KULeuven
Campus De Nayer
Industrieel ingenieur
Opleiding Industriële Ingenieurswetenschappen
afstudeerrichting elektronica-ICT
Bachelor 3e fase en schakelprogramma

C O M P U T E R G R A P H I C S

met OpenGL

Academiejaar 2018-19 H. Crauwels

Inhoudsopgave

1 Inleiding

1.1 OpenGL: introductie

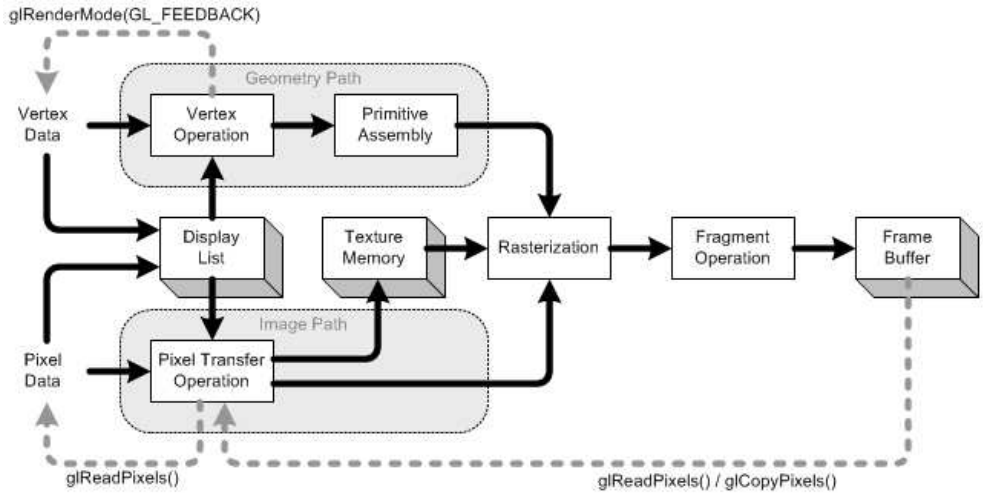
OpenGL bestaat uit drie bibliotheken:

- GL: bibliotheek met low-level grafische functies; deze verzameling van functies laten toe om 2D- en 3D-geometrie te specificeren met behulp van een beperkt aantal primitieven en deze geometrie op het scherm te tonen (rendering);
- GLU: (utility library) combineren van low-level functies voor het opzetten van transformaties voor specifieke projecties; genereren en renderen van lijnen en oppervlakken;
- GLUT: (utility toolkit) *scherm- en venster-systeem onafhankelijke* bibliotheek waarmee allerlei complexe problemen van verschillende windows-API's onzichtbaar gemaakt worden voor de programmeur (o.a. ook genereren van lijnen en oppervlakken).

Voor een correcte prototyping van alle OpenGL-functies en een definitie van alle OpenGL-constanten moet het *headerbestand* GL/glut.h ge-include-d worden (daarin zit een include van GL/gl.h en GL/glu.h).

1.2 OpenGL rendering pijplijn

De sequentie van bewerkingen die moet uitgevoerd worden om een tekening op het scherm te tonen, wordt de OpenGL rendering pijplijn genoemd. Geometrie opgebouwd uit grafische primitieven volgt een pad langs evaluators en per-vertex operaties, terwijl bitmaps (pixel-data) apart verwerkt worden – althans in de beginfase van de pijplijn.
Doorheen de verschillende bewerkingsstappen van de OpenGL pijplijn worden twee grafische informatieelementen verwerkt: vertex gebaseerde data en pixel gebaseerde data. Deze worden gecombineerd en dan in de framebuffer geschreven. Verwerkte data kan door OpenGL terug naar de applicatie gestuurd worden (grijze stippellijnen in figuur 1.1).



Figuur 1.1: OpenGL pijplijn

Vertex operaties. De coördinaten van elke vertex en normaalvector worden getransformeerd door de `GL_MODELVIEW` matrix. Bij ingeschakelde belichting worden belichtingsberekeningen per vertex uitgevoerd, resulterend in een nieuwe kleur voor de vertex.

Primitieven assemblage. De primitieven (punten, lijnen en polygonen) worden door de projectie matrix getransformeerd; het resultaat wordt ge-clip-ped en de perspectief-deling (door

w) wordt uitgevoerd. Tenslotte zorgt de viewport transformatie voor een mapping van het 3D tafereel op de venstercoördinaten.

Pixel transfer operaties. Een aantal operaties worden uitgevoerd op de pixels uit het client geheugen. De resulterende data wordt in het textuur geheugen opgeslagen of onmiddellijk gerasterd naar fragmenten.

Textuuur geheugen. Textuurbeelden worden in dit geheugen ingeladen om afgebeeld te worden op geometrische objecten.

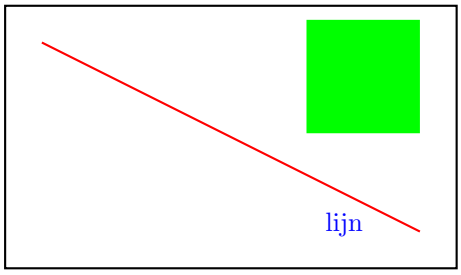
Rasterisatie is de omvorming van geometrische en pixel data naar fragmenten. Een fragment is een rechthoekige array met daarin o.a. kleur, diepte, lijnbreedte, puntgrootte en correspondeert met een pixel in de framebuffer.

Fragment operaties vormen fragmenten om naar pixels in de framebuffer. De eerste stap is texel generatie waarbij een textuur element uit het textuur geheugen gegenereerd wordt en toegepast wordt op het fragment. Dan worden *fog* berekeningen uitgevoerd. En dan worden een aantal testen uitgevoerd. Ten slotte wordt o.a. blending en maskering uitgevoerd. De resulterende pixel data worden in de framebuffer gestockeerd.

Display list : een groep van OpenGL commands die gecompileerd en gestockeerd voor latere uitvoering. Dit kan de performantie verbeteren.

1.3 Eenvoudig voorbeeld in 2D

Het volgende programma tekent een rechthoek, een lijn en een aantal letters. Hiervoor zijn een aantal functieoproepen nodig voor de initialisatie. Ook moet een transformatie van wereldcoördinaten naar schermcoördinaten vastgelegd worden.




```
#include <stdio.h>
#include <GL/glut.h>

char ratio = 'n';  /* aangepaste viewport bij arg == 'k' */
GLdouble xmin = 0.0, xmax = 200.0, ymin = 0.0, ymax = 150.0;
GLdouble verhouding;

void init(void)
{
    glClearColor(1.0, 1.0, 1.0, 1.0);
    verhouding = xmax/ymax;
}

void lijnSegment(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_LINES);
        glVertex2i(180,15);
        glVertex2i(10,145);
```

```
        glEnd();
24      glColor3f(0.0, 1.0, 0.0);
        glRecti(115,90,165,140);
26      glColor3f(0.0, 0.0, 1.0);
        glRasterPos2i(150,20);
28      glutBitmapCharacter(GLUT_BITMAP_9_BY_15, 'l');
        glutBitmapCharacter(GLUT_BITMAP_8_BY_13, 'i');
30      glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_10, 'j');
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_10, 'n');
32      glFlush();
    }
34
    void herschaal(GLint n_w, GLint n_h)
36  {
        glMatrixMode(GL_PROJECTION);
38      glLoadIdentity();
        gluOrtho2D(xmin, xmax, ymin, ymax);
40      if ( ratio == 'k' )
        {
42          if ( n_w <= n_h * verhouding )
                glViewport(0, 0, n_w, n_w/verhouding);
44          else
                glViewport(0, 0, verhouding*n_h, n_h);
46      }
        else
48          glViewport(0, 0, n_w, n_h);
        fprintf(stderr, "nieuw schaal %d %d\n", n_w, n_h);
50  }

52  int main( int argc, char * argv[])
    {
54      fprintf(stderr, "Gebruik: lijn n/k (ratio lengte/breedte)\n");
        if ( argc > 1 )
56          ratio = argv[1][0];
        glutInit(&argc, argv);
58      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
        glutInitWindowPosition(50, 100);
60      glutInitWindowSize(400, 300);
        glutCreateWindow("een lijnsegment");
62      init();
        glutDisplayFunc(lijnSegment);
64      glutReshapeFunc(herschaal);
        glutMainLoop();
66      return 0;
    }
```

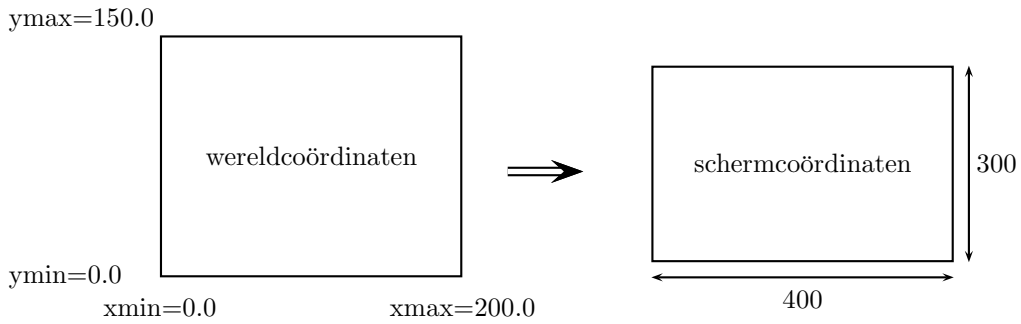
De main functie bevat een aantal functie-oproepen waarmee het systeem geïnitialiseerd wordt:

- glutInit(&argc,argv): initialisatie van de GLUT-library; deze moet uitgevoerd worden voor elke andere glut functie;
- glutInitDisplayMode(...): specificatie van de display mode voor een window; het argument is een bit-OR van GLUT-parameters:
  – GLUT_RGB: definitie van kleuren: op basis van rood-groen-blauw waarden of met bijko-

mende α-waarde om de doorzichtigheid aan te geven (GLUT_RGBA): van totale transparantie ($\alpha = 0.0$) tot volledige ondoorzichtbaarheid ($\alpha = 1.0$);
  - frame-buffering: GLUT_SINGLE maakt één enkele frame buffer; bij animaties is het aan te raden om met dubbele buffering te werken (GLUT_DOUBLE): terwijl de front-buffer stabiel blijft en getoond wordt op het scherm, schrijft het programma in de back-buffer het volgende tafereel in de animatie; pas wanneer de back-buffer volledig opgevuld is, wordt deze getoond op het scherm (front-back buffer swapping); de front-buffer wordt nu back-buffer waarin het volgende tafereel opgebouwd wordt;
  - bij 3D: GLUT_DEPTH: bij elke pixel in het frame wordt een diepte bijgehouden: objecten die zich veraf bevinden, worden bedekt door objecten die zich dichterbij bevinden.
- glutInitWindowSize(breedte, hoogte) : vastleggen van de grootte (in pixels) van het window waarin de tekening gemaakt wordt, op het scherm
- glutInitWindowPosition(x, y) : (x, y) geven de schermcoördinaten (in pixels) van de linkerbovenhoek van het venster t.o.v. het scherm;
- glutCreateWindow(titel): maken van een window met bovenaan het argument als titel; maar dit window wordt nog niet getoond; dit gebeurt pas bij het starten van `Main Loop`.

In de init functie worden een aantal specifieke initialisaties gedaan. In dit eenvoudige voorbeeld wordt de kleur vastgelegd waarmee het window ge-clear-ed wordt, normaal bij het starten van het tekenen. Deze kleur bepaalt de initiële kleur in de frame-buffer en komt dus overeen met de achtergrondkleur.
Dan worden een aantal *callback-functies* ingesteld. Een callback-functie is een functie die een pointer naar een functie (de naam van een functie zonder haakjes) als argument heeft. Deze functie zal uitgevoerd worden telkens een specifiek *event* zich voordoet. De meest essentiële callback functies zijn de display- en reshape-functie. In volgende hoofdstukken komen nog andere call-back functies aan bod.
De glutDisplayFunc stelt de *teken* functie in, in het voorbeeld lijnSegment. Deze functie heeft geen argementen en bevat alle functies die moeten uitgevoerd worden om de volledige tekening op te bouwen. Deze functie wordt uitgevoerd telkens het systeem het nodig vindt om het *teken*venster te hertekenen. Dit kan bijvoorbeeld het geval zijn wanneer een gedeeltelijk overlappend venster verschoven wordt, zodat het *teken* venster volledig zichtbaar wordt. Een hertekening kan geforceerd worden door het oproepen van glutPostRedisplay(). Ook bij het begin van de *Main Loop* wordt deze functie uitgevoerd zodat voor de eerste maal de volledige tekening opgebouwd wordt.
De glutReshapeFunc stelt de functie in die opgeroepen wordt telkens wanneer het tekenvenster van grootte of positie verandert, in het voorbeeld herschaal. Deze functie heeft twee argumenten, de breedte en hoogte van het gewijzigde tekenvenster.
Tot slot wordt de functie glutMainLoop() opgeroepen. Alle windows die tijdens initialisatie gedefinieerd zijn, worden daadwerkelijk getoond op het scherm door de display-callback functie te triggeren. Daarna wordt een *oneindige lus* gestart (`Main Loop`, waarbij bij elke event één of meerdere callback-functies opgeroepen worden.

Functie herschaal definieert de transformatie van het wereldcoördinatensysteem naar het schermcoördinatensysteem.



In een volgend hoofdstuk wordt uitgelegd dat deze transformatie uitgevoerd wordt door middel van matrixvermenigvuldigingen. De functie glMatrixMode geeft aan dat deze transformatie in de GL_PROJECTION matrix moet verrekend worden. Deze matrix wordt als eenheidsmatrix geïnitialiseerd met glLoadIdentity().
Met gluOrtho2D wordt het wereldcoördinatensysteem in twee dimensties vastgelegd, door minimale en maximale $x-$ en $y-$ waarden in te stellen.
De pixelrechthoek op het scherm waarin de uiteindelijke tekening terecht komt, wordt gedefinieerd op basis van de linkeronderhoek en de breedte en hoogte (in pixels) met behulp van de functie glViewport. Bij aanvang hebben breedte en hoogte de waardes gelijk aan de argumenten van glutInitWindowSize. Wanneer het tekenvenster herschaald wordt (bijv. muisoperaties), worden de nieuwe breedte en hoogte aan de callback functie herschaal meegegeven als argumenten. Op basis hiervan kunnen dan gepaste argumenten berekend worden voor de glViewport functie. Wanneer het volledige venster mag gebruikt worden voor de tekening, zijn deze argumenten gelijk aan de nieuwe breedte en hoogte. Hierdoor kan wel de *vormfactor* wijzigen: de helling van de lijn is gewijzigd en het vierkant wordt als rechthoek getekend. Wanneer dit niet gewenst is, moet een viewport gedefinieerd worden met eenzelfde breedte/hoogte verhouding als het oorspronkelijke venster, d.i. (4:3).

Functie lijnSegment bevat alle code om de tekening op te bouwen. Eerst wordt de frame buffer geïnitialiseerd met de *clear* kleur, wat de achtergrondkleur zal zijn. Bij het tekenen van bepaalde onderdelen kunnen modeltransformaties geactiveerd worden. De achterliggende berekeningen hiervoor gebeuren weer door middel van matrixvermenigvuldigingen. De functie glMatrixMode geeft aan dat de eventuele volgende transformaties in de GL_MODELVIEW matrix moeten verrekend worden. Deze matrix wordt als eenheidsmatrix geïnitialiseerd.
Met glColor3f kan een actuele kleur ingesteld worden. De drie argumenten zijn een hoeveelheid rood, groen en blauw.
De functie tekent eerst een lijn (in het rood) tussen de wereldco's (180,15) en (10,145). Dan wordt een rechthoek, eigenlijk een vierkant, (in het groen) met linkeronderhoek in (115,90) en rechterbovenhoek in (165,140) getekend. Tot slot worden vier letters *getekend*, elk in een specifieke font en grootte. De beginpositie van dit woord wordt aangegeven met glRasterPos2i.
OpenGL acties worden deels in de CPU, deels in de GPU uitgevoerd. Om er zeker van te zijn dat de acties van alle opgeroepen functies effectief gestart worden, kan glFlush opgeroepen worden. Dit garandeert dat alle acties in eindige tijd uitgevoerd zijn. Soms is dit niet voldoende en kan beter glFinish opgeroepen worden; deze functie wacht tot alle voorgaande acties gerealiseerd zijn.

1.4 Skeleton

Het programma kan als skeleton gebruikt worden voor OpenGLprogramma:
– een aantal globale variabelen
– een init functie met eenmalige initialisaties;
– een *teken* callback functie waarin de volledige scène opgebouwd wordt, eventueel door het oproepen van andere functies;
– een *herschaal* callback functie waarin de wereld- naar schermcoördinatensysteem transformatie gedefinieerd wordt;
– de main functie die, op enkele details na, telkens integraal kan overgenomen worden.
Bijvoorbeeld, om een grafiek van de functie $\frac{\sin(\pi x)}{\pi x}$ te maken in het interval $[-4.0, 4.0]$, moet het wereldcoördinatensysteem aangepast worden:

```
GLdouble xmin = -5.0, xmax = 5.0, ymin = -0.3, ymax = 1.2;
```

en moet er een andere teken callback functie opgenomen worden:

```
1  void gebogen(void)
   {
3      GLfloat x;
```

```
5       glClear(GL_COLOR_BUFFER_BIT);
        glMatrixMode(GL_MODELVIEW);
7       glLoadIdentity();
        if ( assen )
9       {
            glColor3f(0.0, 1.0, 0.0);
11          glBegin(GL_LINES);
                glVertex2f( xmin+0.6, 0.0 );
13              glVertex2f( xmax-0.6, 0.0 );
                glVertex2f( 0.0, ymin+0.1);
15              glVertex2f( 0.0, ymax-0.1);
            glEnd();
17      }
        glColor3f(0.0, 0.0, 1.0);
19      glLineWidth(2.0);
        glPointSize(3.0);
21      glBegin(GL_LINE_STRIP);
            for ( x = -4.0; x < 4.0; x += 0.1 )
23          {
                glVertex2f( x, sin(M_PI*x)/(M_PI*x) );
25          }
        glEnd();
27      glFlush();
    }
```

De hoofdassen worden getekend door de vertices van lijnen 12-15 per twee met elkaar te verbinden (GL_LINES). De vertices die in lijn 24 geemaakt worden, worden door stukjes lijn (GL_LINE_STRIP) met elkaar verbonden wat resulteert in het functieverloop.

2 Geometrische primitieven

2.1 Data types

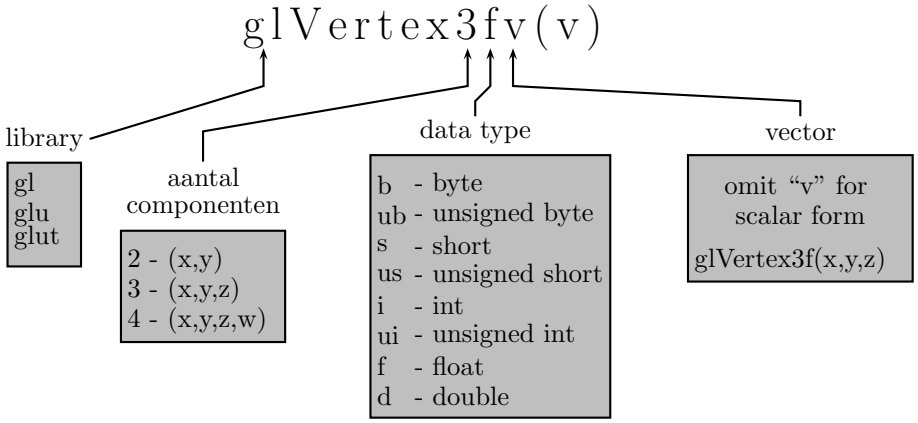
OpenGL is een platform-onafhankelijke API. Dit betekent dat een OpenGL programma in om het even welk besturingssysteem kan gecompileerd, gelinkt en uitgevoerd worden zonder de code aan te passen. Omdat elk systeem zijn eigen woordlengte heeft, bijvoorbeeld 16, 32 of 64 bit, moet bij het definiëren (declareren) van variabelen en functies wel opgepast worden. Een variabele van type **int** neemt in elk systeem niet evenveel plaats in het geheugen in.
Om abstractie te maken van de specifieke datatypering per systeem, zijn in het include bestand `gl.h` een aantal datatypes gedefinieerd in de vorm GLxxx. Deze definities kunnen voor elk platform anders worden ingevuld. De ontwerper van een OpenGL programma kan best gebruik maken van deze *GL*types zodat de overdraagbaarheid geen probleem vormt.
Een voorbeeld van een reeks definities voor een specifiek platform:

```
typedef unsigned int GLenum;
typedef unsigned char GLboolean;
typedef unsigned int GLbitfield;
typedef void GLvoid;
typedef signed char GLbyte;          /* 1-byte signed */
typedef short GLshort;               /* 2-byte signed */
typedef int GLint;                   /* 4-byte signed */
typedef int GLsizei;                 /* 4-byte signed */
typedef unsigned char GLubyte;       /* 1 byte unsinged */
typedef unsigned short GLushort;     /* 2 byte unsinged */
typedef unsigned int GLuint;         /* 4 byte unsinged */
typedef float GLfloat;               /* enkele precisie */
typedef float GLclampf;              /* enkele precisie in [0,1] */
typedef double GLdouble;             /* dubbele precisie */
typedef double GLclampd;             /* dubbele precisie in [0,1] */
```

Een aantal variabelen (bijv. een kleurcomponent) kan slechts waarden in het interval $[0.0, 1.0]$ hebben, van daar het type `GLclamp*`. Wanneer bij een berekening zo'n variabele een waarde krijgt buiten het interval, dan wordt deze waarde *geclamped* (kleiner dan 0.0 wordt 0.0, groter dan 1.0 wordt 1.0).

2.2 Naamgeving

De naamgeving van OpenGL functies voldoet aan een aantal regels. Een functienaam begint met een verwijzing naar de bibliotheek (gl, glu, glut). Dan volgt de naam van de specifieke functie (Vertex). Het laatste deel geeft het aantal argumenten, het type van deze argumenten en eventueel een indicatie dat de argumenten in de vorm van een vector gegeven worden.



2.3 OpenGL state

OpenGL is steeds in een bepaald toestand die bepaalt hoe de tekening er uit ziet. Het systeem wordt in een bepaalde toestand gebracht door het selecteren van *states* (of *modes*). Deze blijven in voege tot ze weer gewijzigd worden. Bijvoorbeeld, de kleur waarmee getekend wordt, wordt bepaald door de toestand GL_CURRENT_COLOR. Deze toestand kan gewijzigd worden met de glColor functie. De actuele waarde van een toestandsvariabele opvragen kan met een glGet∗ functie. Bijvoorbeeld om de actuele kleur op te vragen:

```
GLfloat kleur[4];
glGetFloatv(GL_CURRENT_COLOR, kleur);
```

Dit zou bijvoorbeeld nuttig kunnen zijn wanneer de actuele kleur even moet vervangen worden door een andere kleur voor het tekenen van een specifiek detail, en dat dan terug met de originele kleur verder getekend moet worden. OpenGL biedt hiervoor echter een *attribuut stack* aan:

```
glPushAttrib(GL_CURRENT_BIT);    // bewaar originele kleur
glColor3f(0.4, 0.8, 0.3);
//  tekenfuncties voor specfiek detail
glPopAttrib();                    // herstel originele kleur
```

Andere voorbeelden van toestandsvariabelen zijn puntdikte, lijndikte, lijntype, lichtkarakteristieken, materiaaleigenschappen, modeltransformaties, ...
Daarnaast zijn er heel wat eigenschappen die kunnen in- en uitgeschakeld worden m.b.v glEnable en glDisable functies. De meeste eigenschappen zijn bij default uitgeschakeld, omdat deze extra rekenwerk vragen bij het renderen van een tafereel. Testen of een bepaalde eigenschap ingeschakeld is, kan met de glIsEnabled functie, die naar gelang van het geval GL_TRUE of GL_FALSE teruggeeft.
Voobeelden van mogelijke eigenschappen die kunnen ingeschakeld worden zijn stippelijnmodus, lichtberekening, textuurmapping, fogberekening, ...

2.4 Geometrische primitieven

Een geometrische object in een tafereel wordt bepaald door een geordende set *vertices*. Een *vertex* kan beschouwd worden als de de coördinaat van een punt in een 2D- of een 3D-ruimte. In OpenGL wordt gewerkt met homogene coördinaten: een punt wordt dus aangegeven door 4 waarden (x, y, z, w). De OpenGL functie om een vertex te definiëren is glVertex∗. Van deze functie zijn er verschillende verschijningsvormen voor gebruik in 2D of in 3D en naargelang van het type van de argumenten (eventueel een vector):

```
GLdouble punt[4] = {4.0, 2.0, 6.0, 2.0};
glVertex2s(2,3);
glVertex3f(1.0, 2.0,3.0);
glVertex4dv(punt);
```

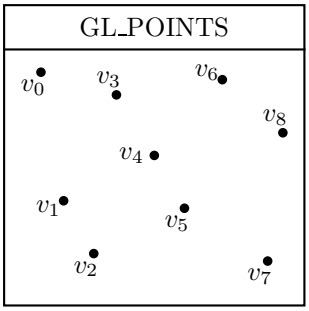
Wanneer er geen 4 coördinaten meegegeven worden, wordt $z = 0$ en/of $w = 1$ gesteld.

2.4.1 Basisvormen

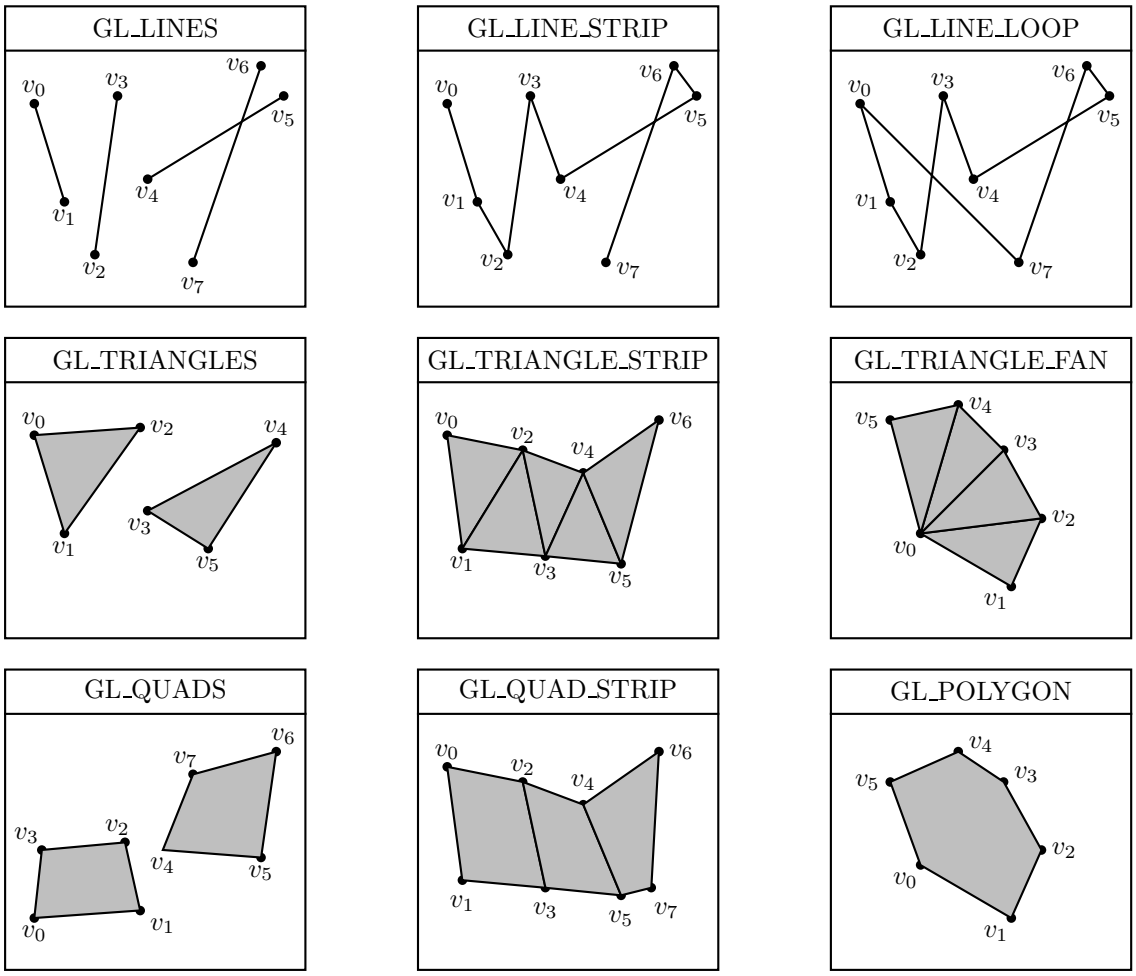
Een glVertex functie kan enkel gebruikt worden binnen een glBegin() - glEnd() sequentie om een basisvorm te definiëren. Het argument van glBegin geeft aan welke basisvorm getekend wordt.

Het argument GL_POINTS geeft aan dat een aantal punten in de tekening opgenomen wordt. Binnen de glBegin() - glEnd() sequentie worden een aantal vertices gedefinieerd:

```
glBegin(GL_POINTS);
    glVertex2i(5,31)
    glVertex2i(37,23)
glEnd();
```

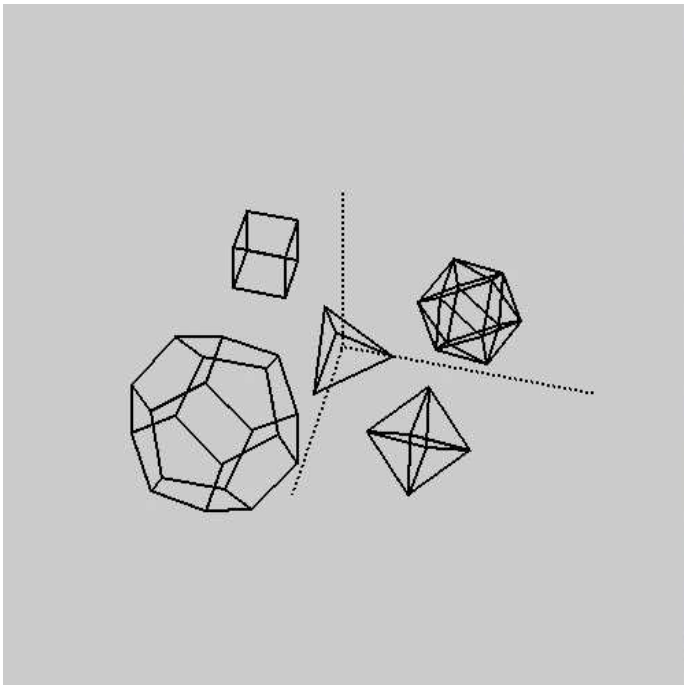


De andere mogelijkheden worden in de figuur aangegeven. De nummering bij de v annotaties geeft de volgorde aan waarin de vertices binnen de glBegin() - glEnd() sequentie opgenomen worden.



