Introduzione alla Generazione di Immagini Fotorealistiche

Corso di Dottorato in Matematica e Informatica
Università degli Studi della Basilicata

Dott. Ugo Erra

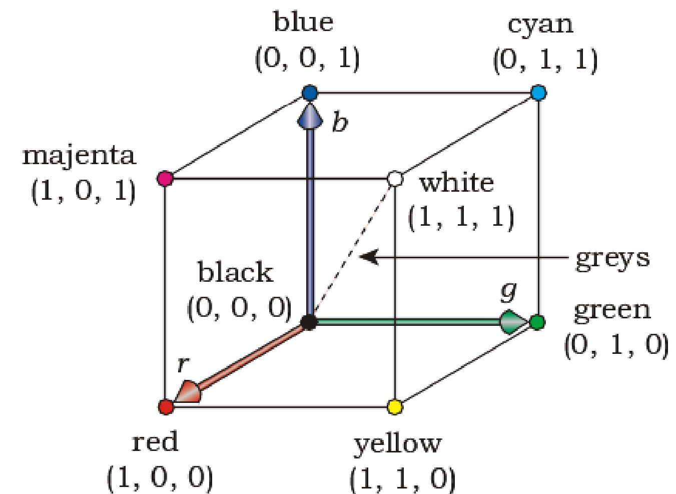
3° Lezione – Ray Casting – II parte

Sommario

- Ray Casting
- Raggi
- Intersezioni
- Un Ray Tracing semplice per viste ortografiche

Rappresentazione del colore

- Il valore finale che ci interessa calcolo è il colore da assegnare al pixel da cui abbiamo generato il raggio
- Per rappresentare il colore utilizziamo una spazio colore 3D tra cui il più comune è RGB
- Nello spazio RGB abbiamo tre assi per definire il rosso, il verde ed il blu (colori primari)
- Un colore è definito mediante sintesi additiva dei tre colori primari $(r,g,b)\in[0,1]^3$



Lo spazio RGB

- Nello spazio RGB possiamo formare lo spettro dei colori visibili
  - rosso+verde+blu=(1,1,1)=bianco
  - rosso+blu=(1,0,1)=magenta
  - rosso+verde=(1,1,0)=giallo
  - verde+blue=(0,1,1)=ciano

La classe RGBColor

- La classe RGBColor permette di definire un colore mediante un terna ordinata (r,g,b) di reali
- Siano $\boldsymbol{c}$, $\boldsymbol{c}_1$ e $\boldsymbol{c}_2$ tre colori ed a e p valori reali,le operazioni possibili sono:

Operation	Definition	Return Type
$\boldsymbol{c}_1+\boldsymbol{c}_2$	$(r_1+r_2, g_1+g_2, b_1+b_2)$	RGB color
$a\boldsymbol{c}$	(ar, ag, ab)	RGB color
$\boldsymbol{c}a$	(ra, ga, ba)	RGB color
$\boldsymbol{c}/a$	$(r / a, g / a, b / a)$	RGB color
$\boldsymbol{c}_1=\boldsymbol{c}_2$	$(r_1 = r_2, g_1 = g_2, b_1 = b_2)$	RGB color reference
$\boldsymbol{c}_1*\boldsymbol{c}_2$	$(r_1 r_2, g_1 g_2, b_1 b_2)$	RGB color
$\boldsymbol{c}^p$	(r^p, g^p, b^p)	RGB color
$\boldsymbol{c}_1+=\boldsymbol{c}_2$	$(r_1 += r_2, g_1 += g_2, b_1 += b_2)$	RGB color reference

Gamma

- Il *gamut* (o gamma) è la gamma dei colori che un dispositivo di output è in grado riprodurre o catturare
  - Generalmente è un sottoinsieme dei colori visibili
- Nella CG i colori vengono trattati come se fossero lineari
  - Sommare due mezzi rossi porta ad un rosso pieno
- La risposta dei monitor non è generalmente lineare
  - Raddoppiare il voltaggio non porta ad ottenere un colore più luminoso del precedente

Correzione gamma

- La luminosità del monitor è modellata come

$$\text{Intensità} = v^{\gamma}$$

  dove:

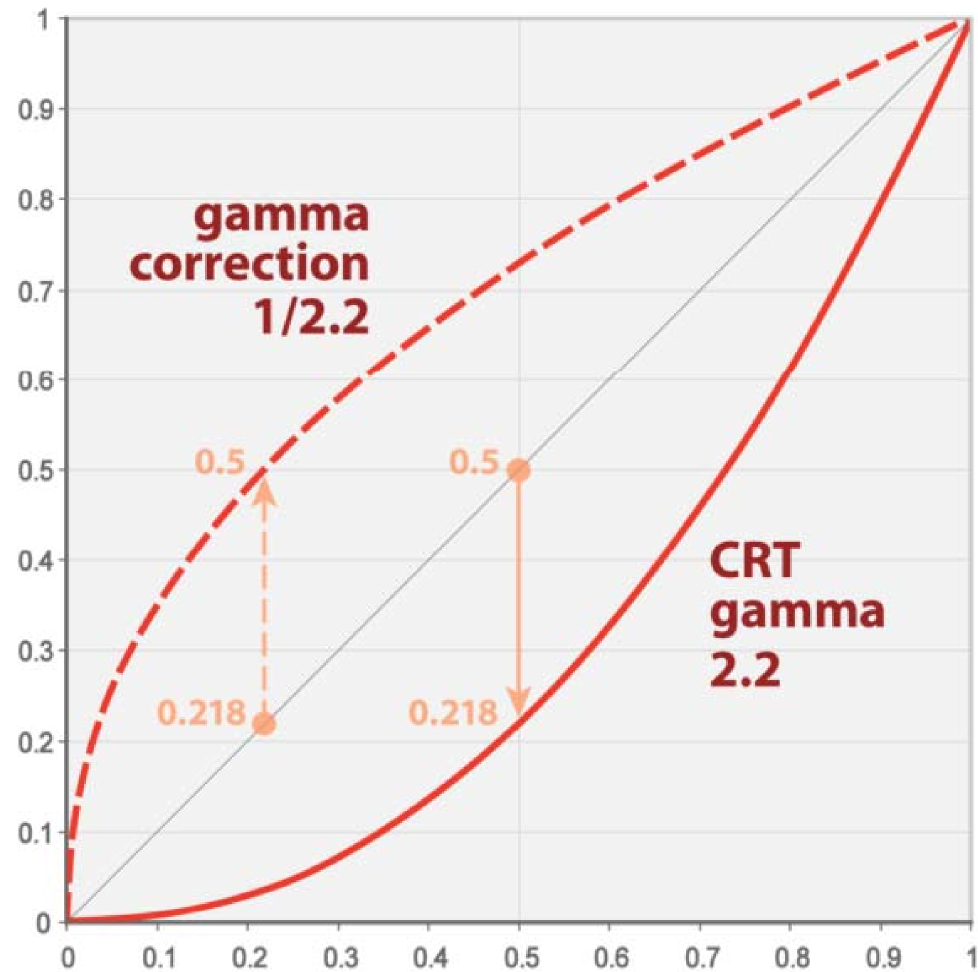
  v è il voltaggio

  γ è il valore gamma del monitor (sui PC = 2.2, sui MAC =1.8)
- Da questo modello si deduce ad esempio che il colore (255, 255, 255) apparirà luminoso più del doppio rispetto a (125,125,125)
- Dato un colore (r,g,b) la *correzione gamma* permette di “aggiustare” l’intensità in modo da avere una risposta lineare usando la seguente conversione

