Praktikum 6: 3D & Depth Buffer

Kopieren Sie passenden Code-Dateien aus den letzten Praktika in den des aktuellen Praktikums.

Aufgabe 1 3D Punkte Wolke

a) Implementieren Sie die Funktion

```
/***
 * Sets this matrix to a perspective projection transform.
 * The viewer gazes into negative z direction.
 * @param n near plane position (Use negative values!)
 * @param f far plane position (Use negative values!)
 * @param fovyRadians field-of-view in y direction in radians
 *                    (i.e., 0..2Pi and NOT 0..360 degrees)
 * @param aspect ratio of width and height of your window.
 */
public
void setPerspectiveProjectionTransform(float n, float f,
                                       float fovyRadians,
                                       float aspect);
```

in `mat4`.

b) Zeichen Sie die Punkte des Modells bunny.smm.

- Transformieren Sie dazu die Punkte vom Object-Space in den Clip-Space mittels der Matrix aus (a). Nutzen Sie als Near-Plane -0.01 und als Far-Plane -5. Für die Multiplikation bietet sich die Funktion `mul\_W1` an.
- Die Punkte im Clip-Space müssen dann in den Window-Space transformiert werden. Nehmen Sie dazu die Matrix aus dem letzten Praktikum. Setzen Sie dabei aber dazu

  ```
  this.a11 = -(float)h / 2.0f;
  ```

  (Minus Zeichen beachten) in der `setWindowTransform`, damit nicht alles kopfüber steht.
- Anschließend führen Sie den “Perspective z-Divide” durch. Implementieren Sie dazu die Methode

  ```
  public float3 homogenize()
  ```

  in `float4`.

- Abschließend zeichnen Sie die Punkte am Bildschirm. Sie müssen für jeden Punkt testen, ob er auch wirklich im Bild ist, sonst kann es zu Speicherfehlern kommen!

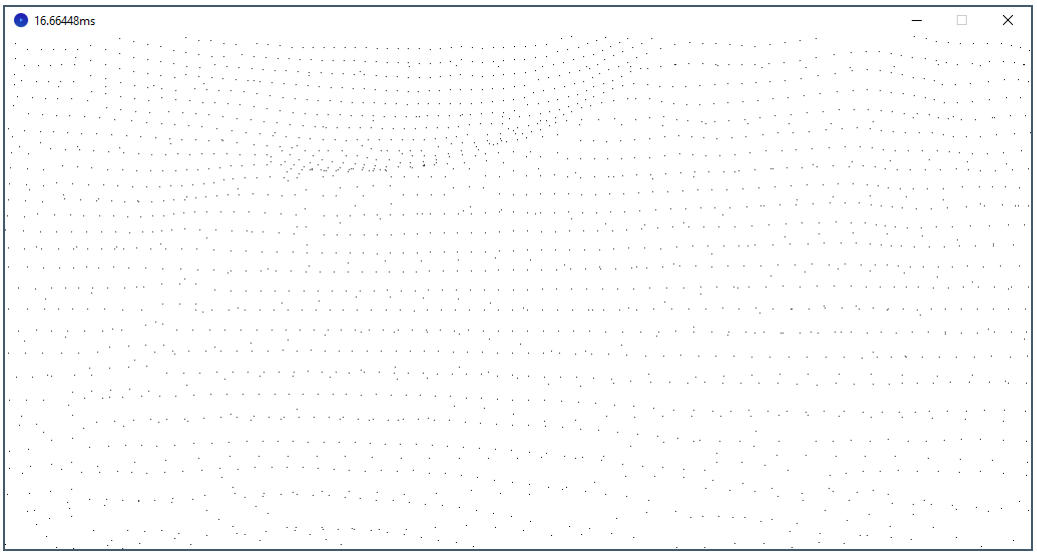

Abbildung 1: Der Hase als Punktewolke

c) Die Kamera befindet sich im Moment noch im Inneren des Modells. Verschieben Sie also das Modell-Position im Object-Space um -2 Einheiten in z-Richtung geeignet. Nutzen Sie dazu Matrix-Vektor Multiplikation. Es bietet sich an, die Methode

```
/**
 * Computes and returns the matrix-vector multiplication of
 * this * [rhs.x, rhs.y, rhs.z, 1]\^T, i.e., assuming that
 * - the 4th component of rhs is 1.
 * - setting the last row of this matrix to [0, 0, 0, 0]
 *
 * @param rhs the x, y, z component of as right-hand side
 *            of the product.
 * @return The x, y, z component matrix-vector product
 *         this * [rhs.x, rhs.y, rhs.z, 1]\^T.
 */
public float3 mul\_Affine\_W1(float3 rhs)
```

in `mat4` zu implementieren und im Hauptprogramm zu verwenden.

