Programmierpraktikum Computergrafik WiSe24/25

GUSTAV OTZEN

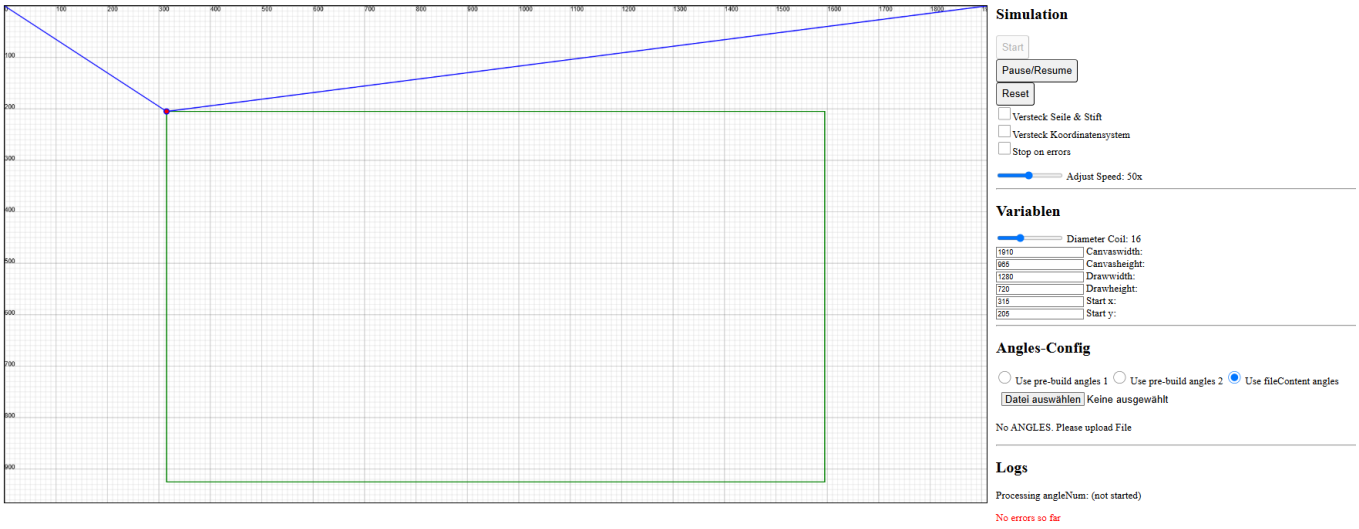

Abb. 1. Bild unseres fertigen Simulationsprogrammes

1 EINLEITUNG

In diesem Bericht geht es um die Inhaltlichen Schwerpunkte der „Simulations-Gruppe“ im Rahmen des Programmierpraktikums Computergrafik im WiSe24/25 der Universität Osnabrück.
Dabei bestand diese Gruppe aus zwei Mitgliedern: Mir, Gustav Otzen, und Lino Viets. Manches haben wir zusammen erarbeitet, uns für unsere Berichte aber unterschiedliche Themenschwerpunkte herausgesucht, über die wir hauptsächlich schreiben; An manchen Stellen sind Doppelungen aber trotzdem nicht auszuschließen.
Mein Bericht setzt sich daher aus den folgenden drei Hauptkapiteln zusammen, wobei ich persönlich besonders an dem dritten und großen Teilen des zweiten Punktes gearbeitet habe:

- Kommunikation mit den anderen Gruppen
- Entwurf der graphischen Benutzeroberfläche
- Entwurf einer flüssigen Animation

Nachfolgend werden diese Kapitel der Reihe nach näher erläutert. Davor wird zwecks Kontextualisierung das Ziel des Praktikums im folgenden Absatz einmal kurz umrissen.

