**"Plane Mania"** 6 Januar, 2005

von Christian Ofenberg (www.ablazespace.de)

## 1. Einleitung

Ebenen müsste jeder bereits aus der Schulmathematik kennen. Aber genauso wie bei Vektoren, Matrizen etc. bekam man nicht wirklich gesagt für was man dieses Dinge überhaupt benötigt, und so verdrängte man das Wissen schnell wieder. Spätestens aber wenn man sich mit z.B. 3D Programmierung befasst kommt man nicht um diese früher als eigentlich nutzlos erschienenen Dinge herum und man erkennt schnell die vielen praktischen Anwendungen von Ebenen, Vektoren und Matrizen.

In diesem Artikel soll gezeigt werden das viele Programmiertechnische Probleme wie die Sichtbarkeitsbestimmung, Ereignis-Zonen oder gar "Selektieren über ein Rechteck ziehen" wie man es aus vielen Echtzeitstrategie-Spielen kennt im Prinzip auf arbeiten mit Ebenen reduziert werden können - und so gesehen alles immer gleich abläuft. Man braucht also keine speziellen und umfangreichen Klassen für Beispielsweise den Kamera Frustum sondern man hat eine Ebenen-Basisklasse und eine Klasse welche eine Menge von Ebenen verwaltet und Methoden darauf anbietet. Von dieser grundliegenden "Ebenen Menge"-Klasse lassen sich dann beispielsweise Anwendungs spezifischere Klassen ableiten welche aus einer Projektion Matrize die Kamera Frustum Ebenen erzeugt, eine Ereignis-Zone darstellen etc.

Ich setzte Basiswissen über Ebenen, Vektoren und Matrizen voraus - daher beschreibe ich nur sehr Oberflächlich diese Mathematischen Dinge um alles noch mal kurz ins Gedächtnis zu rufen. In er Praxis braucht man meist nur Vektoren im $R^3$, 3x3 oder 4x4 Matrizen etc. und daher sind diese Dinge auch sehr Anschaulich… im Gegensatz zu den n-Dimensionalen Dingen die man aus dem Lehrbüchern kennt. ;-)
Teils verwende ich Englische Begriff wie ‚Frustum' da es teilweise schlicht unmöglich ist vernümpftige Deutsche Begriffe für bestimmte Dinge zu finden – des weiteren sind dies allgemein bekannte Begriffe mit denen man z.B. über google auch sofort etwas finden kann.
Hier und da sind immer mal wieder Code Fragmente eingestreut um das Besprochene an einem einfachen C++ Code noch mal zusammenfassend zu präsentieren. Diese Code-Schnipsel sind wirklich nur knappe Beispiele welche die Grundidee vermitteln sollen. Komplette Vektor, Ebene und Matrizen Klassen sind schnell selbst geschrieben – und fast jeder hat bereits solche Klassen in seinem eigenen Code.
Ich verweise auch öfters auf Quellen welche eine bestimmte Thematik genauer behandeln - man muss das Rad ja nicht jedes Mal neu erfinden, also allgemein bekannte Dinge immer und immer wieder von neuem Niederschreiben. Bis jetzt hab ich jedoch noch keinen Artikel im Netz gefunden der die gängigen Anwendungsmöglichkeiten von Ebenen aufzählt, so, das man sich alle 'Häppchen' mühevoll selbst erarbeiten und teils z.B. in Foren zusammenfragen muss - darum schrieb ich diesen Artikel damit man alles endlich mal zusammengefasst hat. :)

## 2. Grundlagen

### 2.1. Die Ebene

Eine Ebene ist ein Mathematisches-Gebilde welches den Raum (wir arbeiten hier nur im $R^3$) so gesehen in zwei hälften 'schneidet' - entweder ist etwas 'darunter' oder "darüber". Es gibt verschiedene Mathematische Darstellungen für Ebenen.
Um zwei zu nennen: Die Hessesche Normalenform (HNF) "E: n*(x-p)" in der man einen Stürzvektor (p) + Normalenvektor (n) angibt und die Parametrisierte-Koordinatenform "E:

