3. Praktikum in Computergrafik und Bildverarbeitung

Lisa Obermaier, Simon Thum

Frist 21 Mai 2019, 23:59

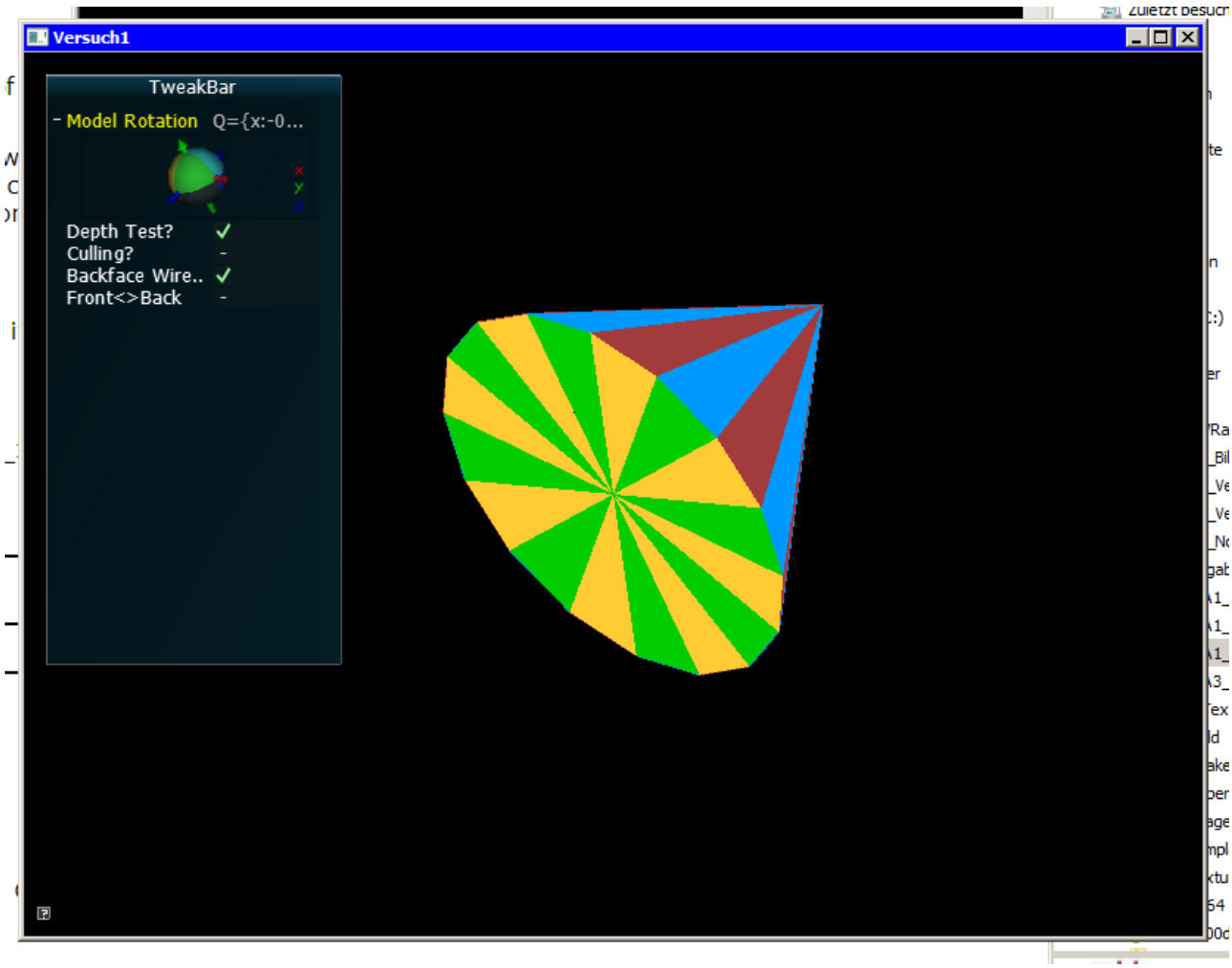
Korrigieren Sie das Programm "A1_Versuch1a" so, dass alle Vorderseiten nach außen zeigen und testen Sie das korrigierte Programm erneut.

Die Rückseite des Kegelbodens wurde angezeigt, da bei einem t*riangle fan* die Vorder- und Rückseite durch die Reihenfolge der Kreispunkte bestimmt wird. Es kann durch die Funktion *glFrontFace* gewählt werden, mit den Parametern *GL_CW* für *clockwise* und *GL_CCW* für *counter-clockwise*. Eine Korrektur des Quelltextes ist entsprechend einfach: Bei der Befüllung des Vertex-Arrays wird einfach die Reihenfolge umgedreht, indem die Schleifenbedingung angepasst wird.

