Un esempio è il progetto car.

Il progetto andrà consegnato e poi ci sarà un orale su Teams.

Probabilmente verrà dato più peso al progetto che all’orale. L’orale saranno 1 o 2 domande sugli argomenti del corso.

**Grafica (C.d.S. Magistrale in Informatica)  
Fondamenti di Computer Graphics (C.d.S. Magistrale in Ingegneria Informatica)  
Grafica (C.d.S. Magistrale in Matematica)**

**Progetto A.A.2019/20 assegnato il 22/04/2020**

**!!! Ma vedi versione aggiornata online !!!**

**Obiettivo**

Sviluppare una "3D-WebApp" usando WebGL (HTML5, CSS e contesto webgl) e linguaggio JavaScript + ES SL con browser Chrome.

**Gruppi di lavoro**

Il progetto e' individuale.

**Testo**

Si progetti ed implementi un'applicazione 3D interattiva composta **da almeno un oggetto principale di tipo mesh poligonale caricato da file (formato OBJ Wavefront**). Si definisca **un'opportuna scenografia colorando/illuminando e texturando gli oggetti della scena**. Sono banditi i videogioco/applicazioni denominati ''sparatutto'' e ''labirinti'' e caldeggiati quelli di tipo ''automotive''… dove si fa fare qualcosa alla macchina. Comunque, non è obbligatorio che sia un videogioco.

Nota: noi avevamo una piccola libreria che gestiva il caricamento di mesh da file .obj ma faceva solo poche cose. Ad esempio, caricava solo mesh di faccette al più quadrilatere e caricava solo alcune cose come vertici, facce ecc…

Se troviamo online e scegliamo una mesh complessa e ben fatta che si porta dietro anche tutte delle informazioni come le sue normali, una texture già applicata ecc… Starà a noi aggiungere alla libreria di caricamento .obj

Esistono delle repository di mesh online (es: <https://www.blender-models.com/> )

**Richieste grafiche obbligatorie (prima di consegnare siate certi che abbiate soddisfatto questo)**

 geometria 3D e vista (gli oggetti devono essere visualizzati in proiezione prospettica);

 input utente (si gestisca l'interazione 3D usando sia la tastiera che il mouse e opzionalmente un gamepad);

 illuminazione e sfumatura (gli oggetti 3D devono essere illuminati da almeno una luce);

 texture mapping (almeno due oggetti 3D devono avere una texture applicata e almeno una deve essere una foto dell'autore)

La mia foto posso metterla dove voglio, spazio alla fantasia.

 pannello di controllo su schermo (si preveda un pannello di controllo in cui usando testo e grafica 2D si visualizzino le funzioni utente, ecc)

Fino ad ora abbiamo sempre fatto dei button Html esterni alla canvas. QUI INVECE viene chiesto di realizzare i button con della grafica 2D quindi interni alla canvas, come dei rettangoli con testo ecc…);

Cosa mettere nel pannello di controllo? Ad esempio, in un videogioco è indispensabile un pannello di controllo per mostrare punteggio/tempo di gioco ma anche istruzioni per giocare.

Se invece facciamo un’applicazione web il pannello dovrà essere uno spazio dove spieghi all’utente come fruire e usare quell’applicazione.

 si ponga attenzione che il tutto sia fruibile anche da tablet, smartphone o mobile in genere; QUINDI AD ESEMPIO l’input da mouse con rotella non va messo altrimenti non è gestibile su un touchscreen mentre il click/spostamento del mouse è perfettamente trasportabile nel touch de dito.

Per quanto riguarda la tastiera, per renderla portabile su altri device si potrebbe pensare che negli altri device si mostrino 4 bottoni cliccabili con il touch (ASDW o le 4 frecce).

 advanced rendering (opzionale)(da menu' si preveda l'attivazione/disattivazione di almeno una tecnica di resa avanzata come per esempio: ombre, trasparenze, riflessioni, bump-mapping, ecc.)

**Elementi di Giudizio**

Elementi qualificanti il progetto saranno l'originalita' delle scelte (tipo di applicazione, oggetti, scenografia, texture, ecc.) e la funzionalita' dal punto di vista grafico del codice realizzato. Si rammenti che il progetto e' sulla grafica 3D.

