LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prof. Dr. Andreas Butz | Prof. Dr. Ing. Axel Hoppe

Dipl.-Medieninf. Dominikus Baur
Dipl.-Medieninf. Sebastian Boring

Übung: Computergrafik 1

Geometrische Primitive
OpenGL Zeichenarten
Kurven

Primitive

2D Objekte

- Beispiel:

```
glBegin(GL_POLYGON);
    glColor3f(1.0f, 0.0f, 0.0f); // red
    glVertex3f(-1.0f, -1.0f, 0.0f);
    glVertex3f(1.0f, -1.0f, 0.0f);
    glColor3f(0.0f, 0.0f, 1.0f); // blue
    glVertex3f(1.0f, 1.0f, 0.0f);
    glVertex3f(-1.0f, 1.0f, 0.0f);
glEnd();
```

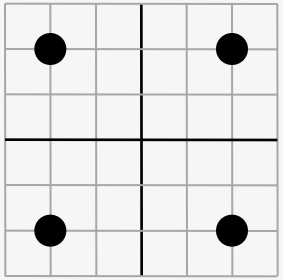
- Weitere Formen:

```
GL_POINTS, GL_LINES (je 2 Punkte verbunden)
GL_LINE_STRIP, GL_LINE_LOOP
GL_QUADS, GL_POLYGON, GL_TRIANGLES,
GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUAD_STRIP
```

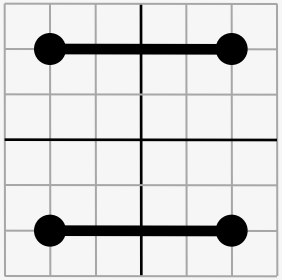
- 3D-Objekte müssen aus 2D-Objekten zusammengesetzt werden (s. Übungsblatt)

(Quelle: http://www.opengl.org)

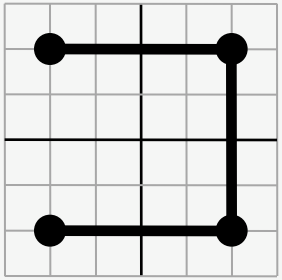
Beispiele:



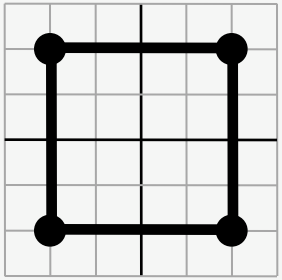
GL_POINTS



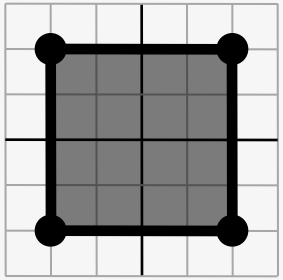
GL_LINES



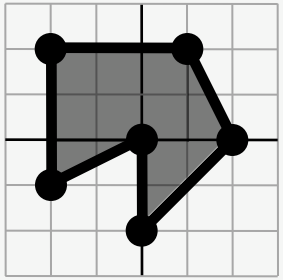
GL_LINE_STRIP



GL_LINE_LOOP



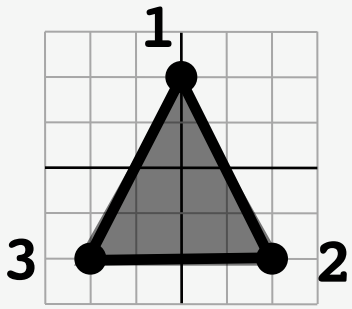
GL_QUADS



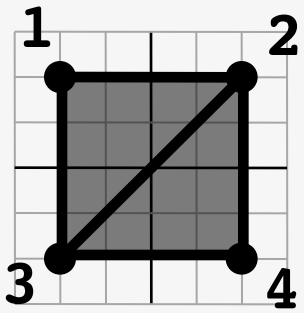
GL_POLYGON

(Quelle: http://www.opengl.org)

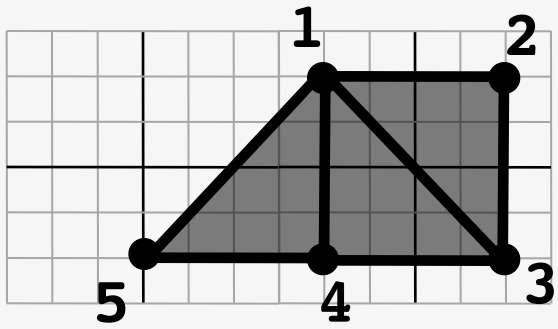
2D Objekte



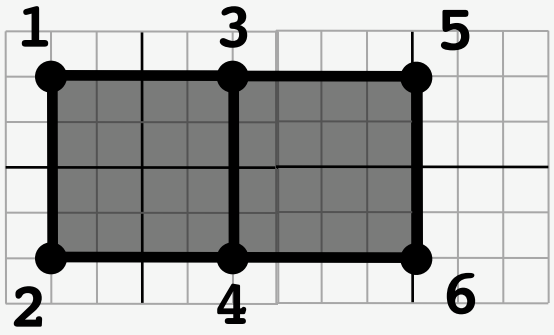
GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN



GL_QUAD_STRIP

(Quelle: http://www.opengl.org)

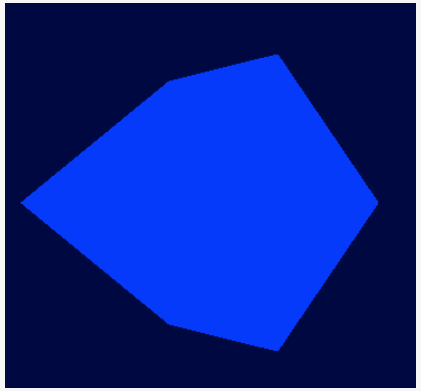
glBegin – glEnd, glVertex

- Innerhalb eines `glBegin ... glEnd` Blocks dürfen nur bestimmte gl-Befehle vorkommen, u.a. `glColor, glVertex, glTexCoord`
- `glVertex` *muss* innerhalb von `glBegin/glEnd` aufgerufen werden!
- Varianten:
  - `glVertex2f(x, y)` - z wird auf 0 gesetzt
  - `glVertex3f(x, y, z)`
  - `glVertex4f(x, y, z, w)` - manipuliert zusätzlich die homogene Komponente des Vektors
  - `glVertex3fv( v )` übergibt einen Pointer auf einen Vektor

