7.5 SPRITE-ANIMATION

EINFÜHRUNG

In der Gamelibrary *JGameGrid* werden alle Spielfiguren aus der Klasse *Actor* abgeleitet, damit sie bereit ohne Programmieraufwand viele wichtige Eigenschaften und Fähigkeiten besitzen. Ihr spezifische Aussehen erhalten sie aber über ihr Erscheinungsbild, das als Bilddatei, auch Sprite genannt, geladen wird

Spielfiguren sind in vieler Hinsicht animiert: Sie bewegen sich über das Spielfeld und ändern dabei ih Erscheinungsbild, z.B. ihre Körperhaltung oder ihre Miene. Aus diesem Grund können einem Actor-Objek beliebig viele verschiedene Spritebilder zugeordnet werden, die über einen ganzzahligen Index, der Sprite ID, unterschieden werden. Dies ist einfacher, als Actors mit unterschiedlichen Sprites durch Klassenableitungen zu modellieren.

Spielfiguren werden in der Regel auch ihren Ort, ihre Bewegungsrichtung und ihren Rotationswinkel verändern. Dabei sollte der Rotationswinkel automatisch der Bewegungsrichtung angepasst werden. I *JGameGrid* muss aus Effizenzgründen bereits bei der Definition der Actors angegeben werden, ob dies rotierbar und welche Spritebilder zugeordnet sind. Diese werden bei der Erstellung des Actor-Objekt i einen Bildpuffer geladen, der auch die rotierten Bilder enthält. Zu Laufzeit müssen dann die Bilder wede von der Festplatte geladen noch sonstwie transformiert werden, was zu einem Performanzverlust führe würde. Standardmässig werden 60 Spritebilder für alle 6 Grad erzeugt.

In *JGameGrid* wird ein Animationskonzept angewendet, das man auch in anderen Game-Librarie insbesondere in Greenfoot [mehr...] findet.

Fundamentales Animationsprinzip:

> In der Klasse *Actor()* ist die Methode *act()* definiert, die einen **leeren** Definitionsteil hat, also sofort zurückkehrt. Die von Actor abgeleiteten benutzerdefinierten Spielfiguren überschreiben *act()* und implementieren dabei das spezifische Verhalten der Spielfigur.
>
> Beim Hinzufügen einer Spielfigur zum Spielfenster mit *addActor(),* wird dieser in eine **Act-Order-Liste** (geordnet nach Actor-Klassen) eingefügt. Eine interne **Game-Loop** (hier auch **Simulationszyklus** genannt), durchläuft periodisch diese Liste und ruft wegen der Polymorphie *act()* aller Actors der Reihe nach auf.

Damit dieses geistreiche Prinzip funktioniert, müssen sich allerdings die Actors **kooperativ** verhalten, d.h **kurz laufenden Code** aufweisen. Insbesondere wirken sich Schleifen und Delays katastrophal aus, d andere Actors dadurch auf den Aufruf ihres eigenen *act()* warten müssen.

Das Zeichnen der Spritebilder erfolgt nach folgendem Prinzip. In der Game-Loop werden die Bilder alle Actoren in der Reihenfolge der **Paint-Order-Liste** in einen Bildschirmpuffer kopiert und dieser am Ende in Spielfenster gerendert. Die Reihenfolge des Durchlaufs legt damit auch die Sichtbarkeit der Spritebilde fest: Spritebilder **später** durchlaufener Actors überdecken die anderen, liegen also sozusagen **weiter oben** Da die Actors in der Reihenfolge des Aufrufs von *addActor()* in die Paint-Order-Liste eingefügt werden liegen später hinzugefügte Sprites oberhalb der anderen. Sowohl die Act-Order-Liste wie die Paint-Order Liste lassen sich nachträglich vielseitig verändern,
insbesondere kann ein Actor mit *setOnTop()* verlangen, an den Anfang gesetzt zu werden und damit sein *act)* zuerst ausführen zu lassen und über allen anderen Actoren zu erscheinen.

Zwar können bei der Initialisierung einem Actor beliebig viele Spritebilder zugeordnet werden, aber dies können zu Laufzeit **nicht verändert** werden. Benötigt man stark ändernde Spritebilder, z.B. einen Bildtext so ist es möglich, den Actor erst zu Laufzeit als **dynamischen Actor** mit Hilfe von übliche Grafikfunktionen zu erzeugen.

