RayTracer

IMPORTANT NOTE: Documentation utilizes LaTeX. For the best experience download readme.pdf

Basic info

- **Author:** Vojtech Proschl
- **Course:** NPRG031
- **Term:** Summer term 2021

Main resources

- Fundamentals of Computer Graphics
- Linearni Algebra (nejen) pro Informatiky

Short describtion

The goal of this program is to allow a user to render 3D scenes that are composed out of primitive objects. Users can adjust various parameters of the sceen that alter the appearance of the render. This project implements raytracing algorithm.

User documentation

Scene settings

Camera
Position X,Y,Z - Position of the camera
Rotation X,Y,Z - Rotation of the camera around X,Y,Z axis (canonical basis)
Lens Angle - Field of View
Projection Distance - Distance between projection plane and camera origin

Light
There are 3 lights in the scene
Position X,Y,Z - Position of the light
Specular R,G,B - Specular highlight color
Intensity - Intensity of the light

Render Settings
Resolution X,Y - Resolution of the render
Phong Exponent - Info
Background R,G,B - Color of the background
Reflection Intensity - Ammount of the reflected light is multiplied by this number, 1 ⇒ 100% reflected light, 0 ⇒ 0% reflected light
Ray Count - Reflection recursion depth
Sample Size - Supersampling grid size, 1⇒1×1, 2⇒2×2... Useful for antialiasing but it is computationaly expenssive

Slider
Zooming the render

Export
Opens dialog window where you can save render

Load Primitives
Its purpose is to import scenes into the render engine. The scene has to be in JSON and has to have the following syntax. There are 2 types of geometry currently supported

- Cube
- Sphere

Rendering box is way slower than rendering sphere because it's defined as polygonal geometry (set of triangles).

Input data format
Input has to have this exact structure. List elements have to be spheres and cubes. Other shapes can be easily added. You can add any ammount of those shapes.

```
{
  "Geometry":[
    {
      "Type": "Sphere",
      "PosX": 2,
      "PosY": 2,
      "PosZ": 2,
      "Red": 255,
      "Green": 0,
      "Blue": 0,
      "Radius": 1
    },
    {
      "Type": "Cube",
      "PosX": -2,
      "PosY": -2,
      "PosZ": -2,
      "Red": 0,
      "Green": 255,
      "Blue": 255,
      "SizeX": 1,
      "SizeY": 1,
      "SizeZ": 1
    }
  ]
}
```

Sphere:

- Type: Sphere
- PosX - double precission
- PosY - double precission
- PosZ - double precission
- Red - 0 to 255 int
- Green - 0 to 255 int
- Blue - 0 to 255 int
- Radius - double precission (positive)

Cube:

- Type: Cube
- PosX - double precission
- PosY - double precission
- PosZ - double precission
- Red - 0 to 255 int
- Green - 0 to 255 int
- Blue - 0 to 255 int
- SizeX - double precission (positive)
- SizeY - double precission (positive)
- SizeZ - double precission (positive)

Programmer documentation

Program structure

RayTracer
RayTracer is the main class of the program. It connects the main user interface with scene and its simulation using the

RayTracing algorithm which is in the simulation class.

Scena
Namespace scene includes all elements that are important in the simulation:

- **Camera** - The camera is described with attributes. Projection rays are generated based on them as well.
- **Geometry** - This class stores geometry
- **Light** - This class stores attributes of light
- **Scene** - This class serves as a container for scene elements (e.g. lights, camera...)

Render
Render is namespace which contains classes that take care of RayTracing algorithm

- RayGeneration
- SphereIntersection
- TriangleIntersection
- ImprovedBlinnPhong
- Simulation

Utilities
Utilities is the namespace that contains various types of geometry representation (data representation). There is also a class responsible for reading input .json files and a class that is responsible for controling user input (GUI).

User interface
The user interface is created using WindowsForms. Class SceneSettings takes care of transfering parameters from GUI to the scene.

At the moment software supports only 3 lights (due to the fixed GUI). But software is built in such a way that you can easily add as many lights as you want.

Input and outut
The input of the program is a .json file which represents the scene and GUI where the user can specify parameters of the simulation, lights etc.
Output is .BMP file of the rendered image

Data interpretation

Internal

Scene
Scene stores elements of the simulation such as geometry, lights, camera, etc. Those objects have their own parameters which are used during the simulation.

