Heute

- Konventionen zu Namensgebung für Variablen
- Kleines Soundbeispiel zum Einstieg
- Beispiel: Auffangspiel

Konventionen zur Namensgebung für Variablen

Motivation:

- Der Datentyp ist aus dem Variablennamen ersichtlich
- Verständnis des Codes verbessern (unter dem Aspekt, das man häufig erst Tage, Wochen oder Monate später wieder „alten" Code liest, oder das andere Personen den Code lesen)
- Vermeidung / Verringerung von semantischen Fehlern
- selbst schlechte (oder nicht perfekte) Konventionen sind besser als gar keine !

Weblinks:

- Zur Sinnhaftigkeit der Camel Case-Konvention:
  http://dotnet.mvps.org/dotnet/articles/camelcase/
  Schlusswort des Artikels: „Aktuell eingesetzte Benennungskonventionen erwecken den Anschein, mehr festgeschriebene Abbilder althergebrachter und teils kruder Gewohnheiten von Entwicklern zu sein, als Ergebnisse einer wissenschaftlichen Auseinandersetzung. Dabei bietet es sich an, die Empfehlung der Konventionen argumentativ durch Hinzuziehung von Linguisten, Typographen, Wahrnehmungsforschern und Informatikern zu unterstützen. Viele bestehende Konventionen reihen sie sich nahtlos in das Bild populärer Programmiersprachen ein, deren Syntax für eine einfache und schnelle Verarbeitung durch den Compiler, weniger aber eine einfache und sichere Codierung durch den Entwickler optimiert ist. Selbst neue, unter dem Gesichtspunkt der Benutzbarkeit geschaffene Programmiersprachen und Konventionen werden wenig ändern, solange deren Verwendung und Einhaltung nicht bereits im Zuge der Ausbildung propagiert werden."
- Interessante Diskussion zum Thema im FlashForum:
  http://www.flashforum.de/forum/showthread.php?s=&threadid=78239
- Ungarische Notation (Wikipedia):
  http://de.wikipedia.org/wiki/Ungarische_Notation

Von mir verwendete Konventionen für Lingo

Syntax: [Scopepräfix][Typpräfix]{_}variablenName

Beispiel: `giTreffer  -- globale Variable, Typ: #integer`

Scopepräfix: (Scope = Gültigkeitsbereich einer Variablen)

- Globale Variablen: "g"
- Property-Variablen: "p"
- Paramter / Argumente einer Funktion (eines Handlers): "a"
- Lokale Variablen: kein Scopepräfix

Typpräfix:

Typ	Präfix	Beispiel
#integer	i	iGanzZahl = 3
#float	f	fFloatWert = 4.0
#string	s	sZeichenkette = "abc123"
#symbol	y	yDirectorSymbol = #blau
#boolean	b	bWert = true
#list	l	lFarben = ["rot", "gruen", "blau"]
#proplist	l	lFarben = ["hellrot": color(240,50,50), "gruen": color(0,255,0)]
#instance	o	oSkriptInstanz = script(„Skriptname").new()
#color	c	cButtonFarbe = color(240,50,50)
#member	m	mDarsteller = member("Darstellername")

(Fortsetzung auf nächster Seite …)

(Fortsetzung Typpräfixe:)

Typ	Präfix	Beispiel
#sprite	SP	SPRahmen = sprite(4)
#point	pnt	pntSpritePosition = point(123,456)
#vector*	v	vYAchse = vector(0,1,0)
#transform*	t	tBallTransformation = transform(fT1,fT2, ... ,fT16)
#model*	mdl	mdlBall = mSw3dWelt.model(„Ball“)
#modelressource*	res	resBallResource = mSw3dWelt.modelressource(„Ball“)
#camera*	cam	cam1 = mSw3dWelt.camera(„DefaultView“)
#group*	grp	grpWelt = mSw3dWelt.group(„World“)
#shader*	sh	shBlau = mSw3dWelt.shader(„Blau“)
#texture*	tex	texSchachbrett = mSw3dWelt.shader(„Schachbrett“)

*) Shockwave-3D

Typpräfixe können in bestimmten Fällen auch kombiniert werden - zB. um den Typ der Einträge in einer Liste anzudeuten. Nach dem Typpräfix kann optional ein Unterstrich "_" folgen.
Danach geht es mit dem Variablennamen in ungarischer Notation weiter.

noch einige Beispiele:

```
psName       -- Propertyvariable, String
pmLocalWorld -- Propertyvariable, Darsteller
lWerte       -- lokale Liste mit irgendwelchen Werten

liWerte      -- lokale Liste mit Integer-Werten
aiListIndex  -- ein Argument aus einer Parameterübergabe, Integer
gloZellen    -- globale Variable, Liste von Objekten
```

