AUF CD 1

Die Quelltexte sowie die fertig übersetzten Routinen finden Sie im Verzeichnis *Heft Add-ons/Programmierung/PC Underground.*

Cg – C for Graphics

Leben im Fraktal

Cg ist C für Grafik. Damit lassen sich die Vertex- und Fragment-Shader der neuen Grafikkarten mit einer Hochsprache programmieren. Sparen Sie sich den steinigen Weg über Low-Level-Assembler.

CARSTEN DACHSBACHER

Vor kurzem hat nVidia das Cg-Toolkit vorgestellt. Mit diesem Werkzeug gelingt es, die Vertex- und Fragment-(Pixel-)Shader der neueren Grafikkarten in C zu programmieren. Dieser Abstraktionsschritt von der Hardware und der Assembler-Programmierung erlaubt es, schnell und vor allem universell verwendbare Shader zu programmieren, ohne zu sehr auf die genaue darunterliegende Hardware einzugehen. Dabei ist Cg über OpenGL bzw. DirectX angesiedelt und daher API-unabhängig.

Der tatsächliche Assembler-Code der Vertex-Shader wird während der Laufzeit erzeugt. Deshalb sind die Cg-Programme plattform- und vor allem Hardware-unabhängig. Momentan ist Cg noch eine nVidia-Domäne, aber es ist zu erwarten, dass auch weitere Grafikkartenhersteller sich anschließen.

Obwohl die Cg-API unabhängig ist, unterscheiden sich die API-Interfaces, um die Programme auf Cg-Basis zu verwenden. In dieser Ausgabe lernen Sie das OpenGL-Interface kennen, um Cg-Vertex-Shader zu programmieren.

INPUT SEMANTIC BINDINGS – UNTERSTÜTZT VON ALLEN PROFILEN

Bezeichner	Bedeutung
POSITION	Koordinate eines Vertex
BLENDWEIGHT	Vertex-Interpolationswert
COLOR0, COLOR1	Farbwerte pro Vertex
NORMAL	Vertexnormale
TEXCOORD0..7	Texturkoordinaten
BINORMAL	Vertex-Binormale, identisch mit TEXCOORD6
TANGENT	Vertex-Tangente, identisch mit TEXCOORD7
TESSFACTOR	Tesselierungsfaktor
PSIZE	Punktgröße für GL_POINTS
ATTR0...15	Alternative Stream-Bezeichnung, wie bei nVidia Vertex Programs

Fakten über Cg

Momentan unterstützt Cg die Programmierung von Vertex- und Fragment-Shadern für die GeForce-2/3/4-Grafikkarten unter DirectX und OpenGL und die CineFX- und NV3x- Features. nVidia hat die CineFX-Architektur dem Cg-Toolkit in der Version Beta 2 schon hinzugefügt. Über die nVidia-NV3x-Architektur finden Sie über die Google-Suchmaschine zahlreiche Einträge wie: www1.sharkyextreme.com/hardware/videocards/article.php/1434621

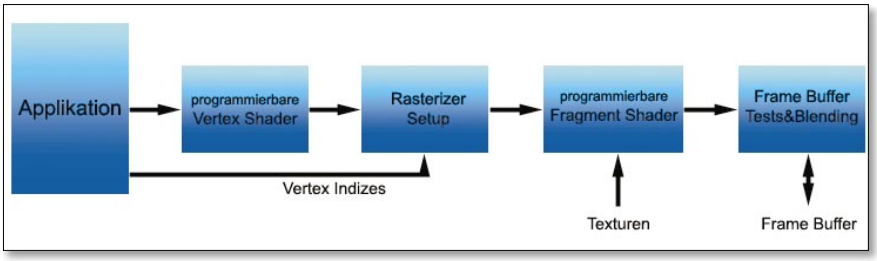
Zudem wird NV3x mit den neuesten Grafikkartentreibern per Software emuliert, da die zugehörige Hardware noch nicht verfügbar ist.

Da der Cg-Compiler aus dem Cg-Quelltext beispielsweise die Vertex-Shader erzeugt, bleibt die maximale Anzahl (und damit Länge) von 128 Instruktionen bestehen, weil dies die Hardware bestimmt. Ist das Kompilat länger, muss der Programmierer selbst Hand anlegen und die Berechnungen vereinfachen oder optimieren. Cg nimmt eine Abstraktion von der Assemblersprache und somit der Register vor. Dadurch ist es einfacher, zwei unabhängige Vertex-Shader miteinander zu kombinieren. Wer auf Assembler-Ebene programmieren wollte, müsste jedoch unter Umständen die Konstanten- oder Registerbelegung modifizieren.

