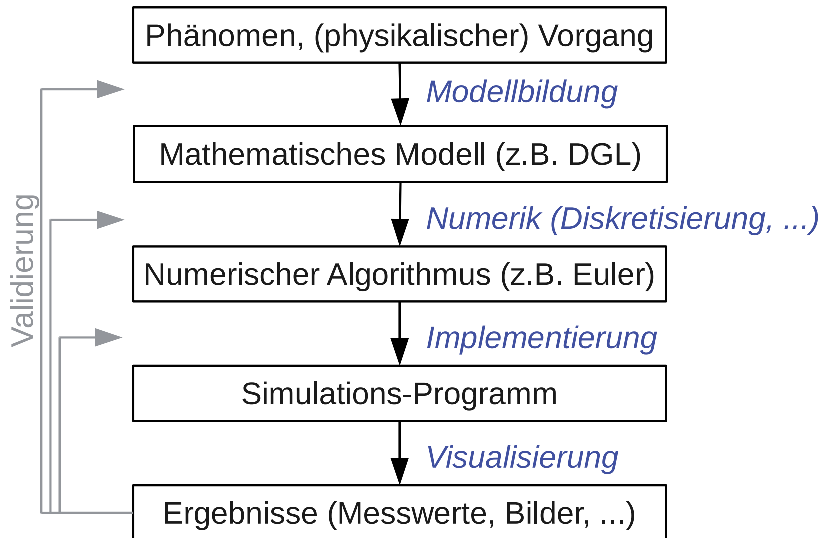
Teil X

Simulation von Partikelsystemen

Simulationspipeline



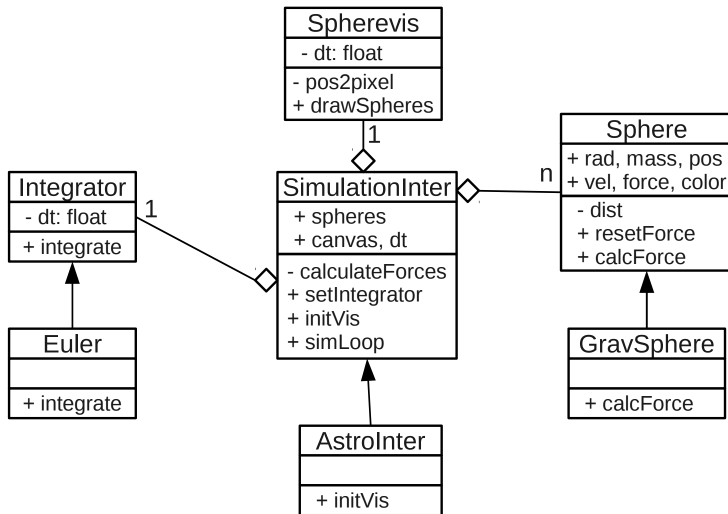
Simulation des Sonnensystems

- Phänomen: Planeten und Sterne ziehen sich an
- Modell: Gravitationspotential/kraft für jedes Partikelpaar
- Numerik: Diskretisierung der DGL
- Simulationsprogramm
  - Datenstrukturen für die Objekte
  - Berechnung der Kraft
  - Lösung der Bewegungsgleichung
  - Kopplung zur Visualisierung
- Visualisierung mit Tkinter

Anmerkungen

- Eine Vorlesung für ein komplettes Programm ist wenig
- Das Programm wird funktional, aber weder effizient noch an allen Stellen sehr sauber
- In den Code-Stücken sind kaum Kommentare, diese werden mündlich in der Vorlesung gegeben. Bei eigenen Programmen gehören die Kommentare in den Code!

