Virtual Optics

by Darrin Fong and Tiémé Togola

Integrative Project in Computer Science and Mathematics

420-204-RE, gr. 00871

Amin Ranj Bar

Final Report

2014 05 20

Table of Contents

Project Overview

Story

Concepts in the science of optics can be hard to grasp for students who cannot easily visualize diverse situations in their heads. There are many formulas to remember, and diagrams are not always helpful. Several teachers have developed java applets to try to cope with this problem. However, we find that their implementations are too limited in their capacity to illustrate a varied range of optical simulations. This is where the Virtual Optics project comes in.

System Objectives

The best way to understand something is to experiment with it. Our goal was to create an entertaining and educative Java application that simulates fundamental optical phenomena. By actively familiarising themselves with the simple rules that govern the behaviour of light, users can develop an intuitive understanding of the basics of optics. The application is divided into three sections, namely the learning section, the game section and the lab section. In the learning section, users have access to some documentation about the laws of optics. This documentation can be customised by a teacher for instance, to include more information if desired. The game section consists of a series of levels in which the user can test his knowledge and skills with the various optical objects. In each level, the goal is to hit a target with the ray of light using available objects. There are 15 levels in total. Finally, users can enter the lab section to freely experiment with almost unlimited amounts of light rays and optical components. Users can save and load their projects.

Project constraints and scope

At the beginning of the project, constraints on the number of reflections of a light ray, the number of active components and on component overlap were envisioned. We had to set a limit on the number of reflections to avoid the case where a ray of light bounces back and forth between two mirrors for ever. Next, we predicted that using a large number of rays and other optical components at once would affect the performance of the application. Our prediction was correct, but we decided not to set a limit on the number of active components to let the user interact with more components if performance is not of concern for him. Moreover, we had to prevent optical objects from overlapping each other because our impact algorithm (described in detail in Algorithms) does not support such cases. This issue was solved later on in the project by delimiting each component with a bounding box in which other components cannot enter.

Throughout the evolution of the project, new unexpected constraints appeared such that in the final stage there were more than had been predicted at the start. The first of these concerns the size of the application frame. At first we were thinking of making the application proportionally resizable, but we realized that our interface would not work optimally if the user decided to make the frame very small. Therefore, we decided to fix the size of interface in order to encourage users to leave the frame in full screen for a better experience. The second new constraint related to the scope of the project. During the development phase, the audience of the project remained unchanged; it was always meant to be for students who take their first optics course. However, we at first planned to make it cross-platform but we did not manage to do so because of time restraints. As a result, our application cannot be run from the web or from a mobile device.

Tools and methodology

To develop our Virtual Optics java application we mainly used the Eclipse IDE. Every graphical element was built using the java Swing API, except a few static images that were designed by Darrin Fong using the Paint software. The major process that allowed us to simplify the implementation of certain sections was the creation of a level editor mode. We started off by implementing the lab section with its user interface and all of its optical components. We then implemented the saving/loading system that allows the creation of projects in the lab section. With these two key parts of the application, we then created a level editor mode that basically allowed us to enter the lab section, and create and save some level projects. We could then move these level files to a different folder containing all of the levels, and our game section was already almost completed. We also used this level edition feature to design the dynamic background in the main menu. Besides, we shared each other's code over the web simply using Dropbox.

Critical project events

From the beginning of the implementation phase of the project, we had assumed that we could not avoid imprecision errors due to the way java uses integer values to represent pixels on its 2D plane. Hence, we used the Point type with integer precision to specify the position of all the objects in the interface and to make all the calculations as well. This caused a lot of imprecision errors as expected, so we devised various ways of dealing with these subtle errors. Yet, some problems could not be fixed, because in these cases the computation really needed to be precise to make it work; for instance, when a ray bounces inside a convergent mirror many times. This requires high precision in the calculations. But suddenly, one day near the end of the project development, we realized what our mistake was. Although java uses integer values to represent pixels on a plane, that does not imply that we necessarily have to do the calculations using integer values as well. Soon after we understood this, we changed all the computations in our code to use the Point2D type with double precision (Point2D.double). All we had to do then was to cast the positions to integer just before displaying the components on the plane. This solution solved most of the problems we had been dealing with from the beginning. Unfortunately, since we were running out of time when this major change was made, some issues can still arise (as explained in Results: System Quality).

Project Design

The main problem that was solved to create Virtual Optics was to create a dynamic simulation of the behaviour of light rays in interaction with various optical objects. This implies several sub problems such as: modelling optical objects and their interactions based on the laws of optics, displaying a graphical representation of the simulation, allowing user interactions with the simulation, offering the option to save and load simulations, and finally allowing the users to navigate between the different sections of the application.

Algorithms

Many algorithms are used in Virtual Optics. Here we describe the most important ones.

First of all, there is the impact method which detects when a ray intersects another game component and computes the direction of the reflected or refracted ray (if applicable). This function has a maximum of three nested loops, although the innermost loop generally does not iterate through the entire list. Therefore, the efficiency of this algorithm is O(n^3).

```
void impact(ArrayList<GameComponent>  components) {

        loop (through list of points in path) {

                loop (through components list) {
                        if (path intersects a component) {

                                replace current point by intersection point in path

                                loop (through path)
                                delete the points after the current one in path

                                reflect, refract or stop the ray
                                (depends on type of component that was hit)

                                add the reflected/refracted new endpoint to the path
                        }
                }
        }
}
```

Then, there is the reflection method which returns the orientation of a reflected ray. This algorithm has constant time efficiency because it contains no loop.

```
double reflection(double m1, double m2) {

        compute the incident angle using m1 and m2
        with the incident angle, compute the slope of the reflected ray

        return slope of reflected ray
}
```

Similarly, the refraction method returns the orientation of a refracted ray. It also has constant time efficiency.

```
double refraction(double n1, double n2, double m1, double m2) {

    compute the critical angle using n1 and n2
    compute the incident angle using m1 and m2

    if (incident angle >= critical angle) {

        reflect the ray using reflection(m1, m2)
        return slope of reflected ray
    }
    else {
        with the incident angle and the indices of refraction n1,n2
        compute the angle of the refracted ray

        use the angle of the refracted ray to find its slope
        return slope of refracted ray
    }
}
```

Next, the overlap method in the Lab class checks whether two game components are about to intersect each other. The algorithm uses the bounding boxes of the components to check if they overlap. The efficiency of this algorithm is O(n) since there is only one loop.

```
boolean overlap(ArrayList<GameComponent>  components, int deltaX, int deltaY, int current) {

    loop (through list of components)

        check if the bounding boxes, plus the delta values, of two components intersect
        if they do return true, return false otherwise
}
```

Still in the Lab class, we have the reposition all method which will try to spread out the components on the plane as long as some of them still overlap. This can occur even if we have a system that generally prevents components from overlapping, because sometimes when the user brings a new component on the plane, the initial position of the new component might be inside the bounding box of another. This is when the reposition all algorithm is called to spread out the components. This algorithm has an efficiency of O(n^4) because it consists of four nested loops.

```
void repositionAll() {

    loop (through the list of components) {

        loop (while some components overlap)
        loop (through the list of components) {

            loop (while the two current components still overlap)
                reposition these two away from each other
        }
    }
}
```

Lastly, there is the indices method which acts much like the reposition all function, except that it is for RefractiveZone objects only. It also determines which refractive zones are nested inside other zones and assigns their outer index properties accordingly. This algorithm has an efficiency of O(n^3) because it consists of three nested loops.

```
void indices() {
    loop (through the list of components) {
        if current component is not a refractive zone, skip it

        loop (through the list of components) {
            if current component is not a refractive zone, skip it

            loop (while the two current components still overlap)
                if first component fits into the other, reposition it inside
                else reposition it outside

            if first component is inside the other, set its outer index to the index of the zone
            inside which it is
        }
        if component is not inside any zone, set its outer index to 1.0
    }
}
```

Some other algorithms used are specific to each component. For example, the intersection method is different depending on the game component type, because they can have different shapes. Therefore, it takes a different algorithm to compute the intersection point of a line segment with that given shape.

Classes

The implementation is split into two packages called game components and user interface.

Game Components

This package contains all the entities with which the user can interact in Virtual Optics. The game components implement the Serializable interface so that they can easily be written to a file. Additionally, the RefractiveZone class implements the Cloneable interface because one functionality in the Lab class (figure 18) needs to make copies of such objects. The types that extend the abstract OpticalObject class are game components that have the ability to bend a ray of light in some way. Game components that are not optical objects simply block incident rays. The LineEq class models a linear equation to help with various kinds of calculations. Mainly, LineEq is used to represent the normal line of an optical object.