1.1 Zielgebung

Das Programmierpraktikum beschäftigte sich mit der Entwicklung einer Zeichenmaschine, welche durch kontinuierliche Kurven ein beliebiges Bild in schwarz-weiß auf einer Unterlage (Whiteboard, Papier) malt. Dazu wurden wir in verschiedene Gruppen eingeteilt, die alle zusammen gearbeitet haben, um am Ende dieses Ziel zu erreichen. Jede Gruppe ist dafür verschiedene Kernpunkte und Probleme angegangen, wie beispielsweise das 3D-Drucken der benötigten Ressourcen oder das Entwerfen eines Algorithmus, der ein gegebenes Bild in Bezierkurven aufteilt. Wie schon oben erwähnt, bestand meine Aufgabe darin, eine Simulation für diesen Zeichenprozess zu entwickeln. Lino und ich haben uns dazu für die Programmiersprache JavaScript (mit HTML und CSS) entschieden; Näheres zu den Gründen steht in Linos Bericht.

Kontakt: Gustav Otzen, gotzen@uni-osnabrueck.de.

2 KOMMUNIKATION

Eine unserer größten Aufgaben als „Simulations-Gruppe“ bestand neben der Programmierung und Implementierung einer funktionierenden und benutzerfreundlichen Simulation in der Kommunikation und Vermittlung mit den anderen Gruppen des Praktikums. Insbesondere erwähnt seien hier die „Arduino-Gruppe“, denen unsere Simulation als Basis dienen sollte und der „Bezierkurven-Gruppe“, welche die für uns benötigten Winkelsequenzen aus einer Menge von Bezierkurven extrahierten.

2.1 Der Input für unsere Simulation

Zu Beginn des Programmierpraktikums stellte sich uns vor allem die Frage nach den Input-Daten, welche wir bekommen und dann im Sinne einer Simulation weiterverarbeiten sollten. Recht schnell wurde uns klar, dass wir mit der reinen Implementation einer vollumfänglichen Simulation schon gut beschäftigt sein würden. Deshalb einigten wir uns darauf, dass die Aufgabe, aus den gegebenen Bezierkurven Winkelsequenzen zu erstellen, nicht bei uns liegen sollte, sondern eine Aufgabe der Gruppe vor uns darstellt. Die Idee hinter der Nutzung dieser Sequenzen als Input war, dass wir mit denselben, sehr hardwarenah gedachten, Größen operieren, welche auch der Arduino später bekommen sollte. Da die Winkelsequenzen direkt die Motoren steuern, erschien uns das hier am sinnvollsten.

Wir wollten unsere Simulation von Beginn an so anpassungsfähig wie möglich gestalten, dazu haben wir uns recht schnell auf eine externe Text-Datei mit den oben erwähnten Winkelsequenzen geeinigt. Diese konnten dann später sowohl von unserer Simulation, als auch (per SD-Karte) vom Arduino selbst eingelesen werden, was uns ermöglichte, sehr flexibel und schnell zwischen verschiedenen zu zeichnenden Formen zu wechseln. Um ein möglichst realitätsnahes Ergebnis zu erreichen, wurde uns zudem pro Winkelsequenz ein diskreter Zeitwert in Millisekunden gegeben, in dem der Arduino diese Winkelsequenz abarbeitet. Zudem wurde nach dem ersten Praxis-Test mit dem Arduino am Whiteboard klar, dass es (auch zur Bemessung der Zeichenungenauigkeiten) hilfreich wäre, eine Winkelsequenz von einem von uns definierten Startpunkt zu dem wirklichen Zeichenstartpunkt des Bildes und zurück zu haben. Nach Besprechung mit den Tutoren haben wir uns darauf geeinigt, die erste Zeile einer Datei zu verwenden, um zum Zeichenstartpunkt zu kommen und die zweite Zeile, um wieder zum Anfangspunkt zu gelangen. Die folgenden Winkelsequenzen werden dann dazu genutzt, das eigentliche Bild zu zeichnen. Unsere Abmachung war es, dass positive Winkel immer mehr Schnur geben, während negative die Schnur verkürzen.
In der folgenden Grafik ist dieses Format einer .txt-Datei schematisch abbgebildet:

Zeilennummer	Zeitwert [ms]	Winkel linker Motor [°]	Winkel rechter Motor [°]
1 (Start -> Zeichenstart)	1230	230	533
2 (Zeichenende -> Start)	340	49	78
3 (Winkelsequenz #1)	1798	761	214
4 (Winkelsequenz #2)	154	45	-45
...	...	...	...