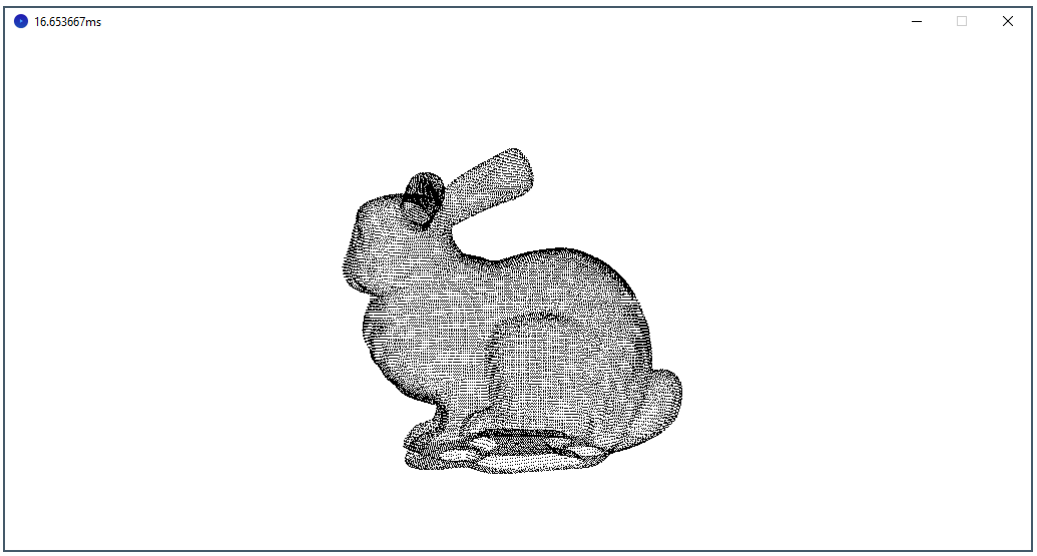

Abbildung 2: Der Hase als Punktewolke

d) Rotieren Sie das Objekt *vor* dem Verschieben erst um die x- dann um die y- und schließlich um die z-Achse und zwar jeweils um den Winkel `Math.toRadians(millis()* 0.1f)`. Implementieren Sie dazu die Methoden der Klasse `mat4 setRotateX` und `setRotateY` analog zu `setRotateZ`. Vereinheitlichen Sie alle Matrizen zu einer Model-View-Projection-Window Matrix.

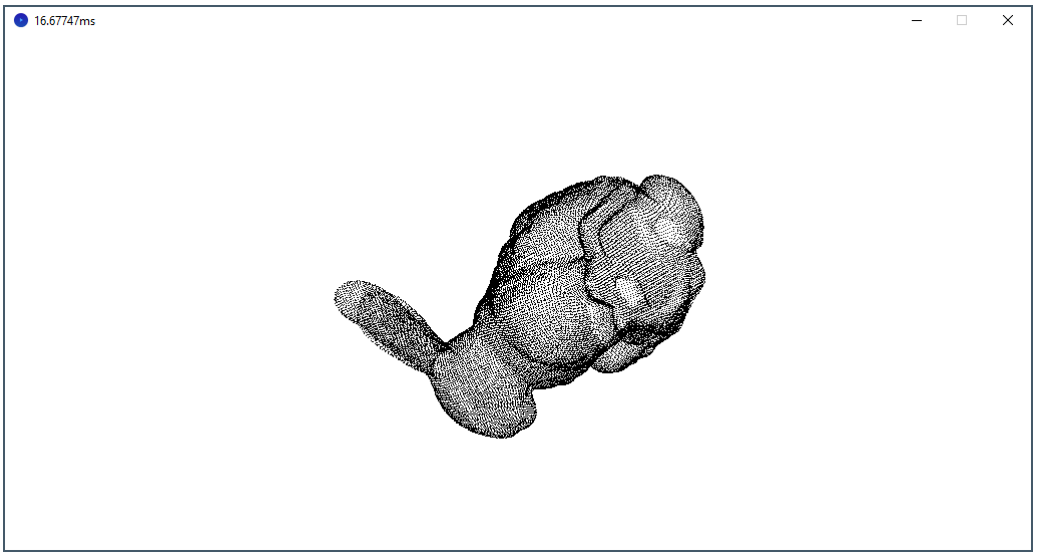

Abbildung 3: Jetzt dreht sich der Hase auch noch!