Installation des Cg-Toolkit

Sie benötigen das nVidia SDK bzw. Cg-Toolkit, das Sie auf der nVidia Homepage unter www.nvidia.com finden. Sollten Sie schon eine ältere Version davon besitzen, benötigen Sie trotzdem die aktuelle Version, um die Beispiele zu dieser Ausgabe kompilieren zu können. Bislang gibt es nur Betaversionen des Compilers, was sich auch bei einigen Programmcode-Konstellationen auswirkt. Nach der Installation finden Sie im Installationsordner das Unterverzeichnis *msdev_syntax_highlighting*. Darin enthalten ist eine Datei mit Registry-Einträgen, damit das Syntax Highlighting der Cg-Befehle in der Visual C++ IDE funktioniert. Außerdem sollten Sie das *bin*-Unterverzeichnis zur *PATH*-Systemvariable hinzunehmen. Als letztes müssen Sie in Visual C++ unter dem Menüpunkt *Tools/Options/Directories* die *Include*- und *Library*-Pfade der Cg-Dateien setzen. Zu Ihren Programmen müssen Sie später die *cg.lib*- und *cgGL.lib*-Dateien linken.

Damit können Sie bereits mit dem Cg-Compiler (*cgc.exe*) Progamme kompilieren und den Output in einer Textda-



DIE WEGE der Daten in einer Grafikkarte

tei betrachten. Im Folgenden werden wir die OpenGL-Funktionen verwenden und die Cg-Programme zur Laufzeit kompilieren.

Cg-Programme besitzen immer die Dateiendung *.cg*. Cg-Programme können Pixel- oder Fragment-Shader für verschiedene Grafik-Hardware darstellen. Diese werden in Cg mit Profilen unterschieden. Es gibt beispielsweise ein Profil für Vertex Programme und Fragment-Shader für GeForce-Karten und ein Profil für Vertex-Programme nach der ARB-Vertex-Programm-Erweiterung.

Cg-Vertex-Shader

Ein Cg-Vertex-Programm enthält mindestens eine Funktion, die, anders als bei C-Programmen, nicht *main()* heißen muss. Sie können auch weitere Subfunktionen deklarieren und verwenden.

Ein Vertex-Programm wird für jeden Vertex, der die OpenGL-Pipeline passiert, ausgeführt. Dabei ist es ausgeschlossen, berechnete Werte von einer Instanz des Vertex-Programms ans nächste zu übergeben.

Die Eingabedaten eines Vertex-Programms sind zum einen die *Varying Inputs*. Diese Daten stehen pro Vertex zur Verfügung, sind also in erster Linie Koordinaten und Attribute wie Textur-Koordinaten oder Farbwerte. Sie werden in OpenGL mit den *Immediate Mode*- oder den *Streaming*-Befehlen an die OpenGL-Pipeline übergeben.

Im Cg-Programm müssen Sie angeben, welche Varying Inputs Sie verwenden wollen und mit welchen Variablennamen Sie diese adressieren wollen. Dazu definieren Sie eine Struktur, in der alle Eingabedaten pro Vertex angegeben sind. Hinter jeder Variablen, getrennt durch einen Doppelpunkt, wird das *Binding Semantic*, die Belegung der Variablen, geschrieben. Damit legen Sie z.B. fest, welche Variable welche Bedeutung und somit welche Attribute enthält:

```
struct myVaryingInput

{
  float4 myPosition : POSITION;
  float3 myNormal : NORMAL;
  float4 myColor : COLOR0;

};
```

Eine Liste der definierten Datentypen finden Sie in der Textbox *Datentypen in Cg* auf Seite 212, der Binding Semantics in den Tabellen *Input*- sowie *Output Semantic Bindings* auf Seite 210/211. Analog zu den Bezeichnern in den Tabellen können die nVidia-Vertex-Programme weitere Bezeichner verwenden. Dazu gehören unter anderen *HPOS, COL0, COL1, BCOL0, BCOL1, TEX0-TEX7, FOGC, PSIZ*.