Kleines Soundbeispiel

- Spielt bei Tastendruck einen Sound ab, der dem gedrückten Buchstaben entspricht
- Balanceregler zum Einstellen der Soundkanal-Eigenschaft `sound(x).pan`
- Play / Pause – Button zum Abspielen eines Sounds

kleinesSoundBeispiel.dir (in kleinesSoundBeispiel.zip)

Beispiel: Auffangspiel


Auffangspiel.dir

- Puppet-Sprites:
  Mit `sprite(x).puppet = true` wird dem Drehbuch die Kontrolle über den Spritekanal entzogen und der Sprite ist nur über Lingo-Befehle steuerbar.

```
SP = sprite(1)
SP.puppet = true -- "verpuppen" des Sprites
SP.member = member("roter Punkt") -- Darsteller zuweisen
SP.ink = 1 -- Hintergrund transparent
SP.loc = point(50, 100) - Position setzen
```

- Verhaltensskripte können auch mit Lingo an Sprites angebracht werden (statt sie mit der Maus auf ein Sprite im Drehbuch zu ziehen). Man erzeugt eine neue Instanz des Verhaltensskriptes, und fügt diese Instanz zur Skriptinstanzliste des Sprites hinzu.

```
oVerhaltensInstanz = script("Bewegung").new(SP)
SP.scriptInstanceList.add(oVerhaltensInstanz)
```

- Nicht nur eins, sondern (fast) beliebig viele Sprites werden erzeugt, wenn man die „Sprite-Verpuppung“ in einer repeat-Schleife mit mehreren Sprites durchführt. Diese Schleife kann in einen Handler in einem Filmskript eingebettet werden. Das ergibt eine globale Funktion, die mit Lingo-Anweisungen eine gewisse Anzahl Sprites erzeugt:

```
on ErzeugeSprites aiAnzahl

  repeat with i = 1 to aiAnzahl
    SP = sprite(i)
    SP.puppet = true
    SP.member = ... -- usw.
    ...
  end repeat

end
```

- An jedes der auf diese Weise erzeugten Sprites wird genau eine Instanz des Verhaltensskriptes „Bewegung“ angebracht. Jede Skriptinstanz steuert dann das Sprite, an dem es angebracht ist. Damit man von einem Skript eine Instanz mit Lingo erzeugen kann, muss das Skript einen speziellen Handler besitzen, den new-Handler. Von diesem Handler muss die Referenz auf die Skripinstanz (me) zurückgegeben werden:

```
on new me
  -- Verschiedene Initalisierungsanweisungen
  return me
end
```

- Handler, die in Verhalten stehen, die an Sprites angebracht sind, können auf verschiedene Arten aufgerufen werden:

  über das Sprite
  ```
  sprite(2).StarteBewegung()
  ```

  über die Skriptinstanz selbst
  ```
  oVerhaltensInstanz.StarteBewegung()
  ```

  durch den sendsprite-Befehl
  ```
  sendsprite( 2, #StarteBewegung )
  ```

  Diese unterschieden sich vor allem in der Fehlerbehandlung.
  Wenn beispielsweise das Sprite die Nachricht doch nicht behandlen kann, weil an ihm es kein Verhalten angebracht wurde, das über einen passenden Handler verfügt, wird bei der ersten Aufrufvariante ein Fehler erzeugt, bei der dritten hingegen nicht.

- In dem Verhaltensskript „Bewegung“ befinden sich folgende Handler, die für gewisse Aktionen verantwortlich sind:

Handler	Aktion
`new`	Erzeugung einer Skriptinstanz, Initialisierung von Propertyvariablen
`StarteBewegung`	stepFrame-Nachrichten erhalten
`StoppeBewegung`	stepFrame-Nachrichten nicht mehr erhalten
`mouseWithin`	Auffangen des Sprites
`mouseLeave`	wieder-los-lassen des Sprites
`stepFrame`	Position des Sprites berechnen und aktualisieren

- Im stepFrame-Handler wird in jedem Frame die neue Position des Sprites berechnet. Das geschieht, indem zwei Vektoren miteinander addiert werden und der resultierende Vektor auf die aktuelle Spriteposition addiert wird.

  die beiden Vektoren sind

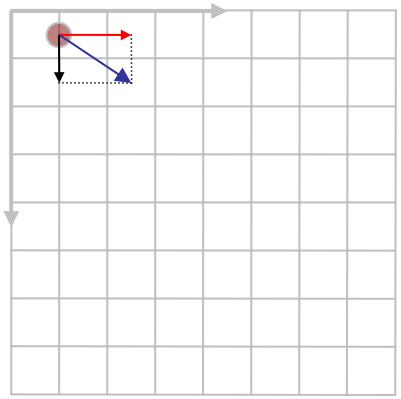
`gpntGravitation`	Gravitation, in einer globalen Variable, Typ `#point`
`ppntGeschwindigkeit`	Geschwindigkeit, in einer Property-Variable, Typ `#point`

