7.3 ARCADE GAMES, FROGGER

EINFÜHRUNG

Viele Computergames, die du auf Spielconsolen und im Internet findest, bestehen aus Bildern, die sich über einen Hintergrund bewegen. Manchmal ist sogar der Hintergrund bewegt, insbesondere wenn die Spielfiguren zum Rand des Bildschirmfenster laufen, damit dem Spieler der Eindruck eines Scenarios vermittelt wird, das wesentlich grösser als das Bildschirmfenster ist. Die Spielanimation erfordert zwar eine grosse Rechenleistung, ist aber grundsätzlich einfach zu verstehen: Zu sich in kurzer Zeit folgenden Zeitpunkten wird in der sogenannten Game-Loop der Bildschirminhalt neu berechnet, der Hintergrund und die Bilder der Spielfiguren in einen nicht sichtbaren Bildpuffer kopiert und dann dieser als Ganzes im Fenster dargestellt (gerendert). Werden mehr als ungefähr 25 Bilder pro Sekunden dargestellt, ergibt sich für das Auge ein fliessende Bewegung, bei weniger Bildern ist die Bewegung ruckartig.

In vielen Spielen interagieren die Spielfiguren durch Zusammenstösse. Die Behandlung von Kollisionen ist also für viele Spiele fundamental. Gut ausgebaute Gamelibraries wie JGameGrid unterstützen den Programmierer dabei durch eingebaute Kollisionsdetektion unter Verwendung des Eventmodells. Man definiert dabei, welches die potentiellen Kollisionspartner sind und das System ruft bei einem Kollisionsereignis automatisch einen Callback auf.

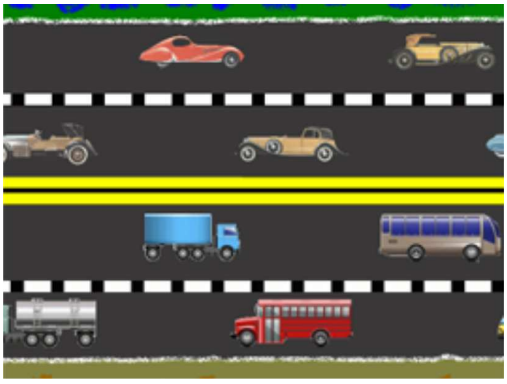
PROGRAMMIERKONZEPTE: *Gamedesign, Sprite, Actor, Kollision, Supervisor*

GAME-SZENARIO

Bei der Entwicklung von Computergames ist es wichtig, dass du dir zuerst ein möglichst detailliertes Spielszenario ausdenkst und stichwortartig als Pflichtenheft aufschreibst. Meist sind deine Ansprüche im ersten Anlauf zu hoch und du musst versuchen, das Game soweit zu vereinfachen, dass du lauffähige Teilversionen entwickeln kannst, die du schrittweise erweiterst. Die Kunst besteht darin, dass du das Programm so allgemein schreibst, dass du bei den nachfolgenden Erweiterungen den bestehenden Code nicht stark abändern musst, sondern nur zu ergänzen brauchst. Auf Anhieb gelingt dies aber selbst bei professionellen Programmieren selten, so dass bei der Entwicklung von Games Euphorie und Frustration eng beieinander liegen. Um so grösser ist deine Freude und Genugtuung, wenn du dein eigenes, ganz persönliches Computerspiel vorführen und spielen lassen kannst.

Der Weg zum kompetenten Gameprogrammierer führt über die Entwicklung von bekannten Spielen, die du in einer persönlichen Variante und mit deinen eigenen Spritebildern implementierst. Dabei ist es in der Ausbildungsphase nicht so wichtig, ob diese Games bereits fix-fertig erhältlich sind, denn es geht ja nicht in erster Linie darum, dass du viel damit spielst, sondern dass du lernst, wie man sie entwickelt.

