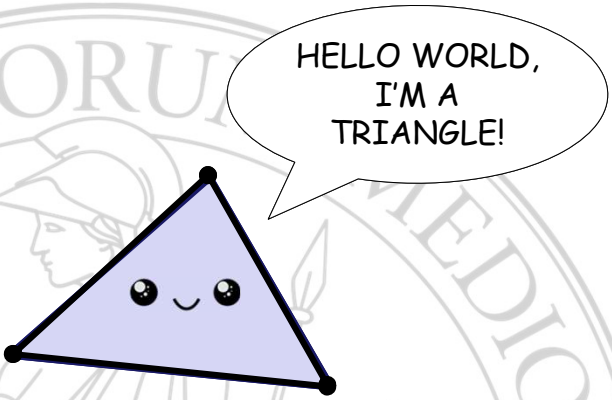
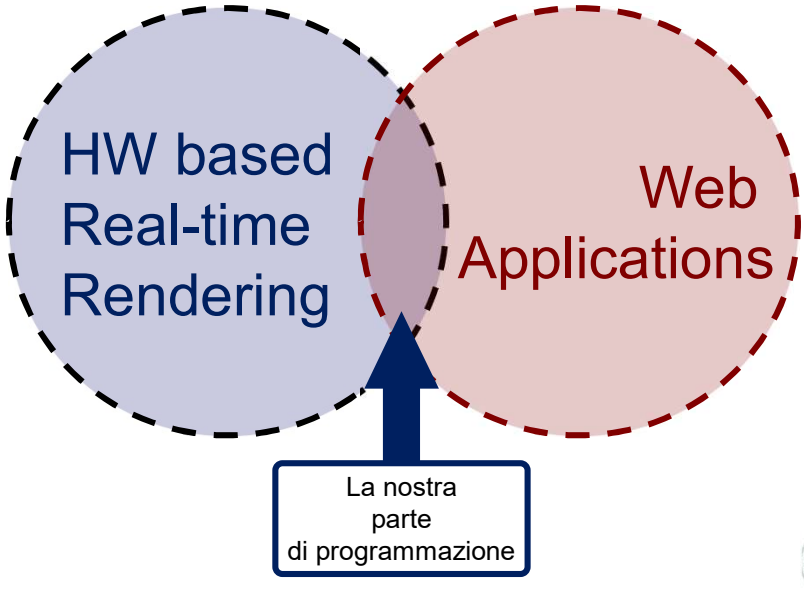
Marco Tarini - Computer Graphics 2022/2023
Università degli Studi di Milano

Hello Triangle (in three.js)



1

Programmazione



2

3D sul the Web: le soluzioni più usate oggi

- ✓ **WebGL** by Khronos (API)
  - ⇒ Graphics Hardware acceleration from browsers
  - ⇒ Javascript based
  - ⇒ Native! – no plugin
  - ⇒ ⊂ **GLES 2.0** ⊂ **OpenGL**
- ✓ **three.js**
  - ⇒ Higher level
  - ⇒ OpenSource, *free*, MIT licensed
- ✓ **Emscripten**
  - ⇒ transpiler: C++ **+** OpenGL → WASM **+** WebGL
- ✓ **Unity** (game engine)
  - ⇒ exports projects as web applications
- ✓ Ad hoc dev-tools:
  - ⇒ **3DHOP** - 3D Heritage Online Presenter
  - ⇒ **Google Earth Engine**
  - ⇒ ...

4

I nostri piccoli progetti

Javascript + HTML page	Applicazione
`three.js`	Libreria
WebGL	API
qualsiasi	Driver
qualsiasi	Scheda video

5

Primo microprogetto – plan of attack
Vedi file: cgLab00 (.html e .js)

1. Costruiamo una paginetta per il nostro programma ← in HTML
2. Perpariamo three.js
3. Prepariamo una mesh in memoria con un solo tri
4. Instanziamo un renderer
5. Mandiamo a schermo la mesh

← in JavaScript + three.js

6

La pagina HTML (esempio)

```
<html>
<head>                                          header
  <title>Hello Triangle</title>
  <style>
        /* qui il css (opzionale) */
  </style>
</head>
<body>                                          body
  <h1>Hello triangle!</h1>
  <canvas  id     = “mioCanvas"
                width  = 500
                height = 500
   ></canvas>
   <script>
         /* qui il JavaScript */
   </script>
</body>
</html>
```

