoader:

Die Class OBJLoader liest alle .obj dateien.

```
public class OBJLoader {

    public static RawModel loadObjModel(String fileName , Loader loader  ) {

        FileReader fr =null;
        try {
            fr = new FileReader(new File("res/"+ fileName + ".obj"));
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            System.out.println("could not find the file");
            e.printStackTrace();
        }

        BufferedReader reader = new BufferedReader(fr);
        String line;
        List<Vector3f> vertices = new ArrayList<Vector3f>();
        List<Vector2f> textures = new ArrayList<Vector2f>();
        List<Vector3f> normals = new ArrayList<Vector3f>();
        List<Integer> indices = new ArrayList<Integer>();
        float [] verticesArray=null;
        float [] normalsArray= null;
        float [] texturesArray = null;
        int [] indicesArray = null;
```

Die Funktion “loadObjModel()“ erstellt vier wichtige Array-Listen:

Vertices, Textures, Normal, Indices .

Die Array List: “vertices“ enthält die x-, y- und z-Koordinaten des gesamten Modells.

Die Array List: “textures“ enthält die u-, v-Koordinaten jedes Vertex des Modells.

Die Array List: “normal“ enthält die x-, y- und z-Koordinaten jeder Normalen des Modells.

Die Array List: “Indizes“ enthält Indizes der Scheitelpunkte des Modells.

Am Ende liefert die Funktion “loadObjModel“ ein Objekt der Klasse: „Raw Model“.

Klasse RawModel:

```
public class RawModel {
private int vaoID; //id of 3d model
private int vertexCount; // number of 3d object's vertex

public RawModel(int vaoID , int vertexCount) {
    this.vaoID = vaoID;
    this.vertexCount = vertexCount;
}
```

Die Klasse RawModel enthält zwei Eigenschaften:

VAOId : Jedes 3D-Modell in OpenGL wird mit einem Objekt dieser Klasse verknüpft, sodass jedes 3D-Modell eine eindeutige ID hat und in der Variablen „VAOId“ gespeichert wird.

VertexCount : Speichert die Gesamtzahl der Scheitelpunkte für ein 3D-Modell.

Wir benötigen den vertexCount-Wert, wenn wir das 3D-Modell zeichnen (rendern).

Klasse Entity:

```
public class Entity {
private TextureModel model ;
private Vector3f position;
private float roX , roY , roZ ;
private float scale;
public Entity(TextureModel model, Vector3f position, float roX, float roY, float roZ, float scale) {
    this.model = model;
    this.position = position;
    this.roX = roX;
    this.roY = roY;
    this.roZ = roZ;
    this.scale = scale;
}

public void increasePosition(float dx, float dy, float dz ) {
    this.position.x+=dx;
    this.position.y+=dy;
    this.position.z+=dz;
}

public void increaseRotation(float roX ,float roY ,float roZ) {
 this.roX += roX;
 this.roY +=roY;
 this.roZ +=roZ;
}
```

Die Klasse “**Entity**“ hilft uns, die Blase zu bewegen und sie nach Bedarf zu skalieren und zu drehen.

Als Beispiel für die Verwendung der Klasse "**Entity**" haben wir in der Datei "**MainLoop.java**" 9 Objekte der Klasse "Entity" deklariert, jede Blase ist 0,25 kleiner als die ursprüngliche Blase :

```
Entity explosion1 = new Entity(myobj_textureModel, new Vector3f(0f,0f ,-10f), 0f, 0f, 0f, 0.25f);
Entity explosion2 = new Entity(myobj_textureModel, new Vector3f(0.5f,0f ,-10f), 0f, 0f, 0f, 0.25f);
Entity explosion3 = new Entity(myobj_textureModel, new Vector3f(-0.5f,0f ,-10f), 0f, 0f, 0f, 0.25f);
Entity explosion4 = new Entity(myobj_textureModel, new Vector3f(0f,0.5f ,-10f), 0f, 0f, 0f,  0.25f);
Entity explosion5 = new Entity(myobj_textureModel, new Vector3f(0f,-0.5f ,-10f), 0f, 0f, 0f,  0.25f);
Entity explosion6 = new Entity(myobj_textureModel, new Vector3f(0f,0f ,-10f), 0f, 0f, 0f,  0.25f);
Entity explosion7 = new Entity(myobj_textureModel, new Vector3f(0f,0f ,-10f), 0f, 0f, 0f,  0.25f);
Entity explosion8 = new Entity(myobj_textureModel, new Vector3f(0f,0f ,-10f), 0f, 0f, 0f,  0.25f);
Entity explosion9 = new Entity(myobj_textureModel, new Vector3f(0f,0f ,-10f), 0f, 0f, 0f,  0.25f);
```

