Εισαγωγή στον προγραμματισμό γραφικών με ray tracing

Γιάννης Τσιομπίκας
nuclear@member.fsf.org

18 August 2012

1 Εισαγωγή

Σε αυτό το άρθρο, θα δούμε τον πιο απλό, αλλά συνάμα άκρως εντυπωσιακό αλγόριθμο rendering 3D γραφικών, γνωστό ως "ray tracing". Αν και ο αλγόριθμος είναι απλός, το πλήρες πρόγραμμα δεν χωράει σε καμία περίπτωση στις σελίδες του περιοδικού. Κατεβάστε λοιπών τον συνοδευτικό κώδικα από την σελίδα: `http://nuclear.mutantstargoat.com/articles/introray_gr`, μιας και θα αναφερόμαστε σε αυτόν για τις λεπτομέρειες της υλοποίησης.

Ο αλγόριθμος πίσω από το ray tracing είναι τόσο απλός που μπορώ να τον περιγράψω περιληπτικά σε δυο-τρεις παραγράφους, και αυτή η περιγραφή θα ήταν αρκετή για τους πιο τολμηρούς αναγνώστες, με μια ελαφριά κλίση στα μαθηματικά, ώστε να τον υλοποιήσουν.

Ο αλγόριθμός βασίζεται στην γεωμετρική οπτική. Θεωρώντας την οθόνη σαν ένα παράθυρο στον 3D κόσμο μπροστά από τον παρατηρητή, το ζητούμενο είναι: για κάθε pixel να υπολογίσουμε πόσο φως έρχεται από την κατεύθυνση που του αντιστοιχεί, και τι χρώμα έχει. Αυτό εξαρτάται από τα αντικείμενα από τα οποία έχει ανακλαστεί πριν φτάσει στον παρατηρητή, καθώς φυσικά και από τις ιδιότητες της φωτεινής πηγής από την οποία ξεκίνησε.

Είναι μη-πρακτικό να ξεκινήσουμε από την φωτεινή πηγή και να αρχίσουμε να στέλνουμε φωτόνια προς όλες τις κατευθύνσεις, όπως συμβαίνει στην πραγματικότητα, μιας και απειροελάχιστα από αυτά θα κατέληγαν στον παρατηρητή.

Ξεκινάμε λοιπών αντίστροφα, από τον παρατηρητή, και για κάθε pixel υπολογίζουμε την κατεύθυνση της ακτίνας που περνάει από αυτό. Ακολουθούμε την πορεία αυτής της πρωταρχικής ακτίνας (primary ray), και βρίσκουμε το πρώτο σημείο τομής της (intersection point), με τα αντικείμενα της σκηνής. Στο κοντινότερο intersection point λοιπών, υπολογίσουμε την ποσότητα και το χρώμα του φωτός που ανακλάται από αυτό το σημείο προς την κατεύθυνση από την οποία ήρθε η

ακτίνα. Αν το αντικείμενο που χτύπησε η ακτίνα είναι ανακλαστικό, μέρος αυτού του φωτός προέρχεται από αντανακλάσεις άλλων αντικειμένων (έμμεσος φωτισμός). Για να το υπολογίσουμε αυτό, βρίσκουμε την κατεύθυνση ανάκλασης της ακτίνας που ακολουθούσαμε, και επαναλαμβάνουμε την ίδια διαδικασία recursively προς τα εκεί, προσθέτοντας και αυτό το φως σε ότι υπολογίσαμε από αυτό που υπολογίσαμε ότι προέρχεται άμεσα από της φωτεινές πηγές της σκηνής.

Σε κάθε περίπτωση το χρώμα που θα μας επιστρέψει το trace του primary ray που αντιστοιχεί σε κάθε pixel, θα το χρησιμοποιήσουμε για να "βάψουμε" το pixel αυτό στην τελική εικόνα (frame buffer).

2 Βασικές Δομές

Για να υλοποιήσουμε τον αλγόριθμο που μόλις περιέγραψα, προφανώς θα χρειαστούμε μερικούς βασικούς μαθηματικούς τύπους στο πρόγραμμά μας. Λόγο περιορισμένου χώρου, θα παραθέσω μόνο το interface εδώ. Για την υλοποίηση δείτε τον συνοδευτικό κώδικα (vmath.h/vmath.inl).

