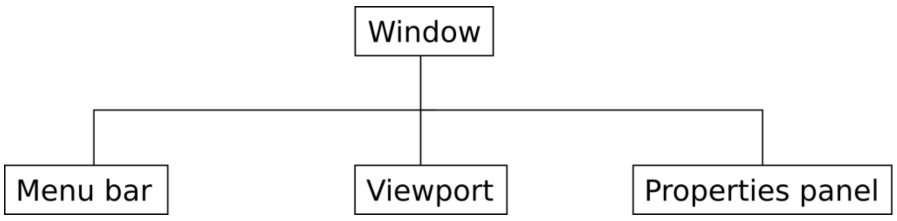
2 Design

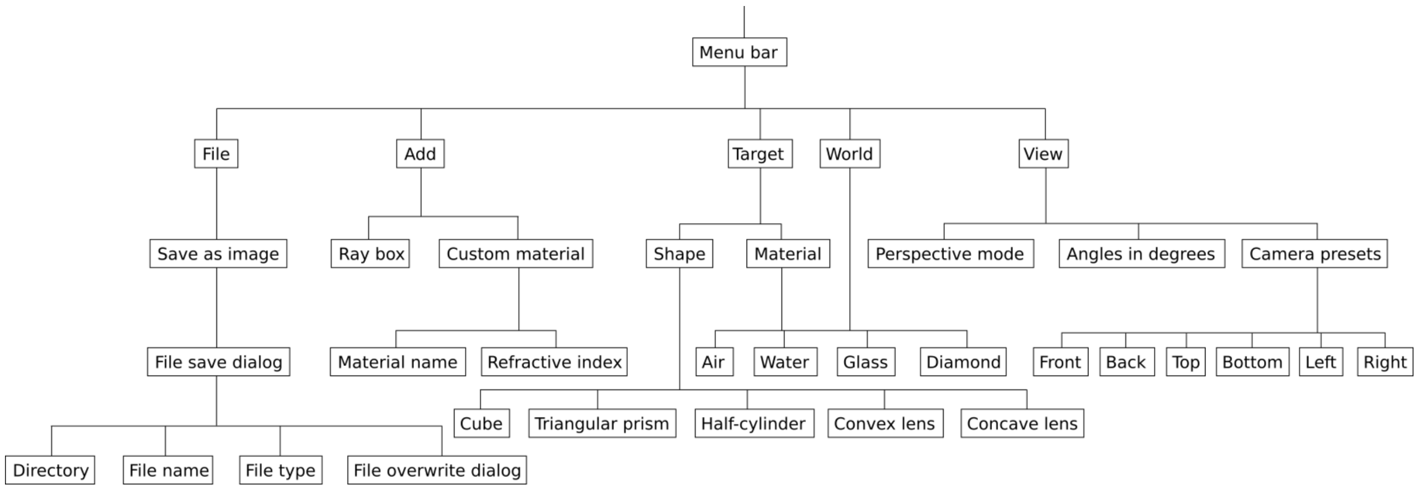
2.1 Overall System Design

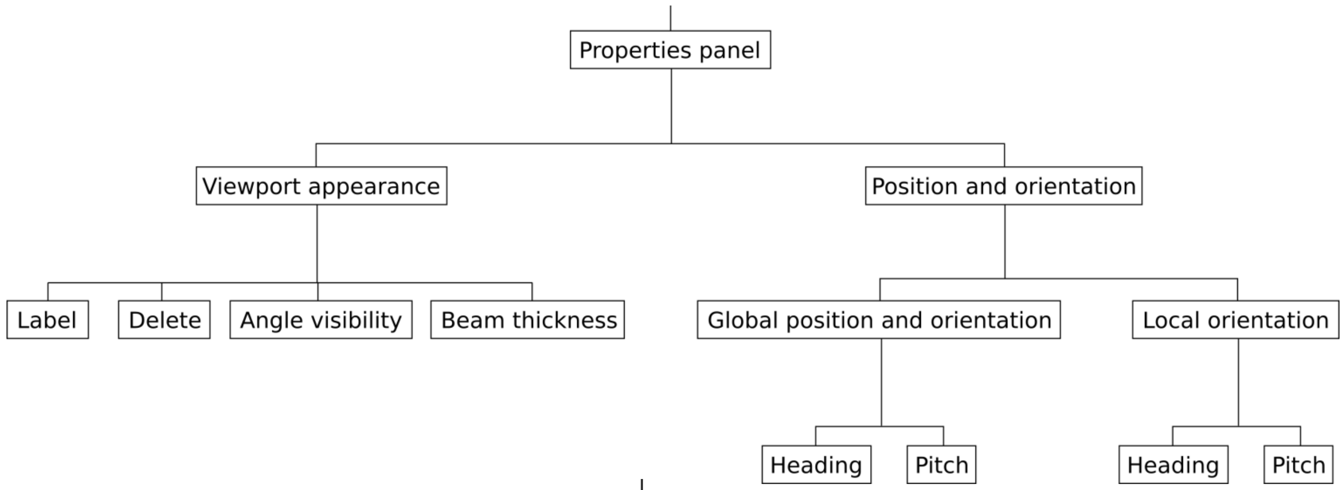
Input	**Processes**
Physics parameters: ray box global heading, ray box global pitch, ray box local heading, ray box local pitch, target material, world material, custom absolute refractive index, target shape User preferences: custom material name, angular units, ray box label, whether angles are visible for a ray box, light beam thickness, file path of output image, file type of output image 3-D view: user location/orientation, perspective mode on/off	Add and delete a ray box (with a light beam) Add a custom material Save an image of the viewport Select and deselect a ray box Modify selected ray box position and orientation Change target shape Change target material Change world material Toggle perspective mode Toggle angular units (degrees or radians) Orbit camera/user around the target shape Zoom (alter the distance between the camera/user and the target shape) Set camera/user position and orientation to a preset Change the label of a ray box Toggle angle visibility for a ray box Change the thickness of a beam of light
Storage	**Output**
Files containing images of the viewport stored in a user-defined area of secondary storage.	Physics parameters: angles of incidence, angles of refraction, angles of reflection, ray box global and local heading and pitch, all shapes, all material names and their refractive indices, selected target material, selected target shape, selected world material Visualisation: 3-D objects (ray boxes, light beams and target), angles in appropriate positions, ray box labels, outline around selected ray box Miscellaneous: angle visibility, light beam thickness, files and directories, available image file formats, perspective mode setting, angular units, preset camera positions and orientations