PROGRAMMIERKONZEPTE:*Simulationszyklus, Kooperativer Code, Fabrik-Klasse, Statische Variable, Entkoppelung*

PFEILBOGEN BEWEGEN UND PFEILE ABSCHIESSEN

Mit einer Armbrust, die du mit der Tastatur steuerst, willst du Pfeile abschiessen, die sich auf eine natürlichen Bahn (Wurfparabel) bewegen. Später willst du diese Pfeile verwenden, um herumfliegend Früchte zu halbieren.

Du schreibst eine Klasse *Crossbow*, die du von der Klasse *Actor* ableitest. Beim Aufruf des Konstruktor der Basisklasse *Actor* sagst du mit *True*, dass es sich um einen rotierbaren Actor handelt. Der Wert 2 gil an, dass er 2 Spritebilder besitzt, nämlich eines mit einer gespannten Armbrust und aufgesetztem Pfeil un eines für die entspannte Armbrust ohne Pfeil. Die Bilddateien werden automatisch unter dem Name *sprites/crossbow_0.gif* und *sprites/crossbow_1.gif* gesucht und befinden sich in der Distribution vo *TigerJython*.

```
Actor.__init__(self, True, "sprites/crossbow.gif", 2)
```

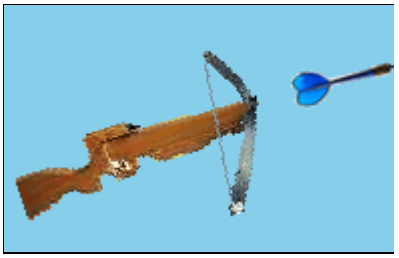
Die Armbrust wird mit Tastaturevents gesteuert: Mit Cursor-up/down veränderst du die Richtung und mit de Space-Taste schiesst du den Pfeil ab. Der Callback *keyCallback()* ist in *makeGameGrid()* als *keyPresse* registriert.

Die Pfeilklasse *Dart* ist bereits etwas komplizierter, müssen sich doch die Pfeile auf einer Wurfparabel i einem x-y-Koordinatensystem mit horizontaler x und nach unten zeigender y-Achse bewegen. Di Flugbahn wird nicht aus einer Kurvengleichung, sondern iterativ als Veränderung in der kurzen Zeit d bestimmt. Aus der Kinematik ist bekannt, dass sich dabei die neuen Geschwindigkeitskoordianten (*vx', vy* und die neuen Ortskoordinaten (*px', py'*) nach der Zeit dt wie folgt berechnen (*g = 9.81m/s^2* ist di Gravitationsbeschleunigung):

$vx' = vx$
$vy' = vy + g * dt$

$px' = px + vx * dt$
$py' = py + vy * dt$

Die Startwerte (Anfangsbedingungen) ermittelst du in der Methode reset(), die beim Hinzufügen der Dartinstanz zum Spielfeld automatisch aufgerufen wird.



