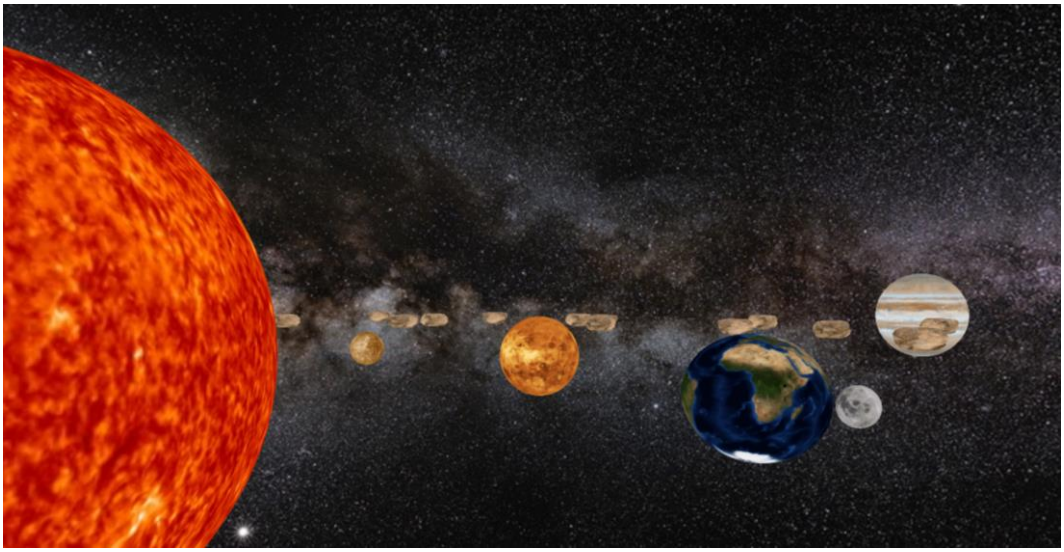


Solar System Voyager

UN VIAGGIO NELL'UNIVERSO CON OPENGL

Federico March | Programmazione ad Oggetti e Grafica | A.A. 2018-2019

Intro

Solar System Voyager è un progetto di grafica 3D che vuole simulare una navicella spaziale volta all'esplorazione del sistema solare.

L'output del programma, infatti, viene presentato come se fossimo a bordo di un'astronave.

Il progetto è realizzato con OpenGl, sfruttando il linguaggio di programmazione C++.

Per realizzare questo progetto sono state necessarie le seguenti classi :

- Camera, che gestisce la creazione, impostazioni e movimenti di camera.
- Model, che permette di importare un modello 3D nel progetto grazie alla libreria Assimp, salvando i dati all'interno di un oggetto della classe Mesh .
- Mesh, che gestisce i dati della mesh in cui vengono salvati modelli 3D importati nel progetto.
- My algebra, libreria che gestisce operazioni matematiche di vettori e matrici .
- Window, che gestisce la creazione di una finestra in cui verrà eseguito il run del programma.
- Shader, che gestisce la creazione di Vertex e Fragment Shader.
- ShaderLinker, che permette di creare un programma di linking per i vari shader programs creati nel progetto.
- Skybox, che permette di creare una cubemap da utilizzare come sfondo, per poter realizzare una scena da esplorare a 360 gradi.
- Texture, che permette di creare e gestire texture nel progetto.
- Audio, che sfruttando la libreria Irrklang, permette di caricare un file audio e di riprodurlo durante l'esecuzione del programma.

A partire da queste classi sono state generate omonime librerie, le quali sono importate e incluse nel progetto.

Il file demo.cpp contiene il metodo main del progetto.

Il progetto presenta una classe aggiuntiva, Planet, in cui vengono gestite rotazione e rivoluzione di ogni singolo pianeta del sistema solare.