$$(r^{\gamma}, g^{\gamma}, b^{\gamma})$$

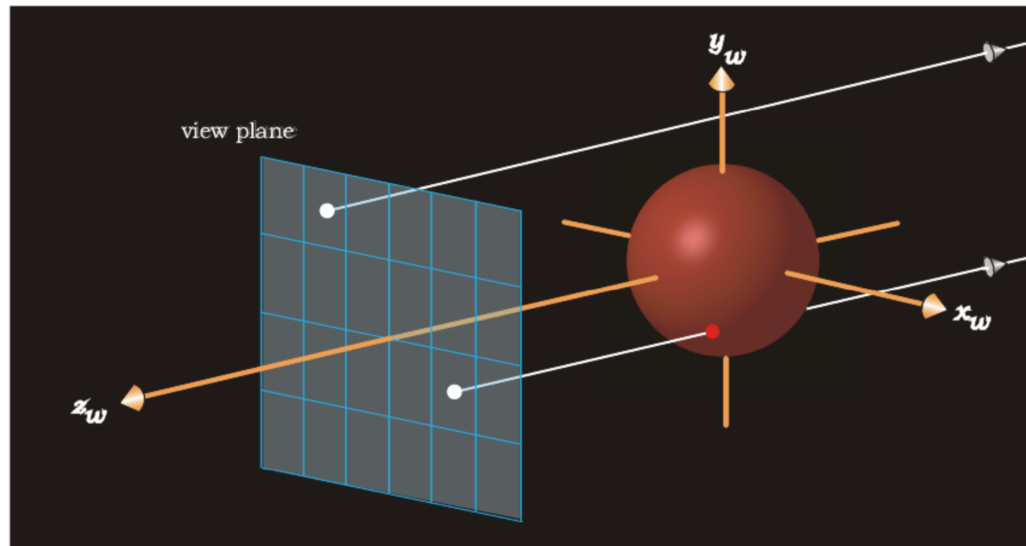
  introducendo quindi un termine $1/\gamma$ che annulla l’effetto gamma

Correzione gamma



Un Ray Casting semplice

- Vogliamo implementare il rendering di una scena composta da:
  - Un viewplane
  - Una sfera rossa centrata nell'origine degli assi
  - Uno sfondo nero



Di cosa abbiamo bisogno?

- Una classe GeometricObjects per gestire gli oggetti presenti nella scena
  - Da cui abbiamo già derivato la sottoclasse Sphere
- Una classe ViewPlane per gestire le informazioni sulla dimensione del viewplane e sulla dimensione dei pixel da cui poter generare i raggi
- Una classe Tracer che gestisca l'intersezione dei raggi generati
  - Da cui deriveremo la classe SingleSphere per gestire l'intersezione dei raggi con la singola sfera
- Una classe World che gestisca gli elementi presenti all'interno della scena e generi i raggi all'interno della scena

La classe ViewPlane

- La classe ViewPlane mantiene le seguenti informazioni:
  - Risoluzione orizzontale
  - Risoluzione verticale
  - La dimensione del pixel
  - Il valore gamma del colore

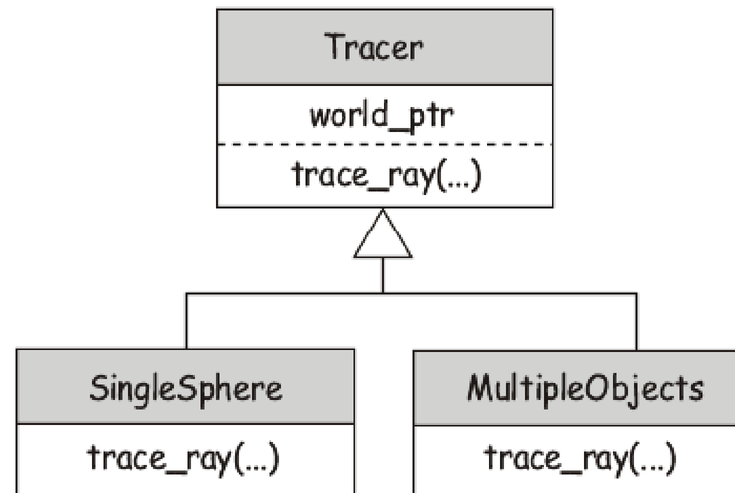
```
class ViewPlane {
   public:
   int    hres;               // horizontal image resolution
   int    vres;               // vertical image resolution
   float  s;                  // pixel size
   int    num_samples;        // number of samples per pixel

   float  gamma;              // gamma correction factor
   float  inv_gamma;          // the inverse of the gamma correction factor
   bool   show_out_of_gamut;  // display red if RGBColor out of gamut

   …
}
```

La classe Tracer

- La classe Tracers ha il compito di gestire i raggi generati in fase di rendering
  - La classe World si occuperà di generare i raggi per la classe Tracer
  - Ogni raggio sarà intersecato con tutti gli oggetti della scena attraverso il metodo `trace_ray`
  - Ricaveremo la lista degli oggetti presenti nella scena attraverso `world_prt`



La classe Tracer

```
class Tracer {
   public:

      Tracer(void);

      Tracer(World* _world_ptr);

      virtual ~Tracer(void);

      virtual RGBColor trace_ray(const Ray& ray) const;

   protected:

      World* world_ptr;
};
```