ax1+bx2+cx3-d" in welcher die ersten 3 Parameter den (normierten) Normalen-Vektor und d den Abstand der Ebene vom Nullpunkt darstellt.
In der 3D Programmierung arbeitet man für gewöhnlich mit der letzteren - also der Parametrisierten-Koordinatenform Darstellung der Ebene.
Da Ebenen eine unendliche Ausdehnung haben kann man immer nur einen Teilabschnitt dieser Visuell Darstellen was aber im Prinzip immer ausreichend ist – aber beim Arbeiten damit muss man diese Tatsache immer im Hinterkopf haben. Im 'Normalfall' erzeugt man die Ebenengleichung mithilfe von 3 Punkten im Raum - z.B. mit den 3 Vektoren eines Dreiecks aus dem sich dann ganze Modelle zusammensetzen können.
Zuerst berechnet man die zwei Richtungsvektoren welche die Ebene 'aufspannen' indem man z.B. Vektor 1 von Vektor 3 abzieht und Vektor 3 von Vektor 2. Anschließend bildet man das Kreuzprodukt dieser beiden Vektoren und erhält somit die Normale der Ebene - also einem Vektor der Senkrecht auf der Ebene steht. Nun gilt es nur noch den Abstand der Ebene zum Nullpunkt zu berechnen was sehr einfach ist da man nur einen gegebenen Punkt (wir haben in dem Fall gleich 3 davon zur Auswahl :) welcher auf der Ebene liegt (z.B. Vektor 1) in die Formel ax1+bx2+cx3=d einsetzen muss um 'd' zu erhalten. Da viele Anwendungen von Ebenen erwarten das die Ebene Normalisiert ist, der Normalenvektor also eine Länge von 1 hat ist es sinnvoll die Ebene immer als normalisierte Variante zu speichern.
Zusammenfassung wie man die Ebenengleichung aufstellen kann: (die Funktion heißt in meiner Ebenen Klasse ComputeND())

```
// Gegeben: Drei Vektoren welche die Ebene bilden
float V1X, V1Y, V1Z, V2X, V2Y, V2Z, V3X, V3Y, V3Z;

// Die zwei Richtungvektoren berechnen welche die Ebene aufspannen
// Vektor 1
float X1 = V3X-V1X;
float Y1 = V3Y-V1Y;
float Z1 = V3Z-V1Z;
// Vektor 2
float X2 = V2X-V3X;
float Y2 = V2Y-V3Y;
float Z2 = V2Z-V3Z;
// Normale der Ebene über Kreuzprodukt V1 kreuz V2 berechnen
float NX = Y1*Z2 - Z1*Y2;
float NY = Z1*X2 - X1*Z2;
float NZ = X1*Y2 - Y1*X2;
// Abstand der Ebene zum Nullpunkt über Skalarprodukt zwischen V1 und N ermitteln
float D = -(V1X*NX + V1Y*NY + V1Z*NZ);
// Ebene Normalisieren
float NL = sqrtf(NX*NX + NY*NY + NZ*NZ);
NX /= NL;
NY /= NL;
NZ /= NL;
D  /= NL;
```

**2.2. Ebene*Matrize**
Desöfteren ist es praktisch Ebenen mit einer Matrize zu transformieren. Ein gutes Beispiel dafür ist z.B. der Quadtree eines Terrains den man dafür verwendet um zu prüfen welche Teile des Terrains gerade sichtbar sind.

Dafür testet man dann diesen Quadtree mit dem aktuellen Kamera Frustum - aber sobald das Terrain in der Welt anders positioniert und/oder rotiert ist, also nicht alles im Ursprung liegt, müsste man von allen Quadtree Bounding Boxes irgendwie wieder die aktuelle Welt Positionen finden da sich der Kamera Frustum im World Space befindet. In diesem Fall ist es deutlich einfacher diesen Frustum einfach in den Object-Space des Terrains zu transformieren so das man quasi dann den Sichtbarkeitstest so machen kann als ob das Terrain nicht in der Welt verschoben wäre. Also muss man alle Ebenen welche diesen Kamera-Frustum bilden mit der inversen der Welt-Matrize multiplizieren welche ein Objekt in der Welt platziert. Direct3D bietet dafür bereits eine passende Funktion an: D3DXPlaneTransform().
In OpenGL muss man das per Hand machen - und wenn man API unabhängig sein will ist es sowieso besser solche Dinge selbst anzupacken. Um nun also eine Ebene mit einer Matrize transformieren zu können muss man zuerst einen Punkt p auf der Ebene wählen (z.B. -D*N) und diesen dann mithilfe der Objekt-zu-Welt-Matrize zu p´ transformieren. Die Normale n der Ebene wird mit der oberen linken 3x3 Untermatrize rotiert (n´). Am Ende muss man nur noch die neue Ebene im Object-Space mithilfe des Stützvektors p´ und des Normalenvektors n´ welche sich nun beide im Object-Space befinden wie oben im Teil ‚Die Ebene' beschrieben berechnen. Das macht man mit allen Ebenen des Frustums und schon hat man ein Problem weniger und die Sichtbarkeitsbestimmung des Terrains klappt auch wenn dieses in der Welt verschoben ist. ☺
Um das Transformieren einer Ebene mit einer Matrize möglichst komfortabel zu machen kann man den *-Operator der Ebenen-Klasse überladen:

```
CPlane CPlane::operator * (const CMatrix4x4 &mMatrix) const
{
   CPlane cPlane;

   // Ermittle einen beliebigen Punkt auf der Ebene (-D*N)
   CVector3D vP(-fD*fN[0], -fD*fN[1], -fD*fN[2]);
   // Transformiere p (-D*N) mithilfe der 4x4 Matrize zu p'
   vP = mMatrix.Transform(vP);
   // Transformiere n mithilfe der oberen 3x3 Untermatrize zu n'
   CVector3D vN = mMatrix.Rotate(CVector3D(fN));
   // Erzeuge neue Ebene durch p' mit der Normalen n'
   cPlane.ComputeND(vP, vN);

   // Gebe die Transformierte Ebene zurück
   return cPlane;
}
```