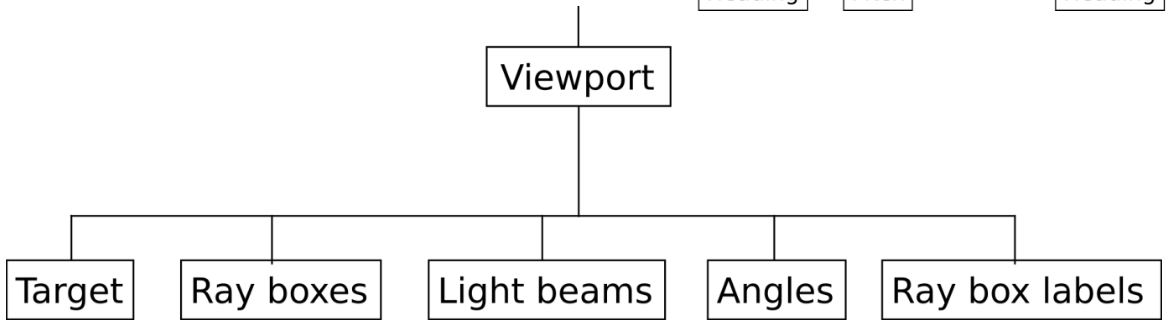
2.2 Modular System Structure

2.2.1 User Interface Structure

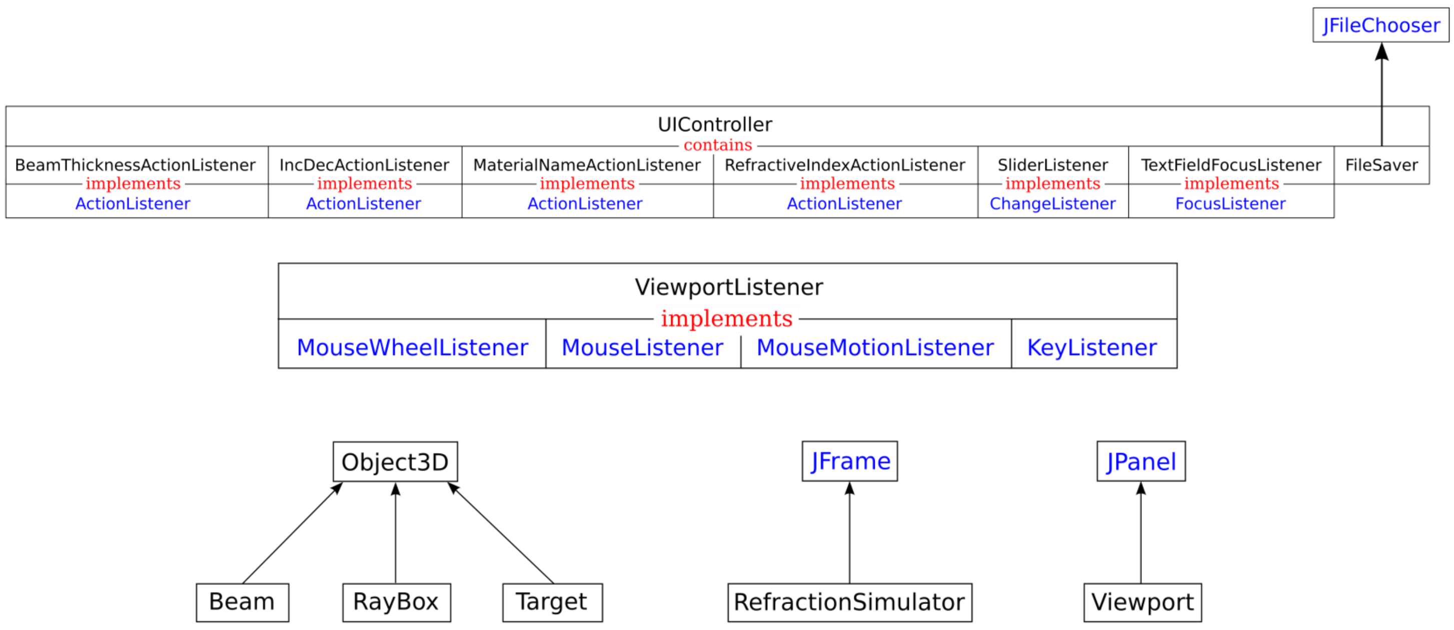








2.2.2 Class Relationships



Key:	class I must develop	pre-defined class or interface	type of relationship	inheritance ⟶

2.3 Descriptions of Classes

- **RefractionSimulator**: a window of the application which starts the rest of the application.
- **UIController**: contains methods for generating and regenerating parts of the user interface and returning them so that the window can display them.
    - **BeamThicknessActionListener**: validates the information entered into a beam thickness text field, rounds it to an integer between 1 and 10 inclusive or the last valid entry and updates the geometry of the beam object.
    - **IncDecActionListener**: enters an appropriate new beam thickness when the user clicks one of the buttons to increment or decrement the beam thickness.
    - **MaterialNameActionListener**: trims whitespace from the beginning and end of the string entered into a custom material name field, restricts the result to 30 characters (and trims again) and replaces a blank name with “Custom material”.
    - **RefractiveIndexActionListener**: validates the information entered into a new material refractive index field and restricts it to a real number between 1 and 100 inclusive.
    - **SliderListener**: when a slider is dragged, this updates other sliders and the position and orientation of the selected ray box.
    - **TextFieldFocusListener**: triggers a text field’s action listener when it loses focus (typically when the user clicks on something else) so that the input can be validated and altered appropriately.
    - **FileSaver**: allows dialog boxes to be created which let the user save an image to a secondary storage device.
- **Viewport**: a user interface component which displays images of 3-D space in real-time as well as handling all of the 3-D objects and updating other areas of the user interface when called to do so by the ViewportListener object.
- **ViewportListener**: responds to both mouse and keyboard events that involve its Viewport object and often call the Viewport object’s methods in order to update 3-D space or the user interface.
- **Object3D**: allows for the creation of generic three-dimensional objects and provides methods for manipulating these objects
- **Target**: 3-D objects which are more specifically target objects, meaning they have a material and must be one of the primitive shapes.
- **RayBox**: 3-D objects which are more specifically ray boxes, meaning they are cubes with centre five units from the world’s origin and each has its own Beam object.
- **Beam**: 3-D objects which represent beams of light and as such their geometry is dependent on their origin’s position, their orientation, the shape of the target object and the materials of the world and target object.
- **Mesh**: represents the geometry (vertices and triangular faces) of a 3-D object as well as its arbitrarily orientated bounding box.
- **Matrix**: represent transformations in n-dimensional space such as projections, rotations and enlargements.
- **Vector**: represent positions or displacements in n-dimensional space for operations that involve matrices.