La sottoclasse SingleSphere

- La sottoclasse SingleSphere sovrascrivere il metodo trace_ray per gestire l'intersezione di ogni singolo raggio con la sfera

```
class SingleSphere: public Tracer {
   public:
      SingleSphere(void);

      SingleSphere(World* _worldPtr);

      virtual ~SingleSphere(void);

      virtual RGBColor trace_ray(const Ray& ray) const;
};

RGBColor SingleSphere::trace_ray(const Ray& ray) const {
   ShadeRec        sr(*world_ptr);     // not used
   double          t;                  // not used

   if (world_ptr->sphere.hit(ray, t, sr))
      return (red);
   else
      return (black);
}
```

La class World

- La gestione dell'intera scena è affidata alla classe World
- La classe World in questa fase si occupa di mantenere all'interno della scena
  - La sfera
  - Il Viewplane
  - Il colore di fondo
- La classe World avrà i seguenti compiti
  - Aprire la finestra di rendering (dipende dal S.O.)
  - Costruire la scena
  - Generare i raggi per il rendering
  - Disegnare i pixel a video (dipende dal S.O.)

La classe World

```
class World {
   public:

      ViewPlane     vp;
      RGBColor      background_color;
      Tracer*       tracer_ptr;
      Sphere        sphere;
      SDL_Surface*  screen;

   public:

      World(void);

      ~World();

      void build(void);

      void render_scene(void) const;

      RGBColor max_to_one(const RGBColor& c) const;

      RGBColor clamp_to_color(const RGBColor& c) const;

      void display_pixel(const int row, const int column, const RGBColor& pixel_color)
const;};
```

Metodo World::build

- Il metodo build si occuperà di:
  - Impostare il viewplane
    - Risoluzione e dimensione del pixel
  - Impostare il colore del background
  - Impostare il Tracer per la gestione dei raggi
  - Impostare i valori dell'unico oggetto sfera presente

```
void World::build(void) {
   vp.set_hres(200);
   vp.set_vres(200);
   vp.set_pixel_size(1.0);

   background_color = black;
   tracer_ptr = new SingleSphere(this);

   sphere.set_center(0.0);
   sphere.set_radius(85.0);
}
```

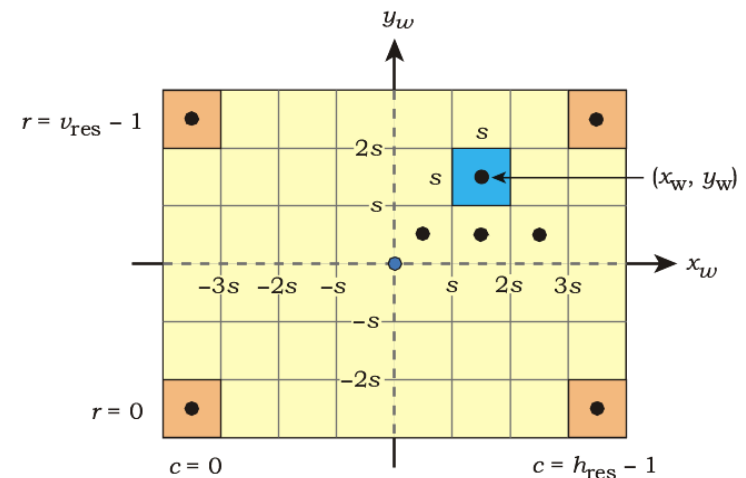
Creazione dei raggi

- Dalla classe viewplane abbiamo i dati necessari per calcolare l'origine e la direzione dei raggi che attraverseranno la scena
- Assumiamo che il viewplane sia per il momento perpendicolare e centrato sull'asse z nella posizione z_w=100.0
- Ogni singolo raggio è generato dal centro del pixel quindi l'origine del raggio sarà

$$x_w = s(c - h_{res} / 2 + 0.5)$$
$$y_w = s(r - v_{res} / 2 + 0.5)$$
$$z_w - 100.0$$