Ein bekanntes Spiel ist das Froschspiel (Frogger). Es hat folgendes lustiges Scenario:
Ein Frosch versucht, sich über eine stark befahrene Strasse zu bewegen und zu einem Teich zu gelangen. Kollidiert er mit einem Fahrzeug, so verliert er sein Leben. Das Ziel ist es, ihn mit den Cursortasten sicher über die Strasse zu bringen. In deiner Implementierung gibt es vier Fahrbahnen, zwei mit gegeneinander laufenden Lastwagen und Bussen, und zwei mit gegeneinander laufenden Oldtimer-Autos.



Zwei mögliche Entwicklungswege stehen dir offen: Du realisierst zuerst die Bewegung des Froschs oder die Bewegung der Fahrzeuge. Nachher fügst du den Kollisionsmechanismus und die Verrechnung der Spielpunkte sowie die Behandlung des Spielendes (Game-Over) hinzu.

In GameGrid werden die Fahrzeuge als Instanzen der Klasse *Car* modelliert, die aus Actor abgeleitet ist. In der Methode **act()** wird die Bewegung der Fahrzeuge programmiert. Als Sprites verwendest du die Bilder *car0.gif,..car19.gif,* die sich in der Distribution von TigerJython befinden. Du kannst natürlich auch eigene Bilder verwenden (sie sollten maximal 70 Pixel hoch und maximal 200 Pixel breit sein und einen transparenter Hintergrund aufweisen).

Für Arcade-Games üblich ist die Verwendung eines Gameboard mit einer Zellengrösse von 1 Pixel. d.h. das Gitter entspricht dem Pixelraster. Als Fenstergrösse wählst du 800 x 600 Pixel und lädst ein Hintergrundbild *lane.gif* der Grösse 801 x 601 Pixel, das das Strassenszenario darstellt. In der Funktion **initCars()** erzeugst du die 20 Car-Objekte und überlegst dir, wo und in welcher Blickrichtung du sie in das Gameboard einfügen willst.

Das Bewegen der Autos mit der Methode **act()** ist einfach: Du schiebst sie mit *move()* weiter und lässt die nach rechts laufenden Autos von rechts nach links bzw. die nach links laufenden Autos von links nach rechts springen, wenn sie aus dem Bildschirmfenster hinausgefahren sind. Beachte dabei, dass die Location-Koordinaten eines Actors auch ausserhalb des Bildschirmfensters verwendet werden können.

```
from gamegrid import *

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)


makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False)
setSimulationPeriod(50)
initCars()
show()
doRun()
```

MEMO

Für Arcade-Games wird meist ein GameGrid mit einer Zellengrösse von 1 Pixel verwendet (Pixelgame). Bei einer Simulationsperiode von 50 ms wird die Spielszene 20 Mal pro Sekunde gerendert, was zu einer relativ gut fliessenden Bewegung führt. Das sporadisch auftretende Rucken ist darauf zurückzuführen, dass der Computer zu wenig Rechenleistung aufweist. Wegen der begrenzten Rechenleistung kann die Simulationsperiode auch nicht wesentlich verkleinert werden.

FROSCH MIT CURSORTASTEN BEWEGEN

Es nächstes stellst du dir die Aufgabe, den Frosch in das Spiel einzubauen. Dieser soll sich bei der Entstehung am unteren Bildrand befinden und mit den Cursor up-, down-, left und right-Tasten bewegt werden.

Da auch der Frosch ein Actor ist, schreibst du zuerst die Klasse *Frog*, die du von Actor ableitest. Ausser dem Konstruktor benötigst du keine Methoden, da der Frosch mit Tastatur-Events bewegt wird. Dazu definierst du den Callback **onKeyRepeated**, den du beim Aufruf von *makeGameGrid()* mit dem benannten Parameter *keyRepeated* registrierst. Dieser Callback wird nicht nur beim Drücken der Taste einmal, sondern auch bei gedrückt gehaltener Taste periodisch aufgerufen. Im Callback prüfst du den Tastencode und bewegst den Frosch entsprechend um 5 Schritte (Pixel) weiter.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
  def __init__(self):
      Actor.__init__(self, "sprites/frog.gif")

