Algorithmenanimation

The Path-transition Paradigm

John T.Stasko

Einführung

- Path-Transition Paradigma:
  Allgemeiner Lösungsansatz, um grafische Objekte in einem Raum mit Hilfe eines Pfades zu animieren. Versuch der Formalisierung mit Hilfe von vier abstrakten Datentypen

- Zweck:
  Schaffung einer theoretischen Grundlage, kein Algorithmenanimationssystem

- Vorgehensweise:
  - Objekt-basierte Annäherung an das Problem
  - formale Modelle und Semantik

Übersicht

- Ziele des Path-Transition Paradigmas
- syntaktische Beschreibung der 4 abstrakten Datentypen
  - Location
  - Image
  - Path
  - Transition
- semantische Beschreibung der Datentypen
- Beispiel: Anwendungsgebiet Algorithmenanimation

Ziele des Paradigmas

- Datenkapselung
  Einbettung der Animation in Prozeduren
  Anwendung kontrolliert das Format der Animation über Parameter

- Benutzerangepaßte Animation
  - flüssige Bildfolgen
  - vom Benutzer regulierbare Geschwindigkeit
    (Schnelldurchlauf oder Einzelschritte)

- leicht erlernbar
  Aufwand für Design und Implementierung der Animation soll minimal sein

Definition

Eine Animation ist eine Menge von Grafikobjekten, auf der eine Menge von Modifikationen vorgenommen wird (wie z.B. Verschiebung, Änderung der Farbe, Skalierung,...)

Hilfsmittel:
Vier abstrakte Datentypen:
Location, Image, Path, Transition, die auf $\mathbb{R}^2$ operieren (läßt sich aber auch auf andere Räume erweitern)
(Vernachlässigung von Problemen wie Darstellung von reellen Zahlen, Größenbeschränkung der Darstellung (Bildschirm))

Location

Location beschreibt einfach eine Position im Koordinatensystem; diese wird durch ein (x,y)-Koordinatenpaar dargestellt

private Daten: Location lo

- **x:real, y:real**

Methoden des Datentyps Location:

- **Location Create(real *x*, real *y*):**
  Kreiert eine neue Location aus dem Koordinatenpaar *(x,y)*
- **Location Modify(Location *loc*, real *x*, real *y*):**
  Ändert den Wert einer gegebenen Location *loc* **in** *(x,y)*
- **boolean Equal(Location *loc1*, Location *loc2*):**
  Test auf Gleichheit
- **real X(Location *loc*):**
  Liefert x-Koordinate von *loc*
- **real Y(Location *loc*):**
  Liefert y-Koordinate von *loc*

Image

Formal : Image ist 6-Tupel (t,o,v,P,R,d)

- t: Typ des Bildes
- o: Ursprung
- v: Sichtbarkeit
- P: lokale Parameter
- R: definierte Transitionen
- d: Zeichenmethoden

Jeder Imagetyp muß definiert haben, wie die Transitionen in R die Parameter verändern,z.B.:

resize beim Kreis: Änderung des Radius

resize beim Rechteck: Änderung der Länge und Breite

primary images:
Kreise, Ellipsen, Linien,etc.
Aber: Keine Einschränkung auf spezielle Grundmenge von Bildern innerhalb des Paradigmas

composite images:
Menge von primary Images

$\rightarrow$ bildet Vereinigung von primary Images (wiederverwendbar)

$\rightarrow$ ist Liste von primary Images (Positionen lokal zum Ursprung des composite Images (formale Darstellung ???)

Bsp: Ampel

private Daten: Image im

- **type:ImageType**
- **pos:Location**
- **vis:Boolean**
- **$\{$local parameters$\}$**

Methoden des Datentyps Image:

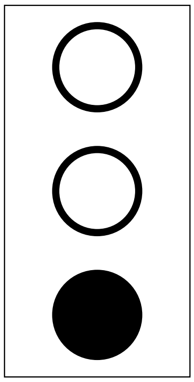
- **Image Create(imagetype *it*, boolean *vis*, Location *loc*, local parameters):**

  Kreiert ein neues Image

- **Location Locate(Image *im*, ImagePart *ip*):**

  ip beschreibt eine Kompaßposition relativ zu einem minimalen umschließenden Rechteck um das Bild (z.B. South, NorthEast, Center,...)

rectangle(0,0,3,6,black,0.0)
circle(1.5,1,0.5,red,0.2)
circle(1.5,3,0.5,yellow,0.2)
circle(1.5,4,0.5,green,1.0)



Path

Pfade beschreiben Wege in einem zweidimensionalen Koordinatensystem

Pfade dienen als Parameter, um Attribute eines Bildes zu ändern

natürliche Betrachtung: Änderung von Bildposition (move)
aber auch Farbe, Größe, Füllung, etc.