- **EulerTriple**: represent orientations and angular displacements in 3-D space using Euler angle triples (heading, pitch and bank). Methods are provided for adding angular displacements and converting the EulerTriple to a Matrix object.
- **Ray**: represent lines in three dimensions each with a starting point and a direction. These are used in calculating the path of a beam of light**.**
- **Edge2D**: represent edges (line segments) in two dimensions. These are used in rendering faces and detecting if a ray passes through a particular face.
- **Math2**: contains static methods (accessible without creating an instance of the class) which perform general-purpose mathematical operations not already provided by the Math class.

2.4 Validation Required

Input	**Data type and format**	**Validation**	**Action taken when input is invalid**	**Valid examples**
Ray box label	String	None; there is no need to restrict the characters or length of string the user can input.	N/A	“” (blank) “Ray box” “\/;^_)”
Light beam thickness	Integer	The input must be a whole number between 1 and 10 inclusive.	If the input is not a number, the field is reverted to its previous value and the beam thickness unchanged. If the input has a fractional part it will be rounded to the nearest integer. An input below 1 will be changed to 1 and an input above 10 will be changed to 10.	1 5 10
Global and local heading angles	Floating point number	None; values in degrees must be in the interval $[-180, 180]$ and values in radians must be in the interval $[-\pi, \pi]$, but a slider will be used to input the data, preventing inputs outside the allowed range.	N/A	$-12°$ $90.3333°$ $\frac{\pi}{3}$
Global and local pitch angles	Floating point number	None; values in degrees must be in the interval $[-90, 90]$ and values in radians must be in the interval $[-\pi/2, \pi/2]$, but a slider will be used to input the data, preventing inputs outside the allowed range.	N/A	$-\frac{\pi}{2}$ 0 $90°$
Save directory	String	Must be an existing directory that the user has permission to write to.	The user will select a directory with a file browser so that the directory must exist. If the user doesn’t have permission to write here then a dialog box will be displayed to alert the user that they cannot save the file here.	Depends on the system.

New file name	String	The filename must not include any of the following characters: ‘/’, ‘\’, ‘<’, ‘>’, ‘?’, ‘:’, ‘|’, ‘”’, ‘%’ or ‘*’. Also, check if a file with this name already exists in this directory.	If the filename contains any of the disallowed characters, a dialog box will be displayed to tell the user that they must change the filename. If a file already exists with that name in this directory, a dialog will ask the user if they would like to overwrite the existing file or cancel saving.	“file #1” “image.png” “.a.b.exe”
New file type	String	Take everything after the final ‘.’ in the filename as the file extension from which the file type can be determined. If there is no file extension, determine the file type from the file type dropdown selection. Only gif, jpg and png file types are supported.	If the file extension is not for one of the supported file types, use the selection from the file type dropdown.	“gif” “jpg” “png”
Custom material name	String	The name must not have whitespace at the beginning or end and must not be blank or longer than 30 characters	Whitespace at the beginning and end will be removed and only the first 30 characters from the result will be accepted. If the result is blank it will be changed to “Custom material”.	“_;-=*\” “Material #1” “5”
Custom material refractive index	Floating point number	Must be between 1 and 100 inclusive	If the input is not a number, the field will be reverted to its previous value (1 will be the starting value). If the number is less than 1 it will be changed to 1 and if it is greater than 100 it will be changed to 100.	100.000000 5 9.9999