UML-Diagram des Astro-Simulationsprogramms



Integrator

- Basisklasse `Integrator` als Schnittstellendefinition
- Unterklasse `Euler`: Einfacher Integrator
- Für jedes Objekt Lösung der Bewegungsgleichung in beiden Raumrichtungen

```
class Integrator(object):
  def __init__(self, dt):
    self.dt = dt
  def integrate(self, sphereList):
    pass

class Euler(Integrator):
  def integrate(self, sphereList):
    for s in sphereList:
      for d in range(2):
        s.pos[d] = s.pos[d] + self.dt*s.vel[d]
        s.vel[d] = s.vel[d] + self.dt*s.force[d]/s.mass
```

Klasse zur Visualisierung: `Spherevis`

Konstruktor `__init__`

- Parameter für den Konstruktor: `canvas`, `posLL` und `posRR`
- `canvas` enthält die Zeichenfläche
- `posLL` und `posUR` enthalten Koordinaten der Ecken des Simulationsgebiets
- Konstruktor speichert übergebenes `canvas`, dessen Größe und die Koordinaten

```
class Spherevis(object):
  def __init__(self, canvas, posLL, posUR):
    self.canvas = canvas
    self.width = canvas.winfo_reqwidth()
    self.height = canvas.winfo_reqheight()
    self.posLL = posLL
    self.posUR = posUR
```

Methode pos2Pixel

- Zu zeichnende Objekte sollen in Koordinaten des Simulationsgebiets angegeben werden
- Diese müssen in Pixel umgerechnet werden
- Dazu wird das Intervall des Simulationsgebiets auf das Intervall zwischen Null und der Anzahl an Pixeln abgebildet
- `pos2Pixel` erhält als Parameter die Simulationskoordinaten und gibt die Pixelkoordinaten zurück

```
def pos2Pixel(self, posx, posy):
  pix_x = self.width*((posx - self.posLL[0])/
                      (self.posUR[0]-self.posLL[0]))
  pix_y = self.height*((posy - self.posLL[1])/
                       (self.posUR[1]-self.posLL[1]))
  return pix_x, pix_y
```

Methode `drawSpheres`

- `drawSpheres` erhält als Parameter eine Liste von Kugeln (Klasse `Sphere`)
- `drawSpheres` löscht die Zeichenfläche und malt alle Kugeln mit deren Radius und Farbe
- Für jede Kugel wird die „Bounding Box"in Pixeln ermittelt, dann mit `create_oval` der ensprechende Kreis gezeichnet
- Dann wird die Zeichefläche aktualisiert

```
def drawSpheres(self, spheres):
  self.canvas.delete("all")
  for s in spheres:
    pixelLL = self.pos2Pixel(s.pos[0]-s.rad,
                             s.pos[1]-s.rad)
    pixelUR = self.pos2Pixel(s.pos[0]+s.rad,
                             s.pos[1]+s.rad)
    self.canvas.create_oval(pixelLL[0], pixelLL[1],
                            pixelUR[0], pixelUR[1],
                            fill=s.color)
  self.canvas.update()
```

Klassen für Kugeln: `Sphere` und `GravSphere`

Konstruktor `__init__`

- Parameter: Position[3], Geschwindigkeit[3], Radius, Masse und Farbe
- Alle Parameter werden gespeichert

Methode `resetForce`

- Setzt die Kraft (`force`) auf Null

Methode `dist`

- Parameter: zweite Kugel
- Gibt den Abstand (Vektor) zwischen den Kugeln zurück

Methode `calcForce`

- Parameter: zweite Kugel
- Berechnet die Kraft, die die zweite Kugel auf `self` ausübt und addiert sie auf `self.force`
- In der Basisklasse `Sphere` nur als „Interface"
- Konkrete Implementierung in Subklassen

Implementierung der Klasse `Sphere`

```python
from math import sqrt
class Sphere(object):
  def __init__(self, pos, vel, rad, mass, color="red"):
    self.rad = rad
    self.mass = mass
    self.pos = list(pos)
    self.vel = list(vel)
    self.force = [0,0]
    self.color = color
  def resetForce(self):
    for d in range(2):
      self.force[d] = 0
  def dist(self, sphere2):
    dist = [0,0]
    for d in range(2):
      dist[d] = sphere2.pos[d] - self.pos[d]
    return dist
  def calcForce(self, sphere2):
    pass
```

