Programmieraufgabe 2

Physikalisch-basierte Simulation und Animation

Dr.-Ing. Johannes S. Mueller-Roemer und Stephanie Ferreira, MSc

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 2024
Programmieraufgabe 2

Einleitung

Die Implementation der Programmieraufgaben erfolgt mit C++ mit OpenGL, Eigen und Qt. Um den Anfang zu erleichtern wird eine einfache Basisapplikation zur Verfügung gestellt (Moodle).

Eine Woche nach Vorstellung der jeweiligen Programmieraufgabe wird es eine Übung (ohne Übungsblatt) geben in der Fragen zu der Programmieraufgabe gestellt werden können. Drei Wochen nach Vorstellung der Programmieraufgabe findet die Abgabe mit Testat statt. Bei der Abgabe werden Fragen zu der Programmieraufgabe gestellt. Alle Gruppenmitglieder müssen die Fragen zu der Implementation beantworten können.

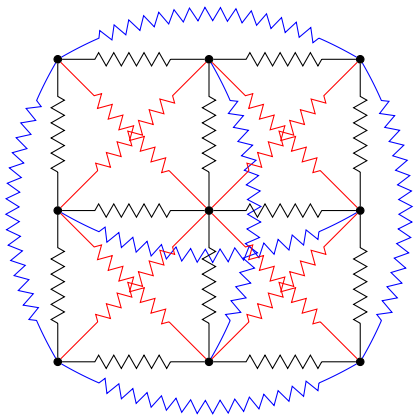
Anmerkung zur Bewertung

Durch das Lösen der unten genannten Aufgaben sind bis zu 13 Punkte erreichbar, jedoch werden je Programmieraufgabe maximal 10 Punkte gewertet. Durch das Bearbeiten aller Aufgaben können somit Fehler ausgeglichen werden.

Aufgabe 2.1: Masse-Feder-Systeme – Simulation von Textilien

Wie in der Vorlesung beschrieben, sind Masse-Feder-Systeme gut geeignet um Textilien in vereinfachter Form zu simulieren. Dabei werden Federn für Strukturkräfte, Scherkräfte und Biegekräfte genutzt. Die Funktion der Federn hängt ausschließlich davon ab wie sie topologisch am Gitter angebracht sind.

Hier sind Strukturfedern in Schwarz, Scherfedern in Rot und Biegefedern in Blau abgebildet:



Dabei wird die Orientierung des Gitters entsprechend der Webrichtung gewählt, womit verschiedene Stoffe nachgebildet werden können. Die Topologie des zur Darstellung genutzten Dreickesnetzes kann sich unterscheiden und ist oft feiner aufgelöst.

Allgemein gilt für ein solches System

$$\ddot{\mathbf{x}} = \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}), \tag{1}$$

wobei die Massematrix $\mathbf{M}$ eine diagonale Matrix mit den Diagonaleinträgen $(m_1, m_1, m_1, m_2, m_2, m_2, \ldots, m_n, m_n, m_n)$ und m_i die Massen der n Knotenpunkte sind. Wenn die Massen alle identisch sind, kann die Matrix durch die skalare Größe m ersetzt werden.

$\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}})$ ergibt sich aus der Summe der Federkräfte, der Dämpfungskräfte, sowie der Einwirkung von externen Kräften wie der Schwerkraft $f_g = mg$. Jede Feder zwischen zwei Knoten erzeugt dabei eine Kraft

$$\begin{aligned} f &= k_s(\|\mathbf{x}_{12}\| - l_0) + k_d \mathbf{v}_{12} \cdot \hat{\mathbf{x}}_{12} \\ \mathbf{f}_1 &= f\hat{\mathbf{x}}_{12} \\ \mathbf{f}_2 &= f\hat{\mathbf{x}}_{21} = -\mathbf{f}_1, \end{aligned} \tag{2}$$

wobei $\mathbf{v}_{12} = \mathbf{v}_2 - \mathbf{v}_1 = \dot{\mathbf{x}}_2 - \dot{\mathbf{x}}_1$ und $\hat{\mathbf{x}}_{12} = \frac{\mathbf{x}_{12}}{\|\mathbf{x}_{12}\|}$, sowie l_0 die undeformierte Länge der Feder, k_s die Federkonstante und k_d die Dämpfungskonstante sind.