2.4.2 Polyhedra van Plato

Een regelmatige polyhedron bevat een aantal gelijke vlakken die elk begrensd worden door een regelmatige veelhoek: alle zijden zijn aan elkaar gelijk, dat geldt ook voor alle hoeken tussen zijden en voor alle hoeken tussen vlakken.
In *Timaeus* ($\pm$ 350 vC.) beschrijft Plato vijf regelmatige polyhedra, zie figuur 2.1.
In GLUT zijn tien functies voorzien om deze polyhedra te genereren, zowel in wire-frame versie als lichamen met opgevulde oppervlakken. Het centrum van zo'n polyhedron is de oorsprong van het assenstelsel. De naamgeving is gebaseerd op het aantal vlakken dat een polyhedron heeft:



Figuur 2.1: Polyhedra van Plato

- tetrahedron: met vier vlakken (een pyramide met driehoekig grondvlak)

```
glutWireTetrahedron();
glutSolidTetrahedron();
```

  de afstand van het centrum tot een hoekpunt is gelijk aan $\sqrt{3}$;
- hexahedron: met zes vierkantige vlakken (een kubus)

```
glutWireCube(lengte);
glutSolidCube(lengte);
```

  De zijden van de kubus zijn gelijk aan de waarde van het argument (`lengte`);
- octahedron: met acht driehoekige vlakken

```
glutWireOctahedron();
glutSolidOctahedron();
```

  de afstand van het centrum tot een hoekpunt is gelijk aan 1;
- dodecahedron: met twaalf vlakken met elk vijf hoeken (pentagon)

```
glutWireDodecahedron();
glutSolidDodecahedron();
```

  de afstand van het centrum tot een hoekpunt is gelijk aan 1;
- icosahedron: met twintig driehoekige vlakken

```
glutWireIcosahedron();
glutSolidIcosahedron();
```

  de afstand van het centrum tot een hoekpunt is gelijk aan 1.

2.4.3 Quadrieken

Een kwadratisch oppervlak (kwadriek) is van de vorm

$$q(x,y,z) = a_{11}x^2 + 2a_{12}xy + a_{22}y^2 + a_{33}z^2 + 2a_{23}yz + 2a_{13}xz + b_1x + b_2y + b_3z + c = 0$$

Een voorbeeld is een bol: $x^2 + y^2 + z^2 = r^2$
Zo'n gebogen oppervlak wordt in OpenGL benaderd door een heleboel kleine drie- of viervlakjes. Hoe meer vlakjes gebruikt worden, hoe beter de benadering maar ook hoe langer de rekentijd. De ontwerper kan bijvoorbeeld bij een bol aangeven hoeveel vlakjes er moeten gebruikt worden in de lengte en in de breedte.
De GLUT bibliotheek bevat functies voor het genereren van een aantal volumes met een quadriek oppervlak.

- bol met een bepaalde straal

```
glutWireSphere(straal, lengtestukken, breedtestukken);
glutSolidSphere(straal, lengtestukken, breedtestukken);
```

- kegel met een bepaalde straal en hoogte

```
glutWireCone(grondstraal, hoogte, lengtestukken, breedtestukken);
glutSolidCone(grondstraal, hoogte, lengtestukken, breedtestukken);
```

- torus met een bepaalde grootstraal en circulaire doorsnede met kleinstraal

```
glutWireTorus(kleinstraal, grootstraal, ncross, nradiaal);
glutSolidTorus(kleinstraal, grootstraal, ncross, nradiaal);
```

Om een kwadriek oppervlak te genereren met een GLU functie moet het oppervlak een naam krijgen en moeten een aantal functies opgeroepen worden:

```
GLUquadricObj bol;
bol = gluNewQuadric();
gluQuadricDrawStyle(bol, GLU_LINE);
gluSphere(bol, straal, lengtestukken, breedtestukken);
```

Met `GLU_LINE` wordt een wire frame getekend. Andere mogelijkheden zijn `GLU_POINT` (puntenplot), `GLU_SILHOUETTE` (vereenvoudigde wire frame) en `GLU_FILL` (opgevuld oppervlak).
Eventueel kan met gluQuadricNormals aangegeven worden hoe de normalen op het oppervlak moeten gegenereerd worden: GLU_NONE, GLU_FLAT of GLU_SMOOTH . Met de functie gluQuadricOrientation kan de oriëntatie aangegeven worden: GLU_OUTSIDE of GLU_INSIDE .
Functies voor andere kwadriek oppervlakken:

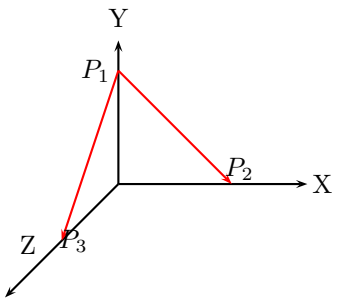
```
gluCylinder(cyl, grondstraal, topstraal, hoogte,
                        lengtestukken, breedtestukken);
gluDisk(schijf, binnenstraal, buitenstraal, radiaalstukken, ringstukken);
gluPartialDisk(stuk, binnenstraal, buitenstraal, radiaalstukken,
                        ringstukken, beginhoek, draaihoek);
```

Het grondvlak ligt in het $xy-$vlak ($z = 0$) en de as van de cylinder is volgens de $z-$as. Door een verschillende grondstraal en topstraal te nemen kan een afgeknotte kegel getekend worden.
Gealloceerd geheugen voor zo'n GLU kwadriek oppervlak kan terug vrijgegeven worden; bijvoorbeeld

```
gluDeleteQuadric(bol);
```

2.5 Clipping

Soms is slechts een gedeelte van een volledig object nodig om in de scene opgenomen worden, bijvoorbeeld een halve bol. Door een *clipping vlak* in te stellen wordt slechts het gedeelte van het object dat zich langs één kant van dit vlak bevindt, getekend.
Om zo'n clipping vlak in OpenGL te definiëren moeten de coëfficiënten van de vergelijking van dit vlak gekend zijn.
De vergelijking van een vlak is $Ax + By + Cz + D = 0$. Hierin zijn (A, B, C) de coördinaten van de normaal op het vlak. De normaal op een vlak gevormd door twee vectoren met coördinaten (a_1, a_2, a_3) en (b_1, b_2, b_3) , wordt berekend als het vectorieel product $(a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1)$ De vierde coëfficiënt D kan berekend worden door de coördinaten van een punt van het vlak in de vergelijking in te vullen
Een voorbeeld:



$P_1 = (0, 1, 0)$ $P_2 = (1, 0, 0)$ $P_3 = (0, 0, 1)$
vector $P_1P_2 = P_2 - P_1 = (1, -1, 0)$
vector $P_1P_3 = P_3 - P_1 = (0, -1, 1)$
normaal
$(-1 \times 1 - 0 \times -1, 0 \times 0 - 1 \times 1, 1 \times -1 - -1 \times 0)$
dus normaal is $(-1, -1, -1)$
P_1 invullen:
$-1 \times 0 + -1 \times 1 + -1 \times 0 + D = 0$ dus $D = 1$
De vergelijking van het vlak is dus: $-x - y - z + 1 = 0$

Merk op: verschil tussen $(-1, -1, -1, 1)$ en $(1, 1, 1, -1)$
Het definiëren van een clipping vlak:

```
GLdouble vlak[4] = {-1.0,-1.0,-1.0, 1.0 };
glClipPlane(GL_CLIP_PLANE0, vlak);
```

De elementen van de array zijn de coëfficienten A, B, C, D van de vergelijking van het vlak.
Op basis van dit clipping vlak, het tekenen van een gedeelte van een bol:

```
glEnable(GL_CLIP_PLANE0);
glutWireSphere(1.0, 18, 14);
glDisable(GL_CLIP_PLANE0);
```

Eventueel kunnen meerdere clipping vlakken geactiveerd worden. In OpenGL zijn minstens 6 clipping vlakken mogelijk:
`GL_CLIP_PLANE0` , `GL_CLIP_PLANE1` ... `GL_CLIP_PLANE5`

2.6 Invoer

2.6.1 Callback functies

Een *callback* functie is een functie die uitgevoerd wordt, telkens een specifiek *event* zich voordoet. Essentiële callbackfuncties zijn de teken- functie en de herschaal-functie. Zoals hoger aangegeven wordt zo'n callback-functie geïnstalleerd door een functie die de naam van de functie als argument heeft

- glutDisplayFunc(teken): de functie teken wordt uitgevoerd telkens het systeem het nodig vindt om het venster te hertekenen, bijv. bij het tijdelijk bedekken van het venster of geforceerd door het oproepen van glutPostRedisplay()
  functie teken: alle routines om de tekening volledig op te bouwen
- glutReshapeFunc(herschaal): telkens een window verandert van grootte of van positie op het scherm
  herschaal heeft twee argumenten: de nieuwe breedte en hoogte.

Om functionaliteit van het programma te verhogen op het vlak van invoer van de gebruiker zijn er callback-functies die reageren op invoer van het toetsenbord of invoer via de muis:

- glutKeyboardFunc(toets): telkens een *normaal* ASCII character ingetikt wordt (m.i.v. escape, spatie en delete), wordt de functie `toets` uitgevoerd; deze functie heeft drie argumenten: de ASCII code van de ingetikte toets, de x- en y-coördinaten van de muispositie (in pixels);
- glutSpecialFunc(spectoets): gelijkaardig aan de vorige, waarbij de functie `spectoets` uitgevoerd wordt wanneer een speciaal teken, bijv. een pijltje, ingetikt wordt;
- glutMouseFunc(muis): de functie `muis` wordt uitgevoerd bij het indrukken van een knop van de muis;
- glutMotionFunc(amuis): de functie `amuis` wordt uitgevoerd bij een muisbeweging met ingedrukte knop;
- glutPassiveMotionFunc(pmuis): de functie `pmuis` wordt uitgevoerd bij een muisbeweging zonder ingedrukte knop.

Een *muis* callback functie heeft 4 argumenten: welke knop, toestand van de knop, x- en y-coördinaten van de muispositie. Welke knop gebruikt wordt en in welke toestand deze is kan getest worden door de argumenten te vergelijken met volgende voorgedefinieerde constanten:

GLUT_LEFT_BUTTON GLUT_MIDDLE_BUTTON GLUT_RIGHT_BUTTON
GLUT_DOWN GLUT_UP

Voorbeeld. Mogelijke reacties op input van toetsenbord zijn in volgende functie voorzien

```
void toetsen( unsigned char key, int x, int y)
{
    switch ( key )
    {
        case 'i' : initkleur(wit,rood,groen,blauw); break;
        case 'g' : draaien = !draaien;      break;
        case 'G' : draaien = -1;            break;
        case 'q' : exit(0);                 break;
    }
    glutPostRedisplay();
}
```

Deze callback functie moet geïnstalleerd worden (in de `main` functie):

```
glutKeyboardFunc(toetsen);
```

2.6.2 Menu

De GLUT bibliotheek ondersteunt eenvoudige pop-up menu's. Zo'n menu wordt niet in de taakbalk van het actieve window ingebouwd, maar verschijnt op de plaats waar met de muis geklikt wordt. Een menu kan een submenu hebben zodat een hiërarchische structuur van menu's kan opgebouwd worden. De structuur moet wel bottom-up opgebouwd worden omdat de submenu identifier (een integer) moet doorgegeven worden aan de GLUT-functie die de parent menu creëert.
Volgende functies zijn beschikbaar:

- glutCreateMenu(menu): creatie van een menu, resultaat is een integer (*menu-identifier*); het argument is de naam van een callback functie; deze functie wordt uitgevoerd wanneer een element van een menu gekozen wordt;
- glutAddMenuEntry("groen", 2): voegt onderaan aan het actieve menu een menu-item toe; het eerste argument is de tekst die in het menu getoond wordt; het tweede argument is het geheel getal dat aan de callback functie gegeven wordt wanneer dit menu-item gekozen wordt;

- glutAddSubMenu("rood", submenu_id): voegt onderaan aan het actieve menu een submenu toe ; het argument is de identifier van een eerder gecreëerd submenu;
- glutAttachMenu(muisknop): koppelt een muisknop aan het actieve menu; wanneer met deze muisknop geklikt wordt, wordt het menu getoond (pop-up).

Voorbeeld. De creatie van een menu kan in de `init` functie gebeuren. Er is een hoofdmenu waarvan één entry gekoppeld is aan een submenu. Vermits er een hoofd- en submenu voorzien is, worden twee callback-functies voorzien.

```
void rgbmenu(int id)
{
    printf("rgbmenu %d\n", id);
    switch ( id )
    {
        // case 1 : is gerelateerd aan het submenu
        case 2 : initkleur(groen,groen,groen,groen);      break;
        case 3 : initkleur(blauw,blauw,blauw,blauw);     break;
        case 4 : exit(0);                          break;
    }
    glutPostRedisplay();
}

void lomenu(int id)
{
    printf("lomenu %d\n", id);
    switch ( id )
    {
        case 1 : initkleur(lila,lila,lila,lila);     break;
        case 2 : initkleur(oranje,oranje,oranje,oranje);      break;
    }
    glutPostRedisplay();
}

void myinit(void)
{
    GLint submenu;
    if ( leegloop == 'i' )
        glClearColor(0.9, 0.9, 0.9, 0.0);
    else
        glClearColor(0.7, 0.7, 0.7, 0.0);
    submenu = glutCreateMenu(lomenu);
        glutAddMenuEntry("lila", 1);
        glutAddMenuEntry("oranje", 2);
    glutCreateMenu(rgbmenu);
        glutAddSubMenu("rood", submenu);
        glutAddMenuEntry("groen", 2);
        glutAddMenuEntry("blauw", 3);
        glutAddMenuEntry("stoppen", 4);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
    initkleur(wit,rood,groen,blauw);
}
```

2.7 Animatie

Een tafereel wordt enkel hertekend door de teken callback functie wanneer dit nodig is, bijvoorbeeld na een herschaling of een bedekking door een ander venster. Het hertekenen kan ook geforceerd worden door de functie glutPostRedisplay op te roepen. Wanneer dit herhaaldelijk gedaan wordt, telkens nadat een aantal geometrische parameters lichtjes gewijzigd zijn, wordt de indruk gewekt van een bewegend beeld.
Om dit oproepen van glutPostRedisplay *automatisch* op geregelde tijdstippen te laten gebeuren, wordt de oproep opgenomen in een callback functie die op een van de volgende manieren geïnstalleerd wordt:

- glutIdleFunc(functie): het argument is een callback functie die uitgevoerd wordt telkens geen enkel ander event moet bediend worden in de `Main Loop`;
- glutTimerFunc(interval,tfun,argum): na een tijdspanne gelijk aan `interval` wordt de callback functie `tfun` uitgevoerd, waarbij `argum` als argument aan `tfun` meegegeven wordt; deze timer callback wordt slechts eenmaal uitgevoerd. Om de animatie op gang te houden, moet dit herhaald worden. De glutTimerFunc wordt hiervoor in `tfun` opnieuw opgeroepen.

Om de animatie vloeiend te laten verlopen wordt gebruik gemaakt van dubbele buffering, namelijk twee totaal onafhankelijke framebuffers. Het tafereel van de ene buffer wordt door de hardware op het scherm getoond, terwijl in de andere het volgende tafereel door de software opgebouwd wordt. Wanneer de opbouw in de tweede buffer klaar is, worden beide buffers gewisseld (door de functie glutSwapBuffers). Deze dubbele buffering moet bij de initialisatie aangegeven worden, door in glutInitDisplayMode GLUT_DOUBLE op te nemen (in plaats van GUT_SINGLE).

Voorbeeld. Er zijn twee mogelijke opties om het tafereel te animeren:

```
if ( leegloop == 'i' )
    glutIdleFunc(animidle);
else
{
    glutTimerFunc(tijd, anim, 1);
}
```

De implementatie van deze twee callback functies:

```
void animidle(void)
{
    if ( draaien )
        hoek += draaien * 1.0;
    glutPostRedisplay();
}

void anim(int delta)
{
    if ( draaien )
    {
        hoek += draaien * delta;
        if ( hoek > 360.0 )
            hoek -= 360.0;
        else if ( hoek < -360.0 )
            hoek += 360.0;
    }
    glutTimerFunc(tijd, anim, 1);
    glutPostRedisplay();
}
```

Het tafereel zelf bestaat uit een vierkant dat via het menu een andere kleur kan krijgen en via toetsenbord kan de animatie ingeschakeld worden.

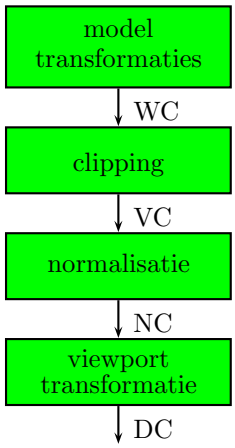
```
void vierkant(void)
{
    GLfloat x;

    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glColor3f(0.1, 0.1, 0.1);
    glBegin(GL_LINES);
        glVertex2f( -1.5, 0.0 );
        glVertex2f( 1.5, 0.0 );
        glVertex2f( 0.0, -1.5 );
        glVertex2f( 0.0, 1.5 );
    glEnd();
    x = M_PI*hoek / 180.0;
    glBegin(GL_QUADS);
        glColor3fv(kleurrechts);
        glVertex2f( cos(x), sin(x) );
        glColor3fv(kleurboven);
        glVertex2f( cos(x+M_PI/2.0), sin(x+M_PI/2.0) );
        glColor3fv(kleurlinks);
        glVertex2f( cos(x+M_PI), sin(x+M_PI) );
        glColor3fv(kleuronder);
        glVertex2f( cos(x+3.0*M_PI/2.0), sin(x+3.0*M_PI/2.0) );
    glEnd();
    glutSwapBuffers();
    glFlush();
}
```

3 2D visualisering

Een 2D of 3D tafereel kan opgebouwd worden met behulp van geometrische primitieven en 2D of 3D-modeltransformaties. De reële afmetingen van de objecten kunnen gaan van zeer klein tot zeer groot. Een polshorloge of een olietanker kunnen met dezelfde functies getekend worden.
Dit tafereel moet omgezet worden naar een afbeelding op het computerscherm (visualisering). Bij de weergave van een polshorloge in ware grootte wordt slechts een klein gedeelte van het scherm gebruikt, terwijl de tanker vele malen zal moeten verkleind worden om in zijn geheel op het scherm getoond te worden.
In dit hoofdstuk

3.1 2D-visualisering pijplijn

Een scène in 2D wordt opgebouwd in het $xy-$vlak. Alle 2D grafische primitieven en eventuele 2D modeltransformaties resulteren bijvoorbeeld in een grafiek van een functie. De coördinaten zijn eigen aan het probleem, niet aan de visualisering ervan, en worden daarom *wereldcoördinaten* genoemd. Via de volgende pijplijn worden de objecten correct op het scherm geplaatst.



Modelering van de scène in wereldcoördinaten met behulp van grafische primitieven

Aflijnen van een twee-dimensionale view: alle objecten buiten het clipping-window: niet zichtbaar

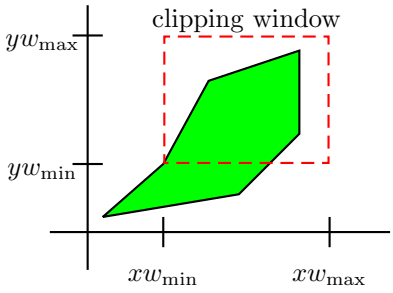
Alle coördinaten worden genormaliseerd naar bijvoorbeeld een eenheidsvierkant

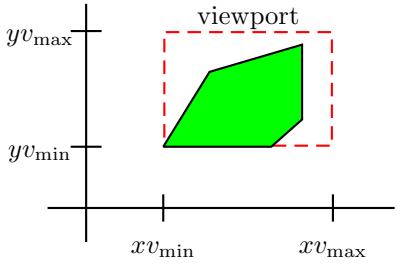
De genormaliseerde 2D-grafiek wordt gemapt op het display-window.

Clipping is een term die refereeert naar een krantenknipsel en wordt in grafische context gedefinieerd als

> The action of truncating data or an image by removing all the display elements that lie outside a boundary

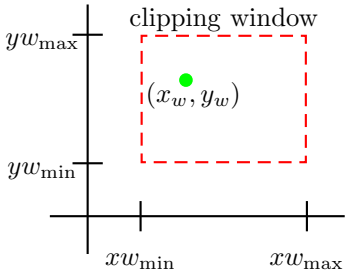
De grenzen vormen het clipping venster.

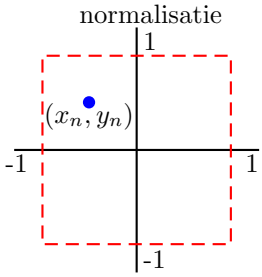




Het clipping window wordt vaak in wereldcoördinaten uitgedrukt (WC = VC).

Normalisatie wordt toegepast om het visualiseringsproces onafhankelijk te maken van de vereisten gesteld door het display device. Het clipping venster wordt geprojecteerd op een genormaliseerd vierkant. De grootte en ligging van dit vierkant is eigen aan de grafische API. OpenGL normaliseert naar het vierkant met linker onderhoek $(-1,-1)$ en rechterbovenhoek $(1,1)$. Het middelpunt is dus $(0,0)$. Het tafereel wordt door de normalisatie omgevormd naar NC (genormaliseerde coördinaten).
Omvorming van een punt met wereldcoördinaten (x_w, y_w) naar genormaliseerde coördinaten (x_n, y_n):





Berekening (x_n, y_n)

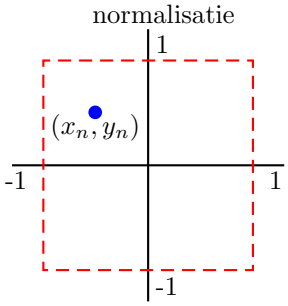
$$\frac{x_n-(-1)}{1-(-1)} = \frac{x_w-xw_{min}}{xw_{max}-xw_{min}}$$
$$\frac{y_n-(-1)}{1-(-1)} = \frac{y_w-yw_{min}}{yw_{max}-yw_{min}}$$

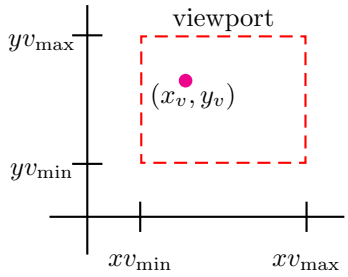
Oplossen naar x_n en y_n geeft:

$$\begin{cases} x_n = s_x \times x_w + t_x \\ y_n = s_y \times y_w + t_y \end{cases} \quad \text{met} \quad \begin{cases} s_x = \frac{2}{xw_{max}-xw_{min}} \\ s_y = \frac{2}{yw_{max}-yw_{min}} \end{cases} \quad \text{en} \quad \begin{cases} t_x = \frac{-xw_{max}-xw_{min}}{xw_{max}-xw_{min}} \\ t_y = \frac{-yw_{max}-yw_{min}}{yw_{max}-yw_{min}} \end{cases}$$

Deze transfomatie komt neer op een *schaling* (bijv. breedte $xw_{max} - xw_{min}$ terugbrengen naar een breedte gelijk aan 2) gevolgd door een *translatie* (middelpunt van clipping venster verschuiven naar $(0,0)$).

Viewport geeft in pixels aan welk gedeelte van het actieve window gebruikt wordt om het tafereel te tonen. Pixelcoördinaten $(0,0)$ verwijzen naar de linkerbenedenhoek van het actieve window. Er is een verband nodig tussen het genormaliseerd tafereel naar de plaats en grootte van een venster dat zich bevindt in het actieve window op het scherm.
Omvorming van een punt met genormaliseerde coördinaten (x_n, y_n) naar scherm- of display coördinaten (x_v, y_v):





Berekening (x_v, y_v)

$$\frac{x_v-xv_{min}}{xv_{max}-xv_{min}} = \frac{x_n-(-1)}{1-(-1)}$$
$$\frac{y_v-yv_{min}}{yv_{max}-yv_{min}} = \frac{y_n-(-1)}{1-(-1)}$$

Oplossen naar x_v en y_v geeft:

$$\begin{cases} x_v = s'_x \times x_n + t'_x \\ y_v = s'_y \times y_n + t'_y \end{cases} \quad \text{met} \quad \begin{cases} s'_x = \frac{xv_{max}-xv_{min}}{2} \\ s'_y = \frac{yv_{max}-yv_{min}}{2} \end{cases} \quad \text{en} \quad \begin{cases} t'_x = \frac{xv_{max}+xv_{min}}{2} \\ t'_y = \frac{yv_{max}+yv_{min}}{2} \end{cases}$$

Ook hier hebben we te maken met een *schaling* gevolgd door een *translatie*.

Matrix notatie van beide transformaties (WC naar NC, NC naar DC):

- wereldcoördinaten naar genormaliseerde coördinaten

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x_w \\ y_w \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

- genormaliseerde coördinaten naar schermcoördinaten

$$\begin{bmatrix} x_v \\ y_v \end{bmatrix} = \begin{bmatrix} s'_x & 0 \\ 0 & s'_y \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} + \begin{bmatrix} t'_x \\ t'_y \end{bmatrix}$$

In beide gevallen is er zowel een matrix vermenigvuldiging als een vector optelling. Zo'n combinatie van operaties kan niet efficiënt door de grafische processor berekend worden.

3.2 Homogene coördinaten

Het begrip homogene voorstelling van objecten is ontwikkeld om problemen in verband met projectiemeetkunde op te lossen. Er is volgende stelling bewezen:

een probleem in een n-dimensionale ruimte heeft telkens een overeenkomstig probleem in een $(n+1)$ dimensionale ruimte

Er is vastgesteld dat de oplossing van een probleem in de ruimte met dimensie $(n+1)$ vaak eenvoudiger te bekomen is dank zij de extra vrijheidsgraad. De oplossing in de $(n+1)$-dimensionale ruimte kan dan op de één of andere manier geprojecteerd worden naar de ruimte met dimensie n. De term *homogoniseren* wordt gebruikt om een punt in de n-dimensionale ruimte om te vormen naar een punt in de $(n+1)$-dimensionale ruimte; het is een één-op-veel mapping. De term *projecteren* wordt gebruikt voor de tegengestelde operatie: een punt in de $(n+1)$-dimensionale ruimte om te vormen naar een punt in de n-dimensionale ruimte; het is een veel-op-één mapping.
Om een punt met twee coördinaten (x, y) te homogeniseren, wordt een derde coördinaat toegevoegd, nl. $w \neq 0$, de schaalfactor. Het punt wordt nu bepaald door drie coördinaten: (wx, wy, w), waarbij w oneindig veel waarden kan aannemen. (1,1,1) en (3,3,3) verwijzen naar hetzelfde punt in het vlak.
Bij projectie van een punt met homogene coördinaten $(16, 8, 4)$ naar een ruimte met één dimensie minder, worden x en y door de schaalfactor gedeeld: $(16/4, 8/4)$ of $(4, 2)$.
Hetzelfde geldt uiteraard voor 3D (wat in volgende hoofdstukken zal gebruikt worden): een punt (x, y, z) homogeniseren komt overeen met een schaalfactor toevoegen in de 4e dimensie $(x, y, z, 1)$.

Toepassing op 2D viewing. Door de overgang naar homogene coördinaten wordt de 2×1 kolommatrix (voorstelling van een punt) omgevormd naar een 3×1 kolommatrix. De transformatiematrices worden 3×3 matrices.

- wereldcoördinaten naar genormaliseerde coördinaten

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x_w \\ y_w \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$\begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ 1 \end{bmatrix} \text{ met } \begin{cases} s_x = \frac{2}{xw_{\max} - xw_{\min}} \\ s_y = \frac{2}{yw_{\max} - yw_{\min}} \end{cases} \text{ en } \begin{cases} t_x = \frac{-xw_{\max} - xw_{\min}}{xw_{\max} - xw_{\min}} \\ t_y = \frac{-yw_{\max} - yw_{\min}}{yw_{\max} - yw_{\min}} \end{cases}$$

- genormaliseerde coördinaten naar schermcoördinaten

$$\begin{bmatrix} x_v \\ y_v \\ 1 \end{bmatrix} = \begin{bmatrix} s'_x & 0 & t'_x \\ 0 & s'_y & t'_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} \text{ met } \begin{cases} s'_x = \frac{xv_{\max} - xv_{\min}}{2} \\ s'_y = \frac{yv_{\max} - yv_{\min}}{2} \end{cases} \text{ en } \begin{cases} t'_x = \frac{xv_{\max} + xv_{\min}}{2} \\ t'_y = \frac{yv_{\max} + yv_{\min}}{2} \end{cases}$$

De volledige transformatie is het product van deze twee matrices:
Schermcoördinaten $\leftarrow$ genormaliseerde coördinaten $\leftarrow$ wereld coördinaten

$$\begin{bmatrix} x_v \\ y_v \\ 1 \end{bmatrix} = \begin{bmatrix} s'_x & 0 & t'_x \\ 0 & s'_y & t'_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ 1 \end{bmatrix}$$

Product uitwerken met invullen van s_x, s_y, t_x, t_y en s'_x, s'_y, t'_x, t'_y :

$$\begin{bmatrix} x_v \\ y_v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{xv_{\max}-xv_{\min}}{xw_{\max}-xw_{\min}} & 0 & \frac{xw_{\max}xv_{\min}-xw_{\min}xv_{\max}}{xw_{\max}-xw_{\min}} \\ 0 & \frac{yv_{\max}-yv_{\min}}{yw_{\max}-yw_{\min}} & \frac{yw_{\max}yv_{\min}-yw_{\min}yv_{\max}}{yw_{\max}-yw_{\min}} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ 1 \end{bmatrix}$$

3.3 2D viewing transformatie in OpenGL

In de voorbeelden in vorige hoofdstukken zijn hiervoor reeds volgende functies gebruikt.

