1A) Object Picking (8P)

Ziele:

- GLM kennen zu lernen,
- Sich mit der Vertex Transformation Pipeline zu befassen.
- C++ zu üben.

In dieser Übung ist die Aufgabe, Objekte (Spheres) per Mausklick auswählen & deren Position ändern zu können. Sie verwenden den Sample Code *ex1a_ObjectPicking* als Ausgangspunkt. Wenn Sie das Programm ausführen, sehen Sie, dass schon eine Anzahl Spheres gerendert werden. Die Aufgabe hat somit nichts mit OpenGL / Rendern zu tun, sondern spielt sich einzig auf der CPU ab. Aus diesem Grund ist der Code fürs Erstellen der Geometrie auch aus der *main.cpp* ausgelagert.

Die Problematik ist nun folgende:

Wenn Sie mit der Maus ins Fenster klicken, bekommen Sie 2D Pixelkoordinaten. Die Spheres sind aber über ihr Center und ihren Radius - beides im World Space gegeben - definiert. Zwischen Spheres und den Koordinaten des Klicks liegt sozusagen die gesamte Vertex Transformation Pipeline (minus des Object Spaces).
Der Koordinaten sind 2D, die Sphere welche - vielleicht - selektiert wird kann sich aber auf beliebiger Distanz zwischen der Near und der Far Clipping Plane befinden. Sie müssen die 2D Koordinaten also als Punkt auf einem Strahl betrachten, der bis zur Far Plane der Kamera - und darüber hinaus - reicht, und demzufolge alle Spheres in der Szene auf Intersection mit diesem Strahl testen. Die Sphere, welche 'gewinnt', ist jene, deren Schnittpunkt

- Am nächsten zu Kamera liegt und
- Sich zwischen Near und Far Clipping Plane befindet.

Sie stünden nun theoretisch vor der Wahl: Alle Spheres (also Center + Radius) in den Screen Space zu transformieren, um dort den Intersection Test durchzuführen, oder den Strahl in den World Space zu transformieren. In der Praxis transformiert man in so einem Fall natürlich den Strahl in den World Space, weil dafür viel weniger Berechnungen nötig sind, daher machen Sie das in der Übung auch so.

Der Ray-Sphere Intersection Test an sich ist eine Funktion, welche man sehr oft benötigt, z.B. auch beim Raytracing. Schreiben Sie sich eine Funktion dafür, falls Sie mathematischen Background brauchen, sehen Sie bitte hier nach.

Die Hauptaufgabe liegt in der Konstruktion des Rays im Weltkoordinatensystem. Sie gehen wie folgt vor:

- Es seien (Mx | My) die Koordinaten des Klicks. Erst einmal rechnen Sie diese von SFML nach OpenGL um: In SFML ist (0, 0) links oben, in OpenGL links unten. Die auf OpenGL bezogenen Koordinaten bezeichne ich im Folgenden mit (Sx | Sy).
- Wandeln Sie diese Pixel Position in eine homogene Koordinate um: (Sx, Sy, 0.f, 1.f). Sie brauchen die 4 Komponenten, weil Sie ja mit 4x4 Matrizen transformieren werden. Die Z-Komponente müsste übrigens nicht 0.f sein, sondern könnte irgendwo in Range [0, 1] liegen.
- Bauen Sie die Viewport Matrix und multiplizieren Sie die Screen Space Position mit der Inversen, um in den Normalized Device Space zu kommen. Bedenken Sie, dass GLM Matrizen im *Column Major* Format speichert. (Siehe auch GLM Intro in den Tutorials)
- Multiplizieren Sie die Position im Normalized Device Space mit der inversen Projection Matrix und führen Sie danach die Division durch W durch. Dies entspricht der inversen Projektion, und bringt sie von Normalized Device Space direkt in den Eye Space.

- Multiplizieren Sie die Position im Eye Space als Letztes mit der inversen View Matrix. Dies bringt Sie in den World Space.
- Im World Space ist der Strahl wie folgt gegeben: Eine Position auf dem Strahl haben Sie gerade eben berechnet. Ein zweite Position ist die Kamera selbst, also das erste Argument von `glm::lookAt`.
- Sie können nun alle Spheres auf gegen den Strahl testen. Sollten zufällig mal 2 Spheres identische Intersections ergeben, ist mir egal welche Sie auswählen.

Solange eine Sphere selektiert ist (also die linke Maustaste gedrückt bleibt) und die Maus gleichzeitig über den Screen gezogen wird, soll wird die Sphere entsprechend verschoben werden, und zwar parallel zur Near Plane. Die Distanz zur Near Plane soll sich dabei nicht ändern.

Zum Vergleichen liegt eine ausimplementierte Version als .exe im CG2 Git Repo (ObjectPickingLösung)

1B) Sie laden ein 3D Modell mit AssImp & rendern es. (7P)

Ziele:

- Selber Geometrie definieren, mit VBOs & VAOs
- Die wichtigsten Datenstrukturen unseres Asset Loaders AssImp kennenlernen

In den Tutorials habe ich Ihnen erklärt, wie man einfache Geometrie auf die GPU lädt und darstellt. Bzw. wird Ihnen Manuel Dobusch das auch nochmals auseinandersetzen. Sie werden das Ganze in dieser Übung mit mehreren Meshes & unserem Asset Loader durchführen.