**Consegna**

Si richiede di consegnare un archivio cognome.tgz o .zip (tarato e zippato) contenente due cartelle:

 la prima si chiami "project" e contenga il codice;

 la seconda si chiami "doc" e contenga una relazione in html sul progetto realizzato (descrizione dell'applicazione e suo utilizzo, spiegazione delle scelte effettuate, funzionalita' WebGL utilizzate, particolarita').

Una piccola relazione che abbia una sezione ‘documentazione’ che spieghi ad un utente come fruire dell’applicazione web ma anche una sezione che spieghi le scelte effettuate, come le abbiamo fatte. Qui dobbiamo mettere in evidenza e fare focus soprattutto su quello che ci sembra lo sforzo maggiore fatto o le parti che ci hanno richiesto più tempo / più importanti in modo che il prof possa notarle e valutarle.

L'archivio contenente il progetto puo' essere consegnato per e-mail (come allegato se non troppo pesante) o meglio indicando da dove scaricarlo, almeno 7-10 giorni prima dell'appello d'esame (nel caso venga inviato per e-mail si ricordi di rinominare i file .js in .cg).  
Per il progetto si stimano necessarie almeno 40 ore di lavoro.

**Avvertenza 1**

Non e' vietato "guardare" codice esistente, anzi si caldeggia di farlo, ma per imparare e non per plagiare! Se trovate qualcosa di carino che funziona, ma non capite perche', non lo usate; vi potrebbe essere chiesto di spiegarlo.

**Avvertenza 2**

Non si possono usare librerie aggiuntive fatta eccezione per jQuery.ajax per caricamento e salvataggio di file locali, pena la non accettazione del progetto; si puo' invece riutilizzare tutto quello che e' stato messo a disposizione (per esempio la libreria \***glm\_ligth.js** per il caricamento di file .obj, webgl-utils.js, m4.js, ecc.).

L’obiettivo infatti è fare un progetto dove mettere in campo le cose viste nel corso con gli strumenti visti nel corso.

Quindi l’idea è di non usare altre librerie che fanno cose in più, effetti speciali ecc… Tutte quelle cose concorreranno magari a far aumentare l’originalità del progetto perché altri non le faranno. Ma non essendo richieste non concorreranno ad alzare la valutazione del progetto.

\*glm\_light.js è una libreria che serve a leggere e caricare dei file obj per importare delle mesh.  
Tuttavia, è molto limitata, permette di caricare solo alcun informazioni come vertici, edge e facce, e solo mesh con faccette quadrilatere (e triangolari).  
Fa parte del progetto d'esame estendere questa libreria per far importare anche ulteriori informazioni come i colori, le normali e altre cose che ci possono servire ecc...

**Progetto**

**RELAZIONE**

**Descrizione**

In questo progetto ho sviluppato una Web Application di grafica 3D a tema Carrera Autopodistica.

La [Carrera Autopodistica](https://en.wikipedia.org/wiki/Carrera_Autopodistica) è una competizione che prende luogo nel mese di settembre per le vie cittadine di Castel San Pietro Terme (BO), città in cui vivo. Le macchinine che gareggiano sono dette carrere e sono veicoli senza motore, a spinta umana. Per ogni carrera l’equipaggio è composto da un pilota più quattro spingitori che si danno il cambio nella spinta della macchina realizzando una staffetta. Protagonista della scena dell’applicazione che ho sviluppato è la carrera del Team Volpe, squadra in cui corro.

**Avvio dell’applicazione e su utilizzo**

Prima di avviare l’applicazione è necessario lanciare un server locale che permetta il recupero di risorse cross-origin quali texture e file.obj ??

1. Aprire una shell dei comandi nella cartella /project.
2. Digitare il comando *python -m http.server 8000*
3. Aprire una pagina browser all’indirizzo *localhost:8000*

Una volta avviata, l’applicazione si presenta composta da quattro elementi principali:

* mainCanvas: è il riquadro principale. È un oggetto canvas con contesto webGL che realizza la grafica 3D dell’applicazione. Tramite CSS è stato portato in background in modo che possa stare a tutto schermo senza coprire gli altri elementi.
* textCanvas: è un oggetto Canvas con contesto 2D che realizza il titolo dell’applicazione.
* touchCanvas1 e touchCanvas2: sono due oggetti canvas con contesto 2D su cui viene disegnata l’immagine di un gamepad e che possono essere usati nei dispositivi touch, per sopperire alla mancanza di mouse e tastiera.
* pannello UI: è un oggetto div che funge da menù dell’applicazione.

