# Alleyway

Mit Containerklassen und polymorphen Methoden ist das Verwalten von Spielfiguren eine einfache Sache. Die Verantwortlichkeit für die richtigen Spielzüge liegen in den Methoden der Spielsteine. Die Spielklasse muß sich um Einzelheiten des Spielgeschehens nicht mehr kümmern.

*Franz Fiala*

Bei einer größeren Anzahl von Objekten, insbesondere bei polymorphen Objekten, besteht der Bedarf, sie einerseits zur Laufzeit zu generieren (`new`) und anderseits gemeinsam zu bearbeiten. Man benutzt dazu sogenannte Containerklassen, die in der Lage sind, beliebig viele Objekte aufzunehmen und gemeinsam zu bearbeiten. Die Containerklasse enthält nicht die Objekte selbst sondern nur Pointer auf die Objekte.

Eine Containerklasse ist ein Sammelbehälter für polymorphe Objekte. Es kann als Array oder als Liste ausgeführt sein. Es gibt solche Containerklassen fertig in den bekannten Klassenbibliotheken, etwa in den **Foundation Classes** von Microsoft oder in der **Class Library** von Borland. Hier unternehmen wir einen Versuch, eine solche Klasse mit einfachen Mitteln nachzubilden.

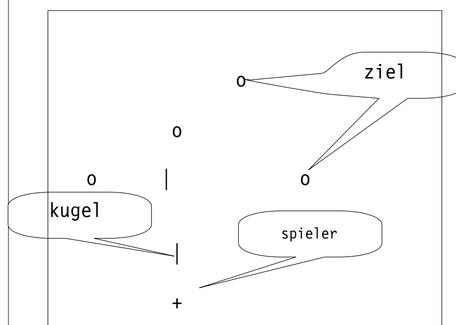
Als Versuchsobjekt dient eine Grundstruktur für das Bildschirmspiel Alleyway (Nintendo), bei dem es gilt, Anordnungen von Klötzen zu treffen. Auf der unteren Bildschirmkante bewegt sich eine Spielfigur (`SPIELER` '+'), die `KUGEL`n abschießen kann ('|') und dabei `ZIEL`e ('o') treffen soll.

Im professionellen Spiel sind die Kugelbahnen nicht nur einfach geradlinig, sondern können durch eine Spielerbewegung gegenüber der senkrechten Richtung abgelenkt werden. Die Bälle werden an den Rändern oder an den anderen Spielfiguren reflektiert. Weiters erfolgt die Darstellung im Grafik-Mode uvam. Der Grundaufbau des Spieles kann für viele ähnliche Spiele (Packman, Space-Invaders u.a.) abgewandelt werden.

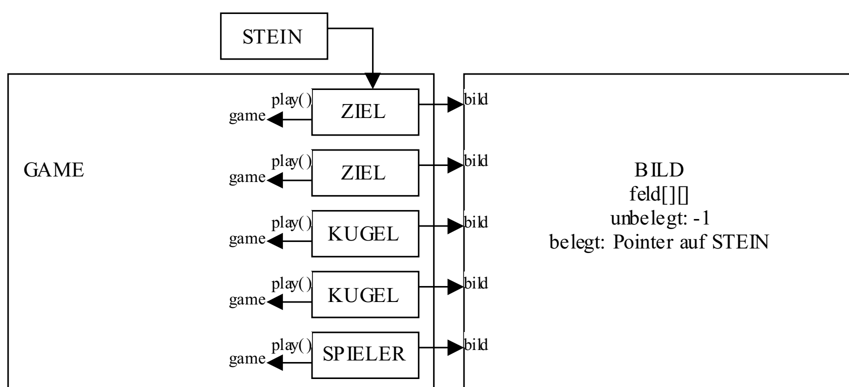
Derzeitiger Programmablauf: Am Beginn gibt es ein einziges Ziel und einen Spieler. Wird das Ziel getroffen, teilt sich das Ziel in zwei weitere Ziele. Die Anordnung der Ziele erfolgt zufällig. Auf Grafik wurde verzichtet, um vom eigentlichen Problem nicht allzusehr abzulenken und den Kode kurz zu halten.

