AUF DER HEFT-CD

Spiele-Programmierung unter Windows 95/98/NT

Gravitation im Spiel

Diesmal schreiben Sie ein komplettes Weltraum-Ballerspiel. Sie erkennen Kollisionen, programmieren einen Computergegner und spielen MIDI-Musik im Hintergrund.

CARSTEN DACHSBACHER/
NILS PIPENBRINCK

In den letzten beiden Ausgaben haben Sie die Grundlagen für ein anspruchsvolles Spiel unter Windows kennengelernt. Nun ist es an der Zeit, dieses Wissen zu einem funktionsfähigen und unterhaltsamen Shoot'em-Up-Spiel zusammenzufügen.

Doch keine Angst, wenn Sie die letzten beiden Ausgaben der Rubrik PC Underground nicht verfolgt haben: Alle benötigten Routinen finden Sie auf der Heft-CD, Sie können sie ohne weitere Vorkenntnisse einsetzen.

Die Spielidee von *Gravity Wars*, so heißt unser Weltraum-Ballerspiel, sieht folgendermaßen aus: Zwei Spieler steuern je einen Raumgleiter und versuchen sich gegenseitig abzuschießen. Dabei verfügen sie über eine begrenzte Energiemenge. Ein optional in der Mitte des Spielfelds plazierter Planet zieht dabei alle Objekte wie Raumgleiter und Raketen an.

ENTWICKLUNGSSTUFEN DES SPIELEPROJEKTS

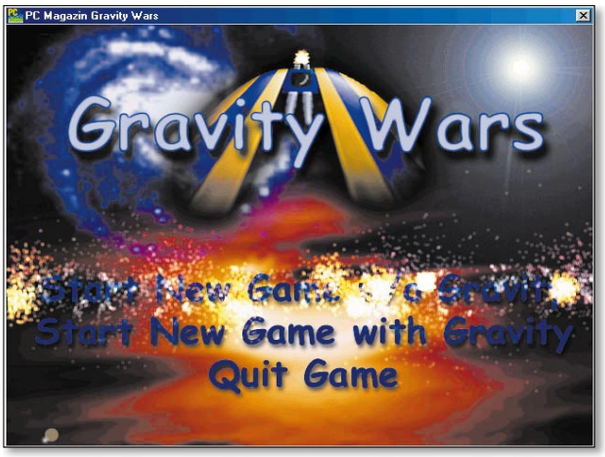
PC Magazin 5/99:
- Entwicklung des Basissystems
- DirectSound-Programmierung
- Soundeffekt-Programmierung/ Klangsynthese

PC Magazin 6/99:
- Sprite-Programmierung
- Partikel- und Effektsystem

PC Magazin 7/99 (diese Ausgabe):
- Algorithmen zur Kollisionsabfrage von Sprites
- Spielelogik und Computergegner
- MIDI-Hintergrundmusik
- Spielegrafik und Highscore-Routinen

Grafik und Logik

An Grafikdaten brauchen Sie das Titelbild und die Hintergrundbilder sowie die Vorlagen der einzelnen Sprites. Aus diesen Vorlagen erzeugen Sie mit dem Sprite-Generator *SpriteGenerierung.exe* aus dem *SPRITE*-Verzeichnis die Sprite-Daten für das Spiel. Diese Sprite-Daten erwartet das Spiel im Unterverzeichnis *data*.



DIE STARKE EMISSION von Partikeln verrät hier, daß Sie sich in der ersten Zeile des Hauptmenüs befinden.

Der Sprite-Generator verwendet die Routine *CreateRotationAnimation(...)* aus der Sprite-Bibliothek und speichert alle Daten der Sprite-Struktur sowie die mit der RLE-Methode (Runtime Length Encoding) komprimierten Bilddaten. Von jedem Sprite werden dabei 64 Einzelbilder – eines für jeden möglichen Drehwinkel – erzeugt. Sie können natürlich auch eigene Sprites zeichnen und einbinden. Eine Auflistung der verwendeten Dateien finden Sie in der Textbox „Individuelle Grafiken und Hintergrundmusik" auf S. 219.

