Universität Leipzig
Institut für Informatik
Abteilung für Bild- und Signalverarbeitung
Praktikum Computergrafik

SoSe 2024
Baldwin Nsonga
Lydia Müller

Aufgabenblatt 1

Aufgabe 1 - Vorbereitung

1. Laden Sie sich das Quellcodegerüst zur Aufgabe von der Praktikumsseite herunter.
2. Installieren Sie Assimp[1] und die Parallelisierungs-Library OpenMP.
3. Testen Sie, ob die von uns vorgegebene Codebasis auf Ihrem Rechner fehlerfrei läuft:
   (a) Erstellen Sie einen *build* Ordner auf der Höhe des *src* Ordners.
   (b) Wechseln Sie mit *cd build* in diesen Ordner.
   (c) Rufen Sie *cmake ..* und danach *make* auf.
   (d) Starten Sie das Programm mit *./CGPraktikum ../results/test.ppm*. In *results* sollte danach ein blaues Bild mit dem Dateinamen *test.ppm* zu finden sein.
4. Arbeiten Sie sich in C++, KDbg[2], eine IDE und das gegebene Quellcodegerüst ein.

Aufgabe 2 - Bresenham-Algorithmus

Setzen Sie den Bresenham-Algorithmus zum Rastern von Geraden in der Methode *WireframeRenderer::drawBresenhamLine* um. Implementieren Sie in dieser Methode ausschließlich die Fallunterscheidung nach der Steigung. Es muss so umgesetzt werden, dass das eigentliche Zeichnen der Linie für jeden Oktanten in einer eigenen Methode ausgelagert werden kann. Testen Sie Ihre Methoden, indem Sie *WireframeRenderer::drawBresenhamLine* in der Main-Methode in *main.cpp* mit einer von Ihnen frei wählbaren Beispielgeraden aufrufen. Konstruieren Sie Testfälle für alle Oktanten und zeichnen Sie ein Ergebnis, in welchem eine Gerade für jeden Oktanten dargestellt wird. Beachten Sie, dass Sie in diesem Aufgabenblatt auf einem Pixelraster arbeiten und der Bresenham Algorithmus auf Gleitkommaoperationen verzichtet! Notwendig zum bestehen sind mindestens 8 Geraden in jeweils unterschiedlichen Oktanten.

Ein Beispiel können Sie in Abbildung 1 sehen. Anmerkung: Uns ist bewusst, dass eine Implementierung auch Oktanten-unabhängig möglich ist, fordern jedoch explizit, die Implementierung Oktanten-spezifisch umzusetzen. Zur Konstruktion des Beispiels (siehe Abbildung 1) wurde der Komplexe Zahlenraum verwendet. Dabei entsprechen die x- und y-Komponenten dem Reelen und Imaginären Anteil. Durch die Eulersche Formel lassen sich die gezeigten Geraden wie folgt generieren:

$$z_n = ae^{n*\frac{1}{8}\pi i} \in \mathbb{C}, \qquad n = [0, 1, 2, ..., 15] \in \mathbb{N}, \tag{1}$$