Das hier gezeigte Programm ist keineswegs ein fertiges Programm. Es ist lediglich ein Programmgerüst, das zeigt, wie Objekte in einem Programm zusammenarbeiten können. Es kann aber Ausgangspunkt für weitere Experimente mit objektorientierten Programmen sein.

*Bildschirmdarstellung*




*Zusammenwirken der Containerklasse GAME mit der polymorphen Methode play() der Spielsteine und deren Orientierung am Spielfeld BILD.*
*STEIN Basisklasse für ZIEL, KUGEL und SPIELER.*
*GAME Container für Pointer des Typs STEIN, spielt in regelmsäsigen Abständen virtuelle Methode play() für alle Spielsteine.*
*BILD ist ein Abbild des Bildschirmspielfeldes und hilft den Spielsteinen, die Umgebung schnell zu erfassen.*




**Aufgaben**

- Am Spielbeginn sind 20 zufällig angeordnete Ziele zu generieren, wobei sichergestellt werden soll, daß keines der Ziele durch den Zufallsgenerator überschrieben wird.
- Ein Ziel soll nicht stillstehen, sondern sich in einer definierten Umgebung frei bewegen können.
- Ziele verändern ihre Form beim ersten Treffer und können danach nur durch Sonderleistungen des Spielers beseitigt werden.
- Der Spielstein soll sich nur für die Dauer des Tastendrucks bewegen.
- Es ist ein Ziel zu generieren, das sich Am Bildschirm in 45 Grad-Schritten bewegt und an den Bildrändern richtig reflektiert wird.
- Ein neuer Spielstein `BARRIERE` ist zu entwerfen, hinter dem sich `ZIEL`e verstecken können und die der Spieler durch geneigt gesendete `KUGEL`n treffen kann.
- Entwurf von Spielfeldern für Brettspiele (Dame, Schach), wobei weitere Spielsteine und Zugregeln zu definieren sind.
- Es sind 5 Ziele zu definieren, die sich in Form eines Verbandes zunächst horizontal bewegen. Bei jeder Berührung mit den Rändern des Spielfeldes kommen die Ziele dem Spieler näher.
- Die Ziele versuchen ab Spielbeginn oder ab einem späteren Zeitpunkt ihrerseits den Spieler zu treffen.
- Die Spielfiguren und das Spielfeld erhalten Farben. Jede Bewegung einer Spielfigur wird akustisch untermalt.
- Das Spielfeld wird eingerahmt und dadurch eingeengt.
- Dreidimensionaler Spielraum mit Ansicht in einem geteilten Bildschirm (Aufriß und Kreuzriß). Neue Bewegungsmöglichkeiten des Spielers und der Ziele.
- Spieler und/oder Ziele können sich jeweils eine kurze Zeit tarnen und können dann nicht getroffen werden.
- Es gibt einen maximalen Vorrat an Kugeln, der gleichzeitig das Spielende markiert.
- Das Spiel ist im Grafik-Modus auszuführen.
- Der Spieler kann sich in allen Richtungen bewegen.
- Es gibt Barrieren, die die Ziele verstecken, daher muß sie der Spieler durch vielseitigere Bewegungsmöglichkeiten aufspüren können.