2.5 File Storage

The source code files and imported libraries will be exported to a single executable jar file less than 200KB. The jar file can then be transferred to the user via the Internet, USB memory stick or CD-ROM; the small file size should mean that speed of transfer is not an issue. If the machine has Java runtime environment (JRE) installed, then the program can be run from a file explorer (such as by double-clicking) or even a command line.

An image of the viewport can be saved to the anywhere on the hard disk or other connected secondary storage device to which the user has permission to write data. These images can be in GIF, JPEG or PNG format; – GIFs are the smallest of these and should be suitable for these diagrams with a small colour palette, but other software applications may prefer JPEG or PNG. Regardless of the file format, I would expect all images to be less than 100 kilobytes. This means that the user can save images to hard disks, memory sticks, DVDs and CDs.

2.6 Example Algorithms

2.6.1 Rendering 3-D Objects

This code maps the vertices of each object to screen co-ordinates, draws faces pointing towards the camera (or faces that are part of an object which is not opaque) into a frame buffer. After the selected ray box's outline is added to the frame buffer, the content of the viewport is set to the image in the frame buffer. Each face is a list of 3 integers which are subscripts for a list of vertices; in this way faces can share vertices so that they don't need to be mapped twice.

```
clear frameBuffer, depthBuffer and objectBuffer
get camera object
for each object in the scene
      if object is in view of the camera
            get objectColor
            for each face in object
                  for each vertex in face
                        if vertex not already mapped to screen space
                              map vertex to screen space
                  calculate screenSpaceNormal from vertices in screen space
                  if screenSpaceNormal is away from the camera or
objectColor is not opaque
                        if face is in view of the camera
                              calculate faceColor
                              rasterise face
get selectedObjID
if selectedObjID refers to a ray box
      set outlineColor to bright orange
      for x from 0 to width of frameBuffer - 1
            for y from 0 to height of frameBuffer – 1
                  set frameBuffer[x][y] to outlineColor when
objectBuffer[x][y] changes between selectedObjID and some other value
      for y from 0 to height of frameBuffer – 1
            for x from 0 to width of frameBuffer – 1
                  set frameBuffer[x][y] to outlineColor when
objectBuffer[x][y] changes between selectedObjID and some other value
draw frameBuffer to viewport
```

2.6.2 Generating a Sphere Mesh

This code generates the vertices and faces that define an approximation of a sphere. The sphere is divided into horizontal loops (rings) of vertices and vertical loops (segments – similar to lines of longitude)of vertices.

```
get numOfSegments
get numOfRings
Set the first vertex (first ring) to a vector with elements [0, -1, 0]
for each segment
      create face between first vertex and two consecutive vertices on the
second ring
pitchIncrement ← pi / numOfRings
pitch ← pitchIncrement – pi / 2
headingIncrement ← 2 * pi / numOfSegments
heading ← -pi
for ring from 0 to numOfRings – 2
      y ← sin(pitch)
      radius ← cos(pitch)
      for each segment
            x ← radius * cos(heading)
            z ← radius * sin(heading)
            set next vertex to a vector with elements [x, y, z]
            heading ← heading + headingIncrement
      if ring is not numOfRings – 2
            for each segment
                  create two faces between ring and ring + 1 vertices
      pitch ← pitch + pitchIncrement
set the last vertex (last ring) to a vector with elements [0, 1, 0]
for each segment
      create face between the last vertex and two consecutive vertices on
the second to last ring
```

