Die Lösungen für diese Übung können bis Sonntag den 26.05.2024 um 18:00h abgegeben werden.

Bézierkurven und Clipping

Praktischer Aufgabenteil (5 Punkte)

Diese Woche gibt es keine Lektion, Ihr findet den gesamten notwendigen Code in `cgintro-5.zip`.

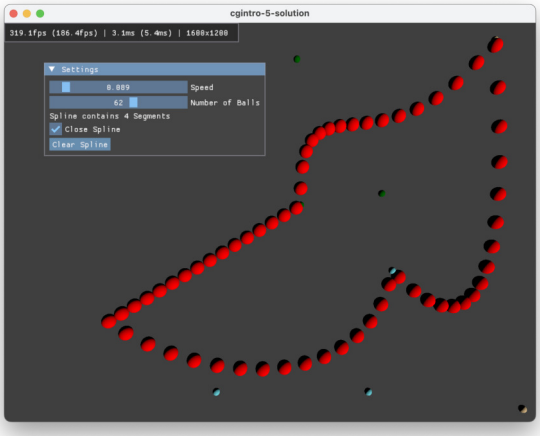

Abbildung 1: Der fertige Spline-Editor

Am Ende der Übung solltet Ihr einen 3D-Spline-Editor zusammengeschrieben haben, bei dem Ihr linksklicken könnt, um Kontrollpunkte hinzuzufügen, und Ctrl-linksklicken, um Stützpunkte hinzuzufügen. Alle diese Punkte sollen in der XZ-Ebene liegen, Eure Aufgabe ist also zuerst den Kamera-Cursor-Strahl mit der XZ-Ebene zu schneiden, um die Koordinaten des Schnittpunktes im World-Space zu ermitteln. Ihr habt dafür in der Funktion `clickCallback`, die bei jedem Mausklick aufgerufen wird, bereits die Koordinaten des Cursors normalisiert gegeben:

```
1  void MainApp::clickCallback(Button button, Action action, Modifier
       modifier) {
2      if (button == Button::LEFT && action == Action::PRESS) {
3          // Convert screen coordinates to viewport coordinates
4          vec2 scale;
5          glfwGetWindowContentScale(window, &scale.x, &scale.y);
6          vec2 screenPos = mouse * scale / resolution * 2.0f - 1.0f;
7          screenPos.y *= -1.0f;
8          screenPos.x *= camera.aspectRatio;
9          ...
10     }
11 }
```

Das Skalieren mit `glfwGetWindowContentScale` ist dabei notwendig, wenn sich die UI-Auflösung von der Auflösung des Framebuffers unterscheidet. In der nächsten Zeile wird die Mausposition dann normalisiert, sodass sie in $[-1, 1]^2$ liegt. Die y-Achse zeigt in Bildschirm-Koordinaten nach unten, deswegen wird diese noch gespiegelt, außerdem müssen wir die x-Achse noch mit dem `aspectRatio` strecken.

1. Generiert aus diesen Koordinaten jetzt einen Kamera-Cursor-Strahl (siehe `raygen.vert` aus vorheriger Übung) und schneidet diesen mit der XZ-Ebene, indem ihr zuerst `t` berechnet und damit `worldPos` ermittelt *(2 Punkte)*.

Der folgende Code fügt `worldPos` nun an einen `std::vector<std::vector<vec3>>` an, der die Spline als Liste von Positionlisten darstellt, wobei jede Positionsliste einem Segment entspricht. Die Funktion `evalSpline` wertet diese Spline nun aus, indem sie die einzelnen Segmente einfach hintereinander hängt und jeweils mit der Funktion `deCasteljau` auswertet. Eure nächste Aufgabe ist jetzt, den Algorithmus von de Casteljau zu implementieren, um beliebige Bézierkurven auszuwerten.

2. Schreibt Euren Code dafür in die Funktion `vec3 deCasteljau(const std::vector<vec3>& points, float t)`, wobei `points` die Liste der Kontrollpunkte der Kurve ist und $t \in [0, 1]$ die Position auf der Kurve. Hinweis: Man kann den Algorithmus rekursiv implementieren, es gibt jedoch auch eine sehr kurze nicht-rekursive Implementation des Algorithmus, die auf dynamischer Programmierung beruht *(3 Punkte)*.

Theoretischer Aufgabenteil (5 Punkte)

1. Betrachtet wird das Polygon wie unten abgebildet. Wendet den Sutherland-Hodgman Polygon-Clipping-Algorithmus für die Schnittkanten c im Bild an. Startet dabei beim Vertex v_0 und wandert dann (gegen den Uhrzeigersinn) entlang der Polygonkanten (dabei sind falls nötig Eckpunkte $s_0, s_1, \ldots$ einzufügen). Die Lösung dieser Aufgabe ist dabei die Ausgabe des Algorithmus in Form einer Liste von Eckpunkten des finalen Polygons *(2 Punkte)*.

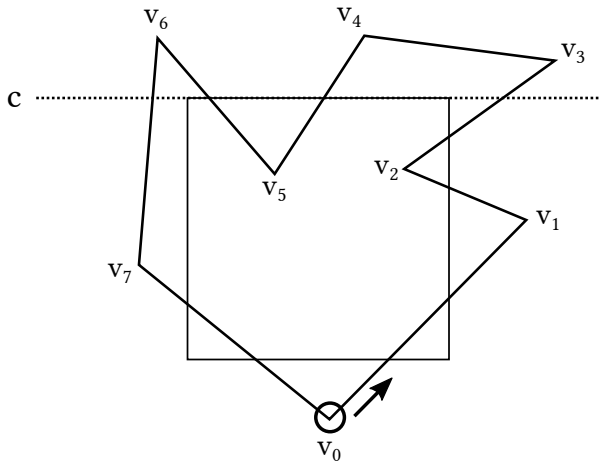

Abbildung 2: Zu clippendes Polygon

2. Konstruiert zeichnerisch durch Anwendung des Algorithmus von de Casteljau den Punkt P, der auf der Bézierkurve mit den Kontrollpunkten $((1,2),(1,0),(3,1),(3,3))$ bei $t = 0.5$ liegt. Gebt die Koordinaten des Punktes an und skizziert den Verlauf der Kurve. Die Zeichnung könnt ihr handschriftlich anfertigen und anschließend in Euer PDF einbetten *(3 Punkte)*.