Zuletzt die Funktionen der Matrizen-Klasse welche einen Vektor transformieren & rotieren:

```
CVector3D CMatrix4x4::Transform(const CVector3D &vV) const
{
   CVector3D vRes;

   // Transformiere Vektor
   float d = 1.0f/(fXW*vV.fX + fYW*vV.fY + fZW*vV.fZ + fWW);
   vRes.SetXYZ((fXX*vV.fX + fYX*vV.fY + fZX*vV.fZ + fWX)*d,
               (fXY*vV.fX + fYY*vV.fY + fZY*vV.fZ + fWY)*d,
               (fXZ*vV.fX + fYZ*vV.fY + fZZ*vV.fZ + fWZ)*d);
```

```
   // Gebe transformierten Vektor zurück
   return vRes;
}

CVector3D CMatrix4x4::Rotate(const CVector3D &vV) const
{
   CVector3D vRes;

   // Rotiere Vektor
   vRes.SetXYZ(fXX*vV.fX + fYX*vV.fY + fZX*vV.fZ,
               fXY*vV.fX + fYY*vV.fY + fZY*vV.fZ,
               fXZ*vV.fX + fYZ*vV.fY + fZZ*vV.fZ);

   // Gebe rotierten Vektor zurück
   return vRes;
}
```

**2.3. Schnittgerade von zwei Ebenen berechnen**

Zwei Ebenen sind entweder Parallel zueinander oder schneiden sich in einer Gerade - da Ebenen unendlich sind also nimmernie in nur einem Punkt! :)
In diesem Artikel findet diese Funktion zwar keine Anwendung – ich wollte es aber zur Vollständigkeit hier auflisten.
Hier eine Funktion welche die Schnittgerade (falls vorhanden) von zwei Ebenen berechnet:

```
bool CPlane::PlaneIntersection(const CPlane &cPlane2, CRay &cRay) const
{
   // Diverse benötigte Infos sammeln
   const float *fPN = &cPlane2.fN[0];  // Zeiger auf die Normale der zweiten Ebene ermitteln
   float fN00 = fN[0]*fN[0] + fN[1]*fN[1] + fN[2]*fN[2];  // Quadratlänge von fN
   float fN01 = fN[0]*fPN[0] + fN[1]*fPN[1] + fN[2]*fPN[2];  // Skalarprodukt fN*fPN
   float fN11 = fPN[0]*fPN[0] + fPN[1]*fPN[1] + fPN[2]*fPN[2]; // Quadratlänge  von fPN

   // Gibt es seine Schnittgerade?
   float fDet = fN00*fN11 - fN01*fN01;
   if (fabs(fDet) < 1e-12)
      return false; // Nope, die Ebenen sind parallel

   float fInvDet = 1.0f/fDet;
   float fC0     = (fN11*fD - fN01*cPlane2.fD)*fInvDet;
   float fC1     = (fN00*cPlane2.fD - fN01*fD)*fInvDet;

   // Richtung der Gerade = fN kreuz fPN
   cRay.SetDir(fN[1]*fPN[2] - fN[2]*fPN[1],
               fN[2]*fPN[0] - fN[0]*fPN[2],
               fN[0]*fPN[1] - fN[1]*fPN[0]);
   cRay.SetPos(fN[0]*fC0 + fPN[0]*fC1,
               fN[1]*fC0 + fPN[1]*fC1,
               fN[2]*fC0 + fPN[2]*fC1);

   // Es gibt eine Schnittgerade
```

```
    return true;
}
```

### 2.4. Schnittpunkt von drei Ebenen berechnen

Drei Ebenen sind entweder Parallel zueinander, bilden Schnittgeraden oder einen Schnittpunkt der in Abschnitt „3.6. Ebene-Menge einschließende Kugel berechnen“ verwendet wird.
Hier eine Funktion welche den Schnittpunkt (falls vorhanden) von drei Ebenen berechnet:

```
bool CPlane::PlaneIntersection(const CPlane &cP2, const CPlane &cP3, CVector3D &vRes)
{
    CVector3D vN1(fN);
    CVector3D vN2(cP2.fN);
    CVector3D vN3(cP3.fN);

    // Ermittle ob sich die Ebenen Schneiden
    float fDenominator = vN1.DotProduct((vN2).CrossProduct(vN3));
    // Wenn null gibt es keinen Schnitt
    if (!fDenominator)
        return false; // Es gibt keinen Schnittpunkt

    // Berechne Schnittpunkt
    CVector3D vTemp1, vTemp2, vTemp3;
    vTemp1 = (vN2.CrossProduct(vN3))*fD;
    vTemp2 = (vN3.CrossProduct(vN1))*cP2.fD;
    vTemp3 = (vN1.CrossProduct(vN2))*cP3.fD;
    vRes = (vTemp1+vTemp2+vTemp3)/(-fDenominator);

    // Es gibt einen Schnittpunkt
    return true;
}
```

## 3. Ebenen Menge

### 3.1. Überblick

In der 3D Programmierung arbeitet man oft mit einem geschlossenen Volumen (convex volume) welches über eine bestimmte Anzahl von Ebenen definiert wird die sich gegenseitig 'anschauen' . Sprich, alle Ebenen Normalen zeigen 'in' das geschlossene Volumen. Mit dieser Menge an Ebenen kann man dann beliebige Dinge machen wie beispielsweise prüfen ob sich etwas darin befindet. Eine entsprechende Klasse welche mit mehren Ebenen jongliert könnte so aussehen:

```
class CPlaneSet {
    public:
        // Prüft ob ein Punkt im Volumen ist
        virtual bool IsPointIn(float fPoint[3]);
        // Prüft ob eine Sphere im Volumen ist
        virtual bool IsSphereIn(float fCenter[3], float fRadius);
        ....
    private:
```

```
    CPlane *m_pPlanes;      // Die Ebenen welche das Volumen bilden
    int    m_nNumOfPlanes;  // Anzahl der Ebenen
};
```

Es ist sinnig bestimmte Methoden dieser Klasse virtuell zu machen, so, dass von dieser Klasse abgeleitete Klassen noch weitere Funktionalität hinzufügen können.

