Primo microprogetto – plan of attack
Vedi file: cgLab01.html

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

← JavaScript + three.js

6

La pagina HTML (esempio)

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

head

body

7

Procurarsi three.js

✓Scaricare la versione min da

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

8

JavaScript: preliminare: caricamento della Liberia Three.js

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

source

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

```
<script>
/* comandi che usano la lib */
</script>
```

9

JavaScript + three.js: inizio

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

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

- ✓ Costuire una classe che avrà tutto lo stato di WebGL e gestirà la pipeline di rendering:
  - ⇒ Come parametro, specifichiamo che il viewport di questo rendering sarà l'elemento del DOM

```
var renderizzatore = new THREE.WebGLRenderer({canvas:elem});
```

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

```
renderizzatore.setClearColor( 0x0000BB , 1.0 );
renderizzatore.clear();
```

Blu scuro (vedi lezione sul colore)

Invoca (tramite three.js) la funzione di WebGL che mette tutto lo screen buffer al colore specificato

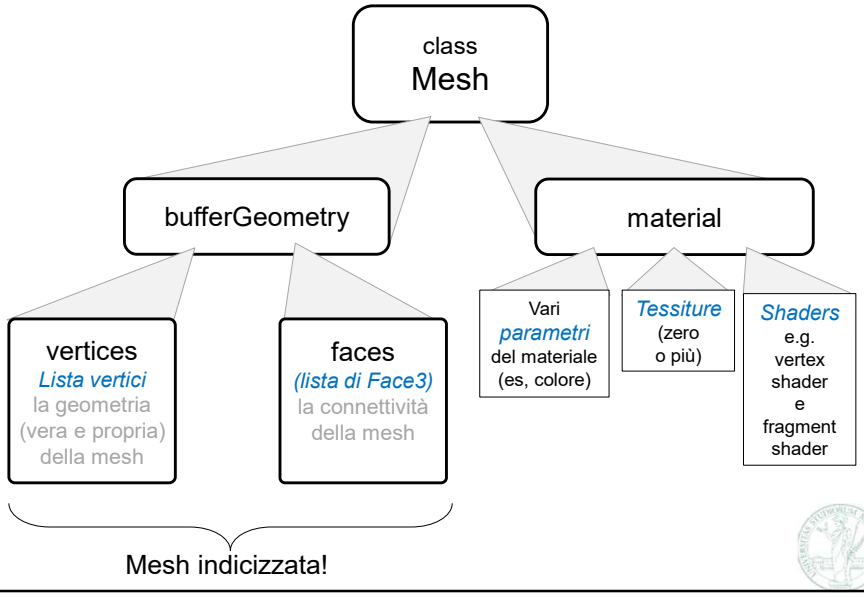
10

JavaScript + three.js: definizione della scena

- ✓ Dobbiamo definire del contenuto 3D da disegnare
- ✓ In three.js, il contenuto è rappresentato in una apposita classe detta "scene"
  - ⇒ Contiene tutti gli oggetti (**mesh**, etc) che compongolo la scena
  - ⇒ Gli oggetti sono tutti espresso nel proprio spazio oggetto
  - ⇒ Ciascuno di loro è provvisto della propria **matrice di modellazione** che determina il suo posizionamento in spazio mondo
- ✓ Costruiamo una scena semplice che contiene una solo oggetto di tipo mesh, che contiene un solo triangolo

11

Struttura per le Mesh su Three.js



12

Costruiamo una mesh di un solo triangolo: Geometria + connettività

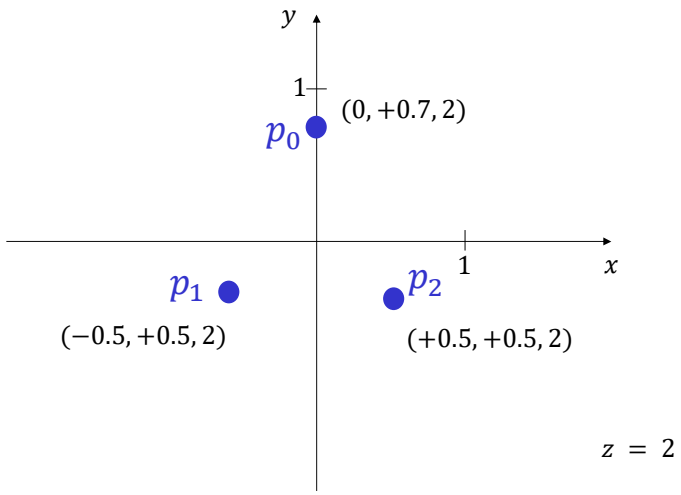
- ✓ Nota: una class punto in three.js
  è solo un oggetto JavaScript (cioè JSON)
  con tre campi: x, y, z

```
var geometria = new THREE.BufferGeometry();

var p0 = { x:+0.0 , y:+0.7 , z:2 };
var p1 = { x:-0.5 , y:-0.5 , z:2 };
var p2 = { x:+0.5 , y:-0.5 , z:2 };

geometria.setFromPoints( [p0,p1,p2] ); // geometria (lista verts)
geometria.setIndex( [ 0,1,2 ] ); // connettività (lista facce)
```

- ✓ Ovviamente, la geometria è specificata in spazio oggetto!

