4. Praktikum in Computergrafik und Bildverarbeitung

Lisa Obermaier, Simon Thum

Frist 4. Juni 2019, 23:59

Animieren Sie Ihre Szene aus Aufgabe 3.2b. Benutzen Sie dazu u.a. alle drei Modell-Transformationen (*Translate(), *Rotate(), *Scale())

Die Animation wird über eine Timerfunktion realisiert, die via *glutTimerFunc(1000.0 / 60.0, timer, 0);* registriert und dadurch jede Sechzigstelsekunde aufgerufen wird.

Die Timerfunktion enthält den Inhalt von RenderScene aus Aufgabe 3.2b, die zeitabhängige Änderung von Parametern erfolgt in den jeweiligen *Draw*-Methoden der Objekte.

Rauch: Translate und Rotate

Der Rauch dreht sich durch einen Rotationsparameter konstant um die eigene Y-Achse, weiterhin wird die Position auf der X-Achse durch einen Translationsparameter vor und zurückbewegt.

```
void DrawSmoke() {
    static float translateSmoke = 0.0f;
    static int rotateSmoke = 0.0f;
    rotateSmoke = (rotateSmoke + 1) % 360;
    static bool directionChange = false;
    if (!directionChange) {
        translateSmoke += 0.5f;
        if (translateSmoke > 100.0)
            directionChange = true;
    }
    else {
        translateSmoke -= 0.5f;
        if (translateSmoke < 0.0)
            directionChange = false;
    }

    modelViewMatrix.PushMatrix();
    modelViewMatrix.Translate(100 + translateSmoke, 200, -230);
    modelViewMatrix.Rotate(rotateSmoke, 0, rotateSmoke, 0);
    shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
    DrawSphere();

    modelViewMatrix.PushMatrix();
    modelViewMatrix.Scale(1.2, 1.2, 1.2);
    modelViewMatrix.Translate(80 + translateSmoke, 50, -100);
    modelViewMatrix.Rotate(rotateSmoke, 0, rotateSmoke, 0);
    shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
    DrawSphere();

    modelViewMatrix.PushMatrix();
    modelViewMatrix.Scale(1.2, 1.2, 1.2);
    modelViewMatrix.Translate(80 + translateSmoke, 80, -100);
    modelViewMatrix.Rotate(rotateSmoke, 0, rotateSmoke, 0);
    shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
    DrawSphere();
    modelViewMatrix.PopMatrix();
    modelViewMatrix.PopMatrix();
    modelViewMatrix.PopMatrix();
```

Schornstein: Scale

Der Schornstein wird durch Rechnung desselben Skalierungsparameters mit unterschiedlichen Vorzeichen abwechselnd lang und dünn und kurz und dick.

```
void DrawChimney() {
    static float translateCh = 0.0f;
    static bool directionChange = false;
    if (!directionChange) {
        translateCh += 0.00025f;
        if (translateCh > 0.1)
            directionChange = true;
    }
    else {
        translateCh -= 0.00025f;
        if (translateCh < 0.01)
            directionChange = false;
    }

    modelViewMatrix.Translate(30, 120, 50);
    modelViewMatrix.Scale(0.4 - translateCh, 0.3 + translateCh, 0.4 - translateCh);
    modelViewMatrix.PushMatrix();
    modelViewMatrix.Rotate(90, 90, 0, 0);
    shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
    DrawCylinder();
    modelViewMatrix.PopMatrix();
}
```

Räder: Rotate

Die Räder drehen sich durch einen Rotationsparameter konstant um die Z-Achse.

```
static float rotWh = 0.0f;
rotWh += 01.0f;

//erstes Set Raeder
modelViewMatrix.Scale(0.5, 0.5, 0.1);
modelViewMatrix.Translate(40, -50, 150);
modelViewMatrix.PushMatrix();
modelViewMatrix.Rotate(rotWh, 0, 0, 150);
shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
DrawCylinder();
modelViewMatrix.PopMatrix();
```

Kuppelstange: Translate in Abhängigkeit von Rotate

Die vier Kuppelstangen zwischen den jeweiligen Rädern werden durch einen Translationsparameter kreisförmig auf der X- und Y-Achse bewegt. Da die Stangen bei einer echten Lokomotive an je einem Radpunkt fixiert sind, musste auch bei der Animation Sorge getragen werden, dass die Translation in Abhängigkeit des aktuellen Rotationsstandes der Räder geschieht. Dies wird in X-Richtung durch den Cosinus des Rotationswinkels realisiert, in Y-Richtung entsprechend durch den Sinus.

```
modelViewMatrix.PushMatrix();
modelViewMatrix.Scale(1.75, 0.1, 0.01);
modelViewMatrix.PushMatrix();
modelViewMatrix.Translate(-137 + 10 * cos(rotWh*0.0174), 45 + 130 * sin(rotWh*0.0174), 100);
modelViewMatrix.Rotate(180, 180, 0, 0);
shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
DrawCylinder();
modelViewMatrix.PopMatrix();
modelViewMatrix.PopMatrix();
```