Formal: endliche, geordnete Folge von Lokationen

Bei move: (x,y)-Werte relativ zur vorhergehenden Position

Pfadlänge: Anzahl der (x,y)-Paare

$\rightarrow$ Kontrolle der Flüssigkeit der Bewegung in der Animation durch relativen Abstand der Koordinatenpaare

private Daten: Path pa

- **offsets:list<Location>**

Methoden des Datentyps Path:

- **Path Create(list<Location> *loc*):**

  Kreiert einen neuen Pfad

- **integer Length(Path *p*):**

  Liefert die Länge von *p*

- **Path Null(integer *length*):**

  Liefert einen Pfad der Länge *length* mit (0.0,0.0)-Einträgen

- **Path Rotate(Path *p*, integer *angle*):**

  Liefert einen Pfad, der eine Drehung von *p* um *angle* Grad beschreibt

- **Path Interpolate(Path *p*, real *num*):**

  Liefert eine Kopie von *p*, die sich genauso verhält wie *p*, aber *num* Einträge enthält

- **Path Distance(Location *loc1*, Location *loc2*, real *length*):**

  Liefert einen geraden Pfad der Länge *length* von *loc1* nach *loc2*

- **Path Example(Location** *loc1*, **Location** *loc2*, **Path** *p***):**

  Liefert einen Pfad von *loc1* **nach** *loc2*, **der so ähnlich aussieht wie** *p*

- **Path Concatenate(Path** *p1*, **Path** *p2***):**

  Liefert Konkatenierung von *p1* **und** *p2*

- **Path Iterate(Path** *p*, **integer** *times***):**

  Liefert p^{times}

- **Path Compose(Path** *p1*, **Path** *p2***):**

  Liefert $p1 + p2$ **komponentenweise**

  (wie Vektoraddition)

⋮

Transition

liefert eine Animation unter Benutzung eines Pfades, um die Attribute der Bilder zu ändern

simple transition: 3-Tupel (t,i,p)

t: Typ der Transition (z.B. move, resize, fill, raise,...)

i: Bild

p: Pfad (Länge $\hat{=}$ #frames dieser Animation)

$\rightarrow$ wie bei images: t nicht auf diese Typen beschränkt

verschiedene Transitionstypen benutzen den Pfad auf verschiedene Weise:

- move: 2D-Pfad
- visibility: nur Länge, um Sichtbarkeit ein/auszuschalten
- fill: benutzt x-Koordinate, um Füllgrad festzulegen ($\triangle$x)

transition unit: 4-Tupel (t,i,x,y)
atomare Einheit

transition units:
einzelne Aktionen in einer Animation

$$(t, i, [(x_1, y_1), \ldots, (x_n, y_n)]) \longrightarrow \\ [(t, i, x_1, y_1), \ldots (t, i, x_n, y_n)]$$

$\rightarrow$ simple transition:
Liste von transition units

transition moments:
zeitgleiche Ausführung von transition units innerhalb einer Transition

private Daten:

TransUnit tu
    type:TransType
    Image:Image
    x:real, y:real

Transition tr
    moment:list<list<TransUnit>>

Methoden des Datentyps Transition:

- **Transition Create(transitiontype *tt*, Image *im*, Path *p*):**

  Liefert eine neue Transition

- **Transition Concatenate(Transition *tr1*, Transition *tr2*):**

  Wie bei Path

- **Transition Iterate (Transition *tr*, integer *times*):**

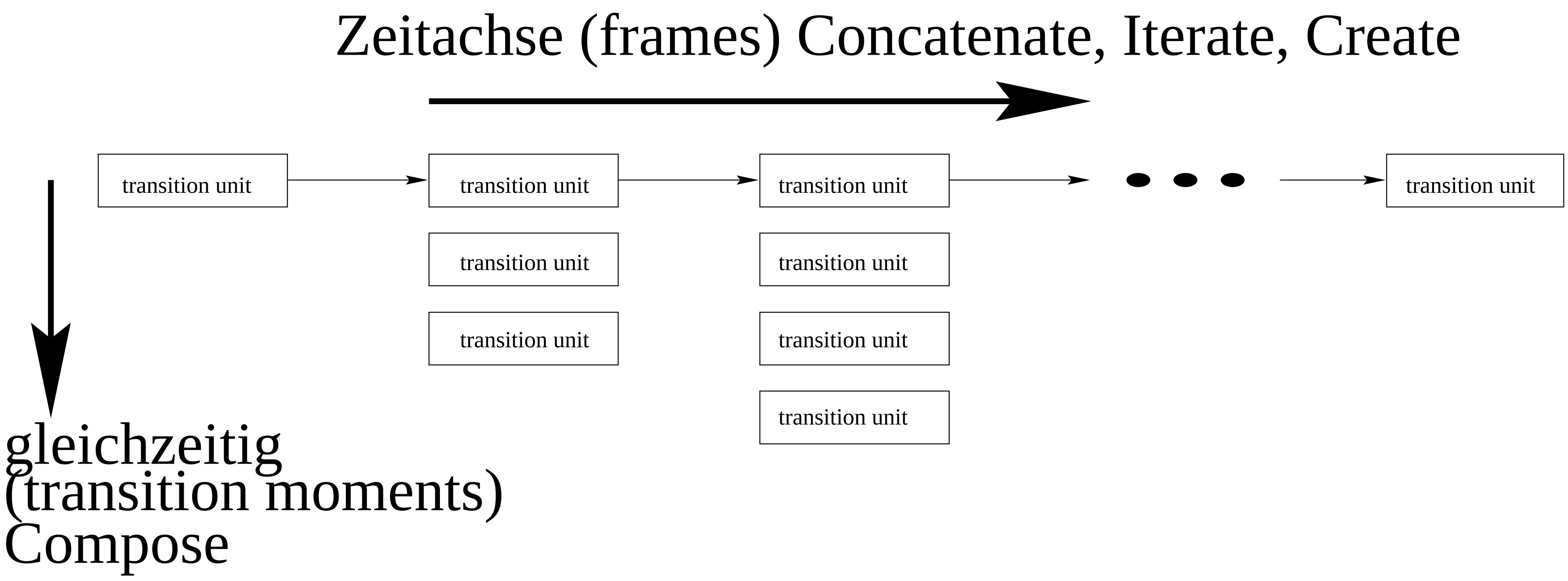
  Wie bei Path

- **Transition Compose(Transition *tr1*, Transition *tr2*):**

  Liefert eine Transition, die aus der simultanen Ausführung von *tr1* und *tr2* hervorgeht ($\rightarrow$ moment)

- **list<list<Image>> Perform(Transition *tr*):**

  Wandelt die Transition in eine Animation um


Zeitachse (frames) Concatenate, Iterate, Create
transition unit
transition unit
transition unit
transition unit
transition unit
transition unit
transition unit
transition unit
transition unit
transition unit
gleichzeitig
(transition moments)
Compose

Semantische Beschreibung

animation state: S: 4-Tupel (L,I,P,T)

L: Menge von Locations
I: Menge von Images
P: Menge von Paths
T: Menge von Transitions

modelliert eine Animation
reicht nicht aus, da Bilder andere überdecken können
$\rightarrow$ configuration von Images C:
geordnete Liste aller aktiven Bilder in C
i_1 überdeckt $i_2 \Rightarrow i_1$ vor i_2 in C

Def.: Animation (innerhalb des Paradigmas):
geordnete Liste von Konfigurationen
($\rightarrow$ Liste von Liste von Bildern)

Um zu beschreiben, welche Bilder man später bei der Animation sehen kann, muß man noch 2 Dinge festlegen:

- Ordnung der Bilder in einem transition moment
- Ordnung der Bilder in der Konfiguration

Bilder werden in einem Moment nur durch Compose zusammengefügt
$\rightarrow$ Compose(t1,t2) $\Rightarrow$ t1 wird vor t2 bearbeitet
Ordnung innerhalb der Konfiguration wird nur durch CreateImage, Raise, Lower verändert:

- Create: Bild wird vorne eingefügt
- Raise: Bild wird nach vorne verschoben
- Lower: Bild wird nach hinten verschoben

Transition

```
im:=ImageCreate(it,vis,loc,{params});
pa:=PathCreate(list<lo>)

TansCreate(ttype,im,pa)→
   if (pa) then
    cons(cons((ttype,im,LocX(car(pa)),
                 LocY(car(pa))),∅),
        TransCreate(ttype,im,cdr(pa)))
   else
    ∅
```

2 Hilfsfunktionen tcopy und mcopy:

```
tr:=TransCreate(ttype,im,pa);
mom:=list<tu>
```

```
tcopy(tr)→
    if (tr) then
      cons(mcopy(car(tr)),tcopy(cdr(tr)))
    else
      ∅

mcopy(tr)→
    if (mom) then
      cons(car(mom).tu,car(mom).im,
              car(mom).x,car(mom).y),
           mcopy(cdr(mom)))
    else
      ∅
```