Κατ' αρχάς χρειαζόμαστε τρισδιάστατα διανύσματα, που μας είναι απαραίτητα για να αναπαραστήσουμε σημεία και κατευθύνσεις στον 3D χώρο. Η κλάση Vector3 ορίζει ένα διάνυσμα ως μια τριάδα floating point αριθμών, και υλοποιεί κάποιες βασικές πράξεις που θα χρειαστούμε, όπως πρόσθεση, αφαίρεση, πολλαπλασιασμός με αριθμό (scaling), υπολογισμός μήκους, κανονικοποίηση, εσωτερικό και εξωτερικό γινόμενο, κ.α.

```
class Vector3 {
public:
    double x, y, z;

    Vector3();
    Vector3(double x, double y, double z);

    Vector3 operator -() const;
    Vector3 &operator +=(const Vector3 &v);

    double length() const;
    double length_sq() const;
    void normalize();
};

Vector3 normalize(const Vector3 &v);

Vector3 operator +(const Vector3 &a, const Vector3 &b);
Vector3 operator -(const Vector3 &a, const Vector3 &b);
Vector3 operator *(const Vector3 &v, double s);
Vector3 operator /(const Vector3 &v, double s);

double dot(const Vector3 &a, const Vector3 &b);
```

```
Vector3 cross(const Vector3 &a, const Vector3 &b);

Vector3 reflect(const Vector3 &v, const Vector3 &n);
Vector3 transform(const Vector3 &v, const Matrix4x4 &m);
```

Για τις ακτίνες θα χρειαστούμε μια κλάση που να περιέχει την αρχή (origin) της ακτίνας, και την κατεύθυνση της (direction).

```
struct Ray {
    Vector3 origin, dir;
};
```

Τέλος χρειαζόμαστε πίνακες (matrices), που χρησιμεύουν για να μετασχηματίσουμε διανύσματα από ένα σύστημα συντεταγμένων σε ένα άλλο. Στον πολύ απλό εισαγωγικό ray-tracer που θα φτιάξουμε δεν θα χρησιμοποιήσουμε πολύ μετασχηματισμούς, μιας και όλα τα αντικείμενα μας θα τα ορίσουμε σε ένα κοινό σύστημα συντεταγμένων. Παρόλα αυτά, θα χρειαστούμε πίνακες για τον μετασχηματισμό της "κάμερας" που γεννά primary rays, όπως θα δούμε παρακάτω.

Για να περιγράψουμε γραμμικούς μετασχηματισμούς σε 3 διαστάσεις, αρκεί ένας πίνακας 3x3. Θα μας βόλευε όμως να μπορούμε να περιγράψουμε affine μετασχηματισμούς, δηλαδή γραμμικούς συν παράλληλη μεταφορά. Αυτό μπορούμε να το πετύχουμε χρησιμοποιώντας μια επιπλέον διάσταση, θεωρώντας δηλαδή ότι τα διανύσματά μας είναι 4-διάστατα στο υπερεπίπεδο w = 1, και χρησιμοποιώντας πίνακες 4x4. Για περισσότερα πάνω σε αυτό το μαθηματικό τέχνασμα βλ. "ομογενείς συντεταγμένες" (homogeneous coordinates).

Τέλος χρώματα θα αναπαραστήσουμε χρησιμοποιώντας ένα 3-διάστατο διάνυσμα, θεωρώντας ότι οι συντεταγμένες xyz αντιστοιχούν στα στοιχεία rgb (red green blue) του χρώματος. Έτσι, μια εικόνα, όπως ο framebuffer μπορεί να είναι απλά ένα array από τέτοια διανύσματα:

```
typedef Vector3 Color;

class Image {
public:
    Color *pixels;
    int xsz, ysz;

    Image();
    Image(int xsz, int ysz);
    ~Image();
    bool save(const char *fname) const;
};
```

3 Σκηνή και αντικείμενα

Ο απλοϊκός ray-tracer που θα φτιάξουμε, θα χρησιμοποιεί σκηνές φτιαγμένες από σφαίρες και επίπεδα. Αργότερα μπορούμε εύκολα να τον επεκτείνουμε να χειρίζεται και αντικείμενα φτιαγμένα από πολύγωνα για πλήρη ευελιξία, αλλά αυτό παρουσιάζει διάφορα άλλα ενδιαφέροντα προβλήματα που θα χρειαστούν ολόκληρο άρθρο από μόνα τους.