Bewegen Sie Ihren Augenpunkt durch Ihre Szene aus den vorigen Praktikumsaufgaben im sog. UFO-Modus

Der **Augenpunkt** wird durch die Pfeiltasten der Tastatur bewegt, der Abfang der Tastatureingaben erfolgt gemäß des in der Aufgabenstellung verlinkten Beispiels.
Bei Druck der Links- und Rechtstasten werden in der X-Komponente einer Augenpunkts-Matrix Werte addiert oder subtrahiert, für Bewegungen nach oben und unten entsprechend in der Y-Komponente.

```
void SpecialKeys(int key, int x, int y)
{
    switch (key)
    {
    case GLUT_KEY_UP:
        m3dLoadVector3(augenpunkt[0], augenpunkt[0][0], augenpunkt[0][1] + 10, 0);
        break;
    case GLUT_KEY_RIGHT:
        m3dLoadVector3(augenpunkt[0], augenpunkt[0][0] + 10, augenpunkt[0][1], 0);
        break;
    case GLUT_KEY_DOWN:
        m3dLoadVector3(augenpunkt[0], augenpunkt[0][0], augenpunkt[0][1] - 10, 0);
        break;
    case GLUT_KEY_LEFT:
        m3dLoadVector3(augenpunkt[0], augenpunkt[0][0] - 10, augenpunkt[0][1], 0);
        break;
    }
    TwEventKeyboardGLUT(key, x, y);
    // Zeichne das Window neu
    glutPostRedisplay();
}
```

Die Augenpunktsmatrix wird zu Beginn der *timer*-Funktion als Translationsmatrix benutzt. Dies bewegt die “Kamera” aber nicht die Objekte, da diese noch nicht auf dem Matrizenstapel liegen.

```
modelViewMatrix.Translate(augenpunkt[0][0], augenpunkt[0][1], augenpunkt[0][2]);
```

Nick- und Gierbewegungen sind mit der linken und rechten Maustaste belegt. Nach Druck wird ein globaler Boolean auf true gesetzt, der innerhalb der Motion-Methode in Abhängigkeit von aktueller und letzter Mauszeigerposition eine Drehung nach links oder rechts resp. oben und unten definiert.

```
if (state == GLUT_DOWN) {
    if (button == 0) { //Gierbewegung
        //printf("Gierbewegung\n");
        start = x;
        rotateGier = true;
    }
    else if (button == 1) { //middle mouse button

    }
    else if (button == 2) { //Nickbewegung
        //printf("Nickbewegung");
        start = y;
        rotateNick = true;
    }
```

Dies geschieht durch Addition oder Subtraktion eines Wertes auf eine Matrix mit den aktuellen Rotationswerten in X- und Y-Richtung.

```
void
Motion(int x, /* I - Current mouse X position */
    int y) /* I - Current mouse Y position */
{
    if (rotateGier) {
        if (x > start) { //mehr links sehen L->R ziehen
            rotateX[0][0] = rotateX[0][0] + 5;
            rotateX[0][2] = 1;
            start = x;
        }
        else {      //mehr rechts sehen R->L ziehen
            rotateX[0][0] = rotateX[0][0] - 5;
            rotateX[0][2] = 1;
            start = x;
        }

    }
    if (rotateNick) {
        if (y > start) {
            rotateY[0][0] = rotateY[0][0] + 5;
            rotateY[0][1] = 1;
            start = y;
        }
        else {
            rotateY[0][0] = rotateY[0][0] - 5;
            rotateY[0][1] = 1;
            start = y;
        }
    }
}
```

Mit dieser Matrix wird die “Kamera” rotiert, da die Rotation in der t*imer*-Funktion aufgerufen wird, bevor Objekte auf den Matrizenstapel gelegt werden.

```
modelViewMatrix.Rotate(rotateX[0][0], rotateX[0][1], rotateX[0][2], rotateX[0][3]);
modelViewMatrix.Rotate(rotateY[0][0], rotateY[0][1], rotateY[0][2], rotateY[0][3]);
```

Eine **Bewegung des Augenpunktes nach vorne und hinten** ist durch eine Z-Translation realisiert. Bei Drehen des Mausrads nach oben (entspricht *zoom in*) oder unten (*zoom out*) wird ein globaler Floatwert erhöht bzw. verringert.

```
else if (button == 3) { //zoom in
    dummy += 5.0f;
}
else if (button == 4) { //zoom out
    dummy -= 5.0f;
}
```

Dieser Wert wird zu Beginn der *timer*-Funktion als Z-Wert einer Translation benutzt. Auch hier bezieht sich die Translation auf die “Kamera”, da sich noch kein zu zeichnendes Objekt auf Matrizenstapel befindet.

```
modelViewMatrix.Translate(0, 0, dummy);
```

Benützen Sie anstatt der Orthogonalprojektion jetzt die perspektivische Projektion

Da im ursprünglichen Anwendungsrahmen die Parameter für die Orthogonalprojektion via *ViewFrustum.SetOrthogographic* ausschließlich in der *ChangeSize*-Funktion gesetzt wurden, wurde als Ansatz auch nur in diese Funktion eingegriffen.
Der durch die GUI gesetzte Boolean-Flag *bPersp* bestimmt, ob die klassische Orthogonalprojektion oder die perspektivische Projektion (via *SetFrustum*) genutzt wird.

```
TwAddVarRW(bar, "Perspective", TW_TYPE_BOOLCPP, &bPersp, "");
```

Bei *SetFrustum* können die meisten Parameter von *SetOrthographic* unverändert übernommen werden, lediglich die Clipping Planes mussten erweitert werden.