CLASSI DEL PROGETTO

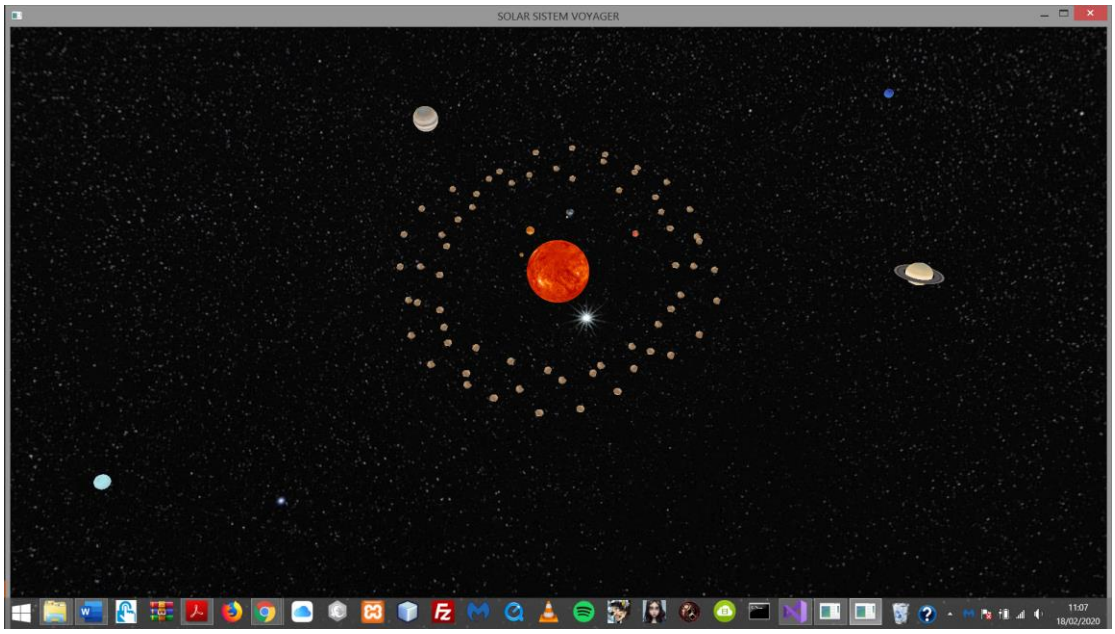
Window

La classe window permette di creare una finestra (con dimensioni inserite dall'utente) in cui apparirà l'output del nostro programma OpenGl.

Il costruttore di questa classe crea una gflw Window, specificando che stiamo lavorando con OpenGL in modalità Core Profile, versione 3.3 di OpenGl ed importando GLAD (libreria avente lo scopo di fornire la locazione delle funzioni di OpenGL che posso essere non note al compilatore).

Inoltre, tale classe possiede particolari funzioni: la funzione Window::attchCamera(),che permette di mettere in relazione la camera di scena con la finestra corrente, ed la funzione Window::processInput()che permette di processare i dati che arrivano in input, come ad esempio, quali comandi devono essere eseguiti quando viene premuto uno specifico tasto della tastiera.

In particolare, premendo le frecce direzionali dalla tastiera sarà possibile spostare la camera (dunque traslare la scena nella direzione opposta) e visualizzare la nuova scena traslata nella finestra corrente.



Camera

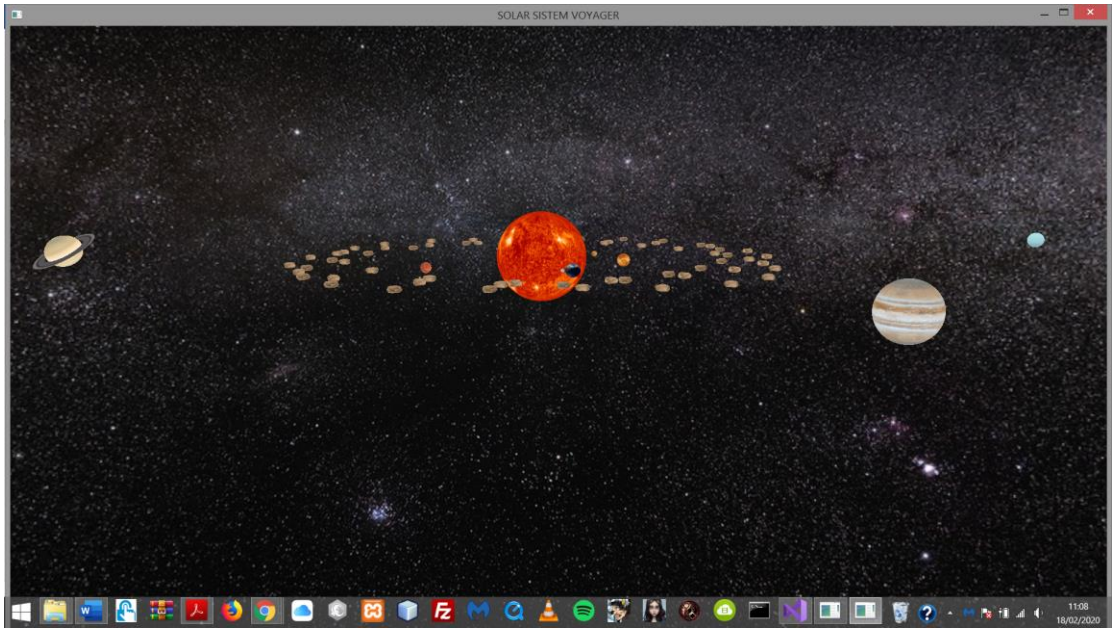
La classe camera presenta

- metodi per le impostazioni di attributi e settings di camera
  (sensitivity, zoom, fov, position, direction, axes (direction z, right x , up y)),
- metodi per le impostazioni di rotazione della camera (yaw e pitch),
- metodi per la gestione di eventi I/O,quali
  1. Camera::processKeyboard() per aggiornare la posizione della camera quando viene premuto una delle frecce direzionali dalla tastiera.
  2. Camera::processMouseMovement() usare il mouse per esplorare la scena a 360 gradi, considerando la posizione del mouse e di conseguenza modificare i valori di pitch(x) e yaw(y) della camera.
  3. Camera::processMouseScroll() per avanzare in avanti o all'indietro all'interno della scena (zoom).

- un metodo per la definizione della View Matrix (grazie alla definizione del metodo lookAt).

La view matrix è definita ricorrendo ai 3 assi locali della camera (direction, right and up) più un vettore di traslazione. Che indica il vettore di posizione della camera.