Würfel in OpenGL

```
glBegin(GL_QUADS);
    glVertex3f( 1.0f, 1.0f,-1.0f);
    glVertex3f(-1.0f, 1.0f,-1.0f);
    glVertex3f(-1.0f, 1.0f, 1.0f);
    glVertex3f( 1.0f, 1.0f, 1.0f);

    glVertex3f( 1.0f,-1.0f, 1.0f);
    glVertex3f(-1.0f,-1.0f, 1.0f);
    glVertex3f(-1.0f,-1.0f,-1.0f);
    glVertex3f( 1.0f,-1.0f,-1.0f);

    glVertex3f( 1.0f, 1.0f, 1.0f);
    glVertex3f(-1.0f, 1.0f, 1.0f);
    glVertex3f(-1.0f,-1.0f, 1.0f);
    glVertex3f( 1.0f,-1.0f, 1.0f);
```

```
    glVertex3f( 1.0f,-1.0f,-1.0f);
    glVertex3f(-1.0f,-1.0f,-1.0f);
    glVertex3f(-1.0f, 1.0f,-1.0f);
    glVertex3f( 1.0f, 1.0f,-1.0f);

    glVertex3f(-1.0f, 1.0f, 1.0f);
    glVertex3f(-1.0f, 1.0f,-1.0f);
    glVertex3f(-1.0f,-1.0f,-1.0f);
    glVertex3f(-1.0f,-1.0f, 1.0f);

    glVertex3f( 1.0f, 1.0f,-1.0f);
    glVertex3f( 1.0f, 1.0f, 1.0f);
    glVertex3f( 1.0f,-1.0f, 1.0f);
    glVertex3f( 1.0f,-1.0f,-1.0f);
glEnd();
```



Primitivmanipulation

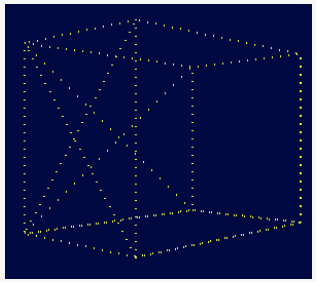
- Wie Punkte und Linien gezeichnet werden lässt sich mit bestimmten Befehlen manipulieren:
  - `glColor3f/glColor4f` setzt einen Farbwert
  - `glPointSize(float)` bestimmt die Dicke von Punkten
  - `glEnable(GL_POINT_SMOOTH)` aktiviert Kantenglättung

  - `glLineWidth(float)` bestimmt die Dicke von Linien
  - `glEnable(GL_LINE_SMOOTH)` aktiviert Kantenglättung

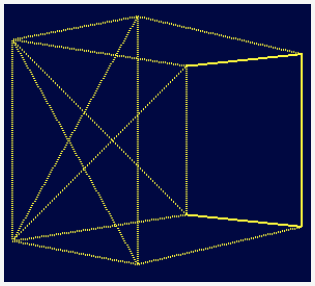
Primitivmanipulation II

- Wie Punkte und Linien gezeichnet werden lässt sich mit bestimmten Befehlen manipulieren:
  - `glLineStipple(int, short)` erzeugt ein Muster in der Linie (muss mit glEnable(GL_LINE_STIPPLE) aktiviert werden)
  - Der 16-Bit Short Wert bestimmt das Muster (Bit gesetzt -> Pixel wird gezeichnet, Bit nicht gesetzt -> Pixel wird nicht gezeichnet
  - Durch Angabe eines Faktors lässt sich das Muster strecken

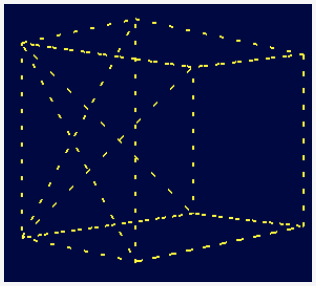
```
glLineWidth(2);
glLineStipple(1, 0x0101);
glEnable(GL_LINE_STIPPLE);
glPointSize(20);
```



```
glLineWidth(2);
glLineStipple(1, 0xAAAA);
glEnable(GL_LINE_STIPPLE);
glPointSize(20);
```

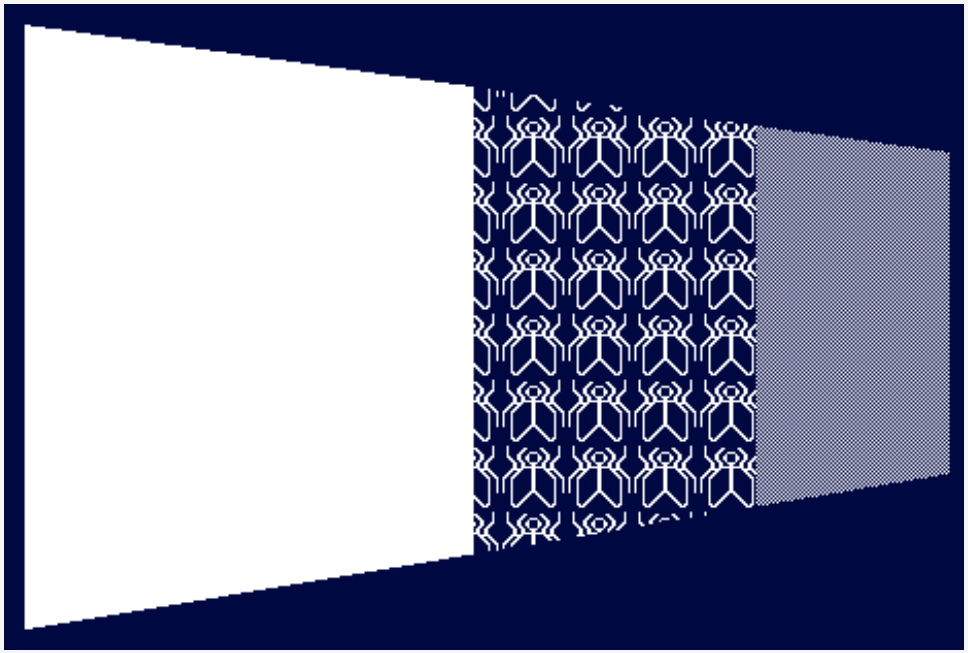


```
glLineWidth(2);
glLineStipple(1, 0x000F);
glEnable(GL_LINE_STIPPLE);
glPointSize(20);
```