1. selecteren van de juiste matrixmode: glMatrixMode(GL_PROJECTION);
   de top-of-stack van de projectiematrix-stack wordt de *actieve matrix*
   deze matrix wordt geïnitialiseerd met de eenheidsmatrix: glLoadIdentity();
   na wijzigen van de projectieparameters kan terug de eenheidsmatrix geladen worden;
2. definitie van een tweedimensionaal clipping window:

   ```
   gluOrtho2D( GLdouble left, GLdouble right,
                                GLdouble bottom, GLdouble top);
   ```

   de argumenten komen op de volgende manier overeen met de waarden uit de transformatieformules: left ($xw_{\min}$), right ($xw_{\max}$), bottom ($yw_{\min}$), top ($yw_{\max}$)
3. bepalen van transformatie van genormaliseerde naar schermcoördinaten:

   ```
   glViewport( GLint x, GLint y, GLint breed, GLint hoog);
   ```

   de argumenten komen op de volgende manier overeen met de waarden uit de transformatieformules: x ($xv_{\min}$), y ($yv_{\min}$), breed ($xv_{\max} - xv_{\min}$), hoog ($yv_{\max} - yv_{\min}$)

Voorbeeld. Bij gebruik van volgende functies

```
glViewport(0, 0, 600, 400);
gluOrtho2D( -2.5, 2.5, -1.0, 3.0);
```

worden volgende transformatiematrices opgebouwd.

$$\begin{bmatrix} 120 & 0 & 300 \\ 0 & 100 & 100 \\ 0 & 0 & 1 \end{bmatrix} \leftarrow \begin{bmatrix} 300 & 0 & 300 \\ 0 & 200 & 200 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.4 & 0 & 0 \\ 0 & 0.5 & -0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

Transformatie van punt $(1, 1)$:

$$\begin{bmatrix} 420.0 \\ 200.0 \\ 1.0 \end{bmatrix} \leftarrow \begin{bmatrix} 0.4 \\ 0.0 \\ 1.0 \end{bmatrix} \leftarrow \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

4 Modeltransformaties

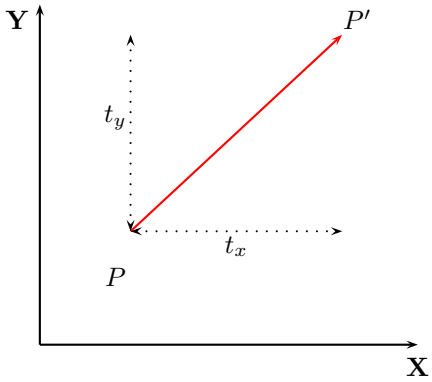
Naast het opbouwen van een tafereel met grafische primitieven moet een grafisch systeem ook in staat zijn de samenstellende objecten in een tafereel te laten evolueren. Voor het laten evolueren van objecten zijn er twee mogelijkheden:

1. met een tussentijd Δt de fysische positie van alle coördinaten van een tafereel herberekenen en dan, telkens opnieuw, alle objecten met hun nieuwe coördinaten tov. oorsprong genereren; deze manier van werken leidt tot zeer omslachtig programmeerwerk en moeilijk te onderhouden code;
2. eenmalig definiëren van alle objecten van het tafereel in de ruimte; en dan een aantal of alle objecten in het tafereel te onderwerpen aan een reeks transformaties om ze op die manier correct in de ruimte te situeren; dit resulteert in mooiere en efficiëntere code.

4.1 Modeltransformaties in het vlak

4.1.1 Translatie in 2D

Verplaatsen van een punt $P(x, y)$ naar $P'(x', y')$ over $\Delta x = t_x$ en $\Delta y = t_y$ in het vlak:



$$\begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$

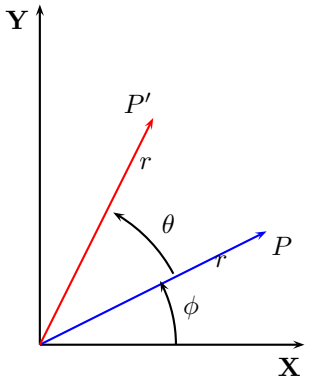
Matrixvorm:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$P' = P + T$$

4.1.2 Rotatie in 2D

Rotatie van een punt $P(x, y)$ in het vlak t.o.v. het rotatiepunt (pivot) over een rotatiehoek θ naar een punt $P'(x', y')$. De nieuwe ligging van het object wordt verkregen door de verplaatsing van alle punten langs een cirkelvormige baan in het xy-vlak. Een positieve rotatiehoek komt overeen met een rotatie tegen uurwijzerzin.



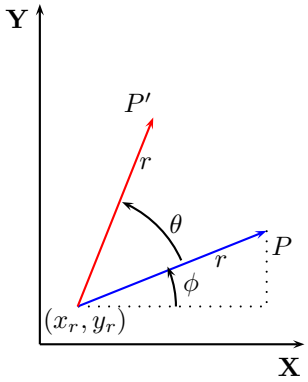
$$\begin{cases} x = r\cos\phi \\ y = r\sin\phi \end{cases} \quad \text{en} \quad \begin{cases} x' = r\cos(\phi+\theta) \\ y' = r\sin(\phi+\theta) \end{cases}$$

$$\begin{cases} x' = r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ y' = r\cos\phi\sin\theta + r\sin\phi\cos\theta \end{cases}$$

Matrixvorm:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} \qquad \text{of} \qquad P' = R \times P$$

Rotatie van een punt rond een willekeurig rotatiepunt (pivot) : het rotatiepunt (x_r, y_r) ligt niet in de oorsprong.



$$\begin{cases} x = x_r + r\cos\phi \\ y = y_r + r\sin\phi \end{cases} \quad \text{en} \quad \begin{cases} x' = x_r + r\cos(\phi+\theta) \\ y' = y_r + r\sin(\phi+\theta) \end{cases}$$

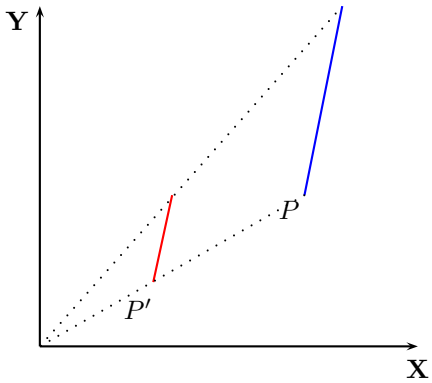
$$\begin{cases} x' = x_r + r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ y' = y_r + r\cos\phi\sin\theta + r\sin\phi\cos\theta \end{cases}$$

$$\begin{cases} x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta \\ y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \end{cases}$$

De formules voor de berekening van de nieuwe coördinaten bevatten zowel *multiplicatieve* als *additieve elementen*. Deze formules omvormen naar een efficiënte matrixvorm is moeilijk. Om deze transformatie te realiseren met een eenvoudige matrixvorm zal het probleem van de rotatie rond een willekeurig punt op een andere manier geformuleerd worden.

4.1.3 Schaling in 2D

Beïnvloeden van de grootte van een object: coördinaten (x, y) van het object worden vermenigvuldigd met respectievelijk schaalfactoren s_x en s_y: Het resultaat $P'(x', y')$ levert een bijschaling op van $P(x, y)$ t.o.v. de oorsprong.



$$\begin{cases} x' = xs_x \\ y' = ys_y \end{cases}$$

Matrixvorm:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

$$P' = S \times P$$

Toepassing: schalen van OpenGL primitieven met vaste initiële afmetingen, bijv. glutWireCube

4.1.4 Samenstelling van transformaties

De drie basistransformaties kunnen in matrixvorm geschreven worden:

$$P' = M_1 \times P + M_2$$

transformatie	M_1	M_2
translatie	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} t_x \\ t_y \end{bmatrix}$
rotatie	$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$
schaling	$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Elke transformatie afzonderlijk is eenvoudig implementeerbaar. Maar een efficiënte combinatie van verschillende transformaties is erg moeilijk realiseerbaar: alle berekeningen moeten voor elke deeltransformatie *punt per punt* uitgevoerd worden.

4.1.5 2D transformaties mbv. homogene coördinaten

Het samenstellen van transformaties vereist het combineren van optelling en vermenigvuldiging van matrices. De optelling zou hierin moeten kunnen vermeden worden. Analoog aan het probleem van vorig hoofdstuk, ligt ook hier de oplossing in het gebruik van homogene coördinaten. Dus de 2×1 en 2×2 matrices worden uitgebreid met een derde dimensie, de homogene schaalfactor. Een punt wordt weergegeven door een 3×1 kolommatrix; de transformatiematrices worden 3×3 matrices.
Elk van de basistransformaties kan in matrixvorm geschreven worden als een vermenigvuldiging van een 3×3 matrix met een 3×1 kolommatrix.

$$\text{translatie} \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad P' = T(t_x, t_y) \times P$$

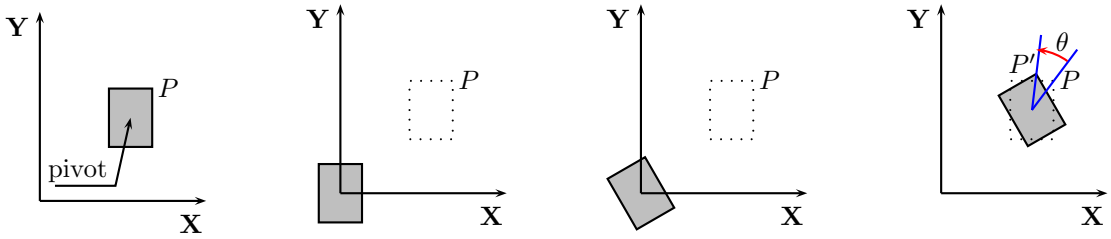
$$\text{rotatie} \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad P' = R(\theta) \times P$$

$$\text{schaling} \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad P' = S(s_x, s_y) \times P$$

Het *samenstellen* van transformaties wordt nu erg eenvoudig: het komt neer op het *vermeniguldigen* van matrices.

Rotatie (θ) **rond een willekeurig punt** (x_r, y_r) komt neer op:

1. translatie van het ganse object zodat het pivot punt in de oorsprong komt te liggen;
2. rotatie rond de oorsprong;
3. omgekeerde translatiebeweging van het geroteerde object.



$$P' = T(x_r, y_r) \times R(\theta) \times T(-x_r, -y_r) \times P$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & x_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -x_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

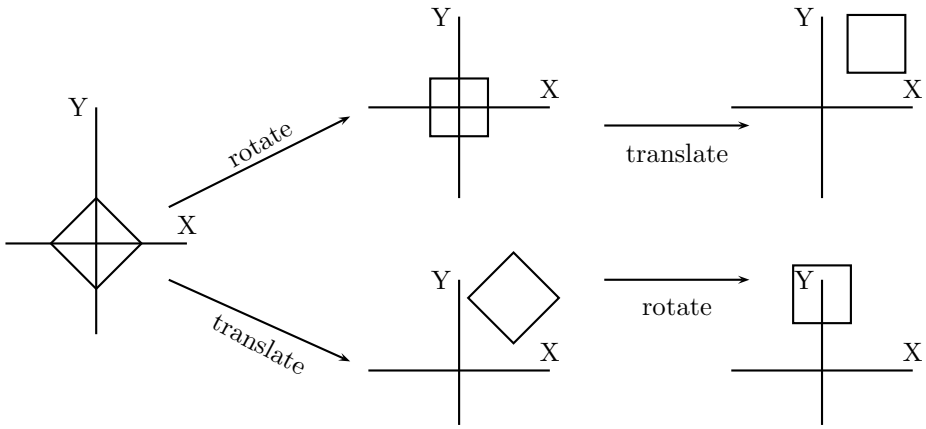
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r\sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r\sin\theta \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

De vermenigvuldiging van matrices is niet commutatief. De volgorde is belangrijk: transformaties worden uitgevoerd *van rechts naar links*.

4.1.6 Voorbeeld

Het roterend vierkant gerealiseerd met een modeltransformatie, in plaats van het herberekenen van de coördinaten van de hoekpunten.
Naast een rotatie transformatie is ook een translatie transformatie opgenomen Het resultaat is afhankelijk van de volgorde van deze transformaties. Indien eerst een rotatie uitgevoerd wordt en dan een translatie, dan roteert het vierkant rond zijn eigen middelpunt. Bij omgekeerde volgorde roteert het vierkant rond de oorsprong.
De rotatie- en translatiematrices kunnen op twee manieren samengevoegd worden: *pre-* of *post-*vermenigvuldiging.
Bij pre-vermenigvuldiging wordt de tweede matrix B langs links met de eerste matrix A vermenigvuldigd: het resultaat is dus $C = B \times A$.
Bij post-vermenigvuldiging wordt de tweede matrix B langs rechts met de eerste matrix A vermenigvuldigd: het resultaat is dus $C = A \times B$.
Beide volgordes zijn mogelijk, maar de juiste volgorde is wel afhankelijk van hoe de transformatie bekeken wordt, ten opzichte van een globaal coördinatensysteem of ten opzichte van een lokaal coördinatensysteem.

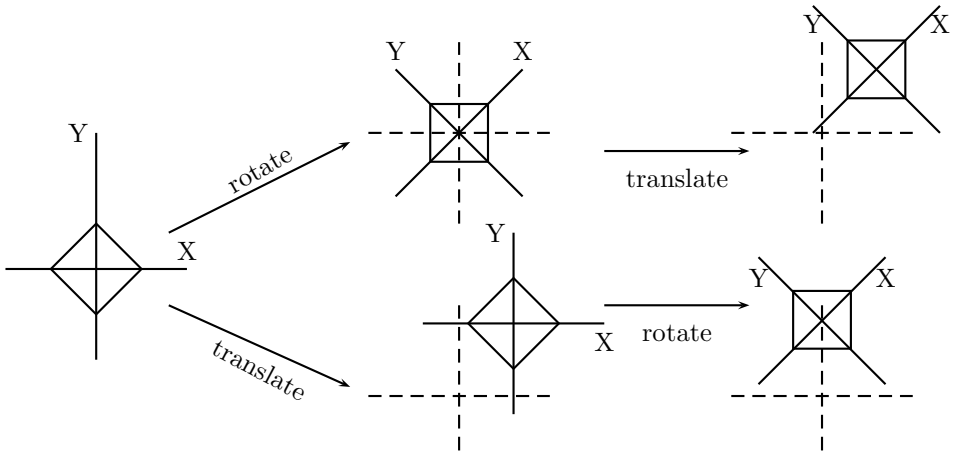
Globale wereldcoördinatensysteem. Elke transformatie die uitgevoerd wordt, is relatief ten opzichte van de globale oorsprong en de hoofdassen.



In dit geval moet pre-vermenigvuldiging gebeuren:
– beginnen met de eenheidsmatrix $M = I$
– eerst een rotatie $M' = R \times M$
– en dan een translatie $M'' = T \times M' = T \times R \times I$
De resulterende matrix wordt dan gebruikt om de vertices van het tafereel te transformeren :

$$v' = M'' \times v \qquad \text{of} \qquad v' = T \times R \times I \times v$$

Lokaal wereldcoördinatensysteem. Elke transformatie die uitgevoerd wordt, is een transformatie van het coördinatensysteem: de oorsprong en de assen worden dus mee getransformeerd.



In dit geval moet post-vermenigvuldiging gebeuren:
– beginnen met de eenheidsmatrix $M = I$
– eerst een rotatie $M' = I \times R$
– en dan een translatie $M'' = M' \times T = I \times R \times T$
De resulterende matrix wordt dan gebruikt om de vertices van het tafereel te transformeren :

$$v' = M'' \times v \qquad \text{of} \qquad v' = I \times R \times T \times v$$

OpenGl gebruikt *post*-vermenigvuldiging. Dus alle transformaties worden uitgevoerd ten opzichte het huidige coördinatensysteem. Elke transformatie wijzigt dus dit lokale coördinatensysteem.

```
void vierkant(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glColor3f(0.1, 0.1, 0.1);
    glBegin(GL_LINES);
        glVertex2f( -1.5, 0.0 );
        glVertex2f( 1.5, 0.0 );
        glVertex2f( 0.0, -1.5 );
        glVertex2f( 0.0, 1.5 );
    glEnd();
    // glPushMatrix();
    glScalef(0.75, 0.75, 1.0);
    if ( rotatieeerst )
    {
        glRotatef(hoek, 0.0, 0.0, 1.0);
        glTranslatef(0.75, 0.75, 0.0);
    }
    else
    {
        glTranslatef(0.75, 0.75, 0.0);
        glRotatef(hoek, 0.0, 0.0, 1.0);
    }
    glBegin(GL_QUADS);
        glColor3fv(kleurrechts);
        glVertex2f( 1.0, 0.0 );
        glColor3fv(kleurboven);
        glVertex2f( 0.0, 1.0 );
```

```
30              glColor3fv(kleurlinks);
                glVertex2f( -1.0, 0.0 );
32              glColor3fv(kleuronder);
                glVertex2f( 0.0, -1.0 );
34          glEnd();
            // glPopMatrix();
36          glutSwapBuffers();
            glFlush();
38      }
```

Merk op dat de volgorde van uitvoeren van transformaties omgekeerd is aan de volgorde van de functieoproepen in het programma. De laatst gespecificeerde transformatie is de eerste die toegepast wordt.

4.2 Modeltransformaties in 3D

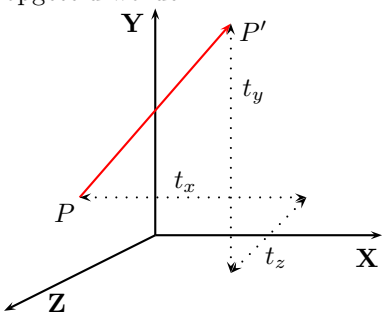
De wiskundige technieken die in 3D toegepast worden, zijn gewoon een uitbreiding van deze die we in 2D gezien hebben.

- elke transformatie wordt gekarakteriseerd door een 4×4 transformatiematrix, omdat er gewerkt wordt in homogene coördinaten;
- samengestelde transformaties, een sequentie van basistransformaties, geven door concatenation aanleiding tot één enkele transformatiematrix; deze levert precies hetzelfde resultaat op als de uitvoering van de opeenvolgende deeltransformaties.

Het grafische systeem hoeft slechts eenmaal de vermenigvuldiging uit te voeren van de verschillende transformatiematrices. De resulterende matrix wordt gebruikt om alle punten van het object aan de transformatie te onderwerpen. Hoe ingewikkeld de combinatie van transformaties ook moge zijn, elk punt van het te transformateren object wordt slechts vermenigvuldigd met één enkele matrix. Bij complexere transformaties is de enige extra berekeningslast de eenmalige berekening van de resulterende matrix.
Bij de bespreking van de verschillende transformaties worden de beschikbare OpenGL functies vermeld. Zo'n functie genereert een transformatiematrix die met de actieve transformatiematrix vermenigvuldigd wordt. *Concatenation* gebeurt dus eigenlijk automatisch.

4.2.1 Translatie in 3D

De transformatie die een punt $P = (x, y, z)$ in de ruimte verplaatst naar een nieuwe positie $P' = (x', y', z')$, moet er voor zorgen dat bij elke coördinaat van P de transformatieparameters opgeteld worden.



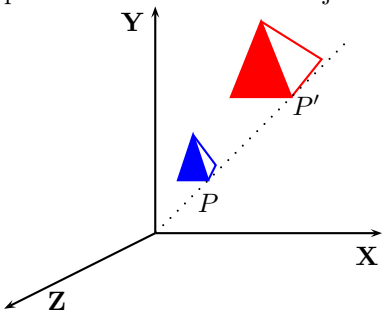
$$P' = T(t_x, t_y, t_z) \times P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

```
glTranslated( GLdouble tx, GLdouble ty, GLdouble tz);
glTranslatef( GLfloat tx, GLfloat ty, GLfloat tz);
```

4.2.2 Schaling in 3D

Afhankelijk van de schaalfactoren wordt het object in de ruimte vergroot of verkleind. Dit kan per dimensie verschillend zijn.



$$P' = S(s_x, s_y, s_z) \times P$$

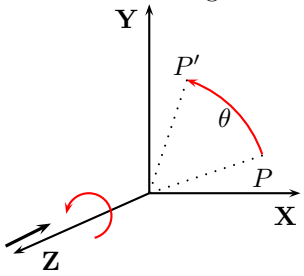
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

```
glScaled( GLdouble sx, GLdouble sy, GLdouble sz );
glScalef( GLfloat sx, GLfloat sy, GLfloat sz );
```

4.2.3 Rotatie

Dit is niet zomaar een uitbreiding van een rotatie in 2D. Een object kan in de ruimte geroteerd worden om elke willekeurige rechte. De eenvoudigste rotaties zijn uiteraard deze met een hoofdas als rotatieas. Er is dus een extra vrijheidsgraad, namelijk de rotatieas. Uit de drie basisrotaties (respektievelijk rond z-, x- en y-as) kunnen complexere rotaties samengesteld worden via *concatenation*.

Rotatie rond z-as in 3D : is vergelijkbaar met een rotatie rond de oorsprong in 2D; de z-coördinaat wordt gewoon toegevoegd en wordt door de rotatie niet gewijzigd.



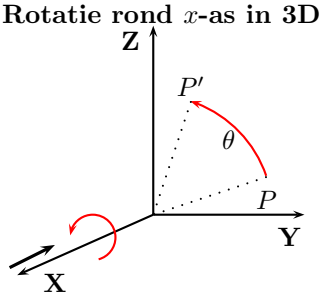
$$P' = R_z(\theta) \times P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

```
glRotated( GLdouble hoek, GLdouble vx, GLdouble vy, GLdouble vz );
glRotatef( GLfloat hoek, GLfloat vx, GLfloat vy, GLfloat vz );
```

Bij een rotatie rond de z-as zijn de argementen `vx` en `vy` gelijk aan 0.0 en heeft het argument `vz` de waarde 1.0.
De twee andere transformatiematrices kunnen afgeleid worden door een cyclische permutatie : x wordt vervangen door y, y wordt vervangen door z, en z wordt vervangen door x,

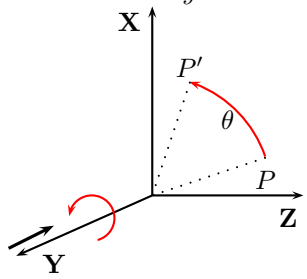
Rotatie rond x-as in 3D :



$$P' = R_x(\theta) \times P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotatie rond y-as in 3D :



$$P' = R_y(\theta) \times P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

4.3 Ondersteuning in OpenGL

Naast de reeds vermelde functies voor elke basistransformatie voorziet OpenGL in een aantal nuttige ondersteuningsfuncties.
OpenGL kent naast modeltransformaties ook projectietransformaties en textuurtransformaties. Bij het oproepen van functies voor modeltransformaties moet het OpenGL systeem zich in de juiste toestand bevinden op het vlak van de toestandsvariabele GL_MATRIX_MODE. Deze toestand van modeltransformaties instellen:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

De tweede functie zorgt ervoor dat begonnen wordt met een eenheidsmatrix als actieve transformatiematrix.
Door bijvoorbeeld glTranslatef(1.0, 0.0, 0.0) op te roepen wordt deze eenheidsmatrix (langs links?) vermenigvuldigd met de translatiematrix en wordt dus de actieve transformatiematrix aangepast. En bij elke volgende oproep van een transformatiefunctie wordt telkens de actieve transformatiematrix aangepast.
Het valt nog al eens voor dat er op een bepaald moment een actieve transformatiematrix A opgesteld is en dat men vertrekkende van deze matrix A eerst een aantal transformaties wilt doen resulterend in een matrix B, en dan vertrekkend van diezelfde A een aantal andere transformaties wilt doen resulterend in een matrix C. Maar eens matrix B gegenereerd is, is A niet meer beschikbaar en om deze te reconstrueren zou men de omgekeerde transformaties op B kunnen uitvoeren zodat A terug gegenereerd wordt om dan de tweede reeks van transformaties te doen om C te maken.
Omdat dit nogal omslachtig is en extra rekenwerk vergt, voorziet OpenGL een LIFO stackstrucuur voor het beheer van transformatiematrices. De stackdiepte voor de mode GL_MODEL_VIEW bedraagt minstens 32. De actieve matrix is de matrix die zich op de top van de stack bevindt.
De push en pop operaties zijn voorzien in OpenGL functies:

```
glPushMatrix();
glPopMatrix();
```

Met de eerste functie worden alle entries op de stack één plaats naar onder geduwd. De huidige actieve matrix (die op de top van de stack stond en één plaats naar onder geduwd is) wordt gedupliceerd op de top van de stack. De tweede functie schuift de ganse matrixstack opnieuw één plaats naar boven. De matrix die net onder de top van de stack staat, wordt opnieuw de actieve matrix.
Om bovenstaand scenario te realiseren, kan deze stack gebruikt worden:

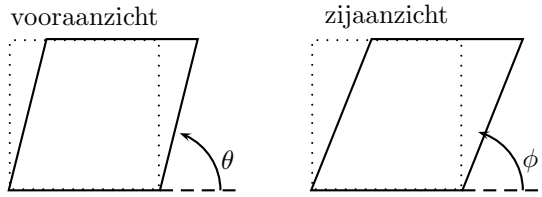
```
// transformaties resulterend in matrix A
glPushMatrix();      // matrix A wordt bewaard op de stack
// bijkomende transformaties resulterend in matrix B
glPopMatrix();      // matrix A wordt terug top van de stack
// bijkomende transformaties resulterend in matrix C
```

Wanneer een transformatie moet uitgevoerd worden die niet door een specifieke OpenGL functie op de actieve transformatie kan toegepast worden, kan gebruik gemaakt worden van functies die een directe manipulatie van de actieve matrix uitvoeren.

```
glLoadMatrixf(const GLfloat *m);
glMultMatrixf(const GLfloat *m);
```

Met de eerste functie wordt de actieve matrix opgevuld met 16 waarden, die aangeduid worden door het argument, de pointer `m`. De tweede functie vermenigvuldigt de actieve matrix met de 16 waarden van de matrix, aangeduid door het argument, de pointer `m`. In tegenstelling tot de rij-per-rij matrix organisatie van bijvoorbeeld de programmeertaal C, moet de matrix aangeduid door de pointer `m` kolom-per-kolom gestockeerd zijn.

