Boris Kravchenko

bkravche

CS488

Final Project: Distributed Raytracer

20332359

Due: Monday, April 8th [Week 13].

Name:

UserID:

Student ID:

Extra objective from A4: Multithreading

__1: Depth of field is implemented

__2: Glossy reflections are implemented

__3: CSG is implemented, including union, intersection and difference

__4: Bump mapping is implemented

__5: Mirror reflections are implemented

__6: Refractions are implemented, and are physically correct according to Fresnel's equations

__7: Super sampling anti-aliasing is implemented, with no jagged edges visible

__8: Area lights and soft shadows are implemented

__9: The final scene is modeled

__10: Additional primitives are added, including cylinder and cone

Declaration:

I have read the statements regarding cheating in the CS488/688 course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature:

Manual

FEATURES

This application features an extension to assignment 4 ray tracer. New features include depth of field, CSG, glossy and mirror reflections, Fresnel-correct refractions, super sample antialiasing, soft shadows and additional primitives. I was unable to implement bump mapping due to running out of time.

EXTRA OBJECTIVES

Translucency: I made an extension to refractions to handle varying degrees of translucency.

DIRECTORY STRUCTURE

/a5/data	The .lua scene files, as well as all the .png image files generated by my ray tracer
/a5/src	The source code for the rt executable
/a5/doc	The documentation for the project

FINAL SCENE

My final scene features a car inside a white garage standing on a platform surrounded by water. Depth of field, soft shadows, reflections and translucency are all demonstrated by this scene. The car mesh is from the web, see implementation section.

HOW TO RUN

./rt scene.lua *n*

n	Number of threads you wish to use. Optional parameter, if not set, ray tracer uses same number of threads as number of cores on machine

LUA OVERVIEW

The LUA interface is identical to the A4 interface, except for the following

Materials are now either gr.phongMat or gr.transMat

gr.phongMat(*kd* , *ks* , *shininess* , *reflectivity* , *glossiness*)
gr.transMat(*kd* , *indexOfRefraction*, *transparency* , *scatter*)

kd	Triple specifying diffuse colour

ks	Triple specifying specular colour
shininess	Integer specifying shininess phong exponent
reflectivity [0 - 1]	Double specifying the fraction of light that's reflected
glossiness [0 - n]	Integer specifying the size of the cone that the reflection is distributed over
indexOfRefraction [0 - n]	Double specifying the index of refraction
transparency [0 - 1]	Double specifying the fraction of light that's refracted
scatter [0 – n]	Integer specifying the size of the cone that the refraction and reflection is distributed over

Constructive solid geometry can be specified recursively to build complex solids.

grNodeA:csg(*grNodeB*, *operation*)

grNodeA	The left side of the binary CSG operation, must be a leaf grNode (contains geometry). This grNode is added to the scene tree (a child of another grNode such as root)
grNodeB	The right side of the binary CSG operation, must be a leaf grNode (contains geometry), must be non-hierarchical. This grNode should not be explicitly added to the scene tree
operation {'u', 'n', 'd'}	The CSG operation – union, intersection or difference

New rendering options

gr.render(///A4 parameters\\\, *aa* , *ss* , *dist* , *{ apetureSize , focalLength , dof }*)

aa {0, 1}	Controls whether or not super sample antialiasing is on or off
ss [0 - n]	Integer that controls the quality of soft shadows; 0 indicates no soft shadows, higher numbers give better image quality. *ss*^2 is the number of rays cast for each shadow calculation
dist [0 - n]	Integer that controls the quality of distribution ray tracing features (glossy reflections and translucency); 0 indicates no distribution ray tracing, higher numbers give better image quality. *dist*^2 is the number of rays cast for the averaging calculation
apetureSize	Double that controls the size of the aperture.
focalLength (0.01 – n]	Double that controls the focal length
dof	Integer that controls the quality of the depth of field effect; 0

Indicates it's off, higher numbers give better image quality. *dof*^2 is the number of rays cast for the DOF calculation

Lights

gr.light(///A4 parameters\\\, *{length,width} , normal , left*)

length (0 – n]	Double that specifies the length of the light
width (0 – n]	Double that specifies the width of the light
normal	A Vector3D that specifies the normal to the light surface. This is where the light is pointing
left	A Vector3D that specifies a vector that's orthonormal to the normal, this controls the rotation of the light

Additional Primitives – hierarchical cylinder and cone

gr.cone(*'name'*)
gr.cyllinder (*'name'*)

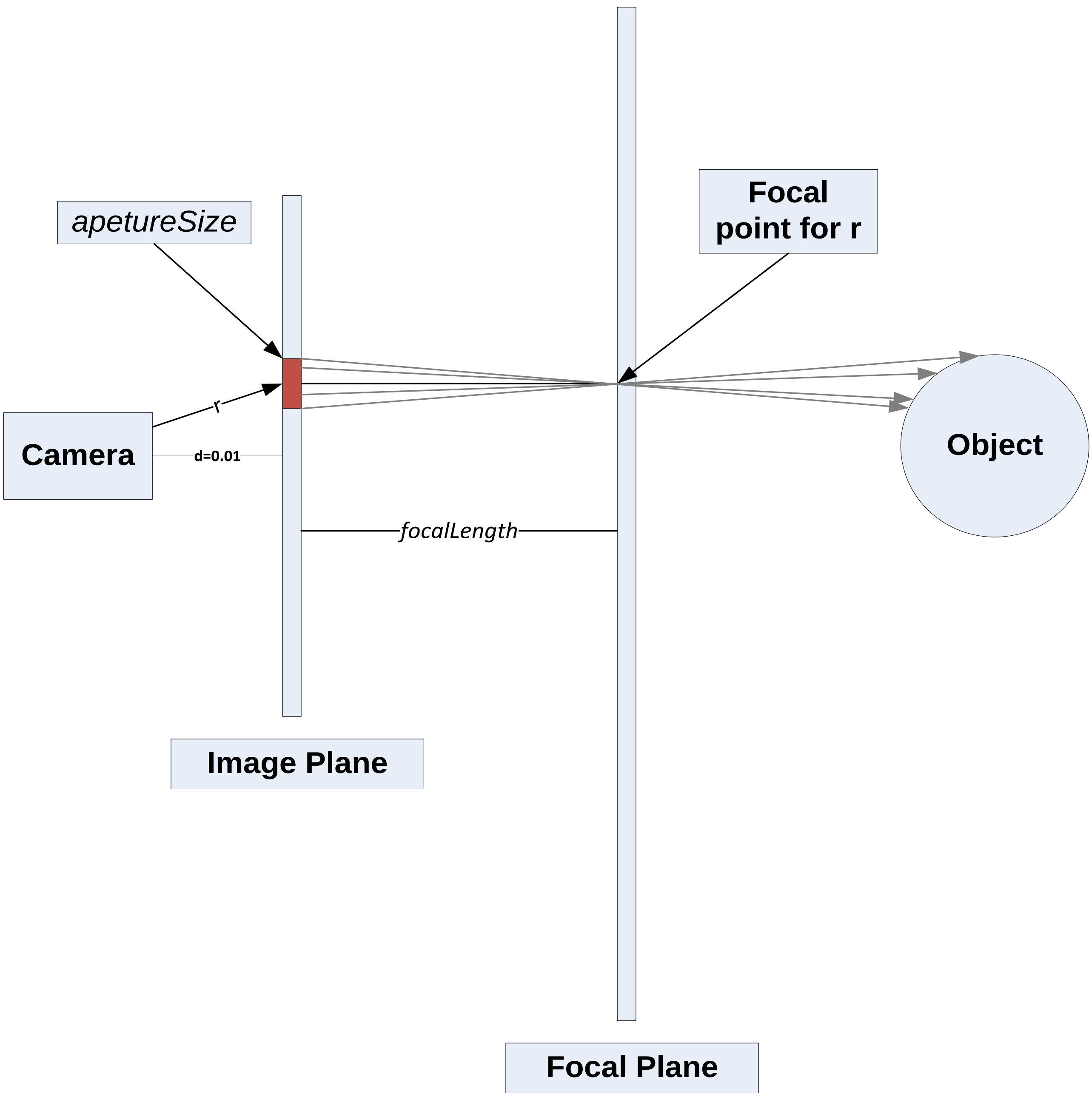
For examples of these new commands, please refer to .lua files inside the data directory.

IMPLEMENTATION

All images represent 2 dimensions

Depth of Field

Code: a4.cpp, line 593



Details: Depth of field is implemented exactly as shown in the image. The rays shot from the aperture square are offset by a random amount to avoid banding in the final image. All the resulting depth of field rays are averaged to get the colour of the original ray r.

Note: This effect is incredibly expensive, as for every pixel in the image multiple rays are shot to calculate the depth of field. Also, my approach is an oversimplification of the physically-correct effect.

Refractions

Code: a4.cpp, line 477

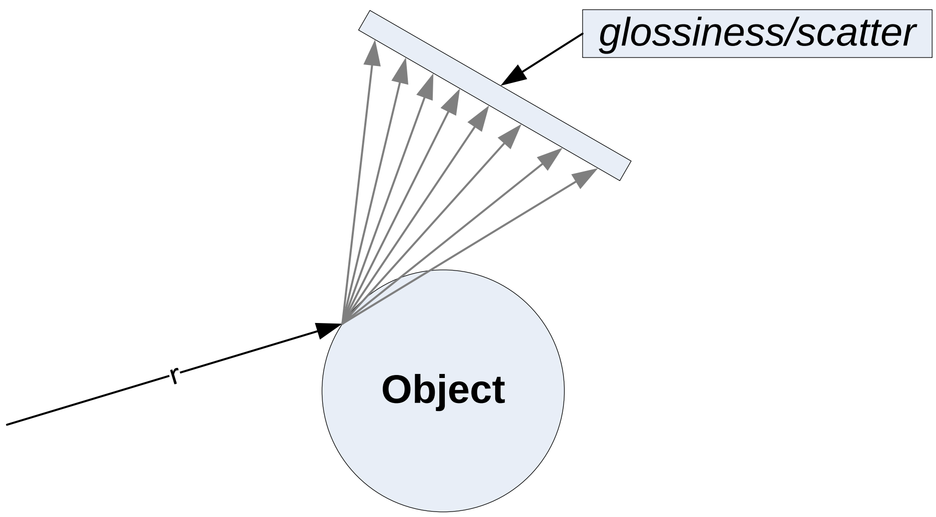
Details: Refractions are implemented according to Snell's laws, with the ray being bent according to the fraction of material indices. Every ray contains a stack with indices of refraction. The stack starts off with "1", since the ray starts off in air. Any time a ray enters a material a new index is pushed on, any time a ray leaves a material the index is popped off. This allows for recursive refractions, it's possible to have a sphere of water inside a glass sphere for example. The colour of refractive materials is implemented by only reflecting/refracting a *transparency* fraction of the ray, and multiplying the result by a factor of 1- *transparency* of the material colour.

Once the initial refraction ray is calculated, I use Fresnel's equation to calculate the fraction of light that's reflected back, as well as refracted. I'm assuming the light is not polarized. One interesting technical detail I encountered was the fact that Fresnel's equation returns many values that are out of range of [0, 1], and any value out of this range should be thrown out. Only values between [0, 1] correspond to actual reflectance values.

Bugs: There is an issue with doing refractions and transparency on cylinders and cones, the disks show up black. I did not have enough time to fix this issue.

Glossy reflections and translucency

Code: a4.cpp, line 356



Details: Glossy reflections and translucency are implemented as shown in image. The *glossiness* or *scatter* parameters control the size of the rectangle that the rays are distributed over. The *dist* parameter controls the number of rays shot over the horizontal and vertical sides of the rectangle. The rays are offset by a random amount to avoid banding. The results are averaged. For any distributed ray that ends up going through the object, the colour black is returned. Translucency is identical, except both reflection and refraction rays of a refractive object are distributed.

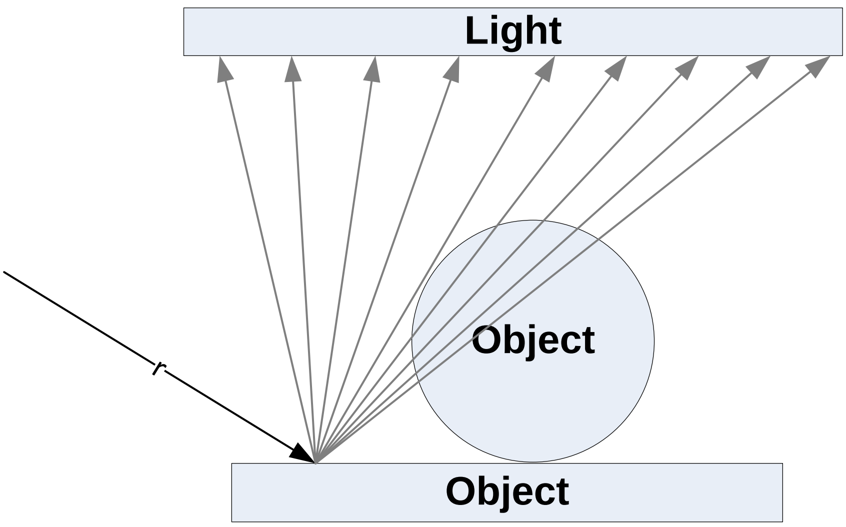
Constructive Solid Geometry

Code: scene.cpp, line 130

Details: Both objects in operation are intersected twice, which gives me two line segments for where the ray passes the object. The binary operation is performed on the segments, and the resulting tValue and normal are returned. An implementation difficulty I ran into was that the second object in the operation must be non-hierarchical, due to the fact that both segments must be relative to one ray, so the ray can't be transformed for one of the segments. Due to time constraints, I ended up making this a limitation of the feature. A possible future enhancement is improving the code to fix this limitation.

Soft Shadows

Code: a4.cpp, line 253



Details: For every ray/object intersection, secondary rays are cast out over the surface of the light. The amount of shadow for that ray is the fraction of secondary rays that aren't occluded by an object over the number of secondary rays shot out. The number of secondary rays shot over the width and length of the light is controlled by the *ss* parameter. Each ray is offset by a random amount to avoid banding in the final image.

Mirror Reflections

Code: a4.cpp, line 456

Details: I simply cast a recursive secondary ray when I encounter any reflective surface, *reflectivity* controls the fraction of light reflected.

Super sampling Antialiasing

Code: a4.cpp, line 156

Details: I simply render the image at double the specified resolution, and for every pixel in the final image, I average the 4 pixels in the double-sized image.

Additional Primitives

Code: primitive.cpp, 101, 204

Details: For cylinder, I first find where the ray intersects the infinite cylinder lying on the y axis, then I find the intersections with the top and bottom disks (height of the default cylinder is 2), and return the minimum tValue out of the 3 intersections. Cone is done in a very similar manner.

Areas for future improvement

Area lights: Area lights work great under the current implementation, except for the fact that the specular highlights are not correct – they are still circular, as for point lights. A nice future improvement would be to fix this. Further research is required, as I'm not sure if the Phong lighting model can be modified to realistically handle non-point lights.

Photon mapping/global illumination: This is also a nice feature for future improvement of the current renderer, because I feel that a lot of my images would have turned out much prettier with global illumination. I wanted to implement this as an extra, but ran out of time.

Shadows for refractive materials: Currently, my renderer ignores the shadows for refractive materials, because I wasn't sure how to make the shadow physically convincing. Implementing caustics would be ideal, as they would greatly improve image quality.

Performance: This is a crucial area for improvement, because the current renderer is painfully slow, especially if there's a large high poly mesh in the scene. A spatial subdivision algorithm would greatly improve the speed of this ray tracer.

References/Resources

Car for final scene taken from

http://www.turbosquid.com/3d-models/free-3ds-mode-car-fictional/640082

Fresnel Equations

http://en.wikipedia.org/wiki/Fresnel_equations

Snell’s Law and vector equations

http://en.wikipedia.org/wiki/Snell's_law

Ideas for depth of field implementation

http://cg.skeelogy.com/depth-of-field-using-raytracing/