Grazie alla matrice LookAt, la View matrix trasforma le coordinate World in coordinate del View Space.



Skybox

La classe Skybox permette di creare una cubemap (ovvero una texture 3D definita da un cubo).

Ad ognuna delle facce di questo cubo si associa una propria texture 2D.

Si usa questa cubemap come sfondo della scena, posto ad una distanza "infinita", sia per dare l'impressione di assistere ad una scena che si svolge in un luogo molto ampio come, ad esempio, l'universo , sia per permettere all'utente di poter sperimentare una visione della scena a 360 gradi, usando il mouse come joystick.

La classe Skybox ha come dato di membro un oggetto di tipo Texture (che viene specificato di essere una GL_TEXTURE_CUBEMAP), il quale è neccessario per

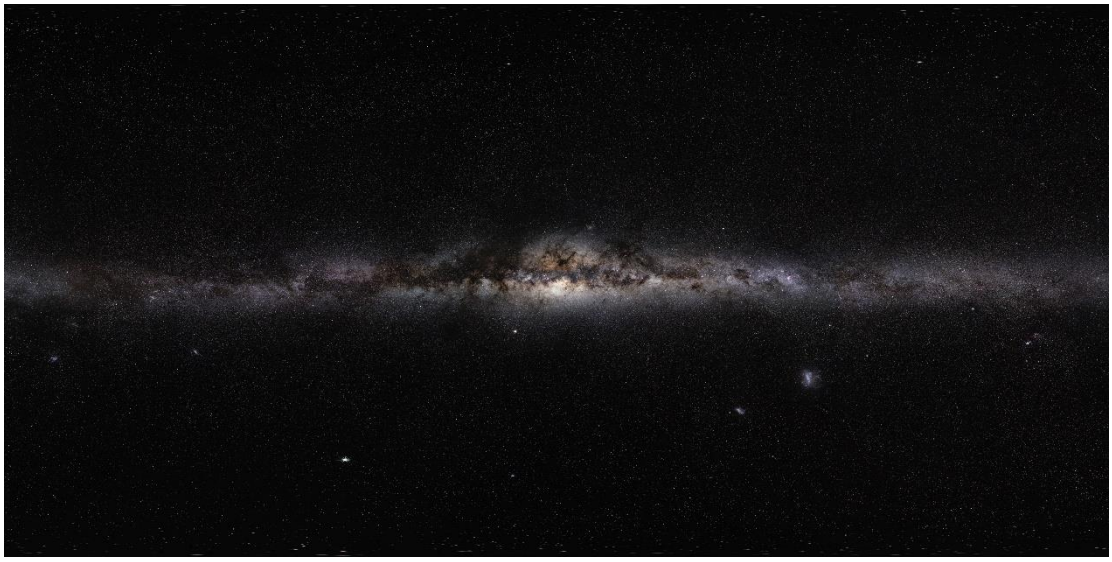
1. Generare i vertici della cubemap.
2. Caricare le oppurtune texture per ognuna della sei facce del cubo.
3. Configurare texture wrapping e texture filtering della cubemap

Il costruttore della classe Skybox imposta i vertex attribute creando un Vertex Array Object (VAO), un Vertex Buffer Object (VBO) in cui vengono copiati i vertici del cubo, ed effettua un Vertex Attribute Linking per specificare le proprietà di cui è composto ogni scomparto del Vertex Buffer Object . Ogni Attribute Pointer è contenuto nel VAO ed ognuno di questi AP viene linkato allo scomparto corrispondente nel VBO.

Il costruttore, inoltre, carica le texture per le rispettive facce del cubo.

Come sfondo della scena è stata caricata una cubemap rappresentante la Via Lattea.

Siamo partiti da un'immagine jpg della Via Lattea, la quale è stata convertita da immagine "Panorama" ad immagine "Cubemap" sfruttando un convertitore on-line, al fine di realizzare sei diverse immagini, una per ogni faccia del cubo.



Per visualizzare la cubemap nella scena è necessario chiamare la funzione Skybox::DrawSB.

Tale funzione abilita un depth test per disegnare la cubemap sullo sfondo, attivando prima la funzione glDepthFunc(GL_EQUAL) che salva nel depth-buffer i valori di profondità del fragment se questo è uguale al valore di profondità salvato nel buffer,

poi viene attivato uno shaderProgram (che sarà lo shader program relativo allo skybox), poi vengono disegnati i 36 vertici del cubo e infine, viene attivata la funzione glDepthFunc(GL_LESS) che salva nel depth-buffer solo i fragmnet che hanno valori di profondità minori o uguali al valore di profondità memorizzato nel depth-buffer corrente (questo servirà per il depth test finalizzato a stabilire quale oggetto di scena starà davanti alla cubemaps).

Mesh

La classe Mesh viene realizzata al fine di salvare i dati che si ottengono dopo aver importato un modello 3D con la libreria Assimp.

La classe Mesh è costituita da

1- una struttura di vertici (in cui sono salvate proprietà dei vertici di un modello, come posizione dei vertici, normali, ecc),
2- Una struttura di indici (serve a specificare con che ordine leggere i vertici salvati)
3- Una struttura di texture (per poter associare la texture o le texture alla mesh corrente).