Θα φτιάξουμε λοιπών μια ιεραρχία κλάσεων με βάση το class Object, και δύο υποκλάσεις: Sphere και Plane. Την κλάση Object θα την κάνουμε abstract base class, με pure virtual συνάρτηση intersect, την οποία θα υλοποιήσουν οι υποκλάσεις που αναφέραμε για να βρίσκουν εάν και πού τέμνει μια ακτίνα το αντικείμενο.

```
class Object {
public:
    Material material;

    virtual ~Object();
    virtual bool intersect(const Ray &ray, HitPoint *pt) const =
        0;
};
```

Η συνάρτηση intersect επιστρέφει true/false αναλόγως αν η ακτίνα τέμνει το αντικείμενο η όχι, και στην πρώτη περίπτωση γεμίζει και ένα HitPoint structure με επιπλέον πληροφορίες για το σημείο τομής μέσω του pointer που περιμένει σαν δεύτερη παράμετρο.

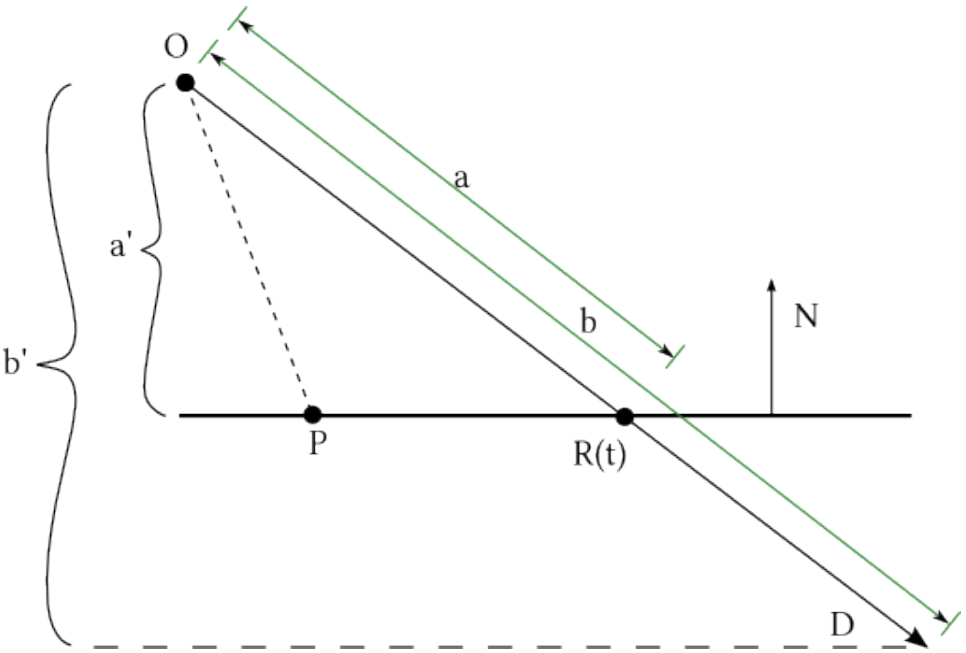
Το struct HitPoint περιέχει καταρχάς την παραμετρική απόσταση της τομής πάνω στην ακτίνα. Δηλαδή έναν αριθμό t που αν τον αντικαταστήσουμε στην παραμετρική εξίσωση της ευθείας $\text{origin} + \text{direction}\ t$ (όπου origin και direction τα δύο διανύσματα που ορίζουν την ακτίνα μας όπως είπαμε παραπάνω), παίρνουμε το ακριβές σημείο της τομής στο σύστημα συντεταγμένων του 3D κόσμου μας. Αυτό τον υπολογισμό τον κάνει ή intersect, και αποθηκεύει στο δεύτερο πεδίο το διάνυσμα αυτού του σημείου. Το τρίτο πεδίο είναι το normal στο σημείο τομής, δηλαδή ένα μοναδιαίο διάνυσμα κάθετο στην επιφάνεια του αντικειμένου σε αυτό το σημείο, που θα μας χρειαστεί για να υπολογίσουμε τον φωτισμό και την γωνία ανάκλασης αργότερα.

```
struct HitPoint {
    double dist;
    Vector3 pos;
    Vector3 normal;
    const Object *obj;
};
```

Ένα επίπεδο ορίζεται από την εξίσωση $\text{A}x + \text{B}y + \Gamma z + \Delta = 0$, όπου (A B Γ) είναι απλά το normal (κάθετο διάνυσμα) του επιπέδου, και Δ

η απόσταση του από την αρχή του συστήματος συντεταγμένων. Οπότε στην κλάση Plane, απλώς έχουμε ένα Vector3 normal, και ένα double dist.

