Grundlagen der C++-Programmierung

Assignment due Wednesday June 20 (23:59)

Assignment 11 - VoxelModel

Ziel dieser Aufgabe ist es, std::unordered_map zu verwenden.

Hintergrund

Wir betrachten eine volumetrische Darstellung von Raumschiffen als Voxel-Bilder, die mit Hilfe eines Octree gespeichert werden. Der Einfachheit halber bleiben wir in 2D und betrachten Voxel in einem Quadtree. (Das Konzept bleibt das gleiche, aber aus Volumenstücken werden Flächenstücke. Wir können uns die Voxel wie Pixel eines digitalen Bildes vorstellen.)

Solche volumetrischen Darstellungen eigenen sich, um große, komplexe Modelle darzustellen und z.B. auf Kollisionen oder Schnitte mit Strahlen etwa für ein ray casting oder Beschuss durch ein virtuelles Projektil zu testen.

Pflichtaufgabe

Nach solch einem Beschuss könnte das Raumschiff *in mehrere Teile zerbrochen* sein. Diese sollt Ihr finden. Dazu ist die Baumstruktur des vorgegeben quadtree aber ungeeignet, weil sie keine Abfragen der Voxel-Nachbarschaft zulässt.

Wir wollen stattdessen das Modell Voxel für Voxel in eine std::unordered_map überführen. Vorstellen können wir uns das wie ein Voxel-Bild, in dem aber nur "ausgefüllte" – also zum Raumschiff gehörende – Voxel tatsächlich gespeichert werden (nicht aber leere Weltraum als "Hintergrund").

Die Map soll als Schlüssel Voxel-Koordinaten vom Typ `IVec2` speichern (siehe `typedef` in `treeprocessor.hpp`). std::unordered_map ist eine Hash-Tabelle. Das heißt, wir müssen – wie in der Vorlesung besprochen – eine Hash-Funktion und einen Test auf Gleichheit angeben.

Nachdem **alle Voxel** des Quadtree **aufgezählt** und in die Map **eingefügt** sind, verwenden wir die Map, um Nachbarschaften von Voxeln abzufragen. Die Nachbarschaft definiert, welche Komponenten des Raumschiffs noch zusammenhängen. (Es wurde ja beschossen.) Diese einzelnen Komponenten werden jeweils wieder in *je einen* eigenen Quadtree eingefügt. Ihr sollt also **zusammenhägende Komponenten** (connected components) ermitteln. Das kann z.B. mit einer Tiefensuche oder einer Breitensuche geschehen. Der zugrunde liegende *Graph* ist durch die Map gegeben: Ein Voxel (als *Knoten*) ist mit seinen Nachbar-Voxeln (über eine *Kante*) verbunden. Wir benötigen keine weitere Datenstruktur, um diesen Graph explizit darzustellen.

Hinweise

- Ihr müsst den Typ `IVec2` (Position eines Voxels) für die Verwendung als **Schlüssel** in einer std::unordered_map vorbereiten.
- Ihr könnt alle Voxel im Quadtree mit Hilfe der `traverse()`-Methode (siehe `quadtree.hpp`) **aufzählen**. Diese nimmt eine Art Funktor, der Euch

als Klasse `FlattenProcessor` (in `treeprocessor.hpp`) zur Verfügung gestellt wird. Ihr müsst dessen `processLeaf()`-Methode implementieren, so dass sie in die Map **einfügt**. (`traverse()` ist die Umsetzung des klassischen visitor pattern mit Hilfe von Templates.)

- **Nachbarschaften** von Voxeln sind 4-Nachbarschaften ("links", "rechts", "oben", "unten"), siehe `main.cpp`.
- Ihr müsst mit Hilfe der Map **zusammenhängende Komponenten finden** und *jede* in eine `QuadTree`-Instanz einfügen. (Es ist hilfreich, eingefügte Teile aus der Map zu löschen!)

Die vorgegebene `main()`-Funktion liest ein Modell ein, zeigt es an und zeigt auch – wenn alles korrekt implementiert ist – die einzelnen Teile an.

Alle nötigen Implementierungen werden von Euch in `treeprocessor.hpp` und `treeprocessor.cpp` vorgenommen.

Nachtrag In der Aufgabe hat sich ein Konzept eingeschlichen das noch nicht in der Vorlesung behandelt wurde. Die Funktion `extractSubModels()` nutzt `std::unique_ptr` zur korrekten Verwaltungen der neuen QuadTrees. Ein Aufruf von delete ist damit nicht notwendig. Im Zweifelsfall lässt sich ein normaler Pointer so in den Container geben: `auto tree = new QuadTree<Voxel>(); subTrees.push_back(unique_ptr<QuadTree<Voxel>>(tree));`