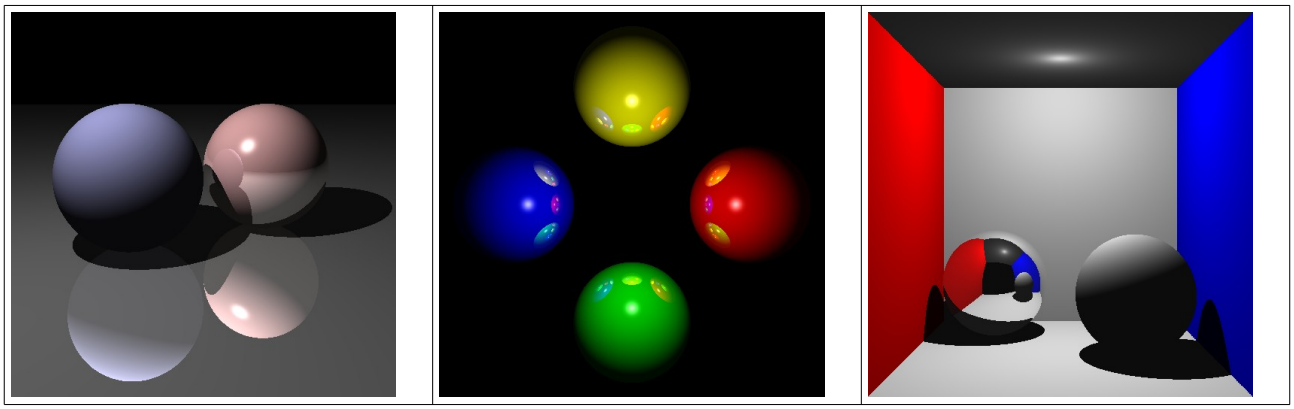
**Progetto finale del corso di Grafica Computerizzata 2018-2019**

Lo scopo del progetto è di sviluppare un semplice ray tracer per renderizzare le scene 3D in immagini 2D. Il metodo di ray tracing può produrre effetti visuali come ombre accurate, riflessione speculare, rifrazione/trasparenza che il rendering realtime tramite la pipeline WebGL non produce se non con tecniche ausiliari.
Al termine del progetto il vostro software sarà capace di:

1. renderizzare sfere e triangoli (e per estensione mesh fatte di triangoli) incluse le superfici trasformate.
2. Impiegare il modello di shading di Phong.
3. Includere illuminazione ambientale, puntiforme e direzionale.
4. Calcolare le ombre proiettate da oggetti che occludono le sorgenti luminose.
5. Calcolare i riflessi speculari.



**Obiettivi**

- essere capaci di implementare gli aspetti matematici e gli algoritmi del ray tracing
  - (includendo luci, ombre, riflessioni e trasformazioni)
- utilizzare i concetti studiati durante il corso come trasformazioni oggetto, trasformazioni vista, illuminazione

I file con il codice iniziale sono contenuti in una cartella zip e contengono un setup dell'elemento di disegno HTML canvas, come i metodi per caricare le informazioni della scena e assegnare il colore ad ogni pixel della canvas.
In particolare da notare che:

- il folder **/assets** contine un numero di file che descrivono le scene da utilizzare come test del vostro codice. In aggiunta nel folder **/examples** trovate le immagini renderizzate di queste scene in modo da poter confrontare le vostre immagini con le immagini fornite.
- In questo progetto dovrete utilizzare una pipeline grafica propria del ray-tracing e non utilizzerete WebGL. Nel folder **/libs** troverete il file **gl-matrix.js** che dovrete utilizzare per il calcolo delle matrici di trasformazione e la matematica dei vettori.

**Impiego del file di scena**

Il vostro codice dovra' renderizzare la scena descritta nel file scene, specificato in formato JSON:

```
{
 "camera": {...},   //  oggetto con i dettagli della camera
 "bounce_depth": 0, // numero di riflessioni della luce
 "shadow_bias": 0.0001, // shadow bias per le ombre
 "lights": [...],    // vettore con sorgenti di illuminazione della scena
 "materials": [...], // array di materiali (valori di riflettanza) per la scena
 "surfaces": [...]   // array di superfici (sfere o triangoli) presenti nella
scena
}
```