Vgl. mit auskommentierter Originalbedingung:

```
//for (float angle = 0.0f; angle < (2.0f*GL_PI); angle += (GL_PI / 8.0f))
for (float angle = (2.0f*GL_PI); angle > 0.0f; angle -= (GL_PI / 8.0f))
{
    // Berechne x und y Positionen des naechsten Vertex
    float x = 50.0f*sin(angle);
    float y = 50.0f*cos(angle);

    // Alterniere die Farbe zwischen Rot und Gruen
    if ((iPivot % 2) == 0)
        boden.Color4f(1, 0.8, 0.2, 1);
    else
        boden.Color4f(0, 0.8, 0, 1);

    // Inkrementiere iPivot um die Farbe beim naechsten mal zu wechseln
    iPivot++;
```

Ergebnis: Auch bei eingeschaltetem *backface wire* ist der Kegelboden kein *wireframe*.

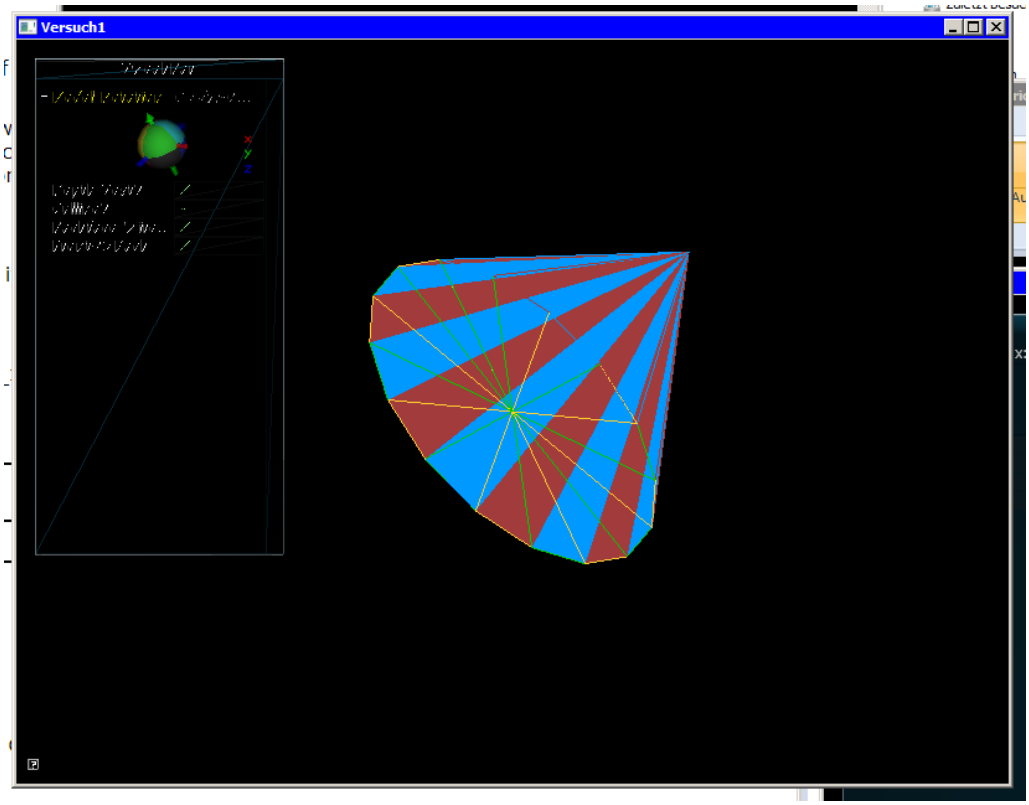


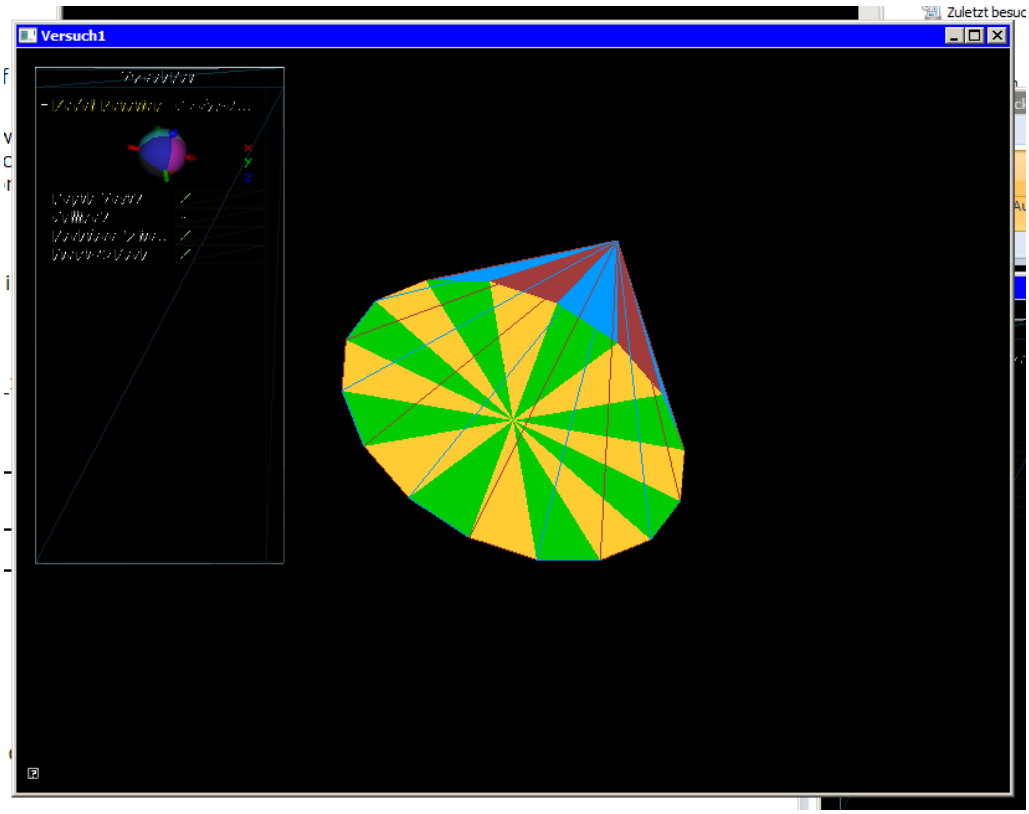
Bauen Sie noch einen weiteren Schalter in das GUI des Programms ein, um Vorder- und Rückseiten der Polygone zu vertauschen.