Bevor Sie mit dem Programmieren beginnen, legen Sie das äußere Design des Spiels fest. Auf dem Spielfeld, also dem Hintergrundbild, sollen sich zwei Raumgleiter bewegen. Als Steuerungsoptionen stehen eine Links- bzw. eine Rechtsdrehung sowie das Beschleunigen der Raumschiffe zur Verfügung.

Ziel des Spiels ist es, den Gegner abzuschießen. Dazu können die Raumgleiter eine Rakete abschießen oder sich mit Lasersalven bekämpfen. Jeder Raumgleiter verfügt über eine Energiemenge, die er auf die Antriebs- und Schildsysteme verteilen kann. Wird ein Raumgleiter getroffen, verliert er dadurch Schildenergie. Dabei flackert der Schild kurz auf, was Sie durch ein zusätzliches Sprite realisieren.

Nach einem Treffer soll der Spieler seinen Raumgleiter für eine kurze Zeit nicht mehr steuern können. Bei schweren Treffern bekommt das Raumschiff außerdem einen Schwung um die eigenen Achse ab. Hat ein Treffer die Schildenergie vollständig aufgebraucht, wird das Raumschiff zerstört. Es bleibt noch kurz sichtbar, bis die Explosion das Sprite möglichst komplett verdeckt.

Wie Sie sehen, benötigen Sie eine ganze Reihe von Variablen, die die zeitliche Abfolge der Ereignisse steuern: Sie brauchen zum Beispiel Informationen darüber, wie lange ein Sprite noch sichtbar ist oder wann ein Raumschiff wieder schießen kann – ganz zu schweigen von den Statusvariablen für den Energiehaushalt, die Richtung der einzelnen Schüsse, Position, Richtung und Geschwindigkeit der Raumschiffe usw.

All diese Daten fassen Sie am besten in einem C++-Objekt zusammen, um sie vernünftig zu gliedern und den Code eleganter zu gestalten. Schauen Sie sich hierzu den Quelltext der *PLAYER*-Klasse genauer an. Nach der Definition dieser Klasse gilt es nun, Schritt für Schritt die gewünschten Funktionen zu implementieren.

TASTATURBEFEHLE ZUR STEUERUNG

Aktion	Spieler 1	Spieler 2
Linksdrehung	*[Cursor links]*	*D*
Rechtsdrehung	*[Cursor rechts]*	*G*
Beschleunigen	*[Cursor auf]*	*R*
Laserschuß	*[Leertaste]*	*W*
Raketenabschuß	*[Enter]*	*S*
Energieverteilung	*[Bild auf]/[Bild ab]*	*Q/A*

Steuerbefehle

Die Methode *Action()* enthält alle Aktionen zur Steuerung der Raumschiffe, die der Spieler durch Tastendrücke auslösen kann. Die Aktionen sind durchnumeriert und tragen symbolische Namen wie *KUP, KLEFT* und *KRIGHT*, die für *Key up, Key left* und *Key right* stehen. Auch ein vom PC simulierter Gegenspieler sollte für die Lenkung der Raumgleiter auf die *Action*-Methode zurückgreifen. Dies spart doppelte Arbeit und vermeidet Fehler.

Die Methode *MoveAndDrawPlayer()* fragt die einzelnen Tasten ab. In dem Array *keys[]* stehen dazu die Tastencodes für die Steuerung. Mit Hilfe definierter Indizes für dieses Array wie *KUP, KLEFT* und *KRIGHT* befragen Sie das vom Basissystem zur Verfügung gestellte Array *KeyStatus[]*, ob die entsprechende Taste gedrückt ist. Ist dies der Fall, wird die Arbeit einfach an die Routine *Action* weitergegeben.

Wie Sie die Raumschiffe mit der Tastatur steuern, zeigt Ihnen die Tabelle „Tastaturbefehle zur Steuerung“ oben. Die Zuordnung der Tasten ändern Sie nach Belieben im Initialisierungsteil der Datei *gameplay.cpp*.