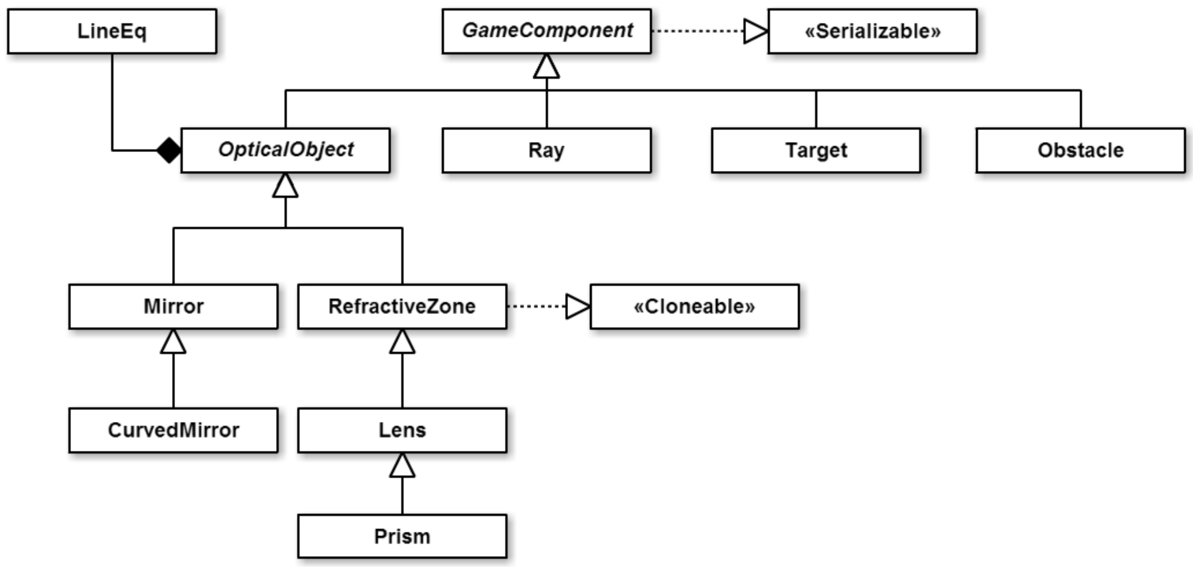


Figure 1: Class structure inside the gameComponents package

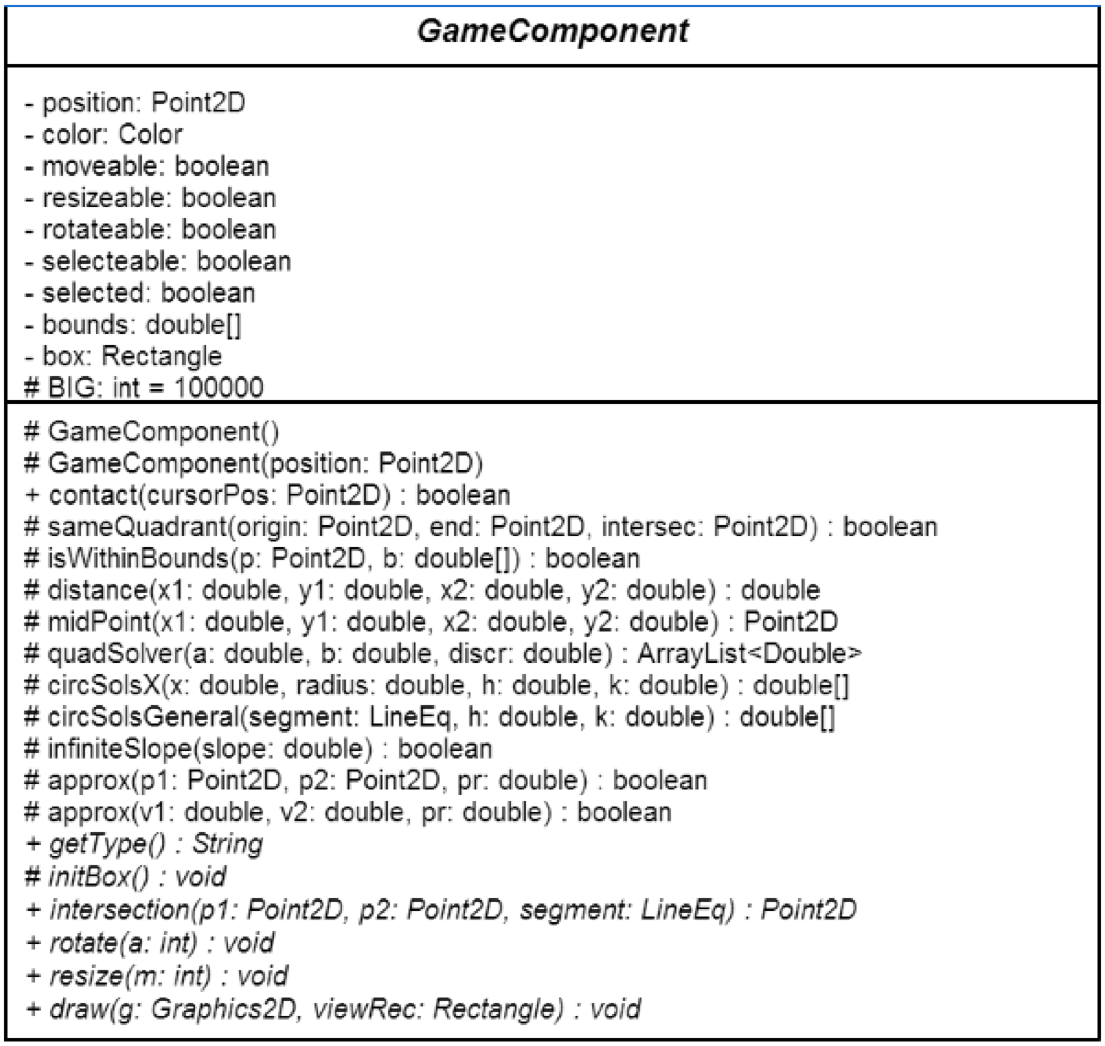

GameComponent
- position: Point2D
- color: Color
- moveable: boolean
- resizeable: boolean
- rotateable: boolean
- selecteable: boolean
- selected: boolean
- bounds: double[]
- box: Rectangle
# BIG: int = 100000
# GameComponent()
# GameComponent(position: Point2D)
+ contact(cursorPos: Point2D) : boolean
# sameQuadrant(origin: Point2D, end: Point2D, intersec: Point2D) : boolean
# isWithinBounds(p: Point2D, b: double[]) : boolean
# distance(x1: double, y1: double, x2: double, y2: double) : double
# midPoint(x1: double, y1: double, x2: double, y2: double) : Point2D
# quadSolver(a: double, b: double, discr: double) : ArrayList<Double>
# circSolsX(x: double, radius: double, h: double, k: double) : double[]
# circSolsGeneral(segment: LineEq, h: double, k: double) : double[]
# infiniteSlope(slope: double) : boolean
# approx(p1: Point2D, p2: Point2D, pr: double) : boolean
# approx(v1: double, v2: double, pr: double) : boolean
+ getType() : String
# initBox() : void
+ intersection(p1: Point2D, p2: Point2D, segment: LineEq) : Point2D
+ rotate(a: int) : void
+ resize(m: int) : void
+ draw(g: Graphics2D, viewRec: Rectangle) : void

Figure 2: GameComponent class

The GameComponent class contains many properties and methods that are common to all game component types, as can be seen in Figure 2[1]. For instance, every component needs a position on the plane as well as a bounding box surrounding it. The BIG constant is used in subclasses to compute the position of the end point of a Ray object. The contact method detects when the user clicks this component, and most of the other methods are utility functions used in subclasses for multiple computations. One last method worth mentioning is the one called intersection. As its name implies it computes and returns the point of

[1] Getters and setters are generally not included in the diagrams. Overridden methods in the subclasses are also omitted. Very detailed information is available in the Javadoc located in the doc folder, inside the project folder.

intersection between a given ray segment and this component. Each game component has a different implementation of this method.

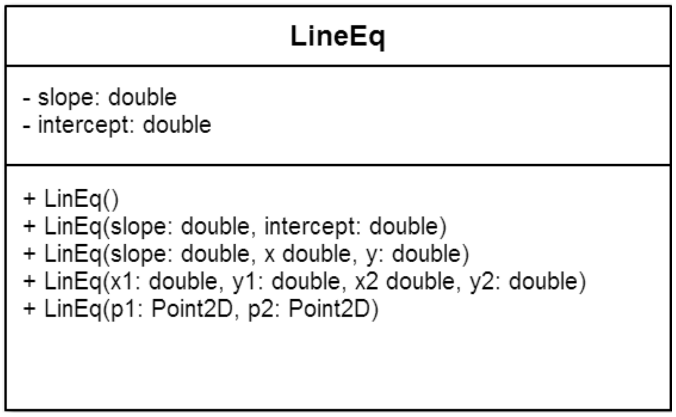

LineEq
- slope: double
- intercept: double
+ LinEq()
+ LinEq(slope: double, intercept: double)
+ LinEq(slope: double, x double, y: double)
+ LinEq(x1: double, y1: double, x2 double, y2: double)
+ LinEq(p1: Point2D, p2: Point2D)

Figure 3: LineEq class

In figure 3 above, the details of the LineEq class show how given a few arguments the appropriate constructor automatically calculates values of the slope, the intercept, or both, to obtain a linear function. Below, in figure 4, notice that a Ray object is made out of an array list of points. Each point in this list indicates a change in direction of the path followed by this ray. Therefore, the path can be decomposed in segments with endpoints path.get(i-1) and path.get(i), and then each modelled with LineEq. These LineEq segments can then be individually tested for intersection with other components in the plane.

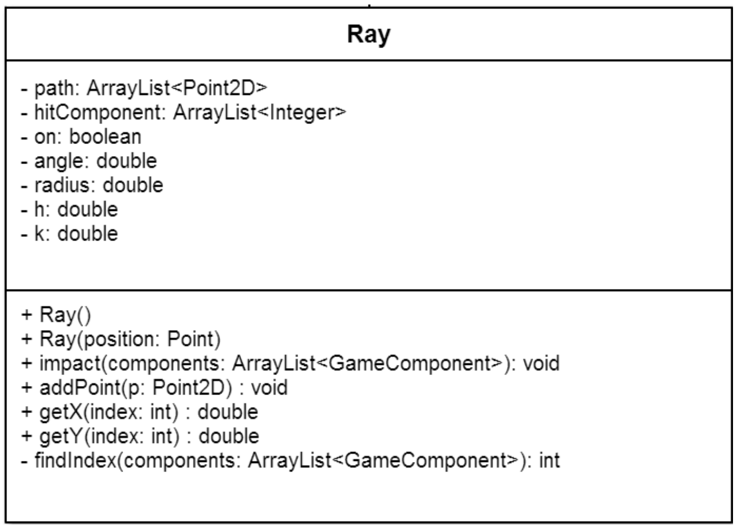

Ray
- path: ArrayList<Point2D>
- hitComponent: ArrayList<Integer>
- on: boolean
- angle: double
- radius: double
- h: double
- k: double
+ Ray()
+ Ray(position: Point)
+ impact(components: ArrayList<GameComponent>): void
+ addPoint(p: Point2D) : void
+ getX(index: int) : double
+ getY(index: int) : double
- findIndex(components: ArrayList<GameComponent>): int

Figure 4: Ray class

The Target type models a simple game component that triggers a reaction when it is hit by a light ray. This is done by the react method, which ensures that the target component only triggers a reaction when the incident ray has the appropriate color. Therefore the user has to make that he hits a given target with the ray of the correct color. Target objects need a radius property as well as h and k (conventional name for x and y coordinates of the center of a circle) because they have a circular shape.

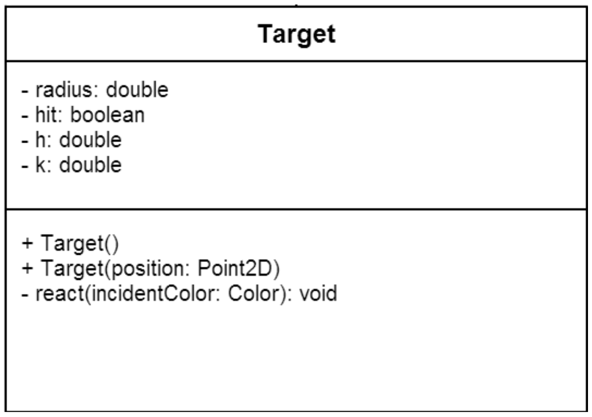


Figure 5: Target class

Obstacle objects need a width and a height because they have rectangular shapes. Their shape can be adjusted with the scale method. This class also needs a list of collision points together with a list of Ray objects. These two properties are then used in the drawing method to know where the obstacle has been hit by a given ray, and thus draw fading light gradients at these positions.

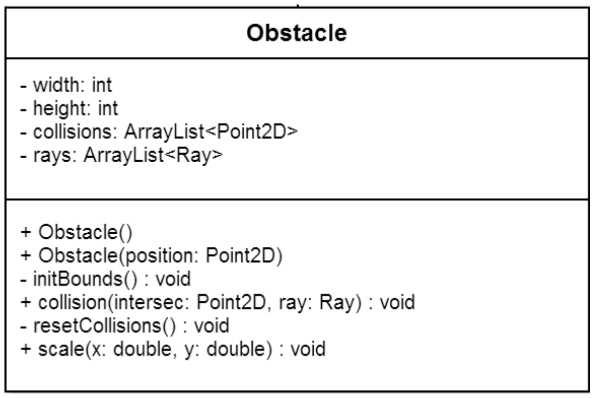
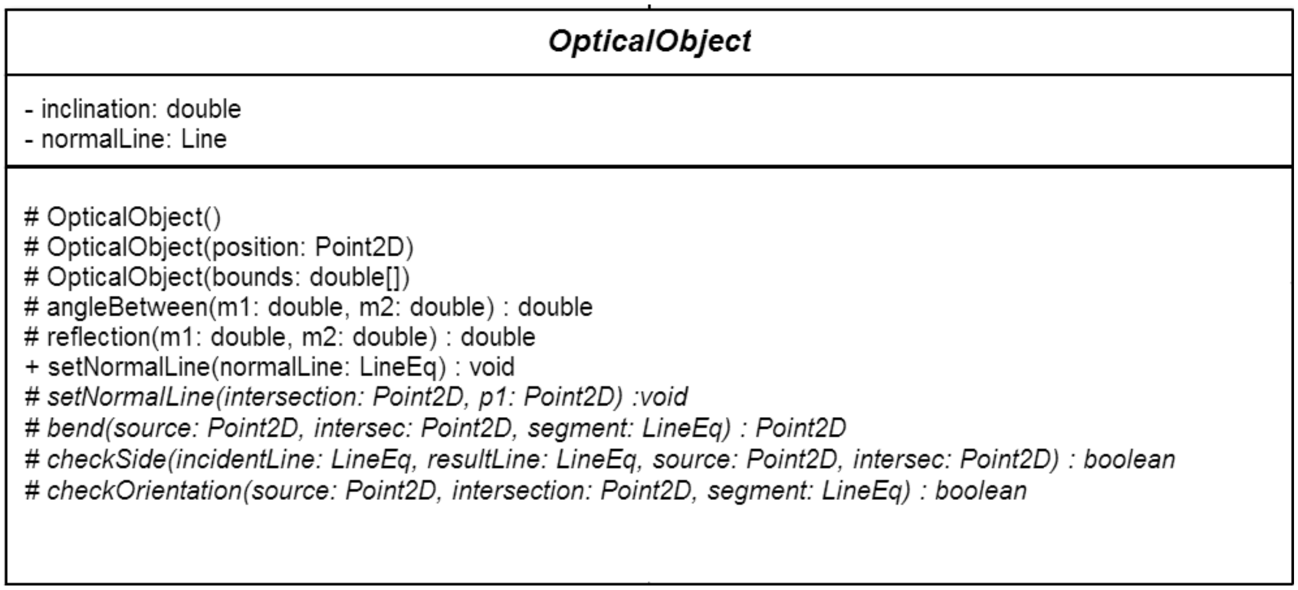

Obstacle
- width: int
- height: int
- collisions: ArrayList<Point2D>
- rays: ArrayList<Ray>
+ Obstacle()
+ Obstacle(position: Point2D)
- initBounds() : void
+ collision(intersec: Point2D, ray: Ray) : void
- resetCollisions() : void
+ scale(x: double, y: double) : void

Figure 6: Obstacle class

As mentioned previously, optical objects are components that have the added ability to interact with Ray objects. This is what the bend method accomplishes. Depending on the type of optical object (whether it is a plane or curved mirror, or a lens, etc.), the inclination of the component and its normal line, the bend function modifies the direction of an incident ray to effectively bend it. The angleBetween method is used in the computation to return the angle between the incident ray and the normal line of an optical object. CheckOrientation and checkSide are used to determine whether the incident ray of light is impacting the optical object on its active surface, and to ensure that the bend method orients the ray in the correct direction, respectively. Furthermore, notice that one of the overloaded setNormalLine methods is abstract. This is so because different optical objects have different ways of setting their normal lines. This method is therefore overridden accordingly in the subclasses.