### 3.2. Sichtbarkeitsbestimmung

Das Klassische Beispiel für eine Ebenen-Menge welches man zuhauf im Netz findet ist das "frustum culling" bei dem mithilfe der Projektion- und View-Matrize die Ebenen aufgestellt werden. Diese Ebenen Menge besteht aus Near- und Far-Ebenen sowie den 4 seitlichen Ebenen.
Eine genaue Beschreibung der Frustum-Culling-Verfahren, samt Implementierungsvorschlägen für OpenGL war auf der Seite von Mark Morley (www.markmorley.com) zu finden welche mittlerweile leider nicht mehr im Netz ist – aber zum Glück gibt's hier ein Backup:
http://crownandcutlass.sourceforge.net/features/technicaldetails/frustum.html

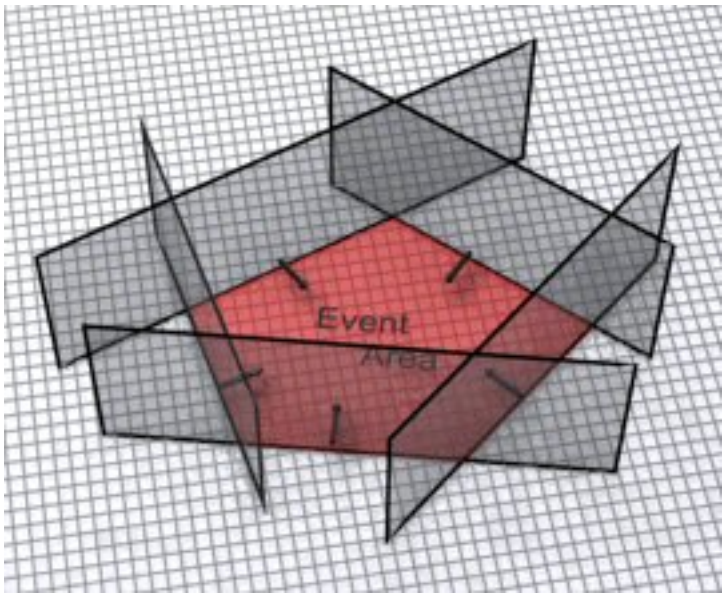
Dort wird des weiteren Beschrieben wie man prüfen kann ob ein Punkt/Kugel/Kiste im Frustum ist. (oder allgemeiner im durch die Ebenen gebildetem Volumen :)
Da auf dieser Seite bereits alles ausführlich steht, entschloss ich mich dazu das in diesem Artikel nicht noch mal aufzuwärmen.

Die folgende Klasse implementiert den Kamera Frustum. Diese bietet im Grunde 'nur' zusätzlich eine Funktion zum automatischen erzeugen der Ebenen – der Rest wie das Testen ob bestimmte Körper in diesem Volumen sind, ist ja bereits in CPlaneSet:

```
CFrustum : public CPlaneSet {
  public:
    // Erzeugt automatisch die Frustum Ebenen mit Hilfe einer Projektion- und View-Matrize
    void Build(float projection[4][4], float view[4][4]);
};
```

### 3.3. Ereignis-Zonen

Aus vielen Spielen kennt man das Phänomen das sobald man eine bestimmte Stelle im Level betritt etwas ausgelöst wird - es bricht beispielsweise eine Wand ein und tausende von Zombies schleifen sich stöhnend einem entgegen. Oder man hat eine Region in welcher man Lebensenergie aufgrund von Strahlung verliert. Kurzum, es passiert etwas sobald man ein Gebiet betritt.
Eine solche 'Zone' lässt sich einfach als eine Anzahl von Ebenen aufbauen welche ein geschlossenes Volumen darstellen. Hier eine Visualisierung einer ‚Event area':




Ist ein Punkt/Kugel/Kiste oder was auch immer darin, wird z.B. eine Nachricht gesendet das etwas passieren soll. Man erzeugt einfach eine neue Klasse welche von CPlaneSet abgeleitet ist und weitere spezialsiertere Dinge hinzufügt.

```
class CEventArea : public CPlaneSet {
   public:
      // Überladene Funktionen von CPlaneSet
      virtual bool IsPointIn(float fPoint[3]);
      virtual bool IsSphereIn(float fCenter[3], float fRadius);
   private:
      // Wird von den überladenen Funktionen von CPlaneSet aufgerufen wenn sich etwas
      // im Volumen befindet
      virtual void Event();
};

bool CEventArea::IsPointIn(float fPoint[3])
{
   // Basis Funktion von CPlaneSet aufrufen und falls der Punkt im Volumen
   // ist, so wird ein Ereignis auslöst
   if (CPlaneSet::IsPointIn(fPoint) == true)
      Event();
}
...
void CEventArea::Event()
{
   // Etwas ist im Volumen...
}
```