<img src=”/img/intro.png” />

L’applicazione presenta due differenti modalità d’uso, che possono essere scelte mediante il pannello UI che si trova in alto a sinistra.

**-Modalità scena:** In questa modalità l’utente è libero di navigare la scena per osservarne la composizione e i dettagli.

La navigazione nella scena è realizzata mediante opportuni movimenti della camera. Per maggiori dettagli si veda la sezione MOVIMENTO CAMERA.

La camera può essere spostata avanti/indietro oppure a destra/sinistra (tasti AWSD da tastiera o touchCanvas1 in basso a sinistra) in alto/basso

**-Modalità gara:** In questa modalità l’utente può pilotare la carrera e muoverla all’interno della scena.

Sono disponibili due differenti inquadrature: “visuale spingitore” e “visuale dall’alto” che settano diverse posizioni iniziali della camera.

In modalità gara poi, è possibile simulare il cosiddetto “lancio della carrera”, ossia il gesto compiuto dallo spingitore per permettere il cambio della staffettista. Viene quindi realizzato un incremento di accelerazione cui segue una progressiva decelerazione della macchina. In tutto questo la posizione della camera che gradualmente si ferma, proprio come lo spingitore che smette di correre avendo terminato la corsa.

È possibile pilotare la macchina usando i tasti da tastiera ASWD, oppure utilizzando con il mouse (o con il dito per i dispositivi touch) la touchCanvas1 in basso a sinistra. Il lancio della macchinina può essere comandato premendo la barra spaziatrice o cliccando con il mouse (o con il dito) sulla Canvas principale.

È possibile infine, tramite il pannello UI, settare alcuni parametri addizionali come la sensibilità di movimento della camera e l’attivazione/disattivazione di tecniche di resa avanzate quali le Ombre.

**Struttura del progetto**

Il contenuto della cartella /project è il seguente:

-index.html: è il file html dell’applicazione. Contiene anche del codice Javascript quali le funzioni da eseguire all’avvio e la funzione di render.

-shaders.js: è un file al cui interno sono definiti i vari Vertex Shaders e Fragment Shaders usati nell’applicazione. Sono definiti nel linguaggio GLSL e salvati come variabili stringhe.

È presente anche una funzione, *initPrograms()*, che a partire dai sorgenti degli shaders crea i programmi si salva i puntatori agli Attribute e Uniform di quel programma. initPrograms verrà invocata nel file index.html.

-/data: è una cartella che contiene i file .obj delle mesh presenti in scena e le immagini texture.

-/resources: è una cartella che contiene i file di script .js. In particolare:

* glm\_light\_plus.js (estensione della glm\_light.js)
* subdiv.js
* m4.js
* webgl-lessons-ui.js
* webgl-utils.js

Sono le librerie viste all’interno del corso. Poi vi sono:

* carrera.js: si occupa della fisica della carrera.
* camera-utils.js: gestione del movimento della camera.
* obj-mesh.js: caricamento e disegno di mesh.
* interaction.js: gestione dell’interazione utente.
* shadow.js: gestione delle ombre e funzione di render con ombre.

Sono i file di script che ho realizzato. All’inizio di ciascuno di questi file ho scritto alcune righe di commento a cui rimando per maggiori dettagli sulle loro funzionalità.

Infine, sono presenti la libreria j-query per l’utilizzo del modulo Ajax (vedi sezione Mesh) e un file .css.

Descrizione: Voglio realizzare una scena in cui è presente una carrera (oggetto mesh principale) sopra una strada rettilinea.

Oltre all’auto potrebbe essere presente una fotocamera che ha come target la macchinina e quindi si muove di conseguenza. Si potrebbe mostrare quello che la fotocamera sta inquadrando in una seconda canvas, ed eventualmente dare la possibilità di fermare l’immagine (scattare una foto).

È possibile navigare nella scena facendo muovere la camera (Sia come movimento della sua posizione ma anche muovere il target).