Abb. 2. Schematische Darstellung einer .txt Datei im festgelegten Format

2.2 Fehlererkennung

Allgemein hat die kontinuierliche Absprache mit den anderen Gruppen sehr dabei geholfen, sowohl eigene Fehler in unserer Simulation, als auch Fehler bei Anderen zu entdecken.
So half unsere Simulation bei dem ersten „Praxis-Test“, wo wir mittels des Arduinos auf einem Whiteboard gezeichnet haben, mit einem verstellbaren Slider für den Spulendurchmesser (s. Section 3.3) Unklarheiten bezüglich diesen zu klären.
Auch haben wir erkannt, dass es wesentliche Missverständnisse bezüglich des Koordinatenursprungs gegeben hatte: So hatte eine Gruppe diesen unten links gewählt, während andere Gruppen aber von oben links ausgegangen sind. Das hat zu falschen Endergebnissen und Diskrepanz zwischen den zu erwartenden Ergebnissen und denen unserer Simulation geführt.
Der letzte große Fehler, der zu einem falschen Endergebnis innerhalb der Simulation geführt hatte, lag allerdings bei uns: Wir hatten durch unsere Algorithmen dafür gesorgt, dass das gezeichnete Bild auch wirklich kontinuierlich ist und nicht etwa nur jeder Endpunkt einer Winkelsequenz gemalt wird. Hier ist es uns aber durch diverse Logikfehler (dazu schreibt Lino mehr) passiert, dass sich der wirkliche Endpunkt (hätten wir direkt die vollen Winkel einer Sequenz genommen) von dem unseren Programms unterschied.
Alle oben genannten Fehler konnten wir als Gruppe und mithilfe der Tutoren gut beheben, sodass wir am Ende des Praktikums die von uns ausgesuchten Bilder erkennbar zeichnen konnten.

3 GRAFISCHE BENUTZEROBERFLÄCHE

Mein Aufgabenschwerpunkt innerhalb unserer Gruppe war die Entwicklung einer bedienbaren graphischen Benutzeroberfläche und die Implementierung wesentlicher Funktionen innerhalb dieser. Deswegen werde ich nachfolgend auf die Funktionalitäten unserer in Abbildung 1 zu sehenden GUI sprechen. Eine vollständige Übersicht mit allen HTML Elementen sowie Funktionalitäten dieser findet sich in der Tabelle 1.

3.1 Die Zeichenfläche

Den meisten Platz unserer Simulation nimmt die Zeichenfläche an sich ein. Diese ist aus insgesamt drei HTML Canvas Elementen aufgebaut:

- Die zwei Seile mit dem eingespannten Stift stellen ein eigenes Canvas im „Vordergrund“ dar.
- Für das per Radiobutton einblendbare Koordinatensystem mit der Einzeichnung einer grün umrandeten Zeichenfläche habe ich mich auch für ein eigenes Canvas entschieden.
- Das Whiteboard, auf dem gezeichnet wird, stellt ein eigenes Canvas dar. Auf diesem werden Pixel farbig eingezeichnet; Denkbar wäre hier eine Speicherung verschiedener Pixel oder Zustände, um beliebig innerhalb der Simulation vor- und zurückzuspulen zu können (das konnten wir leider aufgrund mangelnder Zeit nicht mehr implementierten; Zu weiterführenden Ideen von uns sei auch hier wieder auf Linos Bericht verwiesen)

Diese Entscheidung, drei verschiedene Canvas zu implementieren, brachte unserer Gruppe zwei entscheidende Vorteile: Einerseits war es nun deutlich einfacher, verschiedene Elemente auszublenden, indem wir einfach die jeweiligen Canvas mittels Befehlen wie „canvas.clearRect()“ geleert oder gefüllt haben. Andererseits konnte ich so die Performance unserer Algorithmen deutlich verbessern, da wir davor bei jedem neu gemalten Pixel die bisherigen Pixel gelöscht und alles neu gezeichnet haben. Diese Methode funktionierte auch, allerdings wurde schnell klar, dass das nicht gut mit der Eingabegröße skalieren und wir deutliche Probleme bei komplexeren Bildern mit extremst vielen Pixeln haben würden (In jeder Iteration müsste bei Pixel n alle Pixel $\in \{1, ..., n-1\}$ gelöscht werden).