AA3-10

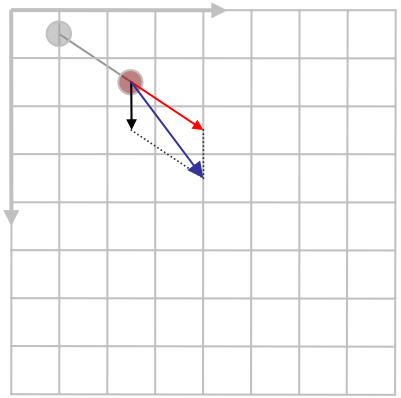
Gravitation
Geschwindigkeit
Gravitation + Geschwindigkeit = neue Geschwindigkeit

```
pntNeueGeschwindigkeit = ppntGeschwindigkeit + gpntGravitation
pSprite.loc = pSprite.loc + pntNeueGeschwindigkeit
ppntGeschwindigkeit = pntNeueGeschwindigkeit
```

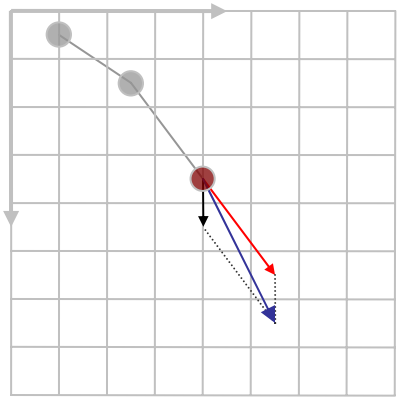
Frame 1



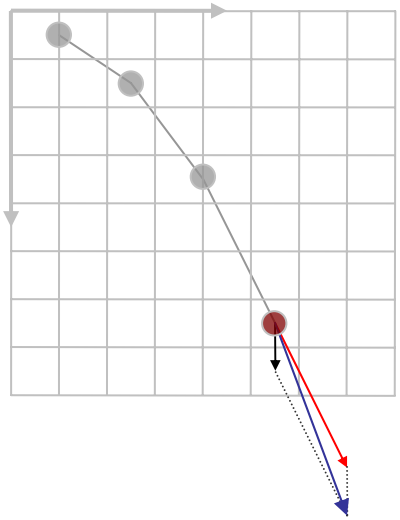
Frame 2



Frame 3

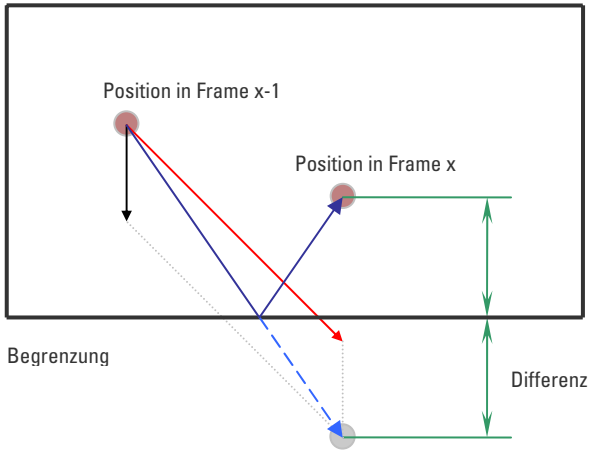


Frame 4



- Nach der Positionsberechnung findet die Reflektionsberechnung statt.

  Dazu wird in jedem neuen Frame für alle 4 Seiten des Begrenzungsrechtecks überprüft, ob sich das Sprite innerhalb oder außerhalb des Rechtecks befindet. Falls es sich außerhalb befindet, hätte es auf dem Weg zu seiner aktuellen Position eigentlich schon reflektiert werden müssen. Da hier im Beispiel nur an achsenparallelen Geraden reflektiert wird, ist die Berechnung der reflektierten Position aber einfach über die Differenz aus Begrenzung und Position bezüglich der betrachteten Achse möglich.




```
if pSprite.locV < prBegrenzung.top then
  ppntGeschwindigkeit.locV = -1 * ppntGeschwindigkeit.locV
  diff = prBegrenzung.top - pSprite.locV
  pSprite.locV = prBegrenzung.top + diff
end if

if pSprite.locV > prBegrenzung.bottom then
  ppntGeschwindigkeit.locV = -1 * ppntGeschwindigkeit.locV
  diff = pSprite.locV - prBegrenzung.bottom
  pSprite.locV = prBegrenzung.bottom - diff
end if

-- wenn man diff gleich einsetzt passt die Berechnung auf eine Zeile
if pSprite.locH > prBegrenzung.right then
  ppntGeschwindigkeit.locH = -1 * ppntGeschwindigkeit.locH
  pSprite.locH = 2 * prBegrenzung.right - pSprite.locH
end if

if pSprite.locH < prBegrenzung.left then
  ppntGeschwindigkeit.locH = -1 * ppntGeschwindigkeit.locH
  pSprite.locH = 2 * prBegrenzung.left - pSprite.locH
end if
```