Computer Graphics SoSe16
Exercise 3

Tobias Knopp & Martin Hofmann
Institut for Biomedical Imaging

Next exercises:
23.05.2016, 14:15-15:45
30.05.2016, 14:45-15:45

In the last two exercises we introduced the basic tools to set up scenes, transform Objects and set up the camera position. The last step necessary for the creation of computer graphics is the coloring of individual pixels, which was done by the `PyPlot` package so far. In this exercise we will implement an image-ordered renderer (ray tracer), where one pixel at a time is considered and all objects influencing the pixel are found to generate a color for that pixel.

1. We start by defining a camera that captures the scene. Based on the human perception one often uses pinhole cameras in computer graphics. Such a pinhole camera is defined by a screen with width `w`, height `h` and resolution `nx`×`ny` and an eye, which is located at $(0, 0, d)$ in camera coordinates (the screen center coincides with the origin of the camera coordinates). The viewing direction is $-z$ in the camera coordinate system.

   In ray tracing for each pixel a ray is shot from the eye position through the pixel into the scene. Then all objects objects intersecting this ray are found and the final color of the pixel is calculated.

   Similar to the `OrthoCamera` it is convenient to store the camera to world and world to camera transformations. Use the position of the center of the screen `rc`, its viewing direction `rv` and its upwards direction `rv` to complete the function `PinholeCamera` in the following code snippet.

```
abstract Camera

type PinholeCamera <: Camera
    camToWorld::Transformation
    worldToCam::Transformation
    # screen resolution in x direction
    nx::Int
    # screen resulution in y direction
    ny::Int
    # screen width
    w::Float32
    # screen height
    h::Float32
```

```
    # distance eye screen
    d::Float32

    PinholeCamera(camToWorld::Transformation,worldToCam::Transformation) =
        new(camToWorld,worldToCam,800,800,2.0f0,2.0f0,2.0f0)
end

function PinholeCamera(rc::Vector{Float32},rv::Vector{Float32},ru::Vector{
    Float32})
    # your code here
    return PinholeCamera(camToWorld,worldToCam)
end
```

2. The central objects in ray tracing are rays. These are characterized by a direction and an origin. In homogeneous coordinates directions have their last component set to 0, whereas points have their last component set to 1. This ensures the correct transformation of vectors and points.

   In `julia` we implement rays as a new type

```
type Ray
    origin::Vec4f
    direction::Vec4f
end
```

   Write a function `generateRay(camera::PinholeCamera, i::Int, j::Int)` which generates an initial ray (origin and direction in world coordinates) running through the eye and the pixel (i, j) and has its `origin` coinciding with the pixel position and a normalized `direction`.

3. To be able to draw a scene, we need scene objects and a methods to intersect rays with these objects.

   As Objects we introduce a sphere (`Sphere`) and an axis aligned bounding box (`AABB`). Both types are subtypes of `SceneObject`.

```
abstract SceneObject

type Sphere <: SceneObject
    center::Vec4f
    radius::Float32
end

Sphere(center::Vector{Float32},r::Float32) = Sphere(Vec4f(center[1],center
    [2],center[3],1),r)

type AABB <: SceneObject
    center::Vec4f
    # positive half length from center to face of box
    hx::Float32
    hy::Float32
    hz::Float32
end
```

```
17
18 AABB(center::Vector{Float32},hx::Float32,hy::Float32,hz::Float32) = AABB(
       Vec4f(center[1],center[2],center[3],1),hx,hy,hz)
```

Mathematically a ray can be describes as the set $R = \left\{\vec{o} + t\vec{d} \mid t \in \mathbb{R}\right\} \subset \mathbb{R}^3$, where $\vec{o} \in \mathbb{R}^3$ and $\vec{d} \in \mathbb{R}^3$ given by the first three components of `origin` and `direction` respectively. A point is in-front of the screen if $t > 0$ and behind the screen if $t < 0$.