Subklasse GravSphere

- Gravitationskraft 1D: $F(d) = G \cdot \frac{m_1 m_2}{d^2}$
- G: Gravitationskonstante ($6.67428 \cdot 10^{-11}$)
- Gravitationskraft: $\vec{F}(\vec{r}) = G \cdot \frac{m_1 m_2}{d^3} \cdot \vec{r}$
- Dabei: $\vec{r}$: Vektor zwischen beiden Kugeln, d: Abstand
- `GravSphere` übernimmt Konstruktor, `resetForce` und `dist` von der Basisklasse
- `calcForce` wird überschrieben

```
class GravSphere(Sphere):
  gravConst = 6.67428e-11
  def calcForce(self, sphere2):
    dist = self.dist(sphere2)
    for d in range(2):
      self.force[d] += (dist[d] * self.gravConst *
                        (self.mass * sphere2.mass)
                        /sqrt(dist[0]**2 + dist[1]**2)**3)
```

Schnittstellenklasse zum Steuern der Simulation

- Eine Klasse, die die bisherigen Teile zusammenführt (`SimulationInter`)
- Dinge, die getan werden müssen:
  - Initialisierung (Erstellung von Kugeln und Rändern)
  - Erstellung des Integrators
  - Berechnung der Kräfte zwischen den Partikeln
  - Anwendung der Randbedingungen
  - Aufruf des Integrators
  - Erstellung der Zeichenfläche
  - Methode, mit der in einer Schleife Zeichnen, Kraftberechnung, Anwendung der Randbedingungen und der Integrator wiederholt aufgerufen werden
- Die Klasse implementiert noch keine konkrete Simulation
- Dazu wird ein Subklassen von `SimulationInter` erstellt: `GravInter`

Nötige Module

```
from spheres import GravSphere, LJSphere
from boundary import Boundary
from spherevis import Spherevis
from Tkinter import Tk, Canvas
from math import sqrt, pi
from random import random
```

Konstruktor `__init__(self, dt)`

- Legt listen für Kugeln und Ränder an (`spheres[]` und `boundaries[]`)
- Speichert Zeitschrittweite `dt`
- Erzeugt ein Canvas zur Visualisierung

Methode `setIntegrator`

- Speichert den übergebenen Integrator ab

Methode `calculateForces`

- Setzt zunächst die Kraft auf alle Kugeln auf Null
- Läuft dann über alle Kugelpaare und berechnet die Kraft

Methode `applyBoundaries`

- Prüft für jede Kugel, ob sie den Rand berührt
- Falls ja, wird der Effekt des Aufpralls berechnet (Position und Geschwindigkeit der Kugel angepasst)

Methode `initVis`

- Initialisiert die Visualisierung
- Wird durch Subklasse implementiert, da nur da die Simulationskoordinaten nicht bekannt sind

Methode `simLoop`

- Führt die Simulation endlos durch
- In jeden Schritt: Visualisierung, Kraftberechnung, Integration und Anwendung der Randbedingungen

Implementierung der Klasse `SimulationInter`

```
class SimulationInter(object):
  def __init__(self, dt):
    self.boundaries = []
    self.spheres = []
    self.dt = dt
    root = Tk()
    self.canvas = Canvas(root, width=800, height=800)
    self.canvas.pack()

  def setIntegrator(self, integrator):
    self.integrator = integrator

  def calculateForces(self):
    for sphere1 in self.spheres:
      sphere1.resetForce()
    for sphere1 in self.spheres:
      for sphere2 in self.spheres:
        if sphere1 != sphere2:
          sphere1.calcForce(sphere2)
```

Implementierung der Klasse `SimulationInter` (cont.)

```
def applyBoundaries(self):
  for b in self.boundaries:
    for p in self.spheres:
      b.boundaryAction(p)

def initVis(self):
  print "Subclasses have to implement initVis"
  exit(1)

def simLoop(self):
  while True:
    self.vis.drawSpheres(self.spheres)
    for i in range(10):
      self.calculateForces()
      self.integrator.integrate(self.spheres)
      self.applyBoundaries()
```

