Vulkan / OpenGL / OpenCL Graphics And Game Physics Framework

Space Combat And Strategy Game Development

OpenAL Audio Framework

Spielephysik (game physics): Kollisionen von Massenpunkten im 3D-Raum

Es gab Zeiten, da konnten Spieleentwickler von den Möglichkeiten heutiger Physik Engines nur träumen. An komplexe Echtzeitsimulationen war wegen der noch viel zu geringen Rechenleistung von CPU und GPU nicht zu denken. Um die Spiele realistischer zu gestalten, beschränkte man sich auf wenige einfache physikalische Effekte (Erdanziehung, Reibungseinflüsse, Stöße, usw.). Deren Berechnung in Echtzeit war jedoch nur im Rahmen einfacher Massenpunkts-Mechanik möglich; die Ausdehnung der Spieleobjekte wird hierbei vernachlässigt.
Kollidieren zwei Objekte mit den Massen **m(a)** und **m(b)** miteinander, lassen sich die Geschwindigkeiten nach der Kollision (**v(a, nachher)**, **v(b, nachher)**) als Folge der Geschwindigkeiten von vor der Kollision (**v(a, vorher)**, **v(b, vorher)**) berechnen. Bleibt die kinetische Energie (Bewegungsenergie) während des Stoßprozesses erhalten, so spricht man von einem elastischen Stoß. Hier die Gleichungen für die Berechnungen der Endgeschwindigkeiten im eindimensionalen Fall:

v(a, nachher) = (2*m(b)*v(b, vorher) + v(a, vorher)*(m(a) - m(b)))/(m(a) + m(b))
v(b, nachher) = (2*m(a)*v(a, vorher) - v(b, vorher)*(m(a) - m(b)))/(m(a) + m(b))

In der Realität verlaufen die Stöße meist nicht elastisch (anelastisch, inelastisch). Zumindest ein Teil der kinetischen Energie geht bei der Kollision verloren und führt zu Verformungen bei den miteinander kollidierenden Objekten. Hier die Gleichungen für die Berechnung der Endgeschwindigkeiten:

v(a, nachher) = ((e+1)*m(b)*v(b, vorher) + v(a, vorher)*(m(a) - e*m(b)))/(m(a) + m(b))
v(b, nachher) = ((e+1)*m(a)*v(a, vorher) - v(b, vorher)*(e*m(a) - m(b)))/(m(a) + m(b))

Setzt man nun e = 1, so lässt sich ein elastischer Stoß simulieren. Verkleinert man den Wert für e, dann wird der Stoß zunehmend anelastisch.

Die Behandlung von dreidimensionalen Stoßprozessen ist ungleich komplizierter. In der nachfolgenden C/C++-Funktionen lassen sich die einzelnen Berechnungsschritte jedoch Schritt für Schritt nachvollziehen:

```
inline bool Compute_3DCollision_Response(D3DXVECTOR3* pVelocity1,
                                         D3DXVECTOR3* pVelocity2,
                                         D3DXVECTOR3* pPosition1,
                                         D3DXVECTOR3* pPosition2,
                                         float &mass1, float &mass2,
                                         float collisionDistanceSq)
{
    // Berechnung der Kollisionsachse (normierte Verbindungslinie zwischen
    // den kollidierenden Massepunkten):
    D3DXVECTOR3 CollisionAxis = *pPosition2 - *pPosition1;

    float DistanceSq = D3DXVec3LengthSq(&CollisionAxis);

    if(DistanceSq > collisionDistanceSq)
        return false;

    float InvDistance = 1.0f/sqrtf(DistanceSq);

    CollisionAxis *= InvDistance;

    // Berechnung der Anfangsgeschwindigkeitsbeträge entlang der
    // Kollisionsachse:
    float tempFactor1 = D3DXVec3Dot(pVelocity1, &CollisionAxis);
    float tempFactor2 = D3DXVec3Dot(pVelocity2, &CollisionAxis);

    D3DXVECTOR3 CollisionAxisVelocity1;
    D3DXVECTOR3 CollisionAxisVelocity2;

    CollisionAxisVelocity1 = tempFactor1*CollisionAxis;
    CollisionAxisVelocity2 = tempFactor2*CollisionAxis;

    // Subtrahiert man nun diese Geschwindigkeitsbeträge
    // von den Anfangsgeschwindigkeiten, so erhält man die
    // Geschwindigkeitsanteile, die während der Kollision
    // unverändert bleiben:
    *pVelocity1 -= CollisionAxisVelocity1;
    *pVelocity2 -= CollisionAxisVelocity2;

    // Berechnung der Geschwindigkeitsanteile entlang
    // der Kollisionsachse nach dem Stoß:
    float tempFactor3 = mass1 + mass2;
    float tempFactor4 = mass1 - mass2;
```

```
    float tempFactor5 = (2.0f*mass2*tempFactor2 + tempFactor1*tempFactor4)/
                        tempFactor3;

    float tempFactor6 = (2.0f*mass1*tempFactor1 - tempFactor2*tempFactor4)/
                        tempFactor3;

    CollisionAxisVelocity1 = tempFactor5*CollisionAxis;
    CollisionAxisVelocity2 = tempFactor6*CollisionAxis;

    // Addiert man die Geschwindigkeitsanteile entlang
    // der Kollisionsachse nach dem Stoß zu den
    // Geschwindigkeitsanteilen, die während des Stoßes
    // unverändert bleiben, erhält man die
    // Endgeschwindigkeiten:
    *pVelocity1 += CollisionAxisVelocity1;
    *pVelocity2 += CollisionAxisVelocity2;

    return true;
}
```