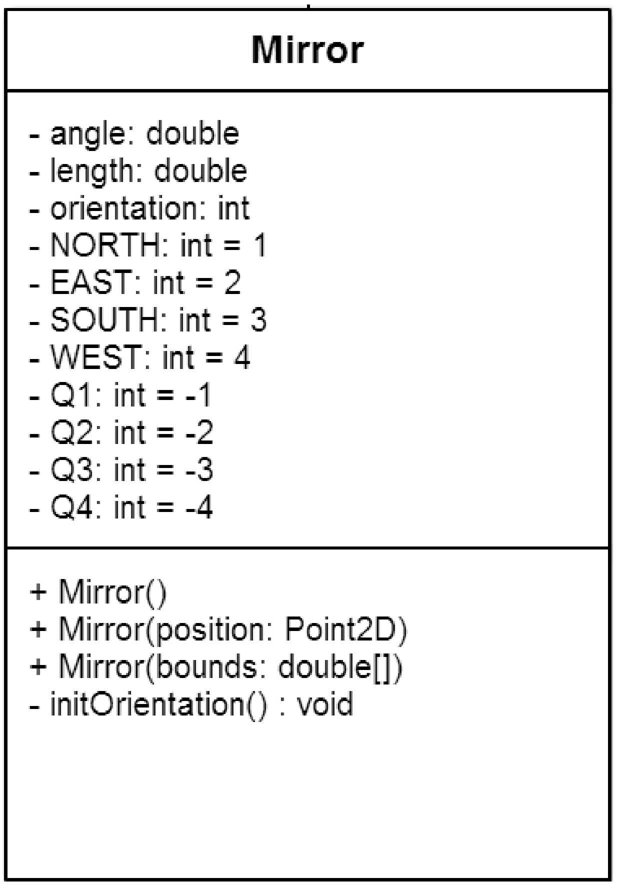


```
OpticalObject

- inclination: double
- normalLine: Line

# OpticalObject()
# OpticalObject(position: Point2D)
# OpticalObject(bounds: double[])
# angleBetween(m1: double, m2: double) : double
# reflection(m1: double, m2: double) : double
+ setNormalLine(normalLine: LineEq) : void
# setNormalLine(intersection: Point2D, p1: Point2D) :void
# bend(source: Point2D, intersec: Point2D, segment: LineEq) : Point2D
# checkSide(incidentLine: LineEq, resultLine: LineEq, source: Point2D, intersec: Point2D) : boolean
# checkOrientation(source: Point2D, intersection: Point2D, segment: LineEq) : boolean
```

Figure 7: OpticalObject class

Finally, still in the OpticalObject class shown in figure 7, there is the reflection method which takes the slope of an incident ray and the slope of a normal line to calculate the slope of the reflected ray. This method is placed here rather than in the Mirror class (figure 8) because mirrors are not the only optical objects that can reflect a ray. Indeed, refractive zones also need the reflection method because they can reflect a ray that has an angle of incidence greater than the critical angle. This case is explained on page 15.

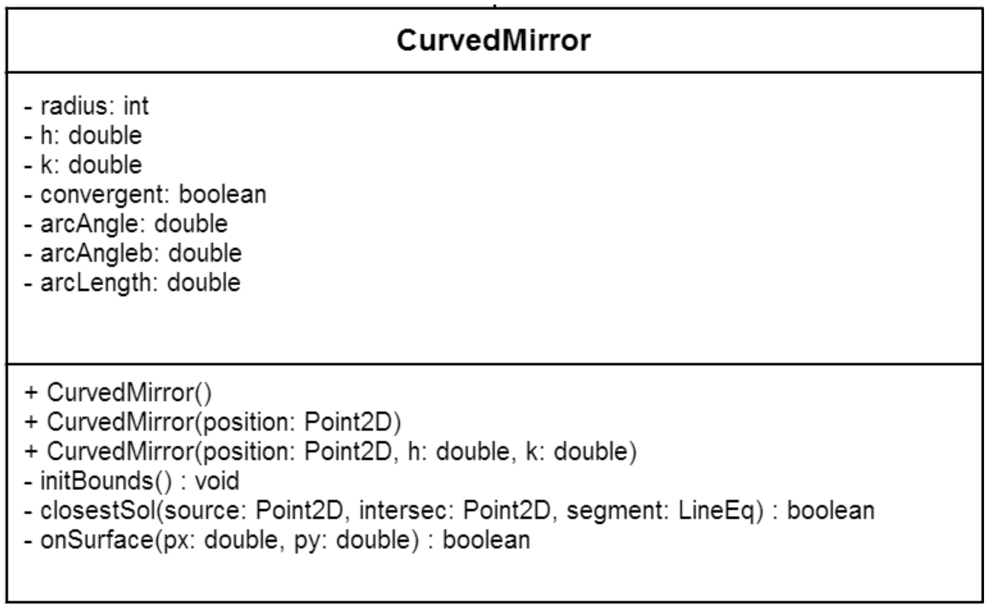
The Mirror class models the behavior of a plane mirror. In Virtual Optics, such a mirror can have a certain angle of inclination, a variable length and a specific orientation which depends on its inclination. The orientation of a mirror is determined inside the initOrientation method, and can take from a discrete set of values stored in the constants. Each constant indicates towards which direction the reflective side of the plane mirror is pointing. Q1, Q2, Q3 and Q4 stand for the four quadrants of a Cartesian plane (e.g. Q1 is also the equivalent of North-East). So, for instance, a plane mirror with reflective side towards the bottom of the screen has the SOUTH orientation. Therefore, in this case if an incident ray is coming from the north, it will not be reflected. It will stop because it hit the non-reflective side (opaque) of the plane mirror. This non-reflective side is represented with another color at rendering time, so that the user can see the orientation of each active plane mirror.



```
Mirror

- angle: double
- length: double
- orientation: int
- NORTH: int = 1
- EAST: int = 2
- SOUTH: int = 3
- WEST: int = 4
- Q1: int = -1
- Q2: int = -2
- Q3: int = -3
- Q4: int = -4

+ Mirror()
+ Mirror(position: Point2D)
+ Mirror(bounds: double[])
- initOrientation() : void
```

Figure 8: Mirror class

The CurvedMirror class illustrated below, obviously shares many characteristics with the Mirror class, so it is only natural that it should extend it. However, since curved mirrors have arc shapes, several properties and methods were added. An object of this type is modelled with the equation of a circle, hence the use of variables radius, h and k again. Curved mirrors also have a reflective and a non-reflective side, but the orientation is not determined in the same way. It partly depends on whether the mirror is convergent (reflective side towards the center of the circle) or divergent (reflective side pointing radially outward from the center of the circle), and also on the return value of the closestSol method. The later tells if the intersection point of an incident ray with this curved mirror is the closest of two possible solutions (a circle equation generally has two solutions).



```
CurvedMirror

- radius: int
- h: double
- k: double
- convergent: boolean
- arcAngle: double
- arcAngleb: double
- arcLength: double

+ CurvedMirror()
+ CurvedMirror(position: Point2D)
+ CurvedMirror(position: Point2D, h: double, k: double)
- initBounds() : void
- closestSol(source: Point2D, intersec: Point2D, segment: LineEq) : boolean
- onSurface(px: double, py: double) : boolean
```

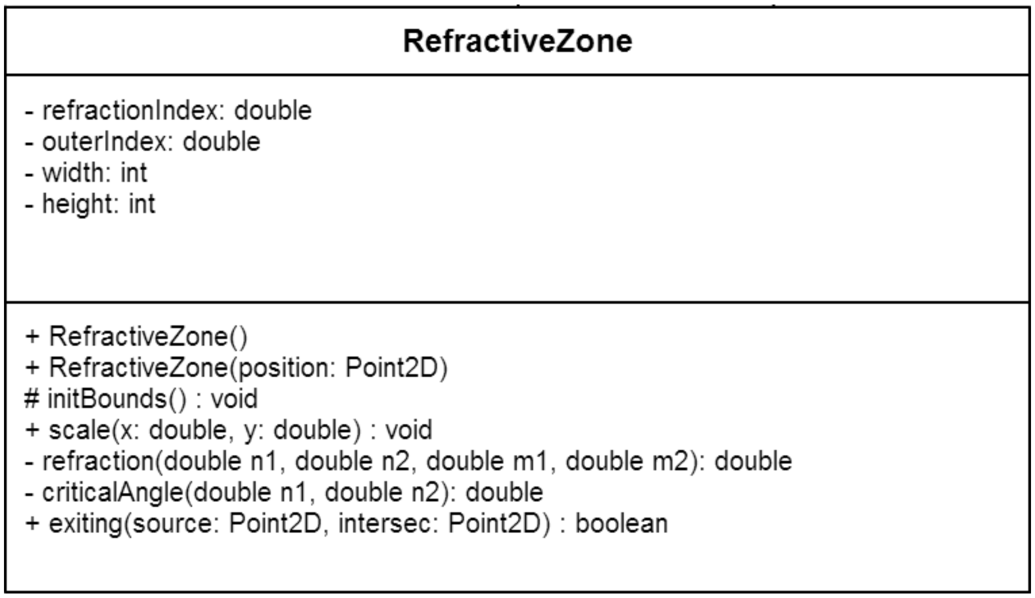
Figure 9: CurvedMirror class

The initBounds method in figure 9 initializes the values of the endpoints of this curved mirror depending on its arc angle and arc length. Using these endpoints, the onSurface method can tell if a point (px, py) is on the surface of the arc or not.

Figure 10 on page 17 shows how the RefractiveZone class models optical objects that have the ability to refract incident ray of lights. In Virtual Optics, general refractive zones have a rectangular shape, hence the width and height properties. But all types of refractive zones have a refraction index, which indicates how much an incident ray will be refracted. This also depends on the value of the outer index. Simply stated, an index tells the density of a material or a zone. So, when a ray goes from a low-density material to a high-density material, it will refract (bend) towards the normal line. Conversely, when a ray was in a zone with a small refraction index and enters a zone with a greater index, it will refract away from the normal line. This is why it is necessary for each zone object to have a refraction index as well as an outer index. The default value for the outer index is 1.0 (refraction index of null-density material, i.e. void or air). However, if a refractive zone is nested inside another, its outer index will take the value of the refraction index of the zone in which it is located.

