Scientific Visualization, 2021, volume 13, number 4, pages 111 - 126, DOI: 10.26583/sv.13.4.09

Spherical Lenses for Virtual Optic Experiments

V.A. Debelov[1,A], N.Yu. Dolgov[2,B]

[A] Institute of Computational Mathematics and Mathematical Geophysics SB RAS
[B] Novosibirsk State University

[1] ORCID: 0000-0002-7577-4700, debelov@oapmg.sscc.ru
[2] ORCID: 0000-0001-5622-3586, nikitabrn1999@gmail.com

Abstract

While the mathematical modeling of optical phenomena, a computer calculation is often performed, confirming the conclusions made. To do this, a virtual computer model of the optical installation is created in the form of a 3D scene. Also, virtual scenes are often used in training when creating presentations.

This paper describes computer models of spherical lenses and the calculation of interaction of linear polarized light rays with them. It is focused on applications that use ray tracing. It is known that light of any polarization can be represented on the basis of the mentioned one. The reflected and all rays passing through the lens that arise due to internal reflections are calculated from the ray incident on the scene object. The number of internal reflections is set by the parameter. All output rays are calculated based on the application of Fresnel's equations and are characterized by intensity values and polarization parameters.

We selected spherical lenses since they are most often used in optic installations. They are constructed on the basis of the application of the set-theoretic intersection of geometric primitives: a half-space, a sphere, a cone, a cylinder and their complements to the scene space. An advanced user can build their own objects by analogy, for example, cylindrical lenses.

Keywords: optical experiment, virtual scene, spherical lenses, optically isotropic objects, linear polarized light.

1. Introduction

In the mathematical modeling of natural phenomena, computer experiments are often performed with the similar computer model to confirm its reliability. The construction and analysis of a mathematical model often leads to the creation of a 3D scene, which is a virtual analogue of a real installation. Another area of creating virtual 3D scenes is education, especially when it is necessary to demonstrate a phenomenon physically correctly. In optics, when obtaining images, lenses are used in one way or another, for example, a lens, an eyepiece, an eye.

The idea to develop a library of lenses arose after in experiments on interference simulation, we often had to change the shape and parameters, and even the type of lens. We came to the conclusion that such objects should be isolated so that the corresponding software modules are responsible for their interaction with linear polarized light rays. Why linear polarized light?

1. Historical background. In connection with the development of algorithms for photorealistic visualization of scenes involving optically anisotropic transparent crystals, only polarized, moreover, linear polarized rays were considered, since a light ray with any polarization incident on these crystals generates up to four linear polarized rays (up to two reflected and two refracted) at the boundary [1].

2. This does not limit the freedom of choice. Light of any polarization (unpolarized, partially polarized, polarized) can be represented as a combination of a certain number of linear polarized rays with a lower intensity, see [1].

3. Please note one very important property for ray tracing: when a ray falls on an optically isotropic object, regardless of the state of polarization, the trajectory of the ray is the same. The polarization of the incident ray affects the state of polarization and the intensity of the reflected and refracted rays obtained when interacting with the surface. This is especially evident when performing calculations using Fresnel's formulas [1, 2]. This property does not hold for anisotropic objects.

In [2], the prerequisites for the development of computer models of lenses, new features compared to existing software were identified. Thus, this article is a natural continuation of the work [2]. First of all, we paid attention to spherical lenses, as the most common ones. Indeed, in the vast majority of cases, in order to visualize the required optical effects, it is necessary to introduce a lens (camera lens, etc.) into a virtual scene, as well as into a real experience stage.

The main purpose of the development is to provide a researcher or an application programmer with the tools to create their own standalone application for modeling optical phenomena in a generally accepted language (for example, C++). Various technical issues can be solved by using suitable libraries in the same language.

In the second section, similar works are considered in order to more clearly show the niche for the created library of spherical lenses. The third section is devoted to the description of the contents of the created lens library and the proposed principles of lens design. The fourth section is devoted to the conclusions.

2. Previous work

Modern renderers do not work with polarized light, so you need to look for the necessary tools in another field related to optics. Indeed, many means of developing optical devices or optical design systems have been developing for a very long time and are quite representative. For example, widely known systems: ASAP [3], TracePro [4], Code-V [5], FRED [6], Zemax [7]. You can continue, because there are at least a dozen of them. We looked at such systems from a slightly different angle in [2]. These programs allow you to perform geometric modeling of various geometric shapes and, accordingly, lenses. Various physical characteristics that are inherent in real objects can be assigned to objects. Including the optical properties of real objects. Rich sets of lenses with various geometric shapes have been created. Moreover, developers of various software products are familiar with the developments of competitors or colleagues. Consider for example the optical design system OptTaliX [8]. Among other operations, it allows its users to perform the following actions:

- Import lenses and related data from Code-V, Zemax and a number of others.
- Export lens data to Zemax, ASAP and others.

There are references to FRED, TracePro and a number of others in the manual [9]. Finally, the OpTaliX user manual (pages 476 and 477 [9]) suggests using models of real lenses from lens catalogues of various manufacturers and distributors. It can be concluded that most optical design systems are connected to a certain extent, by the data, models of optical objects.

It should be noted that within the framework of the mentioned programs, it is possible to perform a physically correct calculation of the passage of light in the constructed optical installations. Calculations are made both in the ray and in the wave formulation. This means that within the design of these systems, the functions of physically correct behavior of light in various situations, i.e. the functional core, have been implemented and debugged for a long time. But it turns out that there is no open set of libraries, there is no open SDK from which a programmer can take and use the desired function. On the contrary, a dialog

interface is being built on the core and is being expanded with each new version of the software product.

The following simple thought comes to mind. Since so many *commercial* optical design systems coexist peacefully, it means that another one can be made. But to do this, you must first develop the core.

