Johannes Gutenberg-Universität Mainz
Institut für Informatik
Einführung in die Softwareentwicklung
Prof. Dr. Michael Wand
Dr. Christian Ali Mehmeti-Göpel
Visual Computing Group

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Einführung in die Softwareentwicklung

ÜBUNGSBLATT 05: Vektorgraphik Teil 3 – Funktionale Programmierung

9. Juni 2025

Über dieses Übungsblatt

Auf diesem Übungsblatt programmieren wir weiter an dem vektororientierten Zeichenprogramm, das wir auf Blatt 03 begonnen haben. Der Fokus liegt diesmal darauf, funktionale Entwurfsprinzipien umzusetzen.

Es ist zu empfehlen, Ihre Lösung von Übungsblatt 04 als Basis zu nutzen, und weiter zu überarbeiten. Auch dieses Mal steht wieder eine Beispiellösung des vorherigen Übungsblattes in LMS bereit. Es ist zulässig, darauf aufzubauen. Aus didaktischen Gründen ist es aber ratsam, den eigenen Code weiterzuentwickeln.

Noch ein didaktischer Hinweis: Wir werden einige der hier angesprochenen Techniken in den kommenden Vorlesungen noch genauer kennenlernen und vertiefen. Die Übung dient (wie immer) dazu eigene Erfahrungen mit den Problemen zu sammeln. Die prinzipiellen Ideen sind in der Aufgabe ausreichend erläutert, um direkt mit der Bearbeitung beginnen zu können.

Abgabe: (Abgabe 11. SW)

Auch für dieses Übungsblatt sind wieder zwei Wochen Bearbeitungszeit vorgesehen. Das Übungsblatt muss entsprechend bis zum **22. Juni 2025, 23:59** (mittags) in LMS abgegeben werden. Laden Sie dazu den Quellcode aller Aufgaben hoch. Die Ergebnisse müssen in den Übungen zwischen dem 23.-27. Juni 2025 vorgestellt werden.

Aufgabe 2: Datenflussarchitekturen für Vektorgraphik (50 Punkte)

Übersicht: In dieser Aufgabe geht es darum, eine komplexere, „nichtlineare" Transformation (jenseits von Rotation, Verschiebung und was `QTransform` sonst so kann) auf geometrische Objekte anzuwenden. Speziell betrachte wir eine (Art von) sogenannten „Freiformdeformationen". Dies kann man nicht mehr ohne weiteres umsetzen, indem man eine neue Klasse mit einer neuen „`draw()`"-Routine baut, sondern man muss die Geometrie selbst verändern. Daher stößt unsere bisherige Abstraktion an Grenzen: Um beliebige Objekte transformieren zu können, müssen wir wissen, wie die Geometrie der Objekte aussieht; eine Kenntnis einer Pixelrepräsentation alleine reicht nicht aus. Daher ändert (bzw. ergänzt) man die Schnittstelle so, dass die geometrischen Objekte, aus denen die Szene besteht („Shapes" in der Vorlesung) sich geometrisch beschreiben, anstatt einfach selbst zu zeichnen. Danach kann man solche komplexeren Operationen umsetzen. Ein Beispiel davon schauen wir uns (schonmal) an.

Theorie: In der Vorlesung werden wir uns das genauer anschauen und sehen, dass es sich hier um einen recht grundsätzlichen Trade-Off (Abwägung) handelt: Entweder beschreibt man explizit, wie die Formen aussehen, nutzt dazu aber einen festgelegten Katalog an Primitiven. Oder man erlaubt beliebigen Code, der die Formen zeichnen, was mehr Gestaltungsspielraum bietet, aber man kann weniger gut kontrollieren, was man erhält (man bekommt eigentlich auch nur Pixel), und entsprechend weniger im weiteren Verlauf „damit machen". Der „zeichne dich selbst" Ansatz ist dabei typische für OOP; der „beschreibe dich in einer Sprache aus festen Datentypen" ist eher im funktionalen Programmieren sinnvoll. Im einen Fall kann man leichter neue (recht beliebige) Formen definieren, im anderen Fall (um den es auf diesem Übungsblatt geht) kann man leichter Operationen („Funktionen") definieren, die die Objekte allgemeiner transformieren können.