Das Programm besteht aus folgenden Klassen

| | |
|---|---|
| `GAME` | erzeugt in seinem Konstruktor die Anfangsstellung (1 Spieler, 1 Ziel) und ist der Motor des Spiels, indem in einer Endlosschleife mit einer wählbaren Zeitverzögerung speed ein quasi kontinuierlicher Spielablauf vorgetäuscht wird. `GAME` ist eine Containerklasse. Es wird ein Array von Pointern verwaltet, dessen Größe auf `SPIELSTEINE` begrenzt ist. Jeder Pointer der Basisklasse `STEIN` zeigt auf einen Spielstein am Bildschirm, egal, ob es Spieler, eine Kugel oder ein Ziel ist. `GAME` verwaltet die Spielsteine, indem es in mit der Methode `play()` jeden Spielstein zyklisch aufruft und diesen bei Bedarf löscht (Wenn der Stein am Bildrand ankommt, wenn er abgeschossen wird, wenn er als ungültig erklärt wird) und diesen Spielstein zum Spielen auffordert (über polymorphe Funktion `play()`, die jeder eizelne Spielstein haben muß). Jeder verwaltete Spielstein hat einen Index im Container. Dieser Index ist auch dem Spielstein bekannt. Daher kann der Spielstein auch mit anderen Spielsteinen in Kontakt treten. |
| `BILD` | `BILD` ist eine Hilfsklasse, die zu einfacheren Orientierung am Spielfeld dient. Die Basis von Bild ist ein Array mit Bildschirmgröße, bei dem unbelegte Positionen auf -1 initialisiert sind und belegte Positionen mit dem Indexwert der Spielverwaltung. Mit `BILD` kann ein bewegter Spielstein einfach feststellen, ob er im Begriff ist, einem anderen Stein zu begegnen. Es wäre prinzipiell auch ohne Bild möglich, nur müßte man dazu bei jedem Zug die Container-Klasse `GAME` durchsuchen lassen, was vielleicht bei wenigen Spielsteinen vorzuziehen wäre, aber bei vielen Spielsteinen unzweckmäßig erscheint. |
| `STEIN` | Stein ist eine abstrakte Basisklasse, die selbst nicht verwendet wird. Sie beschreibt und definiert die Methoden, über die ein Spielstein verfügt. Die virtuelle Funktion `play()` ist in jeder der Kindklassen `SPIELER`, `KUGEL` und `ZIEL` anders definiert und wird durch die Containerklassenmethode play() aufgerufen. Jeder Spielstein hat genaue Kenntnis über das Spielfeld `BILD` und das Spiel `GAME`. Das wird erreicht, indem jedem Spielstein zwei Pointer auf diese wichtigen Spielbestandteile mitgegeben werden. Über `BILD` kann sich jeder Spielstein über seine Umgebung rasch informieren. Diese Orientierung wird in der Funktion `movetest()` demonstriert. `movetest()` entscheidet, ob ein Zug zur Ausführung kommen kann oder ob es zu einer Begegnung mit anderen Steinen kommt. |
| `SPIELER` | `'+'`, abgeleitet von `STEIN`, kann sich mit den Tasten 'S' und 'D' horizontal bewegen, kann mit 'X' gestoppt werden und kann mit 'E' feuern. Weiters bewirken die Tasten '1',...'6' verschiedene Spielgeschwindigkeiten. Am Bildschirmrand bleibt der Spielstein stehen. Die Taste `ESC` beendet das Programm. |
| `KUGEL` | '\|', abgeleitet von `STEIN`, wird durch `SPIELER` generiert (eine Bildschirmzeile über dem Spieler) und hat die Eigenschaft, sich in senkrechter Richtung fortzubewegen Trifft `KUGEL` beim nächsten Zug auf ein Hindernis, oder auf den Bildschirmrand, wird es über die Eigenschaft `tot=1` zum Löschen durch `GAME` vorbereitet. |
| `ZIEL` | 'o', abgeleitet von `STEIN`, kann von `KUGEL` getroffen werden. Ein Treffer bewirkt eine „Zellteilung" die Spielklassenmethode `play()` generiert bei einem Treffer zwei neue Ziele und löscht das getroffene Ziel. |