Ricordiamo che una mesh è una sorta di "maglia poligonale", ovvero un reticolo composto da vertici, spigoli e facce che definisce un oggetto 3D nello spazio .

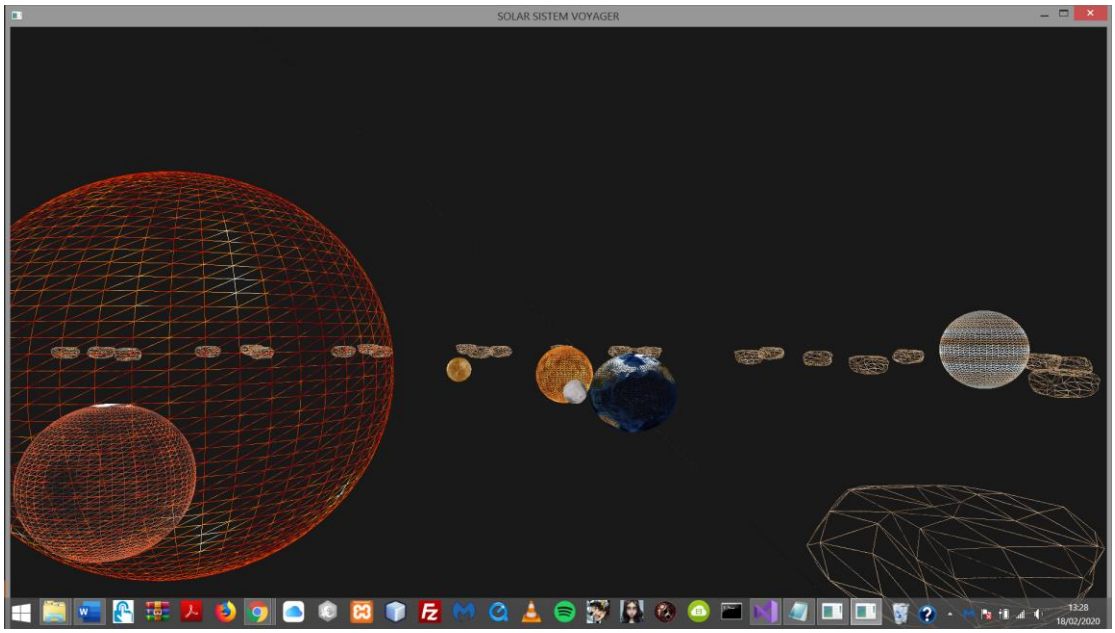
Il costruttore della classe mesh inizializza le strutture dei vertici, indici e texture.

La mesh viene poi impostata nel modo seguente.

Viene creato un Vertex Array Object (VAO), un Vertex Buffer Object (VBO) in cui vengono copiati i vertici che compongono la mesh, un Element Buffer Object (EBO) in cui vengono copiati gli indici necessari a leggere l'ordine con cui vanno rappresentati i vertici.

Gli Attribute Pointer del VAO vengono poi impostati e linkati ai variscomparti di cui si compone il VBO.

La funzione Mesh::draw renderizza la mesh. In questa funzione ogni tipologia di texture viene caricata e bindata, dopo aver ritrovato nel codice la variabile uniform di tipo sampler2D relativa all'unità di texture opportuna.



Model

La classe Model permette di importare un modello 3D nel progetto OpenGl, grazie anche all'ausilio della libreria Assimp.

Una volta che il modello viene importato via Assimp, tale modello viene salvato in un oggetto aiScene, che contiene tutti i dati relativi al modello/alla scena importata.

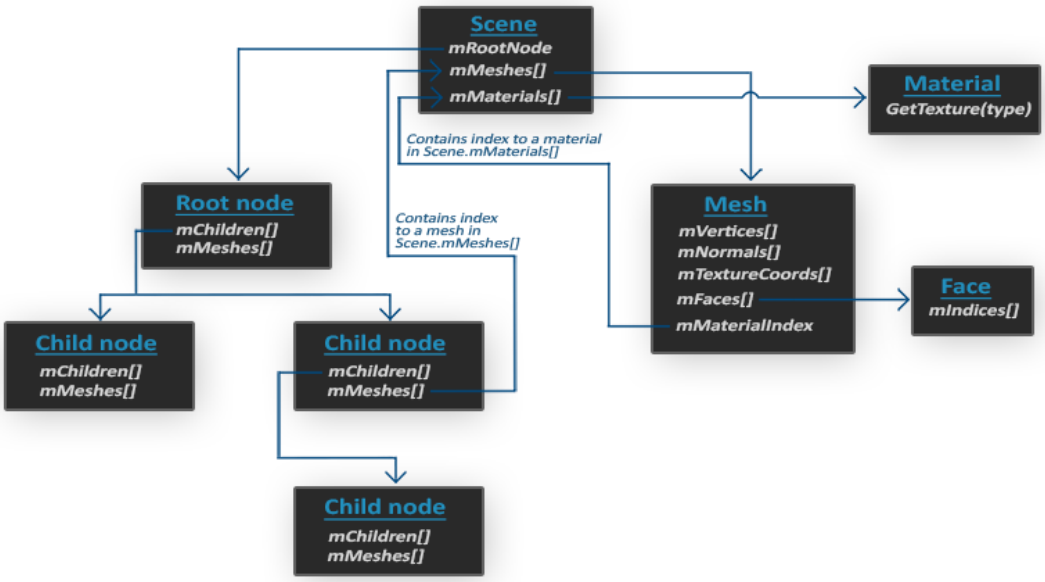
Questo oggetto aiScene contiene tutti i materiali e le mesh della scena, oltre ad un riferimento alla root node della scena.