Die Drehung eines Raumschiffs um die eigenes Achse initiieren Sie beim Drücken der linken bzw. rechten Cursortaste, indem Sie die Richtungsvariable *r* erhöhen oder erniedrigen:

```
case KLEFT:
  r++;
  break;
case KRIGHT:
  r--;
  break;
```

Mit der Blickrichtung aus *r* können Sie sowohl die Nummer des Sprites berechnen, das Sie zeichnen müssen, als auch – falls nötig – den Beschleunigungsvektor ausrechnen. Die Division durch 32 kommt dadurch zustande, daß es für ein Raumschiff 64 Flugrichtungen gibt:

```
bx=+cos(r/32.0f*PI);
by=-sin(r/32.0f*PI);
```

Nun fehlt noch die Bewegung des Raumschiffs:

```
// Richtungsvektor
dx *= VERZOEGERUNG;
dy *= VERZOEGERUNG;

// Bewegung
x += dx;
y += dy;
```

Natürlich plazieren Sie beim Beschleunigen eines Raumschiffs auch Partikel auf dem Bildschirm und spielen einen Soundeffekt ab – aus Gründen der Übersichtlichkeit fehlt all dies in den abgedruckten Beispielen. Die fertige *Action*-Methode in den Quellcodes auf der Heft-CD führt Ihnen auch diese zusätzlichen Spielereien vor.

Waffensysteme

Da Sie die Raumgleiter nun vollständig manövrieren können, bringen Sie ihnen als nächstes das Schießen bei. Wie bereits erwähnt, unterscheiden Sie dabei Laserschüsse und Raketen. Pro Raumgleiter soll immer nur maximal eine Rakete über den Bildschirm fliegen, um das Spiel übersichtlich zu halten. Diese Rakete fliegt dem gegnerischen Raumschiff hinterher, bis Sie es entweder getroffen haben oder der Treibstoff der Rakete verbraucht ist.

Für die Rakete bietet sich eine eigene Klasse an, wir haben sie *Missile* genannt. Der interessante Abschnitt der Raketenklasse ist der Steuerungsteil *HandleMissile()*, der die Rakete zum gegnerischen Raumschiff steuert.

Prinzipiell erfährt die Rakete immer eine Beschleunigung in Richtung des Gegners:

```
// Richtungsvektor zum
// Zielraumgleiter
zx=player[ziel]->x-x;
zy=player[ziel]->y-y;

// Vektor normalisieren
laenge=sqrt(zx*zx+zy*zy);

if (laenge>0)
{
  zx/=laenge;
  zy/=laenge;
}

dx*=VERZOEGERUNG;
dy*=VERZOEGERUNG;
dx+=zx;
dy+=zy;
x+=dx;
y+=dy;
```

Die Richtungsnummer und damit die Nummer des Sprites berechnen Sie mit Hilfe des Arcustangens:

```
r=64-31*atan2(dy,dx)/PI;
```

Die Laserschüsse, von denen eine ganze Menge auf dem Bildschirm herumschwirren können, verwalten Sie innerhalb des *PLAYER*-Objekts. Ihre Bewegung und das Verwalten der freien Einträge in der *lasershot*-Liste programmieren Sie genauso, wie Sie es von den Partikelroutinen der letzten Ausgabe her kennen. Der einzige Unterschied ist, daß Sie ein Sprite abhängig von der Nummer der Flugrichtung zeichnen, die sich dann aber im späteren Verlauf nicht mehr ändert.

Spielablauf

Um die einzelnen Objekte wie Raumschiffe und Raketen leicht zu handhaben, schreiben Sie dafür eigene Verwaltungsroutinen in der Datei *gameplay.cpp*. Zuerst initialisieren Sie in *void InitGame(...)* das Soundsystem, laden die Sounds und Sprites und legen Instanzen der Objekte an.

DIE BESTENLISTE ERSCHEINT nach zehn ereignislosen Sekunden automatisch und besitzt eine eingebaute Demofunktion.

Das Herzstück dieser Datei ist die Funktion *HandleAndDrawGame()*, in der Sie den Spielablauf festlegen. Bevor nun eine genaue Beschreibung dieser Routine folgt, sollten Sie sich noch eines kleinen Problems bewußt werden: Aufgrund verschiedener Einflüsse läuft ein Programm nicht immer gleichmäßig schnell ab. Deswegen ist die Zeitspanne, die von der Berechnung eines Bilds zur Berechnung des nächsten vergeht, nicht immer konstant.