Primitivmanipulation III

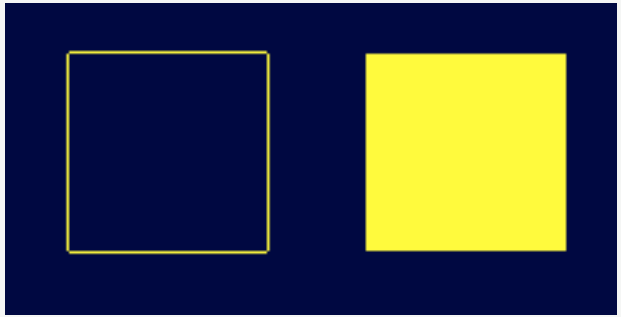
- Wie Punkte und Linien gezeichnet werden lässt sich mit bestimmten Befehlen manipulieren:
  - `glPolygonStipple(GLubyte*)` erzeugt ein Muster in einem Polygon (Pointer auf 32 x 32 Bit Array)
    (muss mit glEnable(GL_POLYGON_STIPPLE) aktiviert werden)



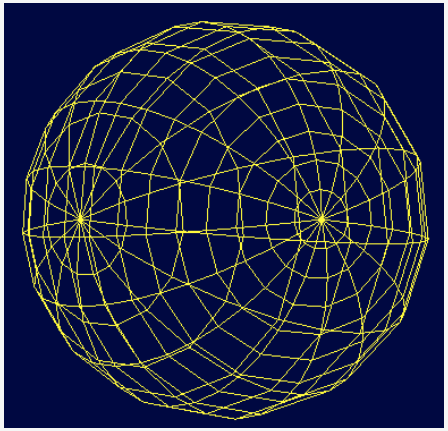
(Quelle: http://www.opengl.org.ru/docs/pg/0204.html)

Primitivmanipulation IV

- Wie Punkte und Linien gezeichnet werden lässt sich mit bestimmten Befehlen manipulieren:
  - `glPolygonMode(face, mode)` beeinflusst, wie Polygone gezeichnet werden:
  - `face` bestimmt den Teil des Polygons (`GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`)
  - `mode` gibt den Zeichenmodus für den gewählten Teil an (`GL_POINT, GL_LINE, GL_FILL`)
- Achtung! Vertices die gegen den Uhrzeigersinn gezeichnet werden gelten als vorne, im Uhrzeigersinn als hinten!



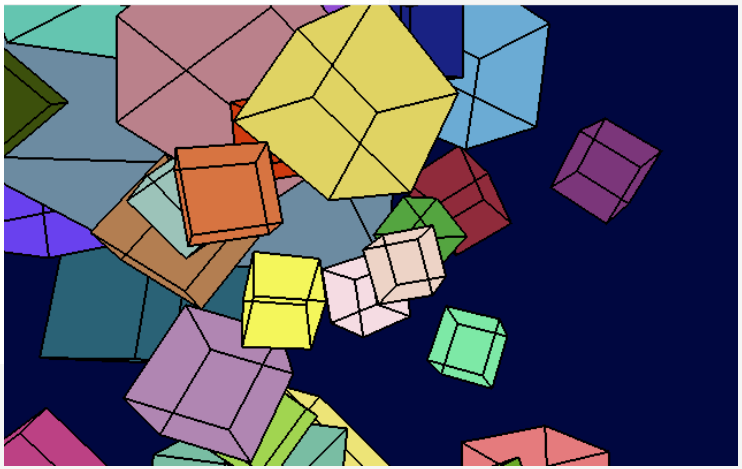
```
glPolygonMode( GL_FRONT, GL_LINE );
```



```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

Primitivmanipulation V

```
for(int i = 0; i < 50; i++){
    glPushMatrix();
    float nupos[] = {rand() % 20 - 10, rand() % 20 - 10, rand() % 10 - 5};
    glTranslatef(nupos[0], nupos[1], nupos[2]);
    glRotatef(rand() % 360, 0, 1, 0);
    glRotatef(rand() % 360, 1, 0, 0);
    glRotatef(rand() % 360, 0, 0, 1);

    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glColor3i(rand(), rand(), rand());
    drawABox();

    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glColor4f(0, 0, 0, 1);
    glLineWidth(2);
    drawABox();

    glPopMatrix();
}
```



OpenGL Zeichenmodi

- Primitive lassen sich in OpenGL auf vier verschiedene Arten zeichnen:
  - Im sogenannten *immediate mode* mit glBegin/glEnd
    - Performanceeinbußen!
    - Nicht vorhanden in OpenGL 3.x und OpenGL ES
  - mit vorkompilierten *Display Lists*
    - Performant aber statisch
    - (auch nicht unterstützt in OpenGL ES)
  - mit Vertex Arrays oder interleaved Arrays
    - Dynamisch aber nicht performant
  - mit Vertex Buffer Objects (VBOs)
    - Performant und dynamisch

(Quelle: www.performanceopengl.com)

OpenGL Display Lists

- Display Lists speichern eine Reihe von OpenGL-Befehlen, kompilieren sie und schicken sie an die Grafikkarte => weniger Overhead, höhere Performance
- Fast alle Befehle sind erlaubt (anders als in glBegin/glEnd)
  - `GLuint glGenLists(range)` erstellt neue Display Listen und gibt die Nummer der ersten Liste zurück
  - `glNewList(list, mode)` beginnt eine neue Liste (`mode` ist entweder `GL_COMPILE` oder `GL_COMPILE_AND_EXECUTE`)
  - `glEndList()` beendet die aktuelle Liste
  - `glCallList(list)` führt die Anweisungen der Liste aus

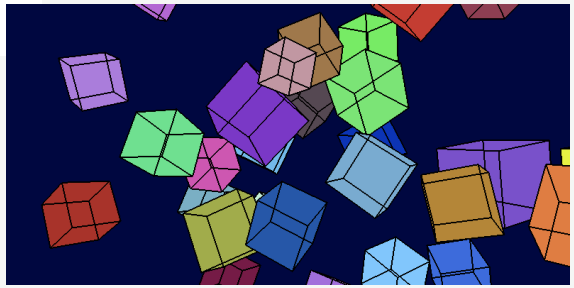
OpenGL Display Lists

```
boxList = glGenLists(1);

glNewList(boxList, GL_COMPILE);
    glBegin(GL_QUADS);
       // front
        glVertex3d(-cubeSize / 2.0, -cubeSize / 2.0, cubeSize / 2.0);
        glVertex3d(cubeSize / 2.0, -cubeSize / 2.0, cubeSize / 2.0);
        glVertex3d(cubeSize / 2.0, cubeSize / 2.0, cubeSize / 2.0);
        glVertex3d(-cubeSize / 2.0, cubeSize / 2.0, cubeSize / 2.0);
```

● ● ●