Cliccando un tasto “Gareggia!” nel pannello cambia l’inquadratura passando ad una camera che si trova nel posteriore dell’auto (vista spingitore).

Da questo momento è possibile muovere l’auto e con un tasto simulare il lancio (incremento di accelerazione e progressiva decelerazione con la pos\_camera che si ferma).

Eventualmente:

Aggiungere il cronometro che da il via e prende il tempo della corsa fino all’arrivo.

Curare la scena.

**Descrizione dell’applicazione e suo utilizzo**

**Processo di sviluppo**

PASSO 1: Movimento della camera

Avere una scena minimale, anche composta da un solo piano, in cui è possibile navigare con la camera.

PASSO 2: Importazione dei file mesh: lettura dal file obj e disegno sulla canvas.

Importare la mesh della carrera dal file obj.

Importare altri oggetti in scena.

PASSO 3:

Curare la fisica della macchina: In particolare il suo movimento, la rotazione delle ruote e l’accelerazione con il lancio.

PASSO 4:

Far sì che la fotocamera segua il movimento della macchinina ed eventualmente mostrare le immagini su una seconda canvas (dove poter scattare una foto?)

PASSO 4: Texture

PASSO 5: Illuminazione

PASSO 6: Ombre/Bump Mapping

PASSO 7: Compatibilità con il touchscreen e curare il pannello di controllo

**Mesh**

Nella scena sono presenti diverse mesh

loadMeshObj: Funzione di caricamento asincrono dellamesh.

1.Legge il file .OBJ desiderato.

2.Per ogni copia richiesta per quella mesh:

-crea un oggetto Mesh.

-lo inserisce nell'array contenente tutte le mesh della scena.

@param ncopies: numero di copie che voglio per quella data mesh (NOTA: Il relativo file .OBJ viene letto solo una volta per ovvi motivi di efficienza).

@param meshNames: array con i nomi per ciascuna copia della mesh.

@param filename: path del file .obj da leggere

@param initial\_mo\_matrixes: array con le matrici di posizionamento iniziale per ciascuna copia della mesh

Nella scena sono presenti 3 mesh che ho importato da altrettanti file .OBJ Wavefront:

-carreraMesh

-fotocameraMesh

-highwaySignMesh

Le mesh vengono caricate in modo asincrono all’avvio dell’applicazione sfruttando il modulo AJAX della libreria JQuery.

Per ogni mesh ho seguito il seguente procedimento di importazione:

1. Lettura dal file obj. [1]
2. Creazione di un oggetto Mesh. [2]
3. Inserimento dell’oggetto Mesh in un array contenente tutte le mesh di scena. [3]

[1] Per rendere possibile una lettura adeguata ho modificato la libreria *glm\_light.js* integrando la funzione readOBJ perché oltre a leggere e salvare i dati relativi ai vertici si salvasse anche i dati relativi alle texture e alle normali. Per ogni faccia della mesh, e per ogni suo vertice la readOBJ () si salva gli indici di vertice, di texture e di normale. Tramite gli indici è possibile poi risalire ai dati veri e propri di ciascun vertice, ossia alle coordinate xyz della posizione, alle coordinate uv del mapping texture, oppure alle componenti della normale a quel vertice.

[2] Per rendere più snello e pulito il codice ho deciso di raggruppare in un costruttore operazioni comuni a tutti gli oggetti Mesh, quali la creazione dei buffer WebGL, il bind e il caricamento in essi dei dati, la creazione di texture?? NO… ecc…

Anche la fase di disegno della mesh è stata generalizzata nella funzione drawMesh() ??? NO..

[3] Alla fine di tutti gli import mi ritrovo quindi con un array di oggetti Mesh e per ciascun oggetto (identificabile tramite la proprietà meshName) posso facilmente risalire ai suoi buffer e ad altri dati significativi (es: la sua matrice di posizionamento iniziale)…

Le funzioni relative all’importazione della mesh le ho raggruppate nel file *obj-mesh.loading.js*, eccezion fatta per la funzione readOBJ che si trova, come già puntualizzato precedente, nel file *glm\_light.js.*

Utilizzo di Blender

Quasi tutte le mesh che ho utilizzato, appena scaricate dal web si presentavano in posizioni, dimensioni e orientamenti diversi da quelli che desideravo.