Nehmen Sie *ex1b_AssetLoading* als Ausgangspunkt. Schreiben Sie ein simples Vertex/Fragment Shader Paar (hardgecodete Farbe/keine Transformationen im Vertex Shader) und benutzen Sie die GlslProgram Hilfsklasse, um den Shader zu kompilieren/linken. Die Shader sollten Sie in *bin\shaders* ablegen.

Gehen Sie dann schrittweise vor:

- Als erstes laden Sie die Assets & kopieren sie in eine passende Datenstruktur.
- Als Zweites schicken Sie die Daten auf die GPU.
- Und als Drittes bauen Sie pro Mesh ein Vertex Array Object.
- Zuletzt rendern Sie die Meshes.

Es ist nämlich so, dass die geladenen Assets mehr als ein Mesh beinhalten können. Und die Meshes können durchaus auch mehr als einmal gezeichnet werden, an unterschiedlichen Positionen in der Szene. Dies wird über den sogenannten Szenegraphen gesteuert, den Sie aber fürs erste aber außen vor lassen können.

Das zu ladende File wird immer aus dem CG2 Datenverzeichnis kommen (CG_2015_DATA_DIR) und das konkrete Asset sollte dann über die Eingabeparameter der Anwendung gesetzt werden. Falls Sie nicht wissen, wie man in der MS Visual Studio Umgebung Eingabeparameter setzt: Projekt -> Eigenschaften -> Debugging -> Befehlsargumente, tragen Sie dort das File ein welches sie laden wollen - bzw. ist es im beigelegten Projekt schon eingetragen. Environment Variablen lassen sich mit `std::getenv` abfragen. Ein File mit AssImp zu laden geht so:

```
Assimp::Importer importer;
importer.SetPropertyInteger(AI_CONFIG_PP_PTV_NORMALIZE, 1);
const aiScene *scene = importer.ReadFile(PATH-TO-FILE, aiProcess_PreTransformVertices |
aiProcess_JoinIdenticalVertices | aiProcess_GenSmoothNormals);
if(!scene) return 0;
```

Dem Importer können Sie beim Laden Optionen mitgeben – diese können Sie hier nachlesen. Im obigen Fall sagen Sie AssImp, dass Sie eine indizierte Vertex Liste wünschen, und dass Normalvektoren berechnet werden sollen falls es keine gibt. Weiters sagen Sie dem Importer, dass die Geometrie in den Wertebereich [-1,1] transformiert werden soll, und so umgebaut werden soll, dass der Szenegraph nicht mehr benötigt wird.

Falls die `aiScene` (http://assimp.sourceforge.net/lib_html/structai_scene.html) NULL ist, ist das Laden fehlgeschlagen, höchstwahrscheinlich weil der Dateipfad falsch war.
Die `struct aiScene` gibt Ihnen Zugriff auf jene Datenstruktur, welche Sie zunächst hauptsächlich interessiert, nämlich die Meshes: `aiMesh **mMeshes` und `unsigned int mNumMeshes`. Und ja, es ist eine C Library – daher der Doppelpointer. Es handelt sich dabei um ein Array von `aiMesh` Pointern, wobei Sie die Größe des Arrays aus `mNumMeshes` bekommen.

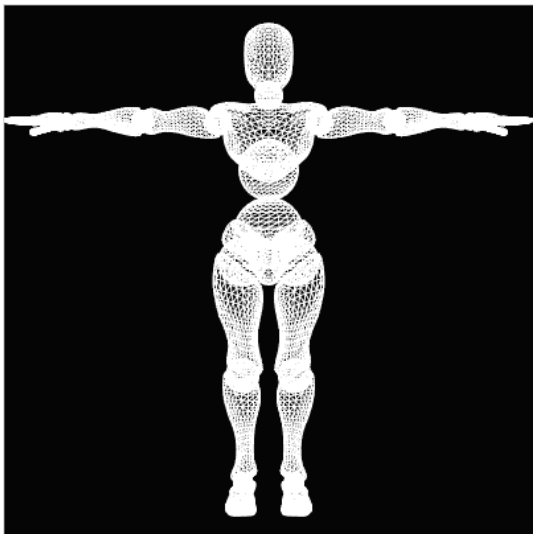
Ähnliche Strukturen werden Ihnen in der Library noch öfter unterkommen. Für jeden der `aiMesh` müssen Sie nun die Vertex Attribute in ein oder mehrere VBOs zu packen, ebenso die Indices, und dann ein Vertex Array Object basteln und damit rendern.

Ein `aiMesh` (http://assimp.sourceforge.net/lib_html/structai_mesh.html) speichert die Vertex Attribute in separaten Arrays: `aiVector3D *mVertices` und `aiVector3D *mNormals`. Falls das jeweilige Attribut vorhanden ist, hat das jeweilige Array genau `mNumVertices` Einträge. Die gute Nachricht ist, dass all diese Attribute bereits in einer Form sind, die man ohne Vorbearbeitung auf die GPU schicken könnte, allerdings sind die Attribute dann nicht interleaved.