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)


makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
             keyRepeated = onKeyRepeated)
setSimulationPeriod(50);
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()
```

MEMO

Um Tastaturevents zu erfassen, können auch die Callbacks *keyPressed(e)* und *keyReleased(e)* registriert werden. Im Unterschied zu *keyRepeated(code)* muss der Keycode aber mit *e.getKeyCode()* aus dem Parameter *e* geholt werden. Zudem ist in diesem Spiel *keyPressed(e)* weniger geeignet, da es nach dem Drücken und Halten der Taste eine Verzögerung gibt, bist die nachfolgenden Press-Events ausgelöst werden. Wenn du die Keycodes nicht kennst, so schreibst du am besten ein kleines Testprogramm, das diese ausschreibt:

```
from gamegrid import *

def onKeyPressed(e):
    print "Pressed: ", e.getKeyCode()

def onKeyReleased(e):
    print "Released: ", e.getKeyCode()

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
  keyPressed = onKeyPressed, keyReleased = onKeyReleased)
show()
```

KOLLISIONSEVENTS

Das Vorgehen zur Detektion von Kollisionen zwischen Aktoren ist einfach: Du sagst bei der Erzeugung eines Fahrzeug *car* mit mit der Methode

```
frog.addCollisionActor(car)
```

dass der Frosch bei einer Kollision mit einem Fahrzeug einen Event auslösen soll, der die Methode **collide()** aufruft, die sich in der Klasse *Frog* befindet. Dort behandelst du den Event nach deinen Wünschen, beispielsweise lässt du den Frosch wieder an die Startposition zurück springen.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/frog.gif")
        self.setCollisionCircle(Point(0, -10), 5)

    def collide(self, actor1, actor2):
        self.setLocation(Location(400, 560))
        return 0

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        frog.addCollisionActor(car)
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
```

```
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
     keyRepeated = onKeyRepeated)
setSimulationPeriod(50)
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()
```

MEMO

Die Methode **collide()** ist kein eigentlicher Callback, sondern eine Methode der Klasse *Actor*, die in *Frog* überschrieben wird. Darum brauchst du *collide()* auch nicht mit einem benannten Parameter zu registrieren. Standardmässig wird der Kollisionsevent dann ausgelöst, wenn sich die umgebenden Rechtecke der Spritebilder überschneiden. Man kann aber die Kollisionsbereiche sowohl in Form, Grösse und Lage bezüglich das Sprite-Bildes verändern. Dazu stehen folgende Methoden der Klasse *Actor* zur Verfügung:

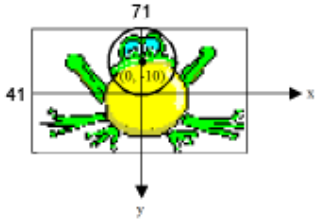
Methode	**Kollisionbereich**
setCollisionCircle(centerPoint, radius)	Kreis mit gegebenem Zentrum und Radius (in Pixel)
setCollisionImage()	Nicht-transparente Bildpixels (nur mit einem Partner der Kreis, Linie oder Punkt als Kollisionsbereich hat)
setCollisionLine(startPoint, endPoint)	Linie zwischen Start- und Endpunkt
setCollisionRectangle(center, width, height)	Rechteck mit gegebenem Zentrum und gegebener Länge und Breite
setCollisionSpot(spotPoint)	Ein Bildpixel

Alle Methoden verwenden ein relatives Pixel-Koordinatensystem mit Nullpunkt in der Mitte des Spritebildes und positiver x-Achse nach rechts und positiver y-Achse nach unten.

Das Froschbild hat eine Grösse von 71 x 41 Pixel. Setzt man daher beispielsweise im Konstruktor von Frog zusätzlich

```
self.setCollisionCircle(Point(0, -10), 5)
```

so muss ein Fahrzeug über den Kreis mit Radius 5 Pixel am Kopf des Frosches fahren, um einen Kollisionsevent auszulösen.



(Da der Kollisionsbereich aus Effizienzgründen zwischengespeichert wird, kann es nötig sein, TigerJython neu zu starten, damit sich Änderungen auswirken.)