Undoubtedly, within the framework of these systems, much of what we needed to solve was done, namely, interaction with the boundary of optically isotropic transparent media of a ray of polarized light. It is not possible to adapt these systems for the following reasons. The systems are closed, very cumbersome and expensive, focused on the construction (geometric and optical modeling) and calculation of paths and rays of light in a complex optical system, to assess the parameters of the beam (ray) at a particular point in the scene. Please note that they are not focused on obtaining photorealistic images.

When developing renderers, special attention is paid to the possibility of their practical application, especially the calculation speed. It is appropriate to mention the opinion of an authoritative expert, which quite accurately defines the difference between the development of optical design systems and the development of a renderer. In the work [10] Glassner was quite accurate: "As almost always seems to be the case, writing a good shader seems to involve some judicious trading off of accuracy and realism with approximations and pragmatism. I mean, we could simulate all of this at the molecular or even atomic level, but it wouldn't show up in the results. The trick is to find a nice balance between simplicity, efficiency, and verisimilitude".

3. Library of spherical lenses

The lens library was built to unite computer models of lenses and make it easier for the user to design spherical lenses, add them to a 3D scene, and process the interaction of rays of linear polarized light with them. This section describes the requirements that the software and its implementation must meet. The book [11] describes in detail all types of thin spherical lenses. Since this is the most common type of lenses, we decided to provide users of the software package with a simple design of all six types of spherical lenses: biconvex, plano-convex, concave-convex, biconcave, plano-concave, convex-concave (see Fig. 1). Each lens can have a cylindrical or conical rim. The rim is the surface of the lens attachment; it can be made of both transparent (working surface) and opaque material. We decided not to limit ourselves to only thin lenses, but to allow the use of similar geometric shapes of any size. Moreover, since we perform physically correct ray tracing, there are no problems with accuracy as in the cases of using the formulas of thin [11] or thick lenses [12].

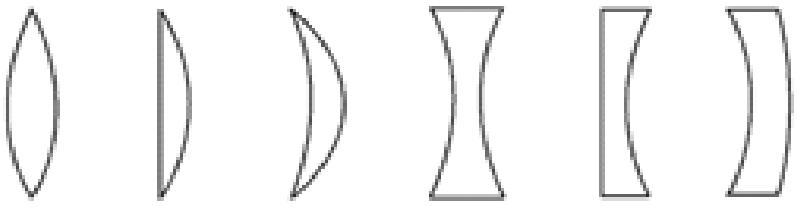

Fig. 1. Six types of spherical lenses: biconvex, plano-convex, concave-convex, biconcave, plano-concave, convex-concave. The image is taken from [11].

Our experience has shown that the set of these lenses significantly facilitates the creation of scenes and the execution of virtual optical experiments. Please note that even complex optical devices, for example, polarizing microscopes (see Fig. 2), are constructed mainly with the help of these lenses: each transparent object in the design of the microscope is a simple spherical lens or is composed of several spherical lenses [13].

Since it is necessary to take into account the polarization of light to calculate interference, the software package must be able to process the interaction of rays of linear polarized monochromatic light with lenses. When tracing a ray through a lens, according to [2], three possible events are taken into account:

- **Event 1**. Total internal reflection (TIR). In this case, the user can receive two calculated linear polarized rays, and further ray tracing stops.
- **Event 2**. In this case, the user receives the ray that hits the rim and the coordinates of the hit point, as well as further ray tracing stops.
- **Event 3**. The next output ray of linear polarized light has been successfully calculated, or the specified tracing depth inside the lens has been reached. Tracing depth is determined by a separate parameter. For example, if the user has set the tracing depth k, then $k + 1$ output rays should be calculated (see Fig. 3).

