Vorwort

Unser Kurs „Breakout, Pinball & Friends“ vereinigt die Themen Informatik, Mathematik und Physik mit dem Ziel, ein „Breakout“-Spiel zu programmieren und dabei die physikalischen Gesetze möglichst genau zu simulieren. Bei dieser Art von Computerspiel muss der Spieler mit einem bewegbaren Paddle den Ball auf die Spielsteine schießen und verhindern, dass ein Ball unten aus dem Bildschirm fliegt. Keiner von uns hätte gedacht, wie schwer es ist, so ein, offenbar simples, Spiel zu programmieren. Wer hätte denn erwartet, dass man mit Vektoren rechnen muss, um eine Kugel an einer Wand abprallen lassen zu können!? Es gab einige Überraschungen.

Nach zwei Wochen voller Blut, Schweiß und Tränen lag schließlich die wundervolle Endfassung einschließlich eines (fast) unmöglichen Endlevels vor.

Nicht zuletzt dank der sympathischen und hilfsbereiten Leiter wird uns diese Zeit als Gegensatz zum oft tristen Schulalltag ewig in Erinnerung bleiben.

In der folgenden Dokumentation wollen wir Ihnen unsere Arbeit näher bringen.

Eröffnungswochenende

Am Eröffnungswochenende wurden wir zunächst mit den Grundlagen des Programmierens vertraut gemacht.

Dabei wurden auch Ziele des Kurses und die Vorgehensweise genauer beschrieben. Nicht weniger wichtig war, dass wir unsere Kursleiter und die anderen Teilnehmer kennen lernten, mit denen wir ja immerhin zwei Wochen zusammenarbeiten würden.

Am ersten Abend legten wir auch gleich los und schrieben unsere ersten Programme. Sie konnten immerhin schon einen Text ausgeben und einfache Berechnungen durchführen. Im weiteren Verlauf erzeugten wir noch unser erstes Programm mit Grafikausgabe.

Nach dem Eröffnungswochenende wurde der Vorkurs per E-Mail fortgesetzt. Zu unserer Erleichterung waren die Aufgaben im Vergleich zu anderen Kursen weder besonders schwer noch besonders viel. Falls es doch Probleme gab, konnten wir uns im Forum der „Science Academy“ an die anderen Kursteilnehmer und die Leiter wenden. Allerdings bereitete das Programmieren uns so viel Spaß, dass wir uns auch mal freiwillig hinter die Tastatur klemmten.

Die Übungsaufgaben des Vorkurses:

- Ein Programm zur Berechnung des eigenen Alters in Tagen
- Ausgabe verschiedener geometrischer Figuren und eines Farbkreises der additiven Farbmischung
- Primzahlenberechnung nach verschiedenen Verfahren
- Ein Programm, in dem sich Kugeln auf einem Billardtisch bewegen und an der Bande abprallen

1. Unser Kurs

Teilnehmer:

Julia Brugger
Ich sag nur: NICHT die Mysteriöse im knappen grünen Kleid von JC … oder doch?
Ging beim Konzertabend mit ihrer Querflöte flöten und zeigte sich im Kurs eher ruhig. Während andere lieber im warmen Bett blieben, lief sie beim morgendlichen Jogging auch mal die doppelte Strecke - mit lockerer Gelassenheit. Und dabei war sie im Kurs mit eifrig-kreativem Elan beim Levelkreieren am Werk - das Ergebnis sieht man ja!

Jana Gessert
Ich sag nur: USB-Stick.
Ohne obigen wäre sie wohl nicht die selbe gewesen. Außerdem ist sie Inhaberin des wohl schönsten kleinen Laptops der Welt ;). War auch während des Kurses dann und wann im Chat der Science Academy Homepage aufzufinden, wo man dann lange, heiße und hoch anspruchsvolle Diskussionen über gerade aktuelle Kursthemen oder Problemen bei der Kursarbeit (in ihrem Fall die Erstellung von Leveln) mit ihr abhalten konnte, um sie dann fröhlich mit oben genanntem USB-Stick blinken zu sehen.

Andreas Geyer-Schulz
Ich sag nur: Unser Özi (man kann sich über Rechtschreibung streiten oder ihn auch einfach „Ösi“ nennen) .
Der AGS programmierte die Vektorkollision. Sein Österreichisch war für uns immer wieder ein Grund zur Freude. Liebte es, in seiner Freizeit darüber fachzusimpeln, wie man Alltagsprobleme programmiertechnisch lösen könnte, wobei er tief in Mathematik und Physik einstieg.

Johanna Grözinger
Ich sag nur: Miss Science Academy 2005!!!
Unsere barfüßige Ballerina mit C++-Wälzer unter dem Arm und Stiften im Haar überzeugte die Jury der Miss Science Academy-Wahl durch ihre großartige Stimme: „We will rock you“! Es konnte einem auch passieren, dass die Mitleiterin der Elbisch-KüA mitten im Gespräch plötzlich sprichwörtlich einen Abgang machte und „mal kurz zwischendurch“ einen sauberen Überschlag hinlegte.

Anton Haffner
Ich sag nur: Mister Pinball im Image des Aldimanns.
Unser Lockenkopf, unübertroffen an Charme und Talent zum Cellospielen. Zeigte rege Freude am Programmieren von eher unkonventionellen Bonusobjekten. Und sind wir mal ehrlich, niemandem stehen Aldi-Tüten so gut wie ihm! Er war nicht nur Mitglied, sondern einer der Gründungsgründe (herrliche Worte gibt’s) der (imaginären?) Band.

Amelie Harbisch
Ich sag nur: Andie.
Die mit den langen Haaren, die sich beim Leveldesignen dezent kreativ zeigte. Begeisterte Teilnehmerin der Theater-KüA (von „Hab ich was gesagt?“ bis zu einer gewaltigen Menge an Tochter- und Vater-programmen). Schwört auf Nahrungs-ergänzungsmittel Nr. 7565286765/E56: Lachen.

Johannes Küchle
Ich sag nur: Das letzte Level...
Unsere Sportskanone - was hätten wir ohne ihn bei den Highland-Games gemacht? Begeisterter, ich möchte fast sagen fanatischer Levelschreiber, der das (beinahe) unschaff-bare Level Nr. 21 unseres Spieles kreierte. War sonst oft in der Stomp-KüA anzutreffen und ein Gründungsmitglied der prominenten Science Academy-Band!

Robin Lingstädt
Ich sag nur: Tears in Heaven.
Er schaffte es aufgrund seines bezaubernden Talents zum Singen mit begleitendem Gitarre-spielen (ja, richtig geraten - der Sänger und Gitarrist der Band) nicht nur eine kleine Gruppe, sondern einen ganzen Fanclub mit (vorwiegend) weiblichen Mitgliedern um sich zu scharen, wodurch er immer wieder dazu überredet wurde, spontan seine Gitarre auszupacken und drauflos zu singen. Dennoch ist ihm der ganze Ruhm nie zu Kopfe gestiegen und er blieb der nette, ruhige Kerl.

Stephan Schmid
Ich sag nur: Das stille Genie.
Der etwas ruhigere Kursteilnehmer war nicht nur unser staatlich geprüfter Spielzusammenbastler, sondern programmierte schwere Brocken ganz „mir nichts, dir nichts“ und ganz alleine. Blickte auch noch nach einem ganzen Tag Oberstufenmathe voll durch.

Martin Schwald
Ich sag nur: „…sonst fängt er noch an zu SINGEN!!!“
Während andere noch am Mönchs- oder Bürgermeisterrätsel verzweifelten und sich entsetzt die Haare rauften, war der Meister der Rätsel mit der Situation völlig unterfordert und widmete sich lieber mit viel Talent der Theater-KüA. War immer, überall und sofort fürs Skat- und Mafia-Spielen zu haben.

Christian Schweizer
Ich sag nur: Meister der Kugel-Kugel-Kollision.
Programmierte liebend gerne Cheats und war in seiner Freizeit oft bei der Tischtennisplatte zu finden. Hat eine Vorliebe für Latein und Altgriechisch, wodurch man sich besonders gut mit ihm unterhalten konnte.

Marco Weber
Ich sag nur: Der Winni.
Er kümmerte sich um die Vektorklasse. Beschäftigte sich sehr gerne und äußerst liebevoll mit einem gewissen kleinen grünen Panda und seinen eigens personifizierten Spielobjekten (siehe Winfried, Hagen, Chlodwig,...). Spricht und schreibt FLIESSEND ;) Elbisch, daher Elbisch-KüA-Leiter Nr.2. Erlag dann und wann dem Heimweh nach seiner geliebten Gitarre.

“Unser” Chinese

Chaoyi
Trotz (vor allen Dingen LEICHT sprachbedingter) Verständigungsschwierigkeiten kam er verblüffend schnell mit und programmierte am Schluss sogar ein eigenes, gut funktionierendes Breakoutspiel mit sprich-wörtlich chinesischem Hintergrund.

Leiter

Kleines Vorwort: Wir möchten allen unseren Kursleitern (Schülermentor inbegriffen) für ihre Geduld danken, dafür, dass sie auch die 225.786 dumme Frage beantwortet haben, für ihr Engagement und für ihre immerwährende Freundlichkeit.

Jörg Richter
Ich sag nur: Mr J. Judge.
Der Physiker (NICHT Mathematiker!!!), der einem sein Fach sehr nahe bringen konnte. Sehr talentiert auch beim Auf-die-Folter-Spannen-wenn-es-um-besonders-fiese-Rätsel-geht.

Daniel Jungblut
Ich sag nur: Meister der 244 Compilerfehler.
Lustiger Typ, der „Erklärbär" des Kurses. Wer Daniel konsultierte, hatte es geschnallt. „Abroad" hielt er sich meistens in der Tanz-KüA auf - und zeigte dort die Profitänzer-Seite des Mr Youngblood.

Schülermentor

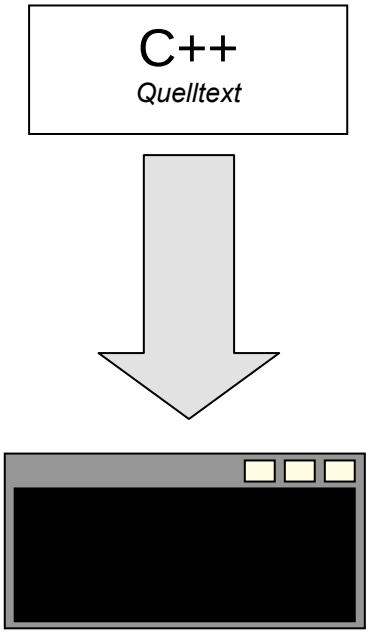
Alexander Schlüter
Ich sag nur: Unser Prof. Dr. Dr. Dipl. Ing. für Idiotenfehler.
Besaß das unglaubliche Talent, in den nötigsten Momenten wieder Stimmung unter die Leute zu bringen. Außerdem unterzog er gestresste Kursteilnehmer dann und wann einer Massagekur, wofür er viel Dankbarkeit erntete. Zudem großer Daddel- und Philosophier-Liebhaber. Entführte uns in die Tiefen der bezaubernden Welt der Dynamik und Kinematik.