Ο υπολογισμός της τομής μεταξύ ακτίνας και επιπέδου είναι πολύ απλός, χρησιμοποιώντας την γεωμετρική κατασκευή του σχήματος 1.



Σχήμα 1: Τομή ακτίνας με επίπεδο.

Αυτό που χρειαζόμαστε είναι να υπολογίσουμε το $t = \frac{a}{b}$. Από το σχήμα είναι προφανές ότι $\frac{a}{b} = \frac{a'}{b'}$, όπου a' είναι η απόσταση της αρχής της ακτίνας από το επίπεδο, και b' η προβολή της κατεύθυνσης D, στην ευθεία που ορίζεται από το normal του επιπέδου N. Ξεκινάμε βρίσκοντας ένα οποιοδήποτε σημείο στο επίπεδο: $P = N\Delta$. Κατόπιν υπολογίζουμε το a' ως το εσωτερικό γινόμενο του N με το διάνυσμα $P - O$, μιας και το εσωτερικό γινόμενο μεταξύ δύο διανυσμάτων πρακτικά μας δίνει την προβολή του ενός στο άλλο επί το μήκος τους, και το μήκος του N είναι 1. Με παρόμοιο τρόπο υπολογίζουμε και το b' με το εσωτερικό γινόμενο του N και του D.

```
bool Plane::intersect(const Ray &ray, HitPoint *pt) const
{
    double ndotdir = dot(normal, ray.dir);
    if(fabs(ndotdir) < EPSILON) {
        return false;  // ray is parallel
    }
    Vector3 planept = normal * dist;
```

```
    Vector3 pptdir = planept - ray.origin;
    double t = dot(normal, pptdir) / ndotdir;
    if(t < EPSILON) {
        // intersection behind the origin
        return false;
    }
    // fill the HitPoint structure
    pt->obj = this;
    pt->dist = t;
    pt->pos = ray.origin + ray.dir * t;
    pt->normal = normal;
    return true;
}
```

Για την αναπαράσταση μιας σφαίρας θα χρειαστούμε απλά την ακτίνα της, και ένα διάνυσμα για το κέντρο. Η σφαίρα με κέντρο (xyz) και ακτίνα r, ορίζεται ως ο γεωμετρικός τόπος των σημείων που ικανοποιούν την εξίσωση $x^2 + y^2 + z^2 + r^2 = 0$. Αντικαθιστώντας την παραμετρική εξίσωση της ακτίνας στην εξίσωση της σφαίρας, και κάνοντας μερικούς αλγεβρικούς μετασχηματισμούς, καταλήγουμε σε μια απλή δευτεροβάθμια εξίσωση που μπορεί να λυθεί ως προς t με τον γνωστό τρόπο.

$$(O_x + D_x t - C_x)^2 + (O_y + D_y t - C_y)^2 + (O_z + D_z t - C_z)^2 - r^2 = 0$$

$$\begin{aligned} &O_x^2 + D_x t - C_x^2 + 2O_x D_x t - 2O_x C_x - 2D_x t C_x + \\ &\quad O_y^2 + D_y t - C_y^2 + 2O_y D_y t - 2O_y C_y - 2D_y t C_y + \\ &\quad O_z^2 + D_z t - C_z^2 + 2O_z D_z t - 2O_z C_z - 2D_z t C_z - r^2 = 0 \end{aligned}$$