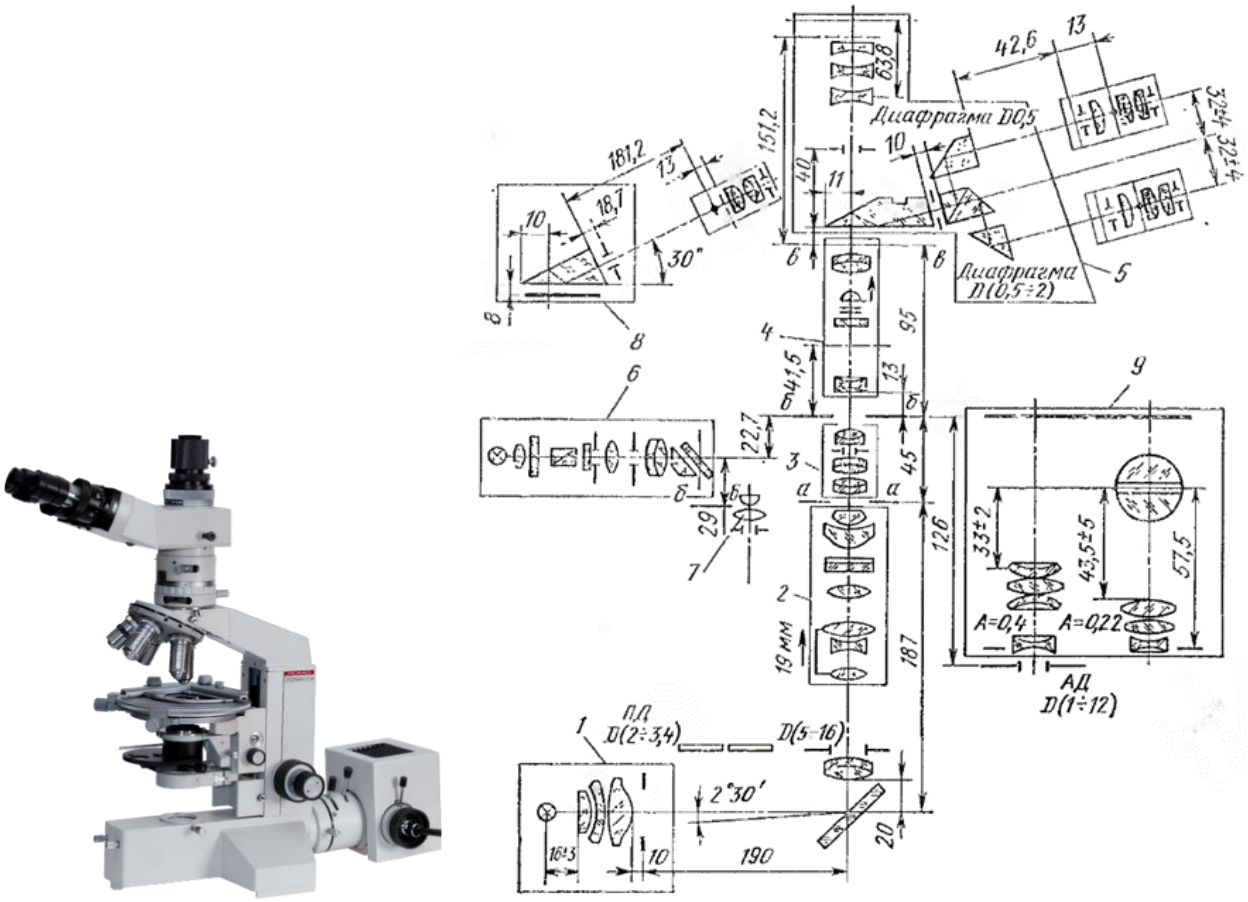

Fig. 2. On the left is a polarizing microscope (photo from [14]), on the right are the components of the device of a polarizing microscope (from the book [13]).

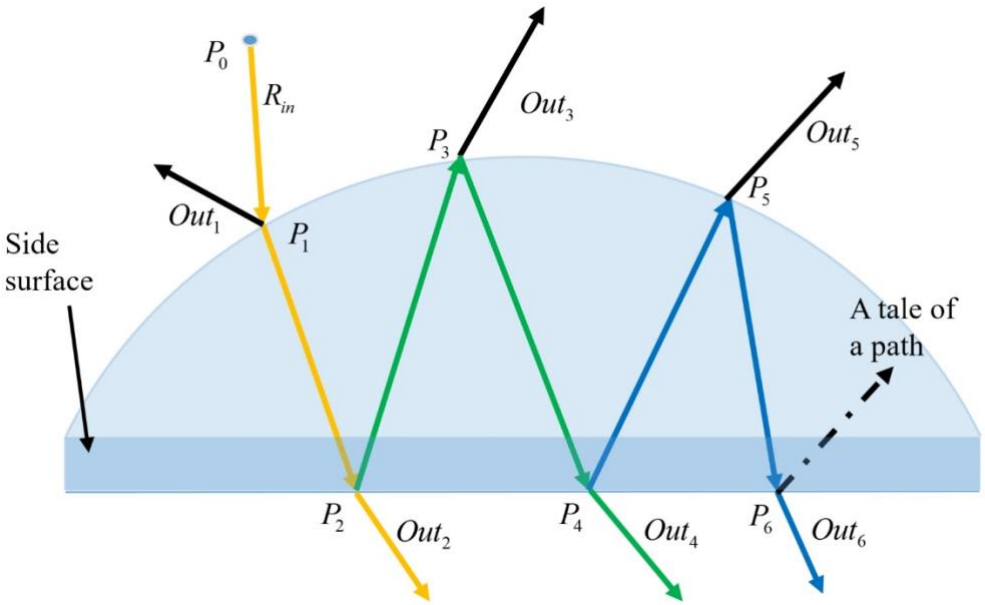

Fig. 3. Subtree of rays of depth 5 in the lens. Notation: R_{in} is incident ray, Out_* are rays coming from the lens, P_* are treetops (nodes), E_* are internal reflected rays (image from [2]).

To construct lenses, a set-theoretic intersection operation is used over several geometric primitives. The set of primitives includes a half-space, a part of the space inside or outside a sphere, a cylinder, a cone. We consider only objects with a simply connected boundary.

3.1 Half-space

The half-space of the scene can be set by specifying a plane in three-dimensional space using a point and a normal. The normal indicates the selected half of the scene space (Fig. 4). Here and further, the wire cube denotes a 3-dimensional space.

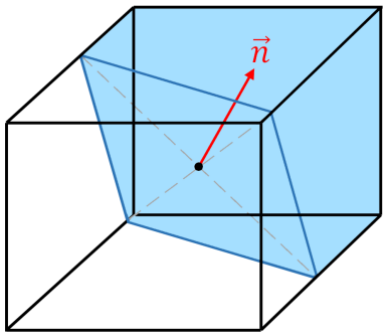

Fig. 4. The plane with the normal **n** separates the half-space.

3.2 Space inside or outside the sphere

The sphere is defined by the center $C(x_c, y_c, z_c)$, the radius R and the Boolean sign INSIDE: *true* means that the inner region is taken, if *false*, then the complement to the space is taken, see Fig. 5.

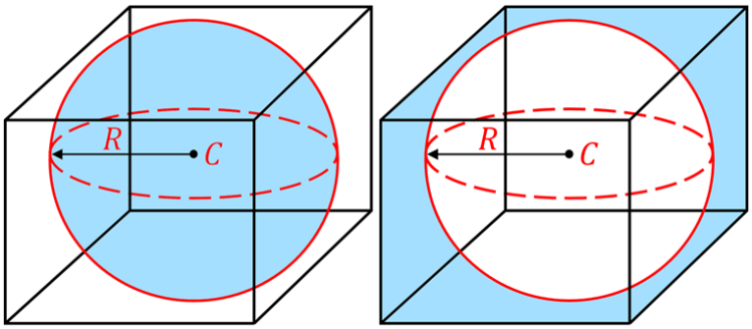

Fig. 5. Part of the space inside or outside the sphere. C is the center of the sphere, R is the radius of the sphere.

3.3 Space inside or outside the cylinder

The infinite cylinder can be set using the coordinates of a point on the axis of rotation, the direction of the axis of rotation, and the radius of the circle. The point is needed to link the vector of the axis of rotation to a certain place in space. Similarly, there are two possible options: a part of the space inside or outside the cylinder is taken (see Fig. 6).