[1] `http://assimp.org/`
[2] `http://www.kdbg.org/`

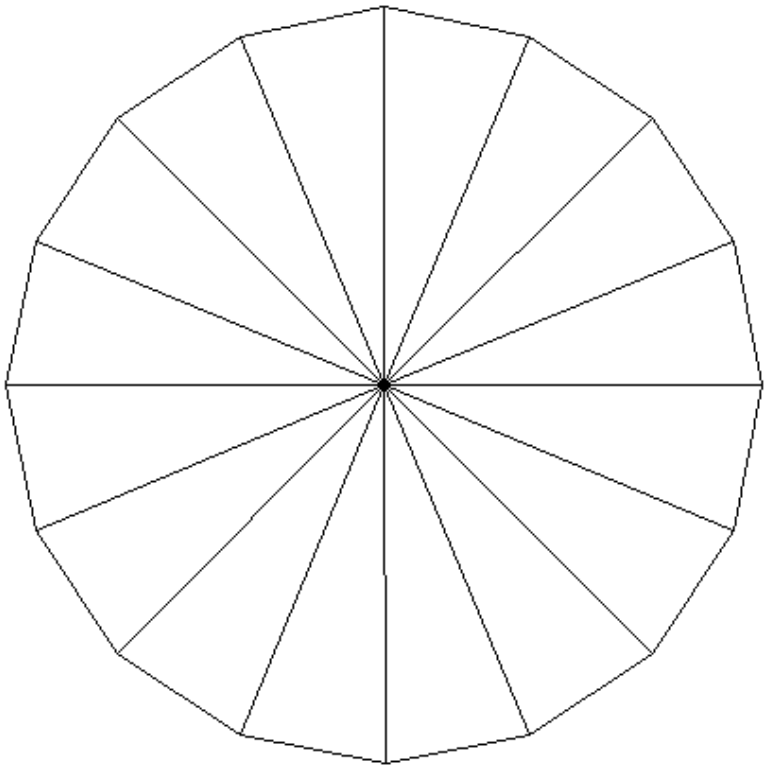

Abbildung 1: Zielsetzung der Bresenham-Implementierung

wobei $a = 200 \in \mathbb{N}$. Da sich diese Punkte Kreisförmig um den Koordinatenursprung generiert werden, ist es nun notwenig diese in das Bild zu verschieben. Mit dem Zentrum c = *GLPoint(img* $\rightarrow$ *getWidth()/2.0, img* $\rightarrow$ *getHeight()/2.0, 0.0)*, welches der Ausgabgspunkt der Oktrantengeraden ist, ergeben sich die generierten Punkte p_n durch addition der Komponenten von z_n und c. Schließen Sie das "Rad", indem Sie die Punkte p_n und p_{n+1} durch eine Gerade miteinander verbinden.

Aufgabe 3 - Seed-Fill

Nutzen Sie die zuvor erstellte *drawBresenhamLine*-Methode, um eine einfache geometrische Form zu zeichnen. Anschließend füllen Sie die Pixel innerhalb dieser Form mit einer beliebigen Farbe. Implementieren Sie dazu die Methode *WireframeRenderer::seedFillArea* (verwenden Sie dabei **keine** Rekursion). Testen Sie die Methode indem Sie die in Aufgabe 2 erzeugten Dreiecke füllen. Notwendig zum bestehen ist das konstruieren und füllen mindestens eines Dreiecks.

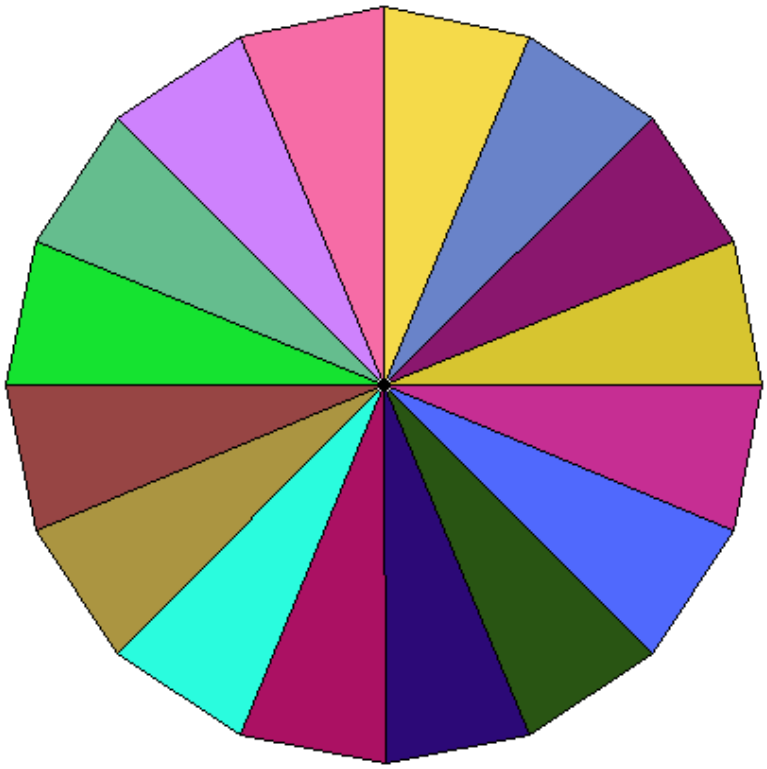

Abbildung 2: Zielsetzung der SeedFill-Implementierung

Ein Beispiel ist in Abbildung 2 gegeben. Die Startpunkte für den Füllalgorithmus werden analog zu den Punkten in Aufgabe 2 erzeugt. Die Skalierung a wird hierbei halbiert, damit die Punkte nicht auf dem äußeren Rand liegen. Zusätzlich werden die Punkte um $\frac{1}{16}\pi$ im Winkel verschoben. Hinweis: Bei komplexen Zahlen, entspricht die Multiplikation einer Rotation. Es wird in diesem Beispiel eine Zahl $o = e^{\frac{1}{16}\pi i} \in \mathbb{C}$ erzeugt. Die Startpunkte s_n ergeben sich somit wie folgt: $s_n = 0.5 * o * z_n$. Die Farben wurden zufällig erzeugt. Hier kam der Pseudozufallsgenerator *std::default_random_engine* mit der Verteilung *std::uniform_real_distribution* im Intervall $[0, 1]$ zum Einsatz. Als *seed* wurde 1 verwendet

Hinweis: Im nächsten Aufgabenblatt wird es unter anderem um die Darstellung der beigefügten Modelle gehen. Versuchen Sie das Laden dieser nachzuvollziehen und informieren Sie sich, welche Attribute Ihnen dann in welcher Form zur Verfügung stehen.