Einbau eines Schalters in die GUI erfolgt über Deklaration einer neuen Booleanvariable und Verknüpfen dieser Variable in der *TweakBar*. Vor dem Zeichnen kann in RenderScene dann in Abhängigkeit der Variable das *glFrontFace* auf *clockwise* oder *counter-clockwise* gesetzt werden, um die Flächen "in der falschen Richtung" zu erstellen und entsprechend die Rückseiten nach außen zu drehen.

```
35    bool bDepth = true;
36    bool bFrontback = false;
37
38
39    //GUI
40    TwBar *bar;
41    void InitGUI()
42    {
43        bar = TwNewBar("TweakBar");
44        TwDefine(" TweakBar size='200 400'");
45        TwAddVarRW(bar, "Model Rotation", TW_TYPE_QUAT4F, &rotation, "");
46        TwAddVarRW(bar, "Depth Test?", TW_TYPE_BOOLCPP, &bDepth, "");
47        TwAddVarRW(bar, "Culling?", TW_TYPE_BOOLCPP, &bCull, "");
48        TwAddVarRW(bar, "Backface Wireframe?", TW_TYPE_BOOLCPP, &bOutline, "");
49        TwAddVarRW(bar, "Front<>Back", TW_TYPE_BOOLCPP, &bFrontback, "");
50        //Hier weitere GUI Variablen anlegen. Für Farbe z.B. den Typ TW_TYPE_COLOR4F benutzen
51    }
52
53    void CreateGeometry() { ... }
115
116   // Aufruf draw scene
117   void RenderScene(void)
118   {
119       // Clearbefehle für den color buffer und den depth buffer
120       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
121
122       // Schalte culling ein falls das Flag gesetzt ist
123       if (bCull)
124           glEnable(GL_CULL_FACE);
125       else
126           glDisable(GL_CULL_FACE);
127
128       // Schalte depth testing ein falls das Flag gesetzt ist
129       if (bDepth)
130           glEnable(GL_DEPTH_TEST);
131       else
132           glDisable(GL_DEPTH_TEST);
133
134       // Zeichne die Rückseite von Polygonen als Drahtgitter falls das Flag gesetzt ist
135       if (bOutline)
136           glPolygonMode(GL_BACK, GL_LINE);
137       else
138           glPolygonMode(GL_BACK, GL_FILL);
139
140       if (bFrontback)
141           glFrontFace(GL_CCW);
142       else
143           glFrontFace(GL_CW);
```

Ergebnis: Durch eingeschaltetes *Backface Wire* werden die Rückseiten der t*riangle fans* als Wireframes angezeigt, und es ist möglich, durch sie hindurch die Nicht-Wireface-Außenseiten im Inneren des Objekts zu erkennen.





Analysieren und bewerten Sie den Unterschied in der Rendertechnik im Vergleich zu "A1_Versuch1a".

In A1_Versuch1a werden die Vertexdaten und Farben direkt in einer Schleife in das Objekt geladen, und in A1_Versuch1b hingegen werden die Daten in der Schleife in ein Array geladen und nach Ende der Schleife via *m3dLoadVector* dem Objekt übergeben.

Eine Bewertung gestaltet sich als schwierig, da nicht genug Debugdaten zur Verfügung stehen, es ist aber davon auszugehen, dass die *LoadVector*-Methode existiert, weil es performanter ist, die Daten auf einmal zu kopieren, als das Objekt die ganze Zeit offen zu halten.

Programmieren Sie mindestens die folgenden komplexeren Objekte selber: Quader, Zylinder, Kugel und zwar so, dass Sie sie von den Ausmaßen und (bei den letzten beiden) der Tessellierung her parametrieren können.