Een parallellepipedum kan bekomen worden door een transformatie op een balk (geschaalde kubus) uit te voeren:
– de vlakken parallel met het yz-vlak wordt schuin gemaakt: $x' = x + \cot\theta \cdot y$
– de vlakken parallel met het xy-vlak wordt schuin gemaakt: $z' = z + \cot\phi \cdot y$



$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & \cot\theta & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & \cot\phi & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Voor deze *shear* transformatie is geen OpenGL functie voorzien. Implementatie kan door de matrix zelf op te bouwen en dan te vermenigvuldigen met de actieve transformatiematix:

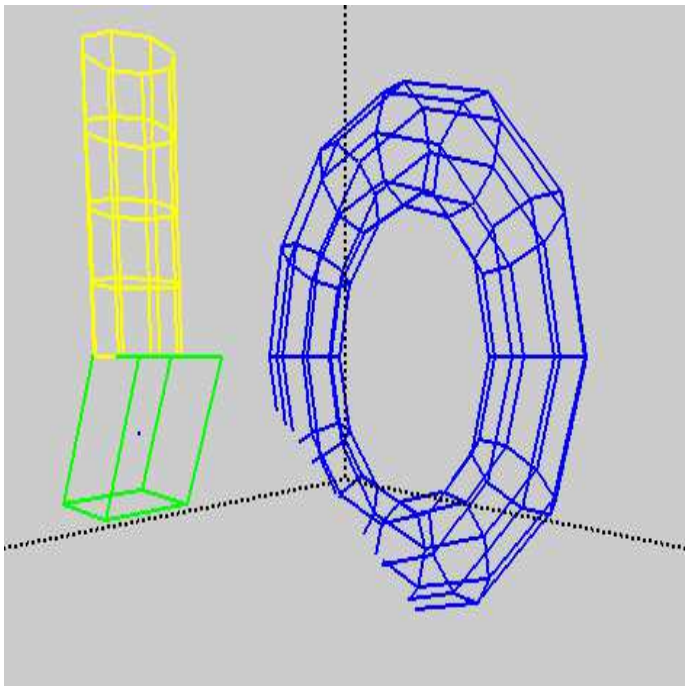
```
GLfloat mshear[16] = { 1.0, 0.0, 0.0, 0.0,  0.8, 1.0, 1.2, 0.0,
                       0.0, 0.0, 1.0, 0.0,  0.0, 0.0, 0.0, 1.0 };
glMultMatrixf(mshear);
```

Voorbeeld. Het tafereel is opgebouwd uit een torus waarvan een deel weggelaten is, een parallellepipedum en hier bovenop een cylinder. De verschillende volumes zijn ten opzichte van elkaar wat verschoven, voor de torus in een eigen PushMatrix - PopMatrix sequentie. Omdat de verschuiving voor de twee andere gedeeltelijk dezelfde is, is er een overkoepelende PushMatrix - PopMatrix sequentie. Maar de shear transformatie wordt enkel op de kubus uitgevoerd in een eigen geneste PushMatrix - PopMatrix sequentie. Het rechtop zetten van de cylinder (standaard is de as evenwijdig met de z-as) zit ook in een eigen geneste PushMatrix - PopMatrix sequentie.

```
void torkubcyl(void)
{
    GLfloat smxz[16] = { 1.0,0.0,0.0,0.0,  1.0,1.0,1.0,0.0,
                         0.0,0.0,1.0,0.0,  0.0,0.0,0.0,1.0 };
    GLdouble xyvlak[4] = { 1.0, 1.0, 0.0, 0.25 };
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glClipPlane(GL_CLIP_PLANE0, xyvlak);
    gluLookAt(xlens,ylens,zlens, 0.0,0.0,0.0, 0.0,1.0,0.0);
    glColor3f(0.0, 0.0, 0.0);
    glPointSize(2.0);
    glLineStipple(1, 0xCCCC);
    glEnable(GL_LINE_STIPPLE);
    glBegin(GL_LINES);
```

```
16              glVertex3f(0.0,0.0,0.0);
                glVertex3f(5.0,0.0,0.0);
18              glVertex3f(0.0,0.0,0.0);
                glVertex3f(0.0,5.0,0.0);
20              glVertex3f(0.0,0.0,0.0);
                glVertex3f(0.0,0.0,5.0);
22          glEnd();
            glDisable(GL_LINE_STIPPLE);
24          if ( projectie == 'o' )
            {
26              glColor3f(0.7, 0.7, 0.7);
                glBegin(GL_QUADS);
28                  glVertex3f(-1.25, 1.0, 0.0);
                    glVertex3f(-1.25, 1.0, -1.0);
30                  glVertex3f(1.0, -1.25, -1.0);
                    glVertex3f(1.0, -1.25, 0.0);
32              glEnd();
            }
34          glLineWidth(2.5);
            if ( torus )
36          {
                glPushMatrix();
38              glEnable(GL_CLIP_PLANE0);
                glColor3f(0.0, 0.0, 1.0);
40              glScalef(tsx, tsy, tsz);
                glTranslatef(ttx, tty, ttz);
42              glRotatef(hoek, 0.0, 0.0, 1.0);
                if ( draad )
44                  glutWireTorus(0.5, 1.8, 8, 12);
                else
46                  glutSolidTorus(0.5, 1.8, 8, 12);
                glDisable(GL_CLIP_PLANE0);
48              glPopMatrix();
            }
50          if ( kubus )
            {
52              smxz[4] = (float)1.0/tan(M_PI*80.0/180.0);
                smxz[6] = (float)1.0/tan(M_PI*70.0/180.0);
54              glPushMatrix();
                glPushAttrib(GL_CURRENT_BIT);
56              glColor3f(0.0, 1.0, 0.0);
                glTranslatef(ktx, kty, ktz);
58              glRotatef(hoek, 0.0, 1.0, 0.0);
                glPushMatrix();
60              glMultMatrixf(smxz);
                glScalef(ksx, ksy, ksz);
62              if ( draad )
                    glutWireCube(1.0);
64              else
                    glutSolidCube(1.0);
66              glPopMatrix();
                if ( cylinder )
68              {
                    GLUquadricObj *rol;
```

```
70                  rol = gluNewQuadric();
                    glPushMatrix();
72                  glColor3f(1.0, 1.0, 0.0);
                    glTranslatef(ctx, cty, ctz);
74                  glRotatef(-90.0, 1.0, 0.0, 0.0);
                    if ( draad )
76                      gluQuadricDrawStyle(rol, GLU_LINE);
                    else
78                      gluQuadricDrawStyle(rol, GLU_SILHOUETTE);
                    gluCylinder(rol, 0.4, 0.4, 1.9, 8, 4);
80                  glPopMatrix();
                    gluDeleteQuadric(rol);
82              }
                glPopAttrib();
84              glBegin(GL_POINTS); glVertex3f(0.0,0.0,0.0);    glEnd();
                glPopMatrix();
86          }
            glFlush();
88      }
```



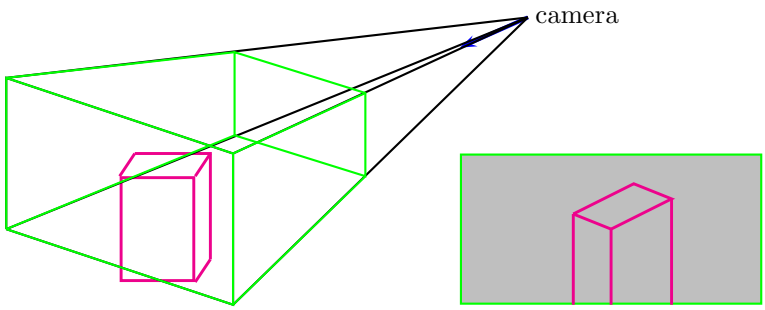
Figuur 4.1: Illustratie van Push/Pop

5 3D visualisering

5.1 Camera analogie

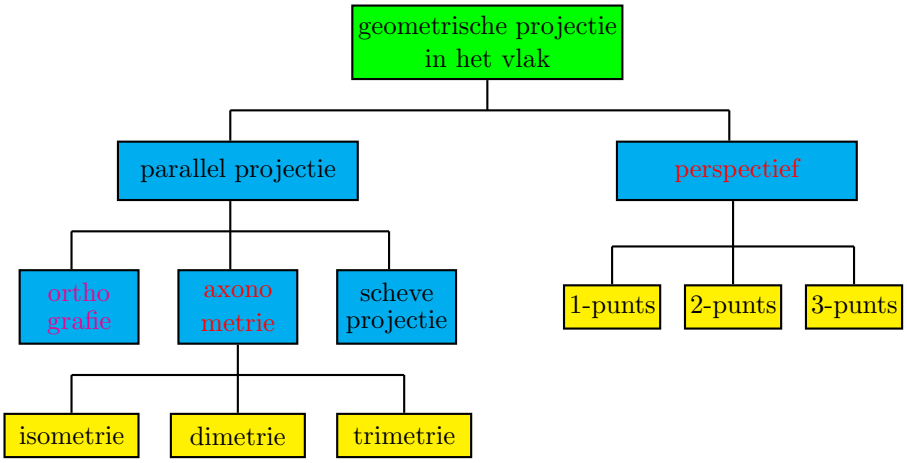
In tegenstelling tot de 2D visualisering van een grafiek, waar alles zich in het xy-vlak afspeelt, heeft de problematiek van de visualisering van 3D taferelen letterlijk een extra dimensie. 3D-visualisering is de creatie van een voorstelling van een 3D tafereel in het vlak. De parameters die we kunnen manipuleren om deze transformatie van 3D naar 2D te realiseren, komen min of meer overeen met de parameters waarover een fotograaf beschikt bij het maken van een foto:

- in functie van het weer te geven tafereel, keuze van de optimale *positie* en *oriëntatie* van de camera;
- keuze van de karakteristieken van de *lens*, bijv. openingshoek;
- bepalen van beeldgedeelte van het tafereel (in/uit zoomen): *clipping*;
- foto nemen : *projectie* op een vlak.



Bij fotografie is het de bedoeling om een voor onze hersenen makkelijk herkenbaar en interpreteerbaar beeld op papier te zetten, dat zo goed mogelijk overeenkomt met onze visuele waarneming. Computer 3D visualisering is echter ruimer. In een technische omgeving moet een ruimtelijk object ondubbelzinnig weergegeven worden door meerdere aanzichten te gebruiken met behulp van orthogonale projectie (iets wat we met onze ogen onmogelijk kunnen uitvoeren). Langs de andere kant beschikken onze hersenen over mogelijkheden om uit een waarneming van beide ogen een soort stereoscopie te distilleren die zeer moeilijk op een 2D computerscherm te vatten is.

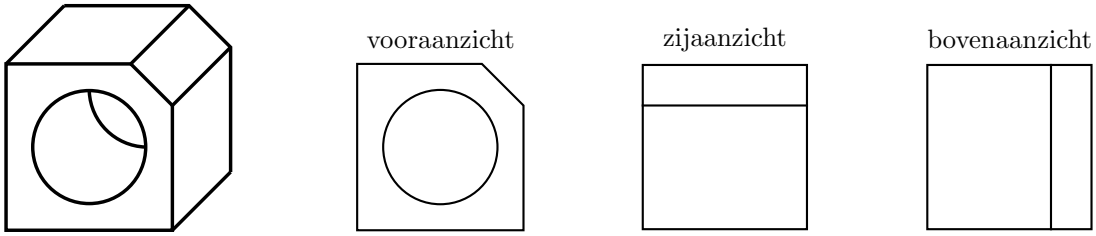
5.2 Projectietransformaties



De visualisering van 3D-taferelen op papier (dus in 2D) is ontwikkeld lang voordat er sprake was van computers en computergrafieken. Er zijn in de loop der jaren verschillende projecties

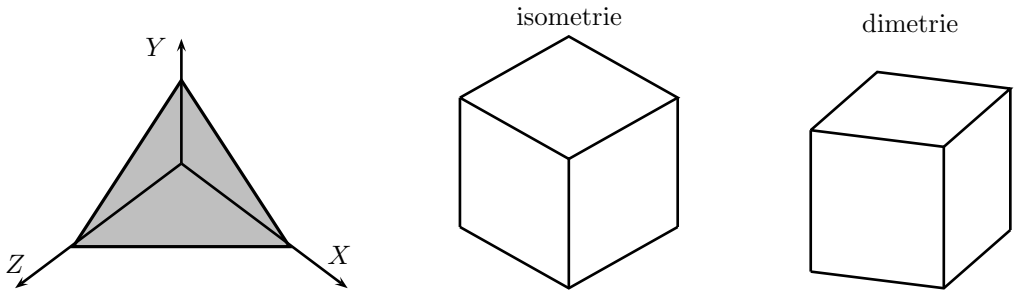
ontwikkeld. Hierbij is telkens gestreefd naar een evenwicht tussen manuele arbeid met pen en papier en het visuele rendement van het resultaat.
In een mechanische productieomgeving heeft men behoefte aan een plan waarop alle afmetingen correct worden weergegeven, indien nodig met meerdere aanzichten. Een architect maakt van een ontwerp een tekening in perspectief om zo de ruimtelijke impact van een gebouw op zijn omgeving te tonen.

Orthografie of orthogonale projectie. De projectie gebeurt via lijnen evenwijdig met de hoofdassen. Op die manier ontstaan de klassieke aanzichten: vooraanzicht, zijaanzicht en bovenaanzicht.

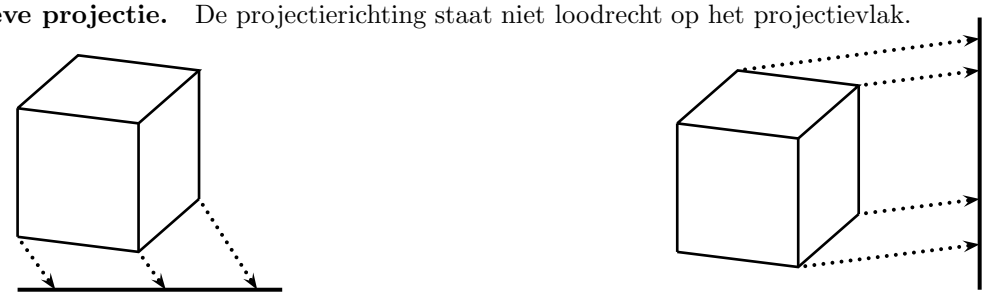


De lengte van de lijnstukken en de hoeken worden bewaard en kunnen in de respektievelijke aanzichten nagemeten worden. Omwille daarvan wordt deze projectiemethode gebruikt bij mechanische productie en bouw.

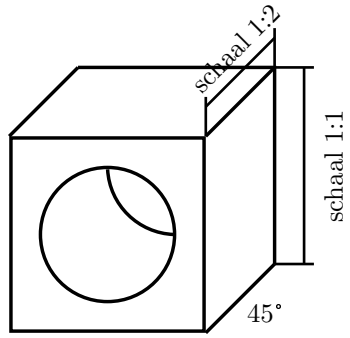
Axonometrie. Het projectievlak is niet evenwijdig met één van de basisvlakken. Het snijdt de drie hoofdassen onder bepaalde hoeken. De projectierichting staat loodrecht op het projectievlak. Axonometrie laat toe een gevoel van perspectief te creëren terwijl toch gebruikt gemaakt wordt van parallelle projectietransformaties (er is geen vluchtpunt).
Wanneer het projectievlak de drie assen op gelijke afstand van de oorsprong snijdt, zijn de drie hoeken gelijk aan elkaar. Men spreekt dan van *isometrie*.
Indien slechts twee van de drie hoeken aan elkaar gelijk zijn, heeft men *dimetrie*. Het resultaat is realistischer dan wat men bekomt met isometrie. Dit heeft te maken met het feit dat de projectie van de $z-$as ingekort wordt.
Met drie verschillende hoeken bekomt men *trimetrie*. Vermits de visuele kwaliteit van een trimetrische projectie nauwelijks beter is dan bij dimetrie, terwijl er nu drie in plaats van twee schaalfactoren gebruikt moeten worden, wordt deze projectiemethode zeer weinig toegepast.

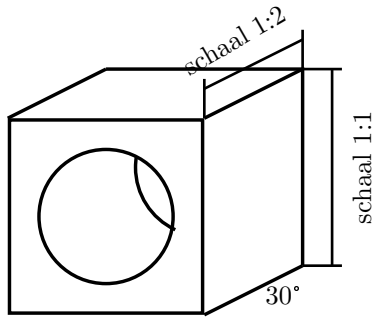


Scheve projectie. De projectierichting staat niet loodrecht op het projectievlak.

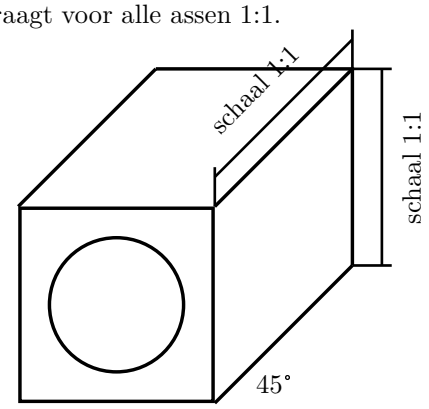


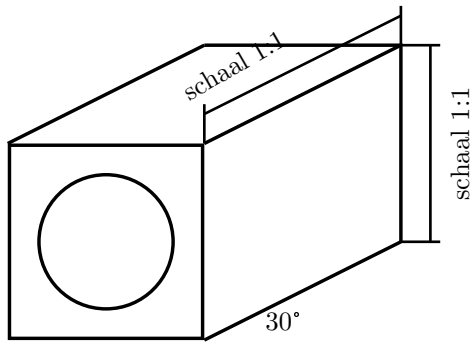
Bij *kabinetprojectie* is het projectievlak evenwijdig met het vooraanzicht, zodat dit onvervormd wordt weergegeven. De $z-$as wordt getekend onder een hoek van 45° of 30° en de schaalfactor voor alle punten langs de $z-$as bedraagt 1:2.





Bij *ruiterprojectie* is het projectievlak evenwijdig met het vooraanzicht, zodat dit onvervormd wordt weergegeven. De $z-$as wordt getekend onder een hoek van 45° of 30° en de schaalfactor bedraagt voor alle assen 1:1.



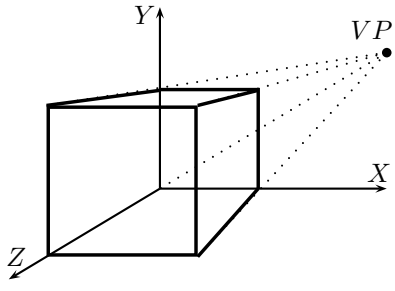


Perspectief. In een voortdurend streven naar een realistische voorstelling van de realiteit, gingen kunstenaars in de vijftiende en zestiende eeuw op zoek naar een formele manier om ruimtelijke objecten correct voor te stellen; d.i. in overeenstemming met de visuele waarneming. Het basisidee is geformuleerd door Brunelleschi (1377-1446)

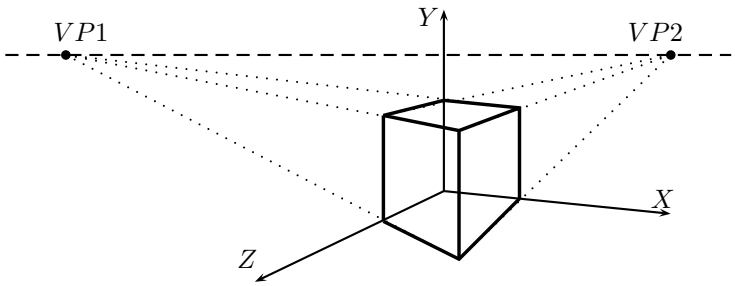
> Alle ruimtelijke lijnen die niet evenwijdig lopen aan het canvas komen samen in een vluchtpunt.

Afhankelijk van de plaatsing van de waarnemer (camera) wordt onderscheid gemaakt tussen perspectief met respectievelijk 1, 2 of 3 vluchtpunten.

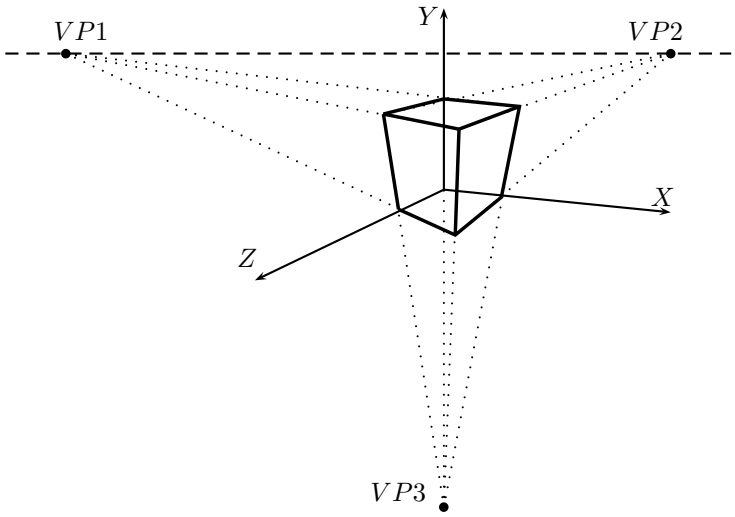
Perspectief met één vluchtpunt. De richtingsvector van de camera staat niet loodrecht op één van de hoofdassen (bijv. de z-as): de camera kijkt in de richting van de z-as. Alle lijnen die gelegen zijn in een vlak evenwijdig aan het xy-vlak, hebben hun vluchtpunt op oneindig. Het projectievlak wordt enkel gesneden door de z-as.



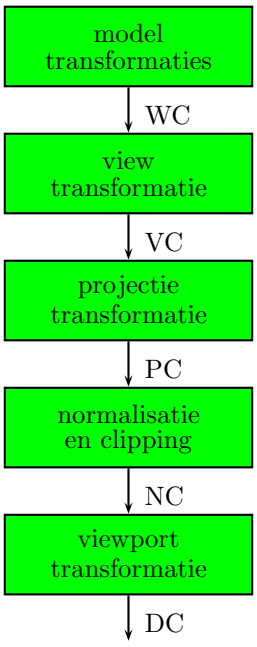
Perspectief met twee vluchtpunten. De richtingsvector van de camera staat niet loodrecht op twee van de drie hoofdassen. In het voorbeeld is een camerapositie gekozen die loodrecht kijkt op de $y-$as, bijvoorbeeld vanuit een punt in het xz-vlak, Het projectievlak heeft een snijpunt met x-as en z-as.



Perspectief met drie vluchtpunten. Alle hoofdassen hebben een vluchtpunt. X-, y- en z-as snijden alle drie het projectievlak. De camera wordt bijvoorbeeld gepositioneerd op $(10, 10, 10)$ kijkend naar de oorsprong $(0, 0, 0)$.



5.3 3D-visualisering pijplijn



Modelering van het tafereel in wereldcoördinaten (WC).

Keuze van de positie en de oriëntatie van de camera. Het resultaat van deze transformatie levert viewcoördinaten (VC) op die nog steeds het ruimtelijk tafereel beschrijven in drie dimensies.

Aflijnen van een drie-dimensionale view: alle objecten buiten het clipping volume zullen niet zichtbaar zijn; alle ruimtelijke coördinaten worden geprojecteerd op het projectievlak (PC)
(systeem behoudt z-coördinaar voor de bepaling van de zichtbaarheid van objecten)

Alle coördinaten worden genormaliseerd (NC) naar bijvoorbeeld een eenheidskubus in functie van de ruimtelijke clipping-operaties

De genormaliseerde 2D-projectie wordt gemapt op het scherm (display-coördinaten, DC).

5.4 View transformatie

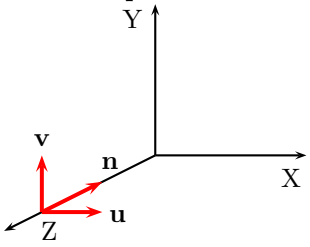
Camera wordt in punt P_0 met coördinaten (c_x, c_y, c_z) geplaatst. In de camera wordt een coördinatensysteem geplaatst bepaald door vectoren (u, v, n). Het **uv**-vlak is het projectievlak en **n** definieert de richting waarin de camera kijkt. De coördinaten van alle objecten in VC worden bekomen door het ruimtelijk tafereel te beschrijven met het **uvn**-coördinatensysteem.
Transformatie van xyz-wereldco's naar uvn view-co's heeft twee componenten:

1. translatie van oorsprong van VC-systeem naar oorsprong van WC-systeem
2. rotaties om de hoofdassen van beide systemen op elkaar te leggen.

$$T = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad M_{WC,VC} = R \times T$$

Enkele speciale gevallen (die nuttig zijn voor de oefening)

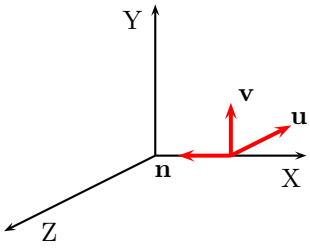
camera op Z-as : rotatie-view-matrix



projectie van n-as op Z-as : tegengestelde richting

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

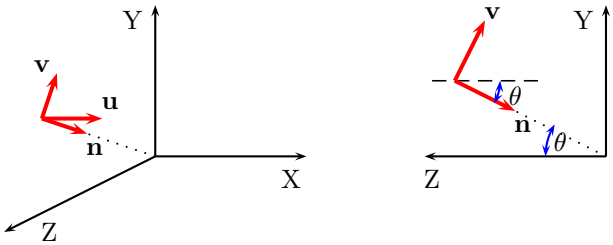
camera op X-as : rotatie-view-matrix



projectie van u-as op Z-as : tegengestelde richting
projectie van n-as op X-as : tegengestelde richting

$$R = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

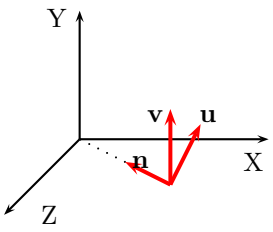
camera in YZ-vlak camera in punt $(0, c_y, c_z)$

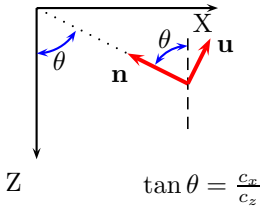


$$\tan\theta = \frac{c_y}{c_z}$$

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{of} \quad R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & -\sin\theta & -\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

camera in XZ-vlak in $(c_x, 0, c_z)$

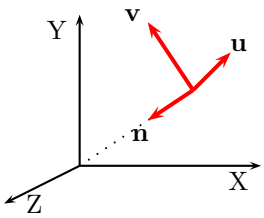


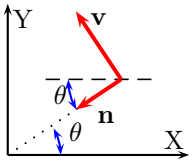


$\tan\theta = \frac{c_x}{c_z}$

$$R = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & -\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

camera in XY-vlak in $(c_x, c_y, 0)$





$\tan\theta = \frac{c_y}{c_x}$

$$R = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ -\cos\theta & -\sin\theta & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.5 Projectie transformatie

5.5.1 Orthogonale projectie

Projectie van het 3D tafereel gebeurt volgens evenwijdige projectielijnen loodrecht op een projectievlak. Met z-as als projectierichting is het projectievlak evenwijdig met xy-vlak. Een punt (x, y, z) wordt geprojecteerd in een punt (x_p, y_p):

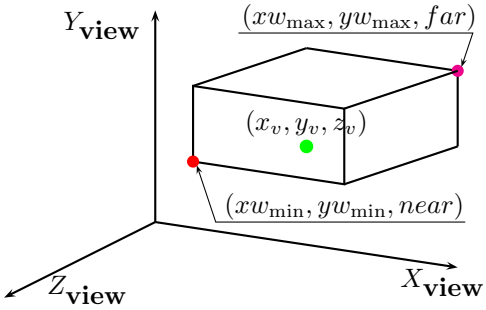
$$\begin{cases} x_p = x \\ y_p = y \end{cases}$$

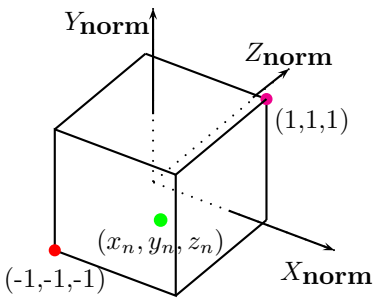
De z-coördinaat is niet relevant voor de projectie, wel voor dieptegevoel. Van objecten waarvan de loodrechte projectie samenvallen, moet de relatieve positie bepaald worden om zo te beslissen welk object zichtbaar is.

Normalisatie van alle coördinaten (analoog aan 2D visualisering): transformatiematrix $M_{VC,NC}$

$$\begin{bmatrix} s_x & 0 & 0 & t_x \\ 0 & s_y & 0 & t_y \\ 0 & 0 & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{cases} s_x = \frac{2}{xw_{\max} - xw_{min}} \\ s_y = \frac{2}{yw_{\max} - yw_{min}} \\ s_z = \frac{2}{far - near} \end{cases} \qquad \begin{cases} t_x = -\frac{xw_{\max} + xw_{min}}{xw_{\max} - xw_{min}} \\ t_y = -\frac{yw_{\max} + yw_{min}}{yw_{\max} - yw_{min}} \\ t_z = -\frac{far + near}{far - near} \end{cases}$$





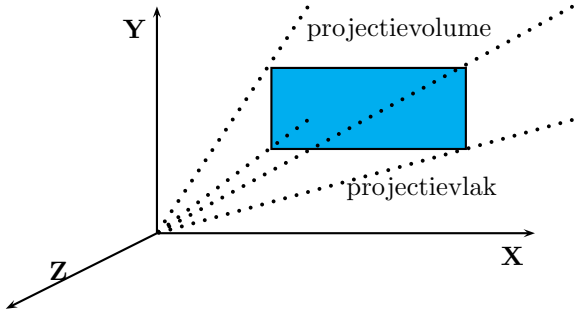
Uit de view-coördinaten worden de genormaliseerde coördinaten berekend:

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & t_x \\ 0 & s_y & 0 & t_y \\ 0 & 0 & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

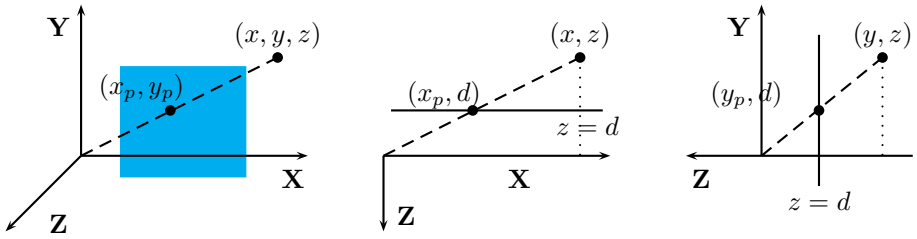
5.5.2 Perspectief projectie

Het gebruik van perspectief in de visualisering van ruimtelijke objecten tracht de visuele waarneming zo goed mogelijk te benaderen. De visuele waarneming van onze omgeving kan benaderd worden door een projectie van de ruimte op het 2D netvlies. Zo'n visuele waarneming ontstaat door het feit dat lichtstralen vanuit alle mogelijke richtingen in ons gezichtsveld van een lichtbron, via reflecties op de objecten tenslotte het netvlies bereiken. Schematisch kan dit voorgesteld worden door een piramide die zich uitstrekt vanaf het oog met een gegeven gezichtshoek horizontaal en vertikaal, waarin zich alle waarneembare punten in de ruimte bevinden.

Nevenstaande figuur toont een convergerende lichtbundel die op het netvlies van een observator of op de lens van een camera die het 3D tafereel waarneemt, invalt. Alle punten in de ruimte worden via convergerende projectielijnen op het projectievlak geprojecteerd.



Bij waarneming vanuit de oorsprong in een richting parallel aan de z-as en het projectievlak evenwijdig met het xy-vlak wordt het punt (x, y, z) in de ruimte geprojecteerd op (x_p, y_p):



Omwille van congruentie van driehoeken geldt er

$$\frac{x}{x_p} = \frac{z}{d} \qquad \frac{y}{y_p} = \frac{z}{d} \qquad x_p = \frac{x}{\frac{z}{d}} \qquad y_p = \frac{y}{\frac{z}{d}} \qquad z_p = d$$

Hieruit kan geen projectiematrix geconstrueerd worden, omdat de noemer van x_p en y_p functie is van z. Zoals in vorige hoofdstukken brengen we een vierde dimensie in het spel.

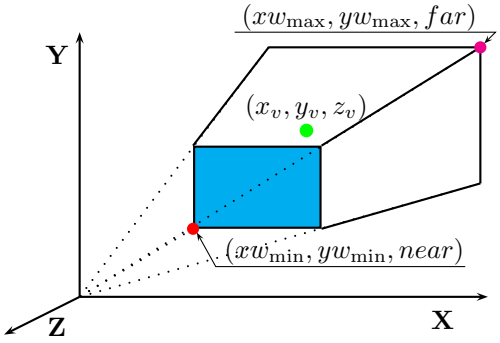
Perspectiefprojectie met homogene coördinaten. Een punt in de ruimte wordt voorgesteld door een kolom-matrix $P = (x, y, z, 1)$. P_h is een kolom-matrix van dat punt met homogene coördinaten (x_h, y_h, z_h, h).
Herschrijven van de formules:

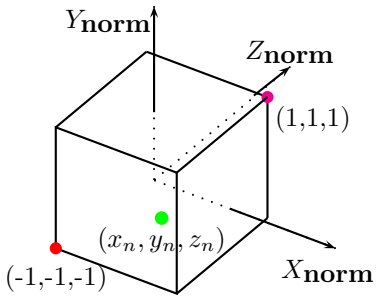
$$x_p = \frac{x_h}{h} \qquad y_p = \frac{y_h}{h} \qquad h = -z \qquad \text{met} \qquad x_h = -dx \qquad y_h = -dy$$

Er kan een transformatiematrix gevormd worden:

$$M_p = \begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad \text{en} \quad \begin{cases} P_h = M_p \times P \\ P_p \text{ gedehomogeniseerde } P_h \end{cases}$$

De nodige coördinaten van het geprojecteerde punt (P_p) worden dus bekomen door de elementen van P_h te delen door de homogene parameter h.
Om de zichtbaarheid van overlappende objecten te kunnen bepalen (wie bedekt wie of wat staat dichter bij de camera) wordt de $z-$coördinaat van elk geprojecteerd punt meegenomen in de berekening als een pseudo-diepte. De s_z en t_z elementen in de matrix zijn verantwoordelijk voor de berekening van de pseudo-diepte parameter.