- La camera è definita da un vettore posizione "**eye**", un vettore "**up**", un vettore "**at**", il field of view "**fovy**" verticale e il rapporto "**aspect**" tra larghezza (**w**) e altezza (**h**).
- Gli elementi di "**lights**" sono oggetti che rappresentano una diversa luce con il seguente formato:
  - il campo "source" specifica il tipo di luce come stringa: "**Ambient**", "**Point**" o "**Directional**".
  - Indipendentement dal tipo, la luce ha un campo "color" che indica l'intensità della luce emessa. Per semplicità, la luce non ha intensità diverse per le diverse componenti diffuse e specular.
  - In aggiunta, la luce di tipo "Point" ha un campo "position" che definisce la sua posizione nello spazio, mentre la luce di tipo "Directional" ha un campo "direction" che ne definisce la direzione.
- Gli elementi nel vettore "**materials**" sono oggetti che rappresentano i vari materiali. Questi materiali specificano il colore della riflettanza ambientale "**ka**", il colore della riflettanza diffusa "kd", il colore della riflettanza speculare "**ks**" e il valore di "shininess". In aggiunta, i materiali specificano una riflettanza di tipo specchio "**kr**" che indica l'intensità della luce riflessa restituita. Ad esempio, "**kr**" di [0.5,0.5,0.5] indica che il materiale dovrebbe riflettere metà della luce proveniente dagli altri oggetti, mentre un valore "**kr**" di [0,0,0] indica che il materiale non avrà riflessione di tipo specchio.
- Elementi nel vettore "**surfaces**" sono oggetti che rappresentano una diversa superficie con il seguente formato:
  - Il campo "**Shape**" specifica il tipo di superficie o "**Sphere**" o "**Triangle**". In funzione del tipo di superficie, l'oggetto ha campi addizionali:
    - "**Sphere**" ha un "**center**" e un "**radius**",
    - "**Triangle**" ha tre punti: "**p1**", "**p2**", "**p3**".
  - Opzionalmente: ciascun elemento può definire un vettore "**transforms**" che rappresentano le trasformazioni da applicare ad un oggetto.
    - Il primo elemento nel sub-array indica il tipo di trasformata "**Translate**", "**Scale**" o "**Rotate**".
    - Il secondo elemento nel sub-array è un array dei dettagli della trasformazione. "**Translate**" include il vettore di traslazione, "**Scale**" include il vettore di scala, "**Rotate**" include i tre angoli di rotazione intorno all'asse X, all'asse Y e all'asse Z rispettivamente.
    - L'ordine delle trasformazioni indica l'ordine di applicazione nel codice.

**OOP in Javascript**

L'algoritmo di ray tracing è un algoritmo abbastanza semplice concettualmente, ma richiede un buon numero di righe di codice perché lo implementiamo praticamente da zero.

Javascript supporta un tipo di programmazione OOP chiamato prototype-based programming. L'idea di base è che le funzioni sono esse stesse oggetti e possono trasportare con lore le variabili scoped. Quindi noi possiamo definire una funzione

```
var Camera = function(eye,at,up,fovy,aspect) {
      this.eye = eye;
      …
};
```

E dopo possiamo creare una nuova istanza di questa funzione:

```
var cam = new Camera(eyeValue,atValue,…);
```

e ciascuno dei valori assegnati continuerà ad essere in scope, permettendo di accedere ad essere come variabili di istanza.

Inoltre, noi possiamo aggiungere funzioni all'interno dell'oggetto Camera e ciascuna istanza dell'oggetto avrà accesso alle funzioni interne. Per evitare di consumare memoria ridefinedo le stesse funzioni per ogni oggetto, è possibile definire la funzione per il prototipo della classe:

```
Camera.prototype.castRay=function(x,y) {
      …
};
```

Cio' assicura che ogni volta che creiamo un nuovo oggetto Camera, la funzione castRay è definita per il nuovo oggetto.