Subklasse `AstroInter`

Konstruktor `__init__`

- Konstruktor der Vaterklasse aufrufen (Zeitschrittweite 1/10 Tag)
- Planeten erzeugen

Methode `initVis`

- Größe des sichtbaren Raumausschnitts festlegen
- Instanz der Klasse `Spherevis` erzeugen

```
class AstroInter(SimulationInter):
  def __init__(self):
    SimulationInter.__init__(self, 8640.)
    c=3.
    self.spheres.append(GravSphere(( 0.00000, 0.),
      (0., 0.), c*20.e9, 1.9891e30, "yellow")) # Sonne
    ...
  def initVis(self):
    self.vis = Spherevis(self.canvas, (-2.e12,-2.e12),
                                      ( 2.e12, 2.e12))
```

Restliche Planeten

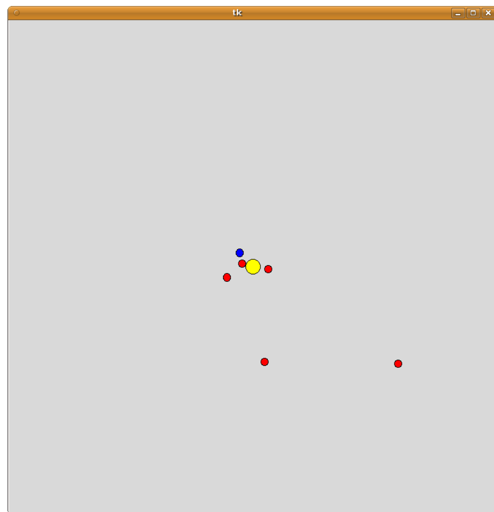
```
self.spheres.append(GravSphere((  57.9e9, 0.),
  (0.,47872.), c*10.e9, 3.302e23)) # Merkur
self.spheres.append(GravSphere(( 108.2e9, 0.),
  (0.,35021.), c*10.e9, 4.8685e24)) # Venus
self.spheres.append(GravSphere(( 149.6e9, 0.),
  (0.,29786.), c*10.e9, 5.9736e24, "blue")) # Erde
self.spheres.append(GravSphere(( 227.9e9, 0.),
  (0.,24131.), c*10.e9, 6.4185e23)) # Mars
self.spheres.append(GravSphere(( 778.3e9, 0.),
  (0.,13070.), c*10.e9, 1.899e27)) # Jupiter
self.spheres.append(GravSphere((1.429e12, 0.),
  (0., 9672.), c*10.e9, 5.6846e26)) # Saturn
self.spheres.append(GravSphere((2.875e12, 0.),
  (0., 6835.), c*10.e9, 8.6832e25)) # Uranus
self.spheres.append(GravSphere((4.504e12, 0.),
  (0., 5478.), c*10.e9, 1.0243e26)) # Neptun
```

Starten der Simulation

- Alle nötigen Klassen wurden implementiert
- Nun müssen nur noch die konkreten Instanzen erzeugt werden

```
>>> # nötige Module einbinden
>>> from integrator import Euler
>>> from simulationinterface import AstroInter
>>>
>>> # Interface für Astro-Simulation erstellen
>>> simInter = AstroInter()
>>> # Integrator mit Zeitschrittweite für Astro-Sim.
>>> simInter.setIntegrator(Euler(simInter.dt))
>>> # Init Vis. (Abbildung Sim-Koord. auf Canvas)
>>> simInter.initVis()
>>> # Simulation starten
>>> simInter.simLoop()
```