Per limitare le operazioni di trasformazione nel codice della mia applicazione ho importato le mesh originali sul software Blender, tramite il quale ne ho definito la geometria iniziale (centrandole nell’origine degli assi, orientandole nel verso desiderato con delle rotazioni, rimpicciolendole…).

Per alcune mesh (fotocameraMesh e highwaySignMesh) ho sfruttato Blender anche in fase di esportazione del nuovo file .obj, per triangolare le facce che in origine si presentavano quadrate. Questo era necessario in quanto WebGL lavora con mesh a faccette triangolari?? (CONFERMARE guardando la teoria?...)

Per quanto riguarda la Mesh Carrera ho utilizzato Blender per separarla in 3 diverse mesh.

In questo modo mi è stato possibile definire movimenti diversi a seconda dell’esigenza della singola mesh.

Le ruote, infatti, hanno necessità di ruotare intorno all’asse X, la carrozzeria invece no.

Ho ottenuto quindi le seguenti 3 mesh:

-carrozzeria

-ruota Destra

-ruota Sinistra

La geometria iniziale delle ruote e anche della carrozzeria è stata definita con centro nell’origine degli assi in modo da poter apportare rotazioni opportune. Altrimenti l’onere di doverle portare nel centro degli assi si sarebbe dovuto pagare nel codice con conseguente appesantimento delle operazioni di render.

**Texture mapping**

**Interattività**

Eventi touchstart, touchmove, touchend….

**Camera**

L’applicazione permette all’utente di navigare all’interno della scena muovendosi con la camera.

È possibile effettuare dei movimenti traslatori (up, down, left e right) in cui viene traslata sia la posizione della camera che quella del target oppure ottenere un effetto di rotazione della visuale (rotateLeft/Right .. rotateDown/Up) ruotando attorno agli assi della camera il target e lasciando fissa la posizione della camera.

ATTENZIONE: Controlla se è normale che per andare avanti con la moveForwardCamera devo dare degli incrementi positivi anche se la sto spostando verso le Z negative:

Forse perché il forward versor è orientato opposto alla Z e quindi la componente forward\_versor[2] è negativa di suo, quindi dando una dist positiva mi muovo verso le Z negative!?

Le funzioni che ho scritto per gestire tutte le operazioni riguardanti il movimento della camera le ho raggruppate all’interno del file *camera-utils.js* e si appoggiano a funzioni della libreria *m4.js*. Oltre a queste è presente la funzione cameraInit() che permette di inizializzare i parametri della camera e la funzione realign() che invece permette di ricalcolare le giuste direzioni degli assi Xe-Ye-Ze della camera a seguito di rotazioni della stessa.

Il loro ricalcolo ci permette di operare delle traslazioni che siano sempre coerenti con l'orientamento attuale della camera, dando all'utente una sensazione di naturalezza nel movimento.

In un’ottica di portabilità dell’applicazione e di compatibilità con dispositivi touchscreen gli stessi movimenti sono comandabili sia tramite tastiera che tramite…

**Movimento della carrera**

**Illuminazione**

**Sole Mesh**

**Particolarità**

**Ombre**

To render shadows, each time we render a fragment to the *color buffer*, we need to ask the question, “Does this pixel receive direct light from the light source?” If the answer is no, then the pixel needs to be assigned only ambient lighting (or perhaps some lower percentage of diffuse lighting).

To answer the question of which surfaces receive direct light we can place a “camera” at the light source and render the scene. The information we want for each pixel is the distance from the light source to the closest surface.

WebGL does allow you to create a separate frame buffer composed of texture objects and then render an image to this “offscreen” frame buffer. This is actually very advantageous because after we render the scene from the light source’s location, we need to use that data during a normal rendering and a texture map image is a convenient way to lookup values at specific locations.

So, we are rendering the scene twice: The first rendering determines which surfaces receive direct light. The second rendering creates the visual image seen by a user.

Perché questo codice funzioni necessità dell’estensione depth\_texture\_extension

A WebGL extension that allows a texture map to be used as the depth buffer of a frame buffer. This extension also added the option to create a texture map that contains 32-bit unsigned integers for each value of the map.