Aufgabe 2 3D Dreiecke

a) Es ist nun hilfreich, folgende Funktion zu implementieren:

```
1 //! Convert from a regular (coarse) pixel to the CENTER a subpixel
2 public static int toSubPixel(float regularPixel);
3
4 //! Convert from a regular (coarse) pixel to the CENTER a subpixel
5 //! For 2D pixels. Provided for convenience.
6 public static int2 toSubPixel(float3 regularPixel);
```

Achtung: Diese verwenden nun float-Werte anstelle von Integer-Werte!

b) Ändern Sie die Signatur Ihrer `TriangleRasterizer.drawTriangle` Methode in folgende Signatur:

```
1 public static final
2 void drawTriangle(int[] framebuffer, int w, int h,
3                   int2 pointA, int2 pointB, int2 pointC,
4                   float zA, float zB, float zC);
```

und Interpolieren Sie für jeden Pixel den z-Wert. Setzen Sie zum Debuggen die Farbe des Pixels auf den z-Wert.

c) Zeichnen Sie nun das gesamte Dreiecksnetz! Übergeben Sie den `drawTriangle` Aufrufen nun die Window-Space z-Werte der Position. Nutzen Sie die in Teilaufgabe (a) implementierten Methoden, um die Werte für Argumente `pointA`, `pointB` und `pointC` aus Window-Space xy-Werten zu berechnen.

Um sinnvolle Debug-Farben zu bekommen, setzen Sie die Near-Plane auf -1.5 und die Far-Plane auf-4.5.

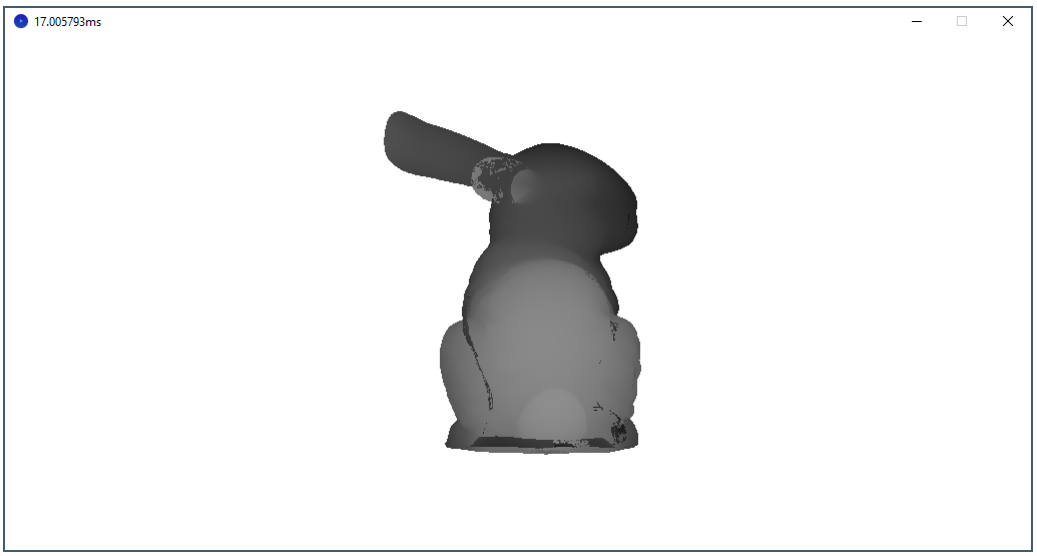

Abbildung 4: Mit jedem Pixel seiner Haut kodiert der Hase nun den Tiefenwert als Graufstufe. Allerding sind die Fragment noch nicht richtig sortiert.

Aufgabe 3 Depth Buffer

a) Vervollständige Sie den Konstruktor in der Klasse `DepthBuffer`. Legen Sie für jeden Pixel einen z-Wert an!

b) Implementieren Sie die `clear` Methode. Setzen Sie dazu den Wert jedes Pixels auf den Wert der Far-Plane *nach* Anwendung aller Transformation (inkl. Homogenisierung). *Hinweis:* Falls Sie nicht draufkommen: Es ist 1.0f. Aber herleiten sollten Sie es schon irgendwann mal können!

c) Ändern Sie die Signatur Ihrer `drawTriangle` Methode in folgende Signatur:

```
1 public static final
2 void drawTriangle(int[] framebuffer, float[] depth, int w, int h,
3                    int2 pointA, int2 pointB, int2 pointC,
4                    float zA, float zB, float zC)
```

implementieren Sie den Depth-Test. Stellen Sie sicher, dass in Ihrem Hauptprogramm der Depth-Buffer vor jedem zeichnen ge-clear-t wird.

Achtung: Ihr Dreieckstest sollte Dreiecke im und gegen den Uhrzeigersinn funktioniert.

Abbildung 5: Der Hase mit korrekt sortierten Fragmenten