Der Quader besteht aus sechs Seiten, eine als Beispiel:

```
// red
m3dLoadVector4(quadColors[3], 0.635, 0.235, 0.235, 1);
m3dLoadVector4(quadColors[2], 0.635, 0.235, 0.235, 1);
m3dLoadVector4(quadColors[1], 0.635, 0.235, 0.235, 1);
m3dLoadVector4(quadColors[0], 0.635, 0.235, 0.235, 1);

m3dLoadVector3(quadVertices[0], 0, 100, 0);
m3dLoadVector3(quadVertices[1], 100, 100, 0);
m3dLoadVector3(quadVertices[2], 0, 0, 0);
m3dLoadVector3(quadVertices[3], 100, 0, 0);

quad1.Begin(GL_TRIANGLE_STRIP, 4);
quad1.CopyVertexData3f(quadVertices);
quad1.CopyColorData4f(quadColors);
quad1.End();
```

Sechs Seiten ergeben einen Quader:

```
void DrawSquare() {
    quad1.Draw();
    quad2.Draw();
    quad3.Draw();
    quad4.Draw();
    quad5.Draw();
    quad6.Draw();
}
```

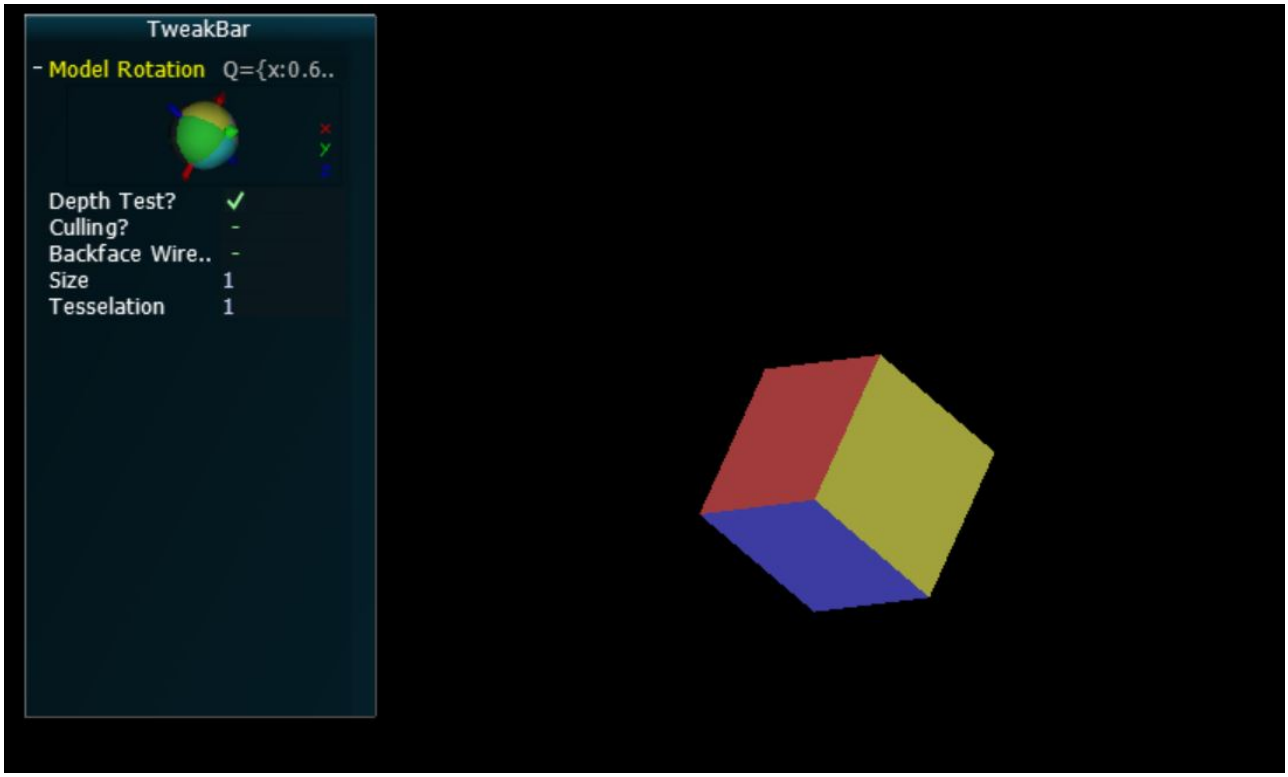
Durch eine weitere in der GUI veränderbaren Variable *scsize* können die Ausmaße der Objekte geändert werden.