La root node della scena contiene può contenere children node e una collezione di indici, ognuno dei quali fa riferimento ad una mesh della scena e questa mesh, è contenuta in un array di tutte le mesh della scena chiamato mMeshes.

Ogni Mesh contiene dati relativi ai vertici della mesh, vettori normali, texture coordinates, facce e materialidi cui è composta la mesh.

Ogni mesh può avere molte facce. Ogni faccia è costituita da una primitiva (come triangoli, quadrati, punti…), mentre l'oggetto Materiale ha delle fuznioni specifiche volte a estrapolare le proprietà di un oggetto, come il colore, texture maps…

Tutto questo è l'output risultante che si ottiene dopo aver importato un modello 3D via Assimp. Purtroppo, però, non possiamo utilizzare questi dati ottenuti così come sono in OpenGl. Occorre, quindi, che la classe Model, oltre a caricare il modello 3D via Assimp, svolga anche una sorta di "traduzione" tra ciò che abbiamo ottenuto con Assimp e quella che è la scena 3D in Opengl.



Il costruttore della classe Model prende come argomento di input il filepath in cui è contenuto il modello 3D in formato OBJ.

Una volta che il modello 3D è stato caricato via Assimp, il modello viene memorizzato all'interno di un oggetto aiScene.

Da questo momento si passa a processare ogni singolo nodo della scena, partendo dal root node e percorrendo ogni ramo del diagramma ad albero.

Ogni nodo della scena contiene soltanto degli indici. Come specificato precedentemente, l'oggetto aiScene ha un array mMeshes, contenente tutte le mesh della scena. Per accedere a questo array, abbiamo bisogno degli indici che ci vengono forniti dal nodo aiNode.

Ora, ogni nodo contiene alcune mesh. Ora, per ogni mesh contenuta in ciascun nodo, vogliamo recuperare l'indice IDX di tale mesh nell mMeshes array.

Una volta ottenuto l'indice IDX, recuperiamo la mesh corrispondente all'indice IDX nell'array di meshes mMeshes e si procede a processare la mesh ricavata.

La Mesh che abbiamo ricavato (oggetto aiMesh) deve essere tradotta in termini utilizzabili da OpenGl. Dunque, processiamo l'oggetto aiMesh al fine di salvare il suo contenuto in un oggetto della classe Mesh, vista in precedenza.

Dunque, per ogni vertice della aiMesh ricaviamo la posizione dei vertici, le normali, le texture coordinates (Assimp crea 8 texture coordinates per vertice, ma di queste 8 noi siamo interessati solo al primo set (u,v) e salviamo tutte queste proprietà nella struttura Vertices.

Abbiamo detto che un oggetto aiMesh è costituito da facce, che stabiliscono l'ordine con cui sono organizzati e come devono essere letti i vertici della mesh.

Ora, per ogni faccia dell'oggetto aiMesh ottenuto, ricaviamo l'indice corrispondente alla faccia in questione. L'indice che otterremo, sarà proprio l'indice che ci permetterà di capire in che ordine sono organizzati i vertici della mesh. Salviamo, poi, questi indici nella struttura Indices.

Ogni oggetto aiMesh possiede informazioni sui materiali che caratterizzano la mesh.

Si passa, dunque, a processare ogni materiale che costituisce la mesh, e , quindi, a tradurre le informazioni contenute nell'oggetto aiMaterial in oggetti di tipo Texture. Per ogni materiale, viene caricata una texture grazie alla classe Texture e alcune proprietà di questo oggetto vengono assegnate ai dati membro della struttura Textures.

Ora, le strutture Vertices, INdices e Textures vengono utilizzate per creare un oggetto di tipo Mesh, contenete le informazioni del modello 3D "tradotte" da Assimp in OpenGl.

La funzione Model::Draw() renderizza il modello caricato renderizzando tutte le mesh che costituiscono il modello3D in questione.

Nel nostro caso, ogni modello 3D importato in questo progetto è costituito dai singoli corpi celesti che compongono il sistema solare.

Texture

La classe Texture si occupa di generare, caricare e gestire le texture, che possono essere poi applicate ai modelli 3D.

Tale classe ha come dati membro il target di texture che questa classe deve generare (GL_TEXTURE_2D, oppure GL_TEXTURE_CUBEMAP), ed il filepath in cui si trova l'immagine che verrà applicata come texture.

Il costruttore inizializza un oggetto di tipo Texture, generando una texture del target richiesto come input.