Amelie (Andie) Harbisch

2. Einführung in die Programmierung

Was ist eine Programmiersprache und welche Funktion hat sie?

Eine Programmiersprache ist eine formale Sprache mit eindeutigen Befehlen zum Programmieren. Sie ist dem Menschen im Gegensatz zum Maschinencode leichter verständlich, für den Computer jedoch unverständlich. Der / Die ProgrammiererIn gibt Befehle ein, den sogenannten Quellcode. Der Quellcode muss für den Computer in eine für ihn verständliche Sprache umgewandelt werden. Diese Arbeit wird vom Compiler erledigt, der eine ausführbare (*.exe) Datei erstellt.



Ausführbare Datei

Warum haben wir die Programmiersprache C++ verwendet?

Es gibt mehrere Gründe, warum wir für unser Spiel die Sprache C++ (sprich: Zeh-Plus-Plus) verwenden.

1. Effizienz

Professionelle Anwendungen und selbst modernste 3D-Spiele werden in C++ geschrieben. Dank eines großen Befehlsumfangs ist die Sprache für den Programmierer sehr komfortabel zu verwenden. Zudem erzeugt die Sprache Programme, die schnell in der Ausführung sind.

2. Verbreitung

Aufgrund der weiten Verbreitung dieser Sprache existieren viele Bücher und Hilfestellungen im Internet.

3. Unterstützung

Aus dem zweiten Punkt resultiert auch, dass es viele technischen Unterstützungen gibt. So gibt es etwa die hardwarebeschleunigende Grafikbibliothek OpenGL.

Unsere ersten einfachen Konsolenbeispiele:

Wir begannen mit einfachen Rechnungen und Textausgaben. Hier ein Beispielprogramm:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])

{
  cout << "Hallo!" << endl;


  int y = 3;

  int x = 5;

  int Ergebnis = x + y;

  cout  << y << "+" << x
   << "=" << Ergebnis << endl;

  system("PAUSE");
  return 0;
}
```

1. Die Mainfunktion wird immer zuerst ausgeführt.

2. „cout" ist ein Befehl zur Textausgabe.
„Hallo" soll ausgegeben werden.

3. Variablen werden deklariert und ihnen werden Werte zugewiesen.

4. x und y werden addiert und das Ergebnis in einer dritten Variable mit dem Namen „Ergebnis" gespeichert.

5. Mit dem Befehl „cout" wird die Rechnung und das Ergebnis ausgegeben.

Auf dem Bildschirm erscheint:

Ändert man die Werte, so kann man neu kompilieren und ein neues Ergebnis wird berechnet.

Julia Brugger und *Marco Weber*

3. OpenGL und Grafik

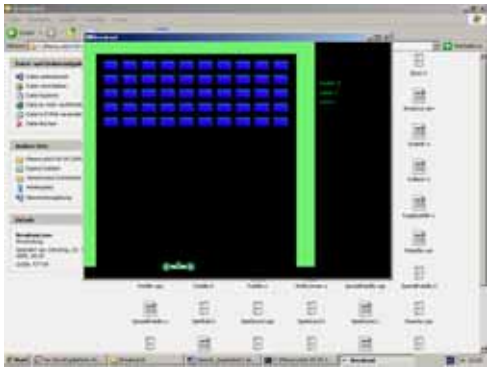
Einleitung

Im "normalen" C++ gibt es keine Befehle zur Erzeugung von grafischen Objekten. Es gibt nur die Standardausgabe, mit der in einer Konsole (z.B. DOS-Fenster) Textausgaben gemacht werden können. Deshalb verwendeten wir die OpenGL Grafikbibliothek, die solche Befehle bereitstellt.

OpenGL

Die Grafikkartenhersteller optimieren ihre Produkte auf diese Bibliothek, sodass eine sehr schnelle Ausführung der Programme möglich ist. Bei den Spieleprogrammierern erfreut sich OpenGL großer Beliebtheit.

Wie wird mit OpenGL Grafik erzeugt?
Der erste Schritt mit Grafik zu arbeiten, ist die Erzeugung eines Grafikfensters. Dieses grafische Fenster besitzt dann einen eigenen Titel sowie die Schaltflächen "Minimieren", "Maximieren" und "Schließen" in der rechten, oberen Ecke.



Der nächste Schritt besteht darin, in das schwarze Fenster etwas hineinzuzeichnen. Dazu gibt es sehr viele Befehle wie z.B.:

- `glColor3f(r,g,b)`legt die Zeichenfarbe fest.
- `glVertex2f(x,y)` legt die Position eines Eckpunktes eines Vielecks fest.
- `glBegin()` und `glEnd()` markieren den Anfang bzw. das Ende eines Codeabschnitts, in dem ein Objekt gezeichnet wird.

Obwohl OpenGL eine sehr hohe Befehlsvielfalt bietet, gibt es auch für vieles keine Befehle. In solchen Fällen mussten wir Wege finden, mit denen sich das Problem umgehen ließ. Das Paradebeispiel dafür war bei uns das Zeichnen eines Kreises bzw. einer Kugel: Da es keine Befehle für runde Objekte gibt, kann man einen Kreis mit OpenGL nicht so zeichnen, wie er in der Natur ist. Die Lösung besteht in einer Näherung. Wir zeichneten den Kreis als regelmäßiges 32-Eck. Das menschliche Auge kann dieses Objekt nicht mehr von einem Kreis unterscheiden und wir hatten die Illusion eines Kreises erzeugt.

Um Punkte im Fenster gezielt bestimmen zu können, verwendet man ein unsichtbares Koordinatensystem. Dieses Koordinatensystem hat seinen Ursprung in der linken

unteren Ecke unseres Fensters. Die positive x-Richtung zeigt nach rechts, die positive y-Richtung nach oben. Wenn man z.B. einen Eckpunkt eines Vielecks festlegt, gibt man einfach die Koordinaten des Punkts an und das Programm weiß, wo der Punkt liegt.

Das Farbsystem von OpenGL

OpenGL verwendet das sogenannte RGB-Farbsystem. Es heißt so, weil es Rot, Grün und Blau als Grundfarben bezeichnet. Aus diesen Farben lassen sich alle anderen Farben mischen. Warum aber kann man Farben als Summe von drei Grundbestandteilen darstellen?

Die Antwort liegt im Bau des menschlichen Auges: Auf der Netzhaut gibt es nur 3 verschiedene Farbrezeptortypen, die jeweils auf rot, grün oder blau reagieren

Das Gehirn berechnet aus dem Verhältnis, wie die drei Typen ansprechen, Farben. Wenn man jetzt jeden Rezeptortyp gezielt über seine spezifische Farbe anspricht, kann man dem Gehirn jede beliebige Farbe simulieren. Man spricht hierbei von additiver Farbmischung.

Stephan Schmid

4.1 Bewegung

Es stellt sich die Frage: Wie funktioniert eigentlich Bewegung im Computer?

Was wir im Allgemeinen unter Bewegung verstehen ist, dass sich irgendetwas von einem Ort an einen andern bewegt, dass sich die Position verändert und zwar fließend, ohne Sprünge oder Unterbrechungen. Im Computer ist dies unmöglich, da er nur statische Bilder zeichnen kann.

Also bedient man sich eines Tricks. Ähnlich wie im Kino werden minimal veränderte, statische Einzelbilder in Sekundenbruchteilen hintereinander gezeigt, so schnell, dass unser Sehapparat die Einzelbilder nicht mehr voneinander unterscheiden kann und der Eindruck entsteht, dass dort eine Bewegung wäre. Es sollten mindestens um die 25 Einzelbilder pro Sekunde sein, da das Bild bzw. die Bewegung sonst ruckelig erscheint, und die Augen zu sehr angestrengt werden. (Kleine Anmerkung: Bei den Computern, die wir benutzten waren es um die 40 Einzelbilder pro Sekunde bei unserem Spiel, bei einem guten PC sind es ca. 1000).

Doch wie lassen sich diese Überlegungen in Quellcode übersetzen?

Am Besten lässt sich das an einem Beispiel erklären. Wir wollen einen Gegenstand, in diesem Fall eine Kugel, über den Bildschirm bewegen. Jetzt dürfen wir die Position der Kugel in der Deklaration der Kugel nicht statisch festsetzen, sondern müssen eine Variable einführen, sodass sich die Kugel an unterschiedlichen Positionen befinden kann:

```
grafikKugel(x, 100, 10);
```

In diesem Fall haben wir eine Kugel, deren x-Position (x) variabel ist, die also in der Horizontalen beweglich ist. Die anderen Werte aber, nämlich die Y-Position (in diesem Fall 100) und der Radius (10) sind festgesetzt.
Nun muss die Variable `x` aber oben im Programm auf einen Anfangswert festgesetzt werden, sodass die Kugel am Anfang überhaupt irgendwo erscheint:

```
float x = 0;
```

Also befindet sich die Kugel beim Start des Programms an der X-Position 0.
Zur Bewegung der Kugel brauchen wir außerdem eine weitere Variable, die die Bewegung der Kugel festlegt:

```
float Dx = 0.05;
```

D.h., wir haben jetzt einen festen Wert (Dx), auf den wir später zur Bewegung der Kugel zugreifen können.
In der Zeichenfunktion müssen wir gegenüber den alten Programmen keine Veränderungen vornehmen, es sieht aus wie früher, bis auf die Variable:

```
void display() {
  glColor3f(1.0, 0.0, 0.0);
  ...
  grafikKugel(x, 100, 10);
  ...
}
```

Jetzt kommt der eigentliche Trick für die Bewegung: Zusätzlich zur Zeichenfunktion und der Hauptfunktion wird eine Rechenfunktion eingefügt (`void idle`). In dieser Rechenfunktion werden alle für die Bewegung nötigen Rechenschritte durchgeführt, in unserem Fall die Verschiebung der Kugel parallel zur x-Achse, also die Änderung der x-Position der Kugel:

```
void idle() {

   x = x + Dx;

   glutPostRedisplay();
}
```

So sieht die idle-Funktion aus. Oben ist der Funktionskopf, ähnlich wie bei anderen Funktionen auch. Dann folgen einfache, bei komplizierteren Programmen auch komplexere, Rechnungen, in diesem Fall die Vergrößerung des X-Wertes der Kugel um Dx, also um 0,05.

Der Ablauf sieht nun folgender Maßen aus: Beim Start des Programmes wird zuerst die Zeichenfunktion aufgerufen, die das Bild zeichnet. Wenn dieser Vorgang abgeschlossen ist und der Computer keine anderen Operationen auszuführen hat wird die Rechenfunktion aufgerufen. Dort werden Werte für ein verändertes Bild berechnet. Nach Beendigung der Rechenoperationen werden die neuen Werte mit Hilfe des Befehls „`glutPostRedisplay();`“ an die Zeichenfunktion zurückgegeben. Diese zeichnet ein neues statisches Einzelbild mit den neuen Werten, in diesem Fall die Kugel ein winziges Stück nach rechts versetzt. Wenn alles gezeichnet ist und der Computer wiederum keine weiteren Operationen auszuführen hat wird wieder die Rechenfunktion aufgerufen. Zu der neuen X-Position der Kugel wird wieder etwas hinzu addiert, die Kugel wandert weiter nach rechts. So wechseln sich die Rechen- und die Zeichenfunktion ab und die Einzelbilder zeigen eine Kugel die sich immer weiter rechts befindet. Das Auge kann, wie oben schon erklärt die Einzelbilder nicht mehr unterscheiden und es entsteht der Eindruck einer Bewegung.

Das Programm im Wesentlichen:

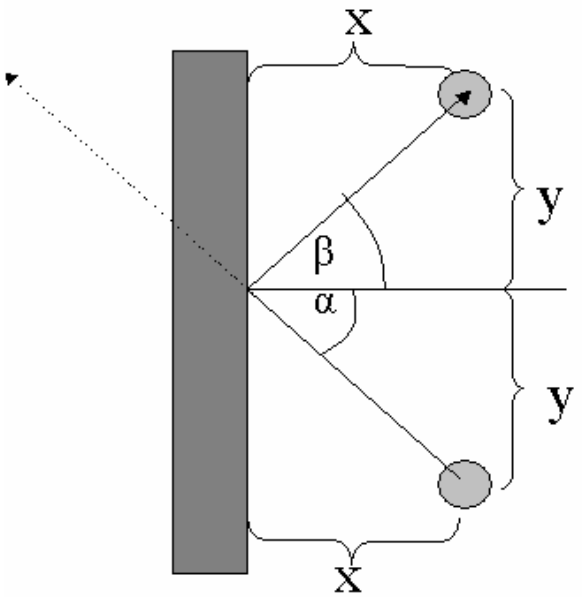
```
// Startposition
float x = 0;
// Geschwindigkeit
float Dx = 0.05;

// Zeichenfunktion
void display() {
  // Farbe
  glColor3f(1.0, 0.0, 0.0);
  ...
  // Initialisierung der Kugel
  grafikKugel(x, 100, 10);
  ...
}

// Rechenfunktion
void idle() {

 //neue Position
 x = x + Dx;
// Aufforderung, das Bild neu zu
zeichnen
  glutPostRedisplay();
}
```

Johannes Küchle

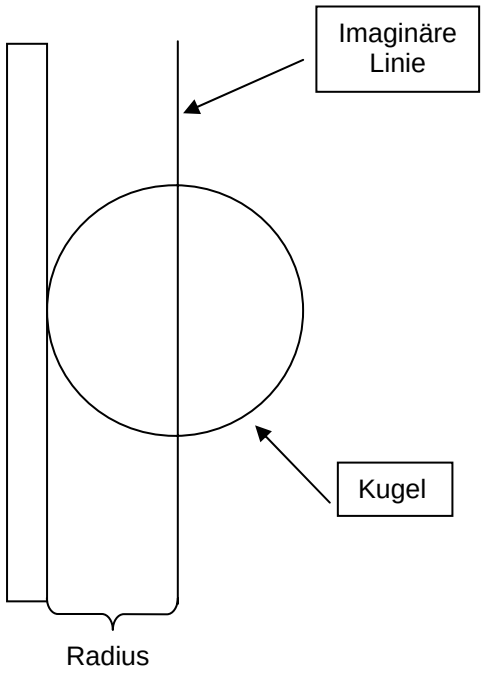


4.2 Kollision

Da in unserem Spiel die Kugeln an Banden und Hindernissen abprallen sollen, mussten wir eine Kollision programmieren. Natürlich sollte so eine Kollision möglichst den Naturgesetzen entsprechen.
Jede Bewegung auf dem Bildschirm lässt sich in zwei einzelne Bewegungen zerlegen, eine in horizontaler und eine in vertikaler Richtung. Wir bezeichnen die zwei Geschwindigkeiten in diese Richtungen mit v_x und v_y. Ist v_x > 0, bewegt sich die Kugel nach rechts, bei v_x < 0 nach links. Ist v_y > 0, bewegt sie sich nach oben, bei v_y < 0 nach unten.
Betrachten wir zunächst die Kollision an einer senkrechten Bande:

Es gilt das bekannte Prinzip Einfallswinkel gleich Ausfallswinkel. Wie man in dieser Abbildung sieht, ändert sich nur die Bewegung in horizontaler Richtung – sie wird umgedreht. Die Bewegung in vertikaler Richtung bleibt jedoch konstant. (Denn in zwei gleich großen Zeitabschnitten legt die Kugel den selben Weg „nach oben“ zurück: y)
Bei der Kollision mit einem horizontalen Bande ist es genau umgekehrt; die Bewegung in vertikaler Richtung wird umgedreht und die in horizontaler Richtung bleibt erhalten.
Da wir nur auf die Koordinaten des Mittelpunktes der Kugel zugreifen können, die Kugel

aber schon reflektiert werden soll, wenn ihr Rand die Bande berührt, denken wir uns zur einfachen Programmierung vor den Banden Linien:

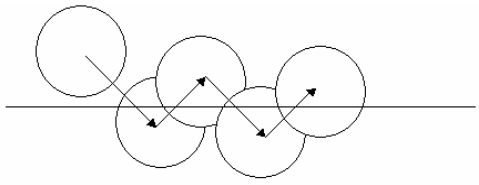


Wenn nun der Kugelmittelpunkt eine solche imaginäre Linie überschreitet, soll die Kugel reflektiert werden. Dies lässt sich so programmieren:
(x und y sind die Koordinaten des Kugelmittelpunktes)

```
// Kollision am linken Rand
if (x <= SPIELFELD::LINKS +
Radius) {
// Wenn die Kugel links
rausfliegt:
      v_x = -v_x;
// Drehe die x-Geschwindigkeit um!
}
// Kollision am rechten Rand
if (x >= SPIELFELD::RECHTS +
Radius) {
// … rechts rausfliegt:
      v_x = -v_x;
// Drehe die x-Geschwindigkeit um!
}
// Kollision am oberen Rand
if (y >= SPIELFELD::OBEN + Radius)
{
// … oben rausfliegt
      v_y = -v_y;
// Drehe die y-Geschwindigkeit um!
}
```

Bei dieser Kollision kam es allerdings manchmal zu Fehlern. Denn manchmal kommt es vor, dass die Kugel im aktuellen Bild noch innerhalb des Spielfeldes ist, jedoch im nächsten Bild schon weit außerhalb, sodass es

zu einem Fehler kommen konnte:



Im ersten Bild ist die Kugel noch innerhalb des Spielfeldes. Im zweiten überschreitet sie die Bande -> sie wird reflektiert. Jedoch ist sie im dritten Bild immer noch nicht wieder im Spielfeld -> Sie wird erneut reflektiert usw…

Behebung des Fehlers:
Die Kugel soll nur reflektiert werden, wenn sie ohne Reflektion im nächsten Bild noch weiter vom Spielfeld entfernt sein würde! D.h. Die Kugel soll nicht reflektiert werden, wenn sie sich (zumindest zum Teil) außerhalb des Spielfeldes befindet, sich aber in Richtung Spielfeld bewegt!

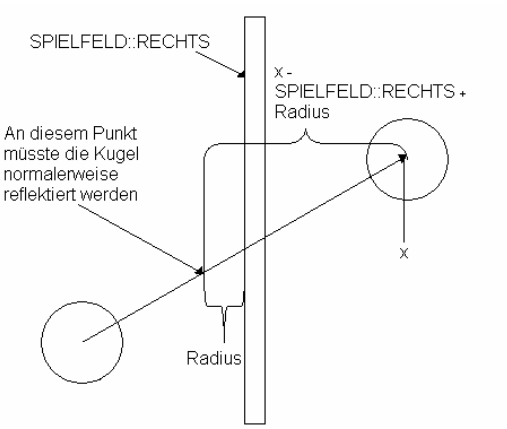
(&& entspricht in C++ dem logischen Und)

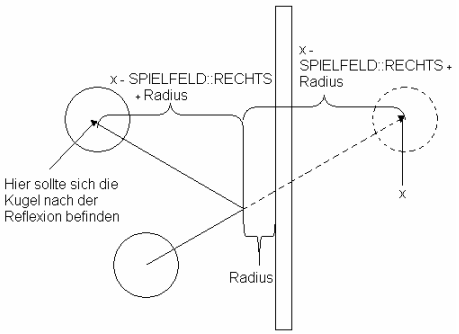
```
// Kollision am linken Rand
if (x <= SPIELFELD::LINKS + Radius
&& v_x < 0) {
// Nun wird nur reflektiert, wenn
      v_x = -v_x;
// sich die Kugel nach links
bewegt
// Zudem wird die Kugel wieder in
das Feld gesetzt (Erklärung siehe
unten):
x = x + 2 * (SPIELFELD::LINKS – x
+ Radius);
}

// Kollision am rechten Rand
if (x >= SPIELFELD::RECHTS +
Radius && v_x > 0) {
// … nach rechts bewegt
      v_x = -v_x;
x = x – 2 * (x – SPIELFELD::RECHTS
+ Radius);
}

// Kollision am oberen Rand
if (y >= SPIELFELD::OBEN + Radius
&& v_y > 0) {
 // … nach oben bewegt
      v_y = -v_y;
y = y – 2 * (y – SPIELFELD::OBEN +
Radius);
}
```

Das Zurücksetzen der Kugel basiert auf folgender Idee:





Der neue Kugelmittelpunkt lässt sich also leicht berechnen: x = x – 2 *
(x – SPIELFELD::RECHTS + Radius)
(Entsprechend an den anderen Banden)
Somit hatten wir eine gut funktionierende Kollision erstellt.

Martin Schwald

5. Objekt-Orientiertes Programmieren

Was ist OOP?

OOP ist ein Programmierkonzept, das näher an der menschlichen Denkweise liegt. Man verwendet so genannte Klassen als Baupläne für Objekte. Ein Objekt wird genauso wie in der Realität durch Eigenschaften („Wie sieht das Objekt aus?“) und Methoden („Was kann man damit machen?“) beschrieben.

Ein Beispiel: Wenn wir einen Fußball sehen, denken wir zuerst, dass es ein Ball ist. D.h. er ist rund, hat einen bestimmten Durchmesser, ist aus einem bestimmten Material, hat eine Farbe... (Eigenschaften). Außerdem kann man ihn bewegen, sei es durch rollen, kicken oder werfen (Methoden).

Nachdem man weiß, dass es ein Ball ist, werden nun die o.g. Eigenschaften und Methoden spezifisch auf den Fußball übertragen. Also z.B.: der bestimmte Durchmesser beträgt vielleicht 17 cm, der Ball ist aus Leder und schwarz-weiß. Und er kann gekickt, geworfen, gerollt werden.

Klassen

Zu Anfang schrieben wir den kompletten Quelltext in die Hauptdatei. Doch mit der

Komplexität des Spieles wuchsen auch die Ausmaße des Quelltextes. So legten wir spezielle Quelldateien für bestimmte Objekte (z.B. Das Paddle, die Kugel,...) an. In diesen Quelltexten wiederum legt man nun eine Klasse an, in der die speziellen Eigenschaften und Methoden verknüpft sind (oben wäre dies die Klasse „Ball“).