Screenshot der Astro-Simulation



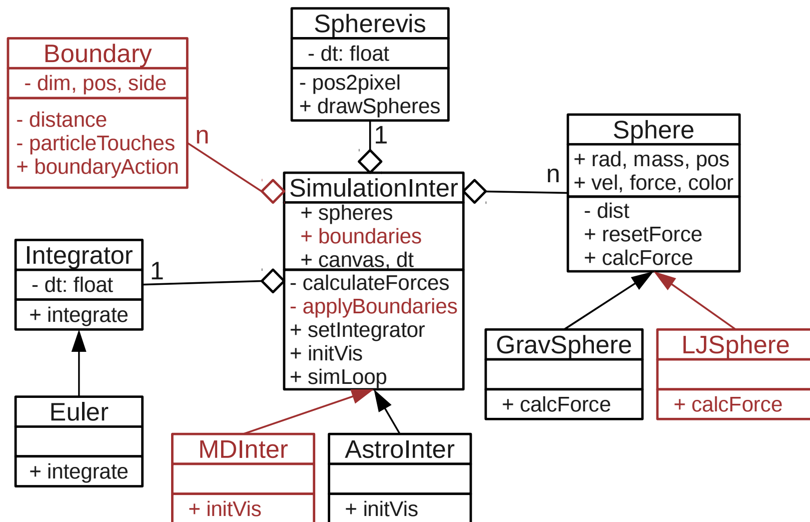
Simulation von Molekülen

- Phänomen: Moleküle stoßen sich ab bzw. ziehen sich an
- Modell: Lennard-Jones-Potential (Kombination aus Pauli-Repulsion und van-der-Waals-Anziehung)
- Numerik: Gleich wie bei der Astro-Simulation
- Simulationsprogramm
  - Im wesentlichen wie bei der Astro-Simulation
  - Wir erweitern das bisherige Programm so, dass damit sowohl Planeten als auch Moleküle simuliert werden können
  - Zusätzlich muss das Gebiet begrenzt werden
- Visualisierung wieder mit Tkinter

Anmerkungen

- Kosten eines Zeitschritts: $O(N * N)$, daher nur wenige Moleküle möglich
- Aufgrund der kurzreichweitigen Kraft (LJ-Potential fällt schnell ab) sollten nur Nachbarn von Molekülen zur Kraftberechnung verwendet werden ($O(N)$)

Erweiterung auf MD-Simulation



Subklasse `LJSphere` der Klasse `Sphere`

- Methode `calcForce` wird überschrieben, der Rest von der Basisklass übernommen
- Lennard-Jones-Potential: $U(r_{ij}) = 4\epsilon((\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6})$
- Zur Vereinfachung: $\epsilon = \sigma = 1$
- Kraft 1D: $F(r_{ij}) = -24(2 \cdot (\frac{1}{r})^{13} - (\frac{1}{r})^{7})$
- Kraft nD: $\vec{F}(\vec{r}_{ij}) = -24(2 \cdot (\frac{1}{d})^{14} - (\frac{1}{d})^{8}) \cdot \vec{r}_{ij}$

```
class LJSphere(Sphere):
  def calcForce(self, sphere2):
    dist = self.dist(sphere2)
    absdist = sqrt(dist[0]**2 + dist[1]**2)
    for d in range(2):
      self.force[d] += 24*(2*absdist**-14
                          - absdist**-8) * dist[d]
```

Randbedingungen

- Bisher ist das Simulationsgebiet prinzipiell unendlich groß
- Die Moleküle würden in alle Richtungen davon fliegen
- Deswegen wird das Gebiet begrenzt
- das Verhalten am Rand des Gebiets muss definiert werden:
  - Aus- bzw. einströmende Ränder
  - Periodische Ränder (was links rausfliegt kommt rechts wieder rein
  - Reflektierende Ränder (Einfach, deswegen nehmen wir diese)
  - Heizende/Kühlende Ränder
  - ...