Natürlich wäre es noch nett zu wissen WAS GENAU denn nun im Volumen ist usw. - aber das hier ist ja nur ein gaannz primitives Beispiel. ;-)

### 3.4. Portale

Ein Portal ist im Grunde genommen ein 'Fenster' welches von einem Raum in einen anderen sehen lässt. 'Schaut' man durch dieses Portal ändert sich der Kamera-Frustum welcher oben beschrieben wurde – und dieser kann nun mehr als 6 Ebenen haben da ein Portal aus beliebig vielen Punkten bestehen kann – wobei das in der Praxis aber meist nur 4 sind um nicht ZU viele Clipping-Ebenen zu bekommen. ☺
Die Ebenen dieses neuen Sichtfeldes lassen sich am einfachsten Berechnen indem man jede Ebene aus drei Vektoren erzeugt wobei die Kamera Position als V1 oder so genommen wird und man für die die restlichen 2 Vektoren pro Ebene die Kanten des Portals durchläuft. Alles was man durch dieses Portal sieht wird dann mit Hilfe dieses neuen Frustums auf seine Sichtbarkeit hin überprüft werden.
Hier etwas Code der zeigt wie man die ‚seitlichen' Ebenen des Portal-Frustums berechnen kann:

```
CPlaneSet cPlaneSet;    // Menge von Ebenen welche den ‚Portal-Frustum' bilden
List<Vector3D> cVertexList;    // Liste welche alle Punkte des Portals beinhaltet
for (int i=0, j=1; i< cVertexList.GetNumOfElements(); i++, j++) {
   // Zeiger auf aktuelle zu berechnende Ebene ermitteln
   pPlane = cPlaneSet.GetPlane(i);

   // Die 2 Punkte der Portal Ecke bestimmen, mWorldMatrix ist die Welt-Matrize des Portals
   // um die im Object-Space liegenden Portal Punkte in den World-Space zu transformieren
   vP1 = mWorldMatrix.Transform(*cVertexList[i]);
   vP2 = mWorldMatrix.Transform(*cVertexList[j]);

   // Die neue Ebene berechnen, vCamPos ist die aktuelle Kamera-Position im World-Space
   pPlane->ComputeND(vP1, vP2, vCamPos);

   // Wrap j
   if (j == cVertexList.GetNumOfElements()-1)
      j = -1;
}
```

Die Punkte welche das Portal bilden müssen in der Praxis alle auf einer Ebene liegen. Diese Ebene speichert man dann am besten auch im Portal gleich mit. Da das Portal zwei Seiten hat, und man im Normalfall nur von einer Seite aus in den durch das Portal sichtbaren ‚Raum' sehen kann, ist es sinnvoll bevor man den Portal-Frustum berechnet erstmal zu prüfen ob man gerade durch die ‚sichtbare' Seite schaut oder durch die Unsichtbare. Im letzteren Fall kann man sich alle weitere Arbeit sparen da man von der aktuellen Kamera Position aus durch das Portal sowieso nichts sehen kann.
Hier etwas Code der prüft auf welcher Seite einer Portal-Ebene sich die Kamera befindet.

```
// Genauso wie die Portal-Punkte liegt auch die Portal-Ebene im Object-Space des
// Portals und muss daher zuerst einmal in den World-Space gebracht werden. Optional kann
// natürlich auch mit der Inversen der Portal-Welt-Matrize die Kamera Position in den
// Object-Space des Portals transformieren werden was sogesehen sogar geschickter wäre da
// dies weniger Aufwand ist. Aber ich will hier einfach mal den in „2.2. Ebene*Matrize“
// überladenen Operator * der Ebenen-Klasse in der Praxis zeigen.
CPlane cPlane = cPortalPlane*mWorldMatrix;
// Prüfe auf welcher Seite der Portal-Ebene sich die Kamera befindet
float fDistance = cPlane.GetDistance(vCamPos);
if (fDistance > 0.0f) { // Wir sind auf der Unsichtbaren Seite!
```

```
    return; // Also nüschts zu sehen von hier aus
}

// Berechnet den Abstand eines Punktes zu der Ebene
inline float CPlane::GetDistance(const CVector3D &vPoint) const
{
    return fN[0]*vPoint.fX + fN[1]*vPoint.fY + fN[2]*vPoint.fZ + fD;
}
```