3.2 Grundlegende Steuerung

Rechts von der eben behandelten Zeichenfläche finden sich verschieden Reiter, dieses Unterkapitel beschäftigt sich mit dem ersten Reiter „Simulation“. Die anderen Reiter werden in den nachfolgenden Unterkapiteln beschrieben.
Zur Steuerung der Simulation operiert unserer Algorithmus auf zwei global angelegten booleschen Variablen *running* und *pause*. *running* gibt an, ob die Simulation gerade läuft und *pause*, ob die

laufende Simulation angehalten werden soll. Durch Flippen der Variable *pause* bei Drücken des Buttons „Pause/Resume“ und Abfragen dieser Variablen mittels „Busy-Waiting“ (nicht optimal, reicht aber für unsere Zwecke), war es uns möglich, die Simulation nach Belieben anzuhalten und fortzuführen.
Auch kann der aktuelle Wert der Variable *running* beliebig innerhalb des Algorithmus abgefragt werden und falls dieser wahr ist, der aktuelle Codeblock beendet werden. Diese Variable wird bei Druck auf den „Restart“-Button auf den Wert $false$ gesetzt.
Der Start-Button erfüllt die Aufgabe einer Initialisierung vor jedem Durchlauf (*running* auf $true$ und *pause* auf $false$ setzen, Canvas leeren, usw.) und erscheint ausgegraut, sollte die Simulation gerade laufen (also *running* wahr sein) oder noch keine Winkelsequenz geladen sein (s. Sektion 3.3).
Das Verhalten von *pause* und *running* innerhalb eines Simulationsalgorithmus (Näheres zur Implementierung davon bei Lino) ist in dem untenstehenden Pseudocode verdeutlicht:

Algorithm 1 Pause/Resume & Restart

Require: *pause*, *resume*
Ensure: /

```
1: while Simulation läuft do
2:     Berechne und zeichne
3:     while pause do
4:         sleep(100)                ▷ Pause für 0.1 Sekunden
5:     end while
6:     if !running then return       ▷ Stoppt Simulation
7:     end if
8: end while
```

Darunter finden sich zwei Funktionen, mit denen man sich ein unter der Zeichnung liegendes Koordinatensystem bzw. die Seile und den Stift anzeigen lassen kann.
Startet man die Simulation, so wird im unteren Reiter „Logs“ eine kleine Übersicht über die aktuelle Winkelsequenz und einer Fehleranzeige angezeigt. Standardmäßig läuft die Simulation weiter, wenn Fehler gefunden werden (außerhalb des Zeichenbereiches o.Ä.); Mithilfe der letzten Checkbox „Stop on errors“ lässt sich dieses Verhalten ändern.
Die Geschwindigkeit, in der das gewünschte Bild gezeichnet wird, ist durch einen Slider darunter einstellbar. Näheres zu meiner Implementierung der Animation erkläre ich in Kapitel 4.2.