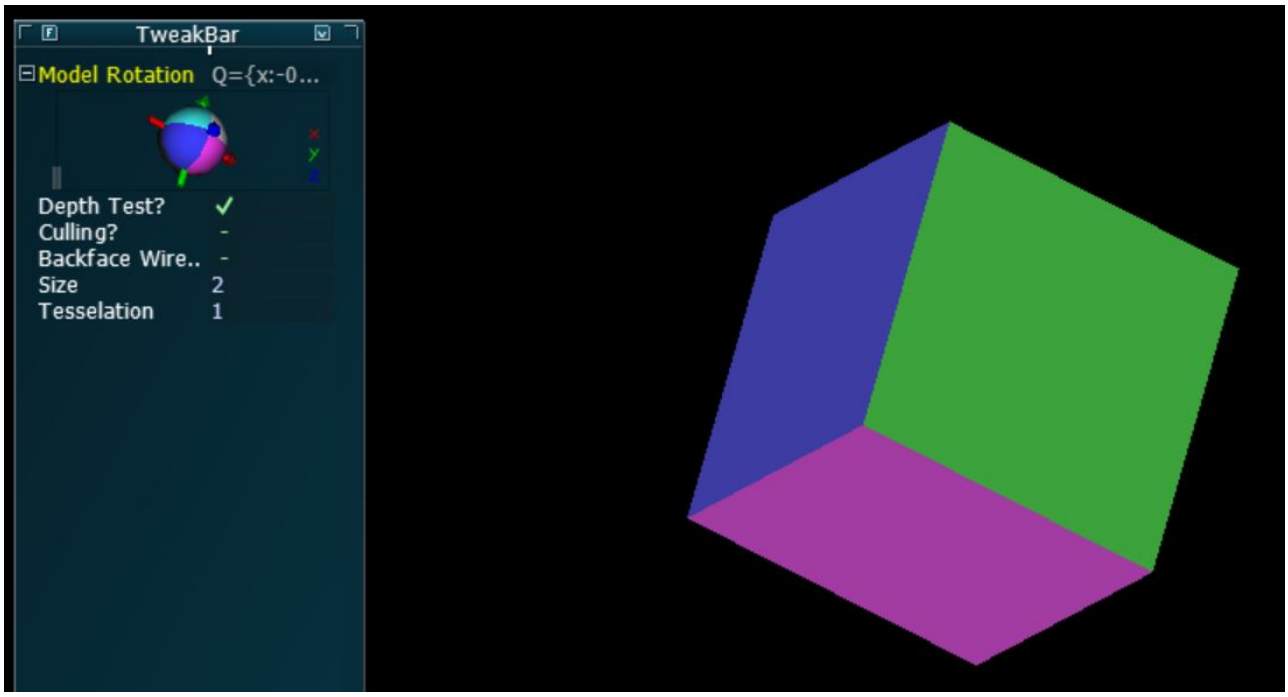
```
TwAddVarRW(bar, "Size", TW_TYPE_FLOAT, &scsize, "");
```

Dafür wird in *RenderScene* vor dem Zeichnen die Identitätsmatrix geladen und in allen drei Dimensionen mit dem gleichen Faktor skaliert, damit keine Verzerrungen entstehen.

```
modelViewMatrix.LoadIdentity();
modelViewMatrix.Scale(scsize, scsize, scsize);

M3DMatrix44f rot;
m3dQuatToRotationMatrix(rot, rotation);
modelViewMatrix.MultMatrix(rot);
//modelViewMatrix.Translate(0.0, 0.0, -200.0);
//setze den Shader f0r das Rendern
shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTR
//Zeichne Konus
//konus.Draw();
//DrawCylinder();
DrawSquare();
//Auf fehler 0berpr0fen
```

Ergebnis: ein Quader (in unserem Beispiel sogar der Spezialfall eines Würfels) in verschiedenen Größen. Zur besseren Unterscheidung haben alle sechs Seiten eine andere Farbe erhalten.





Für den Zylinder wird der Kegelboden aus A1_Versuch1a zweimal mit Abstand in der Z-Achse gezeichnet, und alle Punkte außer dem Zentrum zwischengespeichert. Diese Punkte werden als Eckpunkte für einen *QuadStrip* benutzt, der den Zylindermantel darstellt.

```
void DrawCylinder() {
    boden.Draw();
    zylrand.Draw();
    boden2.Draw();
}
```

Zu beachten ist, die letzten beiden Mantelpunkte auf die Startpunkte zu setzen, um den Mantel abzuschließen.

```
// Erzeuge einen weiteren Triangle_Fan um den Boden zu bedecken
M3DVector3f bodenVertices[18];
M3DVector4f bodenColors[18];
M3DVector3f randVertices[36];
// Das Zentrum des Triangle_Fans ist im Ursprung
m3dLoadVector3(bodenVertices[0], 0, 0, -75.0);
m3dLoadVector4(bodenColors[0], 1, 0, 0, 1);
int i = 1;
for (float angle = 0.0f; angle < (2.0f*GL_PI); angle += (GL_PI / 8.0f))
{
    // Berechne x und y Positionen des naechsten Vertex
    float x = 50.0f*sin(angle);
    float y = 50.0f*cos(angle);
    // Alterniere die Farbe zwischen Rot und Gruen
        m3dLoadVector4(bodenColors[i], 1, 0.8, 0.2, 1);
    // Spezifiziere den naechsten Vertex des Triangle_Fans
        m3dLoadVector3(bodenVertices[i], x, y, -75.0);
        m3dLoadVector3(randVertices[2*i-1], x, y, -75.0);
    i++;
}
boden.Begin(GL_TRIANGLE_FAN, 18);
boden.CopyVertexData3f(bodenVertices);
boden.CopyColorData4f(bodenColors);
boden.End();
```