Genauso wie die Varying Inputs definieren Sie die Varying Outputs, womit Sie die Resultate Ihres Vertex-Shaders an die Rasterizer-Einheit der Grafikkarte übergeben. Wenn Sie beispielsweise einen einfachen Shader programmieren, der die Beleuchtung berechnet, übergeben Sie die transformierte Koordinate und einen Farbwert:

```
struct myVaryingOutput
{
  float4 myHPosition : POSITION;
  float4 myOutputColor : COLOR0;
};
```

Die zweite Form von Daten sind die Uniform Inputs, die sich nicht für jeden Vertex ändern und separat angegeben werden. Typischerweise gehören die Transformationsmatrix oder andere pro Frame bzw. 3D-Objekt konstante Werte dazu. Diese Daten geben Sie im Funktionskopf der Hauptfunktion des Vertex-Shaders an. Unser Beispiel Cg-Programm soll *cgMain* heißen und ist wie folgt deklariert:

```
myVaryingOutput cgMain
  ( myVaryingInput in,
    uniform float4x4
    modelviewProjection )
{
...};
```

Dies bedeutet, dass Sie – wie in normalem C – eine Funktion haben, die als Parameter eine *myVaryingInput*-Struktur und eine Matrix bezeichnet, durch *modelviewProjection* entgegen nimmt und eine *myVaryingOutput*-Struktur zurückliefert. Diese Funktion wird für jeden Vertex ausgeführt.

Cg-Programm in OpenGL

Bevor Sie ein spezielles Cg-Programm entwickeln, sehen Sie zunächst, wie Sie solche Programme in OpenGL einbinden. Als erstes erzeugen Sie einen Cg-Kontext. Dabei sollten Sie immer die Fehlercodes abfragen, damit Ihr Programm, z.B. bei falsch geschriebenen Variablennamen, nicht abstürzt.

```
CgContext = cgCreateContext();
assert( CgContext != NULL );
```

Wenn Sie den Kontext erfolgreich angelegt haben, können Sie anschließend das Cg-Programm per Quelltextdatei schreiben und laden:

```
cgError errorCode;
errorCode =
  cgAddProgramFromFile(
    CgContext,
    „test.cg“,
    cgVertexProfile, NULL );
assert(CgProgram != NULL);

cgProgramIter *CgProgram = NULL;

CgProgram =
  cgProgramByName( CgContext,
„cgMain“ );
assert(CgProgram != NULL);
```

Als Parameter benötigen Sie jeweils den Kontext. Mit *cgVertexProfile* geben Sie das Compiler-Profil an. Das sagt, ob es sich um einen Vertex- oder Fragment-Shader handelt und welche GPU angesprochen wird. Diese Konstanten definieren Sie in der *cg.h*-Datei.

Weitere Cg-Shader wie diese inklusive Source Code finden Sie unter www.cgshaders.org .

OUTPUT SEMANTIC BINDINGS

Bezeichner	Bedeutung
POSITION	Transformierte Vertexkoordinate
FOG	Fog-Wert
COLOR0, COLOR1	Farbwerte
PSIZE	Punktgröße für GL_POINTS
TEXCOORD0..7	Texturkoordinaten

Wenn Sie den erzeugten Vertex Shader Assembler Code betrachten möchten, können Sie sich diesen in einem String übergeben lassen. Dazu verwenden Sie folgende Funktion:

```
char *vp = (char*)
          cgGetProgramObjectCode
                ( CgProgram );
```

Wenn Sie diese Schritte durchgeführt haben, müssen Sie nur noch auf die *uniform*-Variablen des Vertex Shaders zugreifen können. Diesen Zugriff erhalten Sie über einen Zeiger auf eine *cgBindIter*-Struktur. Die Struktur erhalten Sie, wenn kein Fehler wie bei falschen Variablennamen auftritt, mit:

```
cgBindIter
 *CgBindModelviewProjection =
                        NULL;

CgBindModelviewProjection =
  cgGetBindByName( CgProgram,
      „modelViewProjection“ );
```

Die Inhalte der entsprechenden Variablen setzten Sie über die Zugriffsfunktionen:

• Dazu gehören die *cgGLBindUniform4[f,d][v]*-Befehle, mit denen Sie einen *float*-Wert oder Vektor übergeben können. Der erste Parameter ist dabei immer das Cg-Programm, also vom Typ *cgProgramIter*, der zweite Wert ist der Variablen-Identifier, also vom Typ *cgBindIter*.