Diese Unregelmäßigkeit sollten Sie natürlich bei der Berechnung der Beschleunigungen, Bewegungen und dem Ablauf von zeitlichen Werten wie der Lebensdauer einer Rakete berücksichtigen. Darum berechnen Sie für die Zeitkorrektur einen Faktor aus der aktuellen

KOMMT EIN RAUMSCHIFF dem Planeten zu nahe, zerschellt es in einem großen Feuerball an der Oberfläche.

Zeit, dem Zeitpunkt des letzten Durchlaufs der Routinen und einer bestimmten Bildrate:

```
neue_zeit=GetDemoTime();
faktor=
  (neue_zeit-alte_zeit)/20.0f;
alte_zeit=neue_zeit;
```

Dann rufen Sie für jedes Raumschiff die Methode *MoveAndDrawPlayer()* auf, die die Tastatursteuerung, die Bewegung und das Zeichnen der Sprites enthält. Anschließend berechnet diese Routine die neuen Positionen der Laserschüsse und zeichnet sie auf das Spielfeld.

Für jeden Laserschuß überprüft die Routine außerdem, ob eine Kollision mit einem der Raumschiffe vorliegt. Ist dies der Fall, teilt sie es dem *Raumschiff(PLAYER)*-Objekt über den Aufruf *HitMe(int damage)* mit. Diese Routine verringert die Schildenergie des Raumgleiters und veranlaßt bei Bedarf eine Explosion des Raumschiffs. Bei einem Zusammenstoß der beiden Raumschiffe prallen diese voneinander ab, indem sie einfach die jeweiligen Richtungsvektoren *dx* und *dy* der Schiffe ändern.

Die Kollisionsabfragen in diesem Spiel verlassen sich auf eine reine Abstandsberechnung – das reicht für die verwendeten Objekte völlig aus und ist leicht zu implementieren. Alle Kollisionen halten Sie in einer Liste fest, um dann am Ende der *HandleAndDrawGame()*-Routine die entsprechenden Klänge abzuspielen. Hier ein Auszug:

```
int NumCollision=0;
int Collision[MAXCOLLISION];

// Lasertreffer
Collision[++NumCollision]=
  CLASER;
// Crash der Raumschiffe
Collision[++NumCollision]=
  CPLAYER1|CPLAYER2;

...

// Abspielen der Sounds
while (NumCollision>=0)
{
  int c=Collision[NumCollision-];
  if (c==(CPLAYER1|CPLAYER2))
  {
    SoundSys->PlaySound(sCrash);
  }

  if (c & CLASER)
  {
    SoundSys->PlaySound(sHit);
  }
}
```

Für die grafischen Explosions- und Raucheffekte setzen Sie das Partikelsystem ein, das Sie in der letzten Ausgabe entwickelt haben – Sie finden es auch im Quelltext auf dieser Heft-CD. Immer wenn ein Laserstrahl auf den Schutzschild eines Raumschiffs trifft, eine Rakete detoniert – sei es durch einen Treffer oder durch den Ablauf ihrer Lebensdauer – oder ein Raumschiff anderweitig Schaden nimmt, setzen Sie an der entsprechenden Stelle Explosionspartikel frei. Beim Bewegen einer Rakete oder dem Zünden eines Raumgleitertriebwerks stoßen Sie Rauchpartikel aus.

Bevor Sie die Methode *HandleAndDrawGame()* beenden, stellen Sie noch die verschiedenen Energievorräte der Raumgleiter als Balkengrafiken dar. Danach übernimmt wieder das Hauptprogramm die Kontrolle und kann seinerseits das Partikelsystem und andere Routinen aufrufen.

Gravitationskräfte

Da unser Spiel *Gravity Wars* heißt, sollte auch die Anziehungskraft eine wichtige Rolle darin spielen. Deshalb plazieren Sie einen großen Planeten in der Mitte des Spielfelds, der sowohl Raumschiffe als auch Raketen anzieht. Sie sollten mit seiner Oberfläche nicht in Berührung kommen, da das Raumschiff sonst daran zerschellt.