Tuttavia, poiché Javascript è debolmente tipizzato, è possibile guadagnare una specie di polimorfismo della OOP utilizzando il duck-typing. L'idea di base è che possiamo chiamare un metodo su un oggetto, è sufficiente che l'oggetto abbia il metodo, indipendentemente dal tipo di oggetto. Per esempio:

```
var Sphere= function(...){};
Sphere.prototype.intersects = function(ray){…};
var Triangle= function(...){};
Triangle.prototype.intersects= function(ray){…};

var surfaces= [];
surfaces.push(new Sphere());
surfaces.push(new Triangle());

for(var i = 0;i < surfaces.length;i++) {
      surfaces[i].intersects(ray);//invoca il metodo appropriato sull'oggetto
}
```

Potete consultare il tutorial Mozilla per l'OOP in Javascript. C'e' un numero di classi da considerare per l'implementazione del vostro ray tracer:

- **Camera** - Una classe che rappresenta la camera presente nella scena virtuale. Essa deve mantenere informazioni posizione e orientazione della camera e deve essere responsabile della generazione del viewing ray. Essa può mantenere variabili per generare nuovi raggi senza dovere ricalcolare gli appropriati vettori.
- **Sphere** – Una classe che rappresenta una sfera. E' facile calcolare l'intersezione tra viewing ray e sfere.
- **Triangle** – Una classe che rappresenta un triangolo. L'intesezione con i triangoli è più complicata.
- **Material** – Una classe che rappresenta la riflettanza di una superficie. Questa classe potrebbe implementare il calcolo del modello di shading: dato un punto di intersezione, una normale, una view direction e un elenco di luci nella scena, qual'è il colore del materiale?
- **AmbientLight**, **PointLight**, **DirectionalLight** – Classi che rappresentano le sorgenti di luce. Dovrebbe essere capace di fornire la direzione della luce emessa verso il punto della superficie.
- **Ray** - Una classe che rappresenta un raggio che "lanciato" attraverso lo spazio. Ricorda che i raggi sono definiti dall'equazione **a** + **d***t. I raggi dovrebbero informazione riguardo un valore minimo e massimo di t per i quali le intersezioni sono valide.
- **Intersection** – Una classe che rappresenta un'intersezione tra un raggio e una superficie. Essa dovrebbe mantenere informazione riguardo l'intersezione, il valore di t, il punto di intersezione, la normale,…

**Come organizzare lo sviluppo del codice**

1. La prima cosa da impostare è la Camera in modo che sia possibile "lanciare" i raggi attraverso i pixel. I dettagli su come determinare l'origine e la direzione del raggio li abbiamo discussi a lezione e li trovate sul documento allegato. Praticamente dovete calcolare la viewMatrix utilizzando "eye", "at", e "up" e utilizzarli per calcolare la posizione dei pixel sullo schermo.
   - Poichè specifichiamo il FOV in termini di **fovy** e **aspect** invece che left-right-top-bottom è necessario modificare il calcolo delle coordinate (u,v) nel piano immagine. Assumiamo la lunghezza focale 1, otteniamo:

```
h = 2*Math.tan(rad(fovy/2.0));
w = h * aspect;
u = (w*i/(canvas.width-1)) - w/2.0;   //indice bordo sinistro i=0
                                      //indice bordo destro   i=nx-1
v = (-h*j/(canvas.height-1)) + h/2.0; //come sopra per bottom-top
```

   - I valori di h e w rimangono gli stessi quindi basta calcolarli una sola volta
   - Utilizzate la proiezione prospettica
   - Verificate la matematica del ray casting con i seguenti valori (la camera della scena **SphereTest**):

```
width = 512; height = 512;
ray = camera.castRay(0,0);
//=>o: 0,0,0; d: -0.41421356237309503,0.41421356237309503,-1
//raytracer.js:115
ray = camera.castRay(width,height);
//=> o: 0,0,0; d: 0.4158347504841443,-0.4158347504841443,-1
//raytracer.js:117
ray = camera.castRay(0,height);
//=> o: 0,0,0; d: -0.41421356237309503,-0.4158347504841443,-1
//raytracer.js:119
ray = camera.castRay(width,0);
//=> o: 0,0,0; d: 0.4158347504841443,0.41421356237309503,-1
//raytracer.js:121
ray = camera.castRay(width/2,height/2);
//=> o: 0,0,0; d: 0.0008105940555246383,-0.0008105940555246383,-1
```