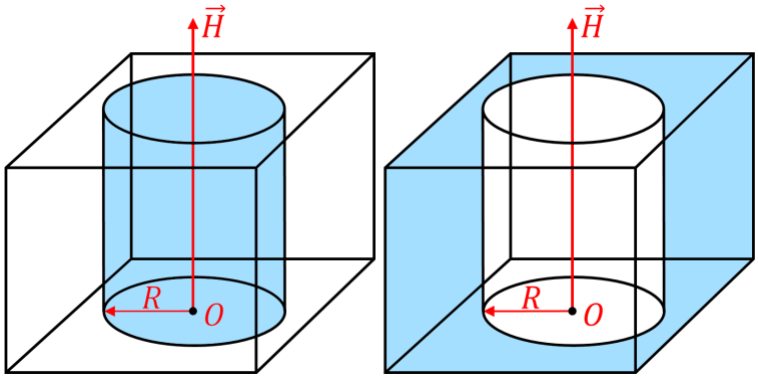

Fig. 6. Part of the space inside or outside the cylinder. O is arbitrary point of the cylinder axis, $\boldsymbol{H}$ is the axis vector, R is the radius of the circle.

3.4 Space inside or outside the cone

The infinite cone is a second-order surface generated by the motion of a straight line (generatrix) passing through a fixed point. In this case, the cone can be set by the coordinate of the vertex of the cone, the vector of the axis of the cone and the value of the angle at the vertex of the cone, i.e., the doubled angle between the axis and the generatrix. As in the case of a sphere, in order to define the primitive completely, it is specified which part of the space is taken: inside or outside the cone (see Fig. 7).

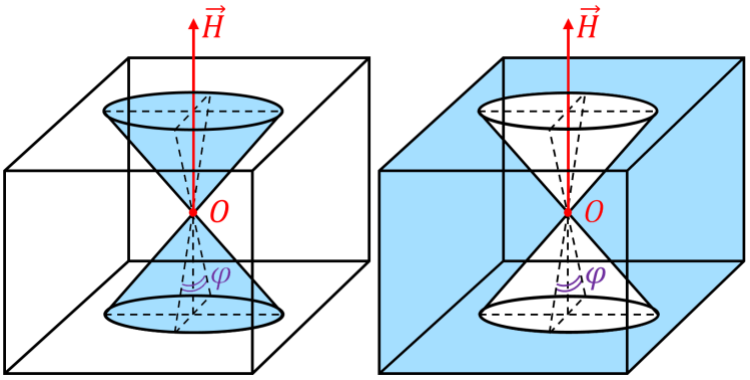

Fig. 7. Part of the space inside or outside the cone. O is the vertex of the cone, φ is the angle at the vertex, $\boldsymbol{H}$ is the axis vector.

3.5 An example of constructing a lens from primitives

Let us consider the application of the set-theoretic intersection operation on a set of primitives by constructing a certain lens. Let us take for example a biconvex lens. We will set in the scene space the coordinates of the lens center point C, the unit vector of the main optical axis of the lens ***axis*** and the width of the lens $width$, see Fig. 8. A biconvex lens has two spherical surfaces, let us denote their radii R_{front} and R_{back}. Obviously, the following conditions must be met:

$$width > 0 \;\&\; R_{front} \geq \frac{width}{2} \;\&\; R_{back} \geq \frac{width}{2}.$$

Let us find the centers of the spheres. If the radius of the sphere R_{front} is the radius of the curvature of the surface in the positive direction of the ***axis*** vector, then the center of this sphere is at the point

$$O_1 = C - \boldsymbol{axis} \cdot (R_{front} - \frac{width}{2}).$$

Similarly, the center of a sphere with radius R_{back} is at the point

$$O_2 = C + \boldsymbol{axis} \cdot (R_{back} - \frac{width}{2}).$$

If the above conditions are met, such spheres will have a non-zero intersection of the $width$ (see Fig. 8). The intersection of the spheres gives a biconvex lens without a rim.