In *act()* gibst du dem Pfeil den neuen Ort und die neue Richtung. Um Ressourcen zu sparen, entfernst du ihn vom Board, sobald er ausserhalb des sichtbaren Fenster ist und bringst die Armbrust wieder in Abschussstellung.

```
from gamegrid import *
import math

# ------------------- class Crossbow -----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        self.vx = self.speed * math.cos(math.radians(self.getDirection()))
        self.vy = self.speed * math.sin(math.radians(self.getDirection()))

    def act(self):
```

```
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

# ------ End of class definitions --------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(),
                                crossbow.getDirection())

screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Beim Aufruf des Konstruktors der Klasse *Actor* wird angegeben, ob der Actor rotierbar ist und ob ihr mehrere Spritebilder zugeordnet sind. [mehr...]

Die Richtung des Pfeils drehst du ständig in Richtung der Geschwindigkeit, damit ein natürliches Flugbil entsteht.

FRÜCHTEFABRIK UND BEWEGTE FRÜCHTE

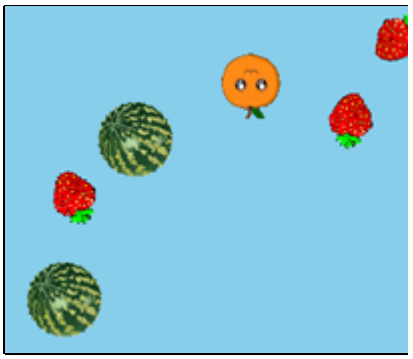
Dein Programm soll drei Sorten von Früchten verwenden: Melonen, Orangen und Erdbeeren. Die Frücht werden laufend in zufälliger Reihenfolge erzeugt und bewegen sich dann vom rechten oberen Bildrand m zufällig variierter Horizontalgeschwindigkeit auf einer Wurfparabel nach links. Die drei Früchtesorten habe viele Gemeinsamkeiten und ein paar wenige Unterschiede. Es wäre daher **kein guter Entscheid**, d Klassen *Melon*, *Orange* und *Strawberry* aus *Actor* abzuleiten, denn dann müsstest du di Gemeinsamkeiten in jeder Klasse neu implementieren, was zu der verpönten **Codeduplikation** führt. Es i angebracht, in dieser Situation eine Hilfsklasse *Fruit* zu definieren, in der die Gemeinsamkeite implementiert sind und die speziellen Früchte *Melon, Orange und Strawberry* aus *Fruit* abzuleiten.

Die Erzeugung der Früchte delegierst du einer Klasse, die man eine **Fabrik-Klasse** (**factory**) nenn

Obschon sie kein Spritebild besitzt, leitest du sie ebenfalls aus *Actor* ab, damit du *act()* verwenden kanns um neue Früchte zu erzeugen. Eine *Factory*-Klasse hat eine spezifische Eigenschaft: Obschon sie mehrer Früchte erzeugt, gibt es davon nur eine einzige Instanz [mehr...]. Es ist daher nicht üblich, den Konstrukt zu verwenden, der ja zur Erzeugung von mehreren Instanzen vorgesehen ist. Factory-Klassen besitze darum eine Methode **create()** (oder mit einem ähnlich vielsagenden Namen), die ein einziges Objekt de Klasse erstellt und es als Funktionswert zurückgibt. Jeder weitere Aufruf von *create()* liefert dann nur d bereits erzeugte Factory-Instanz [mehr...].

Da ja die Methode create() ohne eine Instanz aufgerufen wird, muss sie mit *@staticmethod* **statisch** definiert werden.

Bei der Erzeugung der *FruitFactory* wird mit der Variablen *capacity* auch noch angegeben, welches die maximale Anzahl Früchte ist, welche die Factory erzeugen kann. Zudem kann jeder Actor *setSlowDown()* aufgerufen, um die Aufrufsfrequenz von *act()* zu verlangsamen.



```
from gamegrid import *
from random import randint, random

# ---------- class Fruit -----------------------
class Fruit(Actor):
    def  __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2) # rotatable, 2 sprites
        self.vx = vx
        self.vy = 0

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt # vx = const
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            self.removeSelf()

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", vx)

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
```

```
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory -------------------
class FruitFactory(Actor):
    myFruitFactory = None
    myCapacity = 0
    nbGenerated = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
                 # slows down act() call for this actor
        return FruitFactory.myFruitFactory

    def act(self):
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            print "Factory expired"
            return

        vx = -(random() * 20 + 30)
        r = randint(0, 2)
        if r == 0:
            fruit = Melon(vx)
        elif r == 1:
            fruit = Orange(vx)
        else:
            fruit = Strawberry(vx)
        FruitFactory.nbGenerated += 1
        y = int(random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)