Normalisatie: x-, y- en z- dimensies terugbrengen naar een bereik tussen -1 en +1. Voor de z-dimensie komt dit neer op een goede keuze voor s_z en t_z. Voor de twee andere dimensies betekent dit een schaling en translatie in beide hoofdassen.

Berekening van s_z en t_z in homogene coördinaten geldt voor z:

$$z_h = s_z z + t_z \qquad \text{dus, in projectie coördinaten} \qquad z_p = \frac{s_z z + t_z}{h} \qquad \text{of} \qquad \frac{s_z z + t_z}{-z}$$

interval $[near, far]$ normaliseren naar $[-1, +1]$:

$$\begin{cases} -1 &= \frac{s_z near + t_z}{-near} \\ +1 &= \frac{s_z far + t_z}{-far} \end{cases} \qquad \text{of} \qquad \begin{cases} +near &= s_z near + t_z \\ -far &= s_z far + t_z \end{cases}$$

de twee vergelijkingen van elkaar aftrekken:

$$near + far = s_z(near - far) \quad \text{of} \quad \boxed{s_z = \frac{near + far}{near - far}}$$

s_z invullen in de eerste vergelijking: $t_z = near - s_z near$:

$$\begin{aligned} t_z &= near - \tfrac{near+far}{near-far} near \\ &= \tfrac{near(near-far)-(near+far)near}{near-far} \end{aligned} \qquad \text{of} \qquad \boxed{t_z = -\frac{2 \times near \times far}{near - far}}$$

x- en y-dimensies: eerst een translatie

$$\begin{bmatrix} 1 & 0 & 0 & -\frac{xw_{\max}+xw_{\min}}{2} \\ 0 & 1 & 0 & -\frac{yw_{\max}+yw_{\min}}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} -near & 0 & 0 & 0 \\ 0 & -near & 0 & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad \text{of} \quad \begin{bmatrix} -near & 0 & \frac{xw_{\max}+xw_{\min}}{2} & 0 \\ 0 & -near & \frac{yw_{\max}+yw_{\min}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

en dan een schaling:

$$\begin{bmatrix} \frac{2}{xw_{\max}-xw_{\min}} & 0 & 0 & 0 \\ 0 & \frac{2}{yw_{\max}-yw_{\min}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} -near & 0 & \frac{xw_{\max}+xw_{\min}}{2} & 0 \\ 0 & -near & \frac{yw_{\max}+yw_{\min}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$\text{of} \quad \begin{bmatrix} \frac{-2near}{xw_{\max}-xw_{\min}} & 0 & \frac{xw_{\max}+xw_{\min}}{xw_{\max}-xw_{\min}} & 0 \\ 0 & \frac{-2near}{yw_{\max}-yw_{\min}} & \frac{yw_{\max}+yw_{\min}}{yw_{\max}-yw_{\min}} & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{-2near \times far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

5.5.3 Viewport transformatie

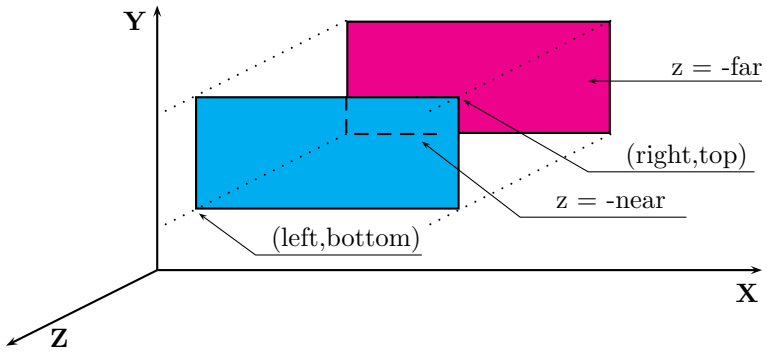
Tot slot moeten de genormaliseerde x- en y-coördinaten nog omgezet worden naar scherm- (of display-)coördinaten via de viewport transformatie (zie 2D visualisering hoofdstuk).
De genormaliseerde z-coördinaten worden in het *hidden surface removal* algoritme gebruikt om te bepalen wat vanuit het standpunt van de camera zichtbaar is en wat niet.

5.6 OpenGL functies

5.6.1 Projectie

Het kiezen van de lens om het 3D tafereel te projecteren op een 2D vlak heeft niets te maken met de opbouw van het tafereel. De eigenschappen van de lens worden uitgedrukt met matrixmode GL_PROJECTION.

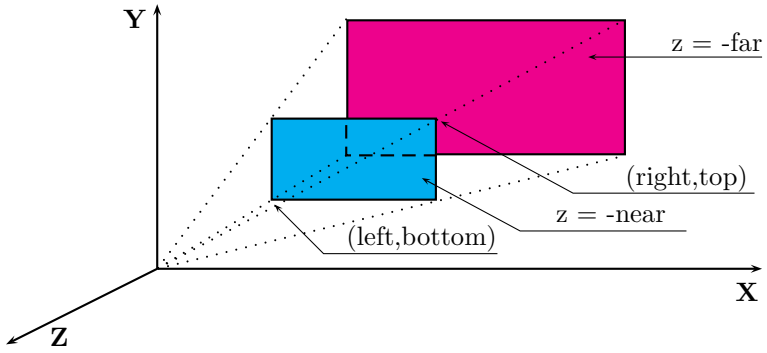
Orthogonale projectie waarbij de projectierichting loodrecht staat op het projectievlak. Wanneer geen andere transformaties actief zijn, is de projectierichting evenwijdig met de z-as.



```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho( GLdouble left, GLdouble right, GLdouble bottom,
         GLdouble top, GLdouble near, GLdouble far);
```

Er wordt een clipping volume (balk) gedefinieerd met hoekpunten (`left`, `bottom`, `-near`) en (`right`, `top`, `-far`). Het vlak op afstand `near` van de camerapositie is het projectievlak.

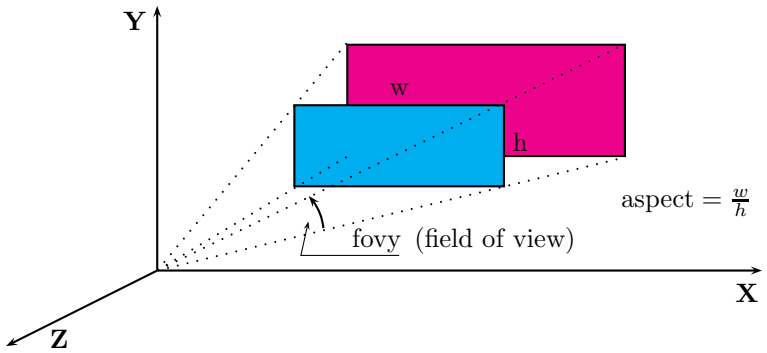
Algemeen perspectief waarbij het perspectiefveld benaderd wordt onder een hoek. Vanuit het standpunt van de camera kan een schuine perspectiefprojectie berekend worden indien het clipping window niet symmetrisch gekozen wordt t.o.v. de view $z-$as.




```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum( GLdouble left, GLdouble right, GLdouble bottom,
           GLdouble top, GLdouble near, GLdouble far);
```

Argumenten `left`, `right`, `bottom` en `top` bepalen bepalen de grenzen van de clipping rechthoek op een afstand `near` van de camerapositie. Hier ligt ook het projectievlak. Argument `far` legt de positie vast van het einde van het view volume.
De term frustum wodt in de meetkunde gebruikt om een volume aan te duiden dat overblijft wanneer een kegel of een piramide wordt afgeknot door een vlak evenwijdig met de basis. In dit geval zit de top van de view piramide in de view oorsprong (d.i. de camerapositie) en wordt deze op `near` afgeknot; `far` bepaalt de ligging van het grondvlak van deze view piramide.
Deze scheve perspectiefprojectie geeft doorgaans een eerder vertekend beeld en wordt daarom slechts zelden gebruikt.

Symmetrisch perspectief is een speciaal geval van een algemeen perspectief: het view volume is symmetrisch t.o.v. de z-as van het camera-assenkruis.

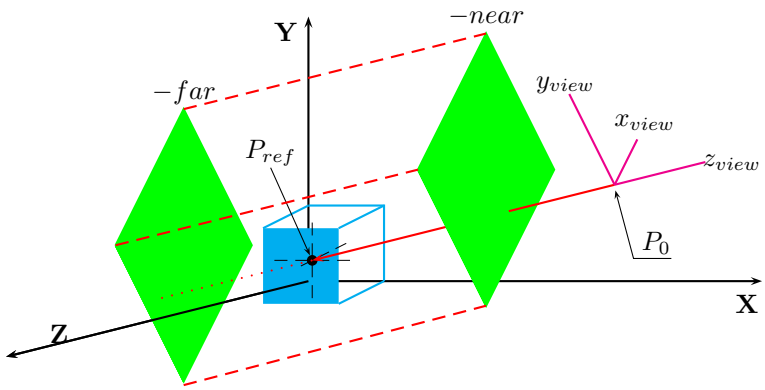


```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);
```

Argument `fovy` bepaalt de gezichtshoek in de $y-$richting. Op een afstand `near` van de camera positie bepaalt dit de hoogte van het projectievlak. Via het argument `aspect` (breedte-hoogte verhouding) ligt de breedte van het viewing volume vast. Het viewing volume strekt zich uit tot een afstand `far` van de camera positie.

5.6.2 Camera positie

De positie van de camera hoort bij de ruimtelijke definitie van het tafereel; deze wordt dus uitgedrukt met matrixmode GL_MODELVIEW.



De camera wordt geplaatst in het punt $P_0(x0, y0, z0)$ en is gericht naar het punt $P_{ref}(xref, yref, zref)$. De kijkrichting van de camera ligt hiermee vast; de oriëntatie niet (een camera kan om zijn viewrichting tollen) De view-up vector (vx, vy, vz) bepaalt welke richting omhoog is, d.i. de richting van onder naar boven van het viewing volume. Gewoonlijk is dit de y_{view}-richting, dus $(vx, vy, vx) = (0.0, 1.0, 0.0)$.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt( GLdouble x0, GLdouble y0, GLdouble z0,
           GLdouble xref, GLdouble yref, GLdouble zref,
           GLdouble vx, GLdouble vy, GLdouble vz);
```

Diepteberekening Bij de initialisatie van de displaymode moet aangegeven worden dat een tekenvenster moet voorzien worden met een dieptebuffer:

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH );
```

In het begin van de *display*functie moet naast de kleurbuffer ook de dieptebuffer geïnitialiseerd worden met een waarde tussen 0 en 1 gezet door glClearDepth (default 1.0):

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

In de displayfunctie zelf kan het al of niet zichtbaar zijn van voorwerpen getest worden op basis van de diepte van het voorwerp ten opzichte van de waarde in de dieptebuffer. Deze test kan geactiveerd of gedesactiveerd worden:

```
glEnable(GL_DEPTH_TEST);
...
glDisable(GL_DEPTH_TEST);
```

5.7 Voorbeeld

De verschillende projectiemethodes worden geïllustreerd, afhankelijk van het argument bij het opstarten. In de herschaal functie wordt de gepaste OpenGL functie opgeroepen:

```
1  void herschaal(GLint n_w, GLint n_h)
   {
3      GLdouble grens;
       glMatrixMode(GL_PROJECTION);
5      glLoadIdentity();
       switch ( projectie )
7      {
           case 'o':
9              glOrtho(xmin, xmax,  ymin, ymax, near, far);
               break;
11         case 'f':
               glFrustum(xmin, xmax,  ymin, ymax, 2*near, 8*far);
13             break;
           case 'F':
15             grens = near*tan(M_PI*(hoek/2.0)/180.0);
               glFrustum(-grens, grens,  -grens, grens, near, far);
17             break;
           default:
19         case 'p':
               gluPerspective(hoek, 1.0,  near, far);
21             break;
       }
23     glViewport(0, 0, n_w, n_h);
   }
```

Het tafereel bestaat uit een kubus met zijde 0.0, die wat verschoven is zodat één hoekpunt in de oorsprong zit. De hoofdassen worden ook getekend en eventueel kunnen de vluchtlijnen zichtbaar gemaakt worden.

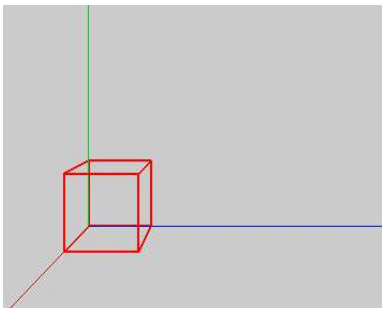
```
   void as(double lengte)
2  {
       if ( lengte > 2.0 )
4      {
           glColor3f(0.0, 0.1, 0.8);
6      }
       else if ( lengte < 2.0 )
8      {
           glColor3f(0.8, 0.1, 0.0);
10     }
       else
12     {
           glColor3f(0.0, 0.8, 0.1);
14     }
       glBegin(GL_LINES);
16         glVertex3d(0.0, 0.0, 0.0);
           glVertex3d(0.0, 0.0, lengte);
18     glEnd();
   }
20
   void kubus(void)
```

```
22  {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
24      glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
26      gluLookAt(xlens,ylens,zlens, xref,yref,zref, xvw, yvw, zvw);
        glLineWidth(1.5);
28      as(1.8);        // z-as
        glPushMatrix();
30      glRotatef(90.0, 0.0, 1.0, 0.0);
        as(2.5);        // x-as
32      glPopMatrix();
        glPushMatrix();
34      glRotatef(-90.0, 1.0, 0.0, 0.0);
        as(2.0);        // y-as
36      glPopMatrix();
        glLineWidth(2.5);
38      glPushMatrix();
        glColor3f(1.0, 0.0, 0.0);
40      glTranslatef(ttx, tty, ttz);
        glutWireCube(0.5);
42      if ( lijnen )
        {
44          glLineWidth(1.5);
            glLineStipple(2, 0xaaaa);
46          glEnable(GL_LINE_STIPPLE);
            glBegin(GL_LINES);
48              glVertex3d(0.25, 0.25, -0.25);
                glVertex3d(0.25, 0.25, -5.5);
50              glVertex3d(-0.25, 0.25, -0.25);
                glVertex3d(-0.25, 0.25, -5.5);
52          glEnd();
            glColor3f(0.0, 0.0, 1.0);
54          glBegin(GL_LINES);
                glVertex3d(-0.25, 0.25, -0.25);
56              glVertex3d(-5.25, 0.25, -0.25);
                glVertex3d(-0.25, 0.25, 0.25);
58              glVertex3d(-5.25, 0.25, 0.25);
            glEnd();
60          glColor3f(0.0, 1.0, 0.0);
            glBegin(GL_LINES);
62              glVertex3d(-0.25, -0.25, 0.25);
                glVertex3d(-0.25, -5.25, 0.25);
64              glVertex3d(0.25, -0.25, 0.25);
                glVertex3d(0.25, -5.25, 0.25);
66          glEnd();
            glDisable(GL_LINE_STIPPLE);
68      }
        glPopMatrix();
70      glFlush();
    }
```

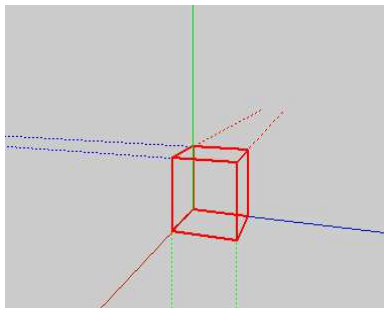
De camerapositie en -oriëntatie wordt gedefinieerd door de gluLookAt functie. Om de positie, het referentiepunt en de oriëntatie te kunnen wijzigen is er een callback functie voor input van

toetsenbord voorzien:

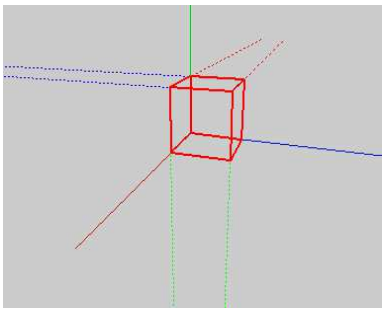
```
void toetsen( unsigned char key, int x, int y)
{
    switch ( key )
    {
        case 'x' : xlens++;  break;     case 'X' : xlens--;  break;
        case 'y' : ylens++;  break;     case 'Y' : ylens--;  break;
        case 'z' : zlens++;  break;     case 'Z' : zlens--;  break;
        case 'u' : xref++;   break;     case 'U' : xref--;   break;
        case 'v' : yref++;   break;     case 'V' : yref--;   break;
        case 'w' : zref++;   break;     case 'W' : zref--;   break;
        case 'j' : xvw = 1.0; yvw = zvw = 0.0;   break;
        case 'k' : yvw = 1.0; xvw = zvw = 0.0;   break;
        case 'l' : zvw = 1.0; yvw = xvw = 0.0;   break;
        case 'i' : xlens = ylens = zlens = 3.0;
                   xvw  = zvw  = 0.0;  yvw = 1.0;
                   xref = yref =  zref = 0.0;    break;
        case '0' : lijnen = !lijnen;          break;
        case 'q' : exit(0);                   break;
    }
    printf("Oog bl%5.1f gr%5.1f rd%5.1f ", xlens, ylens, zlens );
    printf("  Ref %5.1f %5.1f %5.1f ViR %4.1f %4.1f %4.1f\n",
                          xref, yref, zref, xvw,yvw,zvw);
    glutPostRedisplay();
}
```



$(1,1,3) \rightarrow (1,1,0)$
1 vluchtpunt



$(1,1,3) \rightarrow (0,1,0)$
2 vluchtpunten



$(1,1,3) \rightarrow (0,0,0)$
3 vluchtpunten

Figuur 5.1: Perspectief

6 Belichting

De toewijzing van de juiste kleur aan schermpixels is een essentiële stap in de rendering van 3D-taferelen. De perspectiefprojectie bepaalt **waar** een object op het scherm getoond wordt, de relatieve positie in de $z-$richting bepaalt de **zichtbaarheid** van objecten.
De kleuren van oppervlakken die we in realiteit waarnemen zijn het resultaat van een complexe interactie tussen licht en materie. Modelering van dit proces om een implementatie in software te realiseren mag niet leiden naar een al te zwaar wiskundig model. Er zijn twee benaderingswijzen om een belichtingsmodel op te stellen:

- empirisch: berekeningen in overeenstemming met de waarneming brengen, hierbij kunnen nogal wat vereenvoudigingen gedaan worden;
- gebaseerd op strikte wetten van de fysica (optica): levert vermoedelijk meer realistische beelden, maar wel tegen een rekenkundig hogere kost.

Ray-tracing en *path-tracing* algoritmes zijn gebaseerd op een exacte toepassing van de wetten van de fysica, maar vallen buiten de doelstellingen van deze cursus.
In een empirisch model wordt de kleur van 3D-objecten bepaald door twee parameters, materiaaleigenschappen en belichting. Uiteraard is *licht* een belangrijk ingrediënt van een belichtingsmodel. Zonder licht kan er niet gesproken worden over kleur en zouden alle pixels zwart ingekleurd worden. Anderzijds is het licht dat onze ogen bereikt, zelden rechtstreeks van een lichtbron afkomstig. We nemen vooral reflecties waar van lichtstralen op oppervlakken. Dus informatie over de *materiaaleigenschappen* van de objecten in het 3D-tafereel is net zo belangrijk bij het opstellen van een belichtingsmodel

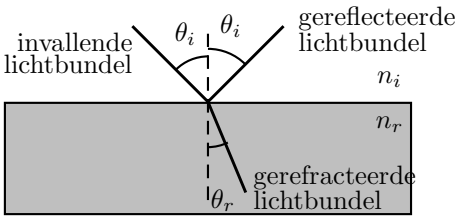
6.1 Materiaaleigenschappen

Het mechanisme van reflectie van licht door oppervlakken is zeer complex en is afhankelijk van een groot aantal parameters, bijvoorbeeld:

- *geometrie*: plaats van het oppervlak tov. lichtbron;
- *positie* van de waarnemer;
- *vorm* van het oppervlak;
- *karakteristieken* van het oppervlak: ruwheid, kleur, transparantie.

In eerste instantie worden in wat volgt, eenvoudige reflectiemodellen beschreven gebaseerd op achromatisch licht (grijstinten). Achromatisch licht wordt bepaald door de parameter **intensiteit**. In een tweede benadering wordt de component **kleur** toegevoegd, door de algoritmes toe te passen op elke hoofdkleur afzonderlijk.

Een lichtstraal die invalt op een oppervlak:
– wordt voor een deel geabsorbeerd door het oppervlak en omgezet in warmte;
– wordt voor een deel gereflecteerd;
– kan ook deels door het oppervlak heen gaan, zoals bijv. bij glas.



In het model in opbouw wordt alleen het gedeelte van het invallend licht dat gereflecteerd of verstrooid wordt, beschouwd. Een gedeelte van dit gereflecteerde licht bereikt de camera en laat toe dat het voorwerp zichtbaar wordt.
Het gedeelte van de lichtbundel dat door het oppervlak gaat wordt *afgebogen*. De hoek θ_r tussen de afgebogen lichtbundel en de normaal wordt bepaald door de wet van Snell: $n_i \cdot \sin\theta_i = n_r \sin\theta_r$ waarbij n_i en n_r de brekingsindices van de twee materialen zijn.
Er zijn twee soorten reflectie: diffuse verstrooiing (*diffuse*) en spiegelachtige reflectie (*specular*).

Vectoren bij reflectie van licht die in het model een rol spelen:

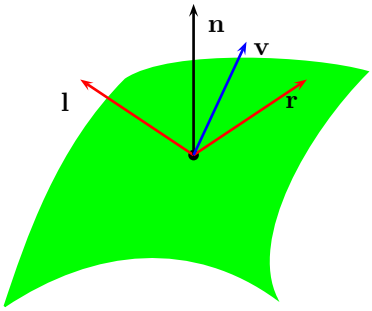
l oriëntatie van de lichtbron

n normaalvector

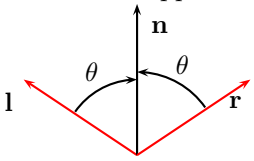
v viewvector : wijst naar de camera

r reflectie :

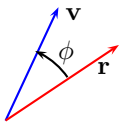
in het vlak (l, n): uitvalshoek = invalshoek



Hierbij zijn twee hoeken belangrijk:
hoek θ tussen invallende lichtstraal en normaal van het oppervlak

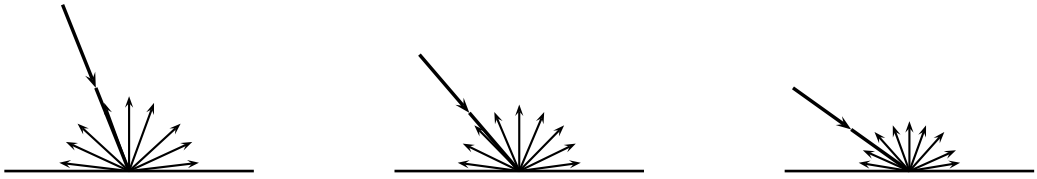


hoek ϕ tussen gereflecteerde lichtstraal en view-vector:



6.1.1 Diffuse reflection

Een ideale diffuse reflector toont op microscopisch niveau een erg ruw oppervlak, bijv. het oppervlak van een stukje krijt. Als gevolg van microscopische oneffenheden wordt een invallende lichtstraal in alle richtingen even sterk verstrooid. Uit metingen blijkt dat de intensiteit van het in alle richtingen gereflecteerde licht functie is van de *invalshoek* en de *intensiteit* van de lichtbron en bijgevolg onafhankelijk van de oriëntatie van de camera.



Deze intensiteit wordt bepaald door de cosinuswet van Lambert:

> Flux per unit solid angle leaving a surface in any direction is proportional to the cosine of the angle between that direction and the normal to the surface. A material that obeys Lambert's cosine law is said to be an isotropic diffuser; it has the same sterance (luminance, radiance) in all directions.

$$D = k_d I_d \cos\theta \qquad \text{of vectorieel} \qquad D = k_d \cdot I_d \cdot (\vec{l} \cdot \vec{n})$$

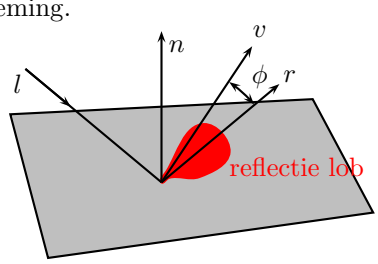
Hierin is I_d de intensiteit van het invallend licht en k_d de reflectiecoëffcient van het oppervlak. Aangezien een invalshoek groter dan 90 graden onmogelijk is door de geometrie van het oppervlak, worden enkel invalshoeken tussen 0 en 90 graden toegelaten.

6.1.2 Specular reflection

Een *specular reflector* (spiegelachtig) is een tweede type oppervlak: bij een glanzend oppervlak, zoals gepolijst metaal of het netjes gepolijste koetswerk van een wagen, zien we de reflectie van een lichtbron als een zeer heldere vlek op een relatief kleine zone, terwijl de rest van het object waargenomen wordt als een diffuse reflector.
Op microscopisch niveau is een specular reflector erg glad; de ideale reflector is een spiegel. Het reflecteerde licht is alleen zichtbaar wanneer de vectoren r en v samenvallen (hoek $\phi = 0$). De

meeste glanzende oppervlakken zijn geen ideale reflectoren. We gebruiken een empirisch model voor deze niet-ideale reflectoren dat consistent is met de waarneming.

Bij een ideale reflector geldt dat de hoek θ_i van de uitgaande straal t.o.v. de normaalvector exact gelijk is aan de hoek van de ingaande straal t.o.v. de normaalvector. In werkelijkheid zorgen microscopische afwijkingen op niveau van het oppervlak voor een zekere verstrooiing in een richting dichtbij de ideale reflectie.



Wanneer de hoek tussen de vectoren r en v groter wordt, neemt de intensiteit van het gereflecteerde licht snel af. Hiervoor wordt een zogenaamde *fall-off* factor (a) geïntroduceerd. De intensiteit van het gereflecteerde licht is evenredig met een macht a van het vectorieel product van vectoren v en r. De factor a is een maat voor de glans van een oppervlak en wordt *shininess* genoemd
Met S de intensiteit van het gereflecteerde licht, k_s de reflectiecoëfficiënt van het oppervlak en I_s de intensiteit van het invallende licht:

$$S = k_s \cdot I_s \cdot \cos^a \phi \qquad \text{of vectorieel} \qquad S = k_s \cdot I_s \cdot (\vec{v} \cdot \vec{r})^a$$

Hoe groter de waarde van de shininess factor a, hoe smaller de reflectie-lob rond de r-vector wordt. Wanneer de v-vector buiten deze lob licht, is er geen specular reflectie zichtbaar.

6.1.3 Ambient reflection

De reflectie is niet tengevolge van een invallende lichtbundel van een specifieke lichtbron maar is de reflectie door een oppervlak van omgevings- of achtergrondbelichting. Deze reflectie vertoont in *alle richtingen* dezelfde intensiteit, en geeft voor een oppervlak, onafhankelijk van de positionering tov. de lichtbronnen en camerapositie, voor elke pixel eenzelfde intensiteit:

$$A = k_a I_a$$

met A de intensiteit van het gereflecteerde licht, k_a de reflectiecoëfficiënt van het oppervlak en I_a de intensiteit van het omgevingslicht.

6.1.4 Het Phong belichtingsmodel

In een tafereel zijn meestal de verschillende vormen van reflectie aanwezig. De intensiteit voor elk punt van een oppervlak kan berekend worden door de individuele effecten bij elkaar op te tellen.

$$I = D + S + A \qquad \text{of} \qquad I = k_d \cdot I_d \cdot (\vec{l} \cdot \vec{n}) \; + \; k_s \cdot I_s \cdot (\vec{v} \cdot \vec{r})^a \; + \; k_a \cdot I_a$$

Bij chromatisch licht wordt deze berekening uitgevoerd voor elke hoofdkleur (rood, groen, blauw) en dit voor elke lichtbron die zich in het tafereel bevindt. Er zijn dus tien parameters om de materiaaleigenschappen van een oppervlak te definiëren:

$$k_{dr} \quad k_{dg} \quad k_{db} \qquad k_{sr} \quad k_{sg} \quad k_{sb} \qquad k_{ar} \quad k_{ag} \quad k_{ab} \qquad a$$

In OpenGL wordt hiervoor de functie glMaterial∗ gebruikt:

```
glMaterialfv(zijde, refectietype, reflectiecoef);
```

De eigenschappen van de voorkant en de achterkant van een oppervlak kunnen onafhankelijk van elkaar ingesteld worden via het eerste argument:

- GL_FRONT: reflectiecoëfficiënten enkel voor de *front face*: het naar de camera gericht oppervlak;
- GL_BACK: reflectiecoëfficiënten enkel voor de *back face*: de achterkant van zo'n vlak (normaal, maar niet altijd, onzichtbaar).

- GL_FRONT_AND_BACK: reflectiecoëfficiënten voor beide zijden.

Het tweede argument bepaalt het type van reflectiecoëfficiënt:

- GL_AMBIENT
- GL_DIFFUSE
- GL_SPECULAR
- GL_AMBIENT_AND_DIFFUSE
- GL_EMISSION: zie wat verder.