3.3 Einstellen gewünschter Parameter

Die Elemente des zweiten Reiters „Variablen“ besitzen die Optionen, den Spulendurchmessers des Motors sowie bestimmte Maße zu ändern, um die Simulation flexibler zu gestalten.
Standardmäßig habe ich die im Rahmen des Praktikums festgelegten Abmessungen implementiert. Diese sind ein Spulendurchmesser von 16mm und stellen das Whiteboard mit dem darauf befindlichen Zeichenbereich dar und sind der Grafik 3 zu entnehmen.
Durch einen Slider kann der Benutzer den gewünschten Spulendurchmesser von 8mm bis 32mm frei wählen. Für diesen Bereich und eine Schrittgröße von 8mm habe ich mich wegen der in der Praxis benutzten Durchmesser sowie der Übersichtlichkeit entschieden. Die Änderung des Spulendurchmessers lässt sich als Skalierung des gezeichneten Bilds auffassen, da sich pro Schritt die auf- bzw. abgerollte Länge des Seils durch den Umfang der Spule, welche das Seil hält, berechnen lässt.
In den sechs weiteren Feldern können nun die gewünschten Maße (in folgender Reihenfolge) eines neuen Whiteboardes, einer neuen Zeichenfläche und eines neuen Startpunktes definiert werden. Dabei ist zu beachten, dass die Zeichenfläche auf dem Canvas immer zentriert wird und einen Abstand von 4cm vom unteren Rand des Canvas hat. Diese Implementation war ein Wunsch von einer Gruppe und vereinfacht den Prozess, indem nicht nochmal extra Punkte für die Zeichenfläche eingegeben werden müssen. An dieser Stelle sei zudem dran erinnert, dass der Startpunkt nur den initialen Startpunkt ändert, nicht aber den Punkt, ab wo gezeichnet wird (wie in Kapitel 2.1 erwähnt).

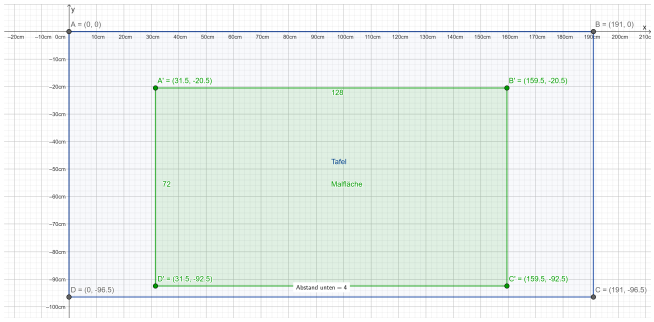

Abb. 3. Initiale Abmesssungen des benutzten Whiteboards

Mit dem Einstellen einer gewünschten abzuarbeitenden Winkelsequenz beschäftigt sich der Reiter „Angles-Config“. Hier habe ich drei verschiedene Optionen implementiert:

- Eine vordefinierte Winkelsequenz „pre-build angles 1“. Dieses Tupel von Winkelsequenzen und den dazugehörigen Zeiten sind im Code vordefiniert und stellen eine Boundingbox der Zeichenfläche dar (bei 8mm Spulendurchmesser und der initialen Zeichenfläche!)
- Wie oben, nur mit „pre-build angles 2“ wird ein Dreieck gezeichnet (Hier wurde auch wieder mit einem Spulendurchmesser von 8mm gearbeitet, andere Durchmesser skalieren das Dreieck).
- Durch die letzte Option „fileContent angles“ lassen sich auch nach Abb. 2 eigens definierte Winkelsequenzen innerhalb einer .txt Datei einlesen. Dazu wird intern mit einem von mir geschriebenen Parser der Inhalt der Datei in ein für JavaScript lesbares Tupel von 3-Tupeln (Zeit, Winkel MotorLinks, Winkel MotorRechts) übersetzt.

Unter dieser Auswahl wird dem Benutzer die erste sowie letzte Zeile und die gesamte Anzahl an Winkelsequenzen der ausgewählten Datei angezeigt.