```
class Kugel {
// Klassenname
  // Eigenschaften/Attribute
    float Radius;
    float Position_x;
    float Position_y;
    float Farbe;
    (...)
  // Methoden
    void Zeichnen();
    void Bewegen();
    void Position_setzen(
      float x, float y);
    void Radius_setzen(
      float r);
    (...)
};
```

In einer weiteren Quelldatei muss dem Computer nun noch in Programmiersprache erklärt werden, was diese Methoden und Eigenschaften bedeuten. Von jeder Klasse kann man beliebig viele Objekte erzeugen, deren Eigenschaften verändern und die Methoden benutzen. Also könnte man auf dieselbe Weise nun auch einen Tischtennisball deklarieren.

Im Hauptprogramm sieht ein solcher Aufruf z.B. so aus:

```
Kugel kugel;
// Hier wird ein Objekt nach dem
Bauplan der Klasse Kugel angelegt.

(...)

Kugel.Zeichnen();
// Hier wird die Kugel gezeichnet
(...)
```

Die Übersichtlichkeit und das Verständnis des Programms wird hierdurch wesentlich erleichtert, da das Programmieren in einzelnen Schritten vermieden wird.

Vererbung

Klassen können vererbt werden. Die so genannte Tochterklasse besitzt alle Eigenschaften und Methoden der Vaterklasse, ist aber durch weitere Eigenschaften spezialisiert. Beispiel in unserem Spiel ist die Ableitung der Paddle- und Hindernisklasse von der Blockklasse. Dadurch konnten wir unnötigen Programmieraufwand vermeiden und die Übersichtlichkeit verbessern.

So ist z.B. unser Paddle ein Block, der sich zusätzlich noch bewegen kann.

Kapselung

Die Eigenschaften eines Objektes sind teilweise voneinander abhängig. Dadurch kommt es zu Problemen, wenn man eine Eigenschaft verändert und die dadurch notwendigen Veränderungen an der anderen Variable nicht berücksichtigt. Dieses Problem kann man durch Kapselung lösen. Dabei werden die Eigenschaften als „private“ deklariert. Um sie jetzt noch zu verändern, benötigt man neue Methoden. Diese wiederum werden als „public“ deklariert. Da in diesen Methoden die benötigten Veränderungen berücksichtigt werden, werden die abhängigen Eigenschaften nun entsprechend verändert.

„Private“ bedeutet, dass man Eigenschaften nicht mehr über eine einfache Änderung der Variablen neu festlegen kann. Die „public“-Methoden jedoch können überall im Programm aufgerufen werden und die Eigenschaften des Objektes verändern. Die abhängigen Eigenschaften werden durch die Methoden automatisch angepasst. Dies ist nun die einzig mögliche Weise Variablen zu verändern.

Im Quelltext sieht dies folgender Maßen aus:

```
class Kugel {
// Klassenname

    private:
    // Eigenschaften/Attribute
    float Hoehe;
    float Position_x;
    float Position_y;
    (...)

    public:
    // Methoden
     void Hoehe_setzen(
           float h);
     void Position_setzen(
           float x, float y);
    (...)
};
```

In unserem Computerspiel hängen zum Beispiel die Eigenschaften „Höhe“ und „Koordinaten“ voneinander ab. Verändert man die Höhe, verändern sich normalerweise auch die x- und y-Koordianten. Deshalb verändert man die Höhe nun nicht mit „`h = 20`“ sondern mit der Methode „`Hoehe_setzen (20)`“.

Johanna Grözinger

6. Texturen

Obwohl das Spiel inzwischen schon recht bunt ist, kann es doch noch etwas Abwechslung im Design brauchen. Gerade um Oberflächen ansprechender zu gestalten, verwendet man gerne Texturen.

Texturen sind grob gesagt zweidimensionale Grafiken, also *Bilder*, die man auf die Oberfläche von Polygonen lädt.

1. Bild erstellen

Mit einem einfachen Bildbearbeitungsprogramm wie beispielsweise MSPaint kann ein Bild erstellt werden. Dieses muss im Bitmap-Format gespeichert werden.

2. Bild laden

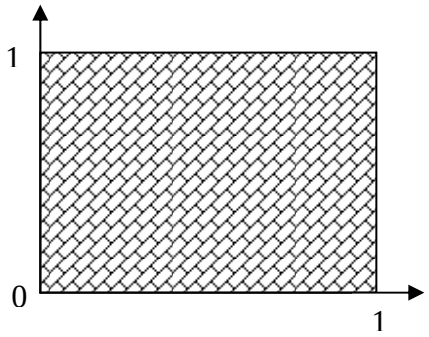
Der Pfad der Grafik wird einer Funktion übergeben. Diese lädt das Bild in den Speicher der Grafikkarte. Der Ort der Grafik auf der Grafikkarte wird in einer Variablen gespeichert.

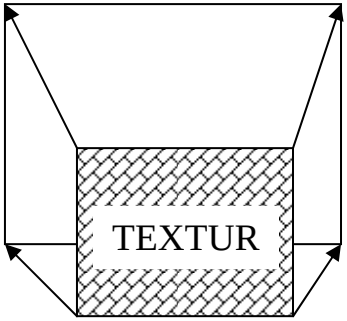
```
int Textur1 =
LadeTextur(„texturen/block.bmp“);
```

3. Vom Bild zur Textur

Nun muss nur noch ein Polygon mit der Textur „überzogen“ werden. Dieser Vorgang wird als „*Texture Mapping“* bezeichnet.

Um die Position innerhalb einer Textur anzugeben, hat sich ein Texturkoordinatensystem eingebürgert. Die linke untere Ecke wird mit (0|0) bezeichnet, die rechte obere mit (1|1) und die anderen beiden mit (1|0) und (0|1).





Nachdem man die Textur eingebunden hat, zeichnet man wie gewohnt ein Rechteck. Allerdings wird, bevor eine Ecke des Rechtecks gezeichnet wird, die Ecke der Textur angegeben, die darauf abgebildet werden soll.

So sieht der Quelltext dazu aus:

```
// Textur laden
int Textur1 =
LadeTextur(„texturen/block.bmp“);