Het derde argument geeft in een vector de verschillende waarden van deze materiaaleigenschap in RGBA conventie.
Wanneer het tweede argument gelijk is aan GL_SHININESS, dan is het derde argument een enkelvoudige waarde, namelijk de shininess die bij specular reflectie een rol speelt.

```
GLfloat difmat[] = { 1.0, 0.0, 0.0, 1.0 };
GLfloat specmat[] = { 0.0, 1.0, 0.0, 1.0 };
GLfloat ambimat[] = { 0.0, 0.0, 1.0, 1.0 };
glMaterialfv( GL_FRONT, GL_DIFFUSE, difmat);
glMaterialfv( GL_BACK, GL_SPECULAR, specmat);
glMaterialf ( GL_BACK, GL_SHININESS, 20.0);
glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, ambimat);
```

Emissie : Een oppervlak kan zelf licht uitstralen, maar wordt geen lichtbron voor andere objecten in het tafereel. Zo'n oppervlak heeft dus geen effect op de rendering van andere objecten.

```
GLfloat emissie[] = { 0.5, 0.0, 1.0, 1.0 };
glMaterialfv( GL_FRONT, GL_EMISSION, emissie);
```

6.2 Lichtbronnen

Elk object in een 3D-tafereel is potentieel ook een lichtbron. Licht kan door een object uitgestraald of gereflecteerd worden. Wanneer een object licht uitstraalt, is het een lichtbron en wordt gekarakteriseerd door het soort licht dat uitgestraald wordt en de intensiteit van het uitgestraalde licht (per hoofdkleur). In vorig deel is beschreven dat reflecterende objecten gekarakteriseerd worden door materiaaleigenschappen zoals reflectiecoëfficiénten en shininess waarde.
Het Phong belichtingsmodel kent drie eenvoudige lichtbronnen:

1. achtergrondbelichting is een gevolg van gereflecteerd licht dat in de ruimte verstrooid wordt en op alle oppervlakken met eenzelfde intensiteit invalt, onafhankelijk van de positie en oriëntatie t.o.v. de lichtbronnen. Een oppervlak dat niet rechtstreeks door een lichtbron belicht wordt, is daardoor toch zwakjes zichtbaar. Bij de materiaaleigenschappen van een oppervlak kan voor elke hoofdkleur een reflectiecoëfficiënt voor *ambient* licht gedefinieerd worden. De intensiteit van deze achtergrondbelichting wordt meestal vrij laag gehouden.
2. lichtbron op oneindig, d.i. een verre lichtbron waarbij alle lichtstralen die het 3D-object bereiken, (in de limiet) evenwijdig met elkaar lopen. Bij zo'n *directional light source* moet enkel de richting waarin de bron zich bevindt en de stralingsintensiteit gespecificeerd worden. Een voorbeeld is zonlicht. De oriëntatie van de lichtbron t.o.v. de 3D-objecten speelt een rol bij de berekening van de *diffuse* en *specular* reflecties.
3. puntbron: de lichtstralen vertrekken radiaal vanuit de lichtbron; de lichtintensiteit is in alle richtingen dezelfde. Een klassieke gloeilamp is een goed voorbeeld. Licht dat vanuit een puntbron op een oppervlak invalt, maakt in elk punt van dit oppervlak een andere hoek met de normalvector. Het vectorieel product $\vec{l} \cdot \vec{n}$ moet dus in elk punt van het oppervlak berekend worden. De intensiteit van een puntbron hangt af van de afstand tot het belichte object, benaderend omgekeerd evenredig met het kwadraat van de afstand.

OpenGL kent acht lichtbronnen: `GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7` die apart kunnen ingeschakeld of uitgeschakeld worden:

```
glEnable(GL_LIGHT0);          glDisable(GL_LIGHT0);
```

Elke lichtbron heeft een aantal eigenschappen met elk een default instelling. Deze instelling kan gewijzigd worden:

```
glLight* ( lichtbron, lichteigenschap, waarde);
```

Positie wordt ingesteld met de eigenschap `GL_POSITION`. Het argument `waarde` is een vector met 4 elementen:

- x-, y-, z-coördinaat;
- al of niet puntbron:
  - 4^e argument: 0.0 : bron op oneindig: de coördinaten geven een richting aan, alle lichtstralen lopen evenwijdig aan deze richting;
  - 4^e argument: 1.0 : puntbron bepaald door de coördinaten (eerste 3 elementen).

De vector met coördinaten die de bronpositie van een lichtbron bepalen, wordt onderworpen aan de *modelview* matrix die geldt op het ogenblik van de uitvoering van de functie `glLight()`.

– voor de bepaling van de positie van het oog:

```
glLightfv ( GL_LIGHT0, GL_POSITION, positie0 );
gluLookAt(oog[0], oog[1], oog[2], 0.0,0.0,0.0, 0.0,1.0,0.0 );
```

de positie van de lichtbron ligt nu vast relatief t.o.v. het oog: wanneer het oog verplaatst wordt, wordt de lichtbron mee verplaatst (te vergelijken met een lamp op een helm van een mijnwerker);

– na de bepaling van de positie van het oog:

```
gluLookAt(oog[0], oog[1], oog[2], 0.0,0.0,0.0, 0.0,1.0,0.0 );
glLightfv ( GL_LIGHT0, GL_POSITION, positie0 );
```

de positie van de lichtbron ligt vast in het tafereel: wanneer het oog verplaatst wordt, wijzigt de hoek ϕ tussen de view-vector en de gereflecteerde lichtbundel.

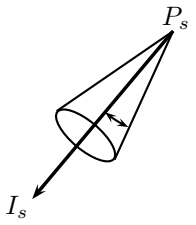
Lichttype aangeven door bij het tweede argment één van volgende waarden te kiezen:

- `GL_AMBIENT`: ambient licht: licht dat zo erg verstrooid is door de omgeving dat het onmogelijk is om nog de richting ervan te bepalen; een voorbeeld is achtergrondverlichting in een kamer; ambient licht dat op een oppervlak valt, wordt in alle richtingen evenveel verstrooid;
- `GL_DIFFUSE`: diffuse component van licht: komt van één bepaalde richting; het is helderder wanneer het ongeveer loodrecht op een oppervlak valt dan wanneer het er meer zijdelings opvalt; invallend diffuus licht op een oppervlak wordt ook in alle richtingen evenveel verstrooid;
- `GL_SPECULAR`: specular licht komt vanuit een specifieke richting en wordt door een oppervlak in een voorkeurrichting gereflecteerd.

Het derde argument geeft de kleur van de lichtbron door per hoofdkleur de intensiteit aan te geven in RGBA-formaat. De *blending factor A* kan gebruikt worden om twee kleuren een zekere transparantie te geven zodat ze op het scherm schijnbaar in elkaar kunnen overvloeien.

```
GLfloat diflicht[] = { 1.0, 0.0, 0.0, 1.0 };
GLfloat speclicht[] = { 0.0, 1.0, 0.0, 1.0 };
GLfloat ambilicht[] = { 0.0, 0.0, 1.0, 1.0 };

glLightfv( GL_LIGHT0, GL_DIFFUSE, diflicht );
glLightfv( GL_LIGHT0, GL_SPECULAR, speclicht );
glLightfv( GL_LIGHT0, GL_AMBIENT, ambilicht );
```

Spot : een richtingsgevoelige lichtbron.
In reële situaties wordt vaak spot-verlichting gebruikt in plaats van een puntbron. Een spot gedraagt zich als een puntbron maar met een stralingskegel. Er moeten drie parameters gespecificeerd worden:



1. de oriëntatie (vectorieel);
2. de openingshoek van de stralingskegel;
3. de intensiteitsdistributie over de stralingskegel.

Buiten de stralingskegel levert de spot geen enkele bijdrage tot de belichting van een tafereel.
Lichtstralen vertrekken binnen een kegel met een gegeven openingshoek(`GL_SPOT_CUTOFF`) vanuit de lichtbron in een gegeven richting (`GL_SPOT_DIRECTION`). De derde parameter geeft aan hoe geconcentreerd het licht is (*falloff-functie*): de intensiteit is het hoogste in het midden van de kegel en deze verzwakt naar de randen toe met een factor gelijk aan $\cos^a \theta$ met a de `GL_SPOT_EXPONENT` (default gelijk aan 0.0) en θ de hoek tussen de vertex die de richting aangeeft en de vertex van de actuele positie.

```
GLfloat richting[] = { -1.0, -1.0, 0.0 };
glLightf ( GL_LIGHT0, GL_SPOT_CUTOFF, 30.0);
glLightfv( GL_LIGHT0, GL_SPOT_DIRECTION, richting);
glLightf ( GL_LIGHT0, GL_SPOT_EXPONENT, 12.0);
```

De cutoff parameter ligt tussen 0 en 90 graden en daarnaast is er de default waarde 180 graden. De default richting is in de richting van de negatieve z-as (0.0, 0.0, -1.0). Merk op dat de spotrichting onderworpen wordt aan de modelview matrix die geldt op het ogenblik van de uitvoering van de functie glLight().

Verzwakking. De intensiteit waarmee een waarnemer een lichtbron waarneemt, is afhankelijk van de afstand van die waarnemer tot de lichtbron. De intensiteit van licht neemt af naarmate dat de afstand tot de bron verhoogt en op basis van empirische vaststellingen is dit een kwadratisch verband.
In OpenGL kan naast een kwadratische, ook een constante en lineaire verzwakkingsparameter ingesteld worden. Deze verzwakking geldt alleen voor *puntbronnen*, omdat gericht licht van oneindig ver verwijderde bronnen afkomstig is, en wordt als volgt berekend:

$$\text{verzwakking} = \frac{1}{k_c + k_l d + k_q d^2}$$

met d de afstand tussen lichtbron en het beschouwde oppervlak
Default waarden zijn 1 voor k_c (constante) en 0 voor k_l (lineaire) en k_q (kwadratische parameter), zodat er eigenlijk geen verzwakking in rekening gebracht wordt. Instellen van andere waarde kan als volgt:

```
glLightf( GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.0);
glLightf( GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.5);
glLightf( GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 1.0);
```

Global parameters die onafhankelijk van de lichtbronnen gelden voor het hele belichtingsmodel:

1. een *ambient achtergrondbelichting* onafhankelijk van een specifieke bron;
2. berekeningswijze van *specular* reflectiehoeken;
3. specular kleuren afzonderlijk berekenen van ambient en diffuse kleuren;
4. worden de belichtingsberekeningen uitgevoerd voor zowel voor- als achterkant van objecten.

Deze parameters worden ingesteld met de glLightModel*() functie:

```
GLfloat glambi[] = { 0.3, 0.1, 0.4, 1.0 };
glLightModelfv( GL_LIGHT_MODEL_AMBIENT, glambi);
glLightModeli ( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
glLightModeli ( GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR);
glLightModeli ( GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

6.3 Polygoon rendering

Voor een willekeurig punt van een oppervlak wordt volgende berekening uitgevoerd om de kleur van die vertex te bepalen:

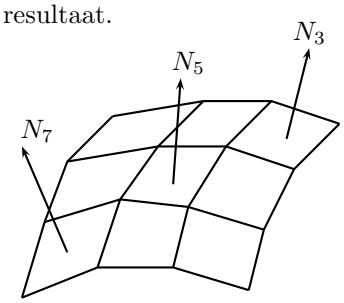
$$\begin{aligned}\text{vertex kleur} = \quad & \text{emissie}_{\text{materiaal}} + \quad \text{ambient}_{\text{lichtmodel}} \times \text{ambient}_{\text{materiaal}} \\ & + \quad \sum_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i \times \text{spoteffect}_i \quad \times \\ & \left[\text{ambient}_{\text{licht}} \times \text{ambient}_{\text{materiaal}} \quad + \right. \\ & \max(l.n, 0) \times \text{diffuus}_{\text{licht}} \times \text{diffuus}_{\text{materiaal}} \quad + \\ & \left. \max(v.r, 0)^{\text{shininess}} \times \text{specular}_{\text{licht}} \times \text{specular}_{\text{materiaal}} \right]_i \end{aligned} \tag{1}$$

$$\text{spoteffect} = \begin{cases} 1.0 & \text{bij cutoff gelijk aan } 180.0 \\ 0.0 & \text{vertex ligt buiten de kegel} \\ \max(l.d, 0)^a & \text{met } d \text{ richting} \end{cases}$$

Omdat OpenGL *state driven* is, moet deze berekening geactiveerd worden:

```
glEnable(GL_LIGHTING);
glEnable(GL_AUTO_NORMAL);              /* generatie normaalvectoren */
glEnable(GL_NORMALIZE);              /* normalisatie normaalvectoren */
glEnable(GL_COLOR_MATERIAL);        /* geen specifieke materiaaleig */
glColorMaterial(GL_FRONT, GL_DIFFUSE);     /* kleur -> diffuse eig.*/
```

Bij de berekening van de kleur van een vertex moet de bijhorende normaalvector gekend zijn. Tussen een glBegin en glEnd sequentie kan bij elke vertex een normaalvector gespecificeerd worden met de glNormal∗() functie. Voor de standaard oppervlakken die met glu- of glutfuncties gegenereerd worden, kan OpenGL de nodige normaalvectoren automatisch genereren. Deze normaalvectoren kunnen best genormaliseerd worden zodat ze een lengte gelijk aan 1 hebben.
In plaats van materiaaleigenschappen te specificeren kan aangegeven worden dat klassieke kleuren (gezet met glColor∗) gebruikt worden voor de waarden van deze materiaaleigenschappen. Met glColorMaterial wordt aangegeven voor welke face (`GL_FRONT` of `GL_BACK`) de klassieke kleur als één van de drie types materiaaleigenschappen (`GL_AMBIENT`, `GL_DIFFUSE` of `GL_SPECULAR`) gebruikt wordt.
Het algoritmisch proces dat het belichtingsmodel toepast op het *volledige* oppervlak i.p.v. één enkel punt wordt shading genoemd. Het is uiteraard onmogelijk om bovenstaande berekening uit te voeren op alle punten van een oppervlak. Er zijn verschillende vereenvoudigingen gedefinieerd die van elkaar verschillen in het gemaakte compromis tussen vereiste rekencapaciteit en het visuele resultaat.



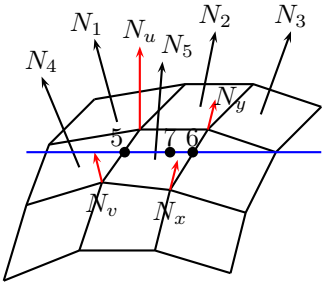
Een willekeurig complex oppervlak wordt door een grafisch systeem gemanipuleerd in de vorm van een *mesh*.

patches, de elementen van deze mesh, zijn meetkundige vlakken met telkens één normaalvector N_x

verfijnen van de mesh geeft kleinere patches zodat het complexe oppervlak dichter benaderd wordt, maar met een grotere rekenintensiteit

flat shading : het belichtingsmodel wordt per patch slechts één maal toegepast, omdat de normaalvector over de volledige patch constant blijft. De resulterende kleurintensiteit wordt op de *volledige patch* toegepast. Dit is een zeer eenvoudig shading model. De vereiste rekencapaciteit is minimaal, maar het resultaat oogt visueel niet erg fraai. Deze snelle techniek wordt gebruikt om bij de opbouw van complexe taferelen een voorlopige rendering te berekenen. Men spreekt van *constant-intensity surface rendering*.

Gouraud shading geeft een meer realistische voorstelling door middel van *interpolatietechnieken*. Henri Gouraud (1971) stelde een algoritme voor om enkel de intensiteiten per patch te berekenen ter hoogte van de vertices om deze intensiteiten daarna te interpoleren over deze patch.



1. Bepaal voor alle vertices van een patch van de mesh de gemiddelde normaalvector: bijvoorbeeld N_u wordt berekend op basis van N_1, N_2, N_4 en N_5.
2. In elk van deze vertices wordt de intensiteit volgens formule (1) berekend. Bijvoorbeeld voor vertex u wordt met behulp van de gemiddelde normaalvector N_u de intensiteit I_u berekend.
3. Interpoleer lineair de vertex-intensiteiten over de patch.

Om de intensiteit te berekenen in punt 7 op de scan-lijn, wordt eerst de genormaliseerde normaalvectoren voor de vertices x, u, x en y berekend:

$$N_u = \frac{N_1 + N_2 + N_4 + N_5}{|N_1 + N_2 + N_4 + N_5|} \qquad N_v = \frac{N_4 + N_5 + N_7 + N_8}{|N_4 + N_5 + N_7 + N_8|}$$

$$N_x = \frac{N_5 + N_6 + N_8 + N_9}{|N_5 + N_6 + N_8 + N_9|} \qquad N_y = \frac{N_2 + N_3 + N_5 + N_6}{|N_2 + N_3 + N_5 + N_6|}$$

In elk van deze vertices wordt formule (1) toegepast. Dit resulteert in de intensiteiten I_u, I_v, I_x en I_y.
Intensiteit I_5 in punt 5 is een lineaire interpolatie van de intensiteiten I_u en I_v. Intensiteit I_6 in punt 6 is een lineaire interpolatie van de intensiteiten I_x en I_y. Tot slot wordt de intensiteit I_7 in punt 7 berekend door een lineaire interpolatie van de intensiteiten I_5 en I_6.
Deze lineaire interpolatie vraagt veel minder rekenwerk dan wanneer voor elk gerasterd punt eerst de werkelijke normaalvector berekend wordt om vervolgens met deze normaalvector de kleur te berekenen met formule (1). Het model wordt ook *intensity interpolation surface rendering* genoemd.
OpenGl ondersteunt zowel flat shading als Gouraud shading:

```
glShadeModel(GL_FLAT);          /* flat shading */
glShadeModel(GL_SMOOTH);        /* Gouraud shading */
```

Phong shading. Bij de vorige shading methode zijn er op niveau van de aansluiting van patches nog discontinuïteiten in de intensiteiten zichtbaar. Phong lost dit probleem op door i.p.v. de intensiteiten de normaalvectoren te interpoleren (*normalvector interpolation surface rendering*):

1. bepaal voor alle vertices van het oppervlak een gemiddelde normaalvector;
2. interpoleer lineair de normaalvectoren over de patch;
3. bereken op alle punten op de scan-lijn de intensiteit volgens formule (1) op basis van de geïnterpoleerde normaalvectoren.

Phong shading levert resultaten op die aanzienlijk beter zijn dan die bekomen met de methode van Gouraud, echter ten koste van een veel rekenintensiever algortime. Deze shading methode wordt niet ondersteund door OpenGL.

7 Kleurenvariaties

7.1 Opacity/blending

7.1.1 Verwijderen van verborgen oppervlakken

Tot nu wordt het beeld gevormd door objecten met oppervlakken die *ondoorzichtig* zijn. Delen van objecten die door andere objecten verborgen zijn, worden geëlimineerd door middel van een *hidden-surface removal* algoritme. Dit algortime gebruikt de informatie die in de z-buffer (diepte-buffer) aanwezig is. Deze diepte-buffer moet bij de initialisatie (in main) voorzien worden via een extra GLUT-parameter (een extra bit in glutInitDisplayMode):

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH );
```

Net zoals de frame-buffer met kleur-informatie moet deze diepte-buffer in het begin van de teken functie geïnitialiseerd worden door elk element in de buffer een waarde te geven die overeenkomt met de verst mogelijke afstand:

```
glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
```

De default waarde van de verst mogelijke afstand is 1.0. Maar deze kan gewijzigd worden:

```
glClearDEPTH(0.8);
```

Het argument is een waarde tussen 0.0 en 1.0.
Hidden-surface removal moet geactiveerd worden

```
glEnable(GL_DEPTH_TEST);
```

en gebeurt als volgt:

- bij het berekenen van de schermcoördinaten wordt ook de diepte bepaald (zie vorig hoofdstuk);
- deze diepte wordt vergeleken met de dieptewaarde die voor deze pixel gestockeerd is in de diepte-buffer;
- indien de pixel dichterbij is, worden de nieuwe waarden gestockeerd: de kleur-informatie in de frame-buffer, de diepte-informatie in de diepte-buffer;
- indien de pixel verderaf is, worden de nieuwe waarden genegeerd.

7.1.2 Doorzichtige objecten

Het is mogelijk om een voorwerp met een doorzichtig oppervlak op te nemen in het tafereel. Achterliggende voorwerpen zijn dan niet onzichtbaar. De resulterende kleur die in de frame-buffer geplaatst wordt, is een menging van de kleur van het vooraan gelegen doorzichtige oppervlak en de kleur van het achterliggende (ondoorzichtige) object. Dit wordt *alpha blending* genoemd omdat de mate van menging bepaald wordt de *alpha factor*, het vierde getal bij de specificatie van RGBA waarden:

- ondoorzichtig oppervlak: alle licht wordt tegengehouden, $\alpha = 1.0$
- transparant oppervlak: alle licht wordt doorgelaten, $\alpha = 0.0$
- waarde van α tussen 0 en 1: oppervlak is gedeeltelijk doorzichtig, $(1 - \alpha)$

De rendering van veelvlakken gebeurt in OpenGL één per één. Bij het verwerken van een volgend fragment met pixels $\mathbf{s} = [s_r, s_g, s_b, s_a]$ (de bronpixels) zijn er reeds pixelwaarden in de frame-buffer ingevuld, namelijk $\mathbf{d} = [d_r, d_g, d_b, d_a]$ (de bestemmingspixels). Wanneer het vooraanliggende veelvlak doorzichtig is, worden de nieuwe waarden voor de frame-buffer berekend door de bronpixels te *mengen* met de bestemmmingspixels.

$$\mathbf{d}' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_a s_a + c_a d_a]$$

met een *bron-blending* factor $\mathbf{b} = [b_r, b_g, b_b, b_a]$ en een *bestemming-blending* factor $\mathbf{c} = [c_r, c_g, c_b, c_a]$. De default operatie bij het mengen is optellen, maar andere bewerkingen kunnen ingesteld worden. Indien de berekening in d' resulteert in getallen groter dan 1.0, dan wordt deze waarde 1.0; negatieve getallen worden ge-clamp-ed naar 0.0.

7.1.3 OpenGL functies

Specificeren hoe de blending factoren bepaald worden, kan met twee functies:

```
glBlendFunc(bronfactor, bestemmingfactor);
glBlendFuncSeparate(bronf_RGB, bestemmingf_RGB,
                    bronf_alfa, bestemmingf_alfa);
glBlendColor(rood, groen, blauw, alfa);
```

Met de tweede functie kunnen verschillende factoren vastgelegd worden voor enerzijds de RGB waarden en anderzijds de α waarden.
Mogelijke factoren die als argument in de functies kunnen gebruikt worden:

	blend factor	
bron-/bestemmingsfactor	RGB	alpha
GL_ZERO	(0,0,0)	0
GL_ONE	(1,1,1)	1
GL_SRC_COLOR	(s_r, s_g, s_b)	s_a
GL_ONE_MINUS_SRC_COLOR	$(1,1,1) - (s_r, s_g, s_b)$	$1 - s_a$
GL_DST_COLOR	(d_r, d_g, d_b)	d_a
GL_ONE_MINUS_DST_COLOR	$(1,1,1) - (d_r, d_g, d_b)$	$1 - d_a$
GL_SRC_ALPHA	(s_a, s_a, s_a)	s_a
GL_ONE_MINUS_SRC_ALPHA	$(1,1,1) - (s_a, s_a, s_a)$	$1 - s_a$
GL_DST_ALPHA	(d_a, d_a, d_a)	d_a
GL_ONE_MINUS_DST_ALPHA	$(1,1,1) - (d_a, d_a, d_a)$	$1 - d_a$
GL_CONSTANT_COLOR	(c_r, c_g, c_b)	c_a
GL_ONE_MINUS_CONSTANT_COLOR	$(1,1,1) - (c_r, c_g, c_b)$	$1 - c_a$
GL_CONSTANT_ALPHA	(c_a, c_a, c_a)	c_a
GL_ONE_MINUSCONSTANT_ALPHA	$(1,1,1) - (c_a, c_a, c_a)$	$1 - c_a$
GL_SRC_ALPHA_SATURATE	(f, f, f) met $f = \min(s_a, 1 - d_a)$	1

De hoger vermelde glBlendColor functie specificeert een *constante kleur* voor gebruik bij `GL_CONSTANT` blending factoren (vanaf versie 1.4).

Voorbeeld: kegel met bronpixels $\mathbf{s} = [0.6, 0.8, 0.3, 0.4]$ en bol met bestemmingspixels $\mathbf{d} = [0.7, 0.7, 0.1, 0.8]$

geval 3: GL_SRC_COLOR $\mathbf{b} = [0.6, 0.8, 0.3, 0.4]$
GL_DST_COLOR $\mathbf{c} = [0.7, 0.7, 0.1, 0.8]$
$\mathbf{d}' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_a s_a + c_a d_a]$
$\mathbf{d}' = [0.49 + 0.36, 0.49 + 0.64, 0.01 + 0.09, 0.16 + 0.64]$ of $[0.85, 1.0, 0.10, 0.80]$
geval 4: GL_SRC_ALPHA $\mathbf{b} = [0.4, 0.4, 0.4, 0.4]$
GL_DST_ALPHA $\mathbf{c} = [0.8, 0.8, 0.8, 0.8]$
$\mathbf{d}' = [0.24 + 0.56, 0.32 + 0.56, 0.12 + 0.08, 0.16 + 0.64]$ of $[0.80, 0.88, 0.20, 0.80]$

Het in- en uitschakelen van blending berekeningen:

```
glEnable(GL_BLEND);
glDisable(GL_BLEND);
```

Hiermee is het mogelijk om in bepaalde gedeeltes van het tafereel doorzichtigheid in rekening te brengen en in andere gedeeltes niet.

In plaats van bron- en bestemmingskleuren op te tellen, kan een andere bewerking gekozen worden:

```
glBlendEquation(mode);
```

mogelijke waarden voor `mode`:

GL_FUNC_ADD	$b.s + c.d$
GL_FUNC_SUBTRACT	$b.s - c.d$
GL_FUNC_REVERSE_SUBTRACT	$c.d - b.s$
GL_MIN	$\min(b.s, c.d)$
GL_MAX	$\max(b.s, c.d)$

7.1.4 3D blending met diepte-buffer

Een tafereel kan zowel ondoorzichtige als doorzichtige objecten bevatten. Het renderen maakt gebruik van de diepte-buffer:

- hidden-surface removal voor objecten achter ondoorzichtige objecten;
- kleurmenging van doorzichtig object met achterliggend object.

De volgorde waarin de objecten getekend worden is belangrijk, maar de juiste volgorde is gemakkelijk realiseerbaar in een statisch tafereel. Wanneer objecten of het viewpunt beweegt is dat moeilijker: soms zit het doozichtig object voor het ondoorzichtig voorwerp en soms zit het er achter.
Oplossing:

- teken de ondoorzichtige objecten eerst (met normale werking van diepte-buffer);
- maak diepte-buffer *read-only* (diepte-waarden niet aanpasbaar);
- teken de doorzichtige objecten, waarbij diepte-waarden vergeleken worden met die in de diepte-buffer:
  - wanneer gelegen achter ondoorzichtige objecten: niet tekenen;
  - wanneer gelegen voor ondoorzichtige objecten: diepte-waarden niet aanpassen maar menging van doorzichtig met achterliggend object.

Waarden in diepte-buffer alleen raadpleegbaar of aanpasbaar maken:

```
glDepthMask(GL_FALSE);   // elementen in diepte-buffer raadpleegbaar
glDepthMask(GL_TRUE);    // diepte waarden kunnen aangepast worden
```

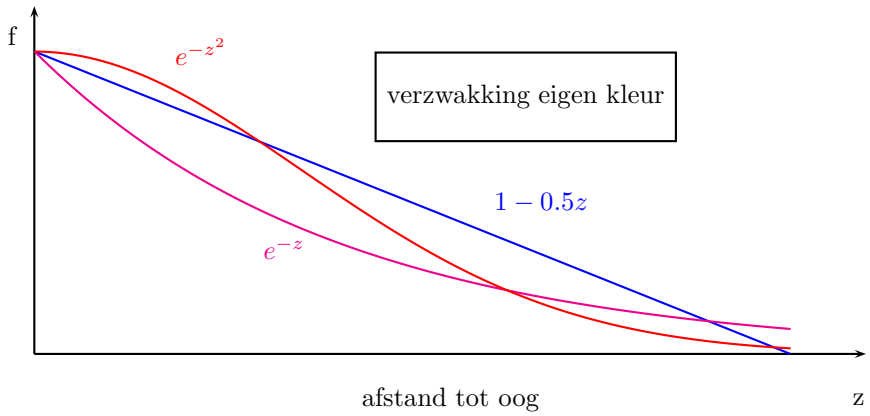
7.2 Fog

En bijzondere vorm van kleurmenging is het nabootsen van mist of vervuilde lucht. De kleur van het voorwerp wordt gemengd met de kleur van de lucht tussen het oog en het voorwerp. Hoe verder het voorwerp van het oog verwijderd is, hoe *minder scherp* het voorwerp getekend wordt. Bij deze kleurmenging evolueren de kleuren in het tafereel naar eenzelfde *fog* kleur in functie van de afstand tot het oog (z).