Reflektierende Ränder

- Sobald ein Partikel den Rand berührt, prallt es ab
- Die Geschwindigkeit senkrecht zur Wand wird invertiert
- Die Strecke, die schon überlappt, wird neuer Abstand zur Wand

Klasse `boundary`

Konstruktor `__init__`

- Parameter dim: gibt die Raumrichtung an (0: x, 1: y)
- Parameter pos: gibt die Position der Ebene an
- Parameter side: -1 für linke/untere Seite, +1 für rechts/oben

Methode `distance`

- Parameter p: Partikel
- Berechnet den Abstand des Partikels zum Rand

Methode `particleTouches`

- Parameter p: Partikel
- Gibt `True` zurück, wenn das Partikel den Rand berührt (Abstand kleiner als Radius)

Methode `boundaryAction`

- Parameter p: Partikel
- Führt bei Berührung die Auswirkung des Aufpralls aus

Implementierung der Klasse `boundary`

```
class Boundary(object):
  def __init__(self, dim, pos, side):
    self.dim = dim # 0 oder 1
    self.pos = pos # beliebiger float-Wert
    self.side = side # -1 oder 1

  def distance(self, p):
    return p.pos[self.dim] - self.pos

  def particleTouches(self, p):
    return abs(self.distance(p)) <= p.rad

  def boundaryAction(self, p):
    if self.particleTouches(p):
      p.pos[self.dim] -= 2*(self.distance(p)
                           +p.rad*self.side)
      p.vel[self.dim] *= -1
```

Subklasse `MDInter`

Konstruktor `__init__`

- Parameter: Anzahl Partikel und Dichte
- Konstruktor der Vaterklasse aufrufen (Zeitschrittweite 0.02)
- Moleküle erzeugen
  - Zufällige Positionen schlecht, da sich Moleküle dann evtl. überlappen (hohen Kräften und damit numerische Problemen) $\Rightarrow$ Positionen auf einem Gitter.
  - Anzahl Partikel so runden, dass ein quadratisches Gitter erzeugt werden kann
  - Geschwindigkeiten prop. zur Temperatur. Wir versuchen nicht, genau zu sein, und wählen deswegen willkürlich gleichverteilte Geschwindigkeiten zwischen -0.25 und 0.25
- Randbedingungen erzeugen (refl. Ränder auf allen Seiten)

Methode `initVis`

- Raum um das Molekülgitter als sichtbaren Bereich festlegen

```
class MDInter(SimulationInter):
  def __init__(self, numParticles, density):
    SimulationInter.__init__(self, 0.02)
    numPerDim = round(sqrt(numParticles))
    self.length = sqrt(pi*numPerDim**2/density)
    for ix in range(int(numPerDim)):
      for iy in range(int(numPerDim)):
        x = (ix+0.5)/numPerDim*self.length
        y = (iy+0.5)/numPerDim*self.length
        vx = (random()-0.5)/2.
        vy = (random()-0.5)/2.
        self.spheres.append(LJSphere((x, y), (vx, vy),
                                      1.0, 1.0, None))
    self.boundaries.append(Boundary(0, 0.0, -1))
    self.boundaries.append(Boundary(0, self.length, 1))
    self.boundaries.append(Boundary(1, 0.0, -1))
    self.boundaries.append(Boundary(1, self.length, 1))
  def initVis(self):
    self.vis = Spherevis(self.canvas, (0.0, 0.0),
                         (self.length, self.length))
```

Einbinden der neuen Klassen

nötige Module

```
from optparse import OptionParser
from integrator import Euler
from simulationinterface import AstroInter, MDInter
```

Option-Parser

```
# Optionen festlegen
p = OptionParser()
p.add_option("-t", "--simType", dest="simType",
             help="'Astro' or 'MD' simulation")
p.add_option("-n", type="int", dest="numParts",
             help="Number of particles", default=30)
p.add_option("--rho", type="float", dest="rho",
             help="Density", default=0.1)
(options, args) = p.parse_args()
```

Simulationsinterface erstellen

```
# Simlationsinterface erstellen
if options.simType == None:
  print "Bitte␣Simulationstyp␣angeben"
  exit(1)
if options.simType == "Astro":
  simInter = AstroInter()
elif options.simType == "MD":
  n = options.numParts
  rho = options.rho
  simInter = MDInter(n, rho)
```

Screenshot der MD-Simulation