• Weiterhin gehören die *cgGLBindUniformMatrix[c,r][f,d]*-Befehle dazu, mit denen Sie den Wert von *uniform*-Variablen der Spalten- bzw. Zeilenmatrizen darstellen und setzen.

• Der wichtigste Befehl ist *cgGLBindUniformStateMatrix*. Damit aktivieren Sie das Matrix-Tracking wie bei den nVidia-Vertex-Programmen: Eine Variable eines Cg-Programms enthält immer die aktuelle abgegebene Transformationsmatrix, also die Modelview, die Projektion oder wie in unserem Beispiel die Konkatenation (Verkettung von benachbarten Symbolen) aus Modelview und Projection Matrix. Außerdem können Sie angeben, ob die Matrix übernommen oder invertiert werden soll:

```
cgGLBindUniformStateMatrix
(
  CgProgram,
  CgBindModelviewProjection,
  cgGLModelViewProjectionMatrix,
  cgGLMatrixIdentity
);
```

Um ein Cg-Programm für das Rendering zu verwenden, müssen Sie es selektieren und aktivieren. Zum Aktivieren müssen Sie wieder das Profil angeben, also in unserem Beispiel *cgVertexProfile* für einen GeForce 3 Vertex Shader:

```
cgGLBindProgram( CgProgram );
cgGLEnableProgramType
            ( cgVertexProfile );
```

Jetzt können Sie die *varying*-Daten pro Vertex an OpenGL übergeben. Dies kann mit den *glVertexPointer* oder *glInterleavedArrays*-Befehlen und *glDrawArrays/glDrawElements*-Befehlen geschehen oder mit den *Immediate Mode*-Befehlen wie *glVertex3f(...)*. Nachdem Sie die 3D-Objekte gezeichnet haben, schalten Sie das Cg-Programm wieder ab:

```
cgGLDisableProgramType
            ( cgVertexProfile );
```

Am Ende Ihres Programms geben Sie die Cg-Pointer wieder frei. Als erstes die Variablen vom *cgBindIter*-Typ mit dem Befehl:

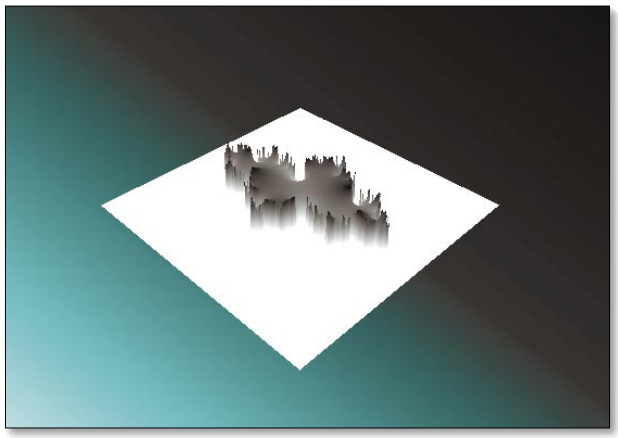
```
cgFreeBindIter(...)
```

Das Programm mitsamt Kontext räumen Sie mit folgenden Befehlen auf:

```
cgFreeProgramIter( CgProgram );
     cgFreeContext( CgContext );
                  cgCleanup();
```

Cg-Julia Fraktal

Ein etwas unkonventionelles Beispiel stellen wir Ihnen im Folgenden vor. Sie können Ihre Grafikkarte mit Hilfe eines Cg-Vertex-Shaders dazu verwenden, animierte Julia-Fraktalgebirge darzustellen.