Virtual Optics allows multiple levels of nesting; there is no limit. Inside a refractive zone, the user can place other refractive zones, but also other game components such as mirrors or targets. When a ray intersects a zone, the refraction method is called. The arguments m1 and m2 are the slopes of the incident ray and the normal line, respectively. The arguments n1 and n2 are the refraction indices of the current and destination zones respectively. When the ray is leaving the zone (checked using the exiting method) n1 is refractionIndex and n2 is outerIndex. When the ray is entering the zone, n1 is outerIndex and n2 is refractionIndex.

Just like the rectangular obstacle objects, a RefractiveZone component can resized with the scale method. The x argument indicates the amount by which the width will be scaled, and the y argument indicates the amount by which the height will be scaled.



```
RefractiveZone

- refractionIndex: double
- outerIndex: double
- width: int
- height: int

+ RefractiveZone()
+ RefractiveZone(position: Point2D)
# initBounds() : void
+ scale(x: double, y: double) : void
- refraction(double n1, double n2, double m1, double m2): double
- criticalAngle(double n1, double n2): double
+ exiting(source: Point2D, intersec: Point2D) : boolean
```

Figure 10: RefractiveZone class

The Lens type extends RefractiveZone and works in a similar way to refract rays. Yet, because it has a very different shape, there are essential modifications to its structure. First of all, a lens is always convergent in our application. Its shape is delimited by two arcs that are joined at their endpoints. This last characteristic greatly increases the complexity of our model of a lens, compared to other components. Since it has two arcs, a lens needs two circle equations, which in turn means that it needs two h values (h1 and h2), two k values (k1 and k2), two arc lengths, and so on. Consequently, the Lens class also needs a way of determining which of the two arcs an incident ray has hit. The which method was implemented to accomplish this task precisely.

Just like curved mirrors, lenses need to evaluate if a given point is on their surface (one of their two arcs) or not. As a result, Lens also has a onSurface method, although slightly different in its implementation. At last, the leaving property acts like the exiting method of the superclass to check whether a ray is leaving or entering the optical object.

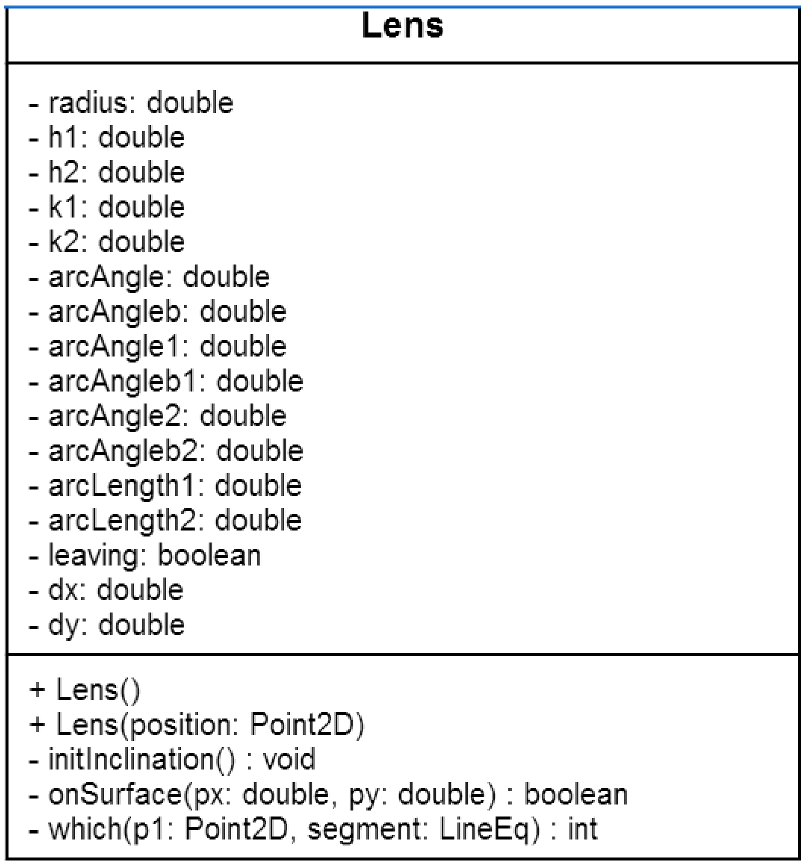


Figure 11: Lens class

Prism is yet another type of refractive zone. It extends Lens rather than RefractiveZone because it shares more similarity with the first than the second. A prism object has the shape of an equilateral triangle. The position of its vertices are stored in vertex1, vertex2, and vertex3 and then used in initPolygon to initialize the shape variable. Line1 is the side of the prism with endpoints vertex1 and vertex2, line 2 is the side with endpoints vertex2 and vertex3, and line 3 has endpoints vertex3 and vertex1. The side variable stores the length of these segments, and the angle property keeps track of the rotation of the prism. Using the sides represented with LineEq type and the inherited refraction method, this class bends a ray that hits its surface.

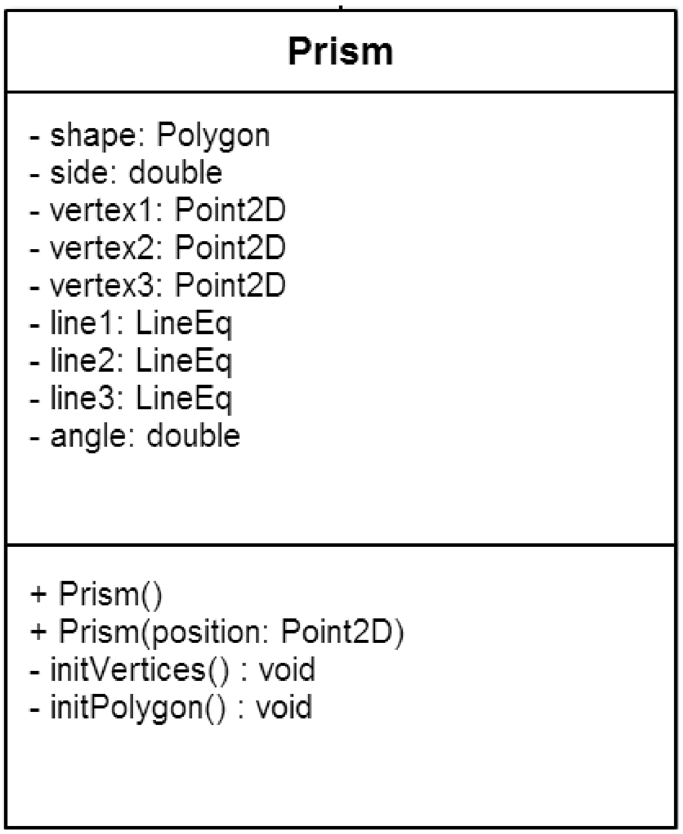


Figure 12: Prism class

User Interface

This package contains all the components that are visible to the user such as JFrames, JPanels and JLabels. They represent what the user will be interacting with when using our program. We decided to use Java's Swing graphical user interface (GUI) because that is what we learned during our courses and also because they are simple to use and are already present in the default JDK library.

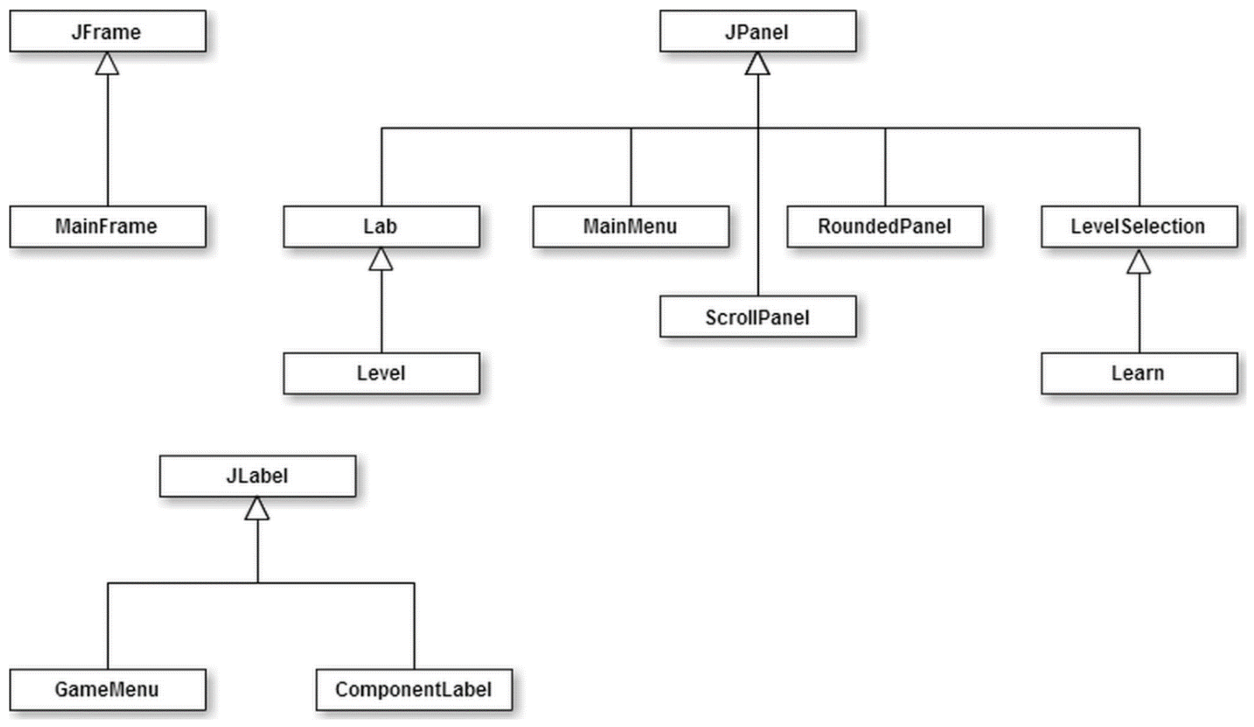


Figure 13: Class structure inside the userInterface package

The MainFrame class represented by figure 14 below contains the frame field which is where the application will be run in. The currentPanel field represents the current panel that is displayed in the MainFrame, there will only be one single panel present at any time in the MainFrame. The final integer constants fields represent their respective panels and will be used to switch between them. The main method will be where the frame of the application will be created. The changePanel class is static because it will be called every time a panel change has to occur, it takes the panel that has to be changed as an integer and the progress as in the level the user has selected to play in case the panel to be changed is a Level panel.

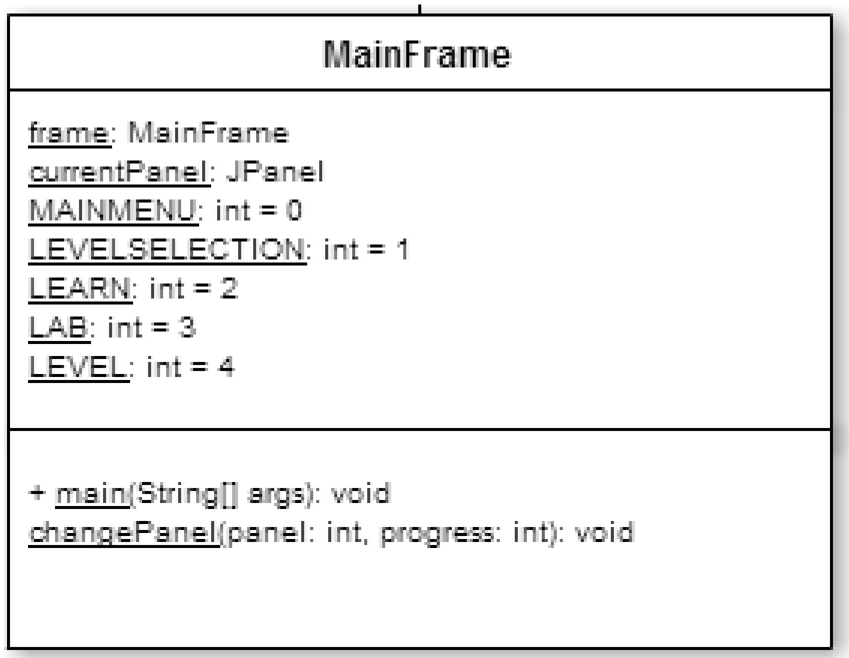


Figure 14: MainFrame class

The MainMenu class represented by figure 15 contains BufferedImage fields that are associated to an image which will be added to their respective JLabel and onto the MainMenu panel. The dynamicBackground field is a Level panel that will be displayed behind everything else on the panel and the dynamicBackgroundTimer is used to animate the dynamicBackground every time a listener is called.