13

In spazio oggetto



14

Costruiamo un “materiale” in three.js

- ✓ Un materiale in CG è
  la descrizione del modo in cui un oggetto reagisce alla luce
  - ⇒ Per es, un materiale può essere opaco, lucido, rosso, cangiante…
- ✓ In contesto di programmazione CG, un materiale è la descrizione dello stato del pipeline al momento di disegnare (ad esempio) una mesh. Quindi:
  - ⇒ i settaggi del rendering (per es, opzioni per il rasterizer),
  - ⇒ le eventuali tessiture da caricare,
  - ⇒ il programma (shader) da eseguire per vertice e per frammento
- ✓ Usiamo il materiale più semplice possibile:

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

- ✓ Il materiale corrisponde a queste scelte:
  - ⇒ settaggi: tutti default
  - ⇒ tessiture da caricare: nessuna
  - ⇒ processo per vertice: il «minimo sindacale»: trasformare gli oggetti da spazio oggetto a spazio clip tramite la matrice di Model-View-Projection
  - ⇒ processo per frammento: a tutti i frammenti creati assegnamo un colore costante: di default, il bianco

15

Rendering

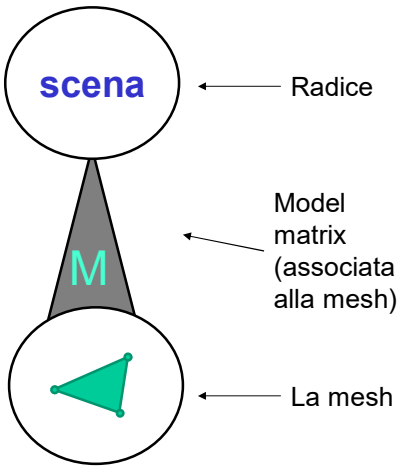
- ✓ La mesh ora può essere costruita:

```
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, gli spazi mondo e oggetto conincidono);

16

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



17

Three.js: scena

✓ Costruiamo la scena e appendiamo la mesh costruita

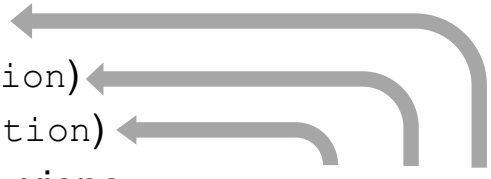
```
var miaScena = new THREE.Scene();
miaScena.add(miaMesh);
```

18

Campo matrix della mesh

✓ Il campo `matrix` di un oggetto contiene la model-matrix di quell'oggetto
  ⇒altri campi utili: matrice `modelViewMatrix` che viene automaticamente aggiornato

✓ Per manipolare la matrice, è possibile specificare separatamente:
  ⇒Scalatura (`scale`)
  ⇒Rotazione (`rotation`)
  ⇒Traslazione (`position`)



✓ Di default, `matrix` viene settato al prodotto di tre metrici: $T \cdot R \cdot S$

19

Camera in three.js: parametri intrinseci

- ✓ Per disegnare la scena, avremo bisogno di una `camera` che rappresenta la macchina fotografica virtuale
- ✓ I suoi parametri intrinseci vengono settati nel costruttore

```
var camera = new THREE.PerspectiveCamera( 60, 1.0, 0.1, 10 );
```

FOV — Aspect ratio (il nostro canvas è quadrato) — Z_near — Z_far

- ✓ Possiamo osservare la **matrice di proiezione** (vedi) che questo produce guardando il suo campo `camera.projectionMatrix`
  - ⇒ Nota: i suoi sedici numeri `camera.projectionMatrix.elements` descrivono la matrice colonna-per-colonna (cioè in «column-major order»): I primi 4 sono la prima colonna, etc.

20

Camera in three.js: parametri intrinseci

- ✓ I suoi parametri estrinseci vengono settati ad esempio da…

```
camera.position.set( 0, 0, 4 ); // setta la posizione (il POV)
camera.up.set( 0, 1, 0 ); // setta l'up vector
camera.lookAt( 0, 0, 0 ); // setta il target position
```

  - ⇒ Vedi lezione corrispondente

- ✓ Possiamo osservare la **matrice di vista** (vedi) che questo produce guardando il suo campo `camera.matrixWroldInverse`
  - ⇒ chiamata così in three.js perché la matrice di vista (che va dal spazio mondo allo spazio camera) è l'inversa della matrice che va da this (la camera) allo sapzio mondo
  - ⇒ Nota: i suoi sedici `elements` sono ancora (come sempre) in «column-major order»

21

Rendering!

- ✓ Possiamo ora eseguire il rendering (su GPU) invocando