SPIEL-SUPERVISOR UND SOUND

Bei vielen Spielen ist es nötig, dass ein "unabhängiger Spiel-Supervisor" für die Einhaltung der Spielregeln, die Verteilung der Punkte und das Ausrufen des Siegers bei Game-Over verantwortlich gemacht wird. Wie im täglichen Leben, ist es auch hier besser, diese Aufgabe nicht einer Spielfigur, sondern einem davon unabhängigen Programmteil zuzuweisen. Besonders gut geeignet ist der Hauptteil des Programms, der ja nach der Spielinitialisierung zu Ende läuft. Du fügst dazu am Ende des bestehenden Programms eine Schleife ein, die mit einer kurzen Periodendauer das Spiel überprüft und entsprechend handelt. Du solltest aber nicht eine ganz enge Schleife ohne **delay()** einbauen, da diese das Programm unnötig belastet, was zu Verzögerungen im übrigen Programmablauf führen kann.

Die Schleife sollte dann abbrechen, wenn das Game-Fenster geschlossen wird, was du mit **isDisposed = True** erreichst. Der Supervisor kann beispielsweise die Anzahl Versuche begrenzen und die Anzahl der erfolgreichen und misslungenen Strassenüberquerungen zählen und anzeigen. Die Behandlung der Situation bei *Game-Over* ist oft speziell trickreich, da an verschiedene Varianten gedacht werden muss. Oft ist es auch so, dass man das Spiel mehrmals spielen will, ohne das Programm neu zu starten. Für den Einbau von Soundeffekten kannst du deine Kenntnisse vom Kapitel Sound beiziehen. Am einfachsten verwendest du die Funktion *playTone()*.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/frog.gif")

    def collide(self, actor1, actor2):
        global nbHit
        nbHit += 1
        playTone([("c''h'a'f'", 100)])
        self.setLocation(Location(400, 560))
        return 0

    def act(self):
        global nbSuccess
        if self.getY() < 15:
            nbSuccess += 1
            playTone([("c'e'g'c''", 200)])
            self.setLocation(Location(400, 560))

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        frog.addCollisionActor(car)
        if i < 5:
            addActor(car, Location(350 * i, 90), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)
```

```
def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
    keyRepeated = onKeyRepeated)
setSimulationPeriod(50)
setTitle("Frogger")
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()

# Game supervision
maxNbLifes = 3
nbHit = 0
nbSuccess = 0
while not isDisposed():
    if nbHit + nbSuccess == maxNbLifes:  # game over
        addActor(Actor("sprites/gameover.gif"), Location(400, 285))
        removeActor(frog)
        doPause()
    setTitle("#Success: " + str(nbSuccess) + " #Hits " + str(nbHit))
    delay(100)
```

MEMO

Das Zählen der Erfolge mit **nbSuccess** und der Misserfolge mit **nbHit** erfolgt in der Klasse *Frog*. Darum müssen die Variablen dort als global deklariert werden. Man könnte auch statische oder Instanzvariablen der Klasse *Frog* verwenden. Bei Game-Over wird ein Actorbild mit einem Text eingefügt, der Frosch entfernt, der Simulationszyklus mit **doPause()** angehalten und dann die Supervisor-Schleife mit break verlassen. Man könnte auch einen *TextActor* verwenden. Damit ist es möglich, den Text zu Laufzeit anzupassen.

```
rate = nbSuccess / (nbSuccess + nbHit)
ta = TextActor("  Game Over: Success Rate = " + str(rate) + " % ",
     DARKGRAY, YELLOW, Font("Arial", Font.BOLD, 24))
addActor(ta, Location(200, 287))
```

AUFGABEN

1. Ersetze das Hintergrundbild und die Oldtimer-Bilder durch Tierbilder, die in einem Fluss schwimmen (Krokodile, usw.).

2. Führe einen Punktezählung (Score) und eine Zeitlimite für die Überquerung ein: Jede erfolgreiche Überquerung gibt 5 Punkte, jeder Hit -5 Punkte. Das Überschreiten der Zeitlimite ergibt 10 Minuspunkte und lässt den Frosch wieder an den Anfangsort zurückspringen.

3. Ergänze das Spiel nach eigenen Ideen.