DAS JULIA-FRAKTALGEBIRGE auf dem Cg-Vertex-Shader

Dazu rendern Sie später ein Polygongitter, dessen *x*/*z*-Koordinaten als Startwerte der Iteration dienen. Den Höhenwert, die *y*-Koordinate, lassen Sie vom Cg-Vertex-Shader berechnen. Die Berechnung eines Julia Fraktals erfolgt iterativ, das Ergebnis dient wiederum als Eingabewert, bis eine bestimmte Abbruchbedingung erreicht wurde. Als initialer Eingabewert dient ein zweidimensionales Koordinatenpaar *(x, y)*. Sie berechnen das neue Paar *(x', y')* wie folgt, wobei *a* und *b* zwei zeitabhängige Parameter sind, die das Fraktal animieren:

$$x' = x^2 - y^2 + a$$

$$y' = 2xy + b$$

Die Abbruchbedingung ist erfüllt, wenn x^2+y^2 größer als ein festgelegter Wert ist. In einem Vertex-Programm können Sie einige Iterationsschritte, die auf maximal 128 Instruktionen begrenzt sind, durchführen. Solange die Abbruchbedingung nicht erfüllt ist, erhöhen Sie einen Zähler. Diesen Zähler verwenden Sie als

DATENTYPEN IN CG

Name	Daten
float	32 Bit IEEE Floating Point Zahl
half	16 Bit IEEE Floating Point Zahl (nur für NV30 Fragment Shader)
fixed	12 Bit Fixed Point Zahl (nur für NV30 Fragment Shader)
bool	Boolsche Variable

Auf der Basis der obigen, einfachen Datentypen sind Vektortypen definiert, wie *float4, float3, float2, float1, bool4, bool3, bool2* und *bool1*. *float3* ist ein dreidimensionale Vektor, *float4* wird für homogene Koordinaten verwendet. Außerdem sind Matrixdatentypen bis zur Größe von 4x4 Matrizen definiert. Ihre Bezeichner sind z.B. *float1x1, float2x3* oder *float4x4*.

Strukturen können Sie in Cg wie von C bekannt definieren und verwenden:

```
struct myStruct{
...
};

myStruct s;
```

Auch Arrays können Sie wie in C-Code deklarieren, allerdings müssen Sie Unterschiede beachten: Cg unterstützt keine Pointer. Deshalb ist die Verwendung von Arrays eingeschränkt: zum einen in der Deklaration, zum anderen bei Aufrufen von Subfunktionen: Dabei werden Arrays kopiert und nicht die Referenz übergeben.

Höheninformation, um das Fraktalgebirge zu rendern.

Sie können die Berechnung etwas vereinfachen und umstellen und in einem Cg-Programm mit folgenden Variablen umsetzen:

```
float x, y, x2, y2, counter;

// Initialisierung
x0 = in.myPosition.x;
y0 = in.myPosition.z;
x2 = x * x;
y2 = y * y;
counter = 0.0;
incr = 1.0;
```

Ein Iterationsschritt sieht dann wie folgt aus:

```
y = 2.0 * x * y + b;
x = x2 - y2 + a;
x2 = x * x;
y2 = y * y;
incr =
 ( x2 + y2 > 4.0 ) ? 0.0 : 1.0;
counter += iter;
```

Eine Schwäche der Betaversion des Cg-Compilers: Wenn Sie diesen Iterationsschritt mehrfach ausführen, werden die Register des Vertex-Shaders nicht genügen. Der Grund dafür ist, dass der Compiler die Zwischenergebnisse des Vergleichs (bei *incr*) speichert und die Register nicht wieder überschreibt. Bei einem handoptimierten Vertex-Programm wäre nur die Programmlänge ein begrenzender Faktor. Eine Weg wäre, die Berechnung zu optimieren und die obige zu ersetzen. Dazu legen Sie folgende Variablen an:

```
float4 f1 =
float4(

in.myPosition.x,

in.myPosition.z,
0.0,
-in.myPosition.z
);
float4 f2 =
float4( a, 0.0,
0.5*b, -0.5*b );
```

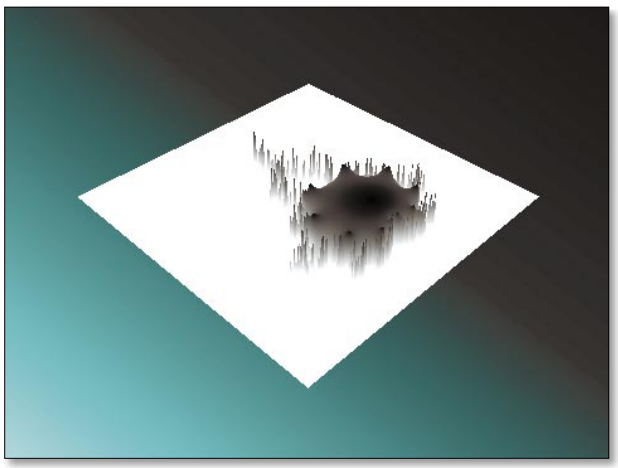
Ein Iterationsschritt lässt sich mit den *Swizzle-Operatoren* (komponentenweise vertauschen/ersetzen) in zwei Zeilen ausdrücken. VollziehenSie folgende Berechnung, die aus einem nVidia-Dokument stammt, auf einem Blatt Papier nach:

```
float4 temp;
temp = f1.xyxx * f1.xyyw + f2;
f1.xyzw = temp.xzww - temp.ywwz;
```