Die Berücksichtigung dieser Anziehungskraft im Spiel ist denkbar einfach: Die Gravitation ist nichts anderes als eine zusätzliche Beschleunigung des Raumgleiters bzw. der Rakete in Richtung der Bildschirmmitte, wo sich der Planet befindet:

```
// Richtungsvektor zur
// Bildschirmmitte
gx=(SCREEN_X/2)-px;
gy=(SCREEN_Y/2)-py;

distanz=sqrt(gx*gx+gy*gy);

if (distanz>0)
{
  gx/=distanz;
  gy/=distanz;
}
dx+=gx*ANZIEHUNGSKRAFT;
dy+=gy*ANZIEHUNGSKRAFT;
```

Das ist schon alles, was Sie für diese Spielvariante, die Sie im Hauptmenü des Spiels auswählen können, hinzufügen müssen. Die Gravitation wirkt natürlich nur auf die Raumschiffe und die Rakete. Wären auch die Laserstrahlen betroffen, hätten Sie es mit einem sogenannten „Schwarzen Loch“ zu tun, und dem sollten Sie bekanntermaßen möglichst fernbleiben.

Computergegner

Jetzt haben Sie ein nettes Ballerspiel für zwei Personen geschrieben. Aber was, wenn gerade niemand gegen Sie antreten möchte? Als Ersatz erschaffen Sie deshalb einen Computergegner.

Nun gibt es 1001 Möglichkeiten, einen Computergegner zu programmieren. Zum leichten Einstieg sollten Sie die Anforderungen jedoch nicht allzu hoch ansetzen. Gute Computergegner verlangen sehr viel theoretisches Wissen und Programmierkenntnis.

Wenn Sie das Spiel im Zweispieler-Modus ausprobieren, werden Sie einen gewaltigen Unterschied zwischen dem Spiel mit und ohne Gravitation feststellen. Die Anforderungen an den Computergegner sind für beide Spielarten sehr unterschiedlich: Beim Spiel ohne Gravitationsfeld reicht es aus, wenn sich der Gegner ständig bewegt und auf Sie – den Spieler – schießt.

Für das Spiel mit Gravitation ist es zwar auch entscheidend, den Gegner zu treffen – viel wichtiger ist es aber, sich von dem Planeten fernzuhalten. Die nötigen Daten des künstlichen Spielers definieren Sie als Struktur in der *PLAYER*-Klasse:

```
typedef struct
{
  // Ziel des CPU-Players
  float x;
  float y;
  float r;  // Ruhe-Radius
  int Angriff;  // Angriff?
  int isCpuPlayer;
  // Pointer auf den Gegner
  PLAYER * enemy;
} CpuPlayerData;
```

Zunächst zum Spiel ohne Gravitation. Eine sehr einfache, aber effektive Strategie für den Computer ist es, seine Aktionen in zwei Phasen aufzuteilen:

- In der Bewegungsphase manövriert er sein Raumschiff und ist daher eher defensiv.
- Benutzt er hingegen seine Laserkanone, befindet er sich in der Angriffsphase. In diesem Fall ist das Status-Flag *Angriff* gesetzt. Die Variable *IsCpuPlayer* zeigt an, ob die betreffende Instanz eines Raumschiffs überhaupt vom Computer gelenkt werden soll.

Bewegungsphase

In der Bewegungsphase soll der Computergegner eine Position im Raum anfliegen. Da die Steuerung auch für den Computer schwer zu kontrollieren ist, definieren Sie einen Kreis als „sichere Position" für den Computer. Dafür geben Sie mit *x* und *y* den Mittelpunkt eines Kreises an, *r* definiert dessen Radius. Solange der Computergegner nicht im Radius war, versucht er, durch Lenkbewegungen in diesen Kreis zu gelangen.

Die Methode

```
float CpuPlayerTargetTo(
  float xx,float yy,int steer)
```

enthält die Hauptlogik zum Zielen und Lenken. In den Parametern *xx* und *yy* teilen Sie dieser Routine eine beliebige Position im Raum mit. Ist der Parameter *steer* gesetzt (*steer = 1*), probiert der Computergegner durch Auslösen der Düsen links und rechts, sich auf diese Position auszurichten.