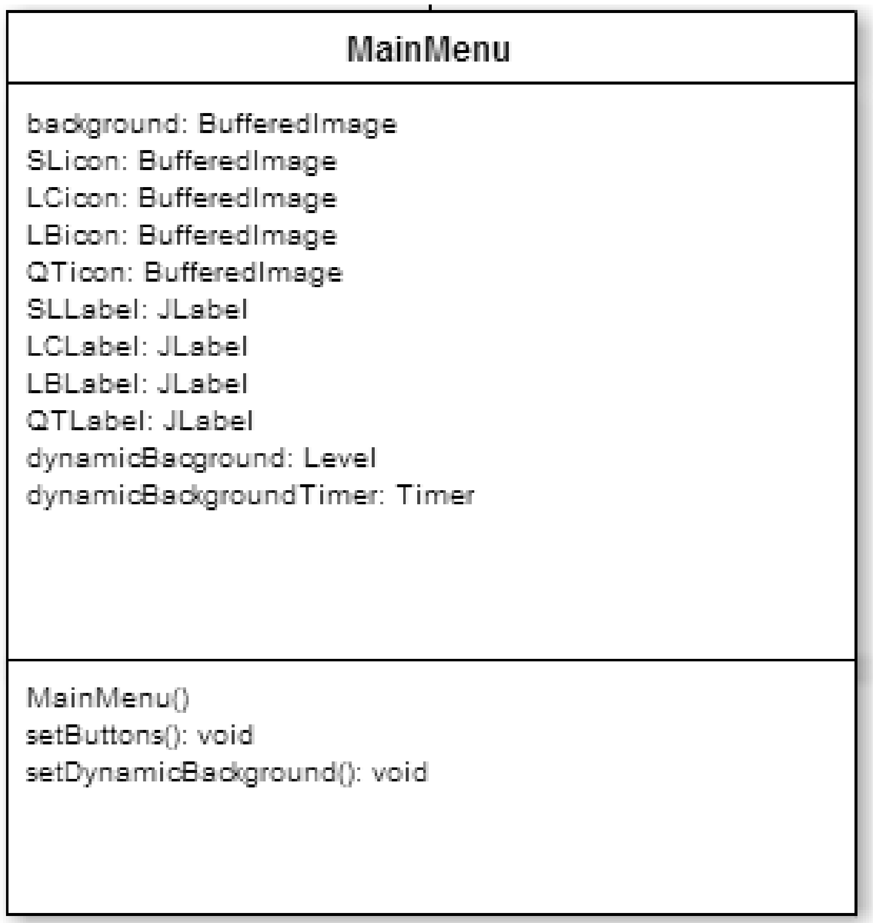


Figure 15: MainMenu class

The LevelSelection class represented by figure 16 consists of a panel where the user can choose which level he wants to play in Level mode. The fields mainly consist of the levels that are currently displayed, the maximum level that can be played, the progress of the user, various BufferedImages and JLabels that represent the icons the user can press. The methods are mainly used to get Images from files, set the images to the JLabels, and check if the user can play the level he clicked and to display all the elements that are visible to the user as well as their position.

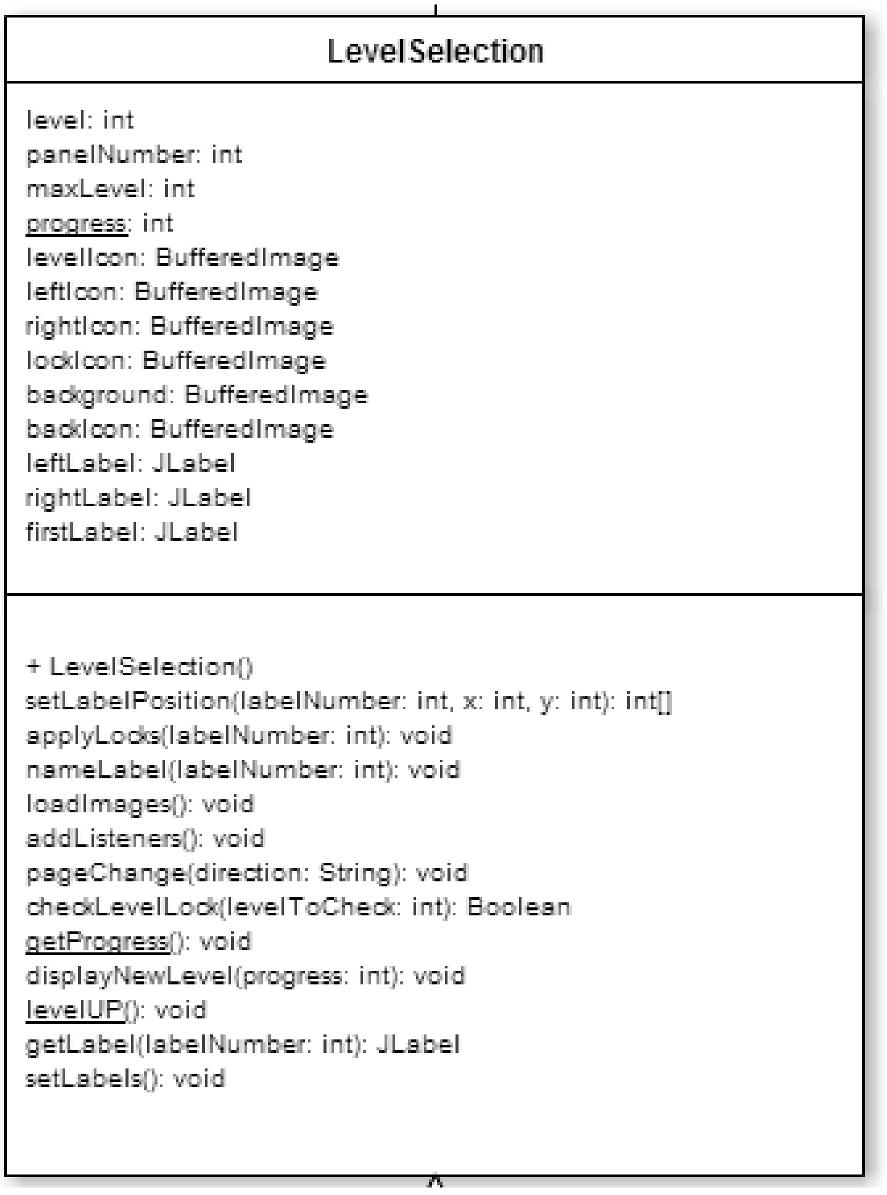


Figure 16: LevelSelection class

The Learn class represented by figure 17 is similar to the LevelSelection class in terms of layout and GUI because it extends the prior mentioned class. However in the place of displaying a Level panel when the icons are clicked, it will show a RoundedPanel in the same panel, which contains information that are predetermined in the form of a text file and picture file.

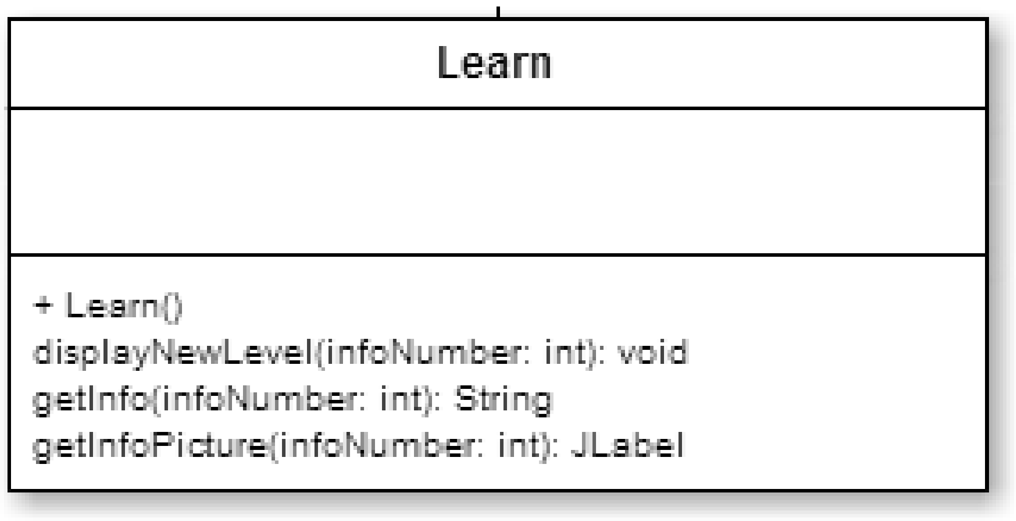
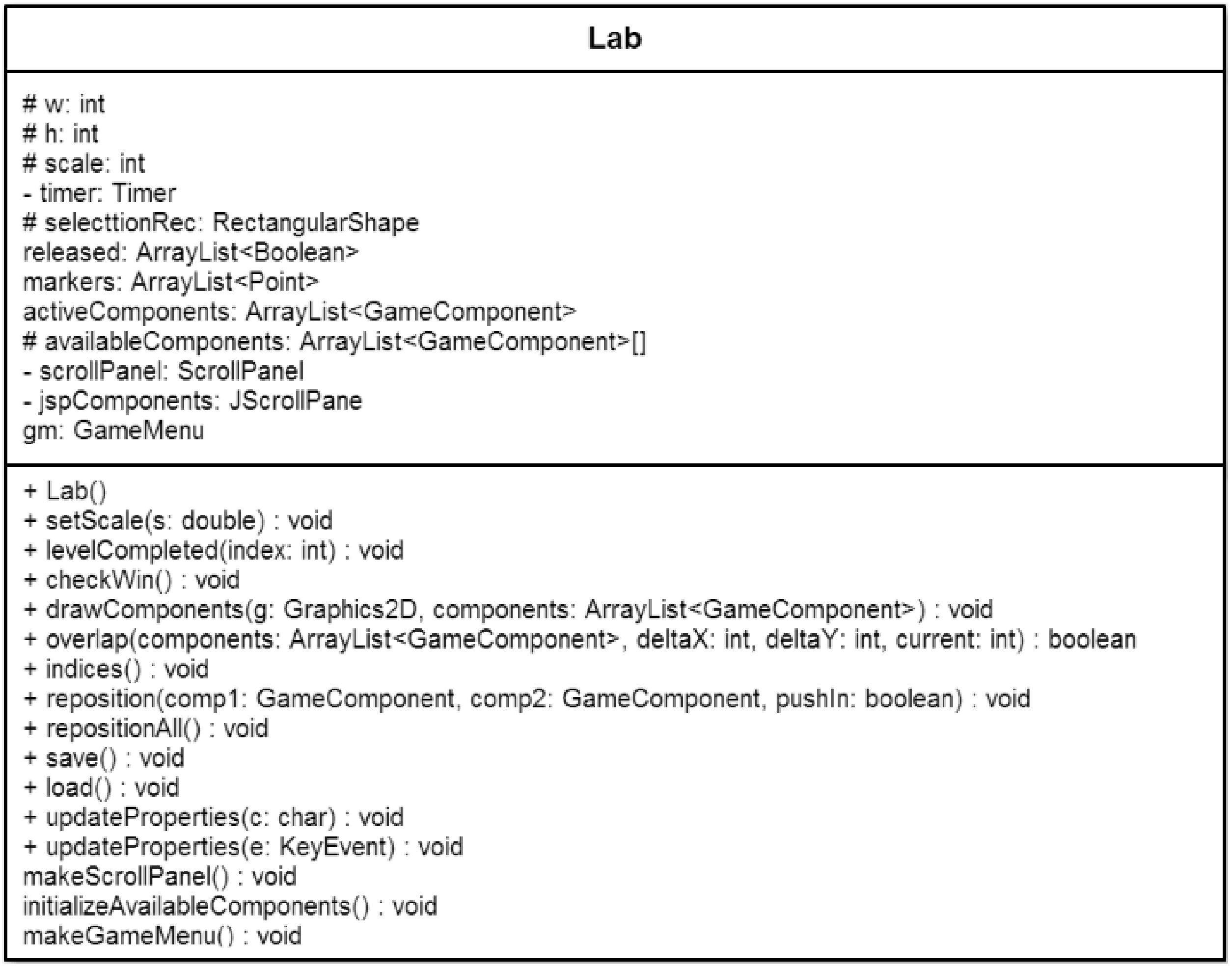