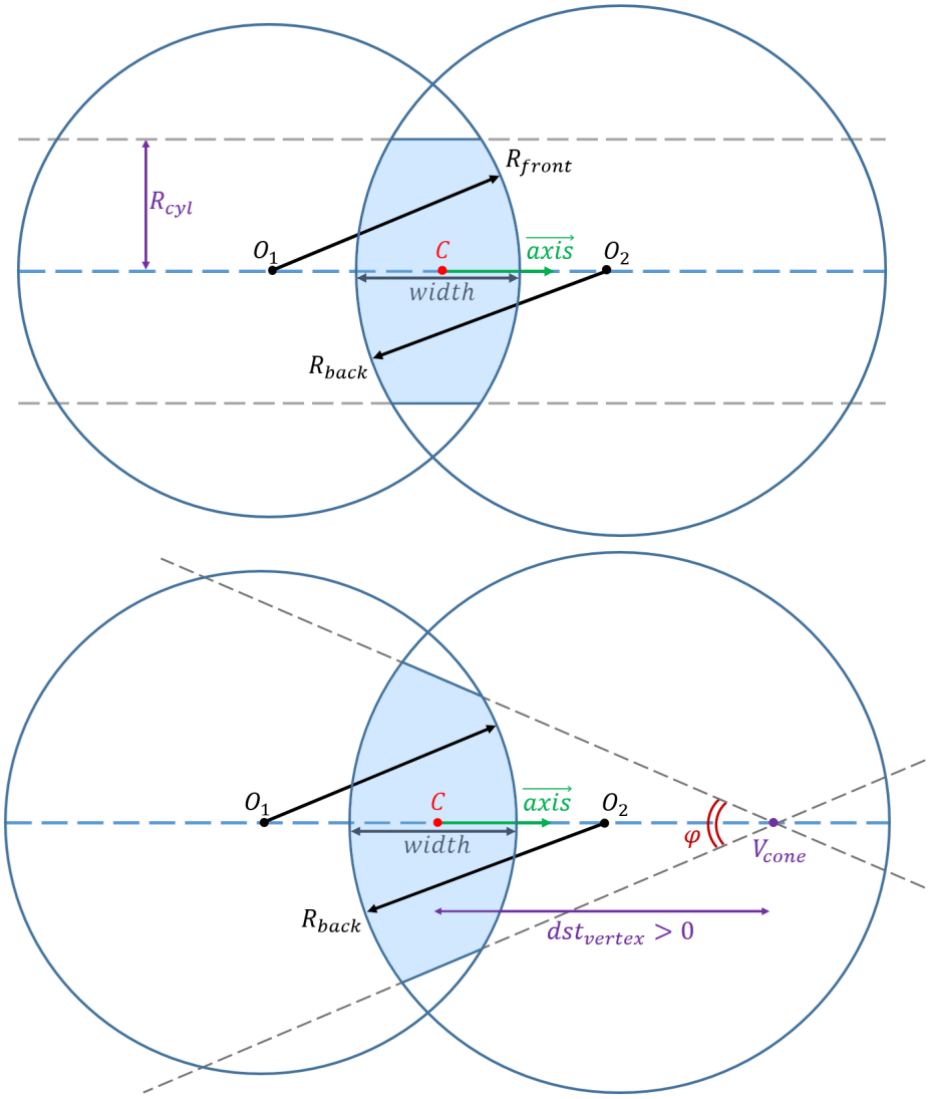

Fig. 8. A biconvex lens is formed at the intersection of the spheres. There is a lens with a cylindrical rim on top, and a lens with a conical rim at the bottom.

Let us add a cylindrical rim to the resulting lens, which is formed when adding a part of the space inside the cylinder to the set of primitives, i.e. the intersection of two spheres and the interior of the infinite cylinder. The cylinder has a rotation axis $\boldsymbol{H} = \boldsymbol{axis}$ and a starting point $O = C$. Let h_{max} be the maximum height of the lens, then the radius of the cylinder $R_{cyl} \leq \frac{h_{max}}{2}$. If this condition is met, a lens similar to the top one shown in Fig. 8 will be obtained.

To set a conical rim, you must specify the vertex V_{cone} and the angle φ of the cone, which will cut off the excess part of the lens. The angle φ should be such that the cone has an intersection with the lens. The vertex of the cone is given by the distance dst_{vertex} from the center of the lens along the main optical axis, then the vertex of the cone can be found by the formula

$$V_{cone} = C + \boldsymbol{axis} \cdot dst_{vertex}.$$

The condition $|dst_{vertex}| \geq \frac{width}{2}$ must be met, and dst_{vertex} can be either positive or negative (see Fig. 8).

Important note: the axis of the rim coincides with the axis of the lens.

The library provides specialized constructors and allows the user to avoid difficulties and errors when designing lenses using set-theoretic intersection.

Below we show how one can create a biconvex lens with a conical or cylindrical rim using the appropriate constructor.

In addition to geometric parameters, refraction indices or materials of the lens interior and the external environment are also set.

3.6 Implementation

The software package is designed as a set of libraries in the C++ programming language. For all six types of spherical lenses, constructor classes are implemented. Examples of possible lenses are shown in Fig. 9. When designing the lens, the correctness of the parameters set by the user is checked. If it is impossible to construct the correct lens based on the parameters, the user receives an error in the form of an exception.

Let us represent a linear polarized zero-thickness tracing ray with attributes based on the work [7] in the form

$$R = \{P_0, \mathbf{d}, \mathbf{X}, \mathbf{Y}, I, \lambda, L_{id}, Op, \Sigma\}, \quad (1)$$

where: $\{P_0, \mathbf{d}\}$ is the mathematical ray, P_0 is the origin of the ray, $\mathbf{d}$ is the direction, $\{\mathbf{X}, \mathbf{Y}, \mathbf{d}\}$ is the associated right-hand coordinate system. In other words, the ray is represented as

$$P(t) = P_0 + t \times \mathbf{d}.$$

The oscillations of the electric vector of electromagnetic wave occur along the $\mathbf{X}$ axis, I is the intensity, λ is the wavelength of the light, L_{id} is the identifier of the point light source that generated the ray. Two rays are coherent if their source IDs are non-zero and match. Op is the optical path of a geometric path traveled from the source, which is used to calculate the current phase of the electromagnetic wave [1]. Σ is the phase jump accumulated during reflections from a denser medium, see [2, 7] for details.