// Textur einbinden
glBindTexture(GL_TEXTURE_2D,
Textur1);
glEnable(GL_TEXTURE_2D);
// Polygon zeichnen
glBegin(GL_POLYGON);
  // Farbe setzen
  glColor3f(1.0, 1.0, 1.0);
  // Erst Texturkoordinate, dann
  Ecke des Polygons
  glTexCoord2f(0.0, 0.0);
  glVertex2f(100, 100);
  glTexCoord2f(0.0, 1.0);
  glVertex2f(100, 150);
  glTexCoord2f(1.0, 1.0);
  glVertex2f(200, 150);
  glTexCoord2f(1.0, 0.0);
  glVertex2f(200, 100);
glEnd();
glDisable(GL_TEXTURE_2D);
```

Für das Spiel wurde noch eine Texturverwaltung geschrieben, die vor dem Laden überprüft, ob sich die Textur bereits im Speicher befindet und somit nicht mehr geladen werden muss.

Marco Weber

7. Physik

Um unser Spiel realitätsnäher zu gestalten, wollten wir zum Beispiel eine Schwerkraft einführen. Um diese allerdings physikalisch korrekt zu gestalten, mussten wir uns erst mit der Kinematik, der Lehre der Geschwindigkeit, vertraut machen.

Dazu führte uns Alex zusammen mit Herrn Richter einen Versuch vor:
An einer Schnur sind in gleichgroßen Abständen kleine Metallplättchen angebracht. Diese Schnur wird nun senkrecht hängend fallen gelassen. Beim Aufkommen auf den Boden geben die Plättchen einen Ton von sich. Es war deutlich zu hören, dass der Abstand zwischen den Tönen immer kleiner wurde. Folglich wurde die Geschwindigkeit immer größer.
Eine andere Kette war präpariert worden, dass die Plättchen in gleichen Zeitabständen auf den Boden kamen. Jetzt waren allerdings die Plättchen in nach oben immer größeren Abständen montiert. So war der Abstand vom ersten Plättchen zum Boden 9 cm, vom 2. schon 36 cm, dann 81, 144 und so weiter. Anhand der Daten konnten wir erkennen, dass es sich um eine Bewegung mit konstanter Beschleunigung handelt. Doch wie definiert man eigentlich diesen Begriff?
Als Maß für die Beschleunigung *a* dient die Geschwindigkeitsänderung pro Zeitintervall. Die Geschwindigkeitsänderung wird als *Δv*, die vergangene Zeit als *Δt* bezeichnet. Man definiert:

$\bar{a} = \frac{\Delta v}{\Delta t}$ Der Strich über dem *a* kennzeichnet eine durchschnittliche Beschleunigung.

Als Einheit der Geschwindigkeit verwenden wir $1\frac{\mathrm{m}}{\mathrm{s}}$, für die Zeit s (Sekunden).

Damit ergibt sich als Einheit für die Beschleunigung: $[a] = 1\frac{\mathrm{m/s}}{\mathrm{s}} = 1\frac{\mathrm{m}}{\mathrm{s}^2}$

Hier ein Beispiel, wie man die Beschleunigung *a* berechnet:
Ein Auto beschleunigt von 80 km/h auf 120 km/h in 10 s . Die Geschwindigkeitsänderung *Δv* beträgt 40 km/h. Da wir aber die Geschwindigkeit in m/s angeben, müssen wir die 40km/h erst in m/s umrechnen.

$$40\frac{\mathrm{km}}{\mathrm{h}} = \frac{40000\mathrm{m}}{3600\mathrm{s}} = 11{,}1\frac{\mathrm{m}}{\mathrm{s}}$$

Dieses ist *Δv* noch durch 10 s teilen.

$$\frac{11{,}1\frac{\mathrm{m}}{\mathrm{s}}}{10\mathrm{s}} = 1{,}1\frac{\mathrm{m}}{\mathrm{s}^2}$$

Das Auto hat also eine Beschleunigung von $\bar{a} = 1{,}1\frac{\mathrm{m}}{\mathrm{s}^2}$.

Anfahren aus dem Stand

Uns interessierte besonders, wie sich Körper beim gleichmäßigen Beschleunigen aus dem Stand verhalten. In diesem Fall sind sowohl *Δt*, *Δs* als auch *Δv* gleich dem Endwert, weil der

Anfangswert jeweils 0 beträgt:

$\Delta t = t_E - t_A = t_E$

Ebenso bei *Δs* und *Δv*.

Somit können wir die Formel $\overline{a} = \frac{\Delta v}{\Delta t}$ in $\overline{a} \cdot \Delta t = \Delta v$ und dann in $v_E = a \cdot t_E$ umwandeln.

Die Durchschnittsgeschwindigkeit beträgt bei einer konstanten Beschleunigung aus dem Stand die Hälfte der Endgeschwindigkeit.

Oder: $\overline{v} = \frac{1}{2} v_E$

Wenn wir dies in unsere Definition für Geschwindigkeit einsetzen, erhalten wir:

$\Delta s = \frac{1}{2} v_E \cdot t_E$

Kombinieren wir dies mit $v_E = a \cdot t_E$, so ergibt sich:

$\Delta s = \frac{1}{2} (a \cdot t_E) \cdot t_E$

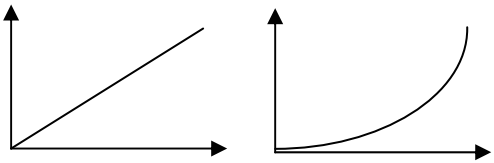
Vereinfacht:

$\Delta s = \frac{1}{2} a \cdot t_E{}^2$

Diagramme

Nun stellten wir konstante Beschleunigung in einem *v – t* - und einem *s – t* - Diagramm dar. Wir stellten fest, dass eine konstante Beschleunigung in einem *v – t* - Diagramm eine Gerade, in einem *s – t* - Diagramm eine Parabel ergibt.

t – v Diagramm *s - t* Diagramm



(*a* jeweils konstant)

Dass dies so ist, liegt an der vorhin erarbeiteten Formel, $\Delta s = \frac{1}{2} a \cdot t_E{}^2$. Hier wird nämlich das *t* quadriert, weshalb die Funktion die Form $f(x) = a \cdot x^2$ hat. Funktionen dieser Art geben immer Parabeln.

Dynamik

Nun stellte sich jedoch die Frage, wieso Körper beschleunigen. Schnell war klar, dass dies aufgrund von Kräften geschieht.
Wir verdeutlichten uns mit Hilfe kleiner Wagen, die von Kraftmessern gezogen wurden, dass eine konstante Kraft eine konstante Beschleunigung bewirkt.
Also gilt (bei gleichbleibender Masse): $F \sim a$

Außerdem ist die erforderliche Kraft proportional zur beschleunigten Masse: $F \sim m$

Daraus folgt: $F \sim m \cdot a$
Oder anders geschrieben: $F = c \cdot m \cdot a$, wobei *c* eine Konstante ist.
Die Einheit der Kraft „Newton“ wurde nun so festgelegt, dass diese Konstante 1 ist. So kamen wir zum zweiten Newtonschen Gesetz:

$$F = m \cdot a$$

Um dies auf unser Spiel zu übertragen, schauten wir uns noch eine besondere Situation an, nämlich den freien Fall.
Die Kraft, die hier wirkt, ist die Schwerkraft. Sie ist abhängig vom Gewicht des Objekts und von einem speziellen Ortsfaktor, der hier auf der Erde $9{,}81 \frac{\mathrm{N}}{\mathrm{kg}}$ beträgt. Also: $F = m \cdot g$

Nun gilt auch: $F = m \cdot a$
Wenn man diese beiden Formeln kombiniert, ergibt sich: $a = g$, der Ortsfaktor ist also identisch mit der Beschleunigung.
Daraus folgt allerdings auch, dass die Beschleunigung für alle Körper gleich ist, nämlich: $\Delta v = g \cdot \Delta t$

Eigentlich müsste demnach eine Feder genauso schnell wie ein Stein fallen. Dass dies nicht so ist, liegt am Luftwiderstand.
Um uns das zu veranschaulichen, führten wir einen Versuch durch: In einem Glasrohr ließen wir eine Feder und einen Metallklumpen fallen. Die Feder kam natürlich deutlich später unten an. Dann pumpten wir die Luft heraus, erzeugten im Fallrohr ein Vakuum, und wiederholten den Versuch. Dieses Mal kamen die beiden Gegenstände wirklich gleichzeitig an.
Jetzt waren alle physikalischen Voraussetzungen für das Weiterprogrammieren an unserem Spiel gegeben.

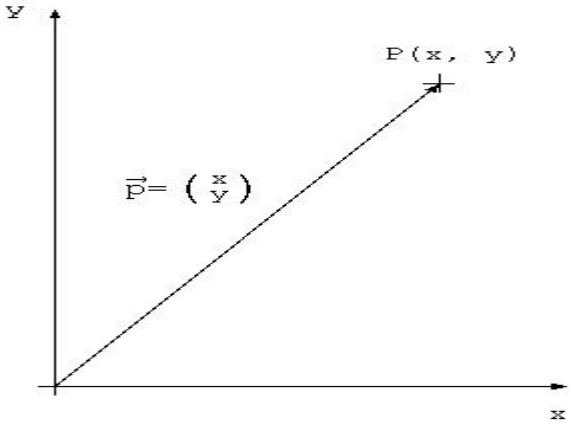
Anton David Haffner

8. Mathematik

Um unsere Kenntnisse in Physik umzusetzen, mussten wir uns mit Mathematik beschäftigen. Insbesondere arbeiteten wir mit Vektoren.

Was ist ein Vektor?
Ein Vektor kann als Ursprungspfeil in einem Koordinatensystem interpretiert werden und wird durch die Koordinaten des Punktes, zu dem er zeigt angegeben:

$$\vec{p} = \begin{pmatrix} x \\ y \end{pmatrix}$$



Betrag

Jeder Vektor hat außerdem einen Betrag. Der Betrag gibt Länge des Vektors an. Mit Hilfe des „Satzes von Pythagoras“ kann man ihn errechnen:

$$|\vec{p}| = \left|\begin{pmatrix} x \\ y \end{pmatrix}\right| = \sqrt{x^2 + y^2}$$

Als nächstes lernten wir, wie man mit Vektoren rechnet.

Addition

Vektoren können addiert werden:

$$\vec{p}_3 = \vec{p}_1 + \vec{p}_2 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \end{pmatrix}$$

Bei der Addition werden die x- und y-Koordinaten addiert. Dieses Verfahren war uns bereits aus der Physik vom sogenannten “Kräfteparallelogramm” bekannt, jedoch stellt dieses das Kommutativgesetz der Addition dar und nicht die eigentliche Addition. Grafisch kann man die Addition veranschaulichen, indem man das Ende des einen Vektors an den Anfang des anderen Vektors setzt. An den gestrichelten Linien ist zu erkennen, dass das Kommutativgesetz auch gilt.

Skalare Multiplikation

Ebenso ist es möglich, einen Vektor mit einer Zahl zu multiplizieren:

$$s * \vec{p} = s * \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s * x \\ s * y \end{pmatrix}$$

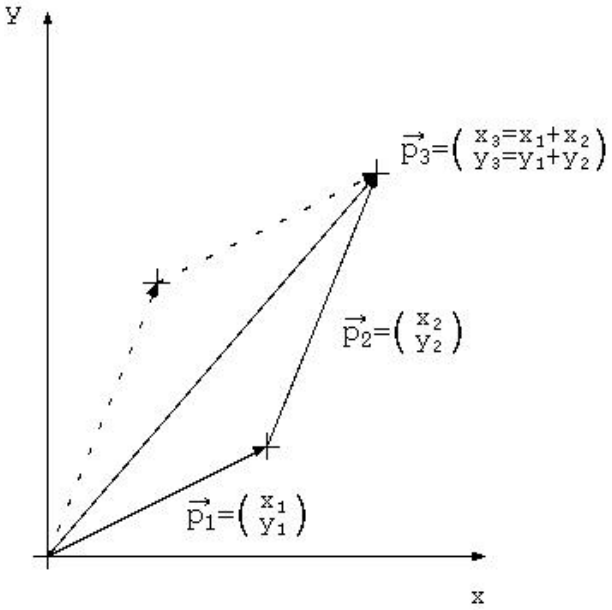
Hier wird jede Koordinate mit der Zahl, dem sogenannten Skalar, multipliziert. Durch die skalare Multiplikation kann ein Vektor verlängert oder verkürzt werden, er behält aber seine Richtung bei. Ist der Skalar negativ so zeigt der Vektor nach der Multiplikation genau in die entgegengesetzte Richtung.
Dividiert man einen beliebigen Vektor durch seinen Betrag, so erhält man einen Einheitsvektor der Länge 1.

Skalarprodukt

Das Skalarprodukt zweier Vektoren ist eine reelle Zahl:

$$\vec{p}_1 \circ \vec{p}_2 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \circ \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = x_1 * x_2 + y_1 * y_2$$

Das Skalarprodukt ist sehr nützlich, denn es gilt folgende Beziehung:



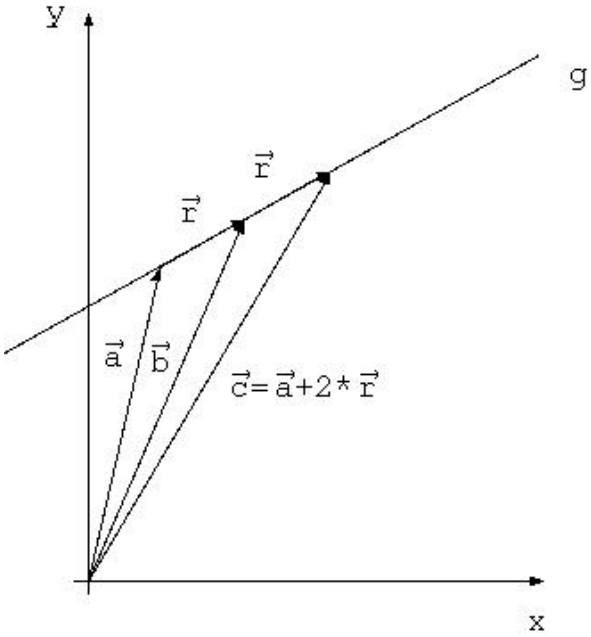
$$\frac{\vec{p}_1 \circ \vec{p}_2}{|\vec{p}_1| * |\vec{p}_2|} = \cos(\alpha)$$

Dabei ist α der Winkel, den die beiden Vektoren einschließen.

Strecken in Vektorschreibweise

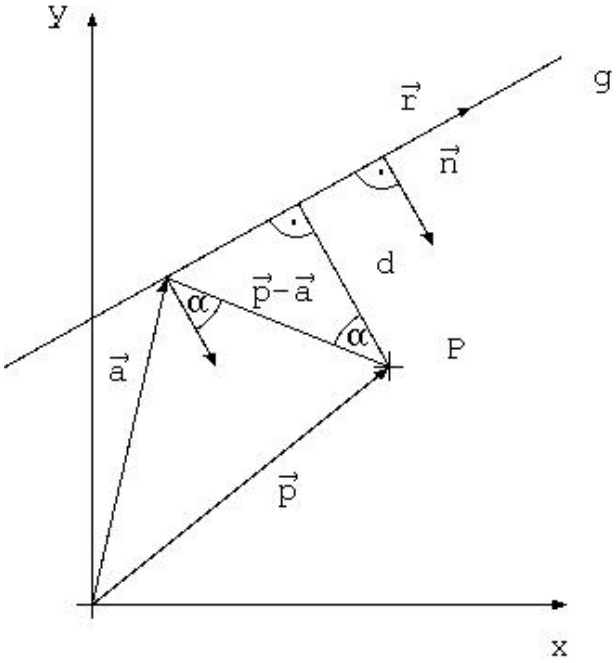
Um eine Gerade eindeutig darzustellen benötigt man zunächst zwei Vektoren $\vec{a}$ und $\vec{b}$, die je einen Punkt auf der Geraden darstellen. Für den sogenannten Richtungsvektor gilt also:

$$\vec{r} = \vec{b} - \vec{a}$$



Einen weiteren Punkt auf der Geraden findet man, indem man den Richtungsvektor zum Beispiel 2-mal zu $\vec{a}$ addiert. Folglich erhält man jeden Punkt der Geraden, in dem man für alle $t \in \Re$ den Richtungsvektor t-mal addiert. Eine Strecke, die durch Anfangs- und

Endpunkt gegeben ist, hat also folgende Darstellung:



$$s: x = \vec{a} + t * \left(\vec{b} - \vec{a}\right) \text{ für } t \in [0;1]$$

Abstand eines Punktes von einer Geraden

Da wir in unserem Spiel eine Kollision erkennen wollten, mussten wir den Abstand eines Punktes von einer Geraden bestimmen. Betrachten wir folgende Zeichnung:

Wir kennen bereits den Vektor $\vec{a}$ und den dazugehörigen Richtungsvektor $\vec{r}$. Außerdem kennen wir den Vektor $\vec{p}$ und wollen nun den Abstand d des Punktes P von der Geraden bestimmen.

Dazu benötigen wir zunächst einmal einen weiteren Vektor $\vec{n}$, der senkrecht zur Geraden steht. Sei $\vec{r} = \begin{pmatrix} x \\ y \end{pmatrix}$ dann gilt $\vec{n} = \begin{pmatrix} y \\ -x \end{pmatrix}$.

Nun wenden wir die weiter oben genannte Formel auf α an:

$$\cos(\alpha) = \frac{(\vec{p} - \vec{a}) \circ n}{|(\vec{p} - \vec{a})| * |\vec{n}|}$$

Da ein rechtwinkliges Dreieck vorliegt, gilt:

$$d = |(\vec{p} - \vec{a})| * \cos(\alpha)$$

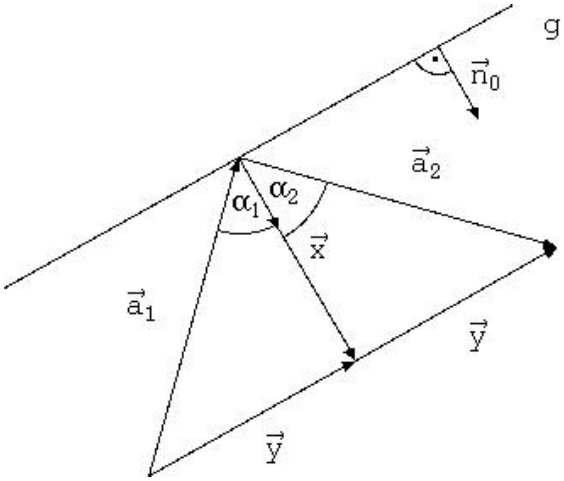
Nun ersetzen wir $\cos(\alpha)$ und erhalten die gewünschte Formel:

$$d = \frac{(\vec{p} - \vec{a}) \circ n}{|\vec{n}|}$$

Nun können wir sehr einfach den Abstand eines Punktes von einer Geraden ausrechnen.

Außerdem erhalten wir noch eine weitere Information: Liegt der Punkt auf der selben Seite der Geraden wie $\vec{n}$, so ist d positiv, liegt der Punkt auf der anderen Seite, so ist d negativ.

Reflexion



Nachdem wir eine Kollision erkannt haben, müssen wir auch entsprechend handeln. Dazu müssen wir uns überlegen, wie sich das Prinzip “Einfallswinkel gleich Ausfallswinkel” mit Vektoren umsetzen lässt.

Wir kennen den Vektor $\vec{a}_1$ und wollen den Vektor $\vec{a}_2$ errechnen, sodass gilt $\alpha_1 = \alpha_2$. Dazu benötigen wir nur noch den Einheitsnormalvektor $\vec{n}_0$ der senkrecht zur Geraden steht und die Länge 1 hat.
Da wir es wieder mit einem rechtwinkeligem Dreieck zu tun haben, gilt für die Länge l des Vektors $\vec{x}$:

$$l = |\vec{a}_1| * \cos(\alpha) = |\vec{a}_1| * \frac{a_1 \circ \vec{n}_o}{|\vec{a}_1|} = \vec{a}_1 \circ \vec{n}_0$$

Somit gilt für den Vektor $\vec{x}$:

$$\vec{x} = \vec{n}_0 * (\vec{a}_1 \circ \vec{n}_0)$$

Den Vektor $\vec{y}$ erhalten wir durch Addition:

$$\vec{y} = \vec{a}_1 + \vec{x} \text{ bzw. } \vec{y} = \vec{a}_1 + \vec{n}_0 * (\vec{a}_1 \circ \vec{n}_0)$$

Nun können wir den gesuchten Vektor durch Subtraktion berechnen:

$$\vec{a}_2 = 2 * \vec{y} - \vec{a}_1$$

Nun ersetzen wir noch $\vec{y}$ und erhalten folgende Formel:

$$\vec{a}_2 = 2*(\vec{a}_1 + \vec{n}_0 * (\vec{a}_1 \circ \vec{n}_0)) - \vec{a}_1 = \\ = \vec{a}_1 + 2*\vec{n}_0 * (\vec{a}_1 \circ \vec{n}_0)$$

Damit haben wir bereits alle Kenntnisse in Mathematik, die für unser Spiel notwendig sind, erworben.

Umsetzung im Programm

In unserem Spiel gibt es eine Klasse, die mit Vektoren umgehen kann und alle hier beschriebenen Grundrechnenarten beherrscht. Dadurch konnten wir die notwendigen Formeln in unser Programm übernehmen und erhielten eine wirklichkeitsgetreue und physikalisch richtige Kollision, die zuverlässig funktioniert.

Andreas Geyer-Schulz

9. Kugel-Kugel-Kollision

Unser Spiel sollte nun auch eine zweite Kugel enthalten. Dies stellte uns vor die größte Schwierigkeit, die wir zu bewältigen hatten: Die Kollision zweier Kugeln untereinander.
Wo liegen die Schwierigkeiten?

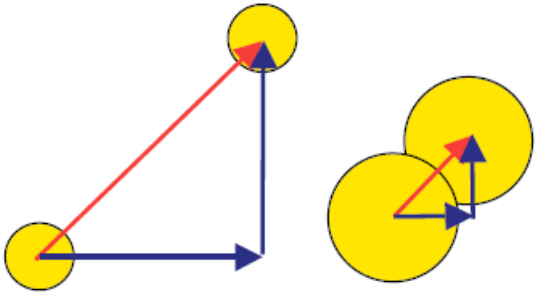
Die Kugeln können unterschiedliche Radien besitzen. Dadurch wird die Kollisionserkennung kompliziert.

- Kugeln können unterschiedliche Massen besitzen. Dadurch wird die Impulsübergabe von Kugel zu Kugel beeinflusst.
- Die Kugeln können unterschiedliche Geschwindigkeiten und damit auch einen unterschiedlichen Impuls haben. Dies führt dazu, dass auch die Impulsübergabe unterschiedlich abläuft.
- Die größte Schwierigkeit: Die Kugeln können aus allen erdenklichen Winkeln aufeinander treffen und auch in unterschiedlichen Positionen. Sie können sich beispielsweise nur streifen oder frontal aufeinander treffen.

Zur Lösung all dieser Probleme braucht man eine Menge Physik und Mathematik:

Zunächst die Kollisionserkennung:
Die Kollisionserkennung kann man sehr einfach mit Hilfe des „Satzes des Pythagoras „herleiten. Man berechnet mit Hilfe vom „Pythagoras“ den Abstand der Kugelmittelpunkte, der dann kleiner oder gleich groß wie die Summe der Radien ist, wenn die Kugeln kollidieren. Also kollidieren die Kugeln, wenn

gilt: $x^2 + y^2 \leq (r_1 + r_2)^2$ wenn $\begin{pmatrix} x \\ y \end{pmatrix}$ der Verbindungsvektor der Kugeln ist.



Im Programm sieht das dann so aus:

```
if ((kugel[i].x - kugel[z].x ) *
    (kugel[i].x - kugel[z].x ) +
    (kugel[i].y - kugel[z].y ) *
    (kugel[i].y - kugel[z].y ) <=
    (kugel[i].radius +
     kugel[z].radius) *
    (kugel[i].radius +
     kugel[z].radius)) {
   ...
}
```

Wobei Kugel z und i beliebige zu überprüfende Kugeln sind.

Nun kann das Programm sämtliche Kollisionen zweier Kugeln erkennen. Was wir jetzt brauchen, ist eine Kollisionsbehandlung. Wie ändern sich die Geschwindigkeiten und Bewegungsrichtungen der Kugeln? Bei der Kollision muss die Physik vollkommen stimmen, deshalb braucht man einige Gesetze und Formeln:

1. Der Impulserhaltungssatz: Der Impuls der beiden Kugeln muss vor und nach der Kollision der gleiche sein. Also muss $\vec{P}_{gesamt}$ konstant bleiben.

2. Nun wird dies verwendet um die Impulsänderungen ($\Delta\vec{p}_1$ und $\Delta\vec{p}_2$) der Impulse der Kugeln ($\vec{p}_1$ und $\vec{p}_2$) zu errechnen (die gestrichenen Werte sind die Werte nach der Kollision):

$$\vec{p}'_1 + \vec{p}'_2 = \vec{p}_1 + \Delta\vec{p}_1 + \vec{p}_2 + \Delta p_2$$
$$\Leftrightarrow$$
$$\Delta\vec{p}_1 = -\Delta\vec{p}_2$$

Deswegen kann man der Einfachheit halber auch sagen: $\Delta\vec{p} = \Delta\vec{p}_1$

3. Nun seien $\vec{x}_1$ und $\vec{x}_2$ die Ortsvektoren beider Kugeln zum Zeitpunkt der Kollision.

Die bei der Kollision erfolgende Impuländerung muss nun die Richtung der Differenz von $\vec{x}_1$ und $\vec{x}_2$ haben.

4. Nun wird gesetzt: $\Delta\vec{p} = \lambda \cdot (\vec{x}_1 - \vec{x}_2)_0$
   (Der Index $_0$ bedeutet dass es ein Vektor der Länge 1 ist).

5. Für einen Körper in Bewegung gilt:
   $$E_{Km} = \frac{m \cdot \vec{v} \circ \vec{v}}{2} = \frac{\vec{p} \circ \vec{p}}{m \cdot 2}$$

6. Jetzt wird λ mittels des Energieerhaltungssatzes bestimmt. Dieser besagt, dass $E_{Km} = E'_{Km}$ was wegen 5. äquivalent ist mit:

   $$\frac{1}{2} \cdot (\frac{1}{m_1} \vec{p}_1{}^2 + \frac{1}{m_2} \vec{p}_2{}^2)$$

   $$= \frac{1}{2} \cdot (\frac{1}{m_1} \vec{p}'_1{}^2 + \frac{1}{m_2} \vec{p}'_2{}^2)$$

7. Jetzt wird 2. verwendet. Wenn die Gleichung vereinfacht wird, erhält man:

   $$\frac{\vec{p}_1{}^2}{m_1} + \frac{\vec{p}_2{}^2}{m_2} = \frac{(\vec{p}_1 + \Delta\vec{p})^2}{m_1} + \frac{(\vec{p}_2 - \Delta\vec{p})^2}{m_2}$$

   Nun wird die rechte Seite ausmultipliziert und umgeschrieben. Man erhält:

   $$\Delta\vec{p}^2 \cdot (\frac{1}{m_1} + \frac{1}{m_2}) + 2 \cdot \Delta\vec{p} \circ (\vec{v}_1 - \vec{v}_2) = 0$$

8. Jetzt wird $\Delta\vec{p} = \lambda \cdot (\vec{x}_1 - \vec{x}_2)_0$ eingesetzt und die Gleichung nach λ aufgelöst. Man erhält:

   $$\lambda = 2 \cdot \frac{\vec{v}_2 \circ \vec{n}_0 - \vec{v}_1 \circ \vec{n}_0}{\frac{1}{m_1} + \frac{1}{m_2}}$$

   wobei $\vec{n}_0 = (\vec{x}_1 - \vec{x}_2)_0$ ist.

Damit wären wir (schon) am Ende der Physik zur Kugel-Kugel-Kollision. Nun muss das ganze in Quellcode übersetzt werden, der wie folgt aussieht:

```
bool Kollision::Kugel_Kugel (Kugel
&kugel1, Kugel &kugel2, double t){

   bool Rueckgabewert = false;

   float n_x = kugel1.Position_x -
```

```
            kugel2.Position_x;
float n_y = kugel1.Position_y –
            kugel2.Position_y;
float n_betrag = sqrt(( n_x *
         n_x ) + ( n_y * n_y));

if ((kugel1.Position_x -
      kugel2.Position_x) *
    (kugel1.Position_x –
      kugel2.Position_x) +
    (kugel1.Position_y -
      kugel2.Position_y) *
    (kugel1.Position_y –
      kugel2.Position_y) <=
    (kugel1.Radius +
      kugel2.Radius) *
    (kugel1.Radius +
      kugel2.Radius)) {

   double delta_t = 0;
   int Teiler = 10;
   while (n_betrag <
          kugel1.Radius +
          kugel2.Radius)  {
    kugel1.Bewegen (-t/Teiler);
    kugel2.Bewegen (-t/Teiler);
    delta_t = delta_t +
              t/Teiler;
    n_x = kugel1.Position_x -
          kugel2.Position_x;
    n_y = kugel1.Position_y –
          kugel2.Position_y;
    n_betrag = sqrt((n_x * n_x)
             + ( n_y * n_y));
   }

   float n0_x = n_x / n_betrag;
   float n0_y = n_y / n_betrag;

   float lamda = 2 *
    (((kugel2.Geschwindigkeit_x
        * n0_x) +
      (kugel2.Geschwindigkeit_y
        * n0_y) -
     ((kugel1.Geschwindigkeit_x
        * n0_x) +
      (kugel1.Geschwindigkeit_y
        * n0_y))) /
    ((1 / kugel1.Masse ) + (1 /
       kugel2.Masse)));
   kugel1.Geschwindigkeit_x =
     kugel1.Geschwindigkeit_x +
     ( 1 / kugel1.Masse * lamda
      * n0_x);
   kugel1.Geschwindigkeit_y =
     kugel1.Geschwindigkeit_y +
     ( 1 / kugel1.Masse * lamda
      *   n0_y);
   kugel2.Geschwindigkeit_x =
     kugel2.Geschwindigkeit_x –
     ( 1 / kugel2.Masse * lamda
      * n0_x);
```

```
        kugel2.Geschwindigkeit_y =
          kugel2.Geschwindigkeit_y -
          ( 1 / kugel2.Masse * lamda
           * n0_y);

        kugel1.Bewegen (delta_t);
        kugel2.Bewegen (delta_t);