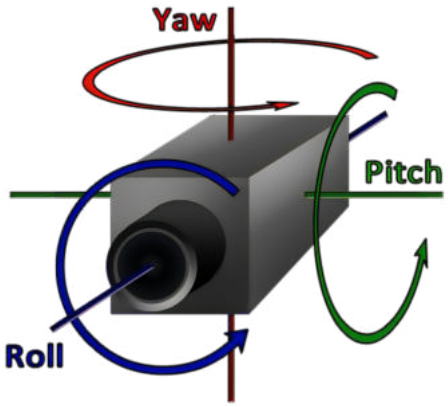
Diese Objekte (Explosion 1 … 9) sind das Ergebnis einer Kollision von zwei Seifenblasen

Klasse Camera:

```
public class Camera {
 private Vector3f position= new Vector3f(0,0,0);
 private float pitch; //how much high or low
 private float yaw;  // how much left or right
 private float roll;

public Camera(){}

public void move() {
    if (Keyboard.isKeyDown(Keyboard.KEY_W)) {
        position.z-=0.02f;
    }
    if (Keyboard.isKeyDown(Keyboard.KEY_S)) {
        position.z+=0.02f;
    }
    if (Keyboard.isKeyDown(Keyboard.KEY_A)) {
        position.x-=0.02f;
    }
    if (Keyboard.isKeyDown(Keyboard.KEY_D)) {
        position.x+=0.02f;
    }
}
```



In der Klasse „**Camera**" haben wir eine Reihe von Haupteigenschaften der realen Kamera definiert, Eigenschaften wie: Position der Kamera „Vektor3f Position" und „pitch" und „yaw" und „roll".

Wenn wir ein Objekt der Klasse "**Camera**" erstellen, besteht keine Notwendigkeit diese Variablenwerte zu ändern.

Außerdem hat die Klasse eine Gruppe von Setter- und Getter-Funktionen, wie z.B.:

```
Vector3f getPosition(), void setPitch(float pitch)..
```

In der Hauptdatei des Projekts: **MainLoop.java**, haben wir ein Objekt der Klasse Camera erstellt und die Kamera an der Position: x=0, y=0, z=6 platziert.

```
Camera camera = new Camera();
Vector3f camera_position =camera.getPosition();
camera_position.z +=6;
camera.setPosition(camera_position);
```

Klasse Light:

Die Klasse "**Light**" hat zwei reale Lichteigenschaften:

- Position des Lichts "vector3f position"
- Farbe des Lichts "vector3f color".

```
public class Light {

    private Vector3f position;
    private Vector3f colour ;
    public Light(Vector3f position, Vector3f colour) {
        this.position = position;
        this.colour = colour;
    }
    public Vector3f getPosition() {
        return position;
    }
    public void setPosition(Vector3f position) {
        this.position = position;
    }
    public Vector3f getColor() {
        return colour;
    }
    public void setColor(Vector3f color) {
        this.colour = color;
    }

}
```

In der Hauptdatei des Projekts: **MainLoop.java**, haben wir zwei Objekte der Klasse Light erstellt:

```
Light light1 = new Light(new Vector3f(400f , 400f, 300.0f ) , new Vector3f(0.7f , 0.9f ,0.07f))
Light light2 = new Light(new Vector3f(-50f , -50f, 300.0f ) , new Vector3f(0.7f , 0.8f ,0.7f));
```

Klasse DisplayManager:

```
public class DisplayManager {

    public static final int WIDTH =1024;
    public static final int HEIGH =800;
    public static final int FPS_CAP = 320;

    public static void createDisplay() {

        ContextAttribs attribs = new ContextAttribs(3,2)
        //.attribs.withForwardCompatible(true)
        //.attribs.withProfileCore(true);
        .withForwardCompatible(true)
        .withProfileCore(true);

        //width , high , bitDepth , refresh rate
        try {
            Display.setDisplayMode(new DisplayMode(WIDTH, HEIGH));
            Display.create(new PixelFormat() , attribs);
            Display.setTitle("bubbles simulation");

        } catch (LWJGLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
```

Die Klasse "**DisplayManager**" enthält wichtige Eigenschaften: **FPS_CAP, WIDTH, HEIGH**

- **FPS_CAP**: Der Wert der Variable bestimmt, wie viele Bilder pro Sekunde angezeigt werden.
- **WIDTH**: Der Wert der Variable bestimmt die Breite des Simulationsfensters.
- **HEIGH**: Der Wert der Variable bestimmt die Höhe des Simulationsfensters.