2.6.3 Calculating the New Direction of a Beam

From the previous direction of the beam, the normal of the face the beam is hitting and the refractive index of the target material relative to the world material, the new direction of the beam is calculated by rotating the normal and previous direction into the plane y = 0 and deciding whether refraction or total internal reflection occurs. The rotation allows the problem to be expressed in two dimensions so that refraction and reflection are simpler. After refraction or reflection, the opposite rotation transforms the vector back from the plane y = 0 to world space and the result is returned.

```
get direction
get faceNormal
get targetIndex
if targetIndex > 1
      criticalAngle ← arcsin(1 / targetIndex)
else
      criticalAngle ← arcsin(targetIndex)
set cosAngle to the dot product of direction an faceNormal
if cosAngle = 0
      return direction
else
      set rotation to a 3 by 3 matrix
      set the first column of rotation to –faceNormal
      set the second column of rotation to the normalised result of -
direction crossed with faceNormal
      set the third column of rotation to the negative of the second
crossed with the first and normalised
      invert rotation
      multiply direction by rotation
      angle ← arcos(cosAngle)
      if cosAngle < 0
            if targetIndex > 1
                  refract direction using targetIndex
            else
                  if angle >= criticalAngle
                        reflect direction by negating the first element
                  else
                        refract direction using targetIndex
      else
            if targetIndex > 1
                  if angle >= criticalAngle
                        reflect direction by negating the first element
                  else
                        refract direction using 1 / targetIndex
            else
                  refract direction using 1 / targetIndex
      invert rotation
      multiply direction by rotation
      return direction
```

2.6.4 Refracting a Vector in the Plane y = 0

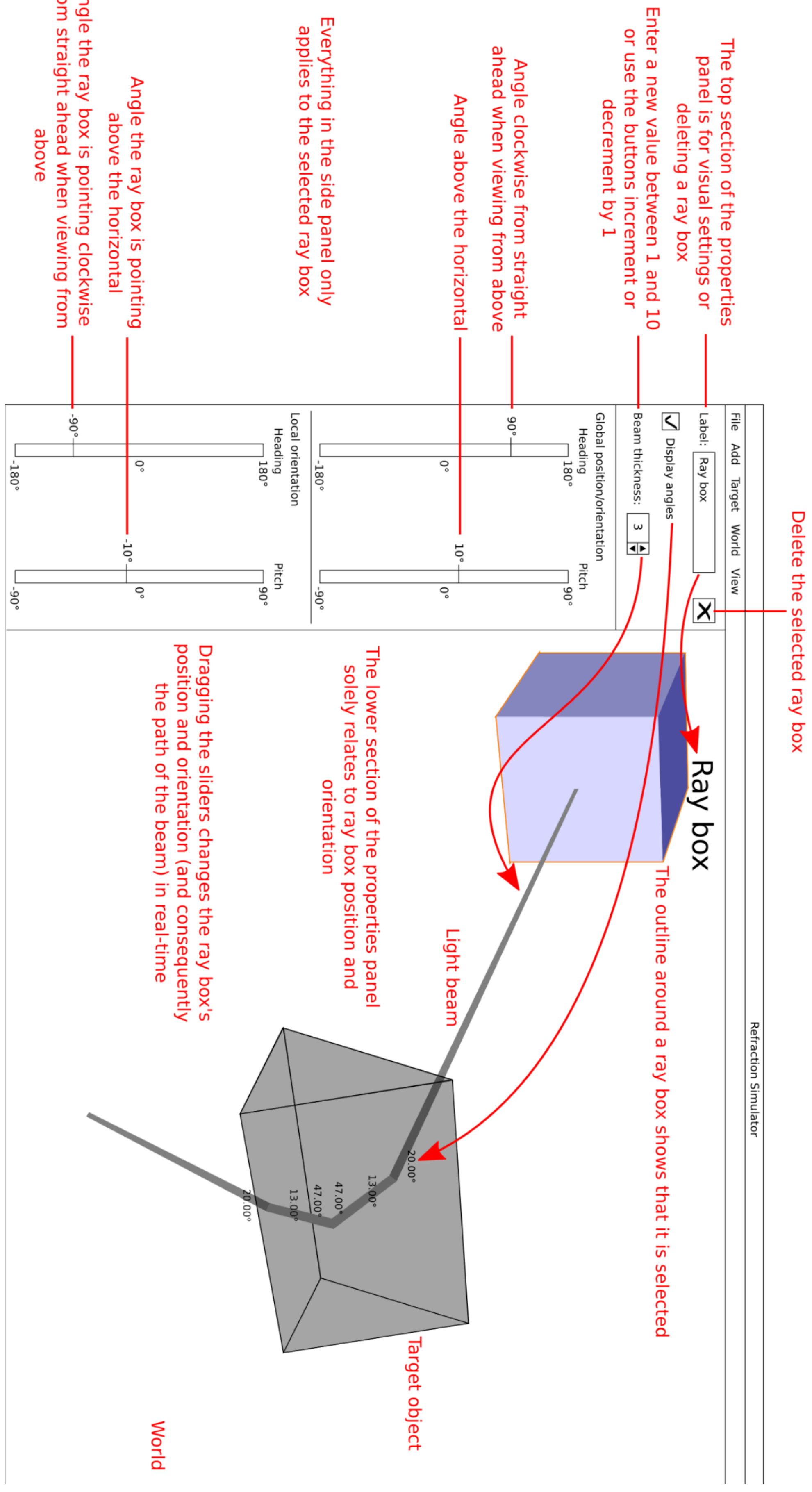
After the previous direction of a light beam and the normal of the face it hits have been rotated into the plane y = 0, this method will be called if the beam is transitioning to a more dense medium or to a less dense medium at an angle to the normal less than the critical angle. This method then returns the new direction of the beam within the plane y = 0, calculated using Snell’s law.