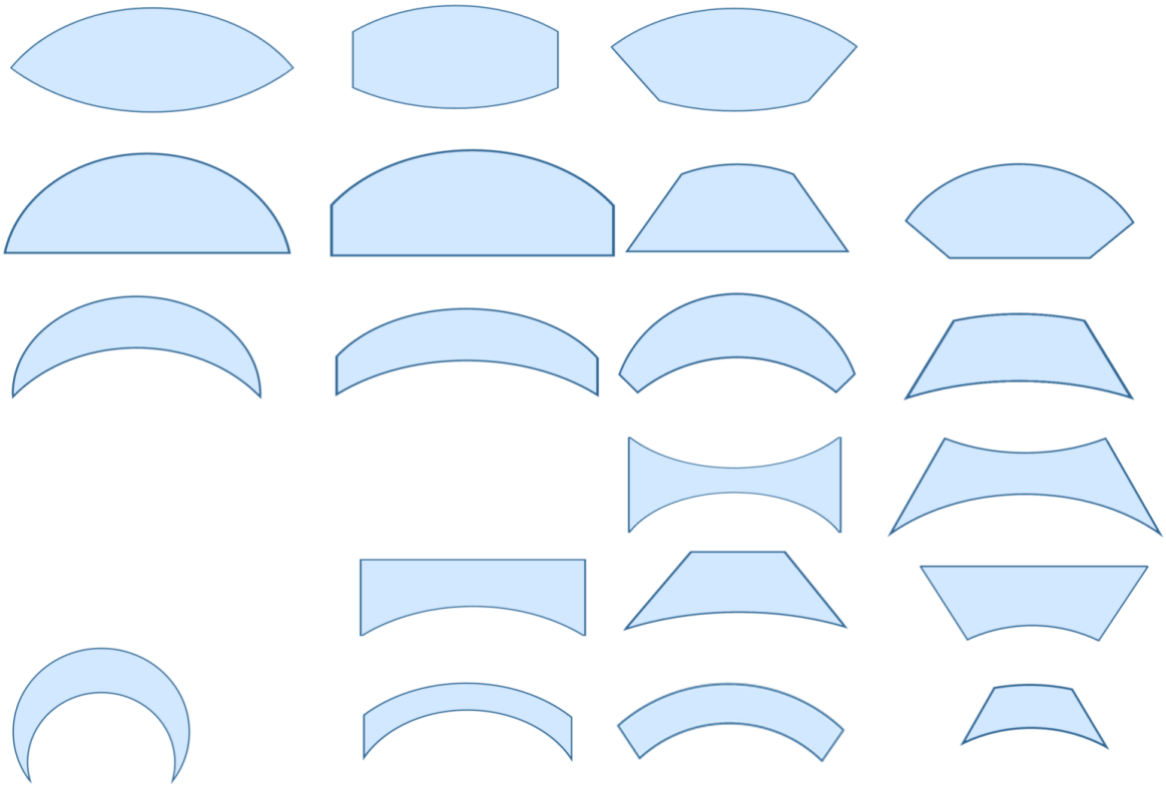

Fig. 9. All types of spherical lenses supported by the software package.

The generated rays (reflected and refracted) inherit some of the attributes, or they are recalculated during the contact of the incident ray with the scene surface. Note that the Fresnel's equations apply.

A corresponding data structure was written for each type of primitive. The primitive is defined, firstly, by the parameters that determine its geometry and location in space, and secondly, by the refraction indices of two media: belonging (interior) and not belonging to the primitive (medium). Basic operations on primitives are the following:

- Use a given point to determine whether it belongs to a primitive.
- Using a given geometric ray, if possible, calculate: a) the value of the parameter t for the point of its intersection with the primitive boundary, b) the coordinates of the intersection point, c) the normal vector at this point. There may be several such points.

The lens is represented as a list of primitives. When tracing, it is necessary to determine which primitive the ray falls on, the coordinates of the hit point, the normal at this point, and the refraction indices of the media. Knowing these values, it is possible to calculate the reflected and refracted rays using Fresnel's formulas. To perform the intersection operation, it is necessary to check for each point that it belongs to all the primitives of the lens list.

Based on the example, we will consider how the set-theoretic intersection operation is implemented. It is based on the well-known *even-odd* principle when calculating intersections of a ray with the boundaries of primitives. In Fig. 10, the ray R falls on the surface of a lens constructed from two primitives: parts of space inside spheres with centers O_1, O_2. The ray R crosses the boundary of the Primitive $Prim1$ at points A and C, the boundary of the Primitive $Prim2$ at points B and D. Thus, we get a set of values of t:

$$t(A, Prim1) - t(C, Prim1) - t(B, Prim2) - t\,(D, Prim2),$$

sort it in ascending order t:

$$t(A, Prim1) - t(B, Prim2) - t(C, Prim1) - t(D, Prim2),$$

analyze the sequence and find the segment BC belonging to both primitives, i.e. to their intersection.

We have considered the simplest case of the mutual arrangement of the ray and the boundaries of primitives. A well-known problem is when the ray hits the intersection point of the primitives themselves. In order to have an accurate idea of how to perform sorting, a different implementation was made for each type of lens.

When creating a lens, the user sets handlers for the three above-mentioned events he is interested in. A user can use the *standard or default handlers* already included in the library, or implement them himself. Each of the three possible events has its own handler class interface. Let us look at the work of standard handlers.

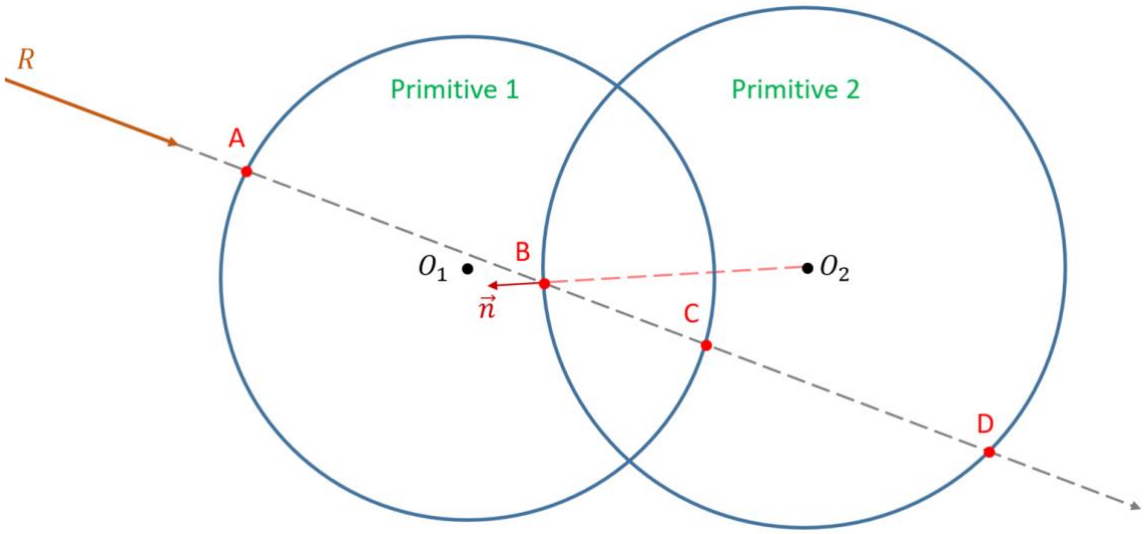

Fig. 10. Intersection points of the ray R with primitives.