Learn
+ Learn()
displayNewLevel(infoNumber: int): void
getInfo(infoNumber: int): String
getInfoPicture(infoNumber: int): JLabel

Figure 17: Learn class

The Lab class represented by the figure 18 is a panel where the user can interact with various game components. The fields w and h hold the width and height of the panel, while the scale variable is used with the setScale method to zoom in or out. The timer is used to animate translations of the camera view on the plane. The four array lists are there to manage the active and available components. Some methods worth mentioning are the reposition functions and the update properties functions. The first are used to spread out components on the plane whenever they overlap, while the later are called by the listeners of the Lab class to modify some fields in the selected components. For instance, when a user rotates a component, its angle or inclination property is updated through the update properties methods.



```
Lab

# w: int
# h: int
# scale: int
- timer: Timer
# selecttionRec: RectangularShape
released: ArrayList<Boolean>
markers: ArrayList<Point>
activeComponents: ArrayList<GameComponent>
# availableComponents: ArrayList<GameComponent>[]
- scrollPanel: ScrollPanel
- jspComponents: JScrollPane
gm: GameMenu

+ Lab()
+ setScale(s: double) : void
+ levelCompleted(index: int) : void
+ checkWin() : void
+ drawComponents(g: Graphics2D, components: ArrayList<GameComponent>) : void
+ overlap(components: ArrayList<GameComponent>, deltaX: int, deltaY: int, current: int) : boolean
+ indices() : void
+ reposition(comp1: GameComponent, comp2: GameComponent, pushIn: boolean) : void
+ repositionAll() : void
+ save() : void
+ load() : void
+ updateProperties(c: char) : void
+ updateProperties(e: KeyEvent) : void
makeScrollPanel() : void
initializeAvailableComponents() : void
makeGameMenu() : void
```

Figure 18: Lab class

The Level class extends the Lab class and is represented by the figure 19, is a panel where the user will be able to interact with various predetermined GameComponents. The fields represent the current level that is being displayed in the panel, the timer that detects for how long the target has been hit by a light ray, an information panel that is present at the beginning of the level and disappear when the user clicks on the Level panel, and a background if the Level panel is used for a dynamicBackground. There are classes used to get the tip that will be displayed in the infoPanel, as well a method that will get the images needed to display this panel, and a method that animates the GameComponents present in the panel every time this method is called.

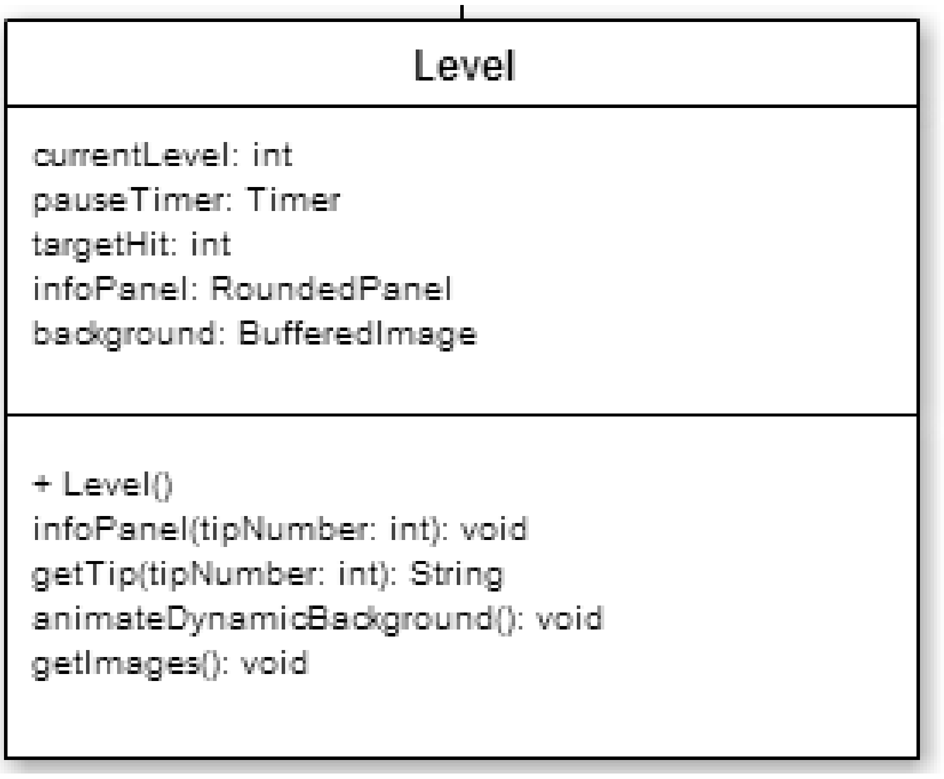

Level
currentLevel: int
pauseTimer: Timer
targetHit: int
infoPanel: RoundedPanel
background: BufferedImage
+ Level()
infoPanel(tipNumber: int): void
getTip(tipNumber: int): String
animateDynamicBackground(): void
getImages(): void

Figure 19: Level class

The ScrollPanel class represented by the figure 20 is a scroll panel where available components to the Lab panel will be displayed in the form of ComponentLabel. There will be a field which represents the components that are available to the user, and an active components list which represent the current active components on the Lab panel. There will be one principal method which is to allow the user to add a GameComponents to the list of active components and therefore adding them to the Lab panel, by clicking on one of the ComponentLabels.

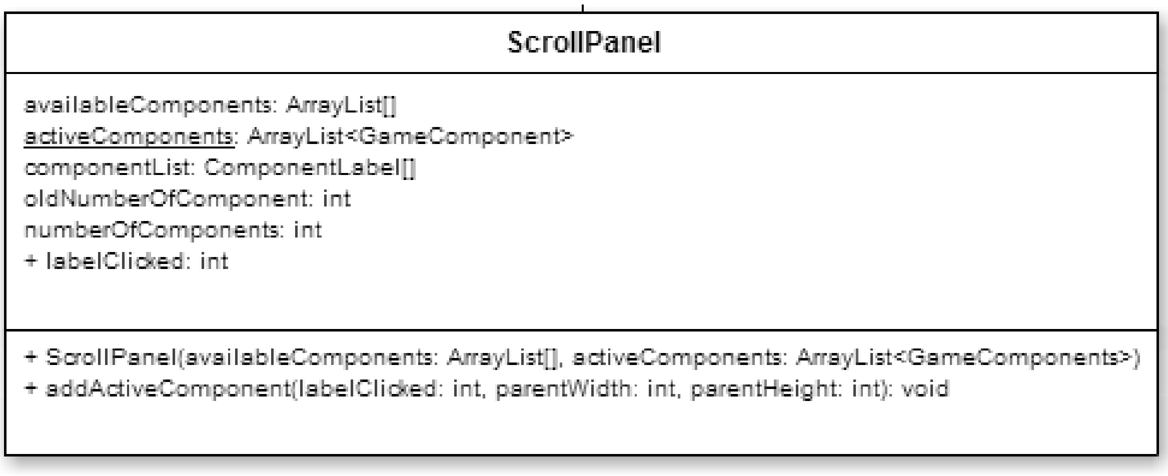

ScrollPanel
availableComponents: ArrayList[]
activeComponents: ArrayList<GameComponent>
componentList: ComponentLabel[]
oldNumberOfComponent: int
numberOfComponents: int
+ labelClicked: int
+ ScrollPanel(availableComponents: ArrayList[], activeComponents: ArrayList<GameComponents>)
+ addActiveComponent(labelClicked: int, parentWidth: int, parentHeight: int): void

Figure 20: ScrollPanel class

The ComponentLabel class represented by figure 21 is a label where the component it represents, contained by the component field, is displayed. It serves the purpose of adding that component to the list of active components.

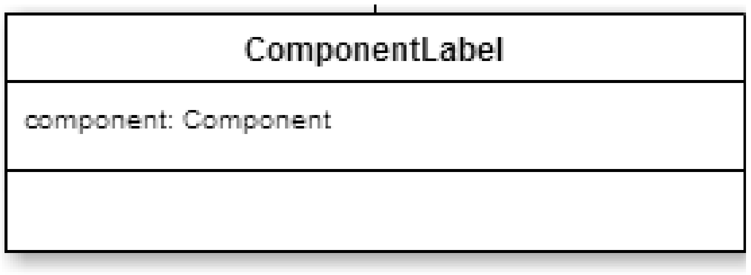
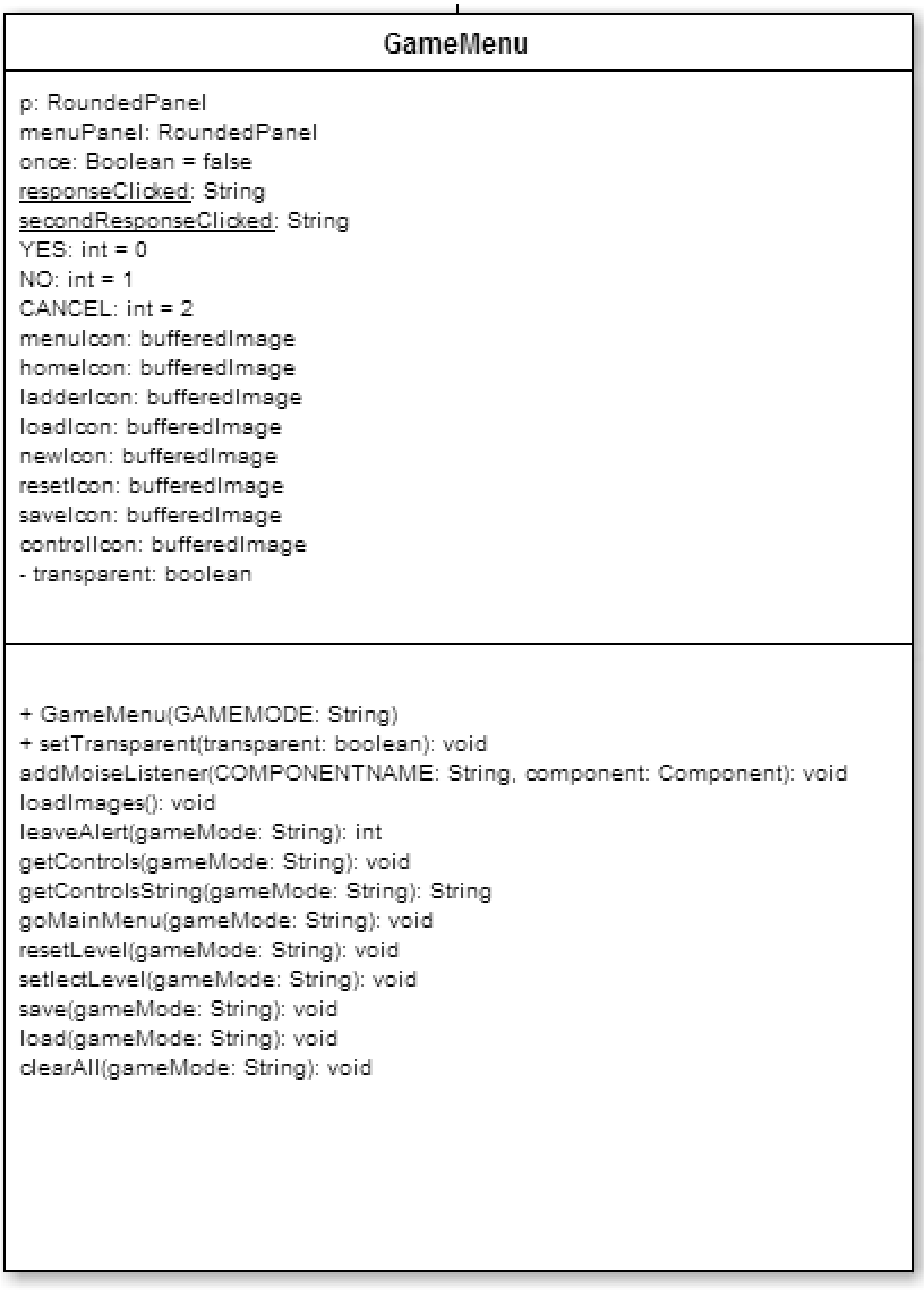