Tab. 1. Funktionen aller HTML-Elemente der GUI; Von oben nach unten aufgelistet

Name	Art	Funktion
Start	Button	Startet die Simulation
Pause/Resume	Button	Pausiert/Startet die Simulation
Reset	Button	Setzt die Simulation zurück
Versteck Seile&Stift	Checkbox	Schaltet Ansicht des Vordergrunds an/aus
Versteck Koordinatensystem	Checkbox	Schaltet Ansicht des Koordinatensystems an/aus
Stop on errors	Checkbox	Entscheidet, ob Simulation bei Fehler stoppen/weiterlaufen soll
Diameter Coil	Slider	Eingabe des verwendeten Spulendurchmessers
Canvaswidth	Input	Eingabe Breite des Canvas
Canvasheigth	Input	Eingabe Höhe des Canvas
Drawwidth	Input	Eingabe Breite der Zeichenfläche
Drawheight	Input	Eingabe Höhe der Zeichenfläche
Start x	Input	Eingabe x-Koordinate Startpunkt
Start y	Input	Eingabe y-Koordinate Startpunkt
Use pre-build angles 1	Radiobutton	Benutzt Winkelsequenzen, die eine Boundingbox ergeben
Use pre-build angles21	Radiobutton	Benutzt Winkelsequenzen, die ein Dreieck ergeben
Use fileContent angles	Radiobutton	Benutzt eigens definierte Winkelsequenzen einer hochgeladenen Datei

4 ANIMATION

Meine letzte große Aufgabe war es, die Simulation anständig und so ruckelfrei wie möglich zu animieren. Einerseits wurden hier unsere Möglichkeiten durch die Framerate des Browsers, andererseits durch die Dauer der Berechnung der einzelnen Koordinatenpunkte limitiert. Trotzdem wollten wir für unsere Simulation eine anständige Art und Weise schaffen, wie man das Tempo selbst einstellen kann.

4.1 Erste Schritte

Die ersten Monate des Praktikums hatten wir eine von mir entworfene, aber recht simple Idee für die Animation genutzt: Nach einer bestimmten Anzahl (hier: 1000) an abgearbeiteten Winkelsequenz 10 Millisekunden warten; Eine Implementierung dieser Idee ist unten im Pseudocode 2 verdeutlicht.

Algorithm 2 Simple Animation

Require: Liste aller Winkelsequenzen *SEQS*
Ensure: /

```
1: while Simulation läuft do
2:     for i = 1 to SEQS.length do
3:         Berechne und zeichne
4:         if i mod 1000 == 0 then
5:             sleep(10)            ▷ Pause für 10 Millisekunden
6:         end if
7:     end for
8: end while
```

Diese extrem simple Implementierung hat tatsächlich recht gut funktioniert und wir konnten für den Anfang damit arbeiten, allerdings konnte man so noch nicht die wirkliche Animationsgeschwindigkeit einstellen und es wurde nicht die vorgegebenen Zeiten pro Winkelsequenz berücksichtigt.
Deswegen hatte ich darüber nachgedacht, wie man diesen Algorithmus verbessern könnte und hatte nach ein paar Anläufen und mit Hilfe von unserem Tutor eine passende Idee, die ich im Folgenden etwas genauer vorstellen werde.

4.2 Finaler Algorithmus

Nachfolgend werde ich meinen final implementierten Algorithmus vorstellen und dabei auch den folgenden Pseudocode 3 erläutern, indem die dort genutzten Variablen bei der jeweils textuellen Erklärung annotiert sind:
Das Grundprinzip aus meinem vorherigen Algorithmus, nach so und so vielen Schritten eine gewisse Zeit zu warten, habe ich beibehalten. Die wesentliche Veränderung, die ich vorgenommen habe, bestand aus einer Anpassung, nach wie vielen Schritten gewartet wird. Statt einer fixen Anzahl von 1000 Schritten habe ich nun die vorgegebenen Zeiten ($timeInterval$) berücksichtigt und konnte mit diesen und der von uns berechneten Anzahl an benötigten Iterationen pro Winkelsequenz ($num\_iterations$) für jede dieser genannten Sequenzen einen durchschnittlichen Wert ermitteln, der nach einer Iteration geschlafen werden sollte, um am Ende auf den gegeben Zeitwert zu kommen ($sleepTime\_per\_iter$). Dieser Durchschnittswert ist in der Regel sehr klein (Größenordnung $\approx 10^{-2} ms$; sehr abhängig von gegebenen Werten). Daher werden diese Werte mit jeder Iteration akkumuliert (acc_sleep) und nur geschlafen, wenn der aufaddierte Wert größer als eine von der eingestellten Animationsgeschwindigkeit (möglich per in 3.2 erwähnten Slider; Variable *animSpeed*) abhängigen Obergrenze ist. Eine weitere wichtige Größe ist die maximale Zeit pro Frame ($frametme$), welche in Millisekunden angegeben ist und sich aus der Bildschirmwiederholungsrate errechnet (hier $60Hz$; also $\frac{1}{60s} \cdot 1000 \approx 16,7ms$). Dieser Wert bestimmt zusammen mit der Animationsgeschwindigkeit die eben erwähnte Obergrenze (ub) und ist die Zeit, die gewartet wird. (Hier lag ein Problem, welches ich sehr lange zu Debuggen versucht habe: JavaScript hat als Wartezeit auch die von mir - im Rahmen einer anderen Idee davor - eingestellten $1ms$ akzeptiert, jedoch war es nicht möglich, im Browser auch wirklich nur $1ms$ zu warten; Dies hat dazu geführt, dass die Animation zwar funktioniert hat, jedoch