### 3.5. Selektieren über einen Rahmen ziehen

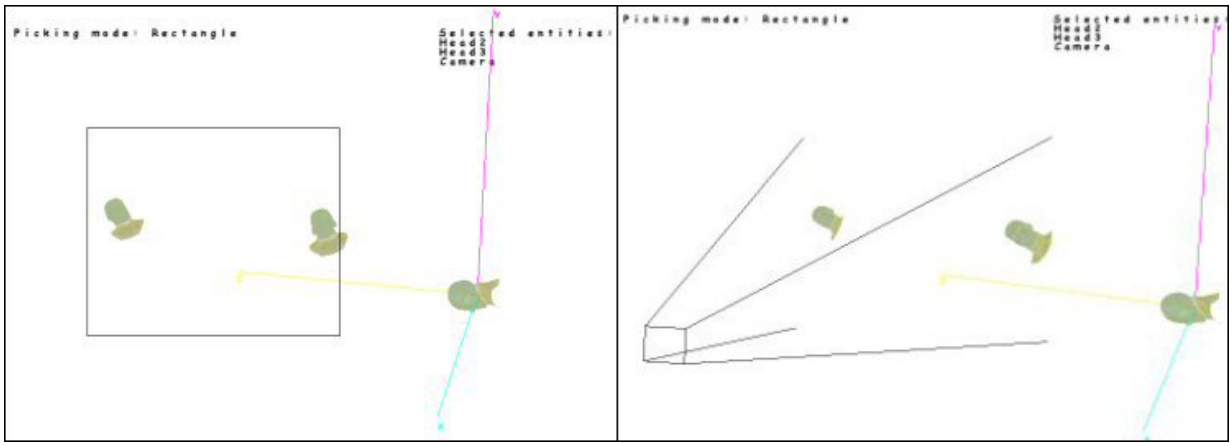
Das Problem "selektieren über einen Rahmen ziehen" ist im Grunde nichts anderes als eine einfache Sichtbarkeitsbestimmung wie wir sie von oben her kennen. Das niedliche kleine Rechteck welches man auf seinem Bildschirm ‚großgezogen' hat wird quasi zum ‚guckloch' in die 3D Welt – also im Prinzip das gleiche wie beim Kamera-Frustum. Hat man erstmal die nötigen Ebenen für den dieses ‚Selektions-Volumen' berechnet, ist das prüfen WAS Selektiert wurde daher wirklich nichts neues und alles dafür nötige wurde bereits in CPlaneSet implementiert.
Um diese Ebenen zum Selektieren zu erzeugen entschloss ich mich dazu die einfachste Variante zu nehmen welche man auch problemlos Bildlich darstellen kann... Der Benutzer ‚zieht' auf dem 2D Bildschirm einen Rahmen mithilfe eines Startpunktes und eines Endpunktes - diesen Rahmen kann man ganz einfach über 2D Linien anzeigen lassen.
Da wir zum berechnen der einzelnen Ebenen je 3 Punkte im Raum benötigen welche auf einer Ebene liegen, müssen wir nun irgendwie die 3D Koordinaten dieser 2D Koordinaten vom Bildschirm finden. Dazu verwende ich eine Funktion welche mithilfe des Viewports, der View- und Projektion-Matrize die 3D Welt Position eines gegebenen 2D Bildschirm Punktes ermittelt. Eine solche Funktion könnte unter OpenGL z.B. folgendermaßen aussehen:

```
CVector3D Get3DCoordinate(int nX, int nY, float *pfDepth)
{
    double fModelViewMatrix[16], fProjectionMatrix[16];
    CVector3D v3DCoord;
    double fX, fY, fZ;
    int nViewport[4];
    float fDepth;

    // Ermittle aktuellen Viewport
    glGetIntegerv(GL_VIEWPORT, nViewport);
    nY = -nY+nViewport[3];
    // Ermittle die ‘tiefe’ der gegebenen 2D Koordinate
    if (!pfDepth) {
        // Lese die ‘tiefe’ aus dem Z-Buffer an der gegebenen Stelle
        glReadBuffer(GL_BACK);
        glReadPixels(nX, nY, 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &fDepth);
    } else {
        // Verwende als ‘tiefe’ die vom Benutzer übergebene
        fDepth = *pfDepth;
    }
    // Ermittle Modelview- und Projektion-Matrize
    glGetDoublev(GL_MODELVIEW_MATRIX,  fModelViewMatrix);
    glGetDoublev(GL_PROJECTION_MATRIX, fProjectionMatrix);
```

```
    // Ermittle die 3D Koordinaten mithilfe der gegebenen 2D Koordinate
    // und der ‚tiefe'
    gluUnProject(nX, nY, fDepth, fModelViewMatrix, fProjectionMatrix,
            nViewport, &fX, &fY, &fZ);
    v3DCoord.fX = (float)fX;
    v3DCoord.fY = (float)fY;
    v3DCoord.fZ = (float)fZ;

    // Fertisch
    return v3DCoord;
}
```

… soweit ich weis kann man unter Direct3D nicht ohne weiteres wie unter OpenGL direkt Werte aus dem Z-Buffer auslesen. Aber da wir für das ‚Selektions-Volumen' die Tiefe selbst entweder auf 0 (near) oder 1 (far) setzen ist das weniger dramatisch. ☺

