Hochschule Darmstadt
B. Frömmer, B. Meyer

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

20. November 2023

Visual Computing SoSe 2023
Aufgabe 3

3.1 Perspektivische Kamera

In der ersten Hälfte dieses Aufgabenblattes wollen wir eine Kamera in unser kleines Beispielprogramm integrieren, um eine korrekte perspektivische Projektion in unseren Bildern zu erhalten.

3.1.1 Die ViewMatrix

Bestimmen Sie, wie aus der Vorlesung bekannt, zuerst die `ViewMatrix`, mit derer die komplette Szene so transformiert wird, dass Ihre (virtuelle) Kamera im Ursprung liegt und entlang der negativen Z-Achse schaut. Die Mathe-Bibliothek `glm` stellt bereits eine entsprechende Methode bereit: `glm::lookAt()`. Sie müssen nur noch die Position der Kamera, einen Betrachtungspunkt sowie einen Up-Vektor definieren.

Achtung: Wenn Sie testweise das Bild nur mit der Viewmatrix rendern, achten Sie bei der Positionierung der Kamera darauf, nicht den *Normalized Device Space* zwischen -1 und 1 zu verlassen.

3.1.2 Die ProjectionMatrix

Im zweiten Schritt müssen Sie die perspektivische Projektion der Kamera bestimmen. Diese ist abhängig von den internen Kameraparametern. Die Methode `glm::perspective()` übernimmt die Berechnung der entsprechenden Matrix.

Damit die Kamera beim Rendervorgang berücksichtigt werden kann, müssen Sie die beiden neuen Matrizen noch in den Shader laden und dort korrekt verwenden.

Achtung: Wir hatten im letzten Aufgabenblatt den Tiefentest angeschaltet, mussten hier jedoch noch "falsch herum" testen, da die Invertierung der Z-Achse fehlte (siehe Vorlesung). Das ist nun nicht mehr notwendig, ändern Sie daher Ihren Tiefentest auf folgende Zeilen:

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS); // Default-Value, könnte auch weggelassen werden
glClearDepth(1.0); // Default-Value, könnte auch weggelassen werden
```

Spielen Sie mit den verschiedenen Parametern der Kamera - was bewirken Änderungen in den Near- und Far-Clipping-Planes, dem (vertikalen) Öffnungswinkel oder dem Seitenverhältnis?

3.2 Freie Anwendung

Testen Sie Ihr gewonnenes Wissen! Bauen Sie eine eigene, kreative Anwendung.

- Verwenden Sie andere Geometrien: Statt des gegebenen Cubes können Sie beliebige andere Objekte einladen. Diese lassen sich entweder selber definieren, im Internet finden sich allerdings auch schon viele vordefinierte geometrische Figuren. Es spricht auch nichts dagegen, *mehrere unterschiedliche* Objekte anzuzeigen.
- Entwerfen Sie kreative Animationen: Implementieren Sie Bewegungsmuster für Ihren Roboter oder andere Figuren, transformieren Sie Ihren Roboter beispielsweise in ein Auto/Flugzeug. Hier können Sie eigene Ideen entwickeln.
- Verändern Sie Ihre Szene per Tastatureingaben - steuern Sie den Roboter oder die Kamera. Gerne können Sie auch Mouse-Inputs verwenden.
- In der update()-Methode der Scene.cpp können Sie über

  `if (m_window->getInput().getKeyState(Key::W) == KeyState::Pressed)`

  auf Tastatureingaben zugreifen. Entwerfen Sie eine eigene Steuerung für Ihre Objekte!
- Ähnlich können Sie in `onMouseMove()` auf Mausbewegungen reagieren (um damit beispielsweise die Kamera zu steuern).

  Tipp: Das `MousePosition`-Objekt speichert sich die neue und die alte Mausposition:

  ```
  auto newPosition = glm::vec2(mouseposition.X, mouseposition.Y);
  auto oldPosition = glm::vec2(mouseposition.oldX, mouseposition.oldY);
  auto direction = (newPosition - oldPosition);
  ```

  kann helfen, um die Bewegungsrichtung der Maus zu ermitteln.