Συλλέγωντας τις δυνάμεις του t και απλοποιώντας:

$$\begin{aligned} &D_x^2 t^2 + D_y^2 t^2 + D_z^2 t^2 + 2O_x D_x t - 2D_x t C_x + 2O_y D_y t - \\ &\quad 2D_y t C_y + 2O_z D_z t - 2D_z t C_z + O_x^2 + O_y^2 + O_z^2 + C_x^2 + C_y^2 + C_z^2 - \\ &\quad 2O_x C_x - 2O_y C_y - 2O_z C_z - r^2 = 0 \Leftrightarrow \end{aligned}$$

$$\begin{aligned} &D_x^2 t^2 + D_y^2 t^2 + D_z^2 t^2 + 2D_x(O_x - C_x)t + 2D_y(O_y - C_y)t + \\ &\quad 2D_z(O_z - C_z)t + O_x^2 + O_y^2 + O_z^2 + C_x^2 + C_y^2 + C_z^2 - \\ &\quad 2(O_x C_x + O_y C_y + O_z C_z) - r^2 = 0 \Leftrightarrow \end{aligned}$$

$$at^2 + bt + c = 0$$

Όπου:

$$a = D_x^2 + D_y^2 + D_z^2 = D \cdot D$$

$$
\begin{aligned}
b &= 2D_x(O_x - C_x) + 2D_y(O_y - C_y) + 2D_z(O_z - C_z) \\
c &= O_x^2 + O_y^2 + O_z^2 + C_x^2 + C_y^2 + C_z^2 - 2(O_xC_x + O_yC_y + O_zC_z) - r^2 \\
&= O \cdot O + C \cdot C - 2(O \cdot C) - r^2
\end{aligned}
$$

Η λύση της δευτεροβάθμιας εξίσωσης μας, δεν έχει πραγματικές λύσεις (αρνητική διακρίνουσα) όταν η ακτίνα δεν τέμνει την σφαίρα, έχει μία λύση όταν η ακτίνα εφάπτεται στην σφαίρα, ενώ όταν την τέμνει κανονικά μας δίνει δύο λύσεις, που αντιστοιχούν στην παραμετρική απόσταση της μπροστά και πίσω τομής (από τις οποίες προφανώς εμάς μας ενδιαφέρει η κοντινότερη). Δείτε την υλοποίηση της συνάρτησης Sphere::intersection στο αρχείο sphere.cc στον συνοδευτικό κώδικα.

Τέλος η κλάση Scene κρατάει σε std::vector όλα τα αντικείμενα της σκηνής, τα φώτα που ορίζονται απλώς από ένα διάνυσμα θέσης, και την κάμερα που περιγράφεται απο έναν πίνακα μετασχηματισμού, και δουλειά της είναι να υπολογίζει το primary ray για κάθε pixel.

```
class Scene {
private:
    std::vector<Object*> objects;
    std::vector<Light*> lights;
    Camera *camera;
public:
    ... constructors/destructors    ...
    bool intersect(const Ray &ray, HitPoint *hit) const;
    Color trace_ray(const Ray &ray, int rdepth) const;
    Color shade(const Ray &ray, const HitPoint &hit, int rdepth)
        const;
};
```

Το σημαντικότερο κομμάτι σε αυτή την κλάση, είναι οι 3 συναρτήσεις που πρακτικά υλοποιούν σχεδόν όλο τον αλγόριθμο του ray tracing. Η trace_ray καλείται από το κεντρικό rendering loop για κάθε pixel, ώστε να ακολουθήσει την ακτίνα και να μας επιστρέψει το φως που λαμβάνουμε από αυτή την κατεύθυνση, και άρα το χρώμα που πρέπει να βάψουμε το pixel. Για να το πετύχει αυτο, η trace_ray καλεί καταρχάς την intersect για να βρει το σημείο που τέμνει η ακτίνα την γεωμετρία της σκηνής, και εάν βρει τομή καλεί την shade με παράμετρο τις πληροφορίες της τομής που είδαμε παραπάνω (HitPoint), για να υπολογίσει το χρώμα που πρέπει να επιστρέψει. Η shade με τη σειρά της αφού υπολογίσει το φως που λαμβάνει αυτό το σημείο από τις φωτεινές πηγές (lights vector), μπορεί να ξανα-κινήσει την διαδικασία recursively προς την ανακλώμενη κατεύθυνση, καλώντας πάλι την trace_ray με άλλη ακτίνα.

```
Color Scene::trace_ray(const Ray &ray, int rdepth) const
{
    HitPoint hit;
    if(intersect(ray, &hit)) {
```

```
        return shade(ray, hit, rdepth);
    }
    // not found, return background color
    return bgcolor;
}
bool Scene::intersect(const Ray &ray, HitPoint *nearest_hit) const
{
    nearest_hit->obj = 0;
    nearest_hit->dist = DBL_MAX;
    // find the nearest hit (if any)
    for(Object *obj: objects) {
        HitPoint hit;
        if(obj->intersect(ray, &hit) &&
                hit.dist < nearest_hit->dist) {
            *nearest_hit = hit;
        }
    }
    return nearest_hit->obj != 0;
}
```