Figure 21: ComponentLabel class

The GameMenu class represented by figure 22 (on the following page) is an icon where when pressed by the user, will display a panel on top of the current Level or Lab panel containing various icons such as home, controls, clear all, reset, level selection, save and load. These images are displayed according to their BufferedImages and JLabels. Listeners will be added to the menu icon, and all the aforementioned icons in the addMouseListener method. A method that does their respective function will be assigned to each of the icon’s listeners so an action will be performed when that icon is pressed.



```
GameMenu

p: RoundedPanel
menuPanel: RoundedPanel
once: Boolean = false
responseClicked: String
secondResponseClicked: String
YES: int = 0
NO: int = 1
CANCEL: int = 2
menuIcon: bufferedImage
homeIcon: bufferedImage
ladderIcon: bufferedImage
loadIcon: bufferedImage
newIcon: bufferedImage
resetIcon: bufferedImage
saveIcon: bufferedImage
controlIcon: bufferedImage
- transparent: boolean

+ GameMenu(GAMEMODE: String)
+ setTransparent(transparent: boolean): void
addMoiseListener(COMPONENTNAME: String, component: Component): void
loadImages(): void
leaveAlert(gameMode: String): int
getControls(gameMode: String): void
getControlsString(gameMode: String): String
goMainMenu(gameMode: String): void
resetLevel(gameMode: String): void
setlectLevel(gameMode: String): void
save(gameMode: String): void
load(gameMode: String): void
clearAll(gameMode: String): void
```

Figure 22: GameMenu class

The RoundedPanel class represented by figure 23 is a panel with rounded corners and has a shadow. This class is created for aesthetic purposes. Its fields consist of the stroke size or the size of the panel's sides, the color of its shadow, the size of the corners, and the panel's background's color.

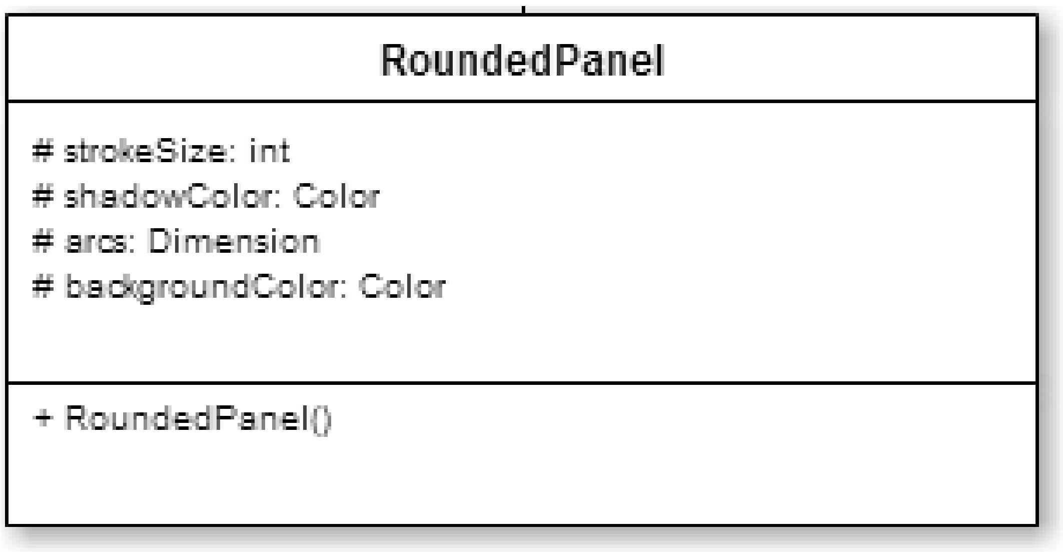


Figure 23: RoundedPanel class

First of all, for the main menu, we decided to incorporate a dynamic background in which the player can interact with the components to get a first impression of how they all function. The dynamic background also moves on its own in a way that every component will interact with each other with or without the user's interference, so it remains entertaining. The four JLabels present will remain simple and representative of what their respective purpose is.

For the level selection section, we decided to keep it simple and only display what is essential to the user i.e. the back button which's design is representative of its function, the level label where their respective level number and availability, in the form of a lock, is painted on the label and the left and right button represent well their functions and is optimally placed on the bottom of the page which is also very intuitively sound for the user. The same can be said to the Learning center since they are visually close to the same. For the information panels that are displayed at the click of the mouse, they are pleasing to look at, and the pictures along with the text area is a good way to tell the user what he has to know, just like the form of a university level textbook.

For the Level and Lab sections, the in game menu is very well placed in the top left corner of the screen, the point farthest from the user, which will cause minimal interference with his activities. The icon that represents the in game menu is simple and yet representative because it consists of the three primary colors of human's color perception, therefore represents well Visual Optics. The icons in that menu are simple and represent well their respective functions such as home button in the form of an icon of a house, the level selection icon in the form of a ladder representing different levels. There is also the clear all button in the form of a blank sheet, the reset button shown by an icon of re-cycling, the available controls function represented by a control button on a Windows keyboard, and the load and save icons represented by their most recognized forms of a floppy disk and download icon respectively.

Furthermore, the scroll panel is on the right side of the frame because the majority of the population is right handed and therefore will grab their tools, in our case is GameComponents, with their right hand. The user is also given the choice to browse the available components with the scroll bar that adjusts its length according to the size of the frame and the choice to browse with his mouse wheel if he has one. The click-to-add method is also very intuitive as the cursor transforms into the "finger selection" cursor when placed on a clickable ComponentLabel.

To finish, all the GameComponents that are drawn represent what they are meant to be because this is how they would be displayed in a 2D manner if viewed from a single side. The yellow highlight tells the user exactly which elements are selected. The marker that are present on the side of the screen also help the user navigate through his GameComponents along with the zoom in and out functions and the background's "drag" function.

It is to be noted that all the icons and backgrounds present in our program are designed by the team except for the font of the titles and numbers, which is imported from the Java Library.

Methods of Evaluation

For this project, we had not planned on doing any kind of formal testing or evaluation, such as having a beta testing phase for instance. We acknowledge that it would have been beneficial to do so, but due to time constraints and lack of experience this was not possible. Instead, our methods of evaluation consisted of individual judgement by the developers and ongoing testing.

What is meant here by individual judgement is the following. Whenever the implementation of a part of the project was completed, we asked ourselves if a user from our audience (science students taking their first optics course) would be satisfied with it. We would pretend to be users for a moment and try to make use of a feature that was just implemented. This is also what we did to test the project at each stage of the development. We would try everything the user could possibly think of doing to see if the system crashed. When it did, or when we were not satisfied with the result, we would go back over the code and see if there were better solutions.

For example, when the dragging system for the game components was first implemented, it seemed to be working nicely. But then we tried moving the mouse very quickly while dragging a component, the component would not follow the mouse pointer as it was supposed to. We modified the code to take this special case into account, and now the user can drag components across the panel as fast as desired without any problem. At the very end of the development phase, we used the same procedure to conduct a final evaluation of the product. The following section describes the results of this evaluation.

Results: System Quality

Virtual Optics was our first experience in developing a Java application of this scale. We are proud of what we have accomplished because it shows how much we have learned in the last two years. In our final product there are things that we appreciate and others that we do not. We measure the quality of our system based on whether or not each of its features fulfill their assigned tasks. Now we will take a closer look at those aspects by going through all three sections of the application. The results of our evaluation are summarized in table 1 at the end of this section.

The learning section was intended to be a place where the user can quickly go refresh his mind about basic optics notions. And this is what it is; it works very much like a student note book. A feature that we particularly like is that the user can even customize the information by modifying the text files in the learning folder. We are also satisfied that the visual design of this section is like we wanted it to be. Namely, that a small panel pops up on top of the rest of the page when the user clicks on a learning topic. On the other hand, we do not like that the different learning topics are titled with irrelevant numbers. Ideally, each topic button would be labelled with a meaningful title. In addition, we knew that we would not cover everything there is to know about optics since this is too much for a simple reference guide, but we should at least have provided a default set of notes for each phenomenon that is simulated in Virtual Optics. We only provide five, but total internal reflection is not one of those (for example) yet it is a phenomenon that the user can observe in our simulation.

Next, in the game section there is one main point with which we are satisfied, and that is the difficulty level. We were afraid at first the levels would be too easy to complete, but they turned out to be much more challenging than what we first thought they would be. What is good about this is that the user really has to think about different ways of controlling the direction of light rays in order to complete a level. The player has to try different strategies and get an intuition for where the ray should go, which is precisely what the goal of this application is in the first place. On the down side however, we do not like the transition between levels. The dialog box that appears to notify the user that the level is completed does not really fit with the rest of the design, and winning a level does not feel very rewarding. Even after the last level, nothing happens. Moreover, we find it annoying that the button to access the game section from the main menu is still labelled “storyline” although we finally decided not include a story to our game. Relatedly, the images that appear in the bottom left and right corners of the main menu page are not very relevant anymore since we abandoned the idea of having a storyline.

Finally, the third and last section of our application, the laboratory, is where we evaluate the quality of the optical simulations. Again, we are not totally pleased with the quality of this section. The main source of our disappointment here is precision. As previously mentioned, we made an important change to the code towards the end of the development phase. We converted a lot of integer-precision calculations to double-precision. However, vertical slopes (very close to infinity) and horizontal slopes (very close to zero) cannot be precisely represented by double values. Therefore, we believe that one of the causes for the minor malfunctions that can occur is that some of the logic we had set in place to deal with vertical and horizontal slopes (as well as a few other functions) can sometimes fail to recognize such cases. We tried, but did not succeed, to fix this issue before the deadline.

Fortunately, the precision problem only occurs in a limited number of cases and does not undermine the user experience too much. Apart from that, the simulation works surprisingly well. Multiple light rays can interact with many components at once to form complex paths. Meanwhile, the user can modify a great range of properties of the system and the simulation will dynamically adapt. It is rewarding to know that we can simulate real physical phenomena using algorithms that we formulated ourselves.

Table 1: Evaluation Sheet

Feature	*Was it implemented?*	*Does it fulfill its task?*	*Does it work properly?*	*Comments*
Learning section	Yes, as intended	Not quite, because the default amount of info is limited	Yes	
5 tutorial levels	Yes, as intended	Not quite, it does not introduce all the controls	Yes	The game is so simple, only the controls should be explained more
10 other levels	Yes, as intended	Yes, users develop an intuition for the behavior of light	Yes	New and better levels can easily be created by developers by uncommenting the code that enables the level editor
Auto-save user progress	Yes, as intended	Yes	Not quite, the user cannot reset the progress from the application	Also, the progress keeps incrementing beyond 15
Save/load user projects	Yes, as intended	Yes	Yes	Users can navigate the file system
In-game menu	Yes, as intended	Yes, offers a lot to the user	Yes	
Scroll pane of available components	Yes, as intended	Yes, users click on component to add it	Yes	The usefulness of the scroll bar is questionable
Reflection	Yes, as intended	Yes	Yes	
Refraction	Yes, as intended	Yes	Yes	
Total internal reflection	Yes, as intended	Yes	Yes	
Dispersion	No			Due to time constraints
Ray	Yes, as intended	Yes, models a ray of light	No, imprecisions can occur	
Obstacle	Yes, as intended	Yes, adds a some challenge in the levels	Yes	We did not implement animated obstacles
Target	Yes, as intended	Yes	Yes	The look should be improved

Feature	*Was it implemented?*	*Does it fulfill its task?*	*Does it work properly?*	*Comments*
Mirror	Yes, as intended	Yes, models a plane mirror	Yes	
Curved mirror	Yes, as intended	Yes, models a curved mirror	Yes, rays can reflect many times inside	Can be convergent or divergent
Refractive Zone	Yes, as intended	Yes, models a refractive zone	Yes	
Lens	Yes, as intended	Yes, models a convergent lens	No, sometimes a ray will diverge for no reason. Linked with imprecision problem	Can only be convergent, and its two surfaces are always symmetric
Prism	Yes, as intended	Yes, models, an equilateral prism	No, sometimes a ray will bend in the wrong direction for no reason. Linked with imprecision problem	It would have been nice to have the dispersion feature. This component would make use of it.
Camera translation	Yes, additionally	Yes, improves user experience	Yes	
Zoom in and out	Yes, additionally	Not quite, the zoom is temporary	Yes	
Markers	Yes, additionally	Yes, improves user experience	Yes, but becomes slow when many active components	
Light effects	Yes, additionally	Yes, makes the simulation a bit more realistic	No, does not propagate on multiple components	Would be nice if did not only work on obstacles, but on all components
Multiple selection rectangle	Yes, additionally	Yes, improves user experience	Yes	

One last thing that we are proud of having achieved is that Virtual Optics provides various features that are generally not available in basic optical simulation applets on the web. These are often restrained to a horizontal optical bench where a limited number of components cannot be rotated. In Virtual Optics, users can place a variety of eight different components on an infinitely big plane. Every component can be rotated and resized, and the number of light rays is unlimited. Users can even manipulate multiple components at once (e.g. select two light rays and rotate them both at the same time), and nest many refractive zones one inside the other. This diversity allows for much more freedom and complexity.