2.1a) Midpoint-Integration (5 Punkte)

Implementieren Sie eine Textil-Simulation auf Masse-Feder-Basis mit dem expliziten Midpoint-Integrationsschema (auch bekannt als Runge-Kutta-Verfahren 2. Ordnung / RK2):

$$\mathbf{q}_{t+1} = \mathbf{q}_t + \delta_t \mathbf{f}\left(\mathbf{q}_t + \frac{\delta_t}{2}\mathbf{f}(\mathbf{q}_t)\right) \tag{3}$$

Nutzen Sie dabei ein Gitter der Auflösung 33×33. Zeitschritt sowie Feder- und Dämpfungskonstanten (getrennt für alle 3 Federkategorien) sollen über die GUI frei wählbar sein. Die Masse der Knoten beträgt jeweils 30 g, der Abstand zwischen den Knoten beträgt 10 cm.

Es sollen zwei mögliche Randbedingungen wählbar sein:

1. Ein Eckknoten ist fixiert.
2. Die Knoten an zwei (nicht diagonal) gegenüber liegenden Ecken sind fixiert mit einem Abstand von 3,2 m (der Breite des Stoffs).

In beiden Fällen wirkt eine Gravitationsbeschleunigung von $g = 10\,\mathrm{m\,s^{-2}}$, das Stoffstück ist zu Beginn undeformiert, alle Geschwindigkeiten sind anfangs $0\,\mathrm{m\,s^{-1}}$, und das Stoffstück liegt parallel zum „Boden“ im Raum.

2.1b) Implizite-Integration (5 Punkte)

Wiederholen Sie a) mit impliziter Euler-Integration. Da das Gleichungssystem

$$\mathbf{q}_{t+1} = \mathbf{q}_t + \delta_t \mathbf{f}(\mathbf{q}_{t+1}) \tag{4}$$

nichtlinear ist, muss $\mathbf{f}$ mittels einer Taylor-Entwicklung erster Ordnung approximiert werden.

2.1c) Echtzeit-Simulation (3 Punkte)

Implementieren Sie eine Variante von a) bei der das Verhältnis zwischen Simulations- und Realzeit konstant und unabhängig von der Render-Geschwindigkeit ist. D. h. der Zeitschritt soll weiterhin konstant, frei wählbar und unabhängig von der Darstellungsgeschwindigkeit sein. Die Bildwiederholrate darf auch nicht als konstant angenommen werden. Ein Echtzeit-Faktor gibt das Verhältnis zwischen den beiden Zeitsystemen an.

Hinweis: Zur Sicherheit sollte eine Maximalanzahl an Zeitschritten pro Darstellung implementiert werden. Zur Zeitmessung soll `QElapsedTimer::nsecsElapsed` benutzt werden, `QTimer` und ähnliche Event-basierte Timer sollen nicht genutzt werden.

Hinweise zur Implementation

Wie bei der ersten Programmierübung dient das SimulationFramework als Basis für die Implementation. Wie bei Programmierübung 1 Aufgabe 1.3, müssen Sie die dargestellte Kugel durch ein Dreiecksnetz ersetzen und die Positionen der Eckpunkte des Dreicksnetzes aktualisieren. Dazu müssen Sie:

- Die Erzeugung des Netzes (VAO), inklusive Positionen (`GL_ARRAY_BUFFER`) und Topologie (`GL_ELEMENT_ARRAY_BUFFER`) in `ExampleRenderer::ExampleRenderer` anpassen.
- Da die Positionen (Vertices) regelmäßig angepasst werden sollen, ist bei dem zugehörigen `glBufferData` Aufruf der Verwendungszweck von `GL_STATIC_DRAW` auf `GL_DYNAMIC_DRAW` abzuändern. Das Aktualisieren an sich wird mittels `glBufferSubData` durchgeführt.
- Der verwendete Shader `glUseProgram` ist anzupassen. Für den Vertex-Shader kann `skybox.vert` als Grundlage dienen. Beachten Sie dabei die Bedeutung der vierten Koordinate.
- Deaktivieren Sie `GL_CULL_FACE`, da das Mesh einseitig ist.
- Nutzen Sie den Kommandozeilen-Flag `--debug-gl` um im Debugger Informationen über etwaige OpenGL-Fehler zu bekommen.