Möchten Sie dagegen nur den Winkel zur Zielposition berechnen, rufen Sie die Routine mit ungesetztem *steer*-Flag auf (*steer = 0*). Diese Zusatzfunktion nutzen Sie später, um zu entscheiden, ob ein Schuß mit dem Laser sinnvoll ist oder nicht.

Die Implementierung der Methode selbst ist wenig spektakulär. Sie berechnet mit der trigonometrischen Funktion *atan2* den Winkel und ruft je nach Ausrichtung zum Ziel die *Action*-Methode auf. Hierbei zahlt sich ein etwas erhöhter Aufwand aus, um die schnellste Lenkbewegung zu ermitteln.

Sorgen Sie auch für einen fließenden Übergang zwischen 0° und 360°. Den sogenannten Wrap Around an dieser Stelle fangen Sie ab und behandeln ihn

DIE RAKETEN RICHTEN ihre Flugbahn immer wieder neu auf das gegnerische Raumschiff aus.

gesondert. Dann können Sie endlich Gas geben:

```
// Winkel messen und
// Ausrichten:
float d=CpuPlayerTargetTo(
  cpu.x,cpu.y,1);

// Falls nicht zu schnell und
// Winkel ungefähr stimmt,
// etwas Schubkraft geben
if ((d<10) && (speed<3.0))
  Action(KUP);
```

Einfach, aber effektiv. Diese beiden Zeilen bringen den Gegner sicher an sein Ziel. Sobald der Computerspieler sein Ziel erreicht hat, geht er in die nächste Phase, den Angriff, über.

Angriffsphase

In dieser Phase gilt es, sich auf den Gegner zu konzentrieren und ihn mit Laserschüssen zu beschäftigen. Dazu dient die bereits entwickelte Routine *CpuPlayerTargetTo*. Damit der Computerspieler ein nicht allzu leichtes Ziel abgibt, achten Sie zusätzlich darauf, daß er nicht immer an einer Stelle steht.

Wird sein Raumschiff zu langsam, gibt der Computer einfach etwas Schubkraft. Der Code hierzu ist dem der ersten Phase sehr ähnlich:

```
CpuPlayerTargetTo(
  cpu.enemy->x,cpu.enemy->y,1);
if (speed<0.4) Action (KUP);
```

Sobald der Gegner seinen Bereich verläßt, sucht er sich per Zufall eine neue Position, und das Spiel beginnt von vorne. Dies kann zum Beispiel nach einer Kollision mit dem Gegner oder einer Rakete nötig sein.

Allgemeine Aktionen

Unabhängig von den Bewegungs- und Angriffsphasen des Computers behandeln Sie den Abschuß einer Rakete und das Verteilen der Energie auf Lenkung und Schildsysteme. Diese beiden Aktionen werden immer ausgeführt. Bei der Verteilung des Energiehaushalts legt der virtuelle Gegner Priorität auf seine Schildenergie. Die Lenkenergie soll immer nur gerade dazu ausreichen, das Schiff zu bewegen:

INDIVIDUELLE GRAFIKEN UND HINTERGRUNDMUSIK

Wollen Sie eigene Grafiken in das Spiel einbauen, sollten Sie diese in Größe und Proportion an den vorhandenen orientieren. Die Dateien für die Sprites liegen im Verzeichnis des Sprite-Generators:

laser.bmp	Laserschuß
rocket.bmp	Rakete
shipx00.bmp	Raumschiff x (x = 1 oder 2)
shipx01.bmp	Raumschiff x mit Laserkanonen
shipx10.bmp	Raumschiff x mit aktivem Triebwerk
shipx11.bmp	Raumschiff x mit aktivem Triebwerk und Laserkanonen
shipxs1.bmp	Raumschiff x mit sichtbarem Schutzschild

Die Dateien, auf die das Spiel letztendlich zugreift, befinden sich im Unterverzeichnis *data*:

back1.bmp	Hintergrundbild mit Planet
back2.bmp	Hintergrundbild ohne Planet
titelbild.bmp	Bild des Hauptmenüs
**.dat*	aus den Sprites generierte Daten