```
        // top
        glVertex3d(-cubeSize / 2.0, cubeSize / 2.0, -cubeSize / 2.0);
        glVertex3d(cubeSize / 2.0, cubeSize / 2.0, -cubeSize / 2.0);
        glVertex3d(cubeSize / 2.0, cubeSize / 2.0, cubeSize / 2.0);
        glVertex3d(-cubeSize / 2.0, cubeSize / 2.0, cubeSize / 2.0);
    glEnd();
glEndList();
```

```
glCallList(boxList);
```



OpenGL Vertex Arrays

- Nachteile der Display Lists:
  - Keine Änderungen mehr möglich nach Kompilieren
  - Redundanz in den Vertices
- Lösung: Vertex Arrays
- Verfügbar für verschiedene Operationen:
  - `glVertexPointer` (Vertices)
  - `glNormalPointer` (Normalen)
  - `glColorPointer` (Farben)
  - `glIndexPointer` (Indizes)
  - `glTexCoordPointer` (Texturkoordinaten)
  - `glEdgeFlagPointer` (Edge Flags)

OpenGL Vertex Arrays

- Für alle `glXXPointer`-Befehle gleiche Syntax:

  `glVertexPointer(int size, enum type, sizei stride, void* pointer)`

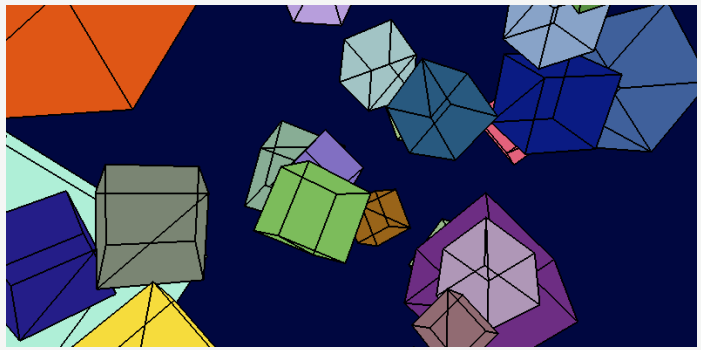
  - `size` Koordinatenzahl im Array (2,3,4)
  - `type` Datentyp (`GL_INT`, `GL_FLOAT`, `GL_DOUBLE`, etc.)
  - `stride` Abstand zwischen einzelnen Elementen im Array in Bytes (z.B. stride = 4 bei GL_FLOAT (4 Byte) um nur jeden zweiten Vertex zu nehmen)
  - `pointer` Der Pointer auf den Speicherbereich der das Array enthält

OpenGL Vertex Arrays

- Weitere Befehle für Vertex Arrays:
  - `glEnableClientState(GL_VERTEX_ARRAY)` aktiviert Vertex Arrays
  - `glDisableClientState(GL_VERTEX_ARRAY)` deaktiviert sie wieder
  - `glDrawArrays(mode, offset, size)` zeichnet ein Vertex Array
    - `mode` entspricht `GL_LINES`, `GL_QUADS`, etc. (s.o.)
    - `offset` erstes Element des Arrays (normalerweise 0)
    - `size` Anzahl der Elemente im Vertex Array

OpenGL Vertex Arrays

```
GLfloat vertices[] = {
    -1, -1, 1, 1, -1, 1,
    1, 1, 1, -1, 1, 1,
    -1, -1, -1, -1, -1, 1,
    -1, 1, 1, -1, 1, -1,
    1, -1, -1, 1, -1, 1,
    1, 1, 1, 1, 1, -1,
    -1, -1, -1, 1, -1, -1,
    1, 1, -1, -1, 1, -1,
    -1, -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1, 1,
    -1, 1, -1, 1, 1, -1,
    1, 1, 1, -1, 1, 1
};

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

glDrawArrays(GL_QUADS, 0, 24);
```

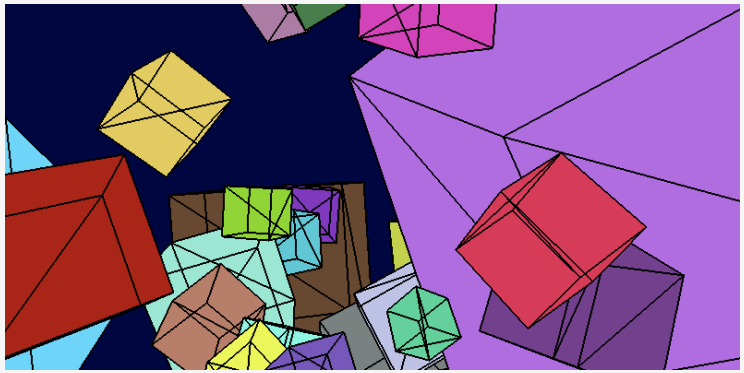


OpenGL Vertex Arrays

- Jeder Punkt in dem Würfel ist weiterhin viermal im Array vertreten
- Daher lassen sich auch Indizes zum Zeichnen von Vertex Arrays angeben
  - `glDrawElements(mode, count, type, indices)`
    - `mode` entspricht `GL_LINES`, `GL_QUADS`, etc. (s.o.)
    - `count` Anzahl der Elemente die gerendert werden sollen
    - `type` Datentyp des Index Arrays (`GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_INT`, etc.)
    - `indices` Pointer auf das Index Array

OpenGL Vertex Arrays

```
GLfloat vertices[] = {
    -1, -1,  1,  1, -1,  1,
     1,  1,  1, -1,  1,  1,
    -1, -1, -1, -1,  1, -1,
     1, -1, -1,  1,  1, -1
};

GLubyte indices[] = {
    0, 1, 2, 4,
    5, 0, 3, 5,
    6, 1, 2, 7,
    4, 6, 7, 5,
    4, 6, 1, 0,
    5, 7, 2, 3
};