| `SPIELTYP` | In diesem Programm zu Demonstrationszwecken wird es zwar nicht verwendet, doch kann es manchmal nützlich sein zu wissen, mit welchem Objekt man es zu tun hat, beispielsweise wenn man ihm begegnet. Dazu ist der Aufzählungstyp `SPIELTYP` gedacht. Damit könnte das Verhalten der `KUGEL` bei Auftreffen auf verschiedene Objekte verschieden reagieren. |

```
// GAME.H

#Include <conio.h>
#include <stdlib.h>
#include <string.h>

// Spielfeld
#define MAXX 80
#define MAXY 25

#define SPIELSTEINE 10

#define SPEED 200

enum SPIELTYP
{ TYPUNDEF, TYPSPIELER, TYPKUGEL, TYPZIEL };

class STEIN;
class BILD;
class GAME;

STEIN *create_ziel_random(GAME *g, BILD *b);
STEIN *create spieler(GAME *g, BILD *b);

class STEIN
{
private:
  unsigned char c;
  int getroffen;
  int index;
protected:
  SPIELTYP typ; // Spieler, Geschoss, Ziel
  int tot; // ist zu l"schen
  GAME *game;
  BILD *bild;
  int x; // aktuelle Position
  int y;
  int dx; // Bewegungsrichtung
  int dy;
public:
  int gettyp() { return typ; }
  void hit() { getroffen=1; }
  char istot() { return tot; };
  char ishit() { return getroffen; };
  virtual int play() { return -1; }
  /* Korrekturaddition beim Zeichnen und
Loeschen */
  void draw() { gotoxy(x+1,y+1); putch(c); }
  void hide() { gotoxy(x+1,y+1); putch(' '); 
}
  void draw(int i) { draw(); index=i; }
  int movetest();
  int getindex() { return index; }
  void move();
  unsigned char getx() {return x;}
  unsigned char gety() {return y;}
  STEIN
 (GAME *g, BILD *b,
  int x0, int y0, int c0,
  SPIELTYP t=TYPUNDEF,
  int dx0=0, int dy0=0)
  {
    x=x0; y=y0;
    dx=dx0; dy=dy0;
    c=c0;
    game=g;
    bild=b;
    getroffen=0;
    tot=0;
    index=0;
    typ=t;
  }
  ~STEIN() { hide(); }
};

/*
Das Spielfeld geht von 0..79 und 0..24,
diese Koordinaten sind in feld gespeichert
die Bildschirmbildkoordinaten verlaufen
im Bereich von 1..80 und 1..25
in diesem Bereich wird gezeichnet.
Die erforderliche Korrekturaddition wird
ausschliesslich beim Schreiben und Loeschen
ausgefuehrt.
*/
class BILD
{
  int feld[MAXX][MAXY];
public:
  int get(int x, int y) { return feld[x][y];
}
  void set(STEIN *s)
  {feld[s->getx()][s->gety()]=s->getindex();}
  void set(int x, int y, int i)
    { feld[x][y]=i; }
  void del(int x, int y)
    { feld[x][y]=-1; }
  int in(int x, int y)
    { return ((x>=0 && x<MAXX) &&
      (y>=0 && y<MAXY)); }
  BILD()
    { memset( feld, -1,
       sizeof(feld[0][0])*MAXX*MAXY);
      clrscr(); }
  ~BILD() { clrscr(); };
};

class SPIELER : public STEIN
{
public:
  int play();
  SPIELER(GAME *g, BILD *b, int x) :
     STEIN(g,b,x,24,'+',TYPSPIELER)
  {  }
};

class ZIEL : public STEIN
{
public:
  int play();
  ZIEL(GAME *g, BILD *b, int x, int y) :
     STEIN(g,b,x,y,'o',TYPZIEL)
  {  }
};

class KUGEL : public STEIN
{
public:
  int play();
  KUGEL(GAME *g, BILD *b, int x, int y) :
     STEIN(g,b,x,y,'^',TYPKUGEL,0,-1)
  {  }
};

class GAME
{
  STEIN *stein[SPIELSTEINE];
  BILD *bild;
public:
  int speed;
  void hit(int i) { stein[i]->hit(); }
  int set(STEIN *);
  void play();
  GAME(BILD *b);
  ~GAME();
};
```

```
// GAME.CPP

#include <stdio.h>
#include <dos.h>
#include <ctype.h
#include "game.h"

int ende=0;

int STEIN::movetest()
{
  if ((dx==0) && (dy==0))
    return 4; // keine Bewegung

  if (bild->in(x+dx,y+dy))
  {
    if (bild->get(x+dx,y+dy)==-1)
      return 0; // leeres Feld
    else
      return 1; // besetztes Feld
  }
  else
    return 2; // Bildrand
}

void STEIN::move()
{
  int index = bild->get(x,y);
  hide();
  bild->del(x,y);
  x+=dx;
  y+=dy;
  bild->set(x,y,index);
  draw();
}

int SPIELER::play()
{
//Tastaturabfrage
//setzt Bewegungswunsch
//und Abschuß
  if (kbhit())
  {
    char c=getch();