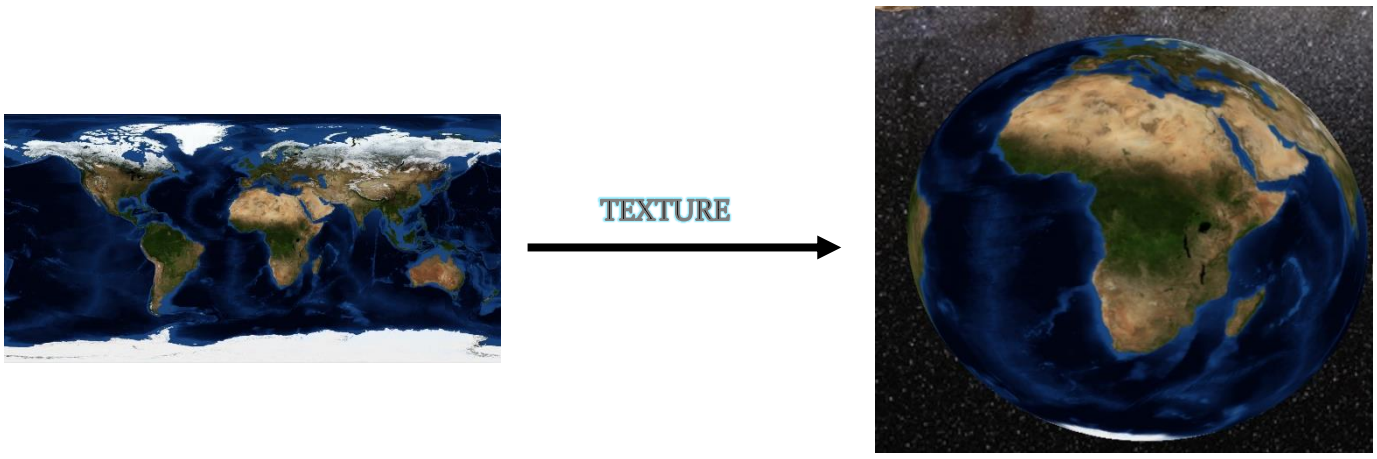
La funzione Tetxure::load(), sfruttando la classe stb_image , carica la texture che si trova nel filepath specificato.

La classe texture presenta delle funzioni per gestire il

1. Texture Wrapping (Texture::texWrapping()), il quale specifica come si deve comportare la texture quando i valori delle texture coordinates cadono fuori dall'intervallo (0,0)x(1,1). In tal caso devono essere applicate delle Wrapping Mode, che consentono, ad esempio, di ripetere l'immagine che viene applicata come texture.
2. Tetxure Filetring (Texture::texFiltering()), con il quale si specifica come OpenGl deve interepretare e mappare un pixel della texture in texture coordinates.

Inoltre tale classe ha metodi per

- Generare vertici di una cubemap (Texture::getSkyboxVertices()).
- Caricare una texture per ogni faccia della cubemap (Texture::cubeMapLoader()).



Shader

La classe Shader si occupa della creazione degli shaders.

Il costruttore di tale classe crea un tipo di shader, a seconda del tipo che viene passato in input e il percorso in cui è contenuto il file in cui è scritto il programma di shading in formato glsl.

Ad esempio, se nel costruttore viene passato GL_VERTEX_SHADER, la classe di tipo Shader creerà un oggetto relativo ad un vertex shader. Analogamente si può ottenere un GL_FRAGMENT_SHADER.

Una volta caricato il programma di shader letto da file attraverso gli stream di input e output, tale programma viene compilato, ed, in caso di successo, il costruttore crea uno shader (glCreateShader) a seconda del tipo di shader richiesto.

Vertex Shader e Fragment Shader, sia per le scene, sia per skybox, vengono generati da questa classe.

Sarà la classe ShaderLinker a mettere in relazione i giusti vertex e fragment shader per generare l'output voluto.

DA RICORDARE che alcune variabili che sono catalogate come variabili di output nel vertex shader sono catalogate come variabili di input nel fragment shader, in quanto il vertex shader riceve i suoi input dal vertex data (che sia un array di vertici definito nel programma o una struttura di vertici di una mesh).

I programmi di shading sono programmi che girano sulla GPU. Il vertex data viene memorizzato sulla GPU grazie al VBO.

Grazie al vertex shader, OpenGL è in grado di capire come interpretare i dati e come inviare i dati alla scheda grafica.

(Tutto questo avviene sulla GPU perché inviare dati dalla CPU alla scheda grafica è un processo estremamente lento, dunque se questo scambio di dati avviene già sulla GPU, allora l'accesso ai dati è immediato)

Una volta processati i dati, il vertex shader produce dati di output. Tali dati di output vengono presi come valori di input dal fragment shader.

ShaderLinker

La classe ShaderLinker crea uno,shader program che combina i vari shader creati.

Per poter usare i vertex e i fragment shader che sono stati creati e compilati, bisogna linkarli ad uno shader program .

Quando si aggiungono vertex e fragment shader ad uno shader program, tale programma crea una connessione tra l'output del vertex shader e l'inout del fragment shader.

Una volta creato questo shader program, tale programma deve essere attivato per poter renderizzare gli oggetti di scena.