Diese Funktion wird nun dafür verwendet um das ‚Selektions-Volumen' zu erzeugen welches im Grunde eine einfache (durch die Projektion 'verzerrte') Kiste mit 4 Punkten auf der Near-Ebene (tiefe = 0) und 4 Punkten auf der Far-Ebene (tiefe = 1) ist. Alles was man nun nur noch machen muss ist die Ebenengleichungen für jede der 6 Kisten Seite mithilfe von 3 Punkten von jeder Seite zu berechnen.
Will man sich das 'Selektions-Volumen' anzeigen lassen muss man im Prinzip nur diese 8 Punkte der Kiste an der aktuellen Kamera Position speichern entsprechend durch Linien verbunden Zeichnen… und sobald sich die Kamera an eine neue Stelle bewegt sieht man diesen alten ‚Selektions-Trichter'. ☺




Diese Screenshots aus einem meiner Test-Programme zeigt das oben Beschriebene

Die gesamte Funktion welche als Parameter die Start- und Endposition des Rechtecks auf dem Bildschirm übernimmt und automatisch die Ebenen von CPlaneSet korrekt setzt könnte so aussehen:

```
void BuildSelectionPlaneSet(CPlaneSet &cPlaneSet,
    const CVector2D &vStartPos, const CVector2D &vEndPos)
{
    // Ermittle die Ecken 'links-oben' und 'rechts-unten'
    CVector2D v2DLeftTop, v2DRightBottom;
    if (vStartPos.fX < vEndPos.fX) {
```

```
      v2DLeftTop.fX     = vStartPos.fX;
      v2DRightBottom.fX = vEndPos.fX;
    } else {
      v2DLeftTop.fX     = vEndPos.fX;
      v2DRightBottom.fX = vStartPos.fX;
    }
    if (vStartPos.fY < vEndPos.fY) {
      v2DLeftTop.fY     = vStartPos.fY;
      v2DRightBottom.fY = vEndPos.fY;
    } else {
      v2DLeftTop.fY     = vEndPos.fY;
      v2DRightBottom.fY = vStartPos.fY;
    }

    // Ermittle die 3D Koordinaten des ‚Selektions-Volumen‘
    float fNear = 0.01f, fFar = 1.0f;
    CVector3D v3DLeftTopNear = Get3DCoordinate((int)v2DLeftTop.fX,
                               (int)v2DLeftTop.fY, &fNear);
    CVector3D v3DLeftBottomNear = Get3DCoordinate((int)v2DLeftTop.fX,
                                  (int)v2DRightBottom.fY, &fNear);
    CVector3D v3DRightTopNear = Get3DCoordinate((int)v2DRightBottom.fX,
                                (int)v2DLeftTop.fY, &fNear);
    CVector3D v3DRightBottomNear = Get3DCoordinate((int)v2DRightBottom.fX,
                                   (int)v2DRightBottom.fY, &fNear);
    CVector3D v3DLeftTopFar = Get3DCoordinate((int)v2DLeftTop.fX,
                              (int)v2DLeftTop.fY,     &fFar);
    CVector3D v3DLeftBottomFar = Get3DCoordinate((int)v2DLeftTop.fX,
                                 (int)v2DRightBottom.fY, &fFar);
    CVector3D v3DRightTopFar = Get3DCoordinate((int)v2DRightBottom.fX,
                               (int)v2DLeftTop.fY, &fFar);

    // Erzeuge die Ebenen
    cPlaneSet.Clear();
    // Near-Ebene
    cPlaneSet.Create()->ComputeND(v3DLeftTopNear, v3DLeftBottomNear,
                                  v3DRightTopNear);
    // Linke Ebene
    cPlaneSet.Create()->ComputeND(v3DLeftBottomNear, v3DLeftTopNear,
                                    v3DLeftTopFar);
    // Rechte Ebene
    cPlaneSet.Create()->ComputeND(v3DRightBottomNear, v3DRightTopFar,
                                  v3DRightTopNear);
    // Obere Ebene
    cPlaneSet.Create()->ComputeND(v3DLeftTopNear, v3DRightTopNear,
                                  v3DLeftTopFar);
    // Untere Ebene
    cPlaneSet.Create()->ComputeND(v3DRightBottomNear, v3DLeftBottomNear,
                                  v3DLeftBottomFar);
}
```

Wie oben bereits erwähnt kann man das alles ganz bestimmt noch mit netten Mathematischen Formeln etc. wesentlich Effektiver machen – aber so wie ich es machte kann man leicht nachvollziehen wie es funktioniert. ;-)

### 3.6. Ebene-Menge einschließende Kugel berechnen

Oft ist es für die Performance gut vor Umfangreicheren Tests erstmal grob zu prüfen ob diese denn überhaupt nötig sind. So ist es Beispielsweise sinnvoll die Kugel zu berechnen welche eine Ebene-Menge ‚einschließt'. Denn nichts ist schneller gemacht als Kugel-Kontakte zu prüfen.
Hier zum Beweis Code der prüft ob sich ein Punkt in einer Kugel befindet: (Kugel hat Position und Radius)

```
bool CSphere::CheckPoint(const CVector3D &vPos) const
{
   if ((vPos-m_vPos).DotProduct(vPos-m_vPos) <= m_fRadius*m_fRadius) return true;
   else return false;
}
```