    switch (toupper(c))
    {
    case 27:  // Abbruch
      ende=1;
      break;
    case '1':
      game->speed=1000;
      break;
    case '2':
      game->speed=500;
      break;
    case '3':
      game->speed=200;
      break;
    case '4':
      game->speed=100;
      break;
    case '5':
      game->speed=50;
      break;
    case '6':
      game->speed=1;
      break;
    case 'S': // nach links
      dx=-1;
      break;
    case 'D': // nach rechts
      dx=1;
      break;
    case 'X': // Halt
      dx=0;
      break;
    case 'F': // Feind
      dx=0;
      break;
    case 'E': // Feuer
      dx=0;
      if (bild->get(x,y-1)==-1)
      {
        STEIN *k =
          new KUGEL(game,bild,x,y-1);
        game->set(k);
      }
      else
      {
        // Stein als getroffen markieren
        game->hit(bild->get(x,y-1));
      }
      break;
    }
  }

  switch (movetest())
  {
  case 0: //  OK, Stein kann bewegt werden
    move();
    return 1;
  case 1: // Feindberührung, stehen bleiben
    dx=0;
    break;
  case 2: // Spielfeldrand, stehen bleiben
    dx=0;
    break;
  case 4: // keine Bewegung
    break;
  }
  return 0;
}

int KUGEL::play()
{
  switch (movetest())
  {
  case 0: // unbelegtes Spielfeld
    move(); // Position verschieben
    return 1;
  case 1: // getroffen
    tot=1;
    game->hit(bild->get(x+dx,y+dy));
    return 1;
  case 2: // Außerhalb des Spielfeldes
    tot=1;
    break;
  case 4: // keine Bewegung
    break;
  }
  return 0;
}

int ZIEL::play()
{
  if (ishit())
  {
    STEIN *s;
    s=create_ziel_random(game, bild);
    game->set(s);
    s=create_ziel_random(game, bild);
    game->set(s);
  }
  return -1;
}

STEIN *create_ziel_random(GAME *g, BILD *b)
{
  // erzeugt ein zufälliges Zeichen
  // im Bereich des Spielfeldes
  // außer in der untersten Bildschirmzeile

  STEIN *t =
 new ZIEL(g, b,(rand()%MAXX),
    rand()%(MAXY-1));
  return t;
}

STEIN *create_spieler(GAME *g, BILD *b)
{
  // erzeugt einen Spieler
 // in der untersten Bildschirmzeile
  STEIN *t = new SPIELER(g,b,41);
  return t;
}

GAME::GAME(BILD *b)
{
  // Spielsteinpositionen initialisieren
  bild=b;
  for (int i=0; i<SPIELSTEINE   i++)
    stein[i]=NULL;
  // Spieler und Ziel einsetzen
  STEIN *s;
  s = create_ziel_random(this,bild);
  if (!set (s))
    printf(„**AUS**");
  s = create_spieler(this,bild);
  if (!set (s))
    printf(„**AUS**");
  speed = SPEED;
}

GAME::~GAME()
{
  // Speicherplatz freigeben
  for (int i=0; i<SPIELSTEINE   i++)
    if (stein[i]!=NULL)
      delete stein[i];
}

int GAME::set(STEIN *s)
{
  // In der Liste der Spielsteine vormerken
  for (int i=0; i<SPIELSTEINE   i++)
  {
    if (stein[i]==NULL)
    {
      stein[i]=s;
      stein[i]->draw(i);
      bild->set(s);
      return -1;
    }
  }
  return 0;
}

void GAME::play()
{
  while (!ende)
  {
    for (int i=0; i<SPIELSTEINE    i++)
    {
      if (stein[i]!=NULL)
      {
         if (stein[i]->istot())
         {
           delete stein[i];
           stein[i]=NULL;
           continue;
         }
         if (stein[i]->ishit())
         {
           stein[i]->play();
           delete stein[i];
           stein[i]=NULL;
           continue;
         }
         if (stein[i]->play())
         { // Stein wurde bewegt
         }
         else
         {
         }
      }
    }
    delay(speed);
  }
}

void main(void)
{
  directvideo=1;
  wscroll=0;
  BILD *bild = new BILD;
  GAME *game = new GAME(bild);
  game->play();
  delete game;
  delete bild;
}
```

*Mit Computern irrt man viel genauer.*