The handler for the next calculated output ray allows the user to get a list of calculated output rays after the tracing is completed.

The handler for the case of total internal reflection allows the user to check whether a total internal reflection has occurred after the tracing is completed and, if it has occurred, to obtain two calculated linear polarized rays (see [2, 15]).

The handler for the case of a ray hitting the rim allows the user, after the tracing is completed, to check whether a certain ray has hit the rim and, if it has happened, to get this ray and the coordinates of the hit point.

The handlers are installed after the lens is set. Different lenses may have different handlers. After setting the handlers, the user can trace the ray through the lens. To do this, the user needs to set the input ray, the tracing depth, the energy level of the ray ε, at which further tracing stops, the lens through which the tracing will be performed, and finally calls the tracing function.

3.7 Additional library objects

Spherical lenses represent only the starting set of the library. Already in the course of our experiments, we were faced with the need for such basic geometric shapes as a cube for the instrument glass, a cylinder for the instrument glass, a wedge for experiments with interference. We illustrate these additional objects in Fig. 11.

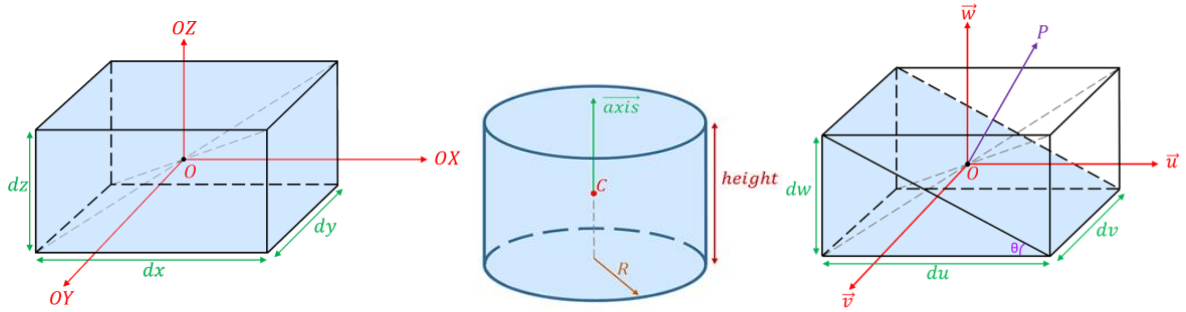

Fig. 11. Additional forms for the interference experiment.

Advanced users can create their own additional forms in a similar way. They should keep in mind possible collisions at the intersection points of two or more primitives and correctly handle in case of a ray hitting such points.

3.8 Sample application

Let us look at the skeleton of a simple application that simulates the passage of rays in a scene consisting of a source, spherical lenses and a screen. Let the light source emit all the rays at once, and we form a set *SR* of linear polarized rays from them. Each ray is described by a starting point and direction:

```
struct LENS_RAY {
  CLENSVector p0, dir;
};
```

And the information load for the ray has the form:

```
struct LENS_PAYLOAD { // see formula (1)
    double lambda;         // wavelength
    double   amplitude;      //   the   amplitude   of   oscillations
                      // of the electric field
    double deltaPhase;     // accumulated jumps of phase
    double oPath;          // the optical path traversed by the ray
    int ch;                // coherence ID
    CLENSVector vp;   // polarization vector
};
```

The ray together with the load is set by the structure:

```
struct LENS_RAY_STORAGE {
  LENS_RAY ray;
  LENS_PAYLOAD payload;
};
```

Let us introduce a set of lenses *SL*. We proceed to the declaration of another spherical lens, namely a biconvex lens with a conical rim using the appropriate constructor (see Fig. 8):

```
using namespace Lens;
double cone_angle = 30;
double vert_dist = 100;
/* Create biconvex lens */
BiconvexLensBuilder builder;
LENS_DATA lens = builder.set_axis(CLENSVector{1, 0, 0})
           .set_center(CLENSVector{0, 0, 0})
           .set_front_radius(200)
           .set_back_radius(400)
```

```
                .set_width(50)
                .set_inner_material(GLASS_INDEX)
                .set_outer_material(AIR_INDEX)
                // Specify conical opaque rim
                .set_rim(coneRim(vert_dist, cone_angle, BORDER))
                .build();
```

The cylindrical rim is set similarly using the following line

```
                .set_rim(cylinderRim(cylinder_radius, BORDER)).
```

One can see that in addition to geometric parameters, refraction indices or materials of the lens interior and the external environment are also set.

And add this lens to *SL*.

Next, we set reactions to events using standard handlers that are available in the library and provide the most natural operations with lenses.

```
LENS_DATA lens = /* . . . declared above */;
/* set default library handlers */
lens.setRayCallback(DefaultRayCallback::create());
lens.setRimCallback(DefaultRimCallback::create());
lens.setTIRCallback(DefaultTIRCallback::create());
```

Let us trace the ray through the *Lens*.

```
/* Starting point and direction */
CLENSVector p0 = { 0, 0 ,0 };
CLENSVector dir = { 1, 0, 0 };
/* Geometrical ray */
LENS_RAY ray = { p0, dir };
/* Information load */
LENS_PAYLOAD payload = /*....*/;
/* Incident linear polarized ray */
LENS_RAY_STORAGE input_ray = { ray, payload };
/* Tracing depth */
LENS_MODE mode = {5};
/* Tracing */
const int result = lensTrace(input_ray, mode, lens);
```

After the trace is executed, the user can get the result and apply the appropriate handler. The value returned by the tracing function is the code of the last event that occurred:

- FRESNEL_SUCCESS means that the required number of rays has been successfully calculated,
- FRESNEL_RIM means a ray was found that hit the rim,
- FRESNEL_TIR means the case of total internal reflection,
- FRESNEL_DATA_ERROR means that error in user data is detected,
- FRESNEL_NO_ENERGY means that the incident ray has an energy less than ε,
- FRESNEL_NO_INTERSECTION means that the ray does not intersect with the lens.