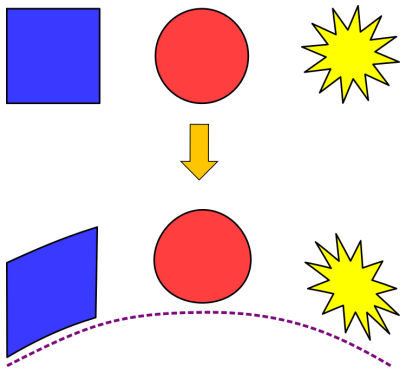

Abbildung 1: Eine Freiformdeformation die die Zeichnung in y-Richtung entlang einer Sinuskurve verschiebt. Die Kurve ist so gewählt, das eine halbe Schwingung erscheint, und die Werte der Kurve werden auf die y-Achse aller Punkte der Objekte draufaddiert.

Konkret

Wir möchten eine ganze Liste von Shapes (Kreise, Rechtecke, Sterne), die wir schon kennen, entlang einer Kurve deformieren, wie in Abbildung 1 gezeigt. Aus Gründen der Einfachheit soll es um die Funktion

$$f(x) = a \cdot \sin(2\pi \cdot b \cdot x + c)$$

handeln, mit sinnvoll gewählten Konstanten a (Höhe der Schwingung in Weltkoordinaten), b (Breite der Schwingungen in Weltkoordinaten) und c (Startverschiebung der Schwingung).

Die Deformation erfolgt nun so: Punkt (x, y) in einer geometrischen Form werden in die Punkte $\big(x, y + f(x)\big)$ umgewandelt (also in y-Richtung wellenförmig verschoben; wer Lust auf mehr hat: Sie dürfen natürlich gerne auch noch andere Ansätze für Deformationen hinzufügen).

Das Problem dabei ist, dass gerade Linien dabei in Kurven umgewandelt werden. Daher können wir nicht einfach nur die Eckpunkte aller Objekte transformieren, sondern müssten eigentlich alles durch entsprechende Kurven ersetzen. Das ist mathematisch und algorithmisch allerdings sehr anspruchsvoll und zu schwer für eine EIS-Übung (und auch für die viele praktische Software, die so etwas macht, nutzt eher Tricks wie den unten beschriebenen).

Daher gehen wir „einfacher" vor: Wir teilen einfach die Objekte in kleine Vierecke oder Dreiecke auf, die wir dann einfach eckpunktweise verformen: Wenn eine Form aus vielen kleinen Fragmenten besteht, sieht man gar nicht mehr, dass der Rand nicht glatt ist. Hier ein Beispiel mit dem Viereck:

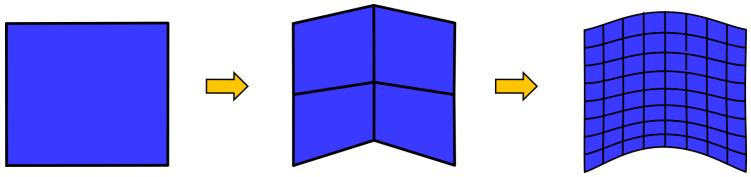

Abbildung 2: Wenn man die Formen in viele kleine Fragmente zerteilt, fällt gar nicht auf, dass deren Kanten eigentlich gerade sind (alle drei Formen bestehen nur als Vierecken mit geraden Seiten)

Die Implementation einer solchen Zerteilung kann etwas aufwendig sein; am einfachsten ist es, wenn man alle Formen bereits in Dreiecke zerlegt (einfacher als Vierecke), und diese dann durch mehrere, wiederholte „1:4"-Splits (Vierfachaufteilungen) verfeinert.

Abbildung 3: Man kann so ziemlich alle Formen leicht durch Dreiecke annähern (siehe rechts), und Dreiecke kann man dann leicht durch 1:4-Splits verfeinern.

Nun die Aufgabe

a) Erweitern Sie Ihr Zeichenprogramm so, dass die Graphikobjekte (Kreise, Rechtecke, Sterne) sich in Dreiecke zerlegen können. Kreisen würde man dann (wie in Abbildung 3 links gezeigt) approximieren. Sehen Sie dazu z.B. eine abstrakte Methode

„`describeShape() -> List[Triangle]`"

oder ähnliches in der Basisklasse (zusätzlich zu „`draw()`") vor.

b) Erweitern Sie nun Ihre Graphikbibliothek um zwei Operationen:

- **Subdivision:** Teilt alle Eingabedreiecke in jeweils vier Ausgabedreiecke auf (wie in Abbildung 3 rechts). Diese Operation kann n-mal (für ein kleines $n = 1,2,3,..$) wiederholt werden. Die „`describeShape()`"-Methode muss hier also die eingehenden Dreiecke in kleinere umwandeln. Sinnvoll in der Praxis sind Werte von n im Bereich von vielleicht 2 bis 4 (4x bis 256x so viele Dreiecke).