# ------ End of class definitions --------------------

FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
setSimulationPeriod(30)
doRun()
show()
```


MEMO

In einer statischen Methode steht der Parameter *self* nicht zur Verfügung. Daher müssen alle Variablen, di in *create()* zugewiesen werden, **statische Variablen** sein (Voranstellen des Klassennamens) [mehr...].

In einer Entwicklungsphase können gewisse Funktionen oder Methoden noch unvollständig codiert sei Man kann beispielsweise lediglich in die Konsole ausschreiben, dass sie aufgerufen wurden. Du mach davon mit print "*Factory expired*" Gebrauch.

Beim Hinzufügen eines Actors ins GameGrid mit *addActor()* wird der Bildpuffer automatisch auf de Bildschirm gerendert, damit der Actor sofort sichtbar ist. Falls der Simulationszyklus gestartet ist, wird da Rendern sowieso in jedem Zyklus ausgeführt. Darum sollte in diesem Fall besser addActorNoRefresh verwendet werden, denn zu häufiges Rendern kann zu Bildschirmflackern führen.

ZUSAMMENBAU UND KOLLISIONEN BEHANDELN

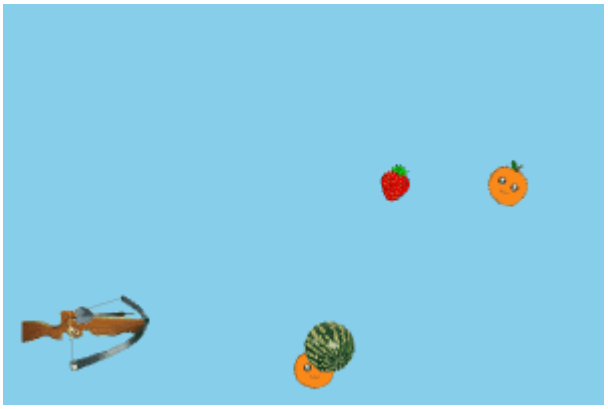
Die beiden eben geschriebenen Programmteile könnten auch von zwei Arbeitsgruppen entwickelt worde sein. Die nächste Aufgabe besteht darin, diese Teile zusammenführen, was nicht immer ganz leicht ist. I aber wie hier der **Programmstil einheitlich** und der Code weitgehend **entkoppelt**, so wird dadurch da "Mergen" des Codes wesentlich erleichtert

Als neue Funktionalität soll zudem das Halbieren der Früchte beim Zusammentreffen mit einem Pfe eingebaut werden. Wir haben dies bereits vorbereitet, da ja die Früchte zwei Spritebilder haben, eines fü die ganze und eines für die halbierte Frucht.

Wie du weisst, werden Kollisionen zwischen Actors durch einen Kollisionsevent erfasst. Dazu legst du fü jeden Actor fest, welche die möglichen Kollisionspartner sind. Überlege dir dazu das Folgende: Erzeugst d beim Abschiessen einen Pfeil, so sind alle gegenwärtig vorhandenen Früchte Kollisionspartner.

Vergiss aber nicht, dass während der Bewegung des Pfeils auch neue Früchte hinzukommen. Daru musst du beim Erzeugen einer Frucht auch alle vorhandenen Pfeile (vielleicht gibt es nur einen) a Kollisionspartner festlegen.

In *JGameGrid* kannst du addCollisionActors() eine ganze Liste von Actoren als Kollisionspartne übergeben. Mit *getActors(Klasse)* kriegst du eine Liste mit allen Actoren der angegeben Klasse, die d übergeben kannst.



```
from gamegrid import *
from random import randint, random
import math

# ---------- class Fruit ------------------------
class Fruit(Actor):
    def  __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2)
        self.vx = vx
        self.vy = 0
        self.isSliced = False

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
```

```
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            self.removeSelf()

    def sliceFruit(self):
        if not self.isSliced:
            self.isSliced = True
            self.show(1)

    def collide(self, actor1, actor2):
       actor1.sliceFruit()
       return 0

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", vx)

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory -------------------
class FruitFactory(Actor):
    myCapacity = 0
    myFruitFactory = None
    nbGenerated = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
        return FruitFactory.myFruitFactory

    def act(self):
        self.createRandomFruit()

    def createRandomFruit(self):
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            print "Factory expired"
            return

        vx = -(random() * 20 + 30)
        r = randint(0, 2)
        if r == 0:
```

```
            fruit = Melon(vx)
        elif r == 1:
            fruit = Orange(vx)
        else:
            fruit = Strawberry(vx)
        FruitFactory.nbGenerated += 1
        y = int(random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)
        # for a new fruit, the collision partners are all existing darts
        fruit.addCollisionActors(toArrayList(getActors(Dart)))

# ------------------- class Crossbow ----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        dx = math.cos(math.radians(self.getDirectionStart()))
        self.vx = self.speed * dx
        dy = math.sin(math.radians(self.getDirectionStart()))
        self.vy = self.speed * dy

    def act(self):
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

    def collide(self, actor1, actor2):
        actor2.sliceFruit()
        return 0

# ------ End of class definitions -------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(),
                                crossbow.getDirection())
        # for a new dart, the collision partners are all existing fruits
        dart.addCollisionActors(toArrayList(getActors(Fruit)))
```

```
FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Hast du mit *addCollisionActor()* oder *addCollisionActors()* die Kollisionspartner deines Actors angegeben so musst du in der Klasse des Actors die Methode *collide()* einfügen, die bei jeder Kollision automatisch aufgerufen wird. Der Rückgabewert muss ein Integer sein, der sagt, wie manchen Simulationszyklus die Kollisionen inaktiv sind (hier 0 sein). Eine Zahl grösser als 0 ist manchmal nötig, damit sich die beiden Partner wieder voneinander entfernen, bevor Kollisionen aktiv werden.

Als Kollisionsgebiete sind standardmässig die umgebenden Rechtecke des Spritebildes aktiv (Sie werden natürlich bei der Rotation der Aktoren mitgedreht). Für den Pfeil könntest du auch mit

```
setCollisionCircle(Point(20, 0), 10)
```

einen Kreis an der Pfeilspitze als Kollisionsgebiet festlegen, damit Früchte, die mit dem hinteren Teil des Pfeils kollidieren, nicht halbiert werden.