Nel nostro progetto creeremo vertex e fragment shader per gli oggetti di scena (i modelli 3D dei corpi celesti che animano il sistema solare) ed uno sceneShaderProgram per renderizzare tali oggetti.

Creeremo poi un vertex e fragment shader per lo skybox ed il rispettivo skyboxShaderProgram per renderizzare la cubemap come sfondo della scena.

Inoltre, la classe ShaderLinker presenta due metodi per ricavare la locazione nel codice di variabili dichiarate come uniform ed inviare il contenuto di tali variabili (matrici) ai programmi di shader (ShaderLinker::getLocAndSendMatToShader()) oppure inviare il contenuto di tali variabili uniform (texture) agli shader (ShaderLinker::getLocAnsSendTex()).

Planet

La classe Planet imposta i parametri relativi alla velocità, raggio dell'orbita che caratterizzano fisicamente ogni pianeta ed il modello 3D associato al pianeta.

Le varie funzioni di cui è costituita questa classe sono finalizzate a

- Calcolare l'angolo di rotazione del pianeta, il quale varia in base alla velocità del pianeta ed in base alla sua vicinanza dal Sole. Più il pianeta è vicino al Sole, più la sua velocità di rotazione sarà bassa, dunque, l'angolo di rotazione spazzato sarà più piccolo.
  Il Sole, invece, è l'unico corpo celeste del sistema solare che gode di un'alta veloctà di rotazione.
  (Planet::getRotAngle()).
- Calcolare lìangolo di rivoluzione del pianeta che è dato dalla sua velocità.
- Al fine di far risultare i pianeti sparpagliati nel sistema solare con diverse posizioni lungo la loro orbita, è aggiunta una quantità al valore finale dell'angolo: 10.of-pow(i,2.739).
  (Planet::getRivAngle()).
- Calcolare la posizione del pianeta lungo la sua orbita. Tale funzione permette di spostare il pianeta lungo la sua orbita. L'orbita del pianeta viene immaginata come una circonferenza che deve essere "disegnata" sul piano XZ, dunque Y=0. Per le formule di trigonometria, segue che la circonferenza sul piano XY è cosi parametrizzata:

  x=radius*cos(RIV);
  z=radius*sin(RIV);

  ottenuti i valori x e z, si crea un vettore t=(x, 0.of, z), che sarà il vettore di traslazione, secondo cui ogni singolo pianeta si sposta lungo la sua orbita.
  (Planet::orbitPosition()).

NOTA

Tale classe non viene caricata ed inclusa nel progetto come libreria per il semplice motivo di facilitare la possibilità di effettuare modifiche sui vari parametri di rotazione e rivoluzione dei pianeti, senza dover ricompilare la libreria ed includerla nuovamente nel progetto.

Main

Il metodo Main per eseguire il programma “Solar System Voyager” è contenuto nel file demo.cpp

Nel metodo main vengono incluse tutte le classi descritte, più altri file di intestazione che permettono di lavorare con OpenGl.

La finestra del programma viene creata come variabile globale, in modo da poter essere visibile all’interno di alcune funzioni che agiscono sul comportamento e contenuto di quest’ultima (la finestra).

Vengono inizializzate alcune variabili globali, come il deltaTime, e le posizioni iniziali del mouse sullo screen (lastX, lastY).

All’interno del Main completiamo la configurazione della finestra su cui verrà passato l’output del programma, in particolare vengono gestiti le funzioni di resize della finestra, prendere le coordinate relative alla posizione del cursore nella finestra, gestire lo scroll del mouse.

Succesivamente vengono, poi, creati i vertex e fragment shader relativi alla scena (sceneVertex/fragmentShader) e relativi allo skybox(skyboxVertex/Fragment).

Ogni shader viene combinato grazie al relativo shaderPogram (sceneShaderPorgram, skyboxPorgram), oggetto della classe ShaderLinker.

Viene caricato il modello 3D di ogni singolo corpo celeste del sistema solare, in formato .OBJ, realizzato con il programma di disegno e modellazione 3D “MAYA”.

Ogni modello è costituito da una sfera (rappresentante il singolo pianeta) a cui viene applicata la relativa texture.

Sono stati importati dal sito Learnopengl.com i modelli 3D relativi agli asteroidi. Per costruire la fascia degli asteroidi, un anello di asteroidi è stato duplicato per due volte e ruotato con angolazioni diverse.

Affinchè ogni pianeta ruoti su se stesso e si sposti lungo l’orbita riservata, nel modello 3D originale, il singolo modello 3D del corpo celeste è stato generato avente centro nell’origine. In questo modo, quando il pianeta è soggetto al moto di rotazione (rotazione intorno al suo asse Y), il pianeta prende come asse y il suo asse Y e non l’asse Y del mondo (o globale, o del centro della scena).

Ogni modello 3D così importato nel progetto via classe Model e Assimp, viene associato ad un oggetto di tipo Planet, che contiene le caratteristiche fisiche di moto del pianeta, oltre al suo modello 3D.

Ogni oggetto Planet rappresenta, dunque, un corpo celeste del sistema solare.

Pertanto, possiamo raggruppare ogni pianeta in un conteiner. In particolare, inseriamo ogni oggetto Planet in un vector<Planet> chiamato solarSystem.