- La direzione del raggio sarà

$\boldsymbol{d} = (0,0,-1)$



Metodo World::render_scene

- All'interno della classe World il metodo `render_scene` si occupa di generare i raggi e passarli al Tracer

```
void World::render_scene(void) const {
   RGBColor         pixel_color;
   Ray              ray;
   int              hres      = vp.hres;
   int              vres      = vp.vres;
   float            s         = vp.s;
   float            xw,yw,zw  = 100.0;            //hardwired in

   ray.d = Vector3D(0, 0, -1);

   for (int r = 0; r < vres; r++)                 // up
      for (int c = 0; c <= hres; c++) {           // across
          xw = s * (c - hres / 2.0 + 0.5);
          yw = s * (r - vres / 2.0 + 0.5);
          ray.o = Point3D(xw, yw, zw);
          pixel_color = tracer_ptr->trace_ray(ray);
          display_pixel(r, c, pixel_color);
}
```

La funzione main

```
int main(void) {

   //Inizializziamo attraverso il S.O. uno screen su cui effettuare il rendering

   World w;
   w.screen = screen;
   w.build();
   w.render_scene();

   return(0);
}
```

La classi attuali

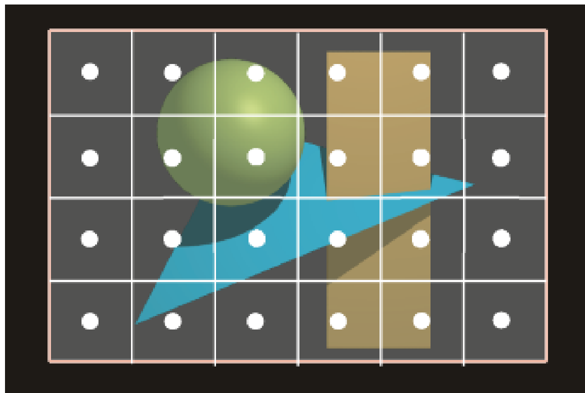
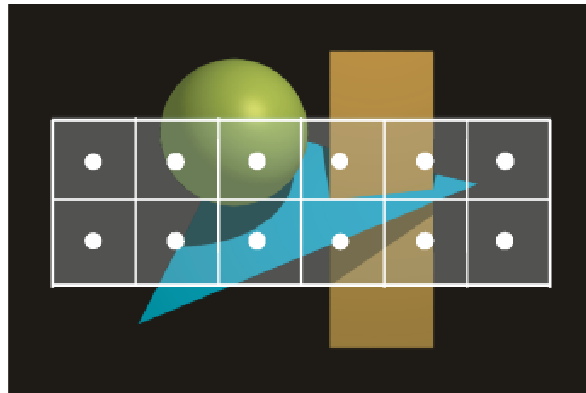
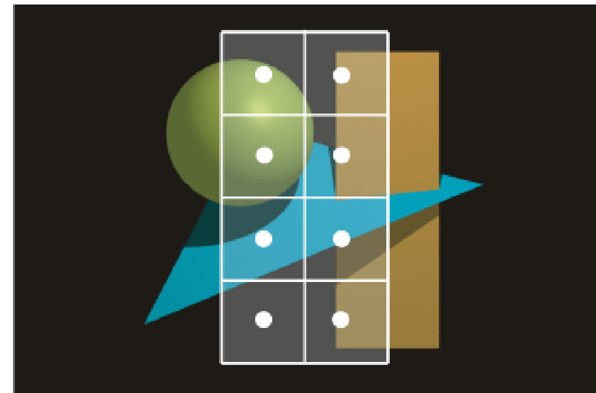
Geometric Objects	Tracers	Utility	World
GeometricObject	Tracer	Normal	ViewPlane
Sphere	SingleSphere	Point3D	World
		Ray	
		RGBColor	
		ShadeRec	
		Vector3D	

Risoluzione e dimensione del pixel

- Il *field of view* (o campo visivo) rappresenta la parte di scena visibile dall'osservatore
- Nella proiezione ortogonale dipende solo dalla risoluzione orizzontale, verticale e dalla dimensione del pixel
  - La dimensione della finestra sarà sh_{res} e sv_{res}
- Questi valori determinano anche in che modo campionare i raggi da generare