4 Φωτισμός

Όπως είπαμε παραπάνω, η δουλειά της shade είναι καταρχάς να υπολογίσει πόσο φως φτάνει από κάθε φωτεινή πηγή σε κάποιο σημείο, και πόσο από αυτό το φως ανακλάται προς την κατεύθυνση από την οποία έρχεται η ακτίνα.

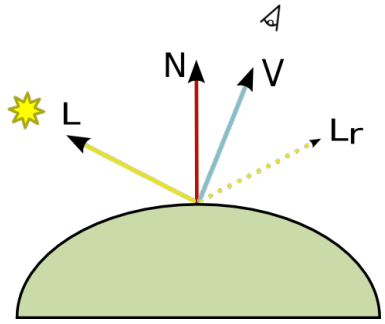
Για κάθε φωτεινή πηγή, πρώτα πρέπει να διαπιστώσουμε αν υπάρχει οπτική επαφή με την πηγή, ή βρισκόμαστε σε σημείο σκιασμένο από κάποιο ενδιάμεσο αντικείμενο. Για να το υπολογίσουμε αυτό, αρκεί να ρίξουμε μια ακτίνα προς την κατεύθυνση του φωτός (shadow ray) και να δούμε αν χτυπάει κάποιο αντικείμενο ενδιάμεσα ή όχι. Αν βρούμε τομή, απλώς αγνοούμε αυτή την φωτεινή πηγή μιας και δεν μπορεί να συνδράμει στον φωτισμό.

Χωρίζουμε την αλληλεπίδραση του φωτός με την επιφάνεια των αντικειμένων σε δύο κατηγορίες, με βάση το υλικό της επιφάνειας που προσπαθούμε να προσεγγίσουμε:

- Οι τραχιές επιφάνειες διαχέουν το φως που λαμβάνουν ισόποσα προς όλες τις κατευθύνσεις του ημισφαιρίου που τις περιβάλλει. Αυτή η αλληλεπίδραση λέγεται diffuse, και συμπεριφέρεται σύμφωνα με τον νόμο του Lambert που λέει ότι το ποσό της ακτινοβολίας που διαχέεται προς οποιαδήποτε κατεύθυνση είναι ίσο με το συνημίτονο της προσπίπτουσας γωνίας από το normal.

- Οι λείες επιφάνειες ανακλούν το φως που λαμβάνουν προς μια μικρή δέσμη κατευθύνσεων γύρο από την κατεύθυνση ανάκλασης της προσπίπτουσας. Αυτή η αλληλεπίδραση λέγεται specular.

Πολλές επιφάνειες στην πραγματικότητα, παρουσιάζουν συνδυασμό των παραπάνω αλληλεπιδράσεων, είτε γιατί βρίσκονται κάπου ενδιάμεσα από τις δύο ακραίες περιπτώσεις που περιγράψαμε, είτε γιατί έχουν τραχιά επιφάνεια καλυμμένη από κάποια γυαλιστερή επίστρωση. Για να προσεγγίσουμε λοιπών αυτές τις επιφάνειες, υπολογίζουμε και τις δύο πιθανές αλληλεπιδράσεις (diffuse και specular), και χρησιμοποιούμε ένα σταθμικό άθροισμα τους, ως το τελικό χρώμα που θα επιστρέψει ο υπολογισμός του φωτισμού. Τα βάρη που θα χρησιμοποιήσουμε τα παίρνουμε από το material του αντικειμένου που περιέχει όλες τις παραμέτρους που χρειαζόμαστε για τους υπολογισμούς φωτισμού για την κάθε επιφάνεια (material.h).



Σχήμα 2: Μοντέλο φωτισμού.