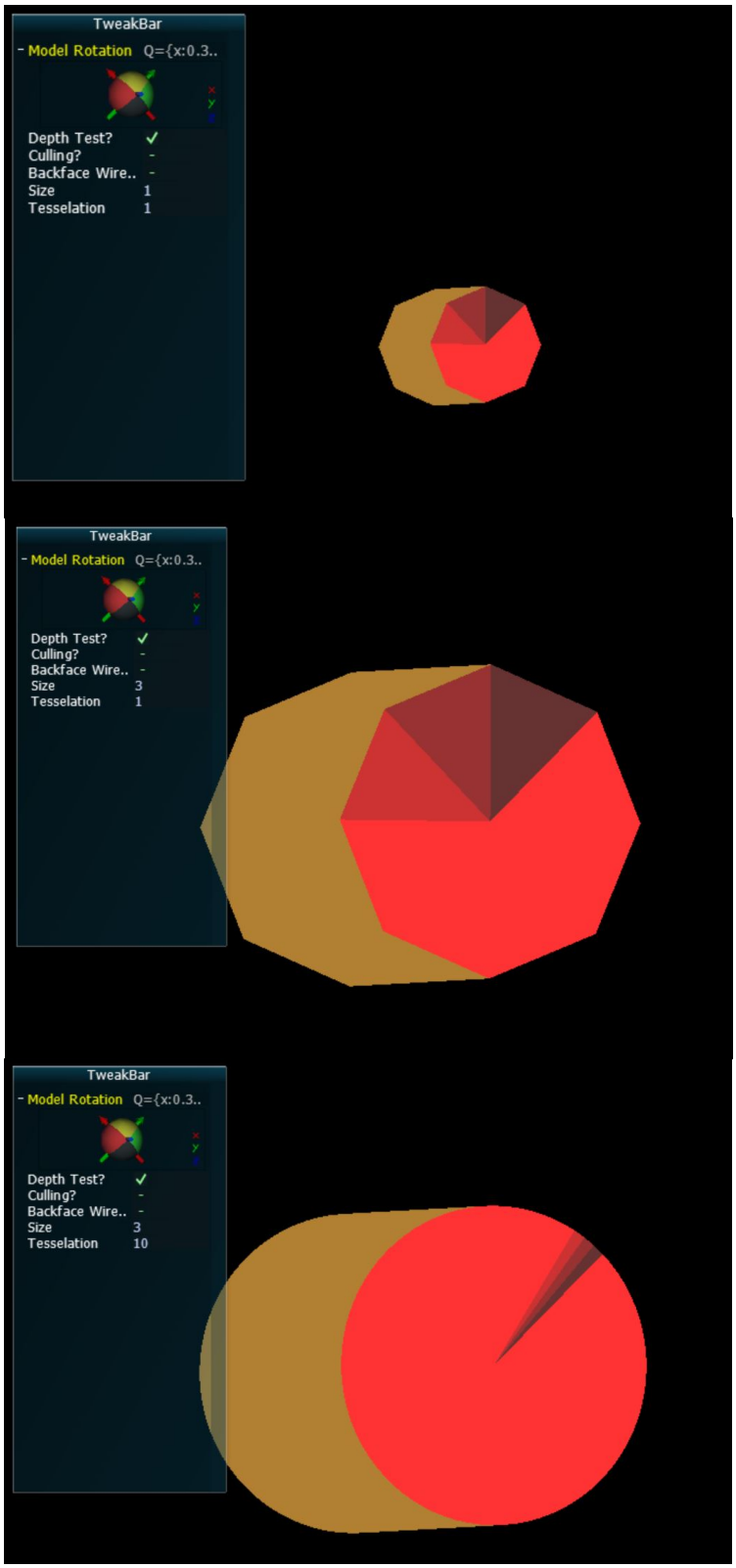
Die Tesslierung wird über einen weiteren GUI-Parameter gesteuert. Dieser ist entsprechend Codebeispiel implementiert und wurde um Safeguards erweitert, um Werte kleiner eins zu verbieten, um Teilungen durch Null und anderes unerwartetes Verhalten zu verhindern.

```
TwAddVarCB(bar, "Tesselation", TW_TYPE_UINT32, SetTesselation, GetTesselation, NULL, "");
```

```
//Set Funktion fuer GUI, wird aufgerufen wenn Variable im GUI geaendert wird
void TW_CALL SetTesselation(const void *value, void *clientData)
{
    //printf("SETTESS CALLED\n");
    //Pointer auf gesetzten Typ casten (der Typ der bei TwAddVarCB angegeben wurde)
    const unsigned int* uintptr = static_cast<const unsigned int*>(value);

    //Setzen der Variable auf neuen Wert
    if (*uintptr > 0 && *uintptr < 33) {
        tesselation = *uintptr;
    }

    //Hier kann nun der Aufruf gemacht werden um die Geometrie mit neuem Tesselationsfaktor zu erzeugen
    CreateGeometry();

}

//Get Funktion fuer GUI, damit GUI Variablen Wert zum anzeigen erhaelt
void TW_CALL GetTesselation(void *value, void *clientData)
{
    //printf("GETTESS CALLED\n");
    //Pointer auf gesetzten Typ casten (der Typ der bei TwAddVarCB angegeben wurde)
    unsigned int* uintptr = static_cast<unsigned int*>(value);

    //  //Variablen Wert and GUI weiterreichen
    *uintptr = tesselation;
}
```