7

Procurarsi three.js

✓Scaricare la versione min da

https://threejs.org/build/three.min.js

8

JavaScript: caricamento della Liberia Three.js e poi uso (inline)

Prima (per es, dentro l'header)…

```
<script src="three.min.js"></script>
```

source

Scaricare ed posizionare nella stessa cartella del file html, oppure specificare il path, oppure anche hot-linking usando il link precedente

Poi (per es, dentro al tag body)…

```
<script>
/* codice JS che usa la lib */
</script>
```

9

JavaScript: caricamento della Liberia Three.js e poi uso (in un file separato)

Prima (per es, dentro l'header)…

```
<script src="three.min.js"></script>
```

source

Scaricare ed posizionare nella stessa cartella del file html, oppure specificare il path, oppure anche hot-linking

Poi (per es, dentro al tag body)…

```
<script src="codice.js"></script>
```

Mio codice JavaScript che usa la lib

10

JavaScript + three.js: inizio

- ✓ Recuperare l'elemento del DOM chiamato "mioCanvas":

```
var ilMioCanvas = document.getElementById("mioCanvas");
```

- ✓ Costuire una classe che avrà tutto lo stato di WebGL e gestirà la pipeline di rendering:

```
var rasterizzatore = new THREE.WebGLRenderer(
                         {canvas: ilMioCanvas} );
```

Tutte le funzioni di rendering sono disponibili come metodi di questa var

Come parametro, specifichiamo che il viewport del rendering è l'elemento del DOM

- ✓ Primo test: cancelliamo lo schermo di un colore prefissato

```
rasterizzatore.clear();
```

Invoca (tramite three.js) la funzione di WebGL che setta tutti i pixel del viewport al colore di cancellazione

- ✓ Opzionalmente, si può prima specificare quale colore vada usato:

Blu scuro (vedi lezione sul colore)

```
rasterizzatore.setClearColor( 0x0000AA );
```

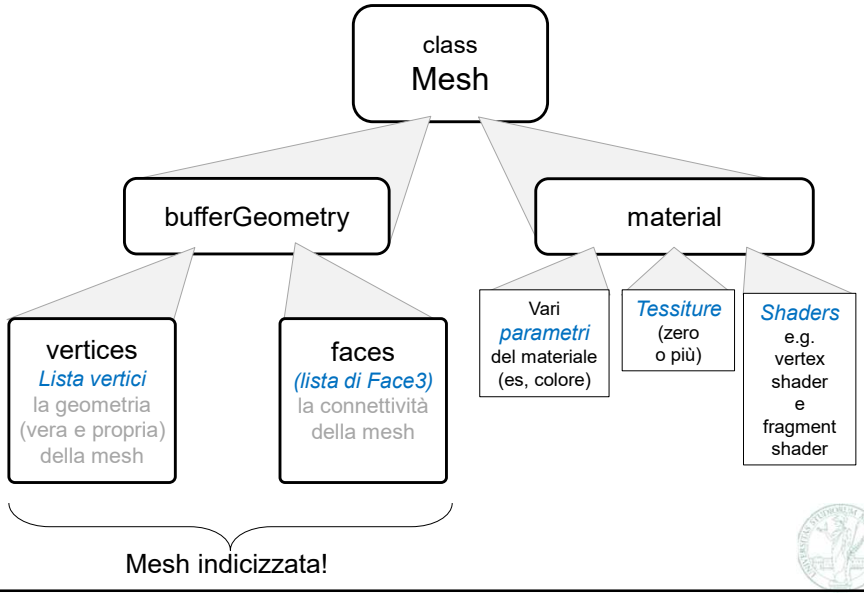
11

JavaScript + three.js: definizione della mesh

- ✓ Definiamo una mesh da renderizzare
- ✓ Come sappiamo, una mesh sono due buffer (due tabelle)
  - ⇒ Uno di vertici, cioè la “geometria”
  - ⇒ Uno di facce cioè la “connettività”
- ✓ In three.js, i due buffer sono in una classe detta bufferGeometry
- ✓ Una Mesh è costituita da un buffer geometry (la mesh stessa) e un materiale, che è una descrizione di tutto quello che serve a disegnare la mesh (vedi dopo)
- ✓ Costruiamo una mesh semplice che contiene solo tre vertici e un triangolo che li connette