Ist Beispielsweise ein Körper nicht in der eine Ereignis-Zone einschließenden Kugel, so weist man das dieser Körper nicht in dieser Zone sein kann und die weiteren xxx Ebenen-Tests kann man sich somit gleich sparen.
Die einfachste (aber dafür nicht so flotte) Möglichkeit um eine solche einschließende Kugel zu ermitteln liegt darin, die Schnittpunkte aller Ebenen zu berechnen, dann alle Schnittpunkte zusammenzuzählen und durch die Anzahl der Schnittpunkte zu teilen um den Kugel Mittelpunkt zu erhalten. Nun nur noch den maximalen Abstand von diesem Mittelpunkt zu allen Schnittpunken ermitteln und man erhält den Kugel-Radius. Also alles in allem ein enormer Aufwand – aber solange diese Kugel nicht ZU OFT Berechnet werden muss kann man damit Leben.
Hier nun also die aufwändige Funktion um die eine Ebene-Menge einschließende Kugel zu ermitteln:

```
void CPlaneSet::CalculateSphere(CSphere &cSphere)
{
   CArray<PLTVector3D> lstPoints;
   CVector3D vRes, vD, *pvP;
   CPlane *pP1, *pP2, *pP3;
   int nNumOfPoints = 0;

   // Kugel initialisieren
   cSphere.SetPos();
   cSphere.SetRadius();

   // Finde alle Schnittpunkte der Ebenen
   for (int nP1=0; pP1=m_lstPlane[nP1]; nP1++) {
      for (int nP2=0; pP2=m_lstPlane[nP2]; nP2++) {
         if (nP2 == nP1) continue;
         for (int nP3=0; pP3=m_lstPlane[nP3]; nP3++) {
            if (nP3 == nP1 || nP3 == nP2) continue;
            // Schneiden sich diese 3 Ebenen? Funktion aus Abschnitt
            // „2.4 Schnittpunkt von drei Ebenen berechnen"
```

```
            if (!pP1->PlaneIntersection(*pP2, *pP3, vRes)) continue;
            // Jup, diesen Schnittpunkt zur Liste der Schnittpunkte hinzufügen
            lstPoints.Add(vRes);
            vD += vRes;
         }
      }
   }

   // Keine Schnittpunkte – keine einschließende Kugel
   if (!lstPoints.GetNumOfElements()) return;

   // Ermittle die Position der Kugel
   vD /= (float)lstPoints.GetNumOfElements();
   cSphere.SetPos(vD);

   // Ermittle den Radius der Kugel
   float fLength, fMaxLength = 0.0f;
   for (int i=0; pvP=lstPoints[i]; i++) {
      fLength = (*pvP-vD).GetLength();
      if (fLength > fMaxLength) fMaxLength = fLength;
   }
   cSphere.SetRadius(fMaxLength);
}
```

**4. Weitere Interessante Anwendungs-Beispiele für Ebenen**

**4.1. Überblick**

In diesem Abschnitt finden sich weitere Interessante Anwendungsbeispiele/Problemlösungen für Ebenen. Jedoch findet man hier nur kurze Beschreibungen und eventuelle Verweise auf Quellen – all diese Dinge ausführlich zu behandeln würde den Rahmen dieses Artikels bei weitem sprengen. ;-)

**4.2. "oblique near-plane clipping"**

Oft ist es notwendig Teile der 3D Welt 'Wegzuschneiten' (Englisch: Clipping) - beispielsweise wenn man Spiegelungen (z.B. Wasserreflektion) Darstellen will. Zwar bieten die APIs wie DirectX oder OpenGL 'Clipping'-Ebenen an jedoch werden diese nicht immer korrekt von allen Grafikkarten unterstützt, sind teils langsam und funktionieren im zusammenspiel mit Shadern nicht immer korrekt - sind also universell gesehen nicht brauchbar, außer für spezielle Situationen. Über www.gamedev.net fand ich eine äußerst Interessante und universelle Methode wie man mithilfe einer Ebene Clipping 'faken' kann. Funktioniert zwar NUR mit einer Ebene aber das ist meistens völlig ausreichend. :)
Diese Technik nennt sich "oblique near-plane clipping" ist unter http://www.terathon.com/code/oblique.html zu finden.

**4.3. Binary Space Partitioning Trees (BSP)**

‘Binary Space Partitioning Trees’ wurden früher, als man noch jedes Polygon sparen musste bei Videospielen u.a. dazu eingesetzt die Sichtbarkeitsbestimmung Performant zu machen. Mittlerweile haben jedoch andere Datenstrukturen wie beispielsweise Octrees BSP’s für diese Aufgabenbereiche abgelöst da die heutigen GPU’s besser damit zurechtkommen ganze

Polygon Haufen auf einmal auszuspucken als viele einzelne Polygone.
Ebenen spielen bei BSP’s eine große Rolle da eine Ebene dazu verwendet wird z.B. ein Level in zwei (möglichst ausgewogene) Teile zu zerlegen. (daher binary :)
Jeder dieser neuen Teilbereiche wird nun so lange weiter durch eine Ebene in zwei hälften geteilt bis in bestimmtes Abbruchkriterium erreicht wird.
Will man dann z.B. prüfen in welchem ‚Teilbaum’ sich die Kamera gerade befindet, wird Rekursiv geprüft auf welcher Seite der Schnitt-Ebene sich die Kamera befindet.
Weitere Informationen zu BSP findet man tonnenweise im Netz.