Die Tesselierung ändert die Anzahl der Schleifendurchläufe für die Generierung der Kreispunkte, je mehr Schleifendurchläufe (und entsprechend Punkte im Vertexarray), desto runder die Kreisannäherung. Der Zylindermantel übernimmt weiterhin alle Punkte für die Kanten. Die ersten Farben im Array der Teildreiecke werden zu Illustrationszwecken dunkler gestaltet, um das Größenverhältnis in Abhängigkeit der Tesselierung darzustellen.

Ergebnis: Zylinder in verschieden Größen und Tesselierungen



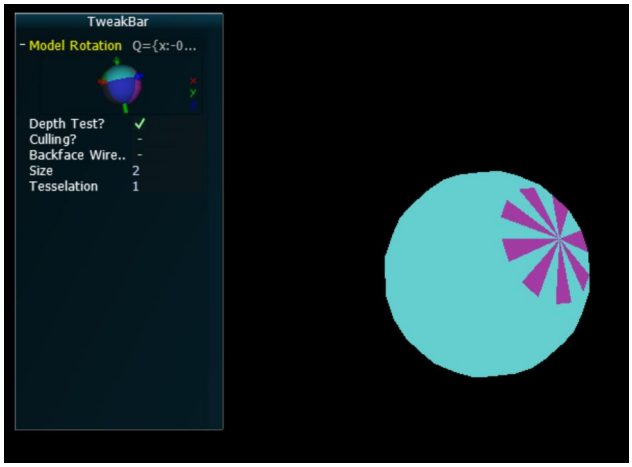
Die Kugel wird über einen *QuadStrip* realisiert. Die Kugel erhält oben und unten je einen Deckel durch Kreispunkte, deren Anzahl analog zu der Kreisfläche des Kegels von der Tesselierung bestimmt wird.

```
//Hütchen vorne, Farben abwechselnd
for (int k = 1; k <= 2 + 16 * tesselation; k++) {
    currRadius = sqrt((pow(radius, 2) - (pow((radius - currentheight), 2)))); //Radius
    xFirst = currRadius * sin((GL_PI / (8.0f * tesselation)) * k);
    yFirst = currRadius * cos((GL_PI / (8.0f * tesselation)) * k);
    xSecond = currRadius * sin((GL_PI / (8.0f * tesselation)) * (k + 1));
    ySecond = currRadius * cos((GL_PI / (8.0f * tesselation)) * (k + 1));

    m3dLoadVector3(sphericalVertices[i], 0, 0, sphericalDepth);
    m3dLoadVector3(sphericalVertices[i + 1], xFirst, yFirst, (sphericalDepth - currentheight));
    m3dLoadVector3(sphericalVertices[i + 2], 0, 0, sphericalDepth);
    m3dLoadVector3(sphericalVertices[i + 3], xSecond, ySecond, (sphericalDepth - currentheight));
```

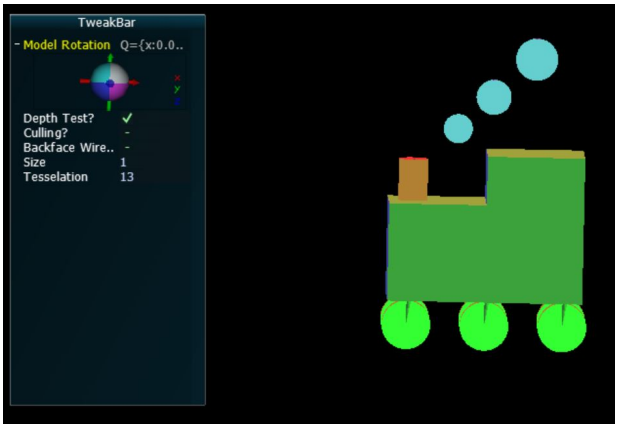
Dazwischen liegt ein Mantel, der aus geschichteten breiten Ringen (*QuadStrips* auf Kreispunkten) mit wachsendem und wieder abnehmendem Radius. Auch die Anzahl der Ringe wird durch die Tesselierung bestimmt. Eine Tesselierung von eins meint zehn Ringe, zwei 20, drei 30, etc.