Let $O \subseteq \mathbb{R}^3$ be an object, then a ray R intersects this object if $R \cap O = RO \neq \emptyset$. For the Objects we use RO will be line segments if non empty. These segments are fully described by $t_{in} < t_{out}$, which correspond to the ray points where the segment starts and ends. For which (t_{in}, t_{out}) is the object in front or behind of the screen? When is the object intersecting the screen?

Write two intersect functions

```
1 function intersect(ray::Ray,sphere::Sphere)
2     # intersection code
3 end
4
5 function intersect(ray::Ray,aabb::AABB)
6     # intersection code
7 end
```

which return a boolean statement, if objects are visible (in front of screen) and returns t_{in} as the second return value. If the object is not visible t_{in} may be set to 0.

4. To fully describe the scene it is convenient to create a new type to contain all the object we want to render.

```
1 type Scene
2     sceneObjects::Vector{SceneObject}
3 end
```

Write a function `intersect(ray::Ray,scene::Scene)` which loops over all objects in `scene` and finds the object closest to the screen. Return (`return hit, tin, objhit`) if any object was hit in the first place, the parameter t_{in} describing the hit point and the object which was hit (return `nothing` if no object was hit).

Write a shader `hitShader(ray::Ray,scene::Scene)`, which returns `1.0f0` if any object was hit and `0.0f0` else.

Patch together all pieces into the now very simple ray tracing function

```
1  function tracerays(scene::Scene,camera::Camera,shader::Function)
2      nx = camera.nx
3      ny = camera.ny
4      screen = Array(Float32,nx,ny)
5      for i=1:nx
6          for j=1:ny
7              # generate ray for pixel i,j
8              ray = generateRay(camera, i, j)
9              # use shader function to calculate pixel value
10             screen[i,j] = shader(ray, scene)
```

```
        end
    end
    # final visualization of image
    figure()
    gray()
    imshow(screen')
    colorbar()
end
```

and render the following scene:

```
# set up individual objects
sphere1 = Sphere(Float32[-0.5,0.5,0],0.25f0)
sphere2 = Sphere(Float32[-0.5,-0.5,0],0.5f0)
aabb1 = AABB(Float32[0.5,-0.5,0],0.25f0,0.25f0,0.25f0)
aabb2 = AABB(Float32[0.5,0.5,0],0.5f0,0.5f0,0.5f0)
# set up scene
scene = Scene(SceneObject[sphere1,sphere2,aabb1,aabb2])

# set up camera
camera = PinholeCamera(Float32[0,0,1],Float32[0,0,-1],Float32[0,1,0])

# render scene
tracerays(scene, camera, hitShader)
```

Set up the camera to different positions as You like to rerender the scene.

5. The result of the first rendering looks somewhat binary. For a more realistic rendering we need to add some light and a more advanced shader.

   Before we do so we need a functions to calculate the surface normal where an object is hit. Implement a function for `Sphere` and `AABB` which calculates the surface normal (return as `Vec4f` in homogeneous coordinates) based on the hit point.

```
function surfaceNormal(ray::Ray,t::Float32,sphere::Sphere)
    # code
end

function surfaceNormal(ray::Ray,t::Float32,aabb::AABB)
    # code
end

# return zero vector if no object is hit
surfaceNormal(ray::Ray,t::Float32,void::Void) = Vec4f(0,0,0,0)
```

   To make the scene more realistic we also need a light source. Therefore we add a new type for point like light sources

```
abstract Lights

type PointLights <: Lights
    positions::Vector{Vec4f}
end
```

Modify the function `tracerays` to be able to accept `lights::Lights` as additional argument and pass this argument to the shader `screen[i,j] = shader(ray, scene, lights)`

Implement a Lambertian shading and ambient lighting by writing the function `lambertShader(ray::Ray,scene::Scene,lights::Lights)`. Ambient light refers to diffuse light which illuminates all surfaces irrespective of their orientation (`shade = 1.0f0`). Lambert shading creates the lighting of a non reflective surface hit by light at a certain angle, which is proportional to the dot product of normalized surface normal and the normalized direction from hit point to light source. Take into account that the light source has to be in front of the surface. Use this shader to rerender all scenes with the default and your custom camera settings.

What would be necessary to create an even more realistic lighting?