Dr. Ole Klein, Dr. Steffen Müthing
IWR, Universität Heidelberg

Abgabe 02. Februar 2018

Allgemeine Hinweise:

- Wir werden kommende Woche die Anmeldung zur Klausur im Müsli freischalten. Bitte tragen Sie sich dort ein, falls Sie die Klausur mitschreiben möchten. Sie haben dafür bis zum 12.2.18 Zeit (eine Woche vor der Klausur), danach ist der Menüpunkt gesperrt und weder eine Annoch Abmeldung möglich.
- Wir werden ebenfalls kommende Woche die eingetroffenen Mails bezüglich fehlender Punkte abarbeiten, so dass Sie rechtzeitig sicher gehen können, dass Sie zur Klausur zugelassen sind.
- Dies ist das letzte reguläre Aufgabenblatt. Falls Sie noch Punkte benötigen sollten, gibt es einige Bonusaufgaben — vergessen Sie in diesem Fall das Votieren nicht! Wir werden noch ein weiteres kleines Aufgabenblatt veröffentlichen, dass auf die Struktur der Klausur einstimmt und nicht mehr votiert wird.

Übung 1 *Simulation von Mehrkörperproblemen*

In unserem Sonnensystem bewegt sich eine Vielzahl von Himmelskörpern unter gegenseitiger Wechselwirkung, der Gravitation. Der Versuch, die Bahnbewegung der Planeten (oder anderer Objekte) vorherzusagen, führt mathematisch auf ein Mehrkörperproblem: mehrere punktförmige Massen, deren Beschleunigung jeweils von der Position der anderen Punktmassen abhängt.

Während das Zweikörperproblem (z.B. Erde — Sonne) noch analytisch lösbar ist, tritt ab drei Körpern chaotisches Verhalten auf, und die Bahnen lassen sich nur noch näherungsweise mit numerischen Methoden berechnen. In dieser Aufgabe wollen wir uns damit näher beschäftigen.

Hinweis: Sie können auf jede der Teilaufgaben jeweils einzeln votieren.

Hinweis: Für die Darstellung der Simulation wird die SDL-Bibliothek [1] benötigt. Installationsanweisungen für diese Bibliothek finden Sie, sofern nötig, im zugehörigen Wiki [2]. Wenn Sie die von uns verteilte virtuelle Maschine verwenden, können Sie die Bibliothek und die Entwicklungspakete mit folgendem Befehl installieren:

```
[ipk@localhost ~]$ sudo dnf install SDL2-devel
```

Sie können CMake verwenden, um die SDL-Bibliothek automatisch in Ihr Projekt einzubinden. Linken Sie dazu Ihr Programm mit der SDL-Bibliothek, indem Sie folgende Zeile in `CMakeLists.txt` einfügen:

```
target_link_libraries(<Programmname> SDL2)
```

Alternativ können Sie auch von Hand kompilieren:

```
[ipk@localhost ~]$ g++ -lSDL2 -o <Programmname> <Quelldateien>
```

[1] `https://www.libsdl.org/`
[2] `https://wiki.libsdl.org/Installation`

- **Vorbereitung und Visualisierung:** Schreiben Sie zunächst ein `struct Point`, wie wir es schon öfters in Aufgaben hatten, und ein `struct Body`, das einen Massepunkt repräsentiert. Ein solcher Massepunkt soll zwei `Point` haben, einen für die Position und einen für die Geschwindigkeit, ein `double` für die Masse, und zusätzlich ein Array aus drei `int`s, die dem Punkt eine Farbe aus dem RGB-Farbraum zuordnen.

  Auf der Homepage finden Sie den Header `sdlwrapper.hh`, der eine Klasse `SDLCanvas` zur Verfügung stellt. Machen Sie sich mit der Funktionalität dieser Klasse vertraut, binden Sie sie in Ihr Programm ein, und erzeugen Sie einen Canvas. Erzeugen Sie einige Massenpunkte mit Positionen, Geschwindigkeiten, Massen, und zugeordneten Farben. Unten finden Sie dafür zwei Beispiele. Benutzen Sie die bereitgestellte Klasse, um die Punkte zu visualisieren. Schreiben Sie dazu eine Funktion `displayBodies()`, die auf den Datentyp der Massepunkte templatisiert ist und `Canvas` und Massepunkte als Argumente bekommt. Nutzen Sie darin die Methoden `drawPixel()` und `display()` des `Canvas`-Objekts . Damit sich das Fenster nicht sofort schließt, müssen Sie *nach* dem Aufruf ihrer Funktion eine `while`-Schleife einfügen, die abwartet, bis die Escape-Taste gedrückt wurde (rufen Sie die Methode `windowClosed` auf, um herauszufinden, ob Escape gedrückt wurde).