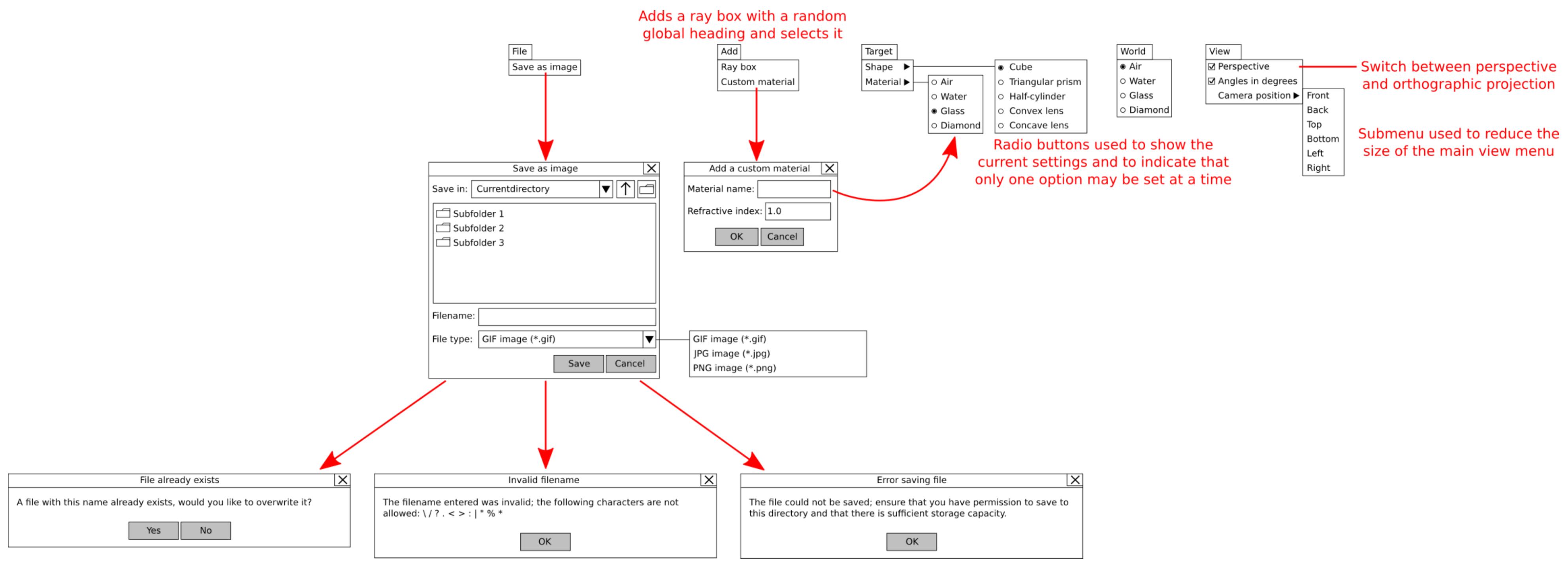
```
get previousDirection
get refractiveIndex
sinR ← z component of previousDirection / refractiveIndex
cosR ← cos(arcsin(sinR))
if x component of previousDirection
      set x component of newDirection to –cosR
else
      set x component of newDirection to cosR
set z component of newDirection to sinR
return newDirection
```