Klasse Renderer:

Die Klasse "Renderer" hat 3 wichtige Eigenschaften:

fov (field of view), NEAR_PLANE, FAR_PLANE

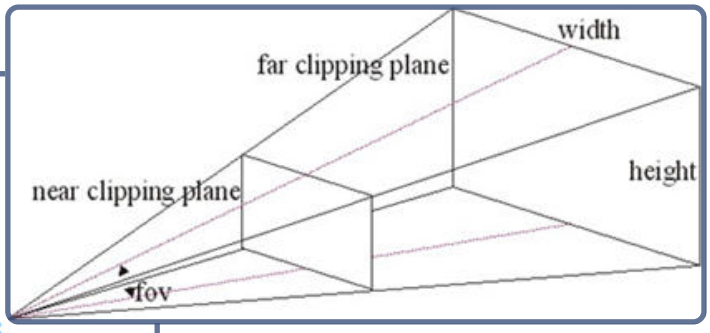
```
package renderEngine;

import org.lwjgl.opengl.GL11;

public class Renderer {

    private static final float FOV=70;
    private static final float NEAR_PLANE = 1f;
    private static final float FAR_PLANE = 1000;
    private Matrix4f projectionMatrix;

    public Renderer(StaticShader shader){
        createProjectionMatrix();
        shader.start();
        shader.loadProjectionMatrix(projectionMatrix);
        shader.stop();
    }
```



Nur 3D Modelle, die sich innerhalb der Pyramide befinden werden schlußendlich auf dem Bildschirm sichtbar sein.

Klasse render:

```
public void render(Entity entity , StaticShader shader) {
    TexturedModel model =entity.getModel();
    RawModel rawModel = model.getRawModel();
    GL30.glBindVertexArray(rawModel.getVaoID()); //bind vao
    GL20.glEnableVertexAttribArray(0);   //enable vbo
    GL20.glEnableVertexAttribArray(1);
    GL20.glEnableVertexAttribArray(2);

    Matrix4f transformationMatrix = Maths.createTransformationMatrix(entity.getPosition(), entity.getRoX(), entity.getRo
    shader.loadTransformationMatrix(transformationMatrix);

    //shinees and reflectivity
    ModelTexture texture= model.getModelTexture();
    shader.loadShineVariable(texture.getShineDamper() , texture.getReflectivity());

    GL13.glActiveTexture(GL13.GL_TEXTURE0);
    GL11.glBindTexture(GL11.GL_TEXTURE_2D, model.getModelTexture().getID());
    //GL11.GL_POINTS GL11.GL_LINES GL11. GL_TRIANGLES GL_TRIANGLE_FAN
    GL11.glDrawElements(GL11.GL_TRIANGLE_STRIP, rawModel.getVertexCount(), GL11.GL_UNSIGNED_INT, 0); //draw from first v

    GL20.glDisableVertexAttribArray(2);
    GL20.glDisableVertexAttribArray(1);
    GL20.glDisableVertexAttribArray(0); //disable vbo
    GL30.glBindVertexArray(0); //unbind vao

}
```

Die Klasse "**render**" hat auch eine wichtige Funktion:
void render (Entity entity, Shader shader), diese Funktion rendert (zeigt) das 3D-Modell mit entsprechender Textur auf dem Bildschirm.

Jedes 3D-Modell hat 3 Arrays (vertices 0, textures 1, normal 2), in OpenGL sollten die 3 Arrays vereint werden, bevor das 3D Model gezeichnet wird.

Nach dem Zeichnen des 3D-Modells, sollten wir die Bindung aufheben oder die 3 vorherigen Arrays deaktivieren.

Klasse ShaderProgram:

```
public abstract class ShaderProgram {
 private int programID;
 private int vertexShaderID;
 private int fragmentShaderID;

 private static FloatBuffer matrixBuffer= BufferUtils.createFloatBuffer(16);

public ShaderProgram(String vertexFile , String fragmentFile) {
    vertexShaderID = loadShader(vertexFile, GL20.GL_VERTEX_SHADER);
    fragmentShaderID = loadShader(fragmentFile, GL20.GL_FRAGMENT_SHADER);
    programID = GL20.glCreateProgram();
    GL20.glAttachShader(programID, vertexShaderID);
    GL20.glAttachShader(programID, fragmentShaderID);
    bindAttributes();
    GL20.glLinkProgram(programID);
    GL20.glValidateProgram(programID);
    getAllUniformLocations();
}
```