SPIELZUSTAND ANZEIGEN UND GAME-OVER BEHANDELN

Zum Dessert verfeinerst du den Code noch, indem du einen Game-Score und Benutzerinformation einbaust. Am einfachsten schreibst du sie in einer Statusbar aus.

Wie du bereits weisst, ist es günstig, im Hauptteil des Programms einen Game-Supervisor zu implementieren. Dieser soll die Anzahl der getroffenen und verpassten Früchte ausschreiben und das Spiel beenden, wenn die Früchtefabrik ihre Kapazität erreicht ist. Er zeigt also den Endstand an, erzeugt einen Game-Over-Actor und verhindert das Weiterspielen.

```
from gamegrid import *
from random import random, choice
import math

# ---------- class Fruit -----------------------
class Fruit(Actor):
    def __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2)
        self.vx = vx
        self.vy = 0
        self.isSliced = False

    def reset(self): # Called when Fruit is added to GameGrid
```

```
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            if not self.isSliced:
                FruitFactory.nbMissed += 1
            self.removeSelf()

    def sliceFruit(self):
        if not self.isSliced:
            self.isSliced = True
            self.show(1)
            FruitFactory.nbHit += 1

    def collide(self, actor1, actor2):
       actor1.sliceFruit()
       return 0

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", vx)

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory ------------------
class FruitFactory(Actor):
    myCapacity = 0
    myFruitFactory = None
    nbGenerated = 0
    nbMissed = 0
    nbHit = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
        return FruitFactory.myFruitFactory

    def act(self):
```

```
        self.createRandomFruit()

    @staticmethod
    def createRandomFruit():
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            return
        vx = -(random() * 20 + 30)
        fruitClass = choice([Melon, Orange, Strawberry])
        fruit = fruitClass(vx)
        FruitFactory.nbGenerated += 1
        y = int(random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)
        # for a new fruit, the collision partners are all existing darts
        fruit.addCollisionActors(toArrayList(getActors(Dart)))
        print type(getActors(Dart))

# ------------------- class Crossbow ----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        dx = math.cos(math.radians(self.getDirectionStart()))
        self.vx = self.speed * dx
        dy = math.sin(math.radians(self.getDirectionStart()))
        self.vy = self.speed * dy

    def act(self):
        if isGameOver:
            return
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

    def collide(self, actor1, actor2):
        actor2.sliceFruit()
        return 0

# ------ End of class definitions -------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if isGameOver:
```

```
            return
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(), crossbow.getDirection())
        # for a new dart, the collision partners are all existing fruits
        dart.addCollisionActors(toArrayList(getActors(Fruit)))


FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81
isGameOver = False

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
addStatusBar(30)
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()

while not isDisposed() and not isGameOver:
   # Don't show message if same
   oldMsg = ""
   msg = "#hit: "+str(FruitFactory.nbHit)+" #missed: "+str(FruitFactory.nbMissed)
   if  msg != oldMsg:
        setStatusText(msg)
        oldMsg = msg
   if FruitFactory.nbHit + FruitFactory.nbMissed == FACTORY_CAPACITY:
       isGameOver = True
       removeActors(Dart)
       setStatusText("You smashed " + str(FruitFactory.nbHit) + " out of "
       + str(FACTORY_CAPACITY) + " fruits")
       addActor(Actor("sprites/gameover.gif"), Location(300, 200))

   delay(100)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Bei Game-Over sollten die meisten Benutzeraktionen verboten werden. Am einfachsten ist es, ein Fla
isGameOver = *True* einzuführen, mit dem man die Aktionen durch vorzeitiges *return* in den betreffende
Funktionen und Methoden unterbindet.

Es soll möglich bleiben, auch bei Game-Over die Armbrust zu bewegen, aber verboten sein, damit z
schiessen.

AUFGABEN

1. Zähle die Anzahl Pfeile und beschränke sie auf eine sinnvolle Maximalzahl. Ist diese erreicht, soll da

Spiel ebenfalls beendet werden. Füge entsprechende Statusangaben hinzu.

2. Füge einen Punktescore für das Halbieren der Früchte hinzu:

   Melone: 5 Punkte
   Orange: 10 Punkte
   Erdbeere: 15 Punkte

3. Wir nach Game-Over die Enter-Taste gedrückt, so soll das Spiel von neuem beginnen.

4. Erweitere oder modifiziere das Spiel nach eigenen Ideen.