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, indices);
```

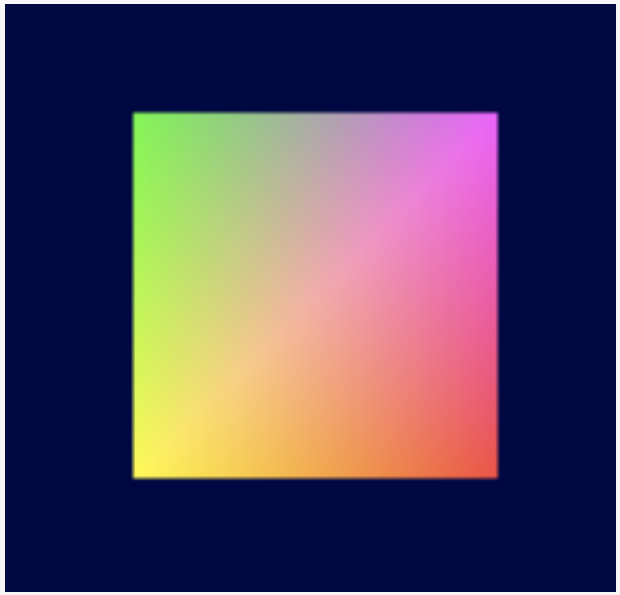


OpenGL Interleaved Arrays

- Um Vertices in verschiedenen Farben zu zeichnen brauchen wir wieder zwei Arrays, eins für die Vertices, eins für die Farben
- Diese Daten lassen sich auch in einem Interleaved Array kombinieren, d.h. abwechselnd Farbwerte und Vertex-Koordinaten
- Der Befehl `glInterleavedArrays (format, stride, pointer)` kümmert sich darum, dass der Client State richtig ist und dass alle internen Arrays passend gefüllt werden.
  - `format` Das Format des Interleaved Arrays
  - `stride` Abstand in Bytes zwischen einzelnen Elementen (d.h. Punkte, Farben, etc.)
  - `pointer` Ein Pointer auf das Interleaved Array
- `format` setzt sich folgendermaßen zusammen:
  - GL_XXX[_YYY][_ZZZ][_AAA]
    - mit Tripeln {V|C|T|N} {2|3|4} {F|D}
    - Vertex, Color, Texture Coords, Normals
    - 2, 3, 4 Elemente im Array pro Vertex, etc.
    - Float, Double

OpenGL Interleaved Arrays

```
GLfloat intertwined[] =
  {1.0, 0.2, 1.0, 1.0, 1.0, 0.0,
   1.0, 0.2, 0.2, 1.0, -1.0, 0.0,
   1.0, 1.0, 0.2, -1.0, -1.0, 0.0,
   0.2, 1.0, 0.2, -1.0, 1.0, 0.0};

glInterleavedArrays(GL_C3F_V3F, 0, intertwined);
glDrawArrays(GL_QUADS, 0, 4);
```



(Quelle: http://www.songho.ca/opengl/gl_vertexarray.html)

OpenGL VBOs

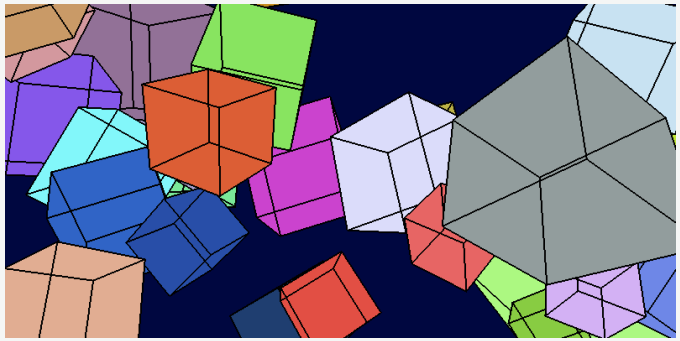
- Vertex Buffer Objects (VBOs) bilden die Brücke zwischen Display Lists und Vertex Arrays
- Sie werden auf der Grafikkarte gehalten (keine Verluste durch Übertragung) können aber trotzdem leicht geändert werden
  - `glGenBuffersARB(GLsizei n, GLuint* ids)` erzeugt neue VBOs
  - `glBindBufferARB(target, id)`
    - `target` entweder `GL_ARRAY_BUFFER_ARB` (Vertices, Farben) oder `GL_ELEMENT_ARRAY_BUFFER_ARB` (Indizes)
    - `id` des VBOs
  - `glBufferDataARB(target, size, void* data, usage)`
    - `size` Größe des Arrays in Bytes
    - `usage` Nutzung des VBOs:
      - `GL_STATIC_DRAW_ARB` wird nicht geändert
      - `GL_DYNAMIC_DRAW_ARB` wird evtl. geändert
      - `GL_STREAM_DRAW_ARB` wird jedes Frame geändert

OpenGL VBOs

- VBOs zu zeichnen funktionert (fast) genauso wie bei Vertex Arrays:
  - Laden des jeweiligen VBOs mit `glBindBufferARB`
  - Aktivieren des `GL_VERTEX_ARRAY` Client states
  - Laden des Arrays aus dem VBO mit `glVertexPointer` (allerdings mit leerem (=0) Pointer!)
  - Zeichnen mit `glDrawArrays, glDrawElements`
  - Deaktivieren des VBOs mit `glBindBufferARB` mit leerem Pointer

OpenGL VBOs

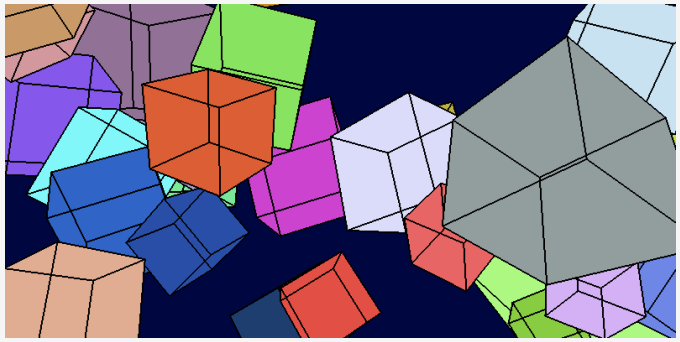
```
GLfloat vertices[] = {
    -1, -1, 1, 1, -1, 1,
    1, 1, 1, -1, 1, 1,
    -1, -1, -1, -1, -1, 1,
    -1, 1, 1, -1, 1, -1,
    1, -1, -1, 1, -1, 1,
    1, 1, 1, 1, 1, -1,
    -1, -1, -1, 1, -1, -1,
    1, 1, -1, -1, 1, -1,
    -1, -1, -1, 1, -1, -1,
    1, -1, 1, -1, -1, 1,
    -1, 1, -1, 1, 1, -1,
    1, 1, 1, -1, 1, 1
};

// generate a new VBO and get the associated ID
glGenBuffersARB(1, &vboId);

// bind VBO in order to use
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vboId);

int dataSize = 12 * 6 * 4;

// upload data to VBO
glBufferDataARB(GL_ARRAY_BUFFER_ARB, dataSize, vertices, GL_STATIC_DRAW_ARB);

// it is safe to delete after copying data to VBO
delete [] vertices;
```



(Quelle: http://www.songho.ca/opengl/gl_vbo.html)

OpenGL VBOs