deutlich länger als von mir berechnet gebraucht hat; Deshalb habe ich den Wert auf die $frametime$ gesetzt).
Um den berechneten Wert *acc_sleep* so genau wie möglich zu gestalten, war es nötig, auch die tatsächlich benötigte Zeit des Algorithmus (*time_per_iter*) zu berücksichtigen und die Variable *acc_sleep* korrekt aufzuaddieren, was in Zeile 4-8 des Pseudocodes passiert.
Schlussendlich müssen noch die aufaddierten Werte im Falle des Wartens zurückgesetzt werden, was in der letzten Zeile 12 geschieht.
Der bereits erwähnte Algorithmus ist in dem untenstehenden Pseudocode verdeutlicht:

Algorithm 3 Animation

Require: $SEQS, num\_iterations, timeInterval, animSpeed, frametime, acc\_sleep$
Ensure: /

```
1:  while Simulation läuft do
2:      for each Winkelsequenz in SEQS do
3:          sleepTime_per_Iter ← timeInterval / num_iterations
4:          iter_startTime ← aktuelle Zeit
5:          Berechne und zeichne
6:          iter_endTime ← aktuelle Zeit
7:          time_per_iter = iter_endTime − iter_startTime
8:          acc_sleep ← acc_sleep + time_per_iter
9:          ub ← animSpeed ∗ frametime
10:         if acc_sleep ≥ ub then
11:             sleep(frametime)
12:             acc_sleep ← acc_sleep − ub
13:         end if
14:     end for
15: end while
```

5 FAZIT

Abschließend lässt sich festhalten, dass unsere Gruppe trotz anfänglicher Schwierigkeiten zu einem verlässlichen Endprodukt gekommen ist. Lino und ich haben Vieles gemeinsam besprochen und Ideen entwickelt, jedoch von Anfang an eine gute Arbeitsteilung gehabt, indem wir die theoretischen Vorüberlegungen noch gemeinsam angestellt haben, ab da sich aber jeder zunehmend auf einen Themenbereich fokussiert hat. Während ich hauptsächlich für unsere GUI und die Implementierung neuer, für die Simulation nützlicher Funktionen zuständig war, hat sich Lino viel um die Implementierung und Optimierung des „Kern-Algorithmus“ gekümmert.
Durch das Praktikum konnte ich meine Kenntnisse in JavaScript deutlich erweitern und auch das erste Mal an einem etwas größeren Projekt mit mehreren verschiedenen Arbeitsaufteilungen/Gruppen arbeiten. Das hat mir gezeigt, wie wichtig eine gute Abstimmung und Kommunikation in der Praxis ist.
Insgesamt haben wir natürlich - im Nachhinein betrachtet, vermeidbare - Fehler gemacht, aber ich denke, das ist normal und hat uns auch viel gelehrt. Mit unserer Simulation bin ich aber durchaus zufrieden.