Riguardo alla tolerance\_constant:

It's kind of working, we can see the shadow of the sphere on the ground but what's with all these funky patterns where there is supposed to be no shadow? These patterns are called *shadow acne*. They come from the fact that the depth data stored in the depth texture has been quantized both in that it's a texture, a grid of pixels, it was projected from the point of view of the light but we're comparing it to values from the the point of view of the camera. That means the grid of values in the depth map is not aligned with our camera and so when we compute currentDepth there are times when one value will be slightly more or slightly less than projectedDepth.

Let's add a bias.

**Fotocamera che segue la macchina**

Se pensiamo alla camera da cui guardiamo la scena come se fosse un qualsiasi oggetto in scena, anch’essa avrà una posizione ben definita nella scena, come tutti gli altri oggetti. Allora possiamo dire che anche la camera avrà una matrice di trasformazione che può essere usata per posizionarla e orientarla nello spazio mondo della scena.

La lookAt matrix è proprio questa matrice di trasformazione.

Ed infatti è la matrice usata per portare il SdR mondo, con origine in un punto detto *target*, a coincidere (sia come posizione che come orientamento degli assi) con il SdR osservatore, avente origine in un punto *camera\_pos* e asse Ze che punta verso *target*.

Cosa succede quindi se sfruttiamo la matrice lookAt come matrice di movimento di una mesh? Essa definirà la sua posizione e il suo orientamento nello spazio mondo a seconda dei parametri *camera\_pos*, *target* (e *up*) specificati.

Definendo come target un punto in movimento, al muoversi del *target* la matrice lookAt ricalcolerà l’orientamento della mesh in modo che l’asse Ze (forward versor della mesh) punti sempre verso l’oggetto.

Ho applicato questa tecnica per animare la mesh “fotocameraMesh” in modo da simulare un fotografo che segue sempre la macchinina in tutti i suoi movimenti.

La matrice lookAt viene calcolata sfruttando il metodo lookAt della libreria m4.js, passando come target il punto [px,py,pz], ossia il centro della carrera, e come pos un punto fisso nella scena in modo che la mesh cambi solo il proprio orientamento ma non la posizione.

**Resize della canvas**

**Pop up di istruzioni all’avvio**

**Render**

(Descrivere il ciclo di render…)

Clip space e resizing della canvas

gl.drawArrays(primitiveType, offset, count);

count mi dice quante volte verrà chiamato il vertex shader.

Because we set primitiveType to gl.TRIANGLES, each time our vertex shader is run 3 times WebGL will draw a triangle based on the 3 values we set gl\_Position to. No matter what size our canvas is those values are in clip space coordinates that go from -1 to 1 in each direction.

Because our vertex shader is simply copying our positionBuffer values to gl\_Position the triangle will be drawn at clip space coordinates

* 0, 0,
* 0, 0.5,
* 0.7, 0,

Converting from clip space to screen space if the canvas size happened to be 400x300 we'd get something like this

* clip space screen space
* 0, 0 -> 200, 150
* 0, 0.5 -> 200, 225
* 0.7, 0 -> 340, 150

WebGL will now render that triangle. For every pixel it is about to draw WebGL will call our fragment shader

In the case above you can see our vertex shader is doing nothing but passing on our position data directly. Since the position data is already in clip space there is no work to do. If you want 3D it's up to you to supply shaders that convert from 3D to clip space because WebGL is only a rasterization API.

I hope you can see that WebGL is actually a pretty simple API. Okay, simple might be the wrong word. What it does is simple. It just executes 2 user supplied functions, a vertex shader and fragment shader and draws triangles, lines, or points. While it can get more complicated to do 3D that complication is added by you, the programmer, in the form of more complex shaders. The WebGL API itself is just a rasterizer and conceptually fairly simple.

When you first create a WebGL context, the size of the viewport will match the size of the canvas. However, if you resize the canvas, you will need to tell the WebGL context a new viewport setting. In this situation, you can use gl.viewport.

The **WebGLRenderingContext.viewport()** method of the [WebGL API](https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API) sets the viewport, which specifies the affine transformation of x and y from normalized device coordinates to window coordinates. Avere una viewport correttamente dimensionata pari all’area di disegno è indispensabile per permettere a webGL di passare dalle coordinate normalizzate del clip space (che vanno da x e y -1 / +1 ) alle coordinate pixel dello schermo!