```
// bind VBOs for vertex array and index array
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vboId);

// do same as vertex array except pointer
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, 0);

glDrawArrays(GL_QUADS, 0, 24);

glDisableClientState(GL_VERTEX_ARRAY);

// bind with 0, so, switch back to normal pointer operation
glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);
```



(Quelle: http://www.songho.ca/opengl/gl_vbo.html)

Primitive mit GLU

GLU Primitive

- Die Hilfsbibliothek GLU bietet verschiedene vorgefertigte Primitive die auf Quadriken basieren
- Eine Quadrik ist eine Fläche deren Punkte durch eine Formel wie

$$x^2 + y^2 + z^2 = r^2$$

beschrieben werden

- Neue Quadrikenobjekte werden mit dem Befehl

```
GLUquadricObj* gluNewQuadric()
```

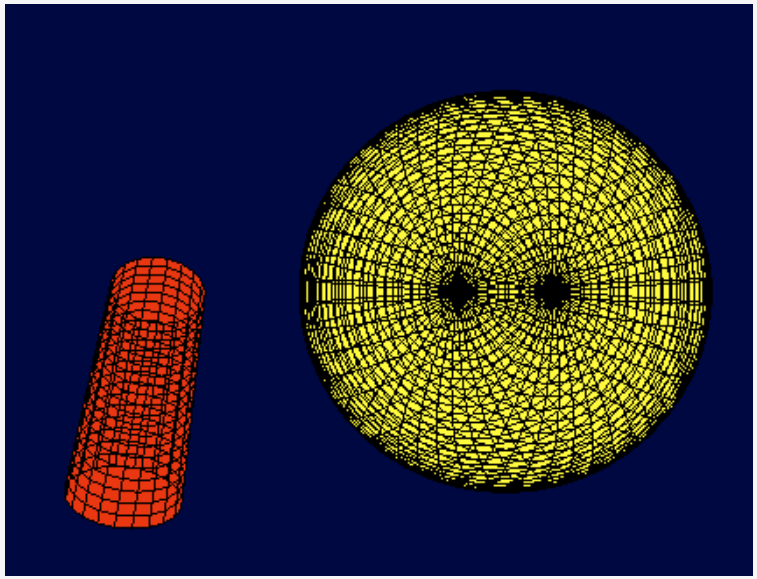
- erzeugt. GLU kümmert sich um die Details der darunterliegen Quadrik.

GLU Primitive

- Verfügbare Primitive in GLU sind (u.a.):
  - `gluCylinder (quad, base, top, height, slices, stacks)`
    zeichnet einen Zylinder entlang der Z-Achse
    - `base` Zylinderradius bei z = 0
    - `top` Zylinderradius bei z = `height`
    - `height` Höhe des Zylinders
    - `slices` Aufteilungen um die z-Achse herum
    - `stacks` Aufteilungen die z-Achse entlang
  - `gluSphere (quad, radius, slices, stacks)`
    zeichnet eine Kugel um den Ursprung
    - `radius` Radius der Kugel
    - `slices` Aufteilungen um die z-Achse herum
    - `stacks` Aufteilungen die z-Achse entlang

GLU Primitive

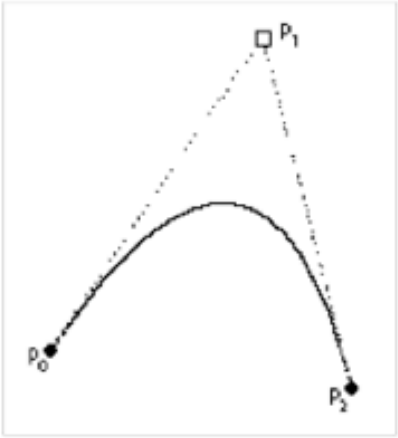
```
GLUquadricObj* quadratic=gluNewQuadric();

glPushMatrix();
    glTranslatef(3, 0, 0);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    gluSphere(quadratic, 3, 50, 50);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glColor3f(0,0,0);
    gluSphere(quadratic, 3, 50, 50);
glPopMatrix();

glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glColor3f(1.0f, 0.0f, 0.0f);
glTranslatef(-3, 0, -5);
glRotatef(45, 1, 0, 0);
gluCylinder(quadratic, 1, 1, 5, 20, 20);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glColor3f(0,0,0);
gluCylinder(quadratic, 1, 1, 5, 20, 20);
```



Kurven

Interpolation und Bezier-Kurven

- Lineare Interpolation:
  - Linie durch zwei Punkte
- Quadratische Interpolation:
  - Parabel durch drei Punkte
- Kubische Interpolation etc.
- Bézier-Kurven:
  - Pierre Bézier (1910-1999)
  - Kubische Interpolation, geeignet zur Zusammensetzung aus mehreren Kurvenstücken
  - Variante davon: Quadratische Interpolation



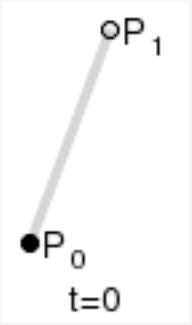
Quadratische Bezier-Interpolation:
$p_0 = (x_0, y_0)$, $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$

Für t zwischen 0 und 1 sind die Punkte auf der Linie gegeben durch:

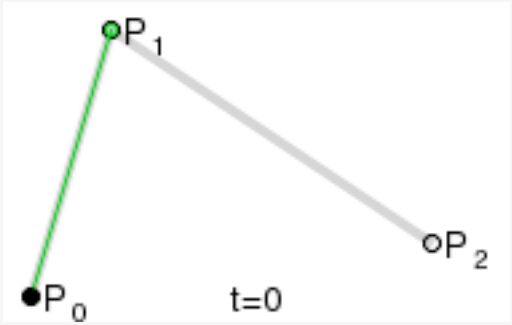
$$x_t = (1-t)^2 x_0 + 2t(1-t)x_1 + t^2 x_2$$
$$y_t = (1-t)^2 y_0 + 2t(1-t)y_1 + t^2 y_2$$

p_0 und p_2 sind Randpunkte *(on-points)*,
p_1 ist Steuerpunkt *(off-point)*

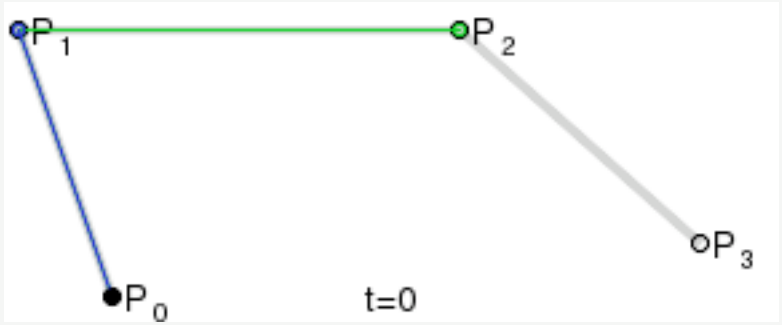
Bézierkurven



n = 1
linear



n =2
quadratisch



n = 3
kubisch

(Quelle: http://de.wikipedia.org/wiki/B%C3%A9zierkurve)

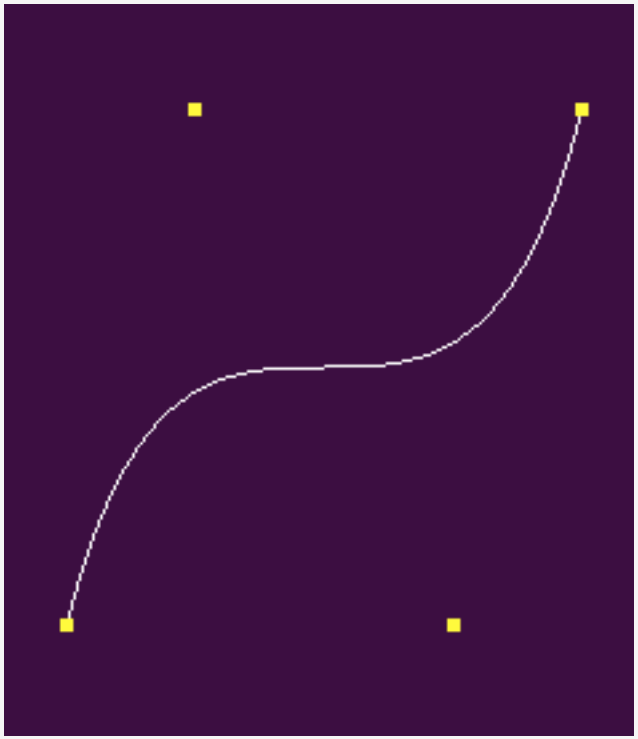
OpenGL Bézierkurven

- Die Punkte, die auf einer Bézierkurve liegen lassen sich in OpenGL automatisch in beliebiger Granularität berechnen
  - `glMap1f(target, u1, u2, stride, order, *points)`
    - `target` Werte die durch die Funktion berechnet werden sollen (z.B. `GL_MAP1_VERTEX_3, GL_MAP1_NORMAL`)
    - `u1, u2` Untere und obere Grenze (0, 1 bieten sich an)
    - `stride` Abstand in floats/doubles zwischen Kontrollpunkten im Array
    - `order` Ordnung der Kurve (= Grad + 1)
    - `points` Array mit Kontrollpunkten
  - `glEvalCoord1f (u)` berechnet einen Punkt der Kurve
    - `u` Die Position auf der Kurve (liegt zwischen u1 und u2)

OpenGL Bézierkurven

```
GLfloat ctrlpoints[4][3] = {
        { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
        {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}
    };

void bezGlTest::initializeGL(void)
{
   glClearColor(0.2, 0.0, 0.2, 1.0);
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
   glEnable(GL_MAP1_VERTEX_3);
}
```

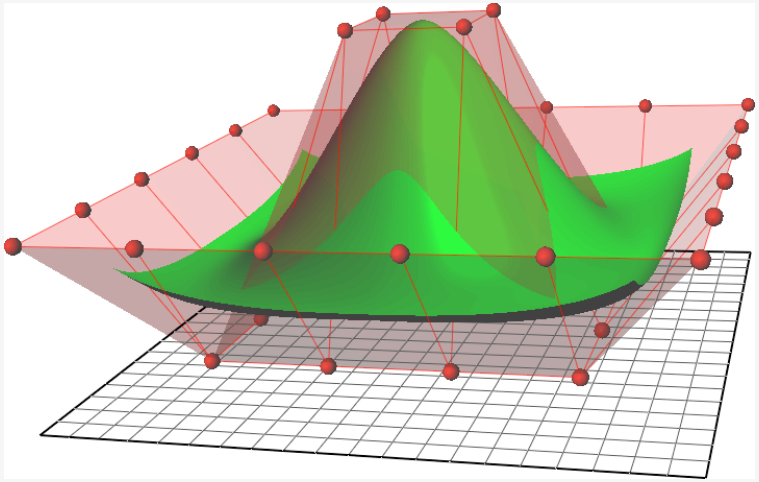
```
glColor3f(1.0, 1.0, 1.0);
glBegin(GL_LINE_STRIP);
    for (int i = 0; i <= 30; i++)
        glEvalCoord1f((GLfloat) i/30.0);
glEnd();