Sphere
Parametricaly described geometry time using its origin and radius.

Mesh
Mesh is other possible representation, which is more complex than the sphere. It is composed out of vertex array which can form triangles. Each triangle has its normal (Support for vertex normals can be easily added).

Cube
Is made using the mesh geometry type. It is composed out of 8 vertices and 12 triangles.

Input & Output
Described in the user documentation

Algorithm Choice

RayTracing
Is made out of three basic steps

- Computing view rays
- Ray-Geometry Intersection
- Shading

Each step is computed independently for each pixel → It's running parallel.

The most time demmanding step is the second step (Ray-Geometry intersection). There is room form improvement (last part of the documentation).

Projection plane
Intersection rays are generated based on the projection plane. Its size is calculated from the camera angle and pixel amount in the camera settings.

Computing view rays
The camera object has rotation attributes (one for each axis). Based on those attributes we rotate the orthonormal basis of the camera (which stays ortonormal because rotation matrix is an orthonormal matrix and orthonormal times orthonormal is still orthonormal). We're creating projection plane with this new orthonormal basis. View rays are using the projection plane.

Ray-Geometry intersection
There are 2 algorithms for the geometry intersection. The first one is ray sphere intersection and the second one is ray triangle intersection.

Shading
During this phase, we're computing appearance of the pixel. It's based on the attributes of the scene objects.

Pixel Sampling
Pixel sampling is used to improve the quality of the render by computing subpixels and averaging the resulting value. It can effectively remove aliasing.

Shading modely

Basic shading model
It is an algorithm that takes results of the previous steps, lights, etc., and based on that computes the appearance of a pixel.

We usually have multiple lights in the scene and the shading model is adapted for that. Let's say we have a set of lights $\{L_1, \dots, L_n\}$, where the lights have (not necessarily) different parameters.

Shading parameters:

- Intersection location $p \in \mathbb{R}^3$
- Normal of the intersected geometry in the place of intersection $n \in \mathbb{R}^3, ||n||_2 = 1$
- Camera location in space $c \in \mathbb{R}^3$
- Distance between the camera and the intersection $d \in \mathbb{R}$
- Color of the intersected geometry $C_G \in [255]^3$
- Projection vector $v \in \mathbb{R}^3, ||v||_2 = 1$
- Vector pointing from intersection to the light $l \in \mathbb{R}^3, ||l||_2 = 1$
- Phong exponenet $P \in (0, +\infty)$

Light parameters

- Light position $L_p \in \mathbb{R}^3$
- Light intensity $L_I \in \mathbb{R}^+$
- Light specular color $C_S \in [255]^3$

Pixel shading algorithm
Based on the Blinn-Phong model
During this phase, we compute the appearance of a single pixel.

This part of the algorithm solves color without the light reflection

- $C := (0, 0, 0)^T$
- foreach $L \in \{L_1, \dots, L_n\}$:
  - $h := ||v + l||^{-1} \cdot (v + l)$
  - $C := C + L_I \cdot max(0, n^T \cdot l) \cdot C_G + L_I \cdot max(0, n^T \cdot h) \cdot C_S$
- return $(max(C_1, 255), max(C_2, 255), max(C_3, 255))$

Using trivial improvement we're able to add reflection to the algorithm. We will add a recursive call that computes the color of the pixel in the reflection direction. Variable recursion depth can be added easily.

Reflection intensity $I_R \in [0, 1]$

- $r := v - 2(v^T \cdot n)n$
- $parameters := RayIntersection(r, p)$
- $C := C + I_R * PixelShade(parameters)$

Variable r is the direction from which the human eye (or the camera) sees reflection.

Possible future improvements

Spatial Data Structures
Due to the mentioned problem with ray-triangle intersection speed it would be beneficial to group geometry spatialy and compute intersection based on bounding boxes. This could be achieved by implementing spatial search data structure such as Octree. This should improve render speed dramaticaly.

Shading model improvement
It would be great to add new functions to the implemented shading model such as transparency, light refraction, etc.

The addition of the HDR env maps would be also beneficial. Using this the renders would look way more realistic and it would simplify the process of setting up lights.

Adding different types of light would be also great (e.g. spotlight).

More complex geometry
At the moment program supports only cubes and spheres. Addition of new primitive shapes should be easy.

There is support for polygonal geometry but there is a need to implement parsing of the geometry.