Auch die MIDI-Datei für die Hintergrundmusik können Sie ganz einfach austauschen. Es handelt sich hierbei um eine ganz normale Standard-MIDI-Datei. Das Internet ist eine gute Quelle für solche Musikstücke. Die MIDI-Datei *gravity.mid* hat der Demomusiker DOJ exklusiv für dieses Spiel komponiert. DOJ heißt eigentlich Dirk (derartige Fantasienamen, auch Handles genannt, sind in der Demoszene üblich) und ist Mitglied der Demogruppe Cubic&Seen, der auch die beiden Autoren dieses Beitrags angehören.

```
if (thrustenergy>20)
  Action(KPOW1);
else Action(KPOW2);
```

Die Logik für den Raketenabschuß ist auch nicht viel komplizierter. Hierzu messen Sie erneut den Winkel zum Gegner. Ist dieser klein genug, schießen Sie eine Rakete ab. Da das Spielfeld quasi unendlich ist, kann eine Rakete den Gegner auch über die Bande des Spielfelds treffen.

Bedenken Sie, daß die Richtungsangaben der *PLAYER*-Klasse von 0 bis 64 definiert sind. Im Vergleich

```
(fabs(d-32)<2.0)
```

subtrahieren Sie vom Ausrichtungswinkel *d* den Wert 32 (entspricht 180°) und bilden den Absolutwert. Ist dieser klein genug, besteht die Chance auf einen Treffer. Die Methode *Action* löst dann den Schuß aus. Dies geschieht allerdings ganz willkürlich. Sobald die Geschwindigkeit den willkürlich festgelegten Wert von 4.0 überschreitet, wird geschossen. Dies passiert in der Regel dann, wenn das Gefecht im vollen Gange ist und das Spielfeld so chaotisch aussieht, daß die Rakete unentdeckt ihren Weg zum Ziel findet.

```
// Winkel zum Feind messen
float d=CpuPlayerTargetTo(
  cpu.enemy->x,cpu.enemy->y,0);

// Lohnt Laser-Schuß?
// eventuell über Bande?
if ((d<6.0) ||
  (fabs(d-32)<2.0))
  Action(KFIRE1);

// Rakete abschießen
if (speed>4.0)
  Action(KFIRE2);
```

Für das Spiel mit Gravitation ist, wie bereits erwähnt, eine etwas andere Strategie sinnvoll. Der Gegner versucht hierbei immer, einen Sicherheitsabstand zum Planeten zu erreichen. Dazu ermitteln Sie ständig den Abstand und die Ausrichtung zum Planeten und berechnen in jedem Aufruf eine neue Position. Die neue Zielposition ist dabei immer etwas weiter vom Planeten entfernt. Damit der Gegner seine Position dennoch wechselt, addieren Sie einen kleinen Zufallswert.

Ist der Abstand zum Planeten groß genug, wechseln Sie in den Angriffsmodus. Den Quellcode der gesamten Computersteuerung finden Sie in der Methode *CpuKeyControl()* der Klasse *PLAYER*.

Auch wenn die Strategie des Computerspielers sehr einfach wirkt, ist sie dennoch äußerst effektiv. Ein Spiel gegen den computergenerierten Astronauten macht wirklich Spaß.

Sie können auch eigene Ideen in den Code einfügen, um einen anspruchsvolleren Gegner zu programmieren. Senden Sie uns ruhig Ihre Verbesserungen an:

```
praxis@pc-magazin.de
```

Wir sind gespannt auf Ihre Einfälle.

Bestenliste

Was wäre ein Ballerspiel ohne eine Bestenliste (englisch Highscore)? Auch bei *Gravity Wars* darf diese nicht fehlen. Einen Spieler bewerten Sie einfach daran, wie schnell er seinen Gegner bezwingt. Das hängt natürlich auch von der Stärke des Gegenübers ab, erspart aber eine aufwendigere Punkterechnung.

Bei jedem Spiel messen Sie die Zeit. Wer das Spiel überlebt, darf sich – falls er schnell genug war – in die Highscore-Liste eintragen. Die Implementierung einer solchen Bestenliste ist relativ leicht. Sie finden den gesamten Code in der Datei *main.cpp*. Interessant ist es jedoch, die Anzeige der Highscore-Tabelle mit einem Demomodus zu kombinieren.