glPointSize(5.0);
glColor3f(1.0, 1.0, 0.0);
glBegin(GL_POINTS);
    for (int i = 0; i < 4; i++)
        glVertex3fv(&ctrlpoints[i][0]);
glEnd();
```



(Quelle: http://www.glprogramming.com/red/chapter12.html)

NURBS

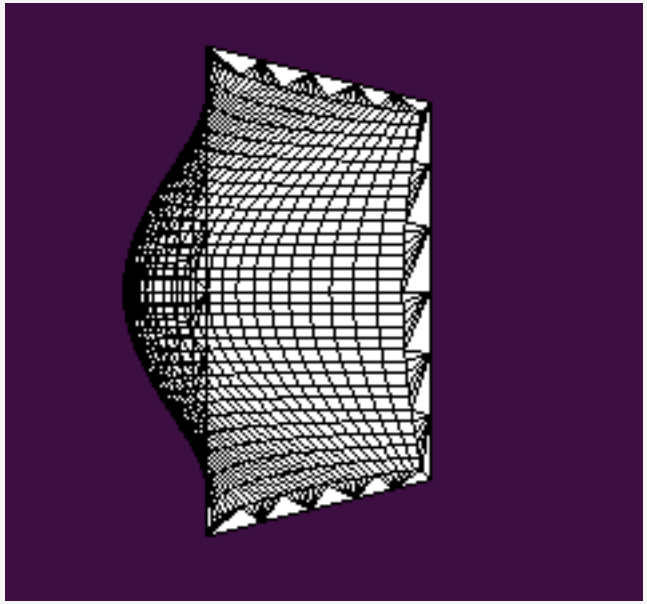
- Non-uniform Rational B-Splines sind die Verallgemeinerung von Bézierkurven
- Dienen der Beschreibung von beliebig geformten Flächen im Raum
- Definition über Kontrollpunkte und Knoten
- Der Knotenvektor definiert den Einfluss der einzelnen Kontrollpunkte auf die jeweiligen Punkte der Kurve
- Es müssen immer Kontrollpunkte + Grad der Kurve + 1 Knoten angegeben werden



(Quelle: http://en.wikipedia.org/wiki/NURBS)

OpenGL NURBS

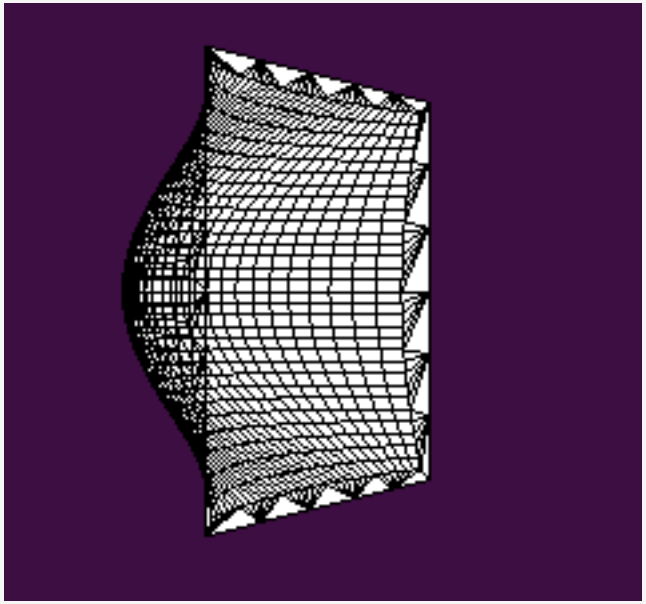
```
GLfloat nurbsctlpoints[4][4][3];
GLUnurbsObj *theNurb;

void bezGlTest::initializeGL(void)
{
   glClearColor(0.2, 0.0, 0.2, 1.0);

   int u, v;
   for (u = 0; u < 4; u++) {
       for (v = 0; v < 4; v++) {
            nurbsctlpoints[u][v][0] = 2.0*((GLfloat)u - 1.5);
            nurbsctlpoints[u][v][1] = 2.0*((GLfloat)v - 1.5);

        if ( (u == 1 || u == 2) && (v == 1 || v == 2))
            nurbsctlpoints[u][v][2] = 3.0;
        else
            nurbsctlpoints[u][v][2] = -3.0;
        }
    }

    theNurb = gluNewNurbsRenderer();
    gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
    gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
```



(Quelle: http://www.glprogramming.com/red/chapter12.html)

OpenGL NURBS

```
GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};

gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
glColor3f(1,1,1);

gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb,
        8, knots,
        8, knots,
        4 * 3,
        3,
        &nurbsctlpoints[0][0][0],
        4, 4,
        GL_MAP2_VERTEX_3);
gluEndSurface(theNurb);

gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_OUTLINE_POLYGON);
glColor3f(0,0,0);

gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb,
        8, knots,
        8, knots,
        4 * 3,
        3,
        &nurbsctlpoints[0][0][0],
        4, 4,
        GL_MAP2_VERTEX_3);
gluEndSurface(theNurb);
```



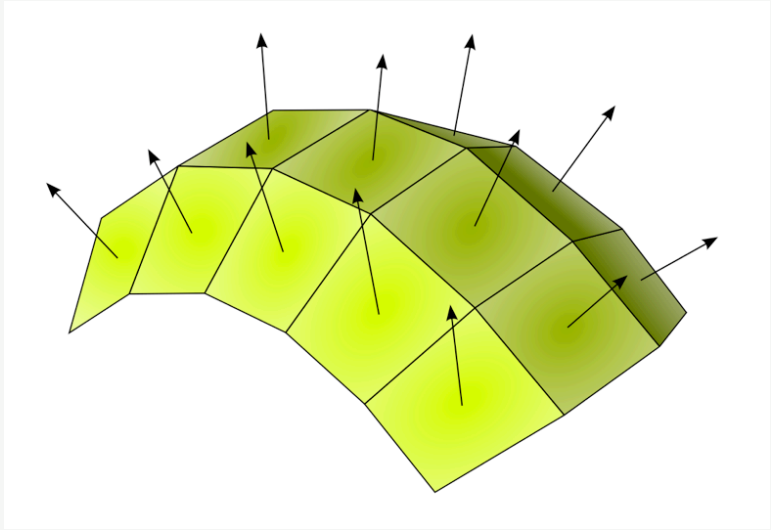
(Quelle: http://www.glprogramming.com/red/chapter12.html)

Mathe

Normalen

- Die Normale einer Ebene bzw. eines Vektorpaars ist derjenige Vektor der darauf senkrecht steht (und ungleich 0 ist)
- Bestimmung entweder über lineares Gleichungssystem oder Kreuzprodukt:

$$\vec{u} \times \vec{v} = \begin{pmatrix} u_2 \cdot v_3 - u_3 \cdot v_2 \\ u_3 \cdot v_1 - u_1 \cdot v_3 \\ u_1 \cdot v_2 - u_2 \cdot v_1 \end{pmatrix}$$



(Quelle: http://de.wikipedia.org/wiki/Normale)

Winkel zwischen zwei Vektoren

- Das Skalarprodukt zweier Vektoren erzeugt einen Skalar als Ergebnis und kann folgendermaßen berechnet werden:

$$\vec{x} \cdot \vec{y} := \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

- Dadurch lassen sich u.a. die Länge (der Betrag) eines Vektors bestimmen:

$$|\vec{x}| = \sqrt{\vec{x} \cdot \vec{x}} = \sqrt{{x_1}^2 + {x_2}^2 + \ldots + {x_n}^2}.$$

- Oder der Winkel zwischen zwei Vektoren:

$$\measuredangle(\vec{x}, \vec{y}) = \arccos \frac{\vec{x} \cdot \vec{y}}{|\vec{x}|\ |\vec{y}|}$$

(Quelle: http://de.wikipedia.org/wiki/Skalarprodukt)

Weiterführende Literatur

- http://www.opengl.org/sdk/docs/man/
- James Van Verth, Lars Bishop: Essential Mathematics for Games and Interactive Applications: A Programmer's Guide
- OpenGL ‘Redbook’: http://fly.srk.fer.hr/~unreal/theredbook/
- NeHe OpenGL Tutorials: http://nehe.gamedev.net/
- Song Ho Ahn Tutorials: http://www.songho.ca/opengl/