Die Indices müssen leider definitiv umkopiert werden. Die Polygone sind in `aiFace *mFaces` gespeichert, und ein `aiFace` (http://assimp.sourceforge.net/lib_html/structai_face.html) speichert die benötigten Indizes im Array `unsigned int *mIndices`, aber leider nicht in einem zur GPU passenden Format (weil `aiFace` einen Pointer als Member hat, mit dessen Adresse die GPU halt nichts anfangen wird). Diese Speicherung liegt daran, dass ein `aiFace` nicht zwangsweise ein Dreieck sein muss. Iterieren Sie also über die Faces, lesen Sie jeweils die ersten 3 Indices aus, und kopieren Sie diese in eine passende Datenstruktur. Sie sollten bei den Assets, die Sie von mir bekommen haben, niemals Faces mit mehr oder weniger als 3 Indices finden, ansonsten können Sie den Import eigentlich als fehlgeschlagen betrachten.

Nach dem Upload bauen Sie das passende Vertex Array Object. Dazu ist nicht viel zu sagen, weil das eigentlich Schema-Code ist: Der Ablauf und die Funktionen, die aufgerufen werden, sind immer sehr ähnlich, und können z.B. in den Tutorials nachgelesen werden, oder auch in den Samples aus CG1. Sie müssen nur die zu Ihrer Geometrie passenden Argumente für die OpenGL Funktionen kennen.

Beim Rendern müssen Sie dann pro Mesh das richtige VAO binden, und mit `glDrawElements` die Geometrie zeichnen. Vergewissern Sie sich, dass auch die Normalvektoren richtig auf der GPU angekommen sind! Ob es geklappt hat, sollten Sie ohnehin sofort erkennen – der Roboter sieht dann z.B. wie Unten gezeigt aus. Testen Sie den Code aber auch mit anderen Assets aus dem Datenverzeichnis.



1C) Szenegraph & Kamera (5P)

Ich gehe davon aus, dass Sie es geschafft haben, das 3D Modell zu laden und darzustellen, andernfalls brauchen Sie mit dem Teil nicht anzufangen. Kopieren Sie alle Ihre C++ Files nach *ex1c_SceneGraph* und fügen Sie sie zum Projekt hinzu; Legen Sie außerdem 2 neue Shader Files an.

Entfernen Sie dann im Code als erstes das Flag `aiProcess_PreTransformVertices` aus den Importer Optionen. Wie schon erwähnt, beinhaltet jede `aiScene` auch einen Szenegraphen, welcher durch das PreTransform-Flag jedoch unterdrückt wurde, mit dem Einstiegspunkt `aiNode *mRootNode`. Dabei handelt es sich um eine Baumstruktur (http://assimp.sourceforge.net/lib_html/structai_node.html), wobei jeder Child Node (`aiNode **mChildren, unsigned mNumChildren`) eine eigene lokale Transformationsmatrix, also relativ zum Parent Node, speichert (`aiMatrix4x4 mTransformation`). Die eigentliche – absolute - Transformationsmatrix eines Nodes bekommen Sie, wenn Sie an der Root beginnend diese Matrizen miteinander multiplizieren.

Die Nodes können für alles Mögliche stehen, aber manche referenzieren einen oder mehrere Meshes (`unsigned *mMeshes, unsigned mNumMeshes`). Die Indices beziehen sich dabei natürlich auf `aiScene::mMeshes`; Das referenzierte Mesh muss dann mit der absoluten Transformationsmatrix des Nodes als Model Matrix gerendert werden. Meshes können ab jetzt mehr als einmal mit unterschiedlichen Model Transformationen gezeichnet werden.

Um die Meshes an der richtigen Position zeichnen zu können, müssen Sie also den Szenegraphen traversieren und die absolute Transformation jedes `aiNodes` berechnen. Da sich diese fürs erste nie ändern (wir wenden keine Animationen an, obwohl einige Assets durchaus welche hätten), können Sie die Transformationsmatrizen prinzipiell direkt nach dem Laden berechnen und speichern. Wichtig hierbei ist noch, dass Sie bitte nicht mit `aiMatrix4x4` rechnen, weil AssImp die Matrizen *Row Major* speichert, der Rest Ihres Codes – auch die GLSL Hilfsklasse, sowie GLSL selbst – aber auf *Column Major* Matrizen ausgelegt ist. Sie verwenden daher die Funktion `toGlm::mat4` aus *common\assimp_and_glm.h*, und wandeln alle Matrizen in `glm::mat4` um, bevor Sie irgendwas ausrechnen. Die Reihenfolge der Matrizen ist dann: Ganz links steht der Root Node, Nodes welche sich tiefer in der Hierarchie befinden kommen weiter rechts.

Im RENDER Abschnitt Ihrer Anwendung traversieren Sie den Szenegraphen ebenfalls und zeichnen die Meshes. Setzen Sie noch eine brauchbare Kamera an – z.B. auf (0|1|8), die nach (0|1|0) blickt und geben Sie im Fragment Shader die Normalvektoren im Eye Space aus.