Project Management

Table 2: Timeline

Task	*Planned Date*	*Actual Date*	*Status*	*Assigned Person*	*Notes*
Find a project idea.	Jan. 31	Jan. 31	Complete	Team	Changed idea two times.
Plan the features to implement.	Feb. 7	Feb. 7	Complete	Team	We kept coming up with new ideas until the end.
Define the class structure.	Feb. 14	Feb. 14	Complete	Team	This changed often.
Develop the essential algorithms.	Feb. 14	Feb. 20	Complete	Togola	Other algorithms were developed later on.
Design the user interface.	Feb. 14	Feb. 14	Complete	Fong	An intuitive interface.
Code the algorithms for the optical simulations and the classes that model game components.	Feb. 28	Apr. 3	Complete	Togola	This took way more time than expected. Worked on other things meanwhile.
Code the classes in the user interface package.	Feb. 28	Mar. 10	Complete	Fong	Modifications were made until the end.
Code the event-driven system for the dynamic simulation.	Mar. 8	Mar. 12	Complete	Togola	This step was easier than expected.
Code the event-driven system for the menus and the scroll pane in the lab.	Mar. 8	Mar. 19	Complete	Fong	Many unexpected complications with the scroll pane.
Finish coding the lab section.	Mar. 14	Mar. 22	Complete	Team	Major event.
Design and implement the levels.	Mar. 24	Apr. 2	Complete	Togola	Easy to do once the level editor was completed.
Design and implement the learning section.	Mar. 28	Apr. 6	Complete	Fong	Reused the layout of the level selection page.
Finish coding the menus and other user interfaces for each section.	Apr. 4	Apr. 7	Complete	Fong	Started to really look like an application at this point.
Merge all the sections together.	Apr. 5	Apr. 10	Complete	Team	The merge created new unexpected problems.
Code the saving/loading system.	Apr. 7	Apr. 7	Complete	Togola	It worked immediately, to our surprise.
Code additional features.	Apr. 11	Apr. 21	Complete	Team	This was also done throughout the development phase.
Test the application.	Apr. 14	Apr. 28	Complete	Team	Did not spend enough time on this task.

Manuals

Installation Instructions

The installation procedure for Virtual Optics is fairly straightforward. The application will work on any computer system with the latest Java Runtime Environment installed (JRE 7). The JRE is available and free to download at:

http://www.oracle.com/technetwork/java/javase/downloads/java-se-jre-7-download-432155.html

Follow the indications given on the web site. Once this is installed:

1. Insert the application CD.
2. Copy the Virtual Optics folder from the CD to a desired location on your machine.
3. Now the CD can be ejected.
4. Double-click on the Virtual Optics folder, and then double-click on the VirtualOptics executable JAR file, or its shortcut, to launch the application.

Please do not, in any case, change the location of the files and directories inside the Virtual Optics folder. Otherwise, the application may not work properly. The only file that may be moved is the VirtualOptics – Shortcut file, to allow easy access from the desktop for instance. Finally, to run the application directly from the CD, simply execute step 4.

User Documentation

This segment will explain the interaction between the user and the program, i.e. how the program will react after an input from a user.

Main Menu

The will be four elements that can interact with the user, they will be represented by four phrases: Storyline, Learning Center, Laboratory, and Quit. When pressed and released by the user, the icon Storyline will register the action and create a new panel from the class LevelSelection and then close the MainMenu panel. When the same action is performed on Learning Center or Laboratory, they will open their respective panel, LearningCenter and Lab, and then the MainMenu panel will close. At last, if the user presses the Quit icon, the program will terminate.

Level Selection

When the Storyline icon is pressed and the LevelSelection panel is opened, there will be 8 icons available to the user. One of them will be the back button that allows the user to go back to the main menu if he presses it, 2 of them will be used as “Next page” or “Previous page” to allow the user to scroll through more levels. 5 of the icons will represent levels which the user can choose to play on, after being pressed and released, the program will open a Level panel that will get its level information from the file that represents the level, and then elements represented by geometric shapes that represent GameComponents will be painted in the panel. The panel LevelSelection will close simultaneously.

Storyline

The user can choose to interact with the GameComponents on the panel if allowed since there are elements that are fixed. All the GameComponents that the user can or cannot interact with will be present on the screen, according to the file where the level’s data is saved in.

These are the commands the user can perform to interact with the GameComponents, it is to be noted that not all these commands will be available to all GameComponents in every level (please refer to the General reference section).

There will also be an icon called GameMenu on the top left corner of the Level panel, it will be explained later since the same icon is present in the Lab panel.

The presence of a light source (present by default when in Storyline mode) will cause the program to compute the trajectory of the ray according to the algorithms. If the path of the ray intersects with the target icon, all events will be stopped (elements painted will remain) and a message dialog containing the phrase “Level passed!” will appear on top of the panel which signifies the player has successfully completed the level. Virtual Optics will then save the user’s progress into the progress file. Then the next level’s information will be retrieved and all the elements from the previous level will be replaced by the newer GameComponents. The loop will repeat until the user has completed all the available levels.

Laboratory

When pressed by the user at the main menu, the Lab panel will open on top of the MainMenu panel and the MainMenu will be closed. The way GameComponents interact with the user will be the same as in the Storyline, but there are some commands that are only available in Lab mode and not in Level mode (please refer to the General reference section).

There will also be a ScrollPanel, which is not present in the Level mode, containing availableComponents to the user represented by ComponentLabels. When one of them is pressed by the user, a new GameComponent represented by that JLabel will appear on the Lab panel. Furthermore, the number of elements available to the user from the ScrollPanel is unlimited and the amount of activeComponents present in the Lab panel is also unlimited, although as the number of GameComponents increase, the frame rate may drop according to the performance of the user’s computer’s CPU because of the amount of calculations performed to display the path of the light ray(s).

The user will also have the option to add multiple light sources, even though the target GameComponent will be available to the user, it will not display a clear level message since the Laboratory part of the application cannot be completed. The GameMenu will also be present in the top left corner of this game mode.

Game Menu

The GameMenu that is displayed for both the Storyline and Laboratory modes are very similar as they have the same icon. When pressed and released, the GameMenu panel will be opened at the center, on top of the current panel but will only occupy around 10% of the screen. If the user chooses the close the GameMenu panel, he will have to position his cursor outside of the panel and click his left mouse button.

Common to both game modes, the GameMenu will have a Home icon available to the user, if clicked in the Storyline mode, there will be an alert message that tells the user that he will lose his current progress if he chooses to leave, if he decides to continue, the MainMenu panel will open on top of the Level panel and the later will close. When clicked in the Laboratory mode, the user will be asked if he desires to save his progress before quitting (the save option will be explained later). There is also the reset button in Level (or ClearAll button in Lab), when clicked in the Storyline mode, the elements of the panel will be erased and new OpticalElements will be repainted on the panel according to the level the user is currently at. When click in the Laboratory mode, the user will be asked if he desires to save his progress before quitting (again, the save option will be explained later). There will also be a Controls icon, when pressed, an information panel will appear showing all the available control commands that the user can perform to interact with the GameComponents present in Level or Lab panel, or with the panel themselves.

The icon Levelselection will only be available for the Storyline mode. By clicking on this icon, there will be an alert message that tells the user that he will lose his current progress if he chooses to leave, if he decides to continue, the LevelSelection panel will open on top of the Level panel, the later will then close.

The icon Save and Open will only be available for the Laboratory mode. By clicking on the Save icon the user will be able to save his progress in the Lab mode in the form of an “.op” file through a JFileChooser. By pressing the Load icon, the user will be asked if he desires to save his progress before quitting, if he chooses to continue, the Load option will allow the user to open his previous GameComponents arrangements saved through the save option.

Learning Center

The Learning Center panel's layout will be very similar with the LevelSelection panel's layout because LearningCenter extends LevelSelection. The LearningCenter will have 8 icons that represent buttons, one of them will be the back button that allows the user to return to the MainMenu, two of them will be used to scroll to another page to allow the user to see additional information icons, if there are more than 5 information boxes available. The information icons will be represented by the 5 remaining icons.

When clicked on one of them, the InfoBox panel will appear on top of the LearningCenter panel in a similar fashion as the InGameMenu. There will be two elements inside the InfoBox panel: Picture, and Information, all of the information displayed will be retrieved by loading the information text file respective to the label the user has pressed.

The information pictures that are in relation with the topic of the opened info box will be displayed in the space allocated to them. It is the same for the Information text. When the user desires to leave the current InfoBox panel, he will have to click outside the panel, the InfoBox panel will then close and LearningCenter will be displayed. The user then will have the choice to view another piece of information, go to another page, or leave the LearningCenter.

General references

Commands available to both Level and Lab sections:

- Double left-click to select a component
- Drag a component to move it around
- Mouse wheel to rotate components
- Right click marker to reach component out of sight
- Press Enter when a ray is selected to turn it on/off
- Left/Right arrow keys to rotate components with bigger angle intervals
- Up/Down arrow keys to resize selected components
- Press D to change the convergence of a curved mirror

While Shift key is down:

- Up/Down arrow keys to set radius of selected lens or curved mirror
- Drag the lower right corner of refractive zone to resize it

While CTRL key is down:

- Roll the mouse wheel to zoom in/out temporarily
- Drag anywhere on the panel to move the view

While alt key is down:

- Drag the mouse to select multiple components at once

Commands available exclusively to Lab section:

- Backspace to delete selected component
- Press M to prevent selected components from being moved

Tutorials

The first five levels in the Storyline will serve as a tutorial for a new user as it offers Tips that informs the user about basic functions of the game, as well as clues on how to clear the current level.

Conclusion

Overall, the project is a success because we reached the initial objective of creating a java application that simulates the laws of optics to help students in their first optics course. In our opinion, any application can always be improved and Virtual Optics is not an exception to this rule. Yet, all things considered, the product is complete although a few features were not implemented. This is compensated by the multiple new features that were added as the project evolved. The thing that lacked the most during the development was testing. As mentioned previously, we had no formal method of ensuring the quality of the implementation. We had to trust our good judgement. In a future project, testing practices will be prioritized.

Developing an application of this scale, from the very beginning and all the way to the end, was a great learning experience for us. We learned new things about Java, but mostly about how to be better developers. We learned, among other things, how important version control is, how to choose variable types carefully to achieve the desired result, how to work with others on the same code and how to communicate effectively. At some moments in the semester, we were facing more challenging and unexpected problems in the implementation. It is only a few weeks after choosing our project topic that we realized that we were attempting to implement a much simpler version of something called “ray tracing”. This is a highly specialized field in computer graphics that deals with optical simulations and lighting. It is therefore understandable that we had a bit of trouble because it is difficult to program. Our final product is not perfect, but we think that it is a good demonstration of our interdisciplinary knowledge of Java programming, optics and general mathematics.