Την diffuse αλληλεπίδραση την υπολογίζουμε σύμφωνα με τον νόμο του Lambert που προαναφέραμε. Για να υπολογίσουμε το συνημίτονο της γωνίας ανάμεσα στο διάνυσμα κατεύθυνσης του φωτός και το normal, αρκεί να υπολογίσουμε το εσωτερικό γινόμενο των δύο διανυσμάτων, αφού φροντίσουμε να είναι μοναδιαία.

Για την specular αλληλεπίδραση, θα χρησιμοποιήσουμε το εμπειρικό μοντέλο του phong, που αν και δεν έχει κάποια εδραίωση στην φυσική, είναι απλό και βγάζει πιστευτά specular highlights. Υπολογίζουμε λοιπών καταρχήν την διεύθυνση ανάκλασης του φωτός, και κατόπιν σηκώνουμε σε κάποια δύναμη το εσωτερικό γινόμενο μεταξύ του διανύσματος αυτού, και του διανύσματος κατεύθυνσης του παρατηρητή (δηλαδή την αντίθετη κατεύθυνση από αυτή της ακτίνας). Όσο μεγαλύτερη αυτή η δύναμη, τόσο πιο συγκεντρωμένο το highlight, και άρα πιο λεία δείχνει η επιφάνεια.

Αφού επαναλάβουμε αυτή τη διαδικασία για κάθε φως και προσθέσουμε τα αποτελέσματα, μένει να δούμε αν το αντικείμενο είναι ανακλαστικό (παράμετρος reflectivity στο material), και αν ναι, να υπολογίσουμε την κατεύθυνση ανάκλασης της ακτίνας, και να καλέσουμε την trace_ray recursively. Το χρώμα που θα πάρουμε από την ανακλώ-

μενη ακτίνα το προσθέτουμε στο τελικό χρώμα που θα επιστρέψουμε, αφού το πολλαπλασιάσουμε πρώτα με το reflectivity factor. Φυσικά πρέπει να φροντίσουμε να μην πέσουμε σε ατέρμονο recursion αν η ακτίνα πιαστεί ανάμεσα σε δυο ανακλαστικά αντικείμενα, οπότε φροντίζουμε να τερματίσουμε το recursion μετά από ένα όριο.

Για την πλήρη υλοποίηση της shade δείτε το αρχείο scene.cc.

5 Συμπληρώνοντας το puzzle

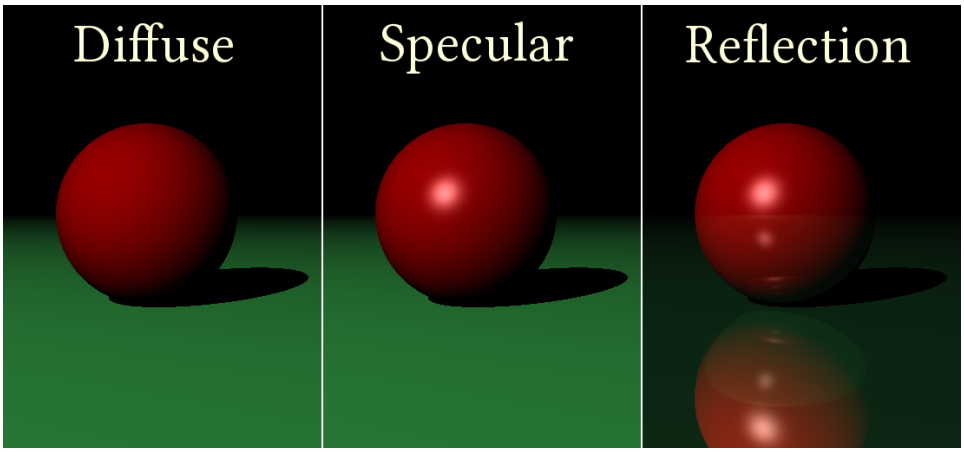
Αφού υλοποιήσουμε όλα τα παραπάνω, ας δούμε και πώς τα συνδυάζουμε για να κάνουμε render 3D σκηνές.

Ξεκινώντας ο απλοϊκός ray-tracer μας, δημιουργεί τον framebuffer, και καλεί μια συνάρτηση create_test_scene, που φτιάχνει όλα τα αντικείμενα που θα περιέχει η σκηνή. Κατόπιν για κάθε pixel ζητά από την κάμερα το primary ray που του αντιστοιχεί, το οποίο και δίνει στην trace_ray για να υπολογιστεί το χρώμα αυτού του pixel.

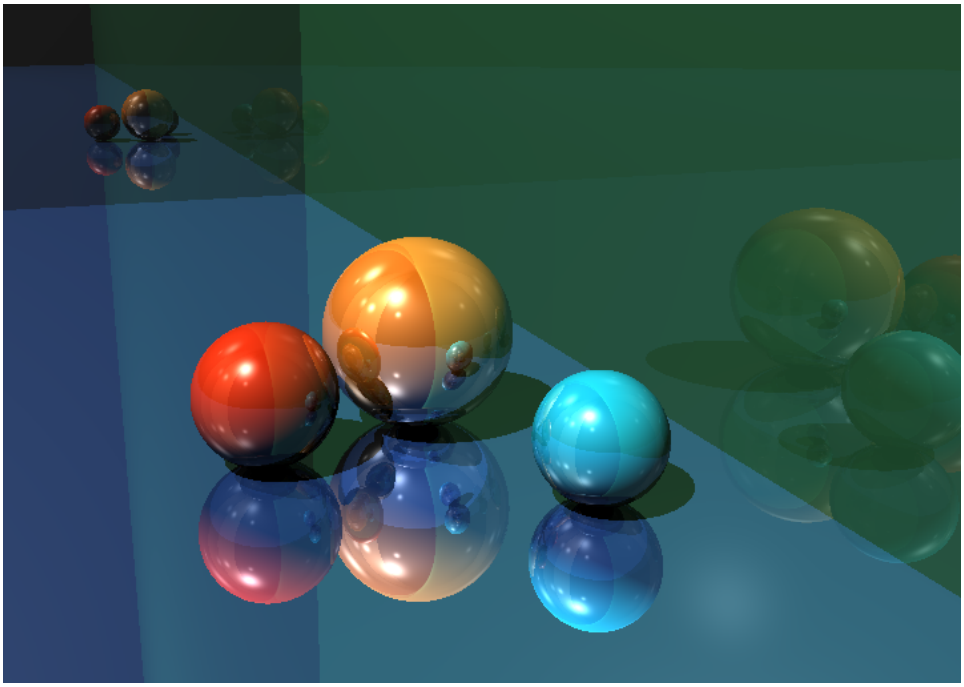
```
Image frame(width, height);
Scene *scn = create_test_scene();
Camera *cam = scn->get_camera();

// for every pixel ...
Color *pixel = frame.pixels;
for(int i=0; i<frame.ysz; i++) {
    for(int j=0; j<frame.xsz; j++) {
        // construct a ray passing through the pixel
        Ray ray = cam->get_primary_ray(j, i, frame.xsz, frame.ysz)
            ;
        // trace the ray and write the pixel
        *pixel++ = scn->trace_ray(ray, 0);
    }
}
frame.save("output.ppm");
```

Το μόνο κομμάτι που δεν καλύψαμε είναι πώς υπολογίζεται το primary ray που αντιστοιχεί σε κάθε pixel. Δυστυχώς λόγο περιορισμένου χώρου δεν μπορούμε να το αναλύσουμε εκτενώς. Συνοπτικά υπολογίζουμε τη θέση του pixel σε ένα επίπεδο μπροστά από την αρχή του συστήματος συντεταγμένων τριγωνομετρικά, και από αυτό υπολογίζουμε την κατεύθυνση της ακτίνας. Τέλος μετασχηματίζουμε την ακτίνα με τον πίνακα μετασχηματισμού της κάμερας, που έχουμε υπολογίσει με βάση την επιθυμητή θέση και κατεύθυνση της (βλ. camera.cc για λεπτομέρειες).

Το αποτέλεσμα του προγράμματός μας δεν είναι καθόλου άσχημο για την απλότητα του αλγορίθμου που υλοποιήσαμε, όπως φαίνεται στις επόμενες εικόνες.



Σχήμα 3: Αποτελέσματα αλγορίθμου.



Σχήμα 4: Πιο ενδιαφέρουσα σκηνή.

Για οποιεσδήποτε απορίες η ερωτήσεις σχετικά με το ray tracing, και τον προγραμματισμό γραφικών γενικότερα, μην διστάσετε να επικοινωνήσετε στο nuclear@member.fsf.org.

Happy hacking!