12

Struttura per le Mesh su Three.js



13

Costruiamo una mesh di un solo triangolo: Geometria + connettività

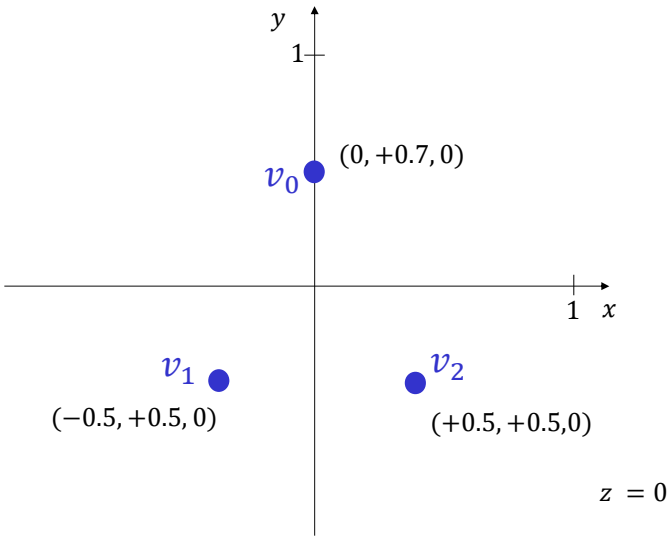
- ✓ Nota: una class punto in three.js è solo un oggetto JavaScript (quindi, espresso in JSON) con tre campi: x, y, z

```
var vertici = [
   { x: -0.5, y: -0.5, z: 0 },   // vertici[0]
   { x: +0.5, y: -0.5, z: 0 },   // vertici[1]
   { x: +0.5, y: +0.7, z: 0 },   // vertici[2]
];

var facce = [ 0 ,1,2,]; // connettività (lista facce)

var geometria = new THREE.BufferGeometry();
geometria.setFromPoints( vertici );
geometria.setIndex( facce );
```

- ✓ Per definizione, la geometria è specificata in spazio oggetto!
  - ⇒ ma visto che M, V e P saranno identità, sono anche già in spazio CLIP

14

In spazio oggetto (ma anche mondo e clip)



15

Costruiamo un “materiale” in three.js

- ✓ In CG, un «materiale» è la una descrizione formale del modo in cui una superficie reagisce alla luce
  - ⇒ Per es, specifica che la superficie sia opaca, o lucida, o rossa, o cangiante…
- ✓ In contesto di programmazione CG, un materiale è la descrizione dello stato del **pipeline** al momento di disegnare una mesh. Quindi:
  - ⇒ i settaggi del rendering (per es, opzioni per il rasterizer),
  - ⇒ le eventuali tessiture da caricare,
  - ⇒ il vertex-program (o -shader) da eseguire per vertice, e uno per fragment-shader
- ✓ Usiamo il materiale più semplice possibile:

```
var materiale = new THREE.MeshBasicMaterial();
```

- ✓ Il materiale corrisponde a queste scelte:
  - ⇒ settaggi: tutti default
  - ⇒ programma per vertice: il «minimo sindacale»: trasforma gli oggetti da spazio oggetto a spazio clip tramite la matrice di Model-View-Projection
  - ⇒ programma per frammento: assegna a tutti i frammenti un colore RGB costante

```
var viola = { r: 1.0, g: 0.0, b: 1.0 };
materiale.color = viola;
```

16

Costruiamo un “materiale” in three.js

- ✓ In CG, un «materiale» è la una descrizione formale del modo in cui una superficie reagisce alla luce
  - ⇒ Per es, specifica che la superficie sia opaca, o lucida, o rossa, o cangiante…
- ✓ In contesto di programmazione CG, un materiale è la descrizione dello stato del **pipeline** al momento di disegnare una mesh. Quindi:
  - ⇒ i settaggi del rendering (per es, opzioni per il rasterizer),
  - ⇒ le eventuali tessiture da caricare,
  - ⇒ il vertex-program (o -shader) da eseguire per vertice, e uno per fragment-shader
- ✓ Usiamo il materiale più semplice possibile:

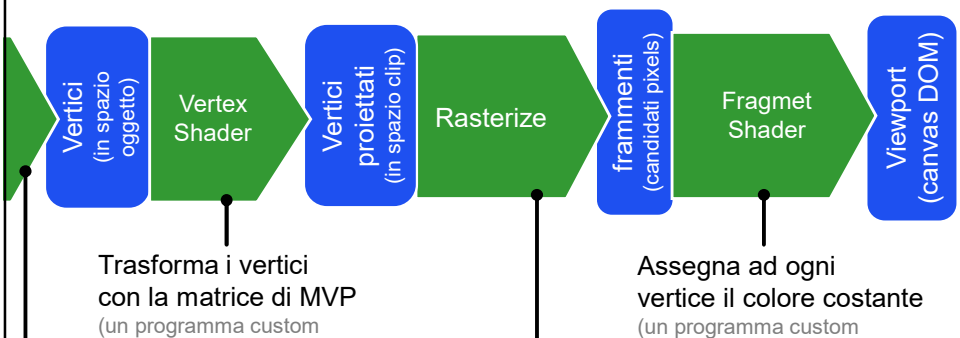
```
var materiale = new THREE.MeshBasicMaterial();
```

- ✓ Il materiale corrisponde a queste scelte:
  - ⇒ settaggi: tutti default
  - ⇒ programma per vertice: il «minimo sindacale»: trasforma gli oggetti da spazio oggetto a spazio clip tramite la matrice di Model-View-Projection
  - ⇒ programma per frammento: assegna a tutti i frammenti un colore RGB costante

```
var viola = { r: 1.0, g: 0.0, b: 1.0 };
materiale.color = viola;
```

17

Cosa fa il materiale «Mesh Basic Material» (cioè: il minimo possibile)



Trasforma i vertici
con la matrice di MVP
(un programma custom
fornito da Three.js
per il materiale dato)

Assegna ad ogni
vertice il colore costante
(un programma custom
fornito da Three.js
per il materiale dato)

Prendi le pos dal buffer dei vertici
come indicizzati dal buffer delle facce
(una funzionalità FISSA dell'HW,
con alcuni settaggi determinati
da Three.js per il materiale dato)

Porta in spazio viewport,
produce i frammenti, etc
(una funzionalità FISSA dell'HW,
con alcuni settaggi determinati
da Three.js per il materiale dato)

18

Rendering

- ✓ L'oggetto di tipo mesh ora può essere costruito:

```
var miaMesh = new THREE.Mesh( geometria , materiale );
```

- ✓ A questa mesh corrisponderà la sua matrice di modellazione **matrice di modellazione** (vedi)
- ✓ La possiamo osservare nel suo campo `miaMesh.matrix`
  - ⇒ Di default, questa matrice è l'identità (quindi, per ora, I due spazi **oggetto** e **mondo** coincidono);

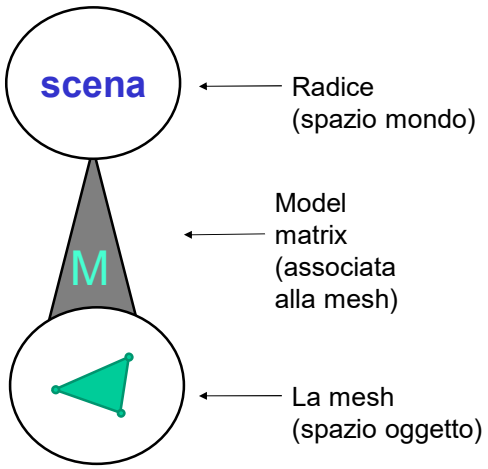
19

JavaScript + three.js: definizione della mesh

- ✓ In three.js, il contenuto è rappresentato in una apposita classe detta “scene”
  - ⇒ Contiene tutti gli oggetti (**mesh**, etc) che compongolo la scena
  - ⇒ Ciascuno di loro è provvisto della propria **matrice di modellazione** che determina il suo posizionamento in spazio mondo
- ✓ Costruiamo una scena semplice che contiene solo la nostra mesh “mono-triangolo”

20

Three.js: scena
struttura ad albero (per ora: un solo oggetto)



21

Three.js: scena e camera

✓ Costruiamo la scena e appendiamo la mesh costruita

```
var miaScena = new THREE.Scene();
miaScena.add(miaMesh);
```

✓ Per disegnare la scena, abbiamo bisogno anche di una camera

```
var miaCamera = new THREE.OrthographicCamera(-1,+1,+1,-1,-1,+1);
```

Modella la (foto) camera.