Der Konstruktor der Klasse "**ShaderProgram**" wird das Vertex- und Fragment-Shader-Programm laden und mit unserem Projekt verknüpfen.

```
protected void bindAttribute(int attribute , String variableName) {
    GL20.glBindAttribLocation(programID, attribute, variableName);
}

protected void loadFloat(int location , float value) {
    GL20.glUniform1f(location, value);
}

protected void loadVector(int location , Vector3f vector) {
    GL20.glUniform3f(location, vector.x , vector.y, vector.z);
}

protected void loadBoolean(int location , boolean value) {
    float toLoad =0;
    if(value) {
        toLoad =1 ;
    }
    GL20.glUniform1f(location, toLoad);
}
protected void loadMatrix(int location , Matrix4f matrix) {
    matrix.store(matrixBuffer);
    matrixBuffer.flip();
    GL20.glUniformMatrix4(location, false, matrixBuffer);
}
```

Außerdem wird die Klasse "**ShaderProgram**" eine einheitliche Variable wie float, Boolean, matrix usw. erzeugen.

Klasse StaticShader:

```
public class StaticShader extends ShaderProgram {

    public static final String VERTEX_FILE="src/shaders/vertexShader.txt";
    public static final String FRAGMENT_FILE = "src/shaders/fragmentShader.txt";

    private int location_transformationMatrix;
    private int location_projectionMatrix;
    private int location_viewMatrix;
    private int location_lightColour;
    private int location_lightPosition;
    private int location_shineDamper;
    private int location_reflectivity;
    public StaticShader() {
        super(VERTEX_FILE, FRAGMENT_FILE);

    }
```

Die Klasse "**StaticShader**" erbt die Klasse "ShaderProgram", diese Klasse hat viele Integer-Variablen, um die eindeutige ID von Matrizen und Vektoren zu speichern.

```
@Override
protected void getAllUniformLocations() {
    location_transformationMatrix=super.getUniformLocation("transformationMatrix");
    location_projectionMatrix=super.getUniformLocation("projectionMatrix");
    location_viewMatrix=super.getUniformLocation("viewMatrix");
    location_lightPosition =super.getUniformLocation("lightPosition");
    location_lightColour = super.getUniformLocation("lightColour");
    location_shineDamper=super.getUniformLocation("shineDamper");
    location_reflectivity = super.getUniformLocation("reflectivity");
}
```

Mit der Funktion "getAllUniformLocations()" erhalten wir die eindeutige ID der Matrizen und Vektoren.

```
public void loadShineVariable(float dumper , float reflectivity) {
    super.loadFloat(location_shineDamper, dumper);
    super.loadFloat(location_reflectivity, reflectivity);
}
public void loadTransformationMatrix(Matrix4f matrix) {
    super.loadMatrix(location_transformationMatrix, matrix);
}

public void loadLight(Light light) {
    super.loadVector(location_lightPosition, light.getPosition());
    super.loadVector(location_lightColour, light.getColor());
}
public void loadViewMatrix(Camera camera) {
    Matrix4f viewMatrix = Maths.createViewMatrix(camera);
    super.loadMatrix(location_viewMatrix, viewMatrix);
}
public void loadProjectionMatrix(Matrix4f matrix) {
    super.loadMatrix(location_projectionMatrix, matrix);
}
```

Die vorherigen Funktionen (loadShineVariable(), loadLight()) ermöglichen es uns, neue Werte in den einheitlichen Variablen zu speichern.

vertexShader.txt

```
#version 400 core

in vec3 position;
in vec2 textureCoords;
in vec3 normal;

out vec2 pass_textureCoords;
out vec3 surfaceNormal;
out vec3 toLightVector;
out vec3 toCameraVector;

uniform mat4 transformationMatrix;
uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform vec3 lightPosition;

void main (void){
  vec4 worldPosition = transformationMatrix * vec4( position , 1.0);
  gl_Position = projectionMatrix * viewMatrix  * worldPosition;
  pass_textureCoords = textureCoords;
  surfaceNormal =(transformationMatrix * vec4( normal , 0.0)).xyz;
  toLightVector = lightPosition - worldPosition.xyz;
  toCameraVector= (inverse(viewMatrix) * vec4(0.0,0.0,0.0,1.0)).xyz -  worldPosition.xyz;
}
```

Das Programm "vertexShader" benötigt drei Eingaben:

Position des Vertex (x, y, z),

Texturkoordinaten des Vertex (u, v),

Normalkoordinaten des Vertex (x, y, z).

Die Ausgabe von "vertexShader" sind:

Texturkoordinaten des Vertex (u, v),

die Normalkoordinaten (x, y, z),

und Vektorpunkte zur Kamera vom Vertex (x, y, z)

und Vektorpunkte zum Licht vom Vertex (x, y, z).