In the first three cases, the programmer can get data using handlers if he has installed them.

```
if (const auto& callback = lens.getRayCallback()) {
  /* Get a list of calculated output rays */
  std::deque<LENS_RAY_STORAGE> rays = callback->getRays();
  /* ..... */
}
```

```
if (const auto& callback = lens.getTIRCallback()) {
  /* Obtain two calculated rays due to
    the case of total internal reflection */
  if (callback->hasTIR()) {
    std::array<LENS_RAY_STORAGE, 2>
         TIR_rays = callback->getTIRRays();
    /* ..... */
  }
}

if (const auto& callback = lens.getRimCallback()) {
  /* Get the ray that hits the side and the coordinates
     of the hit point on this side */
  if (callback->hasRimIntersection()) {
    /* Hit point and ray */
    CLENSVector point = callback->getRimPoint();
    LENS_RAY_STORAGE ray = callback->getRay();
    /* ..... */
  }
}
```

Finally, all the fragmentary operations are defined and we can proceed to consider the structure of the application.

First, we should specify the screen. In order not to program the search for the intersection of rays with the screen separately, we can set it in the form of a thin rectangular parallelepiped (see Fig. 11) and set the trace depth equal to zero. In other words, it is enough for us to catch only the fact that the tracing ray hits it. And by the starting point of the reflected vector, we determine a specific pixel. This element of the scene is also placed in a set of lenses *SL*.

So, we have a set of original light rays *SR* and a set of lenses *SL*. Then the pseudocode of the application looks as follows.

While (*SR* is not empty) {

1. Take a ray *R* from *SR* and remove it from *SR*
2. Look for the nearest intersection of *R* with the objects of the set *SL* via function *trace*
3. If intersection found {
   Get the necessary information using proper handlers
   }
4. If the *result* of *trace* is FRESNEL_SUCCESS {
   Add all output rays to SR}

}

Obviously, the screen handler will collect information from all the rays that hit it and identify proper pixels. It remains only to build an image.

3.9 Function Fresnel: base level of library

Tracing is carried out by rays of linear polarized light. To calculate their interaction with optically isotropic transparent objects, the Fresnel function is developed, which,

based on the application of the Fresnel's equations and the parameters of the ray incident on the boundary between two optically isotropic transparent media, calculates the characteristics of the generated reflected and refracted rays: direction, polarization, and others according to the representation (1). This function also determines the events of total internal reflection and others mentioned above.

4. Conclusions

In this paper, we described the current state of the development of Library of spherical lenses. Testing and debugging were carried out on simple scenes, for example, calculating interference patterns related to fringes of equal thickness [1, 11], including the calculation of Newton's rings in transmitted and reflected light.

We are planning to expand it step by step with other types of geometric shapes: cylindrical and aspherical lenses, other optically isotropic transparent objects necessary for particular experiments, etc.

Note that the library allows one to find out all the data about the progress of ray tracing in the lens (see Fig. 3), and, therefore, the user can link the movement of the rays to the virtual scene, and prepare presentations and reports on the flow of the experiment. Once again, we note that the use of lenses from our library can be done in some part of the user's program. In other part of the program, he can use other objects and lenses from other libraries along with ours.

In our opinion, such autonomous work serves as a convenient means to verify certain solutions. It is likely that in the future the proposed and tested approaches will be adapted one way or another in new versions of existing software.

This work was carried out under state contract with ICMMG SB RAS (0251-2021-0001).

References

1. Born M., Wolf E. Principles of optics: Electromagnetic theory of propagation, interference and diffraction of light. Cambridge University Press, 1980.

2. Debelov V. A., Kushner K. G., Vasilyeva L. F. Lens for a Computer Model of a Polarizing Microscope // Mathematica Montisnigri, Vol. 41, pp. 151–165, 2018.

3. ASAP Optical Software, https://www.photonicsonline.com/doc/asap-optical-software-0001. Accessed 15 Oct 2021.

4. Tracepro, illumination and non-imaging optical design & analysis tool, https://www.lambdares.com/tracepro/. Accessed 15 Oct 2021.

5. Optical Design Software - CODE V | Synopsys, https://www.synopsys.com/optical-solutions/codev.html. Accessed 15 Oct 2021.

6. FRED Software | Photon Engineering, https://photonengr.com/fred-software/. Accessed 15 Oct 2021.

7. OpticStudio – Zemax https://www.zemax.com/pages/opticstudio. Accessed 15 Oct 2021.

8. OpTaliX: Optical Design Software http://www.optenso.com/index.html. Accessed 15 Oct 2021.

9. OpTaliX: Reference Manual, Version 11.10, 2021, http://www.optenso.com/download/optalix_reference.pdf. Accessed 15 Oct 2021.

10. Glassner A. S. Andrew Glassner's notebook soap bubbles: Part 2 // IEEE Computer graphics and applications, Vol. 20, No. 6, pp. 99–109, 2000.

11. Landsber G. S. Optika [Optics], 6th ed. Moscow: FIZMATLIT, 2003 [in Russian].

12. Heidrich W., Slusallek P., Siedel H.-P. An image-based model for realistic lens systems in interactive computer graphics // Proceedings of Graphics Interface '97, Canadian Information Processing Society, 1997, pp. 68–75.

13. Fedotov G. I., Ilin R. S., et al. Laboratory optical devices. Textbook for optical specialties of Universities, 2nd ed. Moscow: Mashinistroenie, 1979. [In Russian].

14. Laboratory polarizing microscope of transmitted light, 2021. URL: http://www.lomo.ru/production/grazhdanskogo-naznacheniya/mikroskopy/mikroskopy-tekhnicheskie/polam-l-213m/. Accessed 15 Oct 2021.

15. Debelov V. A., Vasilieva L. F. Visualization of interference pictures of 3D scenes including optically isotropic transparent objects // Scientific visualization, 2020, Vol. 12, No. 3, pp. 119–136. (doi:10.26583/sv.12.3.11)