Wenn Sie im Hauptmenü des Spiels für etwa zehn Sekunden keine Taste drücken, beginnen zwei Computergegner ein Duell gegeneinander. Auf diesem bewegten Hintergrund stellen Sie die Highscore-Tabelle dar. Dazu benutzen Sie am besten die in den vorangegangenen Artikeln entwickelte Font-Klasse. Sie eignet sich hervorragend, um schnell und einfach Ausgaben auf den Bildschirm zu bringen.

Im übrigen steuern Sie das gesamte Programm per Tastatur. Im Hauptmenü gehen Sie mit den Tasten *[Cursor auf]* und *[Cursor ab]* zum gewünschten Menüpunkt. Die aktuelle Position erkennen Sie dabei nicht etwa an einem Rollbalken, sondern an starker Flammenaktivität. Die Auswahl erfolgt dann mit der *[Enter]*-Taste.

Musik liegt in der Luft

Musik ist ein häufig unterschätzter Bestandteil guter Computerspiele. Viele Spiele sind gerade wegen ihrer guten Musik noch heute bekannt. Computerveteranen erinnern sich vielleicht noch an den Spieleklassiker *Turrican* vom C64 und Amiga.

Die Bibliotheken, mit denen Sie bisher in PC Underground Moduldateien (**.mod* und **.xm*) abgespielt haben, vertragen sich leider nicht mit dem Spiele-Soundsystem. Daher beschreiten Sie einen anderen Weg, und zwar mit den leider etwas aus der Mode gekommenen MIDI-Dateien.

Die Multimedia-API von Windows bietet einen erstaunlichen Komfort beim Abspielen solcher Musikstücke. Zunächst öffnen Sie eine MIDI-Datei über die Multimedia-API:

```
MCI_OPEN_PARMS mciOpenParms;
DWORD dwReturn;

mciOpenParms.lpstrDeviceType=
  "sequencer";
mciOpenParms.lpstrElementName=
  "gravity.mid"

if (dwReturn=mciSendCommand(
  NULL,MCI_OPEN,
  MCI_OPEN_TYPE|
  MCI_OPEN_ELEMENT,
  (DWORD)(LPVOID)
  &mciOpenParms)) return(0);
```

Sequencer ist der Name für den Windows-eigenen MIDI-Mapper. Beim Aufruf des obigen Codes liest Windows die MIDI-Datei *gravity.mid* ein und analysiert, welches der installierten MIDI-Geräte zum Abspielen am besten geeignet ist. In der Regel ist das der Synthesizer-Chip auf Ihrer Soundkarte.

Die MIDI-Datei ist nun geladen und muß nur noch gestartet werden. Sie verzichten darauf, sich Nachrichten über den aktuellen Zustand des Playbacks schicken zu lassen. Diese belasten das System zwar nur gering, sind jedoch für den Spielfluß uninteressant.

```
MCI_PLAY_PARMS mciPlayParms;
DWORD dwReturn;

mciPlayParms.dwCallback=NULL;
if (dwReturn=mciSendCommand(
  mciOpenParms.wDeviceID,
  MCI_PLAY,MCI_NOTIFY,
  (DWORD)(LPVOID)
  &mciPlayParms))
{
  mciSendCommand(
    mciOpenParms.wDeviceID,
    MCI_CLOSE,0,NULL);
  return(0);
}
```

Wenn alles geklappt hat, sollte jetzt Musik aus Ihrer Soundkarte tönen. Die Musikwiedergabe endet automatisch, sobald Sie das Programm verlassen.

Soviel zunächst zur Spieleprogrammierung. In der nächsten Ausgabe berichten wir unter anderem über ein elektronisches Magazin aus der Demo-Szene und stellen Ihnen verschiedene Effekte in Logos vor. PEI/BM

Die Quelltexte zu diesem Beitrag und das fertige Spiel *Gravity Wars* finden Sie zusammen mit der zugrundeliegenden Grafikbibliothek auf unserer Heft-CD im Verzeichnis *praxis\pc-under* und im Internet-Angebot des PC Magazin unter

```
www.pc-magazin.de/magazin/
➥extras.htm
```

Klicken Sie unter *Online Extras* im Menü *Praxis* auf das entsprechende *Download*-Feld.