- **Deformation:** Die Eckpunkte der Dreiecke werden wie oben beschrieben entlang einer Sinuskurve verformt. Die Parameter der Kurve sollten sinnvoll gewählt werden, so dass man den Effekt gut sehen kann.

c) Wenden Sie eine Folge von erst „Subdivision“ (mit sinnvollen Parametern) und danach „Deformation“ auf die Testszenen von Übungsblatt 03 an, und visualisieren Sie das Ergebnis. Es sollte möglich sein, beliebige Szenen bestehend aus Rechtecken und Kreisen (ggf. Sternen, s.u.) zu deformieren (ähnlich Abbildung 1).

Hinweis zur Benotung / Anforderungen: Die Implementation der Operationen für den Stern („Star“) ist freiwillig – hierfür gibt es keine Punkte bzw. auch keinen Punktabzug, falls sie fehlt. Es ist auch nicht notwendig, Randkurven richtig zu handhaben (flächig farbig gefüllte Dreiecke ohne Ränder sind hier gut genug; Linienzüge für Räder zusätzlich zu erzeugen ist nur weitere Arbeit, die wenig neue Erkenntnisse liefert). Wenn Sie möchten, können Sie diese Operationen in das GUI des Zeichenprograms einbauen. Auch dies ist aber freiwillig. Die Wahl der Primitive (Dreiecke, Vierecke oder sonstiges) ist ebenfalls freigestellt (Dreiecke sind nur ein Beispiel, wie man es machen könnte).

Noch ein Tipp: Es kann sinnvoll sein, eine zusätzliche Operation zu definieren, die eine Gruppe von Shapes zu einem zusammengesetzten Objekt zusammenfügt, und dann die anderen Operationen so zu gestalten, dass sie auf genau ein „Shape“ wirken (im Zweifelsfall auf eine ganze Gruppe). Die Musterlösung für Übungsblatt 04 stellt z.B. ein solches Konstrukt bereits in Form einer Klasse „`ShapeList`“ bereit (natürlich noch ohne Operationen für die Zerlegungen in Dreiecke). Überlegen Sie sich auch, wie das Zeichnen realisiert wird (am Ende muss das Ergebnis ja als Bild dargestellt werden). Hierzu macht es Sinn, an der richtigen Stelle doch wieder ein oder mehrere „draw()“-Methoden bereitzustellen.

Aufgabe 2: Export in eine Datei (50 Punkte)

Erweitern Sie Ihre Vektorgraphikbibliothek so, dass man die Zeichnung in einer Textdatei abspeichern kann, in der alle Formen enthalten sind (die Operationen aus Aufgabe 1 selbst brauchen nicht gespeichert werden; in dem Fall reicht es, die Dreiecke anzuzeigen; allgemein ist eine Approximation durch Dreiecke für alle Objekte akzeptabel, und Randkurven brauchen nicht enthalten zu sein).

Eine Implementation, die die Geometrie sinnvoll in Text umsetzt (z.B. Liste von Shapes) erhält 25 Punkte. 25 weitere Punkte gibt es dann, wenn die erzeugten Dateien korrekte SVG-Dateien sind, die man sich mit einem SVG Viewer wie einem Webbrowser oder Inkscape anschauen kann (siehe z.B. https://de.wikipedia.org/wiki/Scalable_Vector_Graphics). Es ist *nicht* notwendig, das Wiedereinladen aus den erzeugten Dateien auch noch zu programmieren (ist aber eine nette Übung, wenn man möchte). Rechtecke, Kreise, Sterne, und von Operatoren (Aufgabe 1) erzeuge Formen sollten aber sinnvoll wiedergegeben werden.

Didaktischer Hinweis: Was ist an Aufgabe 2 „funktional“? Ein wichtiges Muster ist es, Bäume von Objekten in andere Bäume von Objekten (oder sequentielle „linearisierte“ Darstellungen) zu übersetzen. Dazu definiert man Funktionen, die die Teile übersetzen, und fügt diese zu einer *Gesamtübersetzungsfunktion* zusammen. Genau das machen wir hier. :-) Programmieren kann man das aber mit sowohl mit OOP, prozeduralen oder auch funktionalen Konstrukten (überlegen Sie sich einfach einen guten Ansatz; das sollte nicht kompliziert sein).