2. Il passo successivo è di implementare **Sphere Intersection**. Iniziate impostando il pixel ad un colore costante (esempio: white) se c'e' intersezione e un colore differente (esempio: black) se non c'e' intersezione. Questo dovrebbe permettere di ottenere l'immagine dell'esempio SphereTest.
   - Nota: modificate il codice per includere tutte le superfici e verificate il suo funzionamento con le altre scene.
   - Una volta verificato che l'intersezione funziona (e avete specificato i materiali), potete iniziare con l'implementazione **Sphere Shading**.
3. Aggiungete l'implementazione **Triangle Intersection** procedendo come per la sfera. Questo dovrebbe fornire un'immagine dell'esempio TriangleTest.
   - Una volta verificata l'intersezione, aggiungete il codice per il **Triangle Shading**.
4. Per poter applicare il modello di shading, è necessario specificare le luci e i materiali. Una volta letti i file di scena, e aggiungete le funzionalità per calcolare la luce riflessa dai materiali ad una data locazione (fragment) con una data normale. Impiegate il modello di Phong.
5. Una volta implementate la gestione delle luci e il modello di shading, è possibile implementare Sphere Shading. Calcolate il punto di intersezione, la normale, restituite il colore in base a questi valori e impostate il pixel al colore così calcolato. Se non c'è intersezione impostate il valore al colore black.
   - Questo dovrebbe produrre un'immagine SphereShadingTest1 (per luce puntiforme) e SphereShadingTest2 (luce direzionale).
6. Ripetete l'implementazione per il **Triangle Shading** che potete verificare con TriangleShadingTest.
7. Infine aggiungete le **Transformations**. In questo caso conviene calcolare l'intersezione tra il raggio e l'oggetto trasformato trasformando il raggio nel SdR dell'oggetto.
   - Se M indica la matrice di trasformazione dell'oggetto, per calcolare il raggio nel SdR dell'oggetto utilizzando $M^{-1}$ per rasformare sia l'origine che la direzione, e utilizzate il tale raggio per calcolare l'intersezione con la sfera. Potete invertire la matrice con l'ausilio della libreria glMatrix.
     - L'origine è un punto quindi il termine omogeneo deve essere 1, mentre per il vettore direzione il termine omogeneo deve essere 0.

- Per calcolare il punto di intersezione nel SdR della scena bisogna utilizzare la matrice M. Ricordate che per la normale bisogna utilizzare la normalMatrix $(M^{-1})^T$
- Verificate il funzionamento con TrasformationTest e FullTest

8. Una volta verificato che modelli e luci funzionano, è possibile aggiungere funzionalità addizionali, come Shadows. Quando intersecate una superficie, è necessario lanciare un nuovo shadow-ray verso la luce ( o in direzione opposta alla DirectionalLight). Se lo shadow-ray interseca un oggetto il punto è in ombra e quindi non bisogna includere il contributo di quella luce.
   - Il cast del raggio può impiegare lo stesso metodo utilizzato per calcolare il colore di un punto.
   - Verificate questa implementazione con ShadowTest1 (luce puntiforme) e ShadowTest2 (Directional Light). The CornellBox e FullTest include shadows.
9. Infine aggiungete la **Mirror Reflections**. Questo richiede un ray-tracing ricorsivo: quando trovate un'intersezione, attivate la ricorsione nella direzione della **riflessione** (è necessario che calcoliate voi la direzione). Se il raggio riflesso colpisce la superficie, calcolate il colore in questo punto e aggiungetelo al colore del modello di shading (pesandolo con la riflettanza mirror del materiale).
   - Registrate il numero di **bounces** e terminate quando il numero di **hit** è maggiore la limite imposto.
   - Verificate con il **RecursiveTest**; potete modificare il valore di bounce durante il test. **FullTest** e **CornellBox** includono le riflessione.

Quando alla fine avete implementato tutto avrete un ray tracer.

**Test e Debug**

Un software di ray tracing esegue molte operazioni quindi richiede tempo per la sua esecuzione (visto che lavoriamo in Javascript). In fase di implementazione utilizzate canvas di dimensioni ridotte (300x300). Potete utilizzare la variabile DEBUG per limitare le informazioni di output, inoltre con DEBUG impostato a true c'è una funzione che invoca castRay() attraverso il pixel su cui abbiamo fatto click con il mouse.

```
if(DEBUG) {
      console.log(ray);
}
```

**Consegna**

La data di consegna del progetto è fissata per **Venerdì 18 Gennaio 2019**. Ogni gruppo invii un file progetto_2019.zip o un link al file progetto_2019.zip al progetto implementato contenete il codice sviluppato e commentato, un report che descriva l'implementazione del progetto.