(Perché questo nome?
è l'inversa della matrice che porta allo spazio mondo dallo spazio vista)

zNear zFar

Contiene: la **matrice di VISTA** (`miaCamera.matrixWorldInverse`)
la **matrice di PROIEZIONE** (`miaCamera.projectionMatrix`)

Qui: è una camera **ortografica** (non **prospettica**: il view frustum è un cubo)
Entrambe le matrici sono l'identità:
la P: perché abbiamo settato i limiti del view frustum come lo spazio clip.
la V: perché non abbiamo settato i parametri estrinseci.

22

Rendering

✓ Ora possiamo instruire il rendering

✓ I pixel prodotti apparianno nel canvas associato a `rastrizzatore`

```
rasterizzatore.render( scena, miaCamera );
```

Ogni elemento della scena include la sua matrice di Modellazione

Include le matrici di Vista e Proiezione

23

Microprogetto 2 – piano
Vedi file: cgLab01 (.html e .js)

Free style:

1. Passiamo da una mesh “mono-triangolo” ad una mesh “mono-quad” (diagonal split)
2. Settiamo matrici di modellazione “a mano”

24

Disegnamo un quad

- ✓ Modifichiamo la geometria e connettività per produrre un quad costitutito da due triangoli (con un diagonal split):

```
var vertici = [
   { x: -0.5, y: -0.5, z: 0 },   // vertici[0]
   { x: +0.5, y: -0.5, z: 0 },   // vertici[1]
   { x: +0.5, y: +0.7, z: 0 },   // vertici[2]
   { x: -0.5, y: +0.7, z: 0 },   // vertici[3]
];

var facce = [
   0,1,2,
   2,3,0   // diagonal split della nostra unica faccia quad!
];
```

- ✓ Nota che l’edge condiviso, 0,2, è percorso in sensi opposti, quindi la nostra minuscola mesh è “ben orientata”
- ✓ Testare cosa succede altrimenti

25

Campo matrix della mesh: settaggi automatici

- ✓ Il campo `matrix` della mesh contiene la model-matrix
  - ⇒ altri campi utili: matrice `modelViewMatrix` che viene automaticamente aggiornato
- ✓ Per manipolare questa matrice in modo intuitivo, Three.js consente di specificare alcuni campi di mesh che determinano la posizione, orientamento e la scala dell'oggetto:
  - ⇒ Scalatura (`scale`)
  - ⇒ Rotazione (`rotation`)
  - ⇒ Traslazione (`position`)
- ✓ Di default, `matrix` viene automaticamente settata in modo da posizionare e orientare l'oggetto nella posizione voluta
- ✓ Cioè, viene settata al prodotto di tre metrici: di traslazione, rotazione, e scaling $M = T \cdot R \cdot S$
- ✓ Come esercizio, a noi ora interessa settare questa la matrice di modellazione "a mano", settando noi i suoi 4x4 = 16 elementi

26

Campo matrix della mesh: esercizio

- ✓ Disabilitiamo questo comportamento di Three.js

```
miaMesh.matrixAutoUpdate = false;
      // altrimenti Three.js la sovrascrive (come descritto sopra)
```

- ✓ E settiamo la matrice di modellazione "a mano"

```
var M = new THREE.Matrix4();

M.set(
   ? , ? , ? , ? ,     // 1ma RIGA della matrice
   ? , ? , ? , ? ,     // 2da RIGA della matrice
   ? , ? , ? , ? ,     // 3za RIGA della matrice
   ? , ? , ? , ? ,     // 4ta RIGA della matrice
);

miaMesh.matrix = M;
```

- ✓ Vedere il codice online per i valori `?`, e provare altre matrici

27