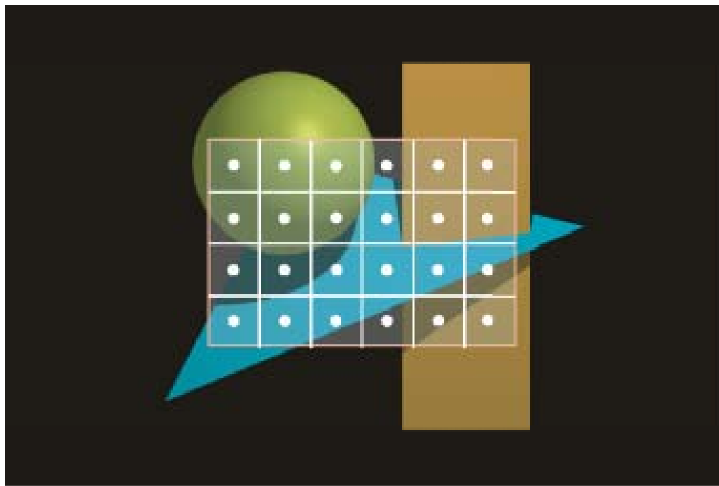
Cambiare la risoluzione

- Variando h_{res} e v_{res} modifichiamo la forma e la risoluzione dell'immagine finale

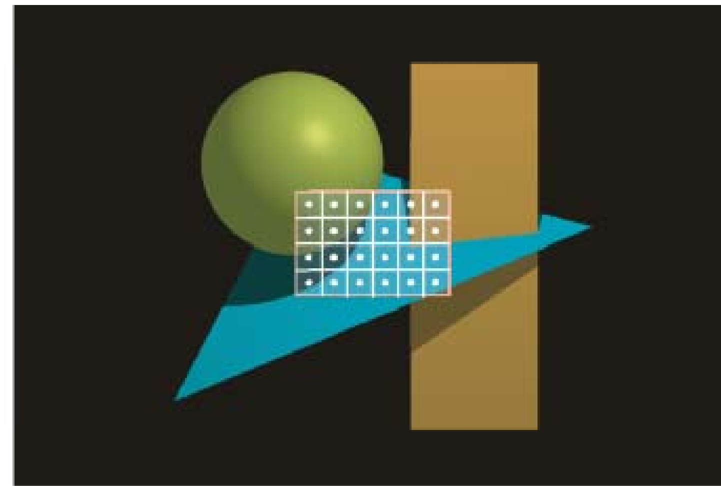




Cambiare la risoluzione del pixel

- Variando la dimensione del pixel s modifichiamo la dimensione della finestra campionando con minore o maggiore frequenza i raggi



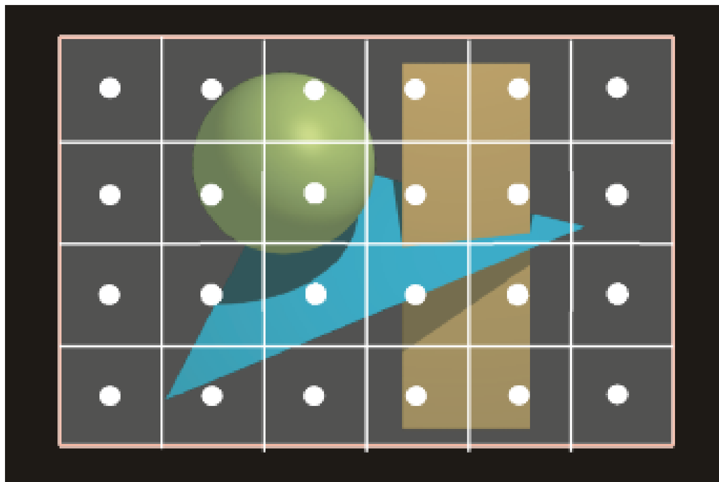
$s=s'$

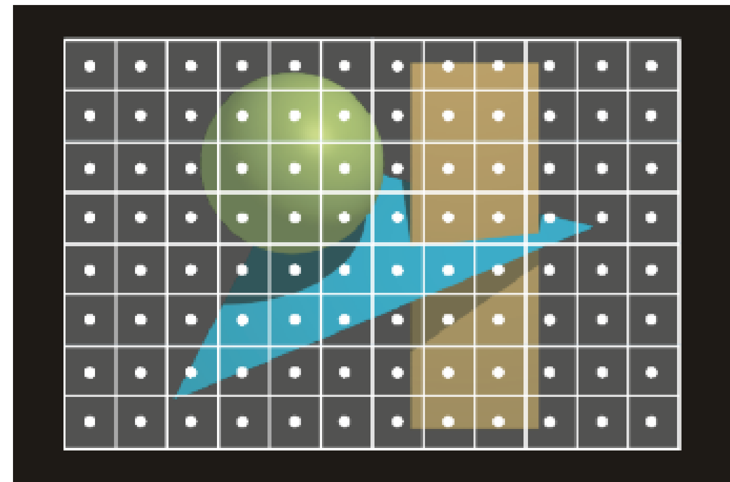


$s=s'/2$

Cambiare risoluzione e dimensione

- Variando la risoluzione e la dimensione del pixel possiamo modificare la resa grafica
  - Ad esempio dimezzando la dimensione del pixel e raddoppiando la risoluzione otteniamo un’immagine con una resa migliore





