**Αλγόριθμοι και Συστήματα για 3Δ Γραφικά**
**Τμήμα Πληροφορικής**
**Πανεπιστήμιο Ιωαννίνων**
**Χειμερινό Εξάμηνο 2006-2007**

# ΑΣΚΗΣΗ 1:Raytracing και Supersampling

Παραδώστε την άσκηση μέχρι την Τετάρτη 22/11/2006. Θα παραδώστε μια εικόνα για κάθε μέρος της άσκησης και τον τελικό μόνο κώδικα όλα μαζί σε ένα CD.

## 0. Πριν αρχίσετε

Κατεβάστε τον πηγαίο κώδικα από το: http://www.cs.uoi.gr/~fudos/grad-exer1/source.zip

Επίσης κατεβάστε το zexr http://www.cs.uoi.gr/~fudos/grad-exer1/zexr_pc.zip πρόγραμμα για να βλέπετε .exr αρχεία και κάντε το unzip στον ίδιο κατάλογο με τον πηγαίο κώδικα της άσκησης.

Χρησιμοποιήστε Visual Studio 2003 ή μεταγενέστερο, κάντε τις ρυθμίσεις που χρειάζονται. Χρησιμοποιήστε το Release mode καθώς το Debug mode μπορεί να είναι πολύ αργό. Κάποια από τα warnings για εξωτερικές βιβλιοθήκες μπορείτε να τα αγνοήσετε. Όταν κάνετε επιτυχημένο compilation θα δημιουργηθεί ένα εκτελέσιμο (asrTracer.exe) στον κατάλογο που εργάζεστε.

Ανοίξτε ένα command prompt και κάντε cd στον κατάλογο εργασίας. Εκτελέστε το asrTracer και μελετήστε τη έξοδό του (αρχείο output.exr) με το zexr. Η έξοδος είναι μια μαυρόασπρη εικόνα 512x512 η οποία έχει άσπρα pixels όπου η ακτίνα χτυπά την σκηνή και μαύρα αλλού

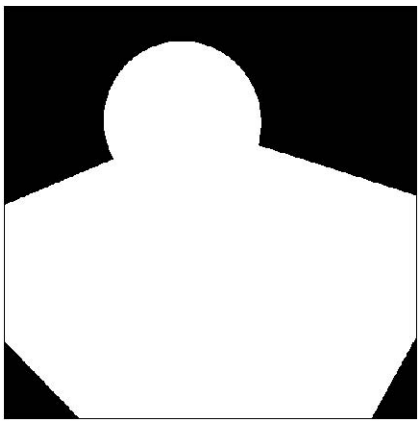
Κοιτάξτε τον κώδικα και την σελίδα

http://www.cs.uoi.gr/~fudos/grad-exer1/doc/hierarchy.html.

Συγκεκριμένα κοιτάξτε περισσότερο το main.cpp και τις classes Raytracerand WhittedTracer. Στην άσκηση αυτή θα κάνετε όλη την υλοποίηη στην class WhittedTracer class.

## 1. Direct Illumination

When the function `computeImage()` is called from the main function, the raytracer loops over all pixels in the image and calls `tracePixel(x,y)` for each of them. The `tracePixel` function is responsible for computing the output color of the pixel at coordinate (x,y) and is implemented in the `WhittedTracer` class. Look at the implementatation in `WhittedTracer::tracePixel`.



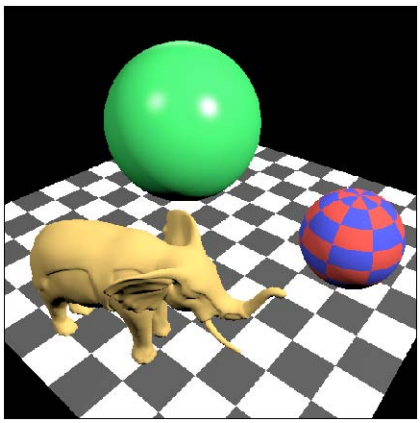
First, it computes the coordinate for the center of the pixel and then asks the camera to setup a suitable ray that goes from the camera origin through the pixel center. We trace this ray in the `trace(ray,depth)` function. In the trace function, the ray is intersection tested against the scene by the lines:

```
Intersection is;
bool hit = mScene->intersect(ray,is);
```

The `intersect` function returns true if the ray hits the scene, and information about the intersection point is returned in the Intersection object. In the default implementation, the output color is simply set to white by: `out = Color(1,1,1)` if the ray hits the scene.

Your first task is to implement direct illumination to make the scene look a little nicer. Replace the constant output color with the call: `out = directIllumination(is)`. This function takes the Intersection object as input and should compute the direct illumination from the light sources in the scene. Implement this function. Do not worry about shadow rays for now, assume all light sources in the scene are visible.

The implementation of `directIllumination` involves the following steps. For each light source, compute the light vector which is the vector from the hit point to the light source. This vector needs to be normalized. The contribution from a light source is computed as the radiance of the light multiplied by the BRDF of the material at the hit point. The output color is the sum of all the contributions. If done correctly, the output will look like this:

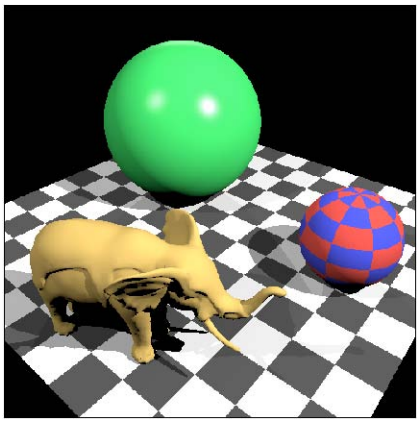


### *Some hints*

```
int n = mScene->getNumberOfLights();                    Returns the number of light sources
PointLight* l = mScene->getLight(i);                    Returns light source number i=(0…n-1)
Point p1 = l->getWorldPosition();                       Position of the light source
Point p2 = is.mPosition;                                Position of the hit point
L.normalize();                                          Normalizes a vector
Color r = l->getRadiance();                             Output light radiance (color)
Color w = is.mMaterial->getBRDF(is,L);                  Returns the BRDF for light vector L
```

## 2. Shadows

The image in the previous exercise looks very artificial because it lacks shadows. Your next task is to incorporate shadows into the computation of the direct illumination. In raytracing, this is easily accomplished by tracing so called shadow rays towards the light sources. A light source is visible only if the shadow ray does not hit anything. If the shadow ray hits something on its way to the light source, the object is in shadow and we shall not take the contribution from the light source into account.



Add code for tracing shadow rays to your implementation of the `directIllumination` function. This should only be a small add on and does not require any major changes. The new output is shown in the image on the right side on the previous page.

### *Hints*

```
Ray r = getShadowRay(is,l); Setup a shadow ray towards light source
bool hit = mScene->intersect(r); Returns true if the ray hit something
```

## 3. Reflections

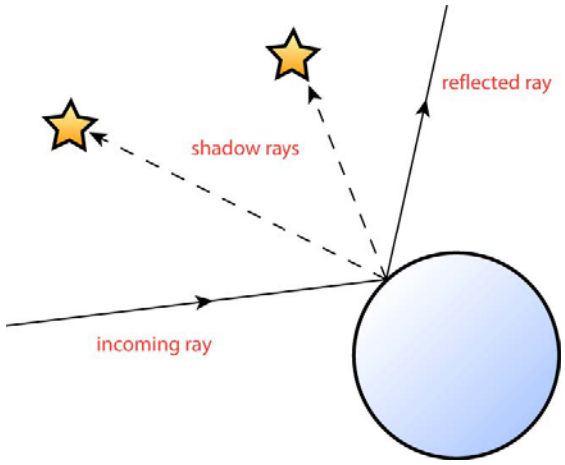
The next step to achieve more realistic images is to add reflections. Similar to how shadows

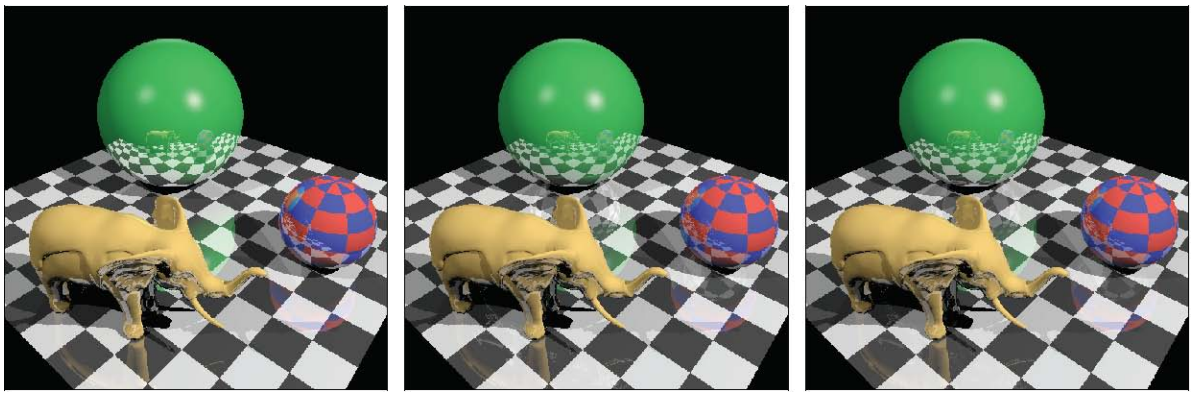
where computed in the last exercise, we do this by tracing reflection rays. There are however a number of important differences. When the reflected ray hits something, we need to compute the direct illumination at the hit point and possibly trace a second reflection ray, and so on. Since it is possible for the raytracer to get stuck in an infinite series of reflections, we need some stopping criteria. The simplest is to terrminate the raytracer at a fixed depth.

First, we need to turn on reflections for the materials in our scene. Go to the function `buildTestScene` in `main.cpp`, and uncomment the lines that say:

xxx->setReflectivity(...);

For the materials called `matteOrange`, `shinyGreen`, `matteBlackWhite`, `matteRedBlue`. Now, implement reflections by modifying your `trace(ray,depth)` function to trace reflection rays recursively down to a certain depth cutoff. Try with a low number first, for example stop when depth is 1 (only first reflection). A correct implementation should look like this:



`depth cutoff = 1` `depth cutoff = 2` `depth cutoff = 3`

In the first image, first reflections appear but the secondary reflections are missing (look under the green ball). There are only small differences when going up to depth 3 or higher.

The amount of reflection is determined by the material of the object where the ray hits. The following function returns the reflection coefficient at the hit point:

```
float r = is.mMaterial->getReflectivity(is);
```

The reflection coefficient is a number between 0.0 and 1.0, where 0.0 means not reflective at all and 1.0 means a perfect mirror. For reflective materials, the number is usually somewhere in between. To get the correct result, the light from the direct illumination must be weighted by the light received from the reflection, so that a proportion r is taken from the reflection and (1-r) from the direct illumination. Hence, the output color should be:

```
out = (1-r)*direct + r*reflected;
```

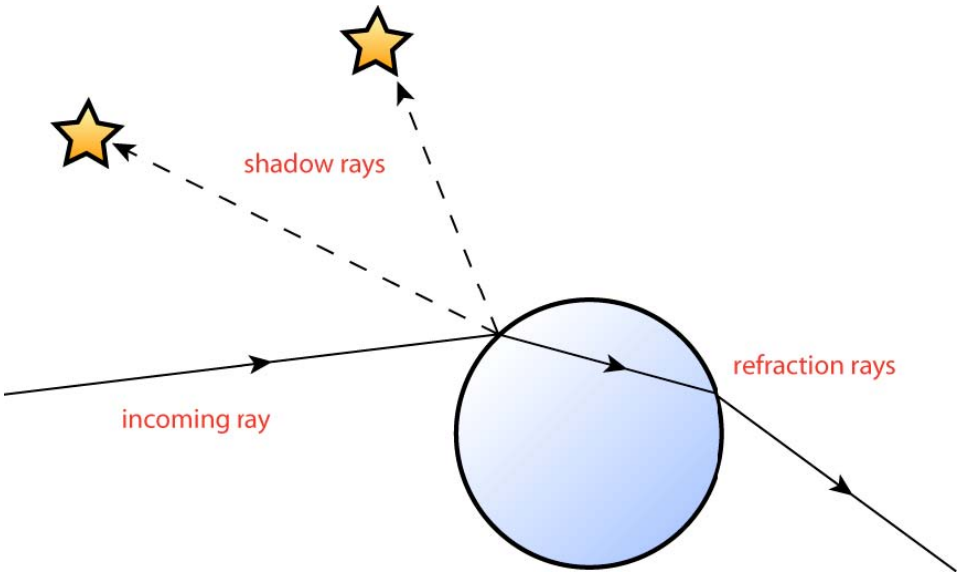
Note that the reflection ray should only be traced if the reflection coefficient is larger than 0, otherwise the material is not reflective and it is completely unecessary to trace reflection rays. Also, do not forget to check that the depth is under the depth cutoff you are using. Otherwise the raytracer might never terminate.

***Hints***

Ray r1 = getReflectedRay(is); Creates a reflection ray Color
reflected = trace(r1,depth+1) Calls trace recursively to get reflection

## 4. Refractions

Another important feature of a raytracer is the ability to handle transparency and refractions. Many real materials are more or less transparent. Examples include glass, plastic, liquids. When light enters a transparent material, it is usually bent or refracted. How much is determined by index of refraction of the material. In this exercise we will simulate refractions by tracing refraction rays:




We start by preparing the materials. First, turn off reflections on these materials: `matteOrange`, `shinyGreen`, `matteRedBlue`, by commenting out the calls to

`setReflectivity` in `main.cpp` that we enabled in the previous exercise. Leave the reflections on for the material on the floor, `matteBlackWhite`. Now, enable transparency for the above mentioned materials by uncommenting the lines that read:

```
xxx->setTransparency(...);
xxx->setIndexOfRefraction(...);
```

In this exercise, you should add code for tracing refraction rays to your tracefunction. The implementation for this will be very similar to the code for tracing reflection rays. The computation of the refracted light should be done recursively in the same way as for the reflected light. It is useful to use a little higher depth cutoff, for example 5, to better see the effect. The amount of transparency for the material at the hit point is returned by:

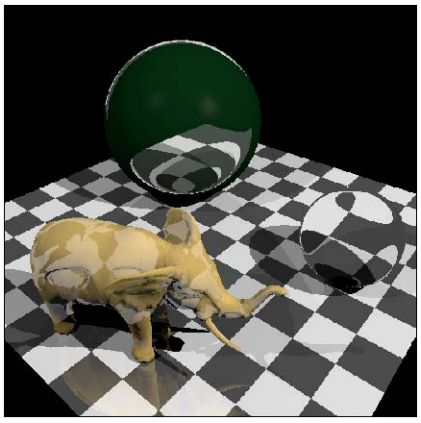
```
float t = is.mMaterial->getTransparency(is);
```

As earlier, this is a number between 0.0 and 1.0, where 0.0 means no transparency and 1.0 means fully transparent. Only if the transparency is larger than 0.0 do we need to trace refraction rays. We assume that a fraction t of the light should come from the refraction ray, and (1-t) from light computed previously (direct + reflected). Therefore, we get the following weighting of the different components:

```
out = (1-t) * ((1-r)*direct + r*reflection) + t * refracted;
```

Correctly implemented, the output will be the image on the right. The smaller ball has full transparency (1.0), the larger ball is almost completely transparent but has a hint of green, while the elephant is only half transparent (0.5).

Notice that the rendering is now a little slower than before since we are tracing many more rays. At each intersection point, we shoot 3 shadow rays, possibly one reflection ray, and one refraction ray. This is done recursively, so the number of rays can grow dramatically.

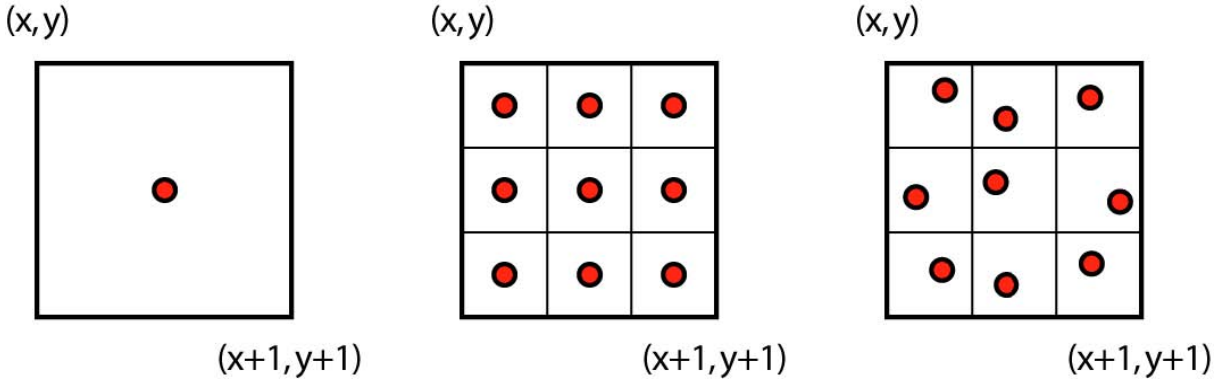


***Hints***

```
Ray r2 = getRefractedRay(is); Creates a refraction ray Color
refracted = trace(r2,depth+1) Calls trace recursively to get refraction
```

## 5. Supersampling

All of the images produced so far appear very aliased when zoomed in. This is because we are only tracing single ray through each pixel (remember the `tracePixel(x,y)` function). To get a smoother result, it is necessary to use some kind of supersampling. In this exercise you will implement a simple supersampling scheme to avoid aliasing. You may choose to implement either a uniform grid sampling or a stratified random sampling. If you want, you can do both.




Single sample | Uniform grid | Stratified random

Since supersampling is much slower than using a single sample, it is a good idea to turn off refractions and go back to one of the previous scenes while experimenting. It can also help to lower the depth cutoff. The images shown here were done on the scene from Exercise 4, with depth cutoff set to 2.

Implement supersampling by modifying the code in the `tracePixel(x,y)` function. The default implementation computes the coordinate of the center of the pixel (as shown in the left image above), and performs just one call to `trace(ray,depth)`. Change it so that a number of sample positions are computed (try with 2x2 or 3x3), and call the trace function for each of your new samples. The output color should be the *average* of the color returned by the samples, so you must not forget to divide by the number of samples.

See the next page for examples of what the output images will look like with 3x3 supersampling (9 samples) using the two methods: uniform grid and stratified random. Notice how much smoother the image is with supersampling enabled. The drawback is of course that the rendering time is 9 times as long as before.

## 6. Optional Exercises

If you want to further explore raytracing, there are a number of optional exercises you may try. Make sure you finish all the compulsory exercises first though. Suggestions are:

- Create a more interesting scene by adding objects, materials, lights, etc. You may

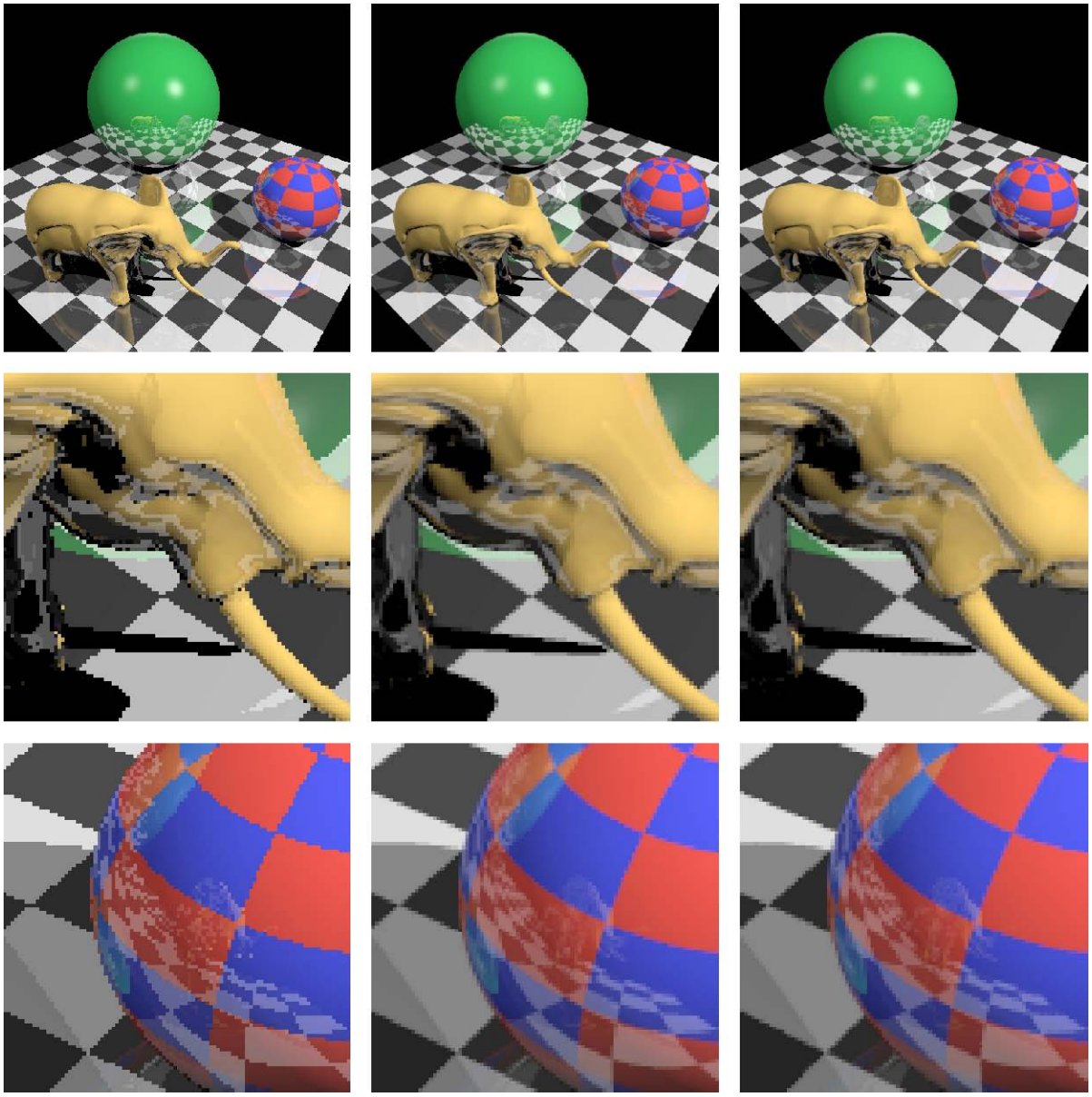
download additional meshes (.obj) files from http://www.cs.uoi.gr/~fudos/grad-exer1/objects/.

- Implement a better technique for depth cutoff in the raytracer. One idea is to terminate when the light contribution from the ray is too small to be noticeable.
- Try other schemes for supersampling.
- Any other extension of your choice.

## Conclusions

In this assignment you have implemented the core parts of a simple Whitted raytracer. Starting with a dull scene without shadows, reflections and refractions, all of these effects were added relatively easy. Finally, supersampling was implemented to reduce aliasing and making the images smoother. However, even an advanced Whitted-style raytracer produces images that look quite artificial.

***Supersampling Images***



Single sample | 3x3 Uniform grid | 3x3 Stratified random