Resulterende kleur: $C_{s'} = fC_s + (1-f)C_f$ met $\begin{cases} C_s & \text{fragmentkleur} \\ C_f & \text{fogkleur} \\ f & \text{fog factor} \end{cases}$

Er zijn verschillende mogelijkheden voor het berekenen van de *fog factor* f :

$$\begin{cases} e^{-(density \times z)} & \text{exponentieel} & \texttt{GL\_EXP} \\ e^{-(density \times z)^2} & \text{kwadratisch exponentieel} & \texttt{GL\_EXP2} \\ \frac{eind - z}{eind - start} & \text{lineair} & \texttt{GL\_LINEAR} \end{cases}$$



OpenGL functies

In- en uitschakelen van deze toestand:

```
glEnable(GL_FOG);
glDisable(GL_FOG);
```

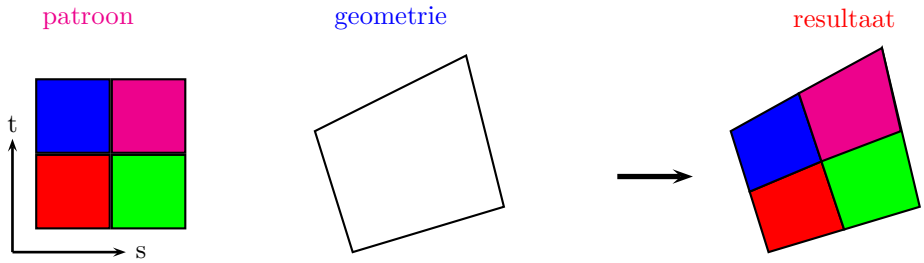
Specificeren van *fog elementen*:

```
glFloat kleur[4] = { 0.6, 0.5. 0.4, 1.0 };
glFogfv(GL_FOG_COLOR, kleur);
glFogf(GL_FOG_MODE, GL_EXP);      // of GL_EXP2  of GL_LINEAR
glFogf(GL_FOG_DENSITY, 0.4);      // bij (kwadratisch) exponentieel
glFogf(GL_FOG_START, 0.0);        // bij lineair
glFogf(GL_FOG_END, 10.0);
```

7.3 Texture Mapping

7.3.1 Inleiding

In de voorbije hoofdstukken is aangegeven hoe geometrische primitieven getekend kunnen worden met een uniform kleuroppervlak of met een menging van de kleuren van de samenstellende vlakjes. Om tot een meer realistische weergave van een voorwerp te komen kan texture mapping toegepast worden. Een *textuur* is een kleurenpatroon (bijv. een prentje, een foto) dat gebruikt wordt om elke veelhoek van het getekende voorwerp een kleuring te geven.
Het textuur patroon $T(s,t)$ is een rechthoekige, twee-dimensionale array waarbij de onafhankelijke variabelen s en t *textuurcoördinaten* genoemd worden. Zonder aan algemeenheid te verliezen kan men aannemen dat deze coördinaten variëren in het interval $[0,1]$.



Hiervoor moet een *mapping* gedefinieerd worden om textuur elementen (*texels*) te koppelen met de punten (*pixels*) van het voorwerp. Bij deze specificatie moeten een aantal keuzes gemaakt worden,

waardoor texture mapping een nogal complex onderwerp is. Volgende stappen moeten uitgevoerd worden:

1. creatie van een textuurobject en de bijhorende textuur specificeren;
2. aangeven hoe de textuur toegepast wordt op de pixels van het voorwerp;
3. texture mapping aanzetten (enable);
4. tekenen van het voorwerp waarbij zowel de textuur- als de geometrische coördinaten aangegeven worden.

7.3.2 Creatie

Textuur objecten. In OpenGL is er een enkele *huidige textuur* van elk type (1D, 2D, 3D) dat deel uitmaakt van de OpenGL toestand. Wanneer er meerdere texturen gebruikt worden, kan glTexImage2D opgeroepen worden telkens een andere textuur gebruikt wordt. Dit is echter niet efficiënt, omdat bij iedere oproep een ander textuur patroon geladen wordt in het textuurgeheugen, waarbij de reeds aanwezige texels overschreven worden, zelfs indien er nog voldoende vrij textuurgeheugen beschikbaar is.
Vanaf OpenGL 1.1 is het mogelijk om textuur objecten te defiëren elk met zijn eigen textuurmatrix en eigen textuurparameters die de mapping op het voorwerp controleren. Bij voldoende groot textuurgeheugen zijn deze objecten tegelijk in het geheugen aanwezig.
Het creëren van een aantal textuurobjecten (bijv. 5):

```
GLsizei aantal=5;
GLuint texobj[5];
glGenTextures( aantal, texobj );
```

Na deze oproep bevat texobj vijf verschillende gehele getallen die elk naar een textuurobject verwijzen.
Eén van deze textuurobjecten de *actuele* textuur, onderdeel van de OpenGL toestand maken, kan als volgt:

```
glBindTexture( GL_TEXTURE_2D, texobj[2] );
```

Hierdoor wordt het derde object *gekozen* waar vanaf nu mee zal gewerkt worden bij de verschillende textuur manipulaties.
Wanneer een specifieke textuur niet meer gebruikt wordt, kan deze best vrijgegeven worden zodat de ingenomen plaats in het textuurgeheugen beschikbaar komt voor eventueel een andere textuur.

```
glDeleteTextures( 1, &texobj[2] );
```

Textuurarray. Een onderdeel van een textuurobject is de effectieve data waaruit de texels gehaald worden om de kleur van de pixels te bepalen.
Bij 2D texturing is de data een drie-dimensionale array waarbij de derde dimensie de RGBA-waarden geeft voor één specifieke texel. De waarden in deze array kunnen in het programma genereereerd worden, of van een bestand ingelezen worden.

```
#define HORI 16
#define VERT 16
GLubyte texels[HORI][VERT][4];
int k, l;
memset(texels, 16, HORI*VERT*4);
for( k=0; k<HORI/2; k+=2 )
{
    for( l=0; l<VERT/2; l+=2 )
    {
        texels[2*k][2*l][1] = texels[2*k][2*l+1][1] = 255;
        texels[2*k+1][2*l][1] = texels[2*k+1][2*l+1][1] = 255;
```

```
        }
    }
    for( k=1; k<HORI/2; k+=2 )
    {
        for( l=1; l<VERT/2; l+=2 )
        {
            texels[2*k][2*l][0] = texels[2*k][2*l+1][0] = 255;
            texels[2*k+1][2*l][0] = texels[2*k+1][2*l+1][0] = 255;
        }
    }
```

Door ook de alfa-waarden voor eventuele doorzichtigheid op te nemen, worden de verschillende texels op veelvouden van 4 gealigneerd, want nooit kwaad kan.
Aangeven dat deze array als twee-dimensionale textuur bij een bepaald textuurobject gebruikt wordt:

```
glBindTexture( GL_TEXTURE_2D, texobj[2] );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, HORI, VERT, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, texels );
```

Het tweede argument is gelijk aan 0 bij eenvoudige toepassingen. Het derde argument (GL_RGBA) geeft aan welke componenten uit de data geselecteerd worden voor de texels. Vierde (breedte) en vijfde argument (hoogte) geven de dimensies van de data; het zesde argument geeft de breedte (b) van de rand (wanneer er geen rand is, is dit argument gelijk aan 0). In de eerste versies van OpenGL moesten breedte en hoogte van de vorm $2^m + 2b$ zijn. Omwille van performatie is het nog steeds aan te raden om hiervoor deze vorm te hebben. De twee volgende argumenten geven het formaat en het data type van de invoerarray (laatste argument).

Vanuit een bestand. In plaats van de array met texels programmatorisch te genereren, kan deze texture data ook ingelezen worden vanuit een binair bestand.
Om bijvoorbeeld de data uit een JPEG bestand te lezen en deze als twee-dimensionale textuur bij een bepaald textuurobject te gebruiken kan volgend code fragment gebruikt worden :

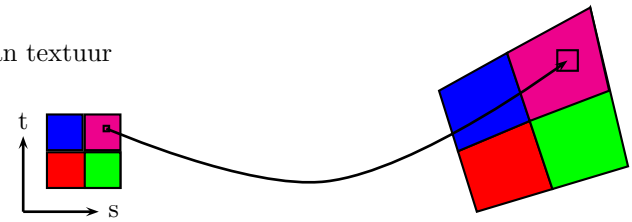
```
#include "InitJPG.h"
tImageJPG *pImage;
pImage = LoadJPG("roos.jpg");
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glBindTexture(GL_TEXTURE_2D, texobj[3]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
                pImage->sizeX, pImage->sizeY, 0,
                GL_RGB, GL_UNSIGNED_BYTE, pImage->data);
```

Het headerbestand InitJPG.h bevat enkele type-definities, constanten en prototypes van functies om met JPEG bestanden te werken. Bij het linken moet het object bestand InitJPG.o en een `jpeg` bibliotheek opgegeven worden.
De argumenten van glTexImage2D zijn eerder al uitgelegd: specificatie van level (normaal 0) en intern formaat van wat geselecteerd wordt uit de data; naast breedte en hoogte wordt aangegeven of er een rand is (hier 0); formaat en type beschrijven de textuur beelddata.

7.3.3 Filtering

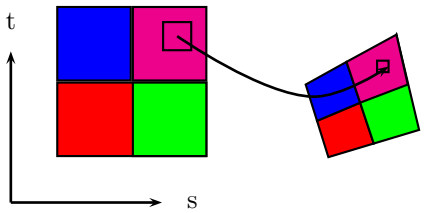
Een textuur is rechthoekig of eventueel vierkantig; maar de oppervlakken van de te tekenen voorwerpen zijn dit niet noodzakelijk. Daarnaast is de grootte van de textuur niet noodzakelijk gelijk aan de geometrie waarop gemapt wordt. Eén pixel op het scherm correspondert dus niet met één texel van de textuur. Er zijn twee mogelijkheden van correspondentie:

- magnification:
  geometrie grootte is groter dan die van textuur
  een pixel op het scherm komt
  overeen met slechts een
  klein deel van een texel



- minification:

  geometrie grootte is kleiner dan die van textuur:
  een pixel op het scherm komt overeen met een aantal texels
  de grootte van de textuur moet verkleind worden.



In beide gevallen moet aangegeven worden welke texelwaarden moeten gebruikt worden en hoe deze uitgemiddeld of geïnterpoleerd moeten worden:

- GL_NEAREST: texel met coördinaten het dichtste bij het midden van de pixel wordt gebruikt, zowel voor magnificatie als minificatie;
- GL_LINEAR: gewogen lineair gemiddelde van een 2×2 array van texels dat het dichtste bij het midden van de pixel ligt, zowel voor magnificatie als minificatie;

De parameterwaarde zetten gebeurt met de GLTexParameteri functie:

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
```

Wanneer de textuur moet uitgerokken worden in de ene richting en ingekrimpt in de andere richting, is het niet duidelijk of magnificatie of minificatie moet toegepast worden. OpenGL maakt dan zelf een keuze die in de meeste gevallen het best mogelijke resultaat geeft. Het is wel aangewezen zo'n situatie te vermijden door gepaste textuurcoördinaten te gebruiken.

7.3.4 Textuurfuncties

Omgeving. Bij eenvoudige toepassing wordt de waarde in de textuurmap gebruikt als kleur voor het object. Het is mogelijk om waarde in de textuurmap te combineren met de oorspronkelijke kleur van het object. Om de combinatieactie aan te geven kan de volgende functie gebruikt worden:

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, argumunt);
```

Het argument kan zijn:

- GL_REPLACE: de oorspronkelijke kleur wordt vervangen door de kleur en alpha data van de texture;
- GL_MODULATE: de kleur en alpha data van de texture wordt vermenigvuldigd met de kleur data van `glColor` en/of het belichtingssysteem;
- GL_BLEND: de kleur en alpha data van de texture wordt gemend met de mengkleur aangegeven door de functie

  ```
  glTexEnv*(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, mengkleur);
  ```

  Afhankelijk van hoe de mengkleur gespecificeerd wordt, wordt het sterretje vervangen door i of f en eventueel een v.

Wrapping. Wanneer de coördinaten in de textuurruimte buiten het bereik tussen 0.0 en 1.0 liggen, wordt normaal het patroon van de textuurarray herhaald (GL_REPEAT). Men kan er ook voor kiezen om de kleur van de rand door te trekken (GL_CLAMP). De keuze kan verschillend zijn in de s-richting en in de t-richting:

```
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

Aan- en uitschakelen. In de teken callbackfunctie kan op de gewenste plaats textuurmapping aangeschakeld worden; en wordt uit de verschillende texturen die geladen zijn, de juiste gekozen:

```
glEnable( GL_TEXTURE_2D );
glBindTexture( GL_TEXTURE_2D, texobj[2] );
```

Nadat de openGL-functies om het voorwerp te tekenen, opgeroepen zijn, kan de textuurmapping terug uitgeschakeld worden:

```
glDisable( GL_TEXTURE_2D );
```

7.3.5 Relatie tussen coördinaten

Een texel $T(s,t)$ uit de textuur moet gekoppeld worden aan elk punt van het object, dat zelf gemapt wordt op schermcoördinaten. Wanneer de homogene coördinaten van zo'n punt gelijk zijn aan (x,y,z,w), dan bestaan er functies

$$\begin{cases} x &= x(s,t) \\ y &= y(s,t) \\ z &= z(s,t) \\ w &= w(s,t) \end{cases}$$

Maar eigenlijk worden we geconfronteerd met het inverse probleem. Uit de gegeven homogene coördinaten (x,y,z,w) moeten de corresponderende textuurcoördinaten gevonden worden:

$$\begin{cases} s &= s(x,y,z,w) \\ t &= t(x,y,z,w) \end{cases}$$

Met deze coördinaten kan dan de texelwaarde $T(s,t)$ bepaald worden.
In OpenGL wordt dit opgelost door de waarden van s en t deel te laten uitmaken van de OpenGL toestand. Het toekennen van dit paar van waarden gebeurt met de functie

```
glTexCoord2f(s,t);
```

Net zoals bijvoorbeeld bij glColor3f worden deze textuurcoördinaten gebruikt bij het verwerken van de volgende vertices. De mapping wordt dus gedefineerd door eerst glTexCoord2f op te roepen en dan bijvoorbeeld glVertex3f.

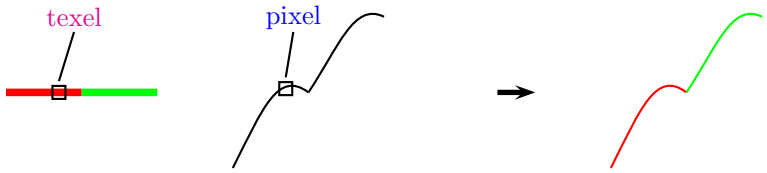
```
glBegin( GL_QUADS );
    glTexCoord2f(0.0, 0.0);    glVertex3f( 0.2,  0.0, 0.5);
    glTexCoord2f(0.0, 1.0);    glVertex3f( 1.0,  0.3, 0.5);
    glTexCoord2f(1.0, 1.0);    glVertex3f( 0.8,  1.0, 0.5);
    glTexCoord2f(1.0, 0.0);    glVertex3f( 0.0,  0.4, 0.5);
glEnd();
```

Soms is het moeilijk om oppervlaktecoördinaten van een object te definiëren. OpenGL heeft voor een aantal gevallen functies ter beschikking om automatisch textuurcoördinaten te genereren. Een voorbeeld is het plakken van een textuur op een kwadriek oppervlak, bijv. een bol:

```
GLUquadricObj *bol;
bol = gluNewQuadric();
glEnable( GL_TEXTURE_2D );
glBindTexture( GL_TEXTURE_2D, texobj[2] );
gluQuadricTexture(bol,GL_TRUE);
gluQuadricDrawStyle(bol,GLU_FILL);
gluSphere(bol,1,8,8);
gluDeleteQuadric(bol);
```

7.3.6 1D texture mapping

Soms is een eendimensionale textuur voldoende, bijvoorbeeld bij het tekenen van een lijn.



Het eerste argument bij veel van bovenstaande functies moet dan vervangen worden door GL_TEXTURE_1D. Bijvoorbeeld:

```
GLubyte texLijn[ZESTIEN];
for ( k=0; k<=ZESTIEN/8; k += 2 )
{
      texLijn[4*k] = texLijn[4*k+2] = 0;
      texLijn[4*k+1] texLijn[4*k+3] = 255;
      texLijn[4*(k+1)+1] = texLijn[4*(k+1)+2] = 0;
      texLijn[4*(k+1)] texLijn[4*(k+1)+3] = 255;
}
glBindTexture( GL_TEXTURE_1D, texobj[0] );
glTexImage1D( GL_TEXTURE_1D, 0, GL_RGBA, ZESTIEN/4, 0,
                    GL_RGBA, GL_UNSIGNED_BYTE, texLijn );
glEnable( GL_TEXTURE_1D );
glBegin( GL_LINES );
   glTexCoord1f(0.0);    glVertex3f(-0.5,  0.5, 0.2);
   glTexCoord1f(1.0);    glVertex3f( 0.5, -0.5, 0.4);
glEnd();
glDisable( GL_TEXTURE_1D );
```

8 Complexe geometrie

8.1 Inleiding

In het ontwerp en de productie van allerlei voorwerpen spelen complexe vormen een belangrijke rol. De vormgeving van het koetswerk van een wagen, de schoep van een turbine, een koffiezetmachine is niet uitsluitend gebaseerd op punten, lijnen en cirkels. Vanuit het standpunt van de ontwerper kunnen twee aparte benaderingen onderscheiden worden:

1. (*analytisch*) de ontwerper wil beschikken over een wiskundig model van een gekende en/of berekende vorm; zo'n eenduidige en beknopte wiskundige definitie laat toe vorm, inhoud, kromming, ... van een structuur te bestuderen;
2. (*synthetisch*) de ontwerper wil op basis van een vaak nog abstract concept, een vorm als het ware creëren; de beschikbare wiskundige modellen moeten hierbij zeer flexibele vormmanipulaties toelaten.

Voor de synthethische benadering kan met beroep doen op *splines*, een ruimtelijke curve bepaald door een lopende parameter u. Bij oppervlakken zijn er twee lopende parameters u en v.

Voorstelling van functies. Een functie kan op verschillende manieren wiskundig neergeschreven worden:

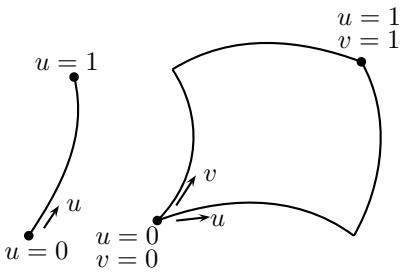
- expliciete vorm: $y = f(x)$
  voorbeeld: een cirkel met middelpunt in de oorsprong en straal R:
  $$y = \pm\sqrt{R^2 - x^2} \text{ met } 0 \leq |x| \leq R$$
- impliciete vorm : $f(x, y) = 0$
  $$x^2 + y^2 - R^2 = 0$$
- parameter vorm: $x = x(u)$ en $y = y(u)$
  $$P(u) = [R\cos(u), R\sin(u)] \text{ met } 0 \leq u \leq 2\pi$$

Omwille van een aantal algoritmische voordelen verkiest men binnen een CAD-systeem een parameterische voorstelling van ruimtelijke vormen boven een cartesiaanse.
Een punt p op een curve kan voorgesteld worden door een vector:

$$P(u) = [x(u), y(u), z(u)]$$

Voor de bepaling van een punt P op een oppervlak zijn twee parameters nodig:

$$P(u, v) = [x(u, v), y(u, v), z(u, v)]$$



Elk punt van de curve of van het oppervlak wordt ondubbelzinnig bepaald door één waarde van de parameters u en eventueel v, die zich tussen 0 en een maximum waarde situeren.

8.2 Spline

Spline is een verzameling van alle complexe ruimtelijke curven (een samentrekking van **S**pace **line**).

- controlepunt: ruimtelijke curven en oppervlakken worden vaak gedefinieerd uitgaande van een aantal punten in de ruimte; zo'n verzameling van punten bepaalt de vorm en ligging van de complexe vorm; indien de spline door alle controlepunten gaat, spreekt men van een *interpolerende* spline; het is mogelijk dat deze punten als een *ruimtelijke referentie* gebruikt worden;
- definiërende polygoon: de resulterende veelhoek door alle controlepunten met rechten te verbinden;

- globale versus lokale controle: de ligging en de vorm van een spline wordt mede bepaald door de ligging van de controlepunten: door één controlepunt te verplaatsen, wijzigt de vorm van de spline; indien de vorm van de *volledige spline* wijzigt, heeft men te maken met *global control*; indien de spline slechts plaatselijk in de buurt van het desbetreffende punt wijzigt, heeft men *local control*; lokale controle biedt de ontwerper meer geometrische flexibiliteit;
- variation diminishing eigenschap : een spline met deze eigenschap kan niet vaker rond een gegeven rechte slingeren dan zijn definiërende polygoon (een garantie dat de spline nooit kan oscilleren).

8.3 Bézier-spline curve

8.3.1 Definitie

Op basis van $n+1$ controlepunten P_i wordt de curve $P(u)$ gedefinieerd:

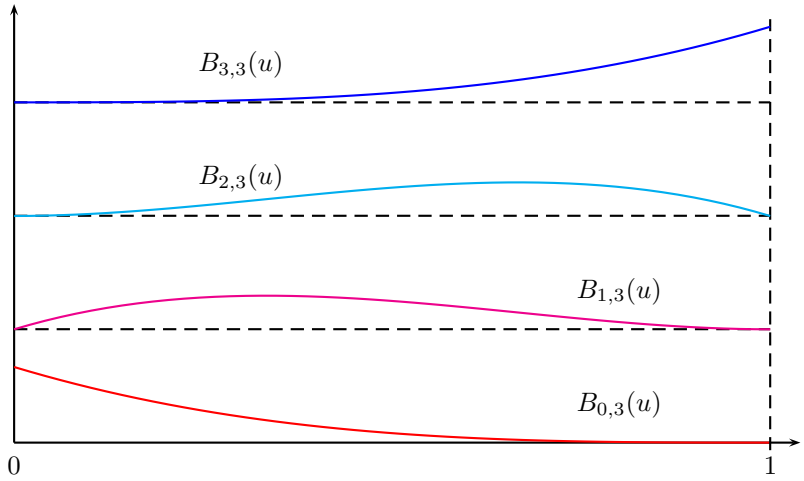
$$P(u) = \sum_{i=0}^{n} P_i B_{i,n}(u) \text{ met } 0 \leq u \leq 1$$

met *blending functie* $B(i,n)$ (de i-de Bernstein veelterm van n-de orde):

$$B_{i,n} = \frac{n!}{i!(n-i)!} u^i (1-u)^{n-i}$$

Voorbeeld met vier controlepunten $P_0(x_0, y_0, z_0)$, $P_1(x_1, y_1, z_1)$, $P_2(x_2, y_2, z_2)$ en $P_3(x_3, y_3, z_3)$. Vier controlepunten resulteert in een Bézier-spline van 3de orde of een cubische Bézier-spline:

$$\begin{aligned}
B_{0,3} &= \tfrac{3!}{0!3!} u^0 (1-u)^3 &&= (1-u)^3 \\
B_{1,3} &= \tfrac{3!}{1!2!} u^1 (1-u)^2 &&= 3u(1-u)^2 \\
B_{2,3} &= \tfrac{3!}{2!1!} u^2 (1-u)^1 &&= 3u^2(1-u) \\
B_{3,3} &= \tfrac{3!}{3!0!} u^3 (1-u)^0 &&= u^3
\end{aligned}$$



De Bézier-spline in vectornotatie:

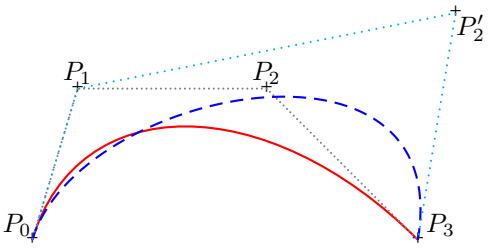
$$P(u) = (1-u)^3 P_0 + 3u(1-u)^2 P_1 + 3u^2(1-u) P_2 + u^3 P_3$$

Parametrische vergelijkingen voor x, y en z afzonderlijk zodat de evaluatie van de coördinaten van individuele punten van de spline eenduidig kan gebeuren:

$$
\begin{aligned}
x(u) &= (1-u)^3 x_0 + 3u(1-u)^2 x_1 + 3u^2(1-u)x_2 + u^3 x_3 \\
y(u) &= (1-u)^3 y_0 + 3u(1-u)^2 y_1 + 3u^2(1-u)y_2 + u^3 y_3 \\
z(u) &= (1-u)^3 z_0 + 3u(1-u)^2 z_1 + 3u^2(1-u)z_2 + u^3 z_3
\end{aligned}
$$

Een kubische Bézier-spline in het vlak $z = 0$ met controlepunten $P_0(0,0,0)$, $P_1(1,2,0)$, $P_2(4,2,0)$ en $P_3(6,0,0)$. Punten van deze curve worden berekend door de som te maken van de producten van de blending functie met het corresponderende controlepunt:

	P_0 $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ $(1-u)^3$	P_1 $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ $3u(1-u)^2$	P_2 $\begin{bmatrix} 4 \\ 2 \end{bmatrix}$ $3u^2(1-u)$	P_3 $\begin{bmatrix} 6 \\ 0 \end{bmatrix}$ u^3	
$u = 0.5$	0.0	+0.375	+1.500	+0.750	$= 2.625$
	0.0	+0.750	+0.750	+0.000	$= 1.500$
$u = 0.6$	0.0	+0.288	+1.728	+1.296	$= 3.312$
	0.0	+0.576	+0.864	+0.000	$= 1.440$



- De functies zijn overal reëel.
- Spline volgt vorm van de definiërende veelhoek.
- Begin- en eindpunt bepaald door $P(0)$ en $P(1)$.
- Spline ligt altijd volledig vervat in de grootste veelhoek die men met de controlepunten kan vormen.
- Spline heeft variation diminishing eigenschap.

De graad van de beschrijvende polynoom is gelijk aan het aantal controlepunten min één. Om geen veelterm van al te hoge graad te krijgen, beperkt men het aantal controlepunten tot 4 à 5.
De raakvectoren in begin- respectievelijk eindpunt van de spline vallen samen met de verbinding $P_0 - - - P_1$ respectievelijk $P_{n-1} - - - P_n$.
De spline is invariant onder elke eindige transformatie en is bijgevolg onafhankelijk van het assenkruis binnen dewelke de controlepunten gedefinieerd werden.
Wanneer $P_2(4,2,0)$ verplaatst wordt naar $P_2'(7,3,0)$ verandert de vorm van de spline over de ganse lengte. Er is dus een gebrek aan lokale controle bij Bézier-splines. Dit volgt ook uit het verloop van de blending functies $B_{(i,n)}$. De functiewaarde van een $B_{(i,n)}(u)$ voor een gegeven u waarde geeft het gewicht aan waarmee het overeenkomstige controlepunt de ligging van het punt op de spline voor deze parameterwaarde mede bepaalt. Deze functiewaarde is over het ganse interval van u-waarden groter dan 0 (behalve bij de grenzen) zodat elk controlepunt mede bepalend is.
De som van alle $B_{(i,n)}(u)$ voor een gegeven parameterwaarde van u is steeds gelijk aan 1. In het midden van de spline is de invloed van P_1 en P_2 uiteraard groter dan deze van P_0 en P_3. Bijvoorbeeld bij $P(0.5) = (2.625, 1.500, 0)$ worden de coördinaten bepaald door de controlepunten P_0, P_1, P_2 en P_3 met een relatief gewicht van respectievelijk 12.5%, 37.5%, 37.5% en 12.5%. De ligging van het begin- en eindpunt van de spline wordt uitsluitend bepaald door P_0 respektievelijk P_3, omdat de functiewaarden van de blending functies bij deze parameterwaarden ($u = 0$ respectievelijk $u = 1$) gelijk zijn aan 0.

8.3.2 OpenGL: Bézier-spline curve

Definitie van *controlepunten*:

```
GLfloat ctrlPnt[4][3] = { { 0.0,0.0,0.0},
        { 1.0,2.0,0.0}, { 4.0,2.0,0.0}, { 6.0,0.0,0.0}  };
```

Activeren en desactiveren van het genereren van een Bézier curve:

```
glEnable(GL_MAP1_VERTEX_3);
...
glDisable(GL_MAP1_VERTEX_3);
```

Het zetten van een aantal parameters door het definiëren van een eendimensionale evaluator:

```
glMap1f(type, uMin, uMax, stap, nPnt, ctrlPnt);
```

De evaluator werkt binnen het bereik tussen `uMin` en `uMax`. Er zijn `nPnt` controlepunten, gespecificeerd door het laatste argument. Dus de graad van de spline is gelijk aan `nPnt` - 1. Het argument `stap` is het aantal waarden tussen het begin van twee opeenvolgende controlepunten. Het argument `type` geeft aan wat de controlepunten aanduiden. Indien dit argument de waarde GL_MAP1_VERTEX_3 heeft, dan zijn de controlepunten een vector van x-, y- en z-waarden en is dus `stap` gelijk aan 3.
De functie glEvalCoord1f(uVal) doet de eigenlijke *evaluatie* volgens de gespecificeerde map voor een bepaalde u-waarde. Een *spline curve* genereren gebeurt dus door herhaaldelijk de glEvalCoord1f functie op te roepen. Bij elke oproep met `uVal` tussen `uMin` en `uMax` wordt een `glVertex3` functie gegenereerd en worden deze punten met lijnsegmenten verbonden:

```
glBegin(GL_LINE_STRIP);
    for (uVal = uMin; uVal <= uMax; uVal += 0.04)
        glEvalCoord1f(uVal);
glEnd();
```

Wanneer de u parameter uniform verdeeld is (n onderverdelingen) tussen `uMin` en `uMax` kan een grid gedefinieerd worden. Op basis van deze grid kan dan de evaluatie met één functieoproep gebeuren.