Un Ray Casting con più oggetti

- Vogliamo implementare il rendering di una scena con più oggetti
- La classe World deve essere modificata in modo tale da
  - Gestire un numero qualunque di oggetti e di qualunque tipo
  - Per ogni raggio l'intersezioni deve essere verificata con tutti gli oggetti della scena
- Ogni oggetto dovrà avere un attributo che ne identifica il colore
  - La classe GeometricObject avrà un attributo color
- Introdurremo la classe ShadeRec per mantenere le informazioni sull'oggetto intersecato dall'ultimo raggio

La classe World

```
class World {
   public:

      ViewPlane                vp;
      RGBColor                 background_color;
      Tracer*                  tracer_ptr;
      SDL_Surface*             screen;
      Vector<GeometricObject*> objects;

   public:

      World(void);

      ~World();

      void build(void);

      void add_object(GeometricObject* object_prt);

      ShadeRec hit_bare_bones_objects(const Ray& ray) const;

      void render_scene(void) const;

      void display_pixel(const int row, const int column, const RGBColor& pixel_color)
const;};
```

La classe ShadeRec

```
class ShadeRec {
   public:

   bool       hit_an_object;   // Did the ray hit an object?
   Material* material_ptr;     // Pointer to the nearest object's material
   Point3D    hit_point;       // World coordinates of intersection
   Point3D    local_hit_point; // World coordinates of hit point on generic object (used for
texture transformations)
   Normal     normal;          // Normal at hit point
   Ray        ray;             // Required for specular highlights and area lights
   int        depth;           // recursion depth
   float      t;               // ray parameter
   World&     w;               // World reference
   RGBColor   color;

   ShadeRec(World& wr);        // constructor

   ShadeRec(const ShadeRec& sr);// copy constructor
};
```

Il metodo World::hit_bare_bones_objects

- Il metodo restituisce un oggetto di tipo ShadeRec con i dati dell'oggetto trovato
- I dati contenuti in ShadeRec saranno utilizzati in fase di shading

```
ShadeRec World::hit_bare_bones_objects(const Ray& ray) {
   ShadeRec sr(*this);
   double   t;
   double   tmin            = kHugeValue;
   int      num_objects     = objects.size();

   for (int j = 0; j < num_objects; j++)
      if (objects[j]->hit(ray, t, sr) && (t < tmin)) {
         sr.hit_an_object      = true;
         tmin                  = t;
         sr.color              = objects[j]->get_color();
      }
   return (sr);
}
```

Il metodo MultipleObject::trace_ray

- Il metodo `trace_ray` dato il raggio determina un'oggetto ShadeRec

```
RGBColor MultipleObjects::trace_ray(const Ray& ray) const {
   ShadeRec sr(world_ptr->hit_bare_bones_objects(ray)); // sr is copy constructed

   if (sr.hit_an_object)
      return (sr.color);
   else
      return (world_ptr->background_color);
}
```

Metodo World::build

```
void World::build(void) {
   vp.set_hres(200);
   vp.set_vres(200);

   background_color = black;
   tracer_ptr = new MultipleObjects(this);

   // use access functions to set center and radius
      for this sphere

   Sphere* sphere_ptr = new Sphere;
   sphere_ptr->set_center(0, -25, 0);
   sphere_ptr->set_radius(80.0);
   sphere_ptr->set_color(1, 0, 0);                  // red
   add_object(sphere_ptr);

   // use a constructor to set center and radius for this sphere

   sphere_ptr = new Sphere(Point3D(0, 30, 0), 60);
   sphere_ptr->set_color(1, 1, 0);                  // yellow
   add_object(sphere_ptr);

   Plane* plane_ptr = new Plane(Point3D(0.0), Normal(0, 1, 1));
   plane_ptr->set_color(0.0, 0.25, 0.0);            // dark green
   add_object(plane_ptr);
}
```