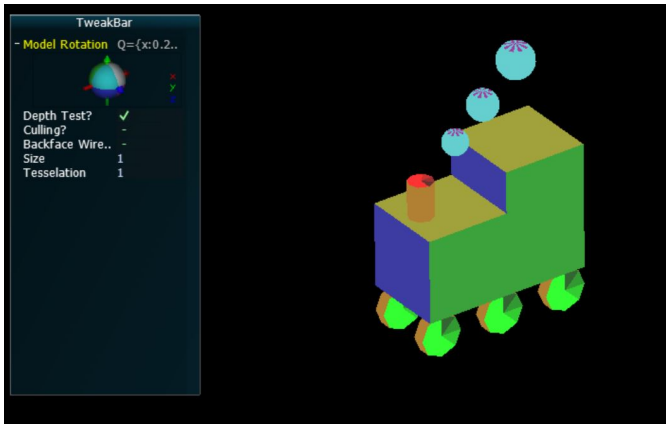
```
//Mantel einfarbig
for (int l = 1; l <= 8 * tesselation + 2 * (tesselation - 1); l++) {
    for (int k = 1; k <= 2 + 16 * tesselation; k++) {
        currRadius = sqrt((pow(radius, 2) - (pow((radius - currentheight), 2))));
        xFirst = currRadius * sin((GL_PI / (8.0f * tesselation)) * k);
        yFirst = currRadius * cos((GL_PI / (8.0f * tesselation)) * k);
        xSecond = currRadius * sin((GL_PI / (8.0f * tesselation)) * (k + 1));
        ySecond = currRadius * cos((GL_PI / (8.0f * tesselation)) * (k + 1));

        float secRadius = sqrt((pow(radius, 2) - (pow((radius - (currentheight + (10.0 / tesselation))), 2))));
        float xNew = secRadius * sin((GL_PI / (8.0f * tesselation)) * k);
        float yNew = secRadius * cos((GL_PI / (8.0f * tesselation)) * k);
        float x1New = secRadius * sin((GL_PI / (8.0f * tesselation)) * (k + 1));
        float y1New = secRadius * cos((GL_PI / (8.0f * tesselation)) * (k + 1));

        float secHeightFirst = (sphericalDepth - (fixedHeight * l));
        float secHeightSecond = (sphericalDepth - (fixedHeight * (l + 1)));

        m3dLoadVector3(sphericalVertices[i], xFirst, yFirst, secHeightFirst); //fixedHeight * l
        m3dLoadVector3(sphericalVertices[i + 1], xNew, yNew, secHeightSecond); //fixedHeight * (l + 1)
        m3dLoadVector3(sphericalVertices[i + 2], xSecond, ySecond, secHeightFirst); //fixedHeight * l
        m3dLoadVector3(sphericalVertices[i + 3], x1New, y1New, secHeightSecond);    //fixedHeight * (l + 1)

        //Farben setzen
        m3dLoadVector4(sphericalColors[i], 0.396, 0.810, 0.810, 1);
        m3dLoadVector4(sphericalColors[i + 1], 0.396, 0.810, 0.810, 1);
        m3dLoadVector4(sphericalColors[i + 2], 0.396, 0.810, 0.810, 1);
        m3dLoadVector4(sphericalColors[i + 3], 0.396, 0.810, 0.810, 1);

        i += 4;
    }
    p++;
    currentheight += (10.0 / tesselation);
}
```

Ergebnis: Kugel mit veränderbarer Tesselierung.



Programmieren Sie Ihre eigene Szene, indem Sie einfache Grundbausteine über mehrere Hierarchie-Ebenen zu komplexeren Objekten zusammensetzen.





Diese Lokomotive ist zusammengesetzt aus drei Räderpaaren aus Zylindern, zwei Quadern für den Korpus, einem Zylinder für den Schlot sowie drei Kugeln als Rauch.

Mit den Funktionen *Translate, Scale* und *Rotate* lassen sich die Körper beliebig verschieben und anordnen. So kann beispielsweise ein Zylinder mit reduzierter Tiefe eines Zylinders ein Rad darstellen.

```
//erster Quader
modelViewMatrix.PushMatrix();
DrawSquare();
```

```
//erstes Set R0der
modelViewMatrix.Scale(0.5, 0.5, 0.1);
modelViewMatrix.Translate(40, -50, 150);
shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
modelViewMatrix.PushMatrix();
DrawCylinder();
modelViewMatrix.Translate(0, 0, 700);
shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
DrawCylinder();
```

```
//Schornstein
//Achtung, wegen Rotate anschlie0end y und z vertauscht D:
modelViewMatrix.Translate(-75, 120, 50);
modelViewMatrix.Rotate(90, 90, 0, 0);
modelViewMatrix.Scale(0.3, 0.3, 0.3);
shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
DrawCylinder();
```

```
//Rauch
modelViewMatrix.Translate(150, 0, -230);
shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
DrawSphere();
modelViewMatrix.Scale(1.2, 1.2, 1.2);
modelViewMatrix.Translate(100, 0, -100);
shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
DrawSphere();
modelViewMatrix.Scale(1.2, 1.2, 1.2);
modelViewMatrix.Translate(100, 0, -100);
shaderManager.UseStockShader(GLT_SHADER_FLAT_ATTRIBUTES, transformPipeline.GetModelViewProjectionMatrix());
DrawSphere();
```