Dor Gross, 039344999

Computer Graphics - Exercise 3

Notes and Comments about the Implementation

Requirements:

I have implemented *all* the basic required parts according to the *Requirements* section in the homework, including:

- Background (color and image)
- Camera control and super sampling
- Display sphere and triangle meshes
- Basic lighting (ambient, omni, spot, self-emission, and material factors)
- Basic shadows
- Reflective surfaces

Please see the appendix discussing the implementation structure.

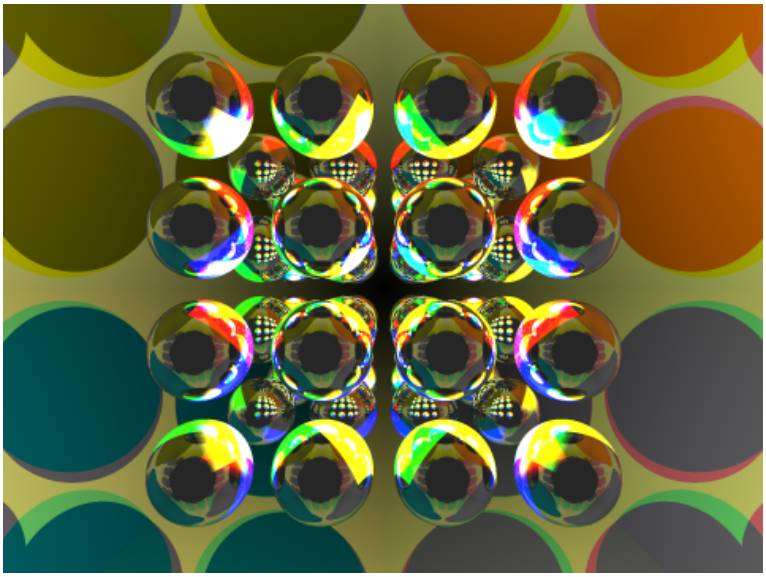
Extras:

In addition, I've also implemented the following parts (will also be discussed later):

- Ray refraction (transparency) and background image support in refraction.
- Texture mapping for spheres
- Additional geometric in space: Cube, Pyramid, Circle (Ring)
- Acceleration: was not implemented fully.

Each extra functionality will be discussed in its own page below.

In addition, I've created a nice scene combining most of the required functionalities. It is called "*complex.xml*" and it was originally created to check the acceleration, but since it wasn't implemented fully, it only remained as a very nice image showing the abilities of ray tracing:



Ray Refraction

Added a support in ray refraction.

Each element can add a factor of refraction to its values (along with the other material attributes). To add transparency, simply use the "*transparency*" attribute which receives a floating point value in the range [0-1]. Default transparency was set to 0, i.e., no refraction.

Apart from the refraction itself, the RayTracer also takes care of delivering the background image, along with the refracted ray, to the next level (recursion depth) of the the tracing, so the background image would remain the background of the refracted ray.

For a nice image showing the refraction abilities, please check *transparency.xml* scene - it contains a partially transparent red game-cube, a partially transparent blue sphere, a mirror and a background image which is visible through the transparent cube:

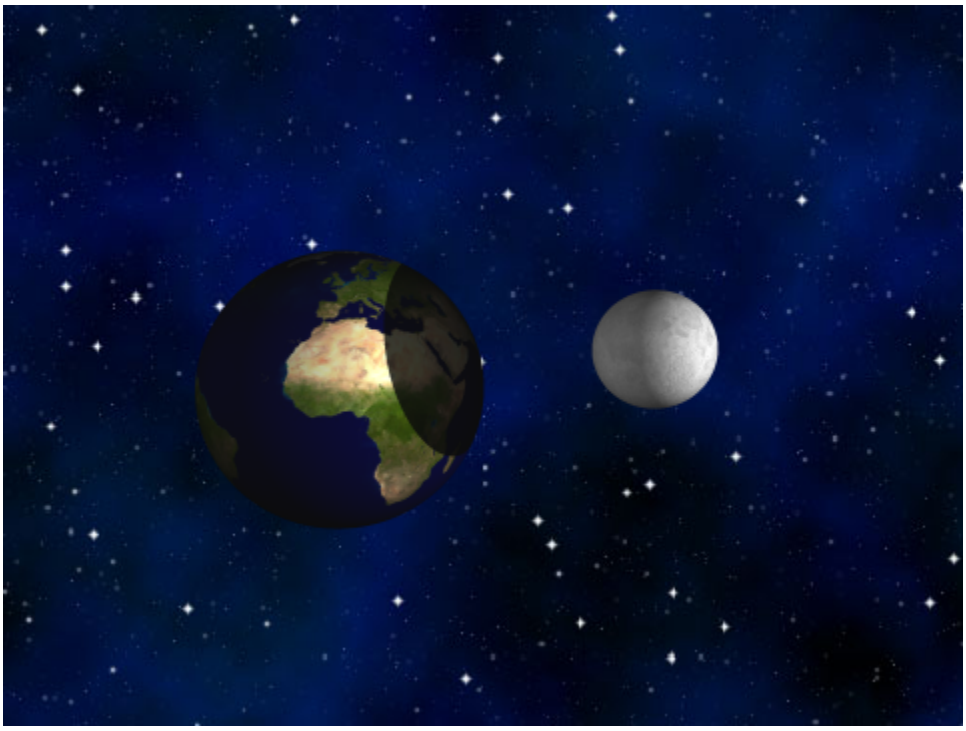
Texture Mapping for Spheres

Added a support for texture mapping over spheres.

You can use the "*mtl-texture*" attribute to determine the texture which lies on the sphere. In addition, controlling the face of the texture is done by using the "*texture-center*" and "*texture-up*" attributes which control the location of the center of the texture and its up-direction (where the top of the texture is).

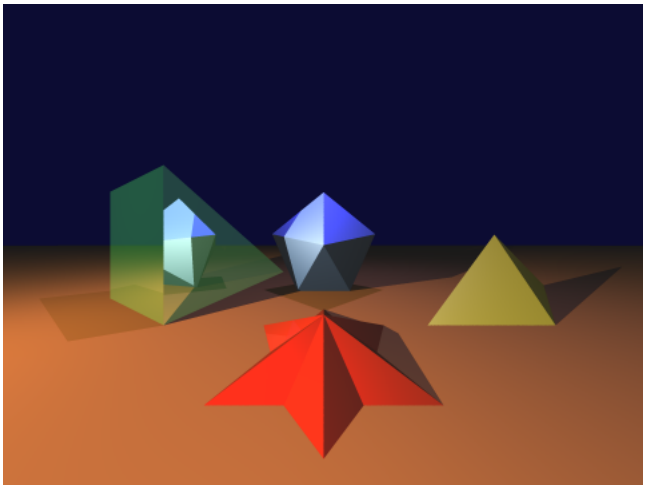
The default values for the texture directions are (0, 0, 1) for the center vector, and (0, 1, 0) for the up vector.

For a nice image showing the texture mapping ability, please see *space.xml* scene - it describes the Earth and the Moon during an eclipse. Both elements are textured layered with surface pictures downloaded from the internet:



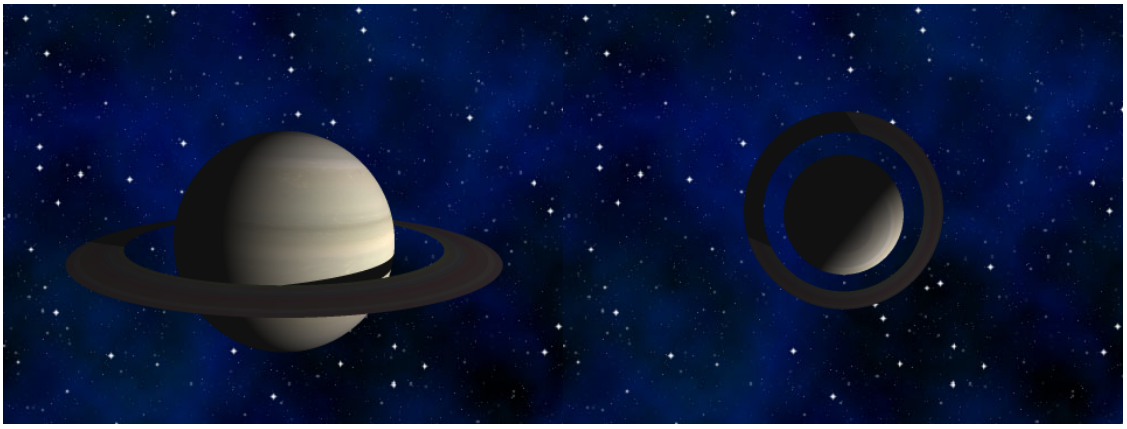
Additional Geometrics in Space

Apart from the Sphere, Triangle and Trimesh that we were requested to implement, I've also added the following geometrics:

- **Pyramid**
  Represents a Pyramid with a polygonal base shape. Its base can be a triangle, a square, a pentagon, and even the star of david. It gets it apex and the list of counter-clockwise vertices (which should be lying on the same plain), and it constructs the triangles on its own. See the *pyramids.xml* scene for a nice image using pyramids of all kinds and in different directions: A simple triangular pyramid, a lying squared pyramid, a standing pentagon and a Star-of-David base shaped pyramid:
  
  To create a pyramid, use the "*pyramid*" XML element, and make sure you provide the "*apex*" and "*base*" attributes in the XML.

- **Aligned Box**
  A box aligned with the axis represented by the maximal and minimal values of its X, Y and Z values. An example of the box can be seen in the *transparency.xml* showed above (the gamecube is actually an aligned box).
  To create an aligned box, you should use the "*box*" XML element, and make sure you provide the "*vertices*" attribute holding the 2 points the box is defined by.

- **Circle (Ring)**
  A flat circle in space. It can be partially empty or a full circle. A circle (ring) is defined by center, radius, inner radius, and directions (for the plain the circle lies on).
  You can see the circle in the *transparency.xml* scene above (as the dots on the gamecube), and as rings (with an inner empty radius) in the *saturn-front.xml* and the *saturn-above.xml* - showing the planet Saturn with it rings, which are actually made of several attached rings (circles) in several different angles:



Acceleration

Was not implemented fully - I didn't create bounding boxes.

However, in order to improve a little bit the rendering speed, I've used a bounding box surrounding each Trimesh.
Byt using the "*use-acceleration*" attribute with the value "*1*", the Trimesh creates a bounding box surrounding it and first check if the box is intersected with the given ray, and only if it does, it iterates over its triangles.

The *pyramids.xml* scene shows how this speeds up the rendering, when without the acceleration (i.e., *use-acceleration="0"* or missing), the scene is rendered about 3 times slower than it is with the acceleration. On my computer, the difference was ~45 seconds without acceleration and ~15 seconds with acceleration.

Appendix - Packages Structure

I've rearranges the structure of the packages a little bit, so it would fit better to the design I used. As I told you (Micheal) after the recitation, I already started implementing the exercise before the basic additional skeleton was supplied, so the structure is a bit different from the one most of the student will probably use.

A short description of each package and its content:

ex3.utils:

Utils abstract class:
Provides utilities methods and constants.
Initable interface:
An interface for all classes which can be initialized from a map of values.

ex3.math:

Point3D, Vec, Ray, RGB classes:
Each describes it real entity in the scene. I decided to differentiate between the RGB class and the other mathematical instances for clarity and simplicity in the code itself.
MatrixUtils class:
Knows how to solve a 3X4 matrix and returns the coefficient values. Used for finding the intersection point with a plain (triangle, circle, etc.) by applying Cramer's rule:
http://en.wikipedia.org/wiki/Cramer's_rule

ex3.parser:

SceneXMLParser class:
As I didn't create this class (was given with the exercise), I slightly modified it so it would create the elements as it finds them, in order to reduce the memory overhead and decoupling between this class and the SceneDescriptor class.

ex3.render.raytracer:

RayTracer class:
Added the following functionalities:

a. Initializes the Camera object for the scene.
b. Initializes the general attributes of the scene, such as the background, ambient light, etc.
c. Line rendering - considers super sampling and averaging between the rays' color through a single pixel.
d. Background color for all rays (for no intersection).
e. Background image for all first degree rays (from the camera) and all following refracted rays.
f. Finds the closest intersection point for each ray.
g. Omits occluded lights from every intersection point.
h. Includes diffusion, emission, ambient light, specularity, reflections and

transparency when determining the color of an intersection point.

ex3.render.scene:

Camera class:
A class representing the pinhole camera. It knows how to shoot rays into its screen.
SceneFactory class:
Follows the Factory design pattern, creates the elements in the scene according to their XML tag.
Hit class:
A simple POJO holding a hit related objects, such as the surface of intersection, the intersection point itself and the normal at the intersection point. Though some of the values can be recalculated by the others (e.g, the normal), it makes the code more readable and efficient.

ex3.render.scene.element:

SceneElement interface and AbstratSceneElement abstract class:
Represent a surface/element in the scene.
Material class:
A POJO that handles material's attributes, such as shininess, diffusion, emission, etc.
Intersectable interface:
An interface defining a surface that may be intersected with a ray.
Plain class and BoundBox class:
Provide simple geometric primitives with the ability to perform intersection with.
Sphere, Triangle, Trimesh, Circle, AlignedBox and Pyrmid classes:
All the supported geometric in space.

ex3.render.scene.light:

Light abstract class:
An abstract class representing a Light in the scene. Supplies some functionalities for calculating the light's values for a single point in space.
OmniLight and SpotLight classes:
Concrete classes representing light objects in the scene.