  Hinweise:

  - Sie müssen hier wieder kontinuierliche Koordinaten in Pixelpositionen umrechnen, wie wir das bereits in früheren Aufgaben hatten. Wenn Sie ein Fenster wählen, das mindestens die Größe 1024×768 hat, können Sie die Koordinaten der Beispiele unten auch direkt verwenden, ohne umskalieren zu müssen. Sie müssen dann lediglich den Punkt $(0, 0)$ in die Bildmitte verschieben. Achten Sie außerdem darauf, dass der Ursprung links oben ist: die x-Achse geht also wie gewohnt nach rechts, die y-Achse zeigt aber nach unten.
  - Hier zwei Beispiele von Massenpunkten, die Sie zum testen verwenden können. Das erste erzeugt ein Planetensystem mit (näherungsweise) geschlossenen Bahnen:

```
std::vector<Body> bodies = {
  {{   0.,  0.}, {   0., 0.}, 1e3, {255,  0,  0}},
  {{ 100.,  0.}, {   0.,0.3}, 10., {  0,255,  0}},
  {{-200.,  0.}, {   0.,0.2}, 10., {  0,  0,255}},
  {{   0.,250.}, {-0.25, 0.}, 10., {255,255,  0}},
};
```

    Das zweite Beispiel erzeugt vier Massenpunkte, die aufgrund der Symmetrie für alle Zeiten umeinander kreisen sollten. Warum geht das nach einiger Zeit schief?

```
std::vector<Body> bodies = {
  {{  150.,    0.}, {  0.,-0.2}, 1e3, {255,  0,  0}},
  {{ -150.,    0.}, {  0., 0.2}, 1e3, {  0,255,  0}},
  {{    0.,  150.}, { 0.2,  0.}, 1e3, {  0,  0,255}},
  {{    0., -150.}, {-0.2,  0.}, 1e3, {255,255,  0}},
};
```

- **Numerische Simulation und Schwerpunktsystem:** Schreiben Sie eine Funktion

```
template<typename Force, typename Body>
void eulerStep (const Force& force, std::vector<Body>& bodies,
                double t, double dt)
```

  die die Bewegung der Massepunkte im Zeitintervall [t,t+dt] mit dem Euler-Verfahren simuliert. Nehmen Sie dazu an, dass die Klasse `Force` eine Methode

```
template<typename Body>
Point accel (const std::vector<Body>& bodies, int i, double t)
```

  besitzt, die die Beschleunigung a_i des i-ten Massepunktes liefert. Führen Sie ein Update der Po-

sitionen x_i und Geschwindigkeiten v_i der Massepunkte gemäß der folgenden Formeln durch:

$$x_i(t+\delta t) = x_i(t) + v_i(t)\cdot\delta t, \quad v_i(t+\delta t) = v_i(t) + a_i(t)\cdot\delta t$$

Je nachdem, wie die Verteilung der Anfangsgeschwindigkeiten und der Massen gewählt ist, driftet die Ansammlung aus Masseteilchen langsam aus dem Canvas. Das liegt daran, dass der Schwerpunkt

$$\bar{x}(t) = M^{-1}\cdot\sum_i m_i x_i(t), \quad M = \sum_i m_i$$

sich mit konstanter Geschwindigkeit fortbewegt. Verschieben Sie die Position der Teilchen so, dass der Schwerpunkt nach jedem Schritt in der Mitte des Canvas liegt (sie rechnen dann im Schwerpunktsystem).