2.7 Class Definitions

See “Class Properties and Methods” in System Maintenance.

2.8 User Interface Design


Refraction Simulator
File Add Target World View
Label: Ray box
Display angles
Beam thickness: 3
Global position/orientation
Heading
180°
90°
0°
-180°
Pitch
90°
10°
0°
-90°
Local orientation
Heading
180°
0°
-90°
-180°
Pitch
90°
0°
-10°
-90°
Ray box
Light beam
Target object
World
20.00°
13.00°
47.00°
47.00°
13.00°
20.00°
Delete the selected ray box
The top section of the properties panel is for visual settings or deleting a ray box
Enter a new value between 1 and 10 or use the buttons increment or decrement by 1
Angle clockwise from straight ahead when viewing from above
Angle above the horizontal
Everything in the side panel only applies to the selected ray box
Angle the ray box is pointing above the horizontal
Angle the ray box is pointing clockwise from straight ahead when viewing from above
The outline around a ray box shows that it is selected
The lower section of the properties panel solely relates to ray box position and orientation
Dragging the sliders changes the ray box's position and orientation (and consequently the path of the beam) in real-time



A file with this name already exists, would you like to overwrite it?

The filename entered was invalid; the following characters are not allowed: \ / ? . < > : | " % *

The file could not be saved; ensure that you have permission to save to this directory and that there is sufficient storage capacity.

2.9 Overall Test Strategy

2.9.1 Testing of Methods

During development, I will test methods individually before integrating them into the rest of the system. This will mainly be using a testing class with a static main method which calls the relevant method with test data and outputs the results to the console. I will then compare the output to the expected results and if there is any difference I shall look for a pattern in the differences so as to understand the mistakes being made by the program and modify the method's code accordingly. If I am not able to understand the differences between the expected and observed results I shall modify the method's code to output intermediate values which I can compare to their expected values and identify where in the code the differences are introduced. By breaking down the problem in this way I should reach a point where I am able to identify the solution and modify the code and rerun the test until all mistakes in the method have been rectified. I will repeat this process using different inputs in order to test as many paths through the code as possible (white-box testing).

2.9.2 Testing of Classes

After implementing a class, I must test that the methods integrate with one another using test objects of that class and calling multiple public methods each object. Again, the results will be output to the console for me to compare against the values I expect. I expect that any erroneous output will usually be due to a certain method failing to update instance variables correctly. Similarly to the testing of individual methods, I shall insert code to output data after each call to a method in order to identify the methods at fault (although one such method could be calling another method where the problem actually resides). Outputting data from within such methods and analysing the errors will locate the source of the mistake more precisely and from there I should be able to correct the code and rerun the test to ensure my modifications work. White-box testing would be impractical for testing classes because there will be too many different routes through the program, but I will perform several black-box tests with different inputs.

2.9.3 Integration Testing

Many classes and methods within classes will depend on other classes such as by using objects of another class or static methods of another class. This means that some classes must be completed first. The testing of these classes upon completion will mean that any problems that the program has in performing a certain function will be due to how a method uses a completed class. The result is that integration testing is included in the testing of methods, but occasionally it may be found that a completed classes needs to be extended to carry out some task so that the new method can perform its function. In this case, new methods (and possibly new instance variables) will be added to the completed class which then must be tested individually and the class as a whole must be tested once again.

2.9.4 Interface Testing

As well as writing code to test specific methods and classes and compare textual output with expected output, I will need to test the application from an end user's perspective whereby I use the graphical user interface to input data or start processes and compare visual and textual output to expected output. This will be a type of black-box integration testing.