```
renderizzatore.render( miaScena, camera );
```

Quale scena (una sola mesh di un solo triangolo)

Vista con quale camera

- ✓ Questo causerà la trasformazione del triangolo
  - ⇒ Usando la matrice di modellazione, vista e proiezione viste sopra
- ✓ Seguita dalla sua rasterizzazione
- ✓ Seguita dal processamento di tutti i frammenti generati
  - ⇒ Per ora, producendo altrettanto pixel bianchi

22

Disegnamo un quad

- ✓ Se ad esempio volessimo disegnare una mesh di un quad solo, cioè costitutita da due triangoli (con un diagonal split):

```
var geometria = new THREE.BufferGeometry();

var vertici = [
        { x:+0.0 , y:+0.7 , z:0 },
        { x:-0.5 , y:-0.5 , z:0 },
        { x:+0.5 , y:-0.5 , z:0 },
        { x:-0.7 , y:+0.7 , z:0 }
];
var facce = [ 0,1,2,  3,1,0  ];

geometria.setFromPoints( vertici );
geometria.setIndex( facce );
```

- ✓ Nota che l'edge condiviso, 0,1, è percorso in sensi opposti, quindi la nostra piccolo mesh è ben orientata

23

Geometria procedurale

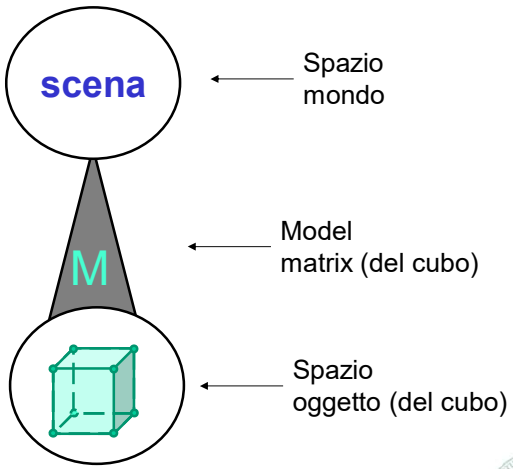
- ✓ Rimpiazziamo la nostra geometria “mono quad” con una mesh di un cubo (un “box”) generato proceduralmente da un’apposite funzione di three.js

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

var geometria = new THREE.BufferGeometry();

var vertici = [
        { x:+0.0 , y:+0.7 , z:0 },
        { x:-0.5 , y:-0.5 , z:0 },
        { x:+0.5 , y:-0.5 , z:0 },
        { x:-0.7 , y:+0.7 , z:0 }
];
var facce = [ 0,1,2,  3,1,0  ];

geometria.setFromPoints( vertici );
geometria.setIndex( facce );

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

```
var geometria = new THREE.BoxBufferGeometry();
```

24

Three.js: scena
struttura ad albero (per ora: di due soli nodi)



25

Animazione in una pagina web, con JavaScript

```
function nuovoFrame(){
   /* fai qualcosa qui ,
      per es, cambia il posizionamento degli oggetti,
      e un nuovo rendering*/

   requestAnimationFrame( nuovoFrame );
}

nuovoFrame();
```

26

Matrici costruite manualmente

La classe `Matrix4` di `three.js` è preposta a rappresentare **matrici di trasformazione**.
Come esercizio, costruiamo manualmente una matrice di rotazione attorno all'asse delle X, settando i suoi 16 valori
(avremmo potuto usare funzioni esistenti di `three.js`)

```
function matriceDiRotazioneX( degrees ){
       var rad = degrees / 180 * Math.PI;
       var s = Math.sin( rad );
       var c = Math.cos( rad );

       var m = new THREE.Matrix4();
       m.set(
              1, 0, 0, 0, // prima riga
              0, c,-s, 0, // seconda riga
              0,+s, c, 0, // terza riga
              0, 0, 0, 1  // quarta riga
       );
       return m;
}
```

27

Matrici costruite manualmente

Assegnamo la nostra matrice come ModelView dell'oggetto miaMesh.

Per far questo, dobbiamo prima disabilitare il meccanismo
che costruisce automaticamente la matrice in funzione dei campi "posizione, rotazione, e scala" dell'oggetto.

```
miaMesh1.matrixAutoUpdate = false;
miaMesh1.matrix = matriceDiRotazioneX( degrees );
```

28

Esperimento con sequenze di matrici

Costruiamo anche una matrice di traslazione,

```
function matriceDiTraslazione( dx, dy, dz ){
  /*...*/
}
```

```
 var T = matriceDiTraslazione( 2 ,0 , 0 );
 var R = matriceDiRotazioneY( time );
miaMesh1.matrix = miaMesh1.matrix.multiplyMatrices( R , T );
```

e sperimentiamo la differenza fra…

```
miaMesh1.matrix = miaMesh1.matrix.multiplyMatrices( R , T );
```

…e…

```
miaMesh1.matrix = miaMesh1.matrix.multiplyMatrices( T , R );
```

29