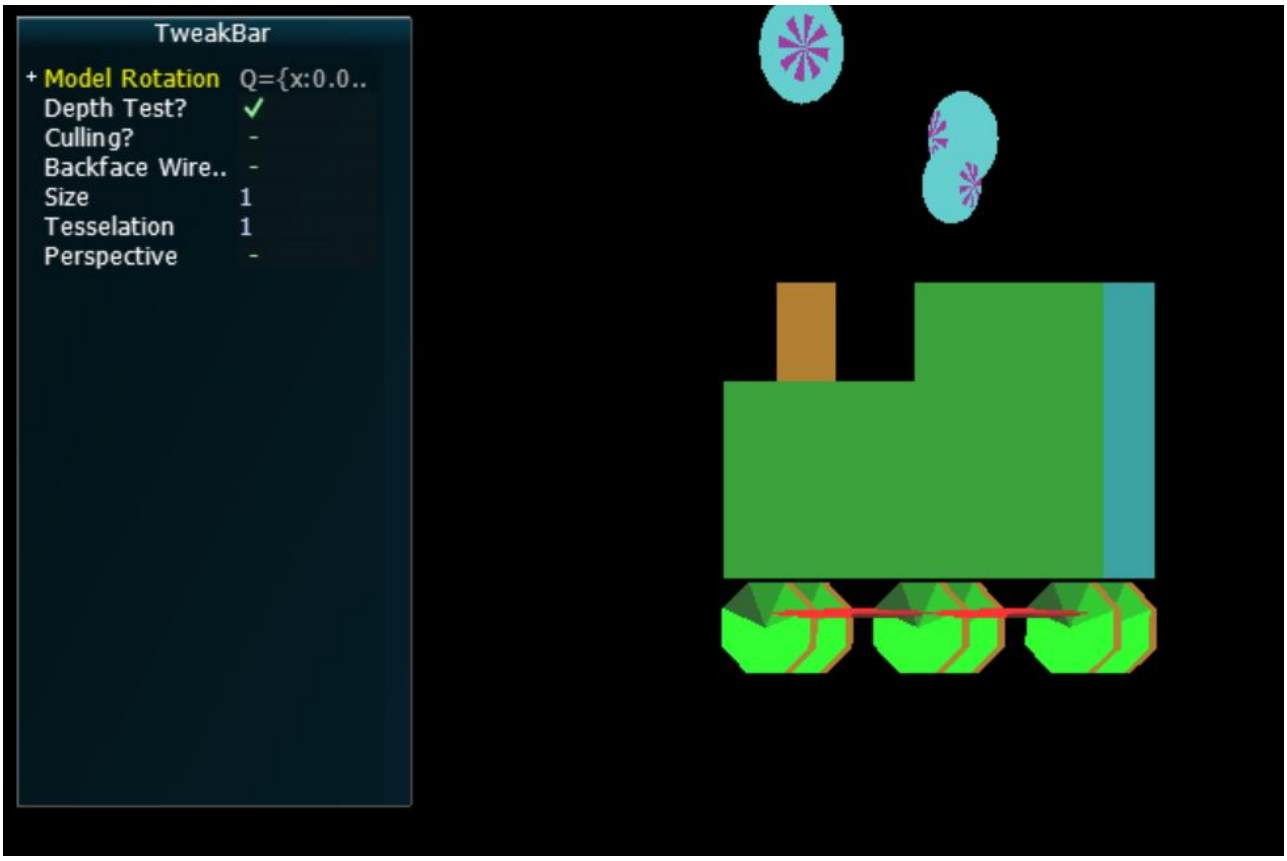
```
if (!bPersp) {
    viewFrustum.SetOrthographic(-nRange * float(w) / float(h), nRange * float(w) / float(h), -nRange, nRange, -nRange, nRange);
}
else {
    viewFrustum.SetFrustum(-nRange * float(w) / float(h), nRange * float(w) / float(h), -nRange, nRange, 250, -5 * nRange);
}
```

Darüberhinaus war es erforderlich, das Modell bei einem Wechsel in die perspektivische Projektion in der *timer*-Funktion zu verschieben und leicht zu skalieren, um eine möglichst ähnliche Darstellung zu ermöglichen.

```
if (bPersp) {
    ChangeSize(gW, gH);
    modelViewMatrix.Translate(0.0, 0.0, -500.0);
    modelViewMatrix.Scale(1.3, 1.3, 1.3);
}
else {

    ChangeSize(gW, gH);
}
```

Orthographisch:



Perspektivisch: