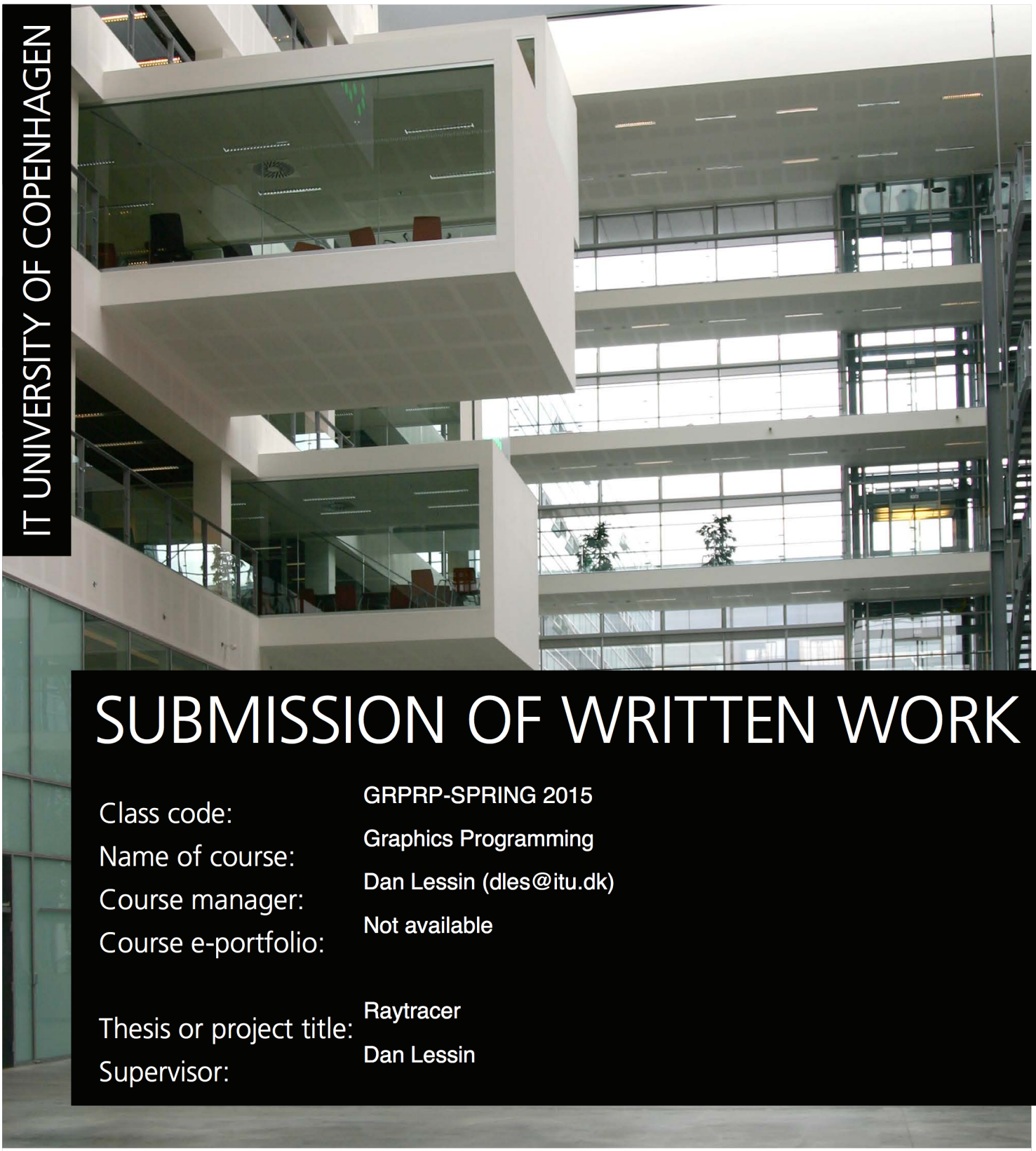
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:	GRPRP-SPRING 2015
Name of course:	Graphics Programming
Course manager:	Dan Lessin (dles@itu.dk)
Course e-portfolio:	Not available
Thesis or project title:	Raytracer
Supervisor:	Dan Lessin

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Anders Wind Steffensen	10/02-1993	awis@itu.dk
2.	Morten Albertsen	01/08-1988	moalb@itu.dk
3.	Rasmus Dyhr Larsen	31/12-1992	rady@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Raytracing
Graphics Programming
Spring Semester, 2015

IT-University of Copenhagen
Spring term 2015

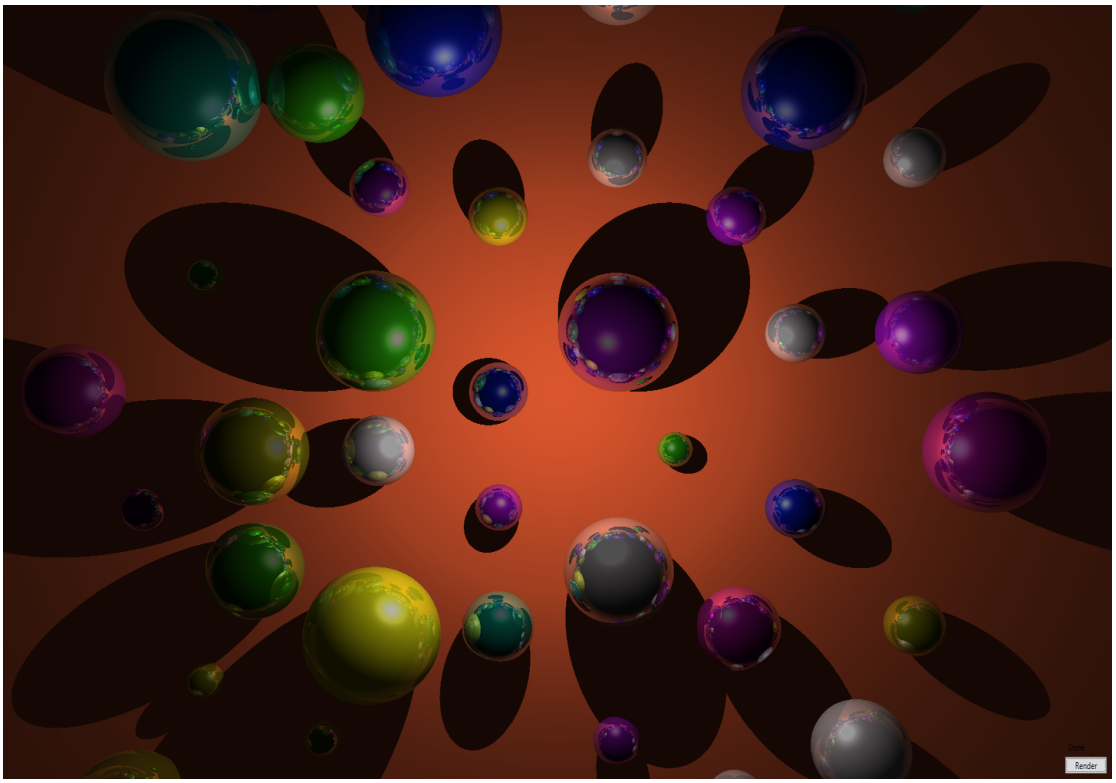


Figure 1: Screenshot from rendering produced by the delivered software

Authors:

Anders Wind Steffensen (awia)
Morten Albertsen (moalb)
Rasmus Dyhr Larsen (rady)

Github-repository:
https://github.itu.dk/moalb/Graphics-Programming
(requires a Github.itu.dk account)

Contents

1 Project Description

The project concerns the programming of a raytracer written in the programming language C++. This report details the project. To see sample renderings from the project, see Appendix B.

1.1 Raytracing explained

Ray tracing is a rendering technique used for producing realistic looking simulations ('*renderings*') based on how light rays will act within a scene of objects. Raytracing hence seeks to imitate the nature of light-rays bouncing around between and being refracted through objects. Rays travel from a light source, interacts with objects in the scene, through either reflection, absorption or refraction, and may eventually reach the viewer's eye (the 'camera'). In ray-tracing, however, to ignore the rays that won't reach the camera, only rays that will eventually reach the camera are processed. This is achieved by reversing the process. i.e. rays are shot from the camera into the scene, where they are allowed to be absorbed, bounce or refract a predetermined number of times. This allows for photo-realistic renderings, but at the cost of processing time.

The core of the ray tracing algorithm is based around intersections between lines (rays) and objects, all of which can be described mathematically, for instance spheres, planes and triangles. When an intersection is found, the shading at that intersection is calculated based on the normal to the surface at that point, and light sources of the scene, and then recursions can be made based on the law of reflection or refraction.

1.2 Planned features (scope and non-scope)

This section will introduce what we planned on implementing during the project span.

1.2.1 Objects

We planned to support three simple types of objects; spheres, triangles and planes. This will enable us to produce simple renderings, with objects that we're familiar with and have an intuition about how should look. Furthermore optionally we also planned to import .obj files and create a mesh of triangles.

1.2.2 Shaders

We planned to implement a basic phong lighting model, which should include shaders for ambient, diffuse, and specular lighting.

1.2.3 Lights

We planned to provide directional lights, point lights and ambient lights. Optionally we would support colored lights, as opposed to the more simple model, where lights just emit white light.

1.2.4 Shadows

To provide a better sense of realism, we planned to include shadows on objects. We will implement these in two stages; first the easier hard shadow, and secondly soft shadows, where gradients are seen across shadows.

1.2.5 Reflection

Reflections are the next obvious candidate to implement to mimic reflective objects in the real world, such as glass and polished metal. Reflection will require recursion in order to work, as new reflection rays are created.

1.2.6 Refraction

For transparent objects, we wish to support refraction, again to produce more realistic final renderings.

1.2.7 GPU utilization

Ray tracing can provide really nice looking renderings, but at the cost of a huge processing load. Since the algorithm does a lot of the same calculations which does not affecting one another, it would be easy to parallelize with multithreading. As a better alternative we would look into using the GPU, with OpenGL or similar, which is more specialized at doing a lot of calculations, and will provide much faster results.

1.2.8 Textures (Optional)

Textures allow for more interesting objects and is a core part of any engine, therefore textures could be a possible optional feature.

1.2.9 Maps (Optional)

Maps such as bump maps and normal maps increases the realism of flat objects by altering the normals and shaders of the object. This creates more interesting surfaces and would be a cool additional feature, which could be easy to implement on top of textures.

2 Implemented features

This section describes the solution, that was handed in. The core rendering engine was written in C++, and a (minimal) UI was written in C#. Also note that the team did not have any prior experience in writing C++.

2.1 Overall architecture

Our goal with the architecture was to create a renderer which was as maintainable as possible, such that new features could be added on top without creating problems with existing features. By creating UML-diagrams, applying basic object oriented design patterns we were able to create a simple and powerful architecture.

Unfortunately, since our scope for the solution was just a static rendition, performance was not a priority. Therefore, many architectural choices were not made with performance in mind. An example of a poor architecture in regards to performance, is the lack of a global convention on normalization of vectors. Since we do not know if a vector is already normalized, the system normalizes vectors whenever it needs them, which is wasted processing if the vector is already normalized. Furthermore, none of the proposed optimization-techniques such as kd-trees or interpolation were implemented.

2.2 Objects

We have implemented the three most basic mathematical objects; spheres, planes, and triangles. Furthermore triangles can be joined together in a mesh.
Triangles also stand out mathematically in that they are basically a clipped plane, and you have to take additional steps to make sure you don't intersect "outside" the triangle. The sphere and plane however are straightforward, and you just have to worry about special cases like rays not intersecting a plane and intersections with a sphere can result in a single or two points.

Meshes include a slight optimization in the form of a bounding sphere, which is checked for intersection before checking any of the potentially many triangles in the mesh. The bounding sphere is calculated as the center position of a minimal bounding box around the mesh and the minimum radius required to enclose all vertices.

We wanted to import models from .obj files, but didn't have time due to pressure from other courses. The ray tracer itself fully supports meshes, and could be improved much further by smoothing the triangles based on normals, or maybe doing some tessellation on them.

2.3 Shaders

Since different shaders are similar and requires mostly the same arguments to compute, an inheritance structure was established, in which shaders of different types derive from the same shader-base class. The shader base allowed us to have a cleaner code style and allows for an object to have an arbitrary amount of shaders on it. That way we could implement an ambient-, a diffuse- and a specular shader, and blend the resulting colors additively together. This allows for a large amount of combinations and implementing a new shader in the system would require very few if any changes to the existing code.

We would have like to added textures and maps to the shaders to allow distortions and more realistic surfaces. These would improve the look of all the other features we have added, but due to time constraints this proved itself impossible, and other tasks had higher priorities.

2.4 Lights

As all lights can be defined by a color, intensity and direction, we used an inheritance structure where lights of different types derive from a light-base class. This light base provides the methods needed to get the relevant information about the light. Then different implementations such as directional- and positional lights could be implemented. For a directional light, the method `GetDirectionOnPoint(Point3d point)`, returns the direction vector which the light is constructed with, but with a positional light, the method returns a new vector from the position of the light to the point given in the parameter.

To allow lights to have a color, during the shading the color of the light and the color of the shader is multiplied together, and the result is then divided by 255. By doing this a red light which shines on a white object will only make the object appear red.

Since positional lights looses intensity the further away from the light source you go, the method `GetIntensityOnPoint(Point3d point)` was added to the light base. Here, directional lights returns the intensity value they were constructed with and the positional light returns a value which is divided by the distance times a fall-off value.

In future versions of our solution we could implement spotlights, area lights, or we could have added a range to the light sources.

2.5 Shadows

In ray tracing, shadows are generated by taking the intersection point and checking from collisions in the direction of the light.
Our solution provides a method `GetLightsThatHitPoint(Point3d hit)`, which returns the list of lights which are not intersected by objects in the scene. By iterating over each light source in the scene and then creating `Line3d objects`, called shadow rays, from we could iterate over the scene objects and check for intersections.This creates a solution which runs in $O(N \cdot L)$ for each intersection, where N is the amount of objects in the scene and L is the amount of light

sources.

In real life shadows are often soft and not hard as the initial solution provides, therefore to improve on this method we added a function called `GetLightsThatHitPointSoftShadows(Point3d hit)`. To create soft shadows each light source generates a number of twisted shadow rays. These shadow rays' direction are all distorted randomly on the x-,y- and z-axis by a small amount. Then for each of these twisted shadow rays which are intersected, the intensity of the lightsource decreases by 1/(the number of twisted shadow rays). This solution runs in $O(N \cdot L \cdot T)$ for each intersection where N is the amount of objects in the scene, L is the amount of light sources and T is the amount of shadow rays.

The implementation of the hard shadows is both efficient and correct, and we believe it to be the best fast way to create hard shadows using ray tracing. The soft shadows on the other hand could use a lot of improvements. First of all since it is based on randomized numbers, its accuracy is low and only when high numbers of shadow rays are created the noise gets diminished. But again, while soft-shadows provides a more realistic-looking rendering, it comes at a price of both processing-time and memory-consumption, since a lot of objects need to be created for each intersection.

2.6 Reflection

Reflectiveness is defined by the material of an object. Our implementation of the algorithm starts when a ray hits a reflective object. The ray is reflected of the surface according to the law of reflection[1] stating that the angle of incident ray (relative to the normal of the surface) is equal to the angle of reflection, and continues in its new direction like usual. The resulting color of this new ray and all subsequent rays are blended together to make the final color, based on the material's reflectivity.

2.7 Refraction

Our refraction-implementation is based on Snell's law[2] and the Fresnel equations[3]. We take into account the possibility of total internal reflection. To get us started, scene-objects has a refraction-index, a reflectiveness and a transparency. The whole scene has a refraction-index, too. When a ray intersects the boundary of an object, we determine whether there's a contribution from refraction and reflection, and then, if relevant, we determine each of these contributions separately and finally blend these contributions together. Notice, that determining either of these contributions usually implies several additional recursive calls to the same algorithm.

Our solution has some flaws, though. We wanted to provide room for an object being shaded in its own color, as opposed to shade (a point on) it only using what reflects and refracts from that point; this approach doesn't would not display the material itself. The "correct" solution is to (aside the aforementioned reflection and refraction) also take into account, what colours the material (of the object) will absorb, and shade it as a mix of these three contributions. Our solution forces a shading-contribution onto a material, and we then decide in the following in what amount we should blend in the refraction- and reflection-contributions.

We had a hard time evaluating if our solution was correct. This is due to two facts. Raytracers are hard to debug, because you do not know which pixel or where in the scene you are currently debugging, while going over the millions of pixels that is to be generated. Secondly, we had a hard time imagining what a 4 meter radius sphere would actually look like in real life

[1] `http://www.physicsclassroom.com/class/refln/Lesson-1/The-Law-of-Reflection`
[2] `http://www.flipcode.com/archives/reflection_transmission.pdf`
[3] `http://en.wikipedia.org/wiki/Fresnel_equations`

if it was made out of glass. Especially, around total internal reflection we did not know what it would / should look like in real life.

3 Retrospect - what could have been improved

This section reflects upon the delivered project.

3.1 Features we wanted and planned for

We wanted to offload the processing across the CPU and the GPU, but failed to. As we're novices in C++ (and had our issues just there), through the tutorials and material found online about the GPU utilization in C++, we realised that it would present a very steep learning-curve for us, taking into account we weren't too familiar with C++ either. Hence, we decided to skip this feature in favor of putting effort into other areas of the program.

Furthermore, we did not get the time to implement textures, although we believe that our architecture would easily allow for this.

We also had planned to allow the user to render an .obj file, and we have a half-finished solution. Unfortunately, there are some hiccups and we cannot say that it is done and implemented well enough to handin as a feature. We do believe that given one more week this feature could have been completed.

3.2 Features that would be cool to implement in the future

As it has proved notoriously difficult to debug the program, we believe we could have benefitted greatly from unit-testing individual methods. We have spent a lot of time tracking down where or whether the logic was wrong. Having known that individual methods worked on their own, we could instead have focused our debugging efforts on the calling of and or the interaction between the methods.

Moreover, there's a lot of space for future improvements. An obvious one would be to store the objects of the scene in a kd-tree, ordered by spatial location within the scene. Currently, when we're checking for ray-intersection, we iterate over all objects within the scene. A kd-tree would allow for only checking for a relevant subsection of the scene-objects.

If we had managed to get textures on objects, then our system would have been greatly improved by other kinds of maps, such as bump maps, specular maps, offset maps. These maps are mostly used for details and perform very well, if you don't want to render huge and very detailed meshes. Specular maps and other maps that transform colors would be essential for more complex models, that are made of different materials. All would be easy to implement if textures were implemented.

In terms of ideal code-practice in C++, there's a huge void surrounding memory-management, as we do not consequently release memory when we should.

4 Conclusion

Ray tracing is a powerful rendering technique, that shines in a fast, object-oriented language like C++. We successfully created a system, which is easy to extend and maintain, and the

system produces good looking renderings with a decent amount of features. We successfully implemented, various ray-object intersection, shaders, lights, shadows, reflections and refractions, while features such as .obj import, meshes, textures and GPU-parallelization unfortunately went short. Some of the features such as soft shadow is poorly optimised and could be improved with a better algorithm, and other optimization techniques, such as Kd-trees, could also be implemented to improve the processing time.

Overall we are satisfied with the result and believe that given more time we would easily be able to implement the last features or optimizations, while GPU-parallelization might be a bit further down the road.

A Work distribution

On the following page, the work-distribution among the group's members presented

Work Distribution

Week 1

Task	Member
Line-sphere intersection method	Morten
Vector methods implemented	Morten
Setup scene	Rasmus & Wind & Morten
Display the result in WPF (Windows Presentation Framework)	Rasmus
Basic Algorithm	Rasmus & Wind & Morten

Week 2

Task	Member
Reflection-implementation	Morten & Wind
Color blend methods	Morten & Wind
Shaders	Morten & Wind
Plane intersection	Rasmus
Triangle intersection	Rasmus
Started on meshes-implementation	Rasmus
Lights	Wind
Hard Shadows	Wind

Week 3

Task	Member
Meshes	Rasmus
Core Algorithm improvements	Wind & Morten
Refractions-implementation	Wind & Morten
Area lights (later discarded)	Wind & Morten

Week 4

Task	Member
Refactoring	Morten & Wind
OBJ import (incomplete)	Rasmus
Soft Shadows	Wind
Fresnel equations for transparent objects	Wind & Morten

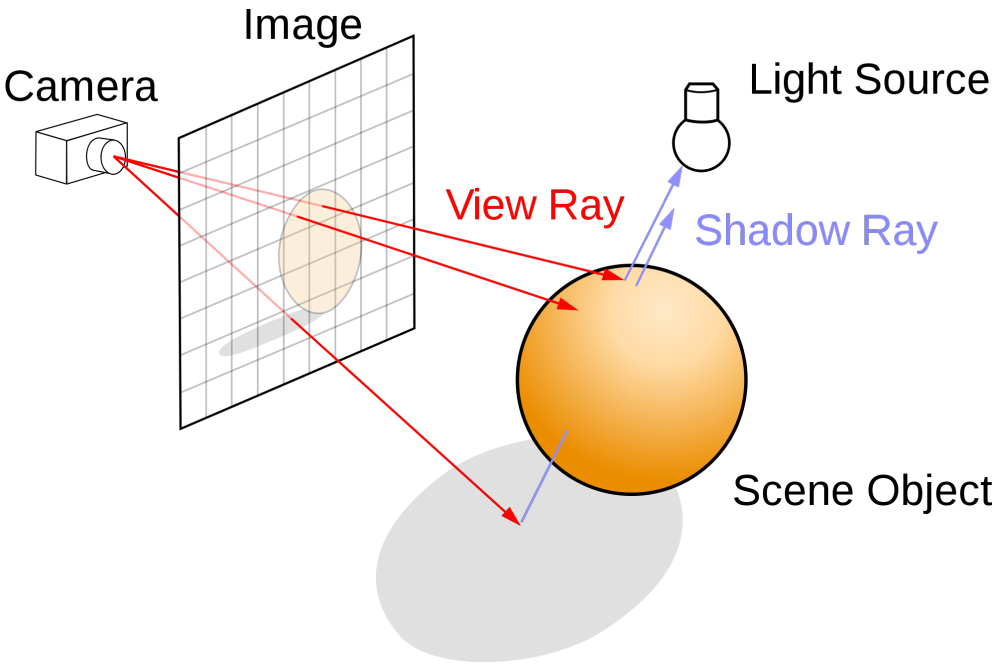
B Presentation slide-deck

On the next page, the presentation slide-deck as given on May 8th is attached. It provides a variety of different renderings produced by the delivered software.

Ray Tracing

Rasmus Dyhr Larsen
Morten Albertsen
Anders Wind Steffensen

Ray tracing - explained



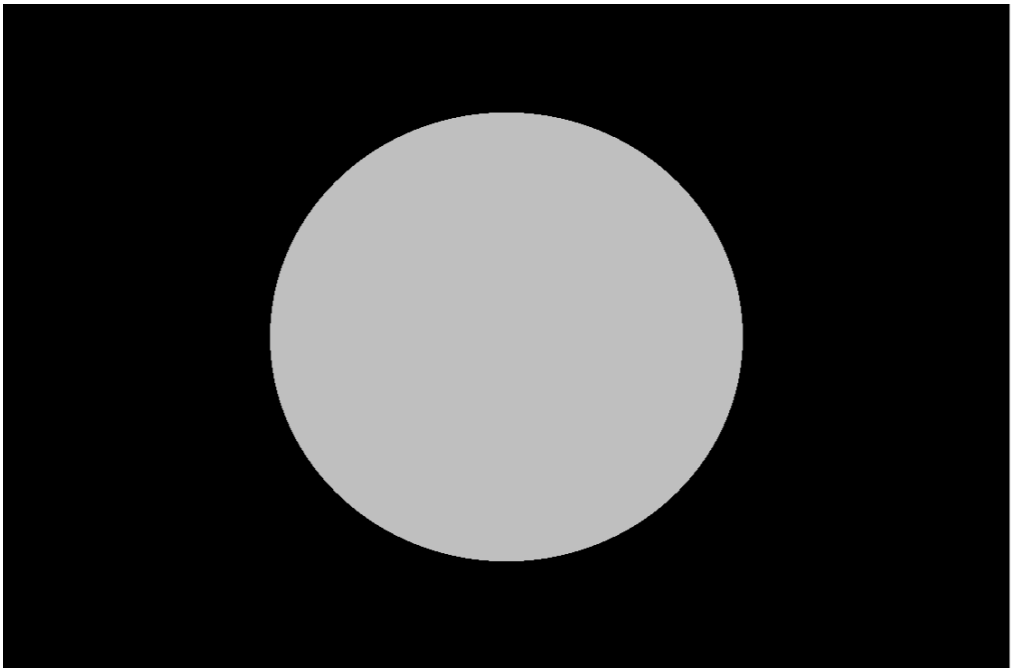
Initial Objectives

- Static rendition
- Simple polygon model
- Hands-on with OpenGL or work with GPU
- Understanding the concepts of ray-tracing
- Gain C++ experience

Features

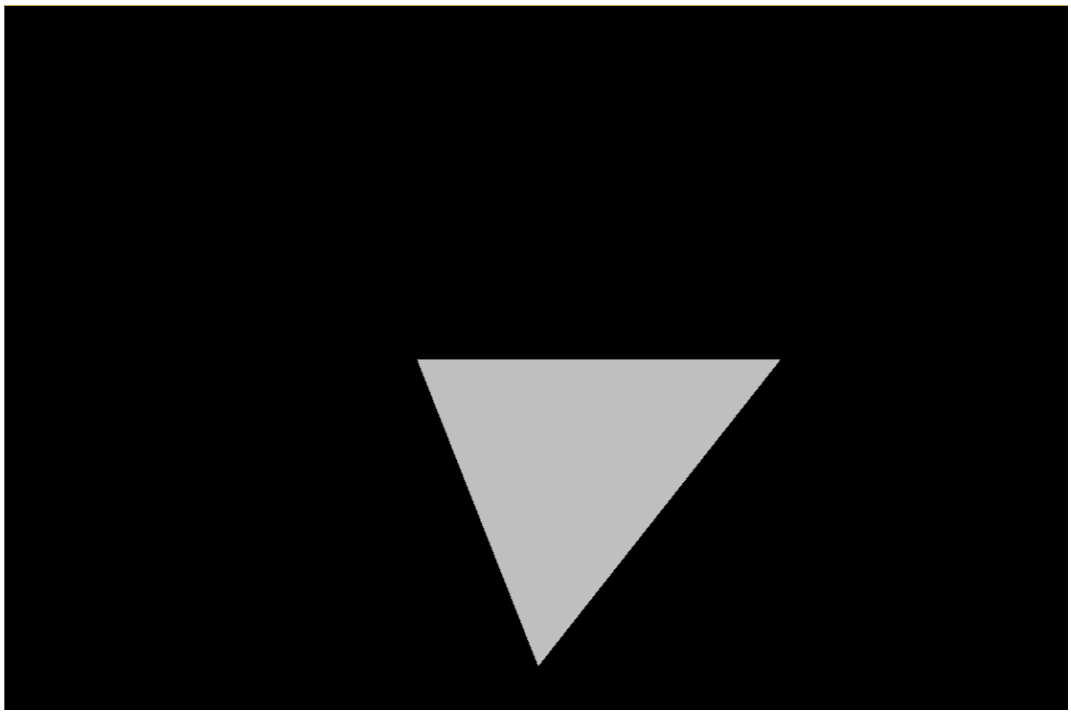
Ray-Object Intersection

Ray-Object Intersection - Sphere



Ray-Object Intersection - Plane

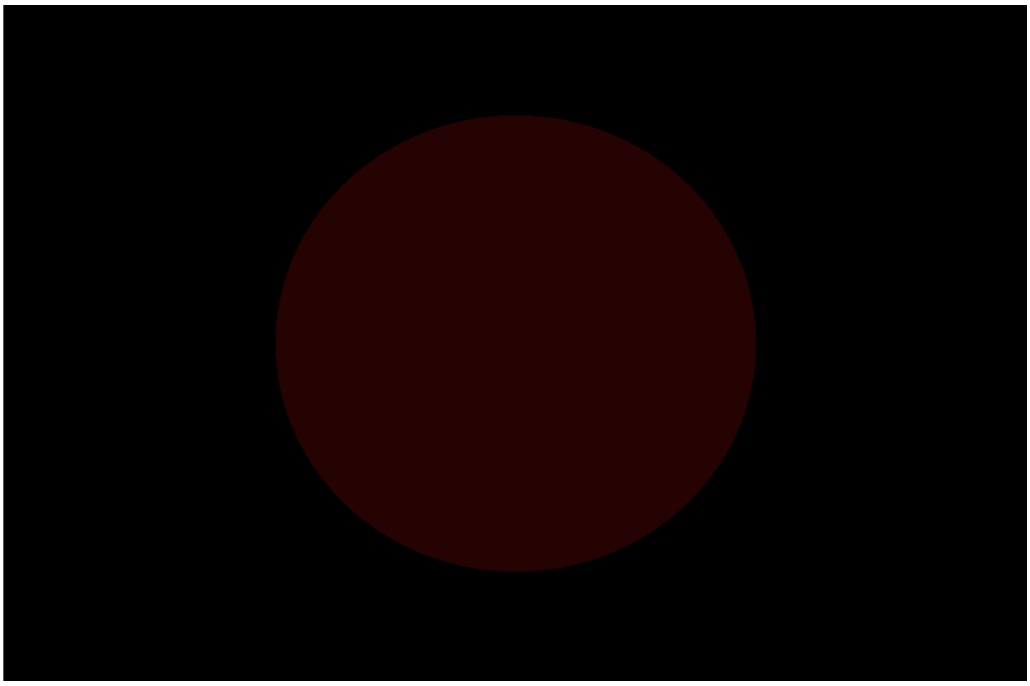
Ray-Object Intersection - Triangle



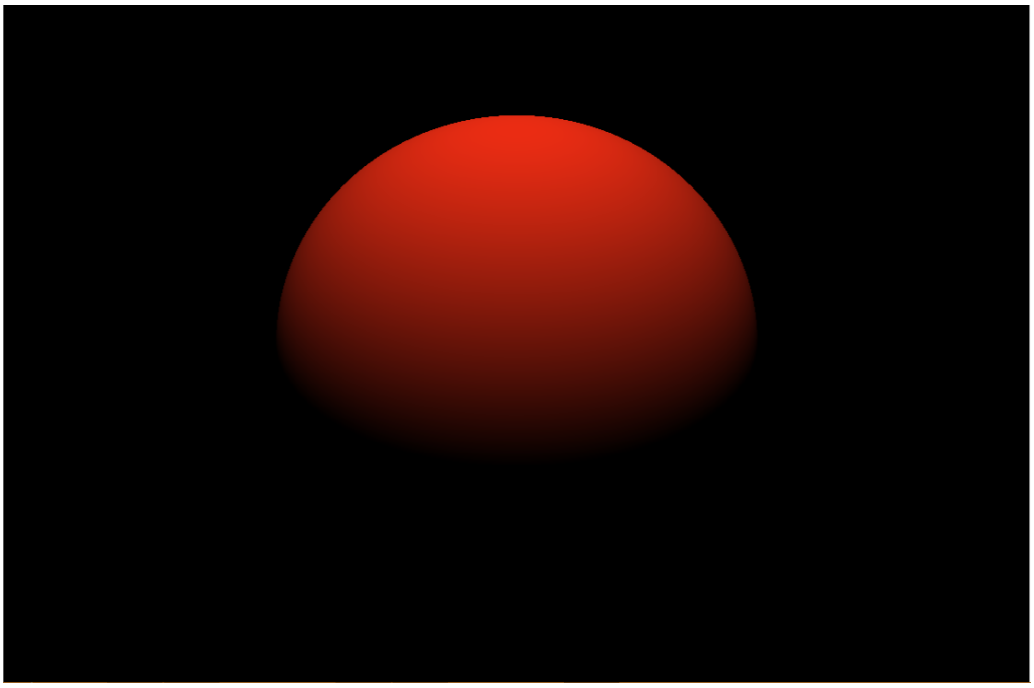
Features

Shaders

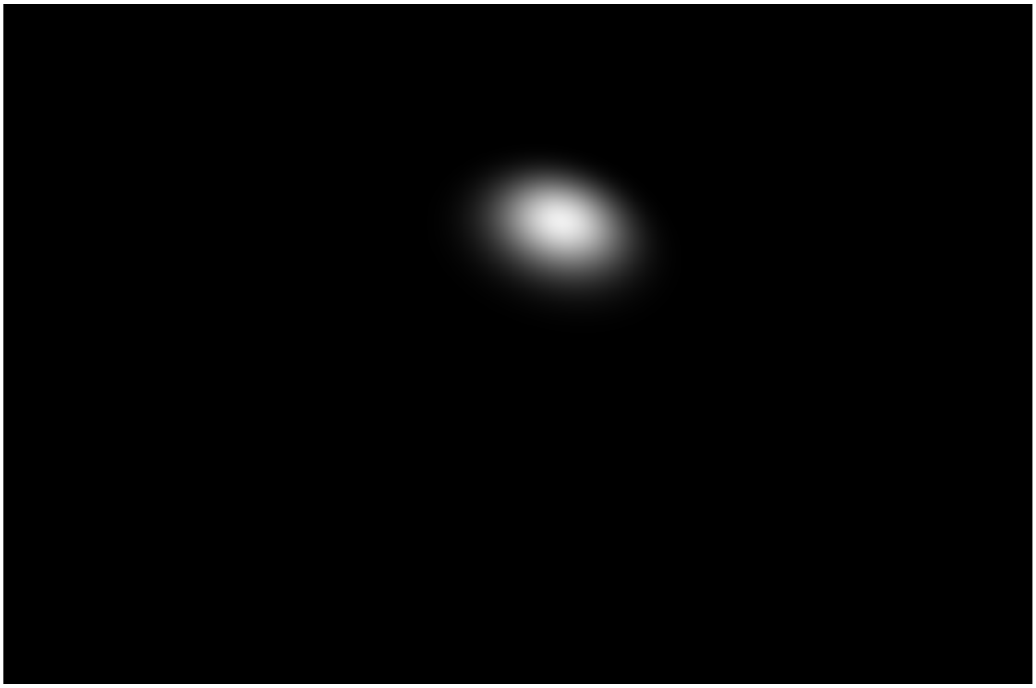
Shaders: Ambient



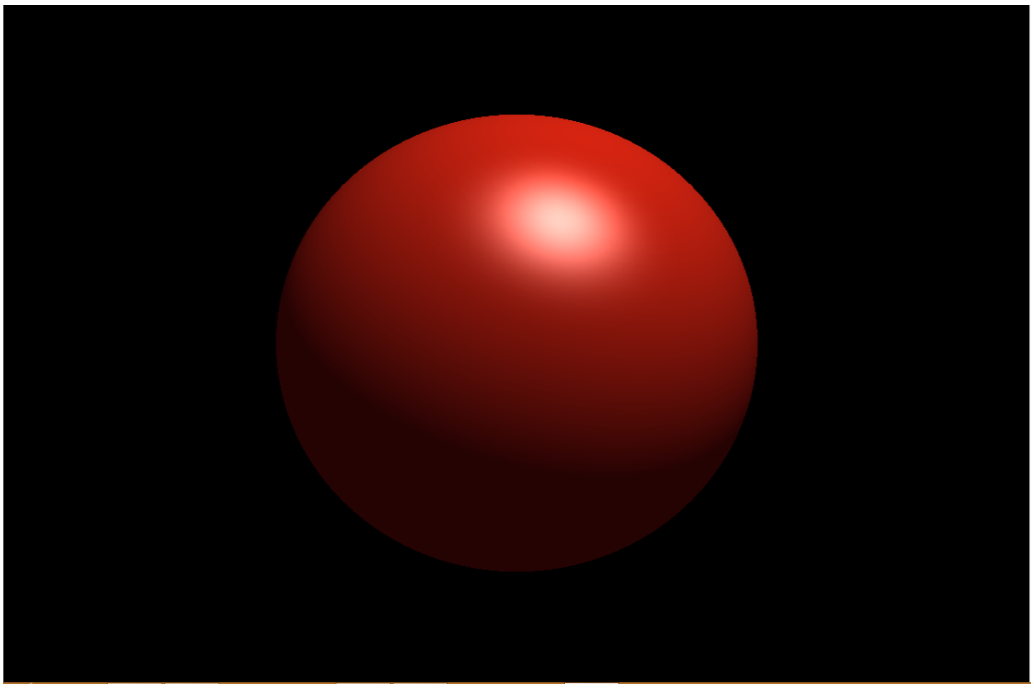
Shaders: Diffuse



Shaders: Specular



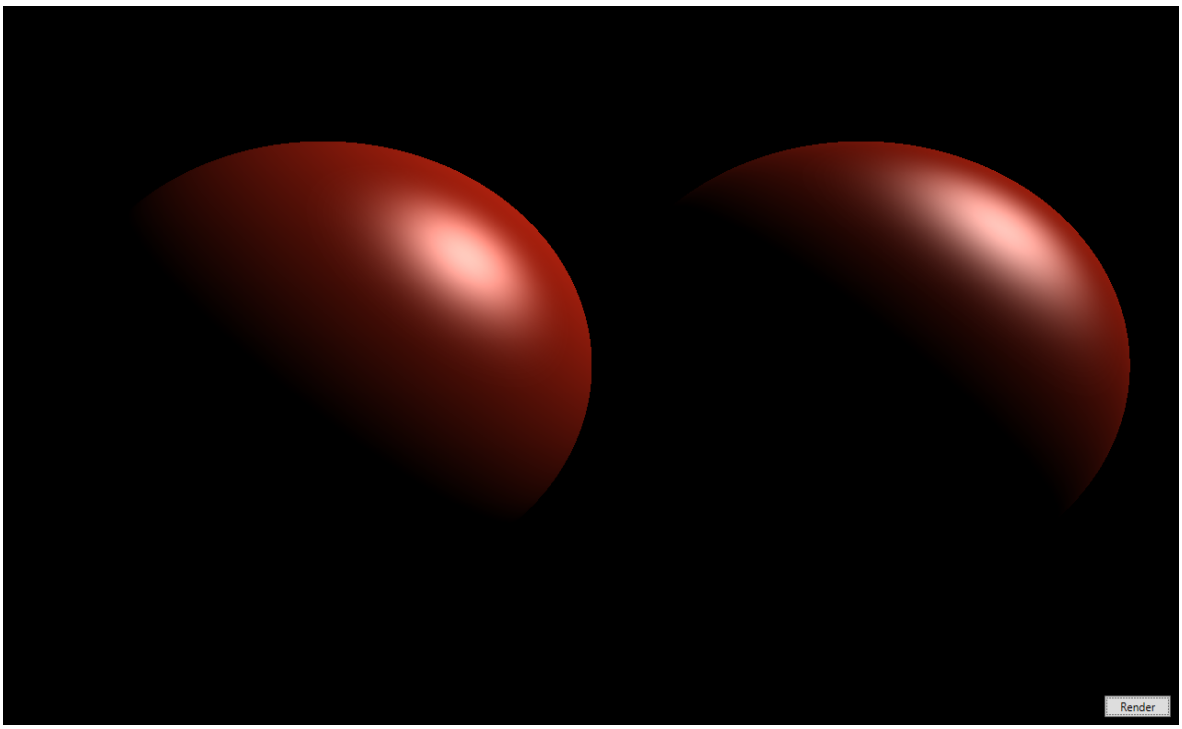
Shaders: Ambient & Diffuse & Specular



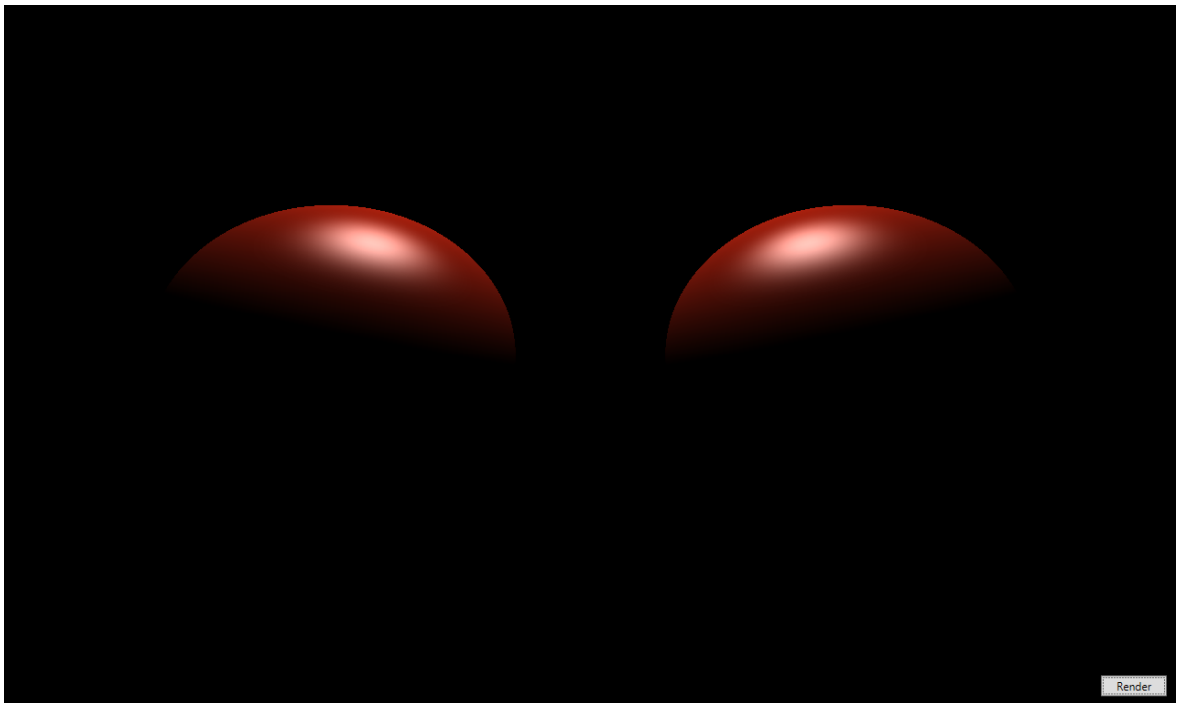
Features

Lights

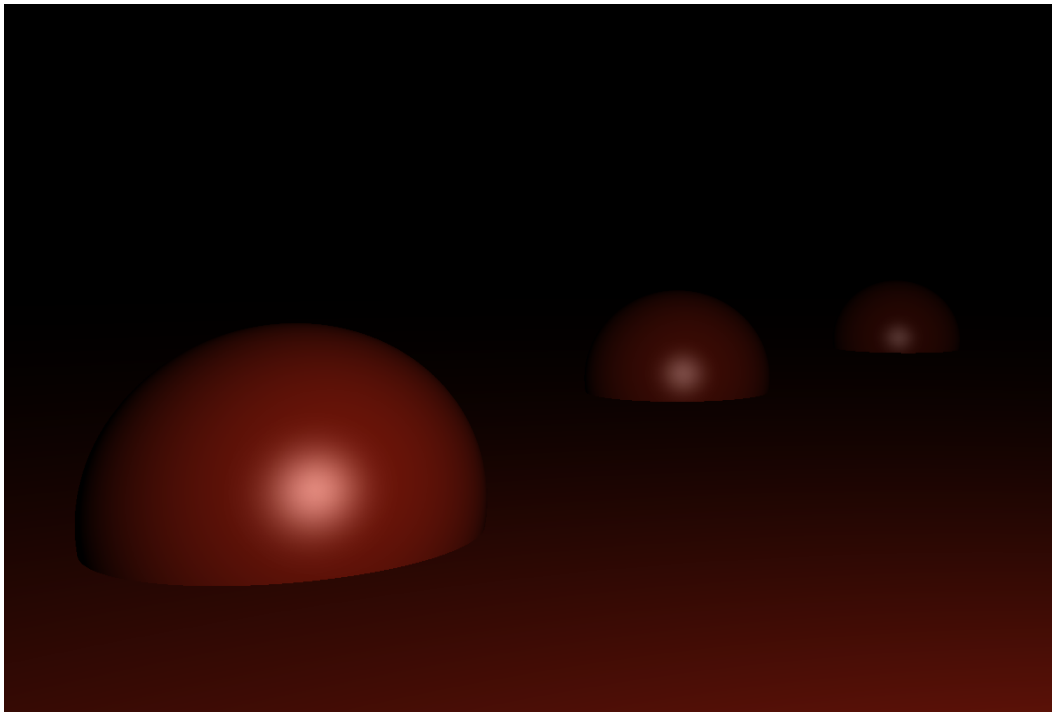
Lights : Directional



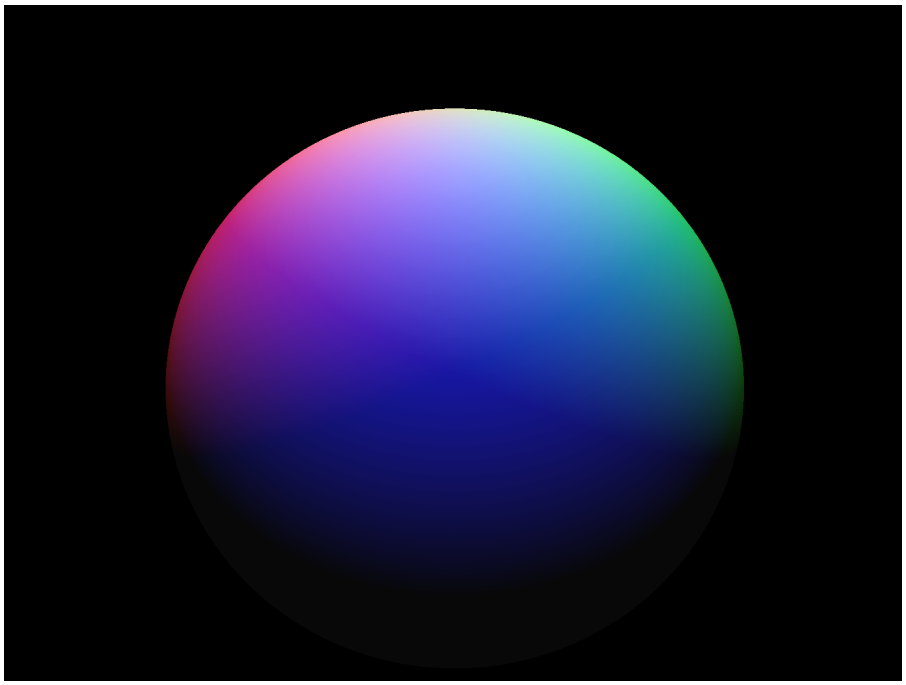
Lights : Point



Lights : FallOff



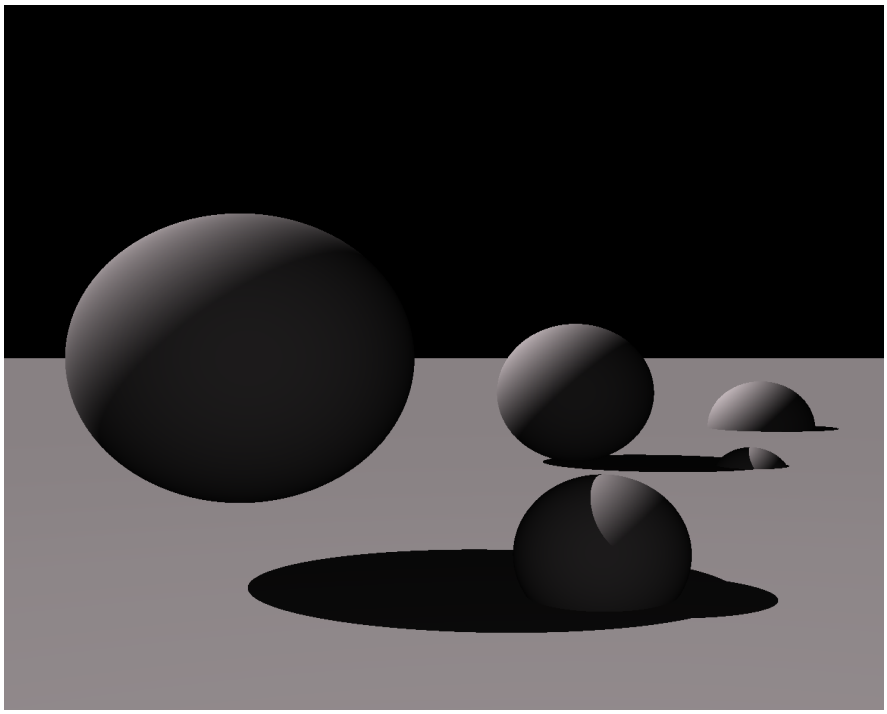
Lights : Colored lights



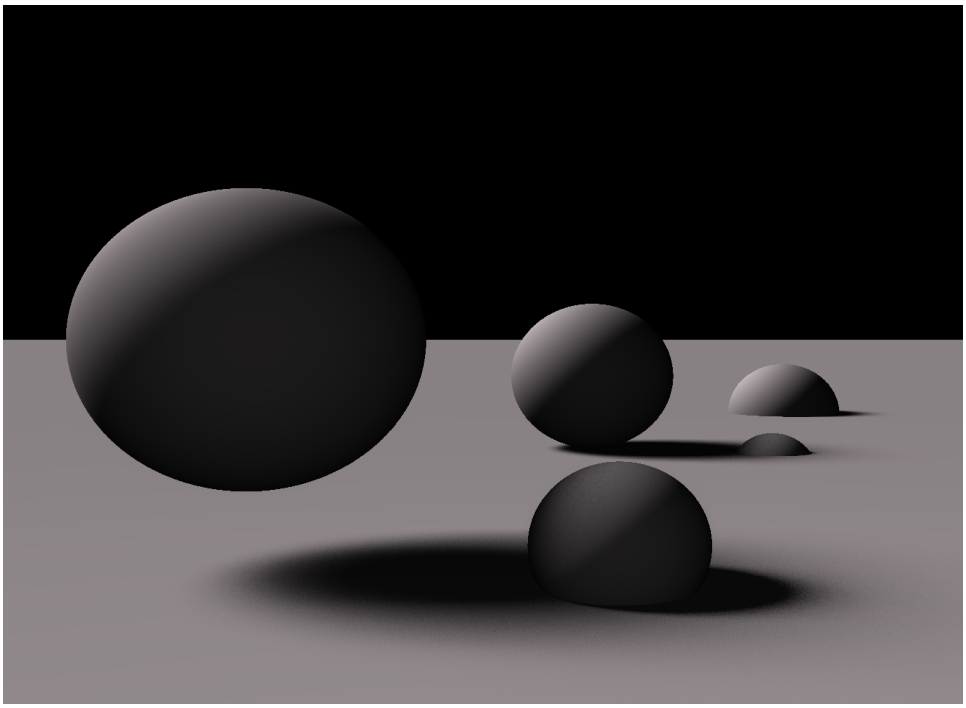
Features

Shadows

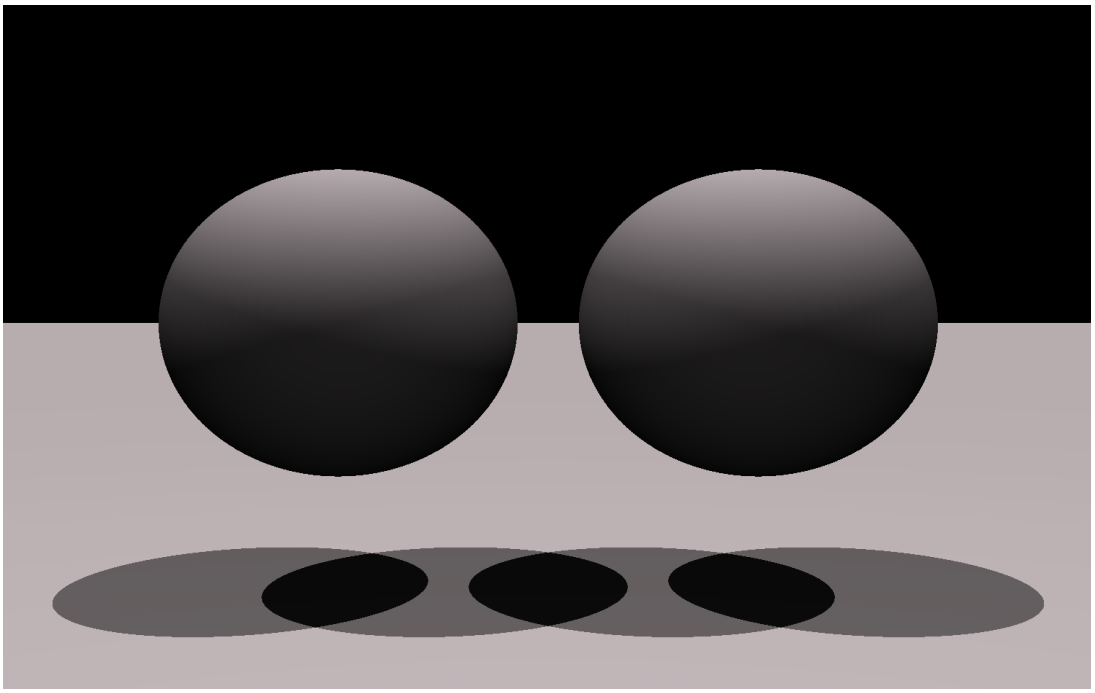
Shadows : Hard shadows



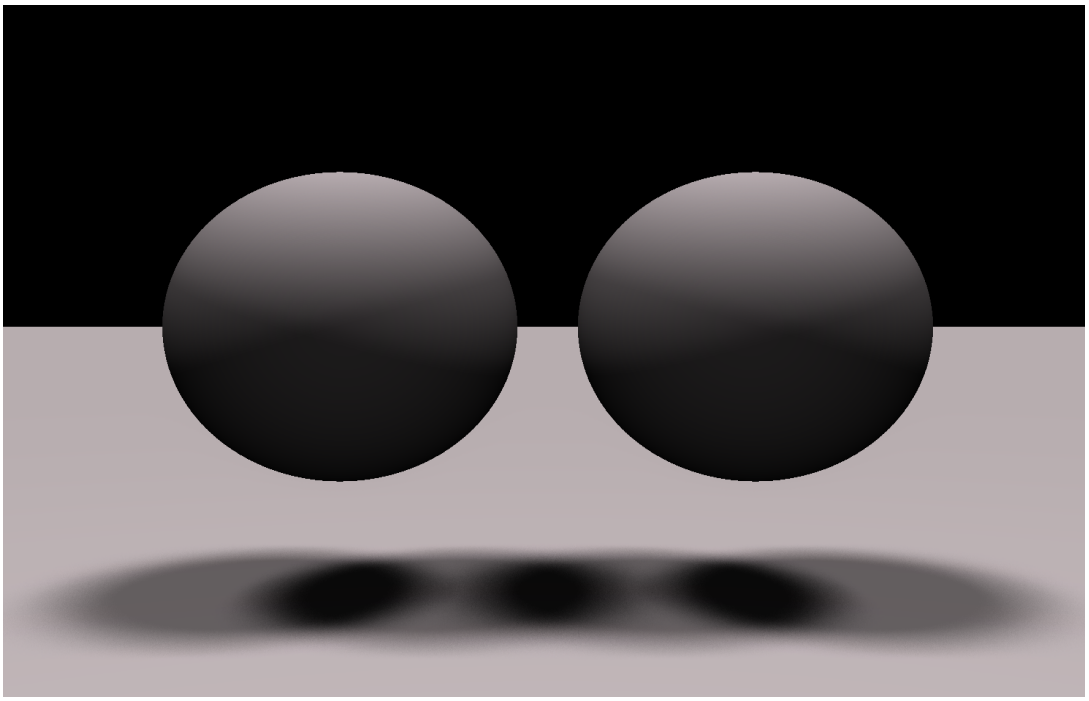
Shadows: Soft shadows



Shadows: Hard shadows blended



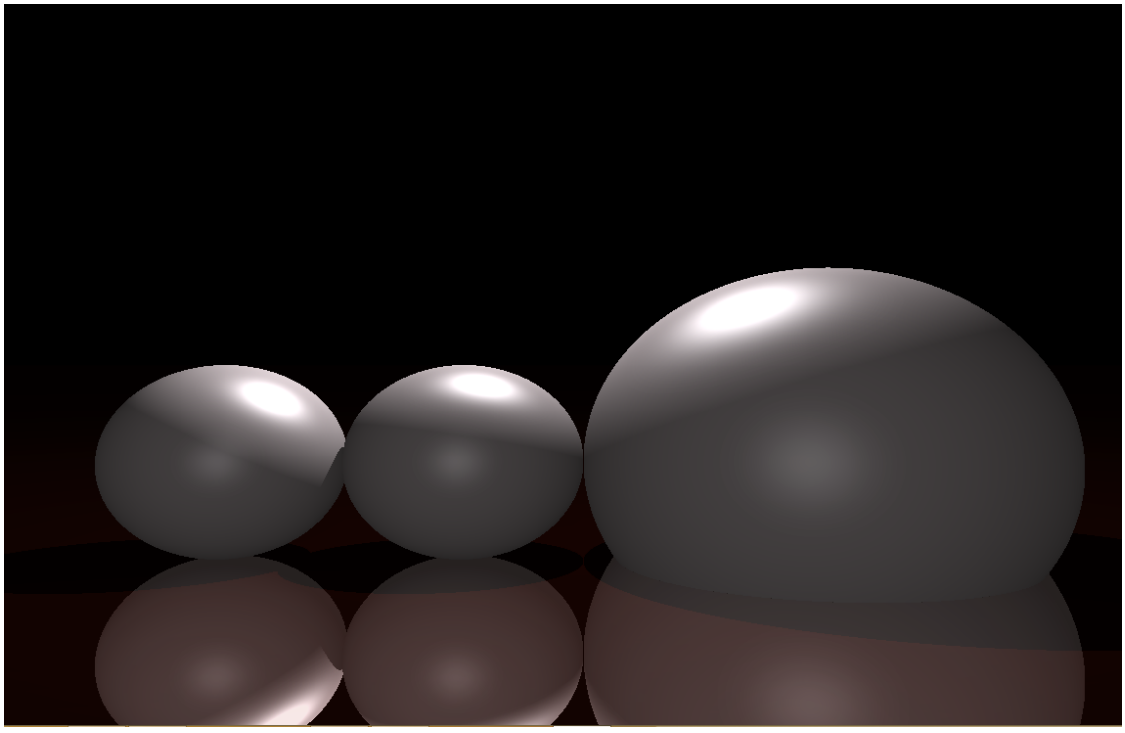
Shadows: Soft shadows blended



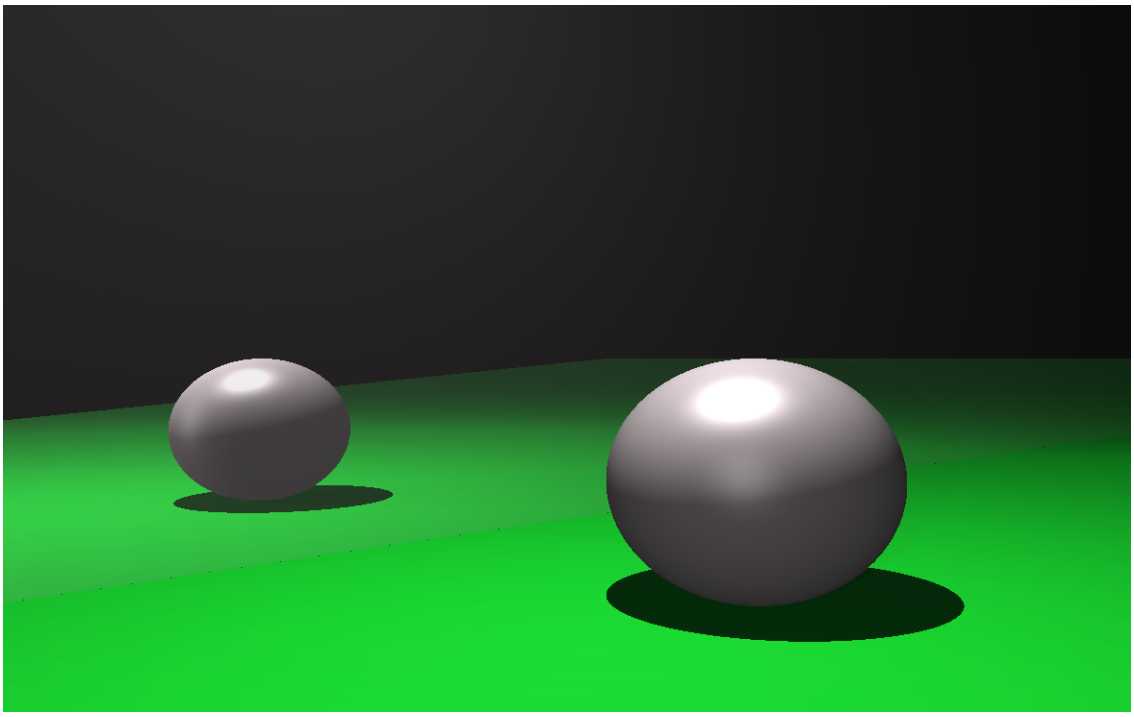
Features

Reflections

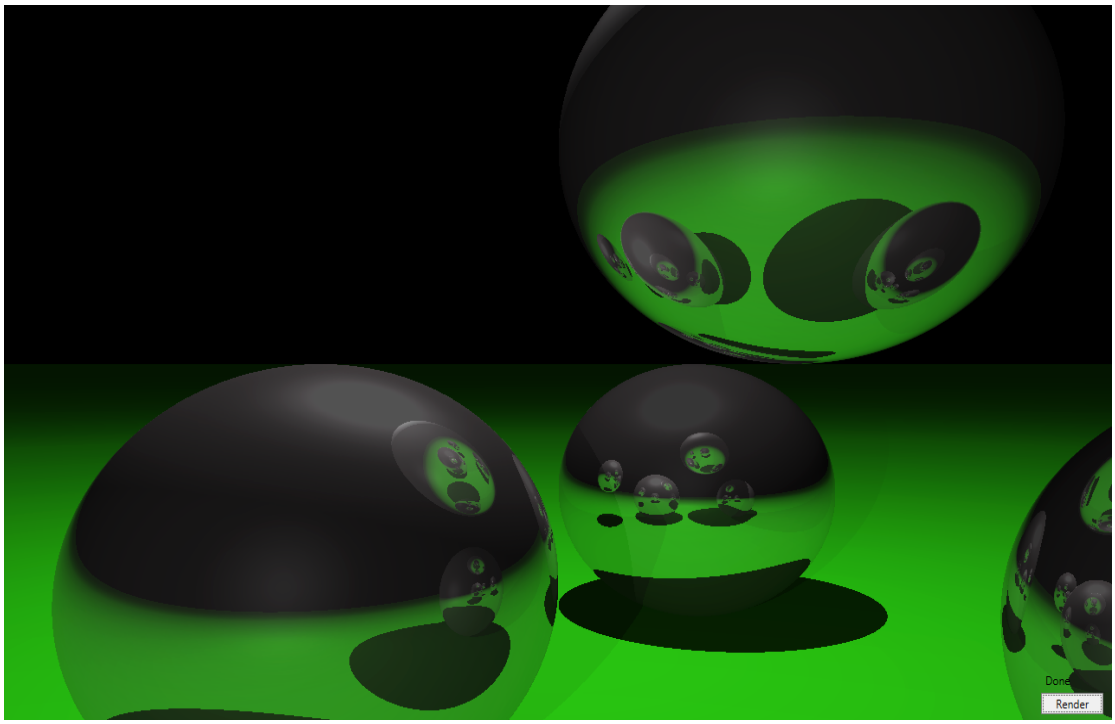
Reflection: reflecting plane



Reflection: Back of object of sphere is seen



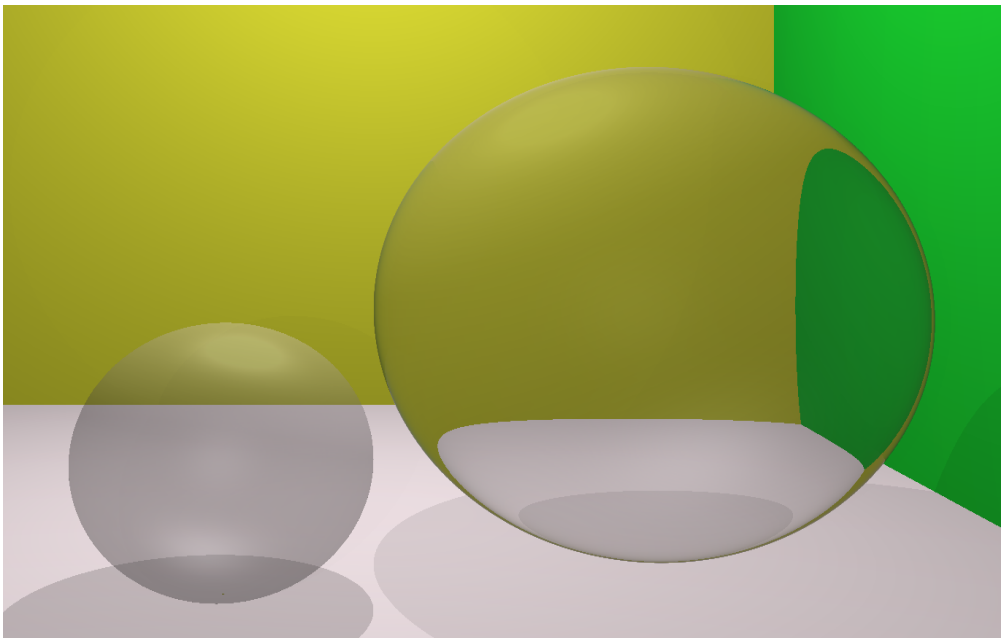
Reflections:



Features

Refraction

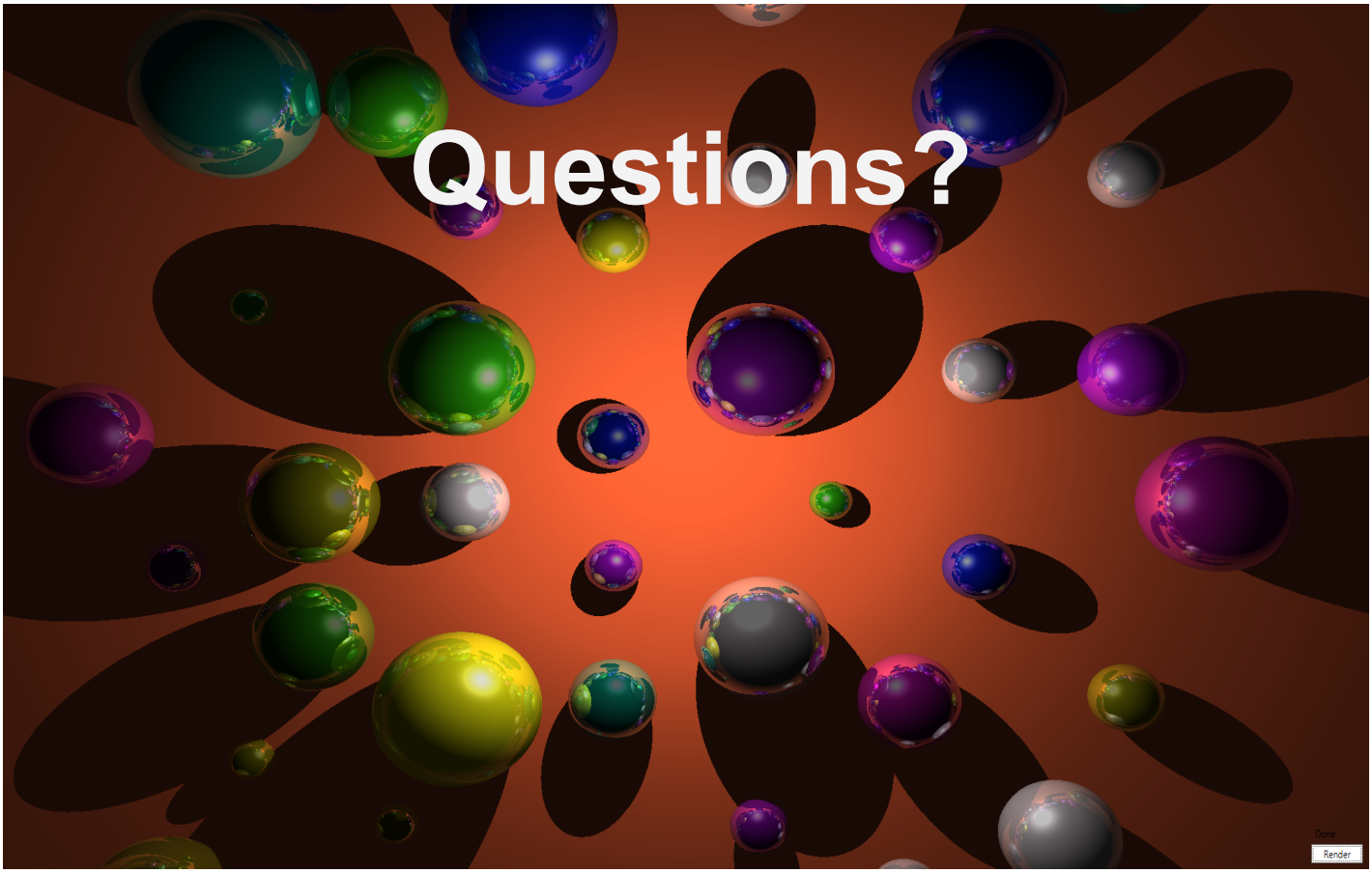
Refraction: refraction index and shadows




Final Rendering
Done
Render

What did we miss

- GPU rendering
- Importing .obj files - almost there
- Textures and “shader-maps”
- Optimization models: Kd-trees, better memory management, interpolation
- Unit-testing - could have been awesome!


Questions?
Done
Render