        Rueckgabewert = true;
    }
    return Rueckgabewert;
}
```

Im Quelltext werden die einzelnen Formeln und Werte berechnet. Schließlich werden die Geschwindigkeiten der Kugeln mit Hilfe von λ und dem Richtungsvektor der Impulsänderungsrichtung neu berechnet.

Johannes Küchle

10. Allgemeines über unseren Kurs

Im Kurs „Breakout, Pinball & Friends“ trafen wir uns alle mit dem Ziel, Computerspiele zu programmieren.
Ohne jegliche Vorkenntnisse lauschten wir den Vorträgen unserer Kursleiter Jörg Richter, Daniel Jungblut und Alexander Schlüter, bis wir uns genügend physikalische, mathematische und programmiertechnische Grundlagen angeeignet hatten. Endlich wurden wir dann auf die Computer losgelassen. Wir haben uns zunächst mit der Visualisierung befasst. Unsere neu erworbenen Kenntnisse haben wir in Form kleiner Computerspiele umgesetzt. Für ein erstes Spiel genügten sehr einfache physikalische Gesetze. Wollten wir jedoch – etwa für ein einfaches Breakout-Spiel – Vorgänge wie Stöße an weichen Hindernissen, Stöße von Kugeln untereinander oder einen schiefen Tisch beschreiben, so benötigten wir dafür weitergehende mathematische und physikalische Konzepte, die wir bei dieser Gelegenheit kennen gelernt haben.

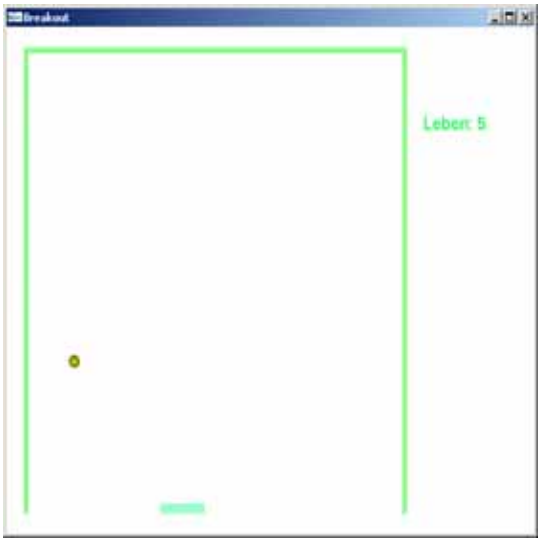
Nach zwei Wochen Arbeit war es so weit: Wir haben es geschafft ein Breakout-Spiel zu

programmieren, in dem man mit einer Art Schläger und einer Kugel Steine abschießen muss. Natürlich wurde unser Spiel noch weiter ausgefeilt, sodass es in nahezu perfektem Zustand unseren Eltern vorgestellt werden konnte, die ohne Ausnahme begeistert waren.

„Step by step“

1. Bande - Paddle - Kugel

Als Erstes programmierten wir für unser Breakout eine Randbegrenzung, eine Kugel und ein Paddle (Spielbrett, beweglich nach links und rechts).



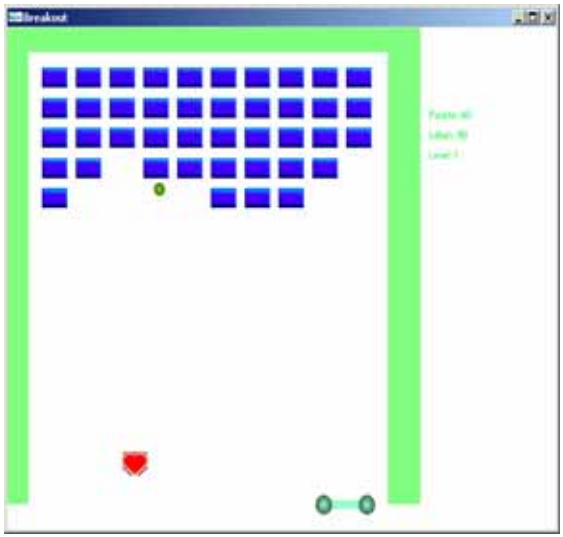
2. Punkte - Leben - Kugelpaddle - Goodies

Jetzt wurde das Spiel umfangreicher: Punkte bekommt man, wenn man ein Hindernis abtrifft.

Ferner kann aus zerstörten Hindernissen ein Goodie fallen, das entweder positive oder negative Wirkung haben kann (z.B. Vergrößerung des Paddles, ein zweiter Ball, etc.).

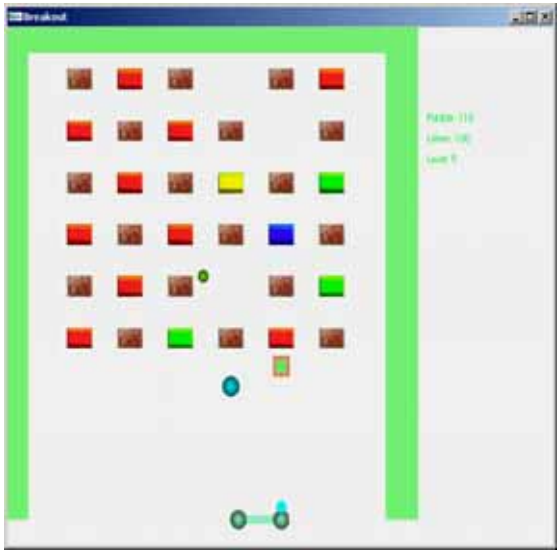
Zusätzlich gibt es eine Lebens- und eine Levelanzeige.

Mit dem „Kugelpaddle“ kann man besser ins Spielgeschehen eingreifen und mit etwas Geschicklichkeit die Kugel in beliebige Richtungen schießen.



3. Level - verschiedene Spielsteine

Ab dann ging es ans Designen von neuen Levels. Dazu programmierten wir neue Steine, die unterschiedliche Farben haben und entsprechend ihrer Farbe unterschiedlich oft abgeschossen werden müssen.



Spielanleitung

Ziel des Spiels

Das Ziel des Spieles ist es, mit dem Paddle die Kugel solange wie möglich im Spiel zu halten, um Punkte zu sammeln und den höchst möglichen Level zu erreichen. Die Kugel kollidiert an jedem Hindernis, am Paddle und an der Bande, kann allerdings auch neben dem Paddle ins Nichts herunterfallen. Dabei verliert man ein Leben. Beim Treffen eines Hindernisses erhält der Spieler Punkte.

Tastenfunktionen

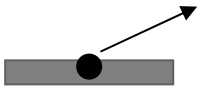
Taste	**Funktion**
a	Paddle nach links
d	Paddle nach rechts
Leertaste	einzelne Kugel losschießen und Kugel beschleunigen (Speed up)
r	Kugel auf das Paddle zurücksetzen
f	FPS- Anzeige an/aus
q	Spiel verlassen (Quit)

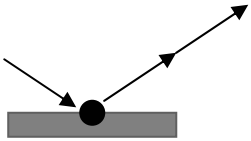
Steuerung des Paddles

Taste a:
(Paddle nach links)

Taste d:
(Paddle nach rechts)

Kugelfunktionen
Leertaste:
(Kugel starten)





Leertaste gedrückt halten:
(Kugel beschleunigen)
Taste r:
(Kugel wird auf das Paddle zurück gesetzt, wenn sie z.B. das Spielfeld verlassen hat)

FPS
Taste f: **FPS: 43**
(FPS- Anzeige an/aus)

Unterschiedliche Arten von Hindernissen

Es gibt rote, blaue, gelbe und grüne Hindernisse. Beim Treffen der Spielsteine werden Punkte angerechnet.
Benötigte Treffer bis der Stein zerstört ist:
Blau: 1, Grün: 2, Gelb: 3 und Rot: 4

Außerdem gibt es auch unzerstörbare Spielsteine (Backsteine). Sie geben keine Punkte und dienen nur dazu, die Kugel etwas von der Bahn abzulenken und das Spiel zu erschweren.

Bonuspunkte

Aus manchen Hindernissen fallen kleine Gegenstände, die mit dem Paddle eingesammelt werden können. Sie bewirken unterschiedliche Effekte.

Leben:

Todespille:

Ball verlangsamen:

Ball schneller:

Paddle breiter:

Paddle schmaler:

Extra Punkte:

Ein Level runter:

L-

Ein Level hoch:

Kanonenkugel:
(Kugel zerstört Steine ohne Kollision)

Extra Kugel:

Panda:
(Viele Pluspunkte)

Jana Gessert

11. Rotation / Präsentation

Am Samstag, den 3. September fand die sogenannte „Rotation“ statt. Dazu wurde jeder Kurs in vier Gruppen (A, B, C und D) unterteilt. So schlossen sich zum Beispiel alle A-Gruppen zusammen, um sich gegenseitig ihre Kursarbeit vorzustellen. Wir vom Kurs „Breakout, Pinball & Friends“ wählten Power-Point als Präsentationsmethode, wozu Johanna ein super Layout erstellte. Dank einer so guten Vorarbeit machte es richtig Spaß unser Wissen zusammenzutragen und eine professionelle Präsentation mit viel Bild-material vorzubereiten. Themen waren unter anderem:

- die Vorführung eines einfachen Konsolen-programms
- Grundlagen der Grafik-Programmierung
- die Grafik-Programmierung selbst
- das RGB-Farbsystem mit seinen bio-logischen Ursachen im menschlichen Auge
- bewegte Objekte und damit das Zusammenspiel zweier Funktionen (zeichnen und berechnen)
- Kollisionen (also eine der Grundlagen unseres Breakout-Spiels)
- das Objekt-Orientierte Programmieren (kurz OOP) mit Vererbung
- die Vektor-Kollision, die allerdings noch nicht vollständig funktionierte und somit auch noch nicht in unser Spiel eingebaut war

Im Anschluss an den Vortrag konnten die Zuhörer noch unser Spiel ausprobieren und uns Anregungen und Verbesserungsvorschläge mitteilen. Es war interessant, auch einmal in die anderen Kurse hineinzuschauen.

Die Rotation war für alle eine gute Vorbereitung auf die am Mittwoch, den 7. September folgende Präsentation, bei der auch unsere Eltern anwesend waren. Viele, die bei der Rotation noch nervös und aufgeregt gewirkt hatten, machten einen sehr viel gelasseneren Eindruck. Wie auch schon bei der Rotation kam unsere Präsentation gut an, sodass sich die Mühen und die damit verbundenen Überstunden gelohnt hatten. Zu den schon vorhandenen Themen kamen noch die Texturen hinzu, die unser Spiel optisch ansprechender machen sollten. Außerdem sprachen wir über einige interessante Fakten, zum Beispiel, dass der Quelltext unseres Programms 2712 Zeilen lang ist und somit jeder von uns Kursteilnehmern im Durchschnitt 226 Zeilen programmiert hat. Wegen unserer chinesischen Gäste hatten wir die Folien ins Englische übersetzt. Chaoyi, der an unserem Kurs teilnahm, bildete mit Johanna, Anton und

Robin die erste Vortragsgruppe. Er erklärte die Themen Grafik und Bewegung und stellte sein selbst programmiertes Spiel vor. Seine Vorträge waren auf Englisch, doch da auch kleinere Kinder und ältere Erwachsene anwesend waren, erklärten wir seine Themen noch einmal auf Deutsch. Zusammenfassend kann man sagen, dass es ein anstrengender aber erfolgreicher Tag war. Wir hatten viel Spaß und haben präsentationstechnisch einiges dazugelernt

Robin Lingstädt

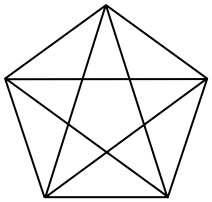
12. Der Exkursionstag in Heidelberg

Am 5.9.2005 gingen alle Teilnehmer der Akademie nach Heidelberg. Dort hörten wir uns im DKFZ (Deutsches Krebsforschungs-zentrum) zwei Vorträge über Krebsentstehung

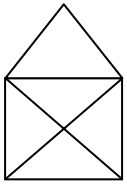
an. Diese wurden, damit sie auch unsere chinesischen Freunde verstehen konnten, auf Englisch gehalten. Danach begann der kursspezifische Teil.

Wir vom Kurs „Pinball, Breakout & Friends“ gingen zuerst ins Kirchhoff-Institut für Physik (KIP), in welchem wir das Foucault’sche Pendel besichtigten. Dieses, von Jean Leon Foucault erdachte Pendel, demonstriert die Drehung der Erde um sich selbst. Es schwingt immer in einer Ebene. Durch die Drehung der Erde dreht sich die unter dem Pendel angebrachte Holzplatte in ca. 32 Stunden einmal um sich selbst, wobei das Pendel ungefähr alle 30 min einen von 48 auf der Platte befestigten Nägeln umwirft. Da ein Beobachter sich auch mit der Erde dreht, kommt es einem vor als würde sich das Pendel drehen, in Wirklichkeit aber dreht sich alles um das Pendel. Da durch Reibung das Pendel gebremst wird und es somit nach einer Zeit zum Stehen kommen würde, wird dieser Verlust durch eine, in der Mitte eingebaute, Magnetspule ausgeglichen, die das Pendel leicht beschleunigt. Da aber auch seitliche Irritationen Auftreten können, ist im KIP 40 cm unterhalb der Aufhängung ein Charron-Ring angebracht, der die seitliche Auslenkung dadurch verhindert, dass sich das Seil bei halber Auslenkung an den Ring legt und die

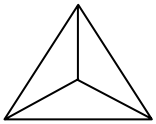
dabei entstehenden Reibungskräfte in Wärmeenergie umgewandelt werden.
Danach gingen wir zum „Interdisziplinären Zentrum für Wissenschaftliches Rechnen“ (IWR). Hier führte uns Dr. Michael Winkler in die Graphentheorie ein, wobei wir immer wieder kleine Aufgaben in Gruppen lösen mussten.
Zuerst einmal: Was sind Graphen? Graphen sind mathematische Modelle, die der Darstellung von Netzwerken dienen. Sie bestehen aus Knoten (Endpunkten von Kanten) und aus Kanten (Verbindungen).
Was gibt es für Graphen? Es gibt vollständige, einfache und planare Graphen. Die Graphen werden je nach Eigenschaften den oben genannten Typen zugeordnet.
Hier 3 Beispiele:



→ Dies ist ein vollständiger Graph, weil jeder Knoten durch eine Kante mit allen anderen Knoten verbunden ist und es keine doppelten oder ringförmigen Kanten gibt.



→ Das ist ein einfacher Graph, da es keine Ringe und doppelten Kanten gibt, er jedoch nicht vollständig ist.



→ Hier ist es ein planarer Graph, denn es gibt keine Kantenkreuzungen und keine Schleifen.

Darstellung von Graphen

Nachdem wir nun verschiedene Graphentypen kannten, wurden uns die verschiedene Darstellungsmethoden gezeigt und erklärt. Im Folgenden eine kurze Zusammenfassung der Darstellungen, die wir bearbeitet haben.

Graph/Bild

Wird ein Graph als Bild dargestellt, so dient dies der übersichtlichen Darstellung des Graphen, weshalb es meist leicht verständlich oft an der Wirklichkeit orientiert ist.

Listenmodell

Ein Listenmodell ist eine Liste von Knoten und Kanten eines Graphen. Es hat den Vorteil, dass es exakter und besser zu verarbeiten ist. In einer Liste werden zuerst alle Knoten (z.B. Städte) angegeben und dann die Kanten (Verbindungen zwischen Städten).

Adjazenzmatrix

Wird ein Graph im Computer bearbeitet, so wird er dort meist als Adjazenzmatrix gespeichert. Dies spart Speicherplatz und erlaubt einen schnelleren Zugriff auf einzelne Verbindungen. Wie links ist es eine 0/1 Matrix. Die 1 steht dafür, dass es eine, die 0 dafür, dass es keine Kante zwischen den Knoten gibt.

Welt → Modell → Graph

Was ist jetzt mit Welt → Modell → Graph gemeint?

Hier ist der Zusammenhang der drei Komponenten gemeint, denn ein Modell ist nichts anderes als eine Abstraktion der Wirklichkeit, der Welt. Und der Graph? Der Graph ist die mathematische Version des Modells.

Minimal Spanning Tree

Beim Minimal Spanning Tree (MST) geht es um die kürzeste Verbindung mehrerer Knoten, wobei es wirklich nur um die kürzeste Verbindung und nicht um einen Rundweg oder anderes geht. Man kann den MST mit verschiedenen Algorithmen berechnen. Hier sind die zwei, die wir behandelt und ausprobiert haben:

Knoten	1	2	3	4
1	0	0	0	1
2	0	0	1	0
3	0	1	0	1
4	1	0	1	0

Der Algorithmus von Kruskal

Dieser von Joseph Kruskal gefundene Algorithmus dient dem Finden des kürzesten Weges. Er sortiert erst alle Kanten der Länge nach und färbt dann die kürzeste Kante. Ist dies geschehen werden die anderen Kanten in aufsteigender Länge durchlaufen und jede Kante zur Lösung hinzugefügt, die keinen Kreis mit den vorhergegangenen bildet. Hier ergibt sich dann die kürzeste Verbindung der Knoten.

Der Algorithmus von Prim

Dieser Algorithmus beginnt bei irgendeinem Knoten im Graphen und sucht die kürzeste Kante die diesen Knoten mit einem anderen verbindet. Diese Kante und der neu

hinzugekommene Knoten werden nun dem MST hinzugefügt. Nachdem dies erledigt ist, wird nun wieder die kürzeste Kante gesucht, die einen der schon im MST vorhandenen Knoten mit einem noch nicht vorhandenen Knoten verbindet, das wiederholt sich so lange, bis alle Knoten im MST vorhanden sind.

Problem des Handlungsreisenden

Im Gegensatz zum MST handelt das Problem des Handlungsreisenden (Traveling Salesman Problem/ TSP) vom Problem eines Rundreisenden der durch bestimmte Orte will und dies auf dem kürzesten Weg. Ein recht alltägliches Problem, denn es beschäftigt täglich Spediteure auf der ganzen Welt: z.B. Welches ist der kürzeste Weg durch die 15 größten Städte Deutschlands? Auf dieses Ergebnis kann man mit Hilfe von verschiedenen Algorithmen kommen.

Im Anschluss an diesen Vortrag durften wir in Gruppen durch die Heidelberger Innenstadt gehen und trafen uns schließlich alle wieder im „Bootshaus“, wo wir den Tag in Heidelberg mit einem Abendessen beendeten, bevor wir zurück nach Adelsheim fuhren.

Christian Schweizer