TransConcatenate(tr_1, tr_2)→
    append(tcopy(tr_1),tcopy(tr_2))

```
TransIterate(tr,l)→
    if (l>0) then
      append(tcopy(tr),TransIterate(tr,l-1))
    else
      ∅
```

```
TransCompose(tr_1, tr_2)→
    if (tr_1 and tr_2) then
      cons(append(mcopy(car(tr_1)),
                  mcopy(car(tr_2))),
           TransCompose(cdr(tr_1),cdr(tr_2)))
    else if (tr_1) then
      tcopy(tr_1)
    else if (tr_2) then
      tcopy(tr_2)
    else
      ∅

TransPerform(tr)→
    if (tr) then
      moment=car(tr)
      while (moment) do
        trans_unit=car(moment)
        apply(trans_unit,C)
        moment=cdr(moment)
      cons(C,TransPerform(cdr(tr)))
    else
      ∅
```

apply((move,im,$x^{'}$,$y^{'}$),C)$\Rightarrow$
im:[circle,(loc.x+$x^{'}$,loc.y+$y^{'}$,vis,
radius,color,fill]$\in$I

apply((resize,im,$x^{'}$,$y^{'}$),C)$\Rightarrow$
im:[circle,(loc,vis,radius+$x^{'}$,color,fill]$\in$I

apply((raise,im,$x^{'}$,$y^{'}$),C)$\Rightarrow$
C-im; cons(im,C)

Jetzt erhalten wir unsere Animationsliste durch

AnimList = append(AnimList,Perform(tr))

(Am Anfang ist AnimList=$\emptyset$)

Anwendungsgebiet Algorithmenanimation

Def:

- Algorithmenanimation ist der Prozeß des Abstrahierens von Daten, Operationen und Semantik von Computerprogrammen, um danach dynamische grafische Objekte von diesen Abstraktionen zu bilden

- Algorithmenanimation ist verständlicher als simple Übersetzungen von Programmdaten in grafische Objekte

- Algorithmenanimation kann eine Sichtweise liefern, die keinen direkten Zusammenhang mit den eigentlichen Programmdaten hat

→ Implementierung eines Algorithmen-animationssystems: Tango

existierende Algorithmen werden mit interesting events angereichert

Beispiel: Türme von Hanoi

benutzen Assoziationen, die Verbindungen zwischen Programmobjekten und Animationsobjekten herstellt

(Make, Store, Retrieve)

Twist-Makros: dienen hauptsächlich dazu, den Assoziationen zuzuweisen

Assoziationen:

PEG	Speichert die Bilder der Stäbe
PEGTOP	Speichert zu gegebenem Stab die Location über dem Stab
POINTS	Speichert zu jeder möglichen Scheibenposition die entsprechende Location
DISK	Speichert die Bilder der Scheiben
OUTLINE	Speichert die Bilder der Umrandungen
DISK_AT	Speichert zu jeder Location die Scheibe, die sich dort befindet
OUTLINE_AT	Speichert zu jeder Location die Umrandung, die sich dort befindet

```
#include <stdio.h>
#include "xtango.h"

#define  PEG  0   /* ids used in associations */
#define  DISK  1
#define  OUTLINE 2
#define  POINTS  3
#define  PEGTOP  4

#define  DISK_HT   0.1

/* draw the 3 pegs, create a two-dimensional array of positions */
/* for the disks to go onto the pegs, and draw the initial set  */
/* of disks on peg 0                                             */

void
ANIMInit(disks)
   int disks;
{

   /* Initialisierung der assoziativen Arrays /*

   path = TANGOpath_null(3);
   trans = TANGOtrans_create(TANGO_TRANS_TYPE_DELAY,NULL,path);
   TANGOtrans_perform(trans);
   TANGOpath_free(1,path);
   TANGOtrans_free(1,trans);
}

/* given there are fdisks on peg fpeg, move the top disk           */
/* onto peg tpeg which currently has tdisk number of disks on it.  */
```

```
void
ANIMMove(fpeg,fdisk,tpeg,tdisk)
   int fpeg,fdisk,tpeg,tdisk;
{
   TANGO_LOC  frompt,topt,topeg_top,frompeg_top;
   TANGO_IMAGE  disk,rect;
   TANGO_PATH  path[3],movepath;
   TANGO_TRANS  move[2],compose;

   /* Berechnung der neuen Positionen /*

   path[0] = TANGOpath_distance(frompt,frompeg_top,0.03);
   path[1] = TANGOpath_distance(frompeg_top,topeg_top,0.03);
   path[2] = TANGOpath_distance(topeg_top,topt,0.03);
   movepath = TANGOpath_concatenate(3,path[0],path[1],path[2]);

   move[0] = TANGOtrans_create(TANGO_TRANS_TYPE_MOVE,disk,movepath);
   move[1] = TANGOtrans_create(TANGO_TRANS_TYPE_MOVE,rect,movepath);
   compose = TANGOtrans_compose(2,move[0],move[1]);
   TANGOtrans_perform(compose);

   /* Abspeichern der neuen Positionen */

   TANGOpath_free(4,path[0],path[1],path[2],movepath);
   TANGOtrans_free(3,move[0],move[1],compose);
}
```