Si genera poi lo skybox, come oggetto della classe Skybox.

Si genera una camera come oggetto della classe Camera e si eseguono dei settings dei vari attributi di camera, che possono essere modificati manualmente dall'utente.

La camera poi, viene, allegata alla finestra del progetto.

Si importa il file audio che verrà riprodotto quando sarà avviato il run del programma.

Si abilitano le funzioni per il depth-test.

Si impostano le varie matrici di trasformazione di coordinate :

- Model (da local to world space)
- View (da world to view or eye space)
- Projection (da view a clipping space). Per la matrice projection si usa la trasformazione di prospettiva, un cui si passano come parametri il camera FOV, Aspect-Ratio, Frustum Near Plane Position, Frustum Far Plane Position.

La matrice planetModel gestisce il movimento di rotazione e il moto di rivoluzione dei pianeti ed è vista come una composizione tra una rotazione ed una traslazione.

La rotazione avviene intorno all'asse y con un angolo pari a Planet::RotAngle(), ovvero l'angolo di rotazione che spazza il pianeta, mentre la traslazione avviene lungo un vettore definito da Planet::orbitPos() che calcola la posizione del pianeta lungo l'orbita in base all'angolo di rivoluzione spazzato dal pianeta.

Viene creata una matrice planetModel per ogni corpo celeste del sistema solare.

Lo sceneShaderProgram si occuperà, poi, di renderizzare ogni pianeta.

Tutte queste informazioni vengono prese considerando solarSystem[i], ovvero l'oggetto Planet in posizione i nel vector<Planet> solarSystem.

Una volta renderizzati i pianeti, si passa a renderizzare lo skybox.

Dal momento che per caricare il modello occorrono tre texture units (necessarie per caricare più texture in un solo modello), lo skybox dovrà avere come riferimento una quarta texture unit.

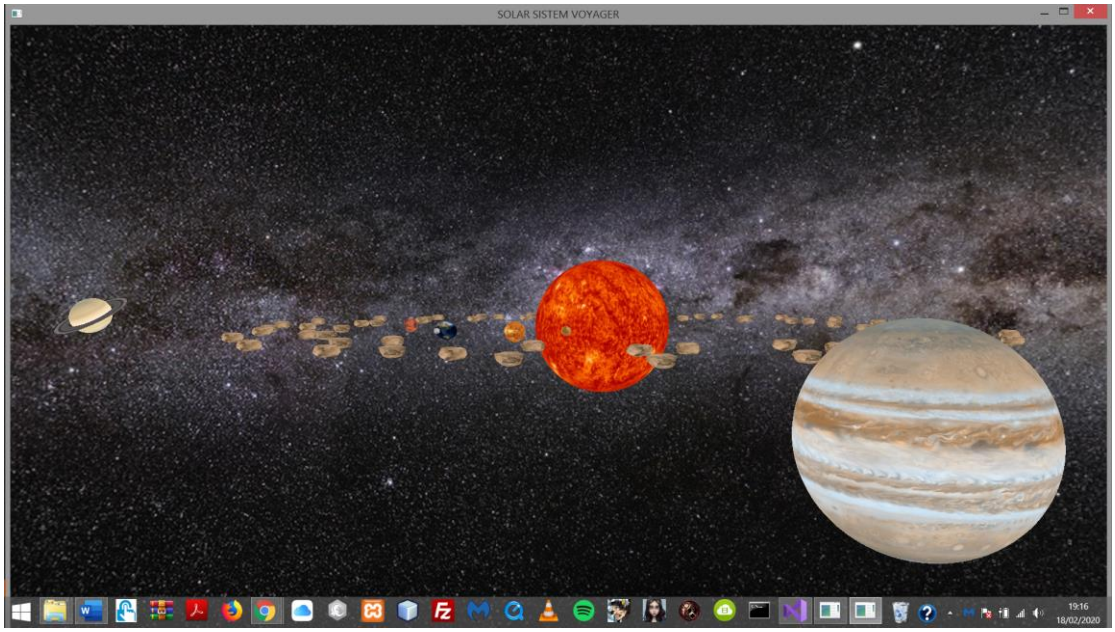
In questo modo, potranno essere renderizzati pianeti e skybox.

Durante il render loop, sia il color buffer che il depth buffer vengono ripuliti, in modo da essere aggiornati con i nuovi valori ad ogni iterazione, in modo da non "sporcare" il viewport con i valori di colore e di profondità calcolati nell'iterazione precedente.

Prima di terminare il programma vengono eliminati tutti gli oggetti che sono stati creati dinamicamente sulla heap, poi, quando l'utente chiude la finestra, il programma viene chiuso.

Durante l'esecuzione del programma è possibile esplorare a 360 gradi il sistema solare, sorvolando i pianeti, orbitare intorno ad un pianeta specifico, volare tra gli asteroidi, e avere l'ebbrezza di potersi allontanare così tanto dal sistema solare, da farlo sparire all'orizzonte nell'infinità della Via Lattea e dell'Universo!

Buon Viaggio a bordo della Solar System Voyager!



(Un frame tratto dal programma)

INDICE