- **Kräfte und Interaktion:** Schreiben Sie nun eine Klasse `Gravitation`, die das vom Template-Parameter `Force` benötigte Interface erfüllt, sprich die erwartete Methode zur Verfügung stellt. Die Kraft, die das j-te Teilchen auf das i-te ausübt, ist

  $$F_{ij} = Gm_i m_j \|x_j - x_i\|^{-3}(x_j - x_i),$$

  und daher ist die verursachte Beschleunigung

  $$a_{ij} = Gm_j \|x_j - x_i\|^{-3}(x_j - x_i).$$

  Dabei ist G die Gravitationskonstante, deren Wert von Ihnen angepasst werden kann. Beachten Sie, dass die Masse m_i jeweils auf sich selbst keine Wirkung hat, und daher bei der Berechnung ausgelassen werden muss.

  Schreiben Sie eine **`while`**-Schleife, die abwechselnd die neuen Positionen und Geschwindigkeiten berechnet und die Positionen der Massenpunkte graphisch darstellt. Experimentieren Sie dazu mit der Gravitationskonstante G, der Schrittweite δt und der Größenordnung der Anfangsgeschwindigkeiten. Für die oben angegebenen Beispiele verwenden Sie $G = 0,01$ und $\delta t = 1$. Finden Sie eine weitere Konstellation aus einem sehr schweren Zentralgestirn und drei bis vier weiteren Objekten, die ein stabiles Sonnensystem darstellen.

  Hinweis: Wenn Sie nach jeder Iteration die Methode `clear` aufrufen, wird stets nur die aktuelle Position der Punkte dargestellt. Falls Sie den Aufruf weglassen, bleiben bisher gezeichnete Pixel mit ihrer Farbe erhalten, so dass die Trajektorien der Massenpunkte sichtbar werden.

- **Weitere Möglichkeiten für Wechselwirkungen (Bonus-Votierpunkt):** Neben der hier angesprochenen Gravitationswechselwirkung können Sie Ihren Programmcode ohne große Änderungen für viele weitere Simulationen verwenden. Dazu müssen Sie lediglich den `Force`-Templateparameter austauschen. Probieren Sie die folgenden weiteren Kräfte aus (für die Beschleunigung müssen Sie jeweils durch m_i teilen):
  - Die Schwerkraft $F_g = m_i \cdot g$. Die Massepunkte sind hier keine Planeten, sondern kleine Partikel oder Regentropfen. Setzen Sie Punkte, die unten aus dem Canvas fallen, in diesem Fall mit Geschwindigkeit Null am oberen Rand wieder ein.
  - Die elastische Kraft $F = -k\cdot\delta x_i$. Der Term δx_i ist dabei die Distanz von x_i zu einem Referenzpunkt. Falls dieser konstant ist, schwingen die Punkte unabhängig voneinander um ihren Punkt. Man kann aber auch einige der Punkte (oder alle) als Referenzpunkte voneinander festlegen, was zu gekoppelten Schwingungen führt.
  - Die elektromagnetische Kraft $F_{\text{em}} = m_i\cdot(E + v_i \times B)$. Hier übernimmt m_i die Rolle der Ladung statt Masse, und kann auch negativ gewählt werden. Das elektrische Feld E und das magnetische Feld B sind vorgegebene Vektorfelder. Damit die Bewegung im Zweidimensionalen stattfindet, sollte E in der Bildebene und B senkrecht dazu liegen. Diese Kraft können Sie auch mit der obigen Kraft F_{ij} kombinieren — sie steht dann für die elektrodynamische Wechselwirkung zwischen den Punkten, statt für die Gravitation.

- **Fortgeschrittenes numerisches Verfahren (Bonus-Votierpunkt):** Das obige Verfahren ist relativ leicht zu implementieren, hat aber in der Praxis gravierende Nachteile. Daher wird für Mehrkörpersimulationen in der Praxis typischerweise ein anderes Verfahren implementiert, das klassische Runge-Kutta-Verfahren (auch RK4 genannt). Informieren Sie sich in der Wikipedia über dieses Verfahren [3] und implementieren Sie eine Funktion `rk4Step`, die statt des Euler-Verfahrens dieses Verfahren verwendet.

(Grundlagen + Fortgeschritten)

[3] `https://de.wikipedia.org/wiki/Klassisches_Runge-Kutta-Verfahren`