Dieser Vergleich lässt sich wie folgt formulieren:

```
incr = (float)
 ( dot( r0.xyzz, r0.xyzz )
                  > 4.0f );
```

So können Sie Register einsparen und mehrere Iterationsschritte ausführen. Dieser Bug dürfte in den nächsten Compilerversionen behoben sein.



UNSER BEISPIELPROGRAMM kann auch Mandelbrotmengen berechnen.

Jetzt muss Ihr Vertex Shader nur noch die Ausgabewerte an die *Fragment*-Abteilung der Grafikkarte übergeben. Dazu definieren Sie in der *cgMain*-Funktion eine *myVaryingOutput*-Struktur, die Sie ausfüllen, indem Sie die Höhenverschiebung aus der Anzahl der Iterationen vor dem Abbruchkriterium berechnen und den Farbwert setzen:

```
myVaryingOutput out;

// verschobene Vertexkoordinate

float4 newPos = in.myPosition;
newPos.y = clamp
   ( counter * 0.1, -1.0, 1.0 );

// und Transformieren

out.position = mul
( modelViewProjection, newPos );
// Graustufen Farbwert

out.color0 = counter.xxxx * 0.1;
return out;
```

Ein Fraktalgebirge erhalten Sie, indem Sie ein genügend fein aufgelöstes Polygongitter in Form eines Quadrates (in Ihrem OpenGL Programm) zeichnen:

```
#define STEP 0.02f
for ( float j = -2.0f;
    j < 2.0f; j += STEP )
{
  glBegin( GL_TRIANGLE_STRIP );
  for ( float i = -2.0f;
      i < 2.0f; i += STEP )
  {
    glVertex3f( i, 0, j );
    glVertex3f( i, 0, j+STEP );
  }
  glEnd();
}
```

Dieses Beispiel ist ein eher untypischer Verwendungszweck für Vertex-Shader, aber es zeigt auch, wie vielfältig Sie mit wenig C-Code interessante Effekte schnell und einfach testen. Eine Übersicht über einen Teil der Befehle, die in der Cg-Standard-Library vorhanden sind, zeigt die Tabelle links. ET

CG-STANDARD-LIBRARY-FUNKTIONEN

Mathematische Funktion	Bedeutung
abs(x)	Betrag von x
sin(x), cos(x)	Trigonometrische Funktionen
acos(x), asin(x), atan(x), atan(y,x)	Arcus Funktionen
sinh(x), cosh(x)	Hyperbolikus-Funktionen
ceil(x), floor(x)	wie in C
clamp(x, a, b)	Bereichsbeschränkung von x auf [a;b]
cross(a,b)	Kreuzprodukt zweier float4
dot(a,b)	Skalarprodukt zweier float4
mul(v,M)	Zeilenvektor mal Matrix
mul(M,v)	Matrix mal Spaltenvektor
exp(x), exp2(x), log(x), log2(x), log10(x)	Exponential- und Logarithmus-Funktionen
min(a,b), max(a,b)	Minimum-/Maximum-Funktion
pow(x,y)	x^y
sign(x)	Signum-Funktion
frac(x)	Nachkomma-Anteil von x
round(x)	x gerundet
lerp(a,b,f)	(1-f)*a+b*f für Float oder Vektorvariablen
sqrt(x)	Quadratwurzel von x
Geometrische Funktion	**Bedeutung**
distance(pt1,pt2)	Euklidischer Abstand zweier Punkte
faceforward(N,I,Ng)	Resultat ist N, wenn dot(Ng,I)<0 sonst -N
length(v)	Länge des Vektors v
normalize(v)	Normalisierter Vektor zu v
reflect(i,n)	Reflexionsvektor zu i an n, für float3
refract(i,n,eta)	Refraktionsvektor zu i an n mit Brechzahl eta