```
glMapGrid1f(n, uMin, uMax);        /* verdeling tussen uMin en uMax */
glEvalMesh1(mode, n1, n2);         /* GL_POINT of GL_LINE */
```

Voor `mode` gelijk aan GL_LINE is dit equivalent met:

```
glBegin(GL_LINE_STRIP);
    for (k=n1; k<=n2; k++)
        glEvalCoord1f(uMin + k * (uMax - uMin)/n );
glEnd();
```

8.4 B-spline curve

8.4.1 Definitie

Op basis van $n+1$ controlepunten en $n+k+1$ *knoop*waarden wordt de curve $P(u)$ gedefinieerd:

$$P(u) = \sum_{i=0}^{n} P_i N_{i,k}(u) \text{ met } 0 \leq u \leq n-k+2$$

Hierin zijn $N_{i,k}$ blending functies (zie verder) en de parameter u zit in het interval $[t_0, t_{n+k+1}]$.
De parameter k bepaalt de graad van de B-spline ($2 \leq k \leq n+1$): lineair ($k=2$), kwadratisch ($k=3$) of kubisch ($k=4$). De graad $(k-1)$ is dus niet afhankelijk van het aantal controlepunten. Naast het kiezen van controlepunten moeten er ook knoopwaarden t_i bepaald worden waarbij geldt dat $t_i \leq t_{i+1}$. Afhankelijk van de gekozen waarden voor deze knopen, kunnen verschillende soorten B-splines gedefinieerd worden:

- uniform periodic B-spline: interval tussen twee knoopwaarden is telkens even groot: voorbeeld $n = k = 3$: $n+k+1$ knoopwaarden: $\{0, 1, 2, 3, 4, 5, 6\}$

- open uniform B-spline: interval tussen twee knoopwaarden is telkens gelijk, behalve bij begin en einde: de kleinste en de grootste knoopwaarde worden k maal herhaald: knoopwaarden t_i bij een specifieke n en k waarde:

$$\begin{array}{lll} t_i = 0 & & 0 \leq i < k \\ t_i = i - k + 1 & \text{voor} & k \leq i \leq n \\ t_i = n - k + 2 & & n < i < n + k + 1 \end{array}$$

  voorbeeld $n = 5$ en $k = 3$: $\{0, 0, 0, 1, 2, 3, 4, 4, 4\}$
- nonuniform B-spline: knoopwaarden zijn oplopend, maar niet noodzakelijk gelijkmatig verspreid, sommige knoopwaarden kunnen ook een aantal maal herhaald worden: voorbeeld $n = 5$ en $k = 3$: $\{0, 0, 1, 1, 2, 3, 3, 3, 4\}$

De blending functies worden gedefineerd op basis van de Cox-deBoor recursie formules:

$$\text{Initialisatie:} \qquad N_{i,1}(u) = \begin{cases} 1 \text{ voor } t_i \leq u < t_{i+1} \\ 0 \text{ in alle andere gevallen} \end{cases}$$

$$\text{Recursievergelijking:} \qquad N_{i,k}(u) = \frac{(u - t_i)N_{i,k-1}(u)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - u)N_{i+1,k-1}(u)}{t_{i+k} - t_{i+1}}$$

De noemers van deze recursievergelijkingen kunnen gelijk worden aan nul; om hier een mouw aan te passen wordt $0/0$ gelijk genomen aan 0.

Voorbeeld met $n = 5$ en $k = 3$. De knoopwaarden voor een open uniforme B-spline zijn:

$$\begin{array}{lll} t_0 = 0 & t_3 = 1 & t_6 = 4 \\ t_1 = 0 & t_4 = 2 & t_7 = 4 \\ t_2 = 0 & t_5 = 3 & t_8 = 4 \end{array}$$

Recursie 1

$$\begin{array}{llll} N_{0,1}(u) & = 0 & & \\ N_{1,1}(u) & = 0 & & \\ N_{2,1}(u) & = 1 & \text{voor } 0 \leq u < 1 & \\ N_{3,1}(u) & = 1 & \text{voor } 1 \leq u < 2 & \end{array} \qquad \begin{array}{llll} N_{4,1}(u) & = 1 & \text{voor } 2 \leq u < 3 \\ N_{5,1}(u) & = 1 & \text{voor } 3 \leq u < 4 \\ N_{6,1}(u) & = 0 & \\ N_{7,1}(u) & = 0 & \end{array}$$

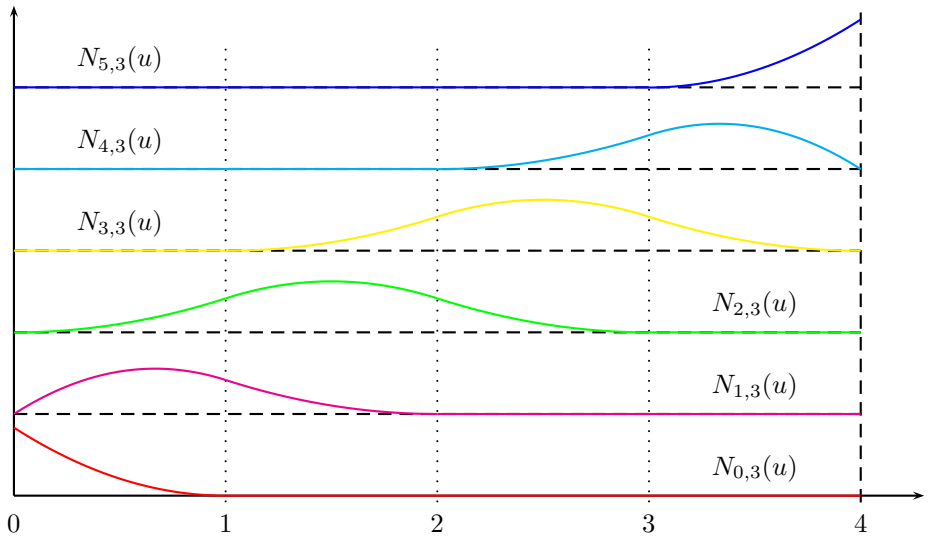
Recursie 2

$$\begin{array}{lll} N_{0,2}(u) & = \frac{(u-t_0)N_{0,1}(u)}{t_1-t_0} + \frac{(t_2-u)N_{1,1}(u)}{t_2-t_1} & = 0 \\ N_{1,2}(u) & = \frac{(u-t_1)N_{1,1}(u)}{t_2-t_1} + \frac{(t_3-u)N_{2,1}(u)}{t_3-t_2} & = (1-u)_{0\leq u<1} \\ N_{2,2}(u) & = \frac{(u-t_2)N_{2,1}(u)}{t_3-t_2} + \frac{(t_4-u)N_{3,1}(u)}{t_4-t_3} & = u_{0\leq u<1} + (2-u)_{1\leq u<2} \\ N_{3,2}(u) & = \frac{(u-t_3)N_{3,1}(u)}{t_4-t_3} + \frac{(t_5-u)N_{4,1}(u)}{t_5-t_4} & = (u-1)_{1\leq u<2} + (3-u)_{2\leq u<3} \\ N_{4,2}(u) & = \frac{(u-t_4)N_{4,1}(u)}{t_5-t_4} + \frac{(t_6-u)N_{5,1}(u)}{t_6-t_5} & = (u-2)_{2\leq u<3} + (4-u)_{3\leq u<4} \\ N_{5,2}(u) & = \frac{(u-t_5)N_{5,1}(u)}{t_6-t_5} + \frac{(t_7-u)N_{6,1}(u)}{t_7-t_6} & = (u-3)_{3\leq u<4} \\ N_{6,2}(u) & & = 0 \end{array}$$

Recursie 3

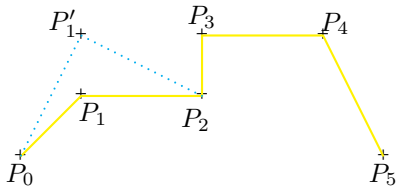
$$
\begin{aligned}
N_{0,3}(u) &= \tfrac{(u-t_0)N_{0,2}(u)}{t_2-t_0} + \tfrac{(t_3-u)N_{1,2}(u)}{t_3-t_1} \\
&= (1-u)^2_{0\le u<1} \\
N_{1,3}(u) &= \tfrac{(u-t_1)N_{1,2}(u)}{t_3-t_1} + \tfrac{(t_4-u)N_{2,2}(u)}{t_4-t_2} \\
&= u(1-u)_{0\le u<1} + \tfrac{(2-u)u}{2}{}_{0\le u<1} + \tfrac{(2-u)^2}{2}{}_{1\le u<2} \\
&= \tfrac{u}{2}(4-3u)_{0\le u<1} + \tfrac{(2-u)^2}{2}{}_{1\le u<2} \\
N_{2,3}(u) &= \tfrac{(u-t_2)N_{2,2}(u)}{t_4-t_2} + \tfrac{(t_5-u)N_{3,2}(u)}{t_5-t_3} \\
&= \tfrac{u}{2}\left(u_{0\le u<1} + (2-u)_{1\le u<2}\right) + \tfrac{(3-u)}{2}\left((u-1)_{1\le u<2} + (3-u)_{2\le u<3}\right) \\
&= \tfrac{u^2}{2}{}_{0\le u<1} + \tfrac{u}{2}(2-u)_{1\le u<2}) + \tfrac{(3-u)(u-1)}{2}{}_{1\le u<2} + \tfrac{(3-u)^2}{2}{}_{2\le u<3} \\
&= \tfrac{u^2}{2}{}_{1\le u<2} + \left(-u^2+3u-\tfrac{3}{2}\right)_{1\le u<2} + \tfrac{(3-u)^2}{2}{}_{2\le u<3} \\
N_{3,3}(u) &= \tfrac{(u-t_3)N_{3,2}(u)}{t_5-t_3} + \tfrac{(t_6-u)N_{4,2}(u)}{t_6-t_4} \\
&= \tfrac{(u-1)}{2}\left((u-1)_{1\le u<2} + (3-u)_{2\le u<3}\right) + \tfrac{(4-u)}{2}\left((u-2)_{2\le u<3} + (4-u)_{3\le u<4}\right) \\
&= \tfrac{(u-1)^2}{2}{}_{1\le u<2} + \tfrac{(u-1)(3-u)}{2}{}_{2\le u<3} + \tfrac{(4-u)(u-2)}{2}{}_{2\le u<3} + \tfrac{(4-u)^2}{2}{}_{3\le u<4} \\
&= \tfrac{(u-1)^2}{2}{}_{1\le u<2} + \left(-u^2+5u-\tfrac{11}{2}\right)_{2\le u<3} + \tfrac{(4-u)^2}{2}{}_{3\le u<4} \\
N_{4,3}(u) &= \tfrac{(u-t_4)N_{4,2}(u)}{t_6-t_4} + \tfrac{(t_7-u)N_{5,2}(u)}{t_7-t_5} \\
&= \tfrac{(u-2)}{2}\left((u-2)_{2\le u<3} + (4-u)_{3\le u<4}\right) + \tfrac{(4-u)}{1}(u-3)_{3\le u<4} \\
&= \tfrac{(u-2)^2}{2}{}_{2\le u<3} + \tfrac{(u-2)(4-u)}{2}{}_{3\le u<4}) + (4-u)(u-3)_{3\le u<4} \\
&= \tfrac{(u-2)^2}{2}{}_{2\le u<3} + \left(-\tfrac{3}{2}u^2+10u-16\right)_{3\le u<4} \\
N_{5,3}(u) &= \tfrac{(u-t_5)N_{5,2}(u)}{t_7-t_5} + \tfrac{(t_8-u)N_{6,2}(u)}{t_8-t_6} \\
&= (u-3)(u-3)_{3\le u<4} \\
&= (u-3)^2_{3\le u<4}
\end{aligned}
$$

Blending functies bij $n=5$ en $k=3$ (2e orde) knoopwaarden: 0.0, 0.0, 0.0, 1.0, 2.0, 3.0, 4.0, 4.0, 4.0 (open uniform)

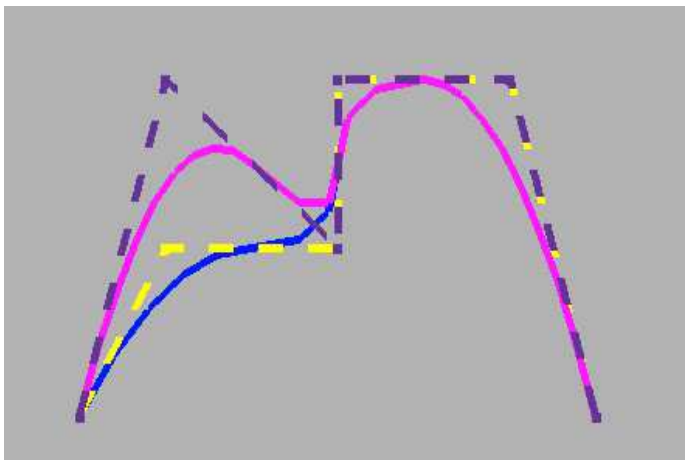


Met controlepunten

$P_0 = (0,0,0)$ $P_1 = (1,1,0)$ $P_2 = (3,1,0)$
$P_3 = (3,2,0)$ $P_4 = (5,2,0)$ $P_5 = (6,0,0)$



wordt in figuur 8.1 de kwadratische B-spline getoond.



Figuur 8.1: open uniform B-spline

De coördinaten van het punt op de B-spline bij parameterwaarde $u = 0.6$ worden bepaald door drie van de zes blending functies: $N_{(0,3)}(u)$, $N_{(1,3)}(u)$ en $N_{(2,3)}(u)$. Bij $u = 0.6$ zijn dus slechts drie controlepunten (P_0, P_1 en P_2) bepalend voor de vorm van de spline. Door de ligging van één van deze controlepunten te wijzigen (P_1 verschuift naar P'_1i), wordt de vorm van de B-spline alleen gewijzigd in de buurt van dit gewijzigde controlepunt. Dus, lokale controle is mogelijk.

Verder heeft een B-spline volgende eigenschappen:

- de functies zijn overal reëel;
- begin- P_0 en eindpunt P_n;
- de spline volgt de vorm van de definiërende veelhoek;
- variation diminishing eigenschap;
- de graad van de beschrijvende veeltermen is louter afhankelijk van de keuze van de parameter k en dus niet van het aantal controlepunten;
- de spline is invariant onder een affiene transformatie.

8.4.2 OpenGL: B-spline curve

```
GLUnurbsObj *curNaam;

curNaam = gluNewNurbsRenderer();
gluBeginCurve(curNaam);
    gluNurbsCurve(curNaam, nKnp, KnpVector, uStap, ctrlPnt,
                  uDegParam, GL_MAP1_VERTEX3);
gluEndCurve(curNaam);
gluDeleteNurbsRenderer(curNaam);
```

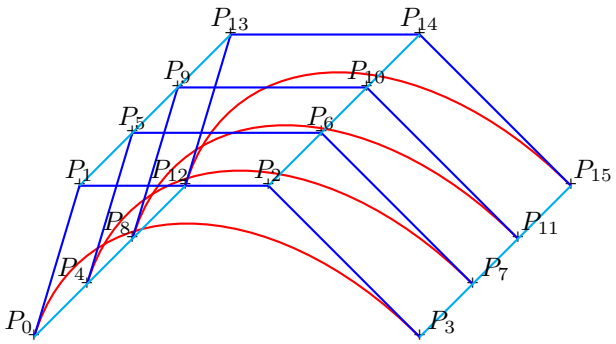
Voorbeeld:

```
GLfloat knot[9] = { 0.0, 0.0, 0.0, 1.0, 2.0, 3.0, 4.0, 4.0, 4.0 };
GLfloat ctrlPnt[6][3] = {
    { 0.0, 0.0, 0.0}, { 1.0, 1.0, 0.0}, { 3.0, 1.0, 0.0},
    { 3.0, 2.0, 0.0}, { 5.0, 2.0, 0.0}, { 6.0, 0.0, 0.0}   };
 gluNurbsCurve(curNaam, 9, knot, 3, ctrlPnt, 3, GL_MAP1_VERTEX3);
```

8.5 Oppervlakken

8.5.1 Definities

De definitie van een complex oppervlak gevormd door splines is een uitbreiding van een complexe curve door naast de u parameter een tweede parameter v in rekening te brengen. Het oppervlak wordt gevormd door het carthesiaans product van twee orthogonale curven. In plaats van een rij van controlepunten, gerangschikt in een definiërende veelhoek, wordt voor de vormgeving van het oppervlak een *point mesh* gebruikt van n bij m controlepunten.



Bézier-spline oppervlak met als parameters u en v:

$$P(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} P_{i,j} B_{i,n}(u) B_{j,m}(v) \quad \text{met} \quad \begin{cases} 0 \le u \le 1 \\ 0 \le v \le 1 \end{cases}$$

B-spline oppervlak met als parameters u en v: (analoog)

$$P(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} P_{i,j} N_{i,k}(u) N_{j,l}(v) \quad \text{met} \quad \begin{cases} 0 \le u \le n-k+2 \\ 0 \le v \le m-l+2 \end{cases}$$

8.5.2 OpenGL: Bézier-spline oppervlak

Zetten van parameters, activeren van routines voor display en desactiveren:

```
glMap2f(GL_MAP2_VERTEX_3, uMin, uMax, uStap, nuPnt,
                          vMin, vMax, vStap, nvPnt, ctrlPnt);
glEnable(GL_MAP2_VERTEX_3);
...
glDisable(GL_MAP2_VERTEX_3);
```

Evaluatie van posities langs het spline pad: glEvalCoord2f(uVal, vVal);
Generatie van een spline mesh: glEvalCoord2f functie herhaaldelijk oproepen met `uVal` tussen `uMin` en `uMax` en `vVal` tussen `vMin` en `vMax` en deze punten verbinden met lijnsegmenten
Met uniform verdeelde parameter waarden (nu en nv onderverdelingen) :

```
glMapGrid2f(nu, u1, u2, nv, v1, v2);
glEvalMesh2(mode, nu1, nu2, nv1, nv2);
```

Voor `mode` gelijk aan GL_FILL is dit equivalent met:

```
for (k=nu1; k<=nu2; k++ )
{
  glBegin(GL_QUAD_STRIP);
  for (l=nv1; l<=nv2; l++ )
  {
    glEvalCoord2f(u1 + k * (u2 - u1)/nu, v1 + l * (v2 - v1)/nv);
    glEvalCoord2f(u1 + (k+1) * (u2-u1)/nu, v1 + l * (v2-v1)/nv);
  }
  glEnd();
}
```

Alternatieven voor mode glBegin zijn `GL_POINT`, `GL_LINE`, `GL_FILL`.

Voorbeeld: een Bézier oppervlak (uniform verdeelde parameterwaarden, u, v) met een textuur:

```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texobj[0]);
glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
                  0, 1, 12, 4, &ctrlpoints[0][0][0]);
glEnable(GL_MAP2_VERTEX_3);
glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 3, 4,
                  0, 1, 12, 4, &ctrlpoints[0][0][0]);
glEnable(GL_MAP2_TEXTURE_COORD_2);
glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
glEvalMesh2(GL_FILL, 0, 20, 0, 20);
glDisable(GL_MAP2_TEXTURE_COORD_2);
glDisable(GL_MAP2_VERTEX_3);
glDisable(GL_TEXTURE_2D);
```

8.5.3 OpenGL: B-spline oppervlak

```
GLUnurbsObj *oppNaam;
oppNaam = gluNewNurbsRenderer();
gluNurbsProperty(oppNaam, eigenschap, waarde);
gluBeginSurface(oppNaam);
  gluNurbsSurface(oppNaam, nuKnp, uKnpVector, nvKnp, vKnpVector,
                           uStap, vStap, &ctrlPnt[0][0][0],
                           uDegParam, vDegParam, GL_MAP2_VERTEX3);
gluEndSurface(oppNaam);
```

De functie gluNurbsProperty zet een aantal eigenschappen van het B-spline oppervlak. Bijvoorbeeld, voor de eigenschap GLU_DISPLAY_MODE zijn mogelijke waarden `GLU_OUTLINE_POLYGON` of `GLU_FILL`.

Het is mogelijk om een textuur te mappen op het B-spline oppervlak. In de functie gluNurbsSurface moet het laatste argument vervangen worden door GL_MAP2_TEXTURE_COORD_2:

```
GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
GLfloat ctlpoints[4][4][3];

gluNurbsSurface(theNurb, 8, knots, 8, knots, 4 * 3, 3,
          &ctlpoints[0][0][0], 4, 4, GL_MAP2_TEXTURE_COORD_2);
gluNurbsSurface(theNurb, 8, knots, 8, knots, 4 * 3, 3,
          &ctlpoints[0][0][0], 4, 4, GL_MAP2_VERTEX_3);
```

Project Computergrafieken: 2018-19

Deel 1 : coördinaten systemen

Gegeven: twee punten P_1 en P_2 in wereldcoördinaten (WC), $(a, b, c, 1)$.
Bereken

1. de transformatiematrices naar viewcoördinaten; de camera staat ofwel in het xz-vlak ofwel in het xy-vlak ofwel in het yz-vlak en kijkt in de richting (0,0,0), de specifieke x-, y- en z-coördinaten kunnen in onderstaande tabel gevonden worden;
2. de transformatiematrix van de algemene perspectief projectie; hierbij is $xw_{\min}$, $xw_{\max}$, $yw_{\min}$, $yw_{\max}$, *near* en *far* in onderstaande tabel gegeven;
3. de transformatiematrices van de normalisatie naar een kubus met linkeronderhoek (-1,-1,-1) en rechterbovenhoek (1,1,1);
4. de transformatiematrix van de viewport om de x en y genormaliseerde coördinaten om te zetten in display coördinaten; de breedte (b) en hoogte (h) van de viewport kan in onderstaande tabel gevonden worden.

		b	h	x	y	z	$xw_{\min}$	$xw_{\max}$	$yw_{\min}$	$yw_{\max}$	near	far
1	Beuselinck	500	500	5.0	3.0	0.0	-2.0	3.0	-1.5	2.5	2.0	15.0
2	Bleyaert	500	400	4.0	3.0	0.0	-2.0	3.0	-1.5	3.5	2.5	14.0
3	Caerlen	500	300	3.0	4.0	0.0	-2.0	3.0	-1.5	1.5	1.5	13.0
4	Colignon	400	500	4.0	0.0	5.0	-1.5	3.5	-1.0	2.0	2.0	12.0
5	DeBels	400	400	3.0	0.0	2.0	-1.5	3.5	-1.0	3.0	1.0	11.0
6	Dehaene	400	300	2.0	0.0	4.0	-2.5	2.5	-2.0	3.0	1.0	12.0
7	DeSmedt	300	500	0.0	3.0	5.0	-1.0	3.0	-1.5	2.5	2.0	13.0
8	DeSmet	300	400	0.0	2.0	4.0	-1.0	3.0	-2.5	2.5	3.0	14.0
9	DeWit	300	300	0.0	3.0	5.0	-1.0	3.0	-1.5	2.5	3.0	15.0
10	Gadisseur	500	300	0.0	3.0	4.0	-1.5	2.5	-2.0	3.0	2.0	14.0
11	Hendrickx	500	400	0.0	4.0	3.0	-1.5	2.5	-1.0	2.0	1.0	13.0
12	Kindt	500	500	0.0	2.0	5.0	-1.5	2.5	-2.0	2.0	1.0	12.0
13	Lerner	400	300	4.0	0.0	2.0	-2.0	2.0	-1.5	3.5	2.0	11.0
14	Rousseeuw	400	400	3.0	0.0	5.0	-2.0	2.0	-1.5	2.5	3.0	12.0
15	VanAeken	400	500	3.0	0.0	4.0	-2.0	2.0	-0.5	2.5	3.0	13.0
16	Vanlathem	300	300	4.0	3.0	0.0	-1.5	3.5	-1.0	3.0	1.5	14.0
17	Eind	400	300	5.0	0.0	4.0	-1.5	4.5	-1.0	2.0	4.0	15.0
18	Boeckx	300	400	3.0	5.0	0.0	-2.5	3.5	-2.0	1.0	3.0	15.0
19	CHuybrechts	300	500	4.0	2.0	0.0	-2.5	2.5	-2.0	3.0	2.0	13.0
20	JHuybrechts	500	300	0.0	3.0	4.0	-2.0	3.0	-1.5	3.5	3.0	14.0
21	VanBeylen	300	400	0.0	4.0	3.0	-1.0	2.0	-1.5	3.5	2.5	12.0
22	Vastmans	300	300	0.0	5.0	3.0	-1.0	3.0	-1.5	2.5	2.5	14.0
23	Vermeiren	400	500	3.0	0.0	4.0	-1.5	3.5	-1.0	2.0	3.5	15.0
24	Wuyts	400	400	3.0	0.0	5.0	-2.5	2.5	-2.0	3.0	2.5	13.0

Bereken de VC, PC, NC en DC voor de punten $P_1 = (1, 1, 1, 1)$
en $P_2 = (2, 1, -1, 1)$ (wanneer de camera zich in het xy-vlak bevindt)
en $P_2 = (2, -1, 1, 1)$ (wanneer de camera zich in het yz-vlak bevindt)
en $P_2 = (2, 2, -1, 1)$ (wanneer de camera zich in het xz-vlak bevindt).
Bepaal op basis van de resulterende NC z-coördinaten welk van deze punten het dichtste bij de camera ligt.

Inlevermoment : verslag (op papier):
donderdag 4 april 2019 voor 13.15 uur bij Herman Crauwels.

Deel 2 : complexe geometrie

C-programma met OpenGL bibliotheek-functies voor de voorstelling van een reuzenrad. Het reuzenrad bestaat uit vier steunbalken, het rad zelf en een aantal cabines. Dit aantal is een argument van het programma (bijv. -z5) De steunbalken staan langs de twee kanten van het rad en ondersteunen de centrale as. Door middel van een clipping vlak hebben de schuinstaande steunbalken toch een rechthoekig grondvlak Het rad bestaat uit een centrale (as)cylinder, twee schijven met elkaar verbonden door cylinders en spaken naar de as. Een cabine bestaat uit een kuipje en een dak (in de eenvoudige versie een kegel) gekoppeld via een ophangstaaf, die zelf hangt aan een verbindingscylinder.
Een kuipje op dit reuzerad wordt voor de helft gevormd door een complex oppervlak (Bézier) op de juiste manier verschoven, geschaald en gedraaid. De andere helft van een kuipje wordt gevormd uit hetzelfde Bézier oppervlak (via een spiegeling).
Als eerste uitbreiding kan het dak vervangen worden door een complex oppervlak, bestaande uit een B-spline.
Volgende elementen worden voorzien:

1. **raam**: callback functie bij herschaling van het window; voorzie een optie in het programma om te kunnen kiezen tussen een ortogonale, symmetrische perspectief of algemene perspectief transformatie; kies zelf gepaste waarden voor de verschillende argumenten van deze functies;
2. **kermis**: basisfunctie waarin o.a. xyz-assen en plaats van lichtbronnen kunnen weergegeven worden; deze functie roept de functies op om de verschillende elementen weer te geven; met de toets n worden meerdere exemplaren van het reuzenrad naast elkaar geplaatst;
3. **kuipje**: een complex oppervlak gevormd door Bézier functies gebaseerd op 6×4 controlepunten (4 controlepunten in de breedte en 6 controlepunten in de lengte); dit oppervlak wordt gebruikt voor een half kuipje; (voorzie eventueel toets l om de lijnenmesh te tonen);
4. **dak**: (belangrijkste uitbreiding, ipv. kegel) een complex oppervlak gevormd door B-spline functies van orde 4 (graad 3) gebaseerd op $C \times C$ controlepunten met $C >= 4$; dit oppervlak wordt gebruikt voor een kwart van het dak; door dit oppervlak driemaal telkens over 90 graden te draaien komt het volledige dak tot stand;
5. **licht**: vier lichtbronnen die apart aan- en uit-geschakeld kunnen worden: één voor ambient wit licht, één voor diffuus licht (vooral groen en blauw) en één voor specular licht (vooral rood); kies zelf goede posities voor deze lichtbronnen zodat de effecten duidelijk kunnen gedemonstreerd worden; een vierde lichtbron met geel ambient, geel diffuus, geel specular licht en spoteffect wordt geplaatst op instelbare hoogte en gericht naar het rad; de hoogte kan aangepast worden met de toets h/H;
6. in de basisopgave voorzie je voor de kuipjes, de stangen (ondersteuning,spaken en as) en het dak verschillende materiaaleigenschappen, bijv. grijs, brons en geel;
   menu (uitbreiding): kiezen van de materiaaleigenschappen van kuipjes, stangen en dak:

	materiaal	ambient			diffuus			specular		
kuipje	grijs	0.22	0.22	0.22	0.33	0.33	0.33	0.11	0.11	0.11
	witachtig	0.66	0.66	0.66	0.77	0.77	0.77	0.55	0.55	0.55
stangen	chroom	0.46	0.58	0.35	0.23	0.29	0.17	0.69	0.87	0.52
	brons	0.21	0.13	0.10	0.39	0.27	0.17	0.71	0.43	0.18
dak	geel	0.65	0.55	0.15	0.75	0.45	0.15	0.85	0.35	0.15
	lila	0.45	0.15	0.75	0.55	0.15	0.65	0.35	0.15	0.85

7. **toets**: callback routine bij het indrukken van een toets. Voorzie x/X, y/Y, z/Z voor het verplaatsen van het *oog*; a/A, b/B, c/C, d/D voor het aan/uitschakelen van de lichtbronnen; s/S voor de keuze tussen FLAT en SMOOTH shading;
   met toets v/V kan eventueel de spothoek van de spot verkleind of vergroot worden; spot exponent kan in stappen van 5 aangepast worden met toets w/W, initiële waarde 20; ma-

teriaaleigenschap *shininess* kan in stappen van 5 aangepast worden met toets e/E, initiële waarde 10;
met toets l kunnen de draadmodellen van de cylinders, schijven en de complexe oppervlakken al of niet getoond worden; de xyz-assen kunnen met toets j al of niet getoond worden, en de controlepunten van de oppervlakken met de toets k;
meerdere exemplaren van het reuzenrad tonen kan met de toets n;
hoogte van de spot aanpassen kan met de toetsen h/H;

8. **anim**: het reuzenrad kan draaien (toets g); het wiebelen van de cabines kan met de toets G geactiveerd worden;
9. **textuur** (uitbreiding): het kuipje en het dak van en cabine kunnen voorzien worden van een textuur op basis van een kleurenpallet of een jpeg (in/uitschakelen met toets t); ook kan een grondoppervlak (bijvoorbeeld kasseien) weergegeven worden;
10. **glas**: de twee draaiende schijven van het rad zijn doorzichtig (toets f)
bij de materiaaleigenschappen van deze *doorzichtige* oppervlakken een gepaste α nemen, zodat o.a. de achterliggende stangen en cabines zichtbaar zijn;
11. **mist** (uitbreiding): met *fog* kan ervoor gezorgd worden dat bij meerdere reuzenraden het achterste rad een waziger uitzicht krijgt (in/uitschakelen met toets m); met M kan eventueel het exponentiële effect geactiveerd worden.

Inlevermoment

Maak ook een verslag met daarin een opsomming van de verschillende ontwikkelde routines en een lijst van letters met betekenis voor het (des)activeren van de verschillende fenomenen (in PDF).

Het verslag (in PDF) en de bestanden met de C-broncode en eventuele jpeg's in aparte attachments mailen voor maandag 27 mei 2019 naar herman.crauwels@kuleuven.be .