Path Tracer

Generated by Doxygen 1.9.8

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 AABB Struct Reference

Struct representing an Axis Aligned Bounding Box.

```
#include <bvh.hpp>
```

Public Attributes

- Vector **min**
- Vector **max**

4.1.1 Detailed Description

Struct representing an Axis Aligned Bounding Box.

The documentation for this struct was generated from the following file:

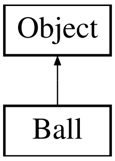
- objects/bvh.hpp

4.2 Ball Class Reference

Representation of a mathematical ball object in the scene.

```
#include <ball.hpp>
```

Inheritance diagram for Ball:



Public Member Functions

- **Ball** (Vector position, float radius, std::shared_ptr< Material > material)
- void collision (Ray &ray, Hit &rayHit, float &smallestDistance)

  Calculate whether a given ray collides with the ball.
- float **getRadius** () const
- void printInfo (std::ostream &out) const

  Print ball info to the desired output stream.

Public Member Functions inherited from Object

- **Object** (Vector position, std::shared_ptr< Material > material)
- Vector **getPosition** () const
- std::shared_ptr< Material > **getMaterial** () const

4.2.1 Detailed Description

Representation of a mathematical ball object in the scene.

4.2.2 Member Function Documentation

4.2.2.1 collision()

```
void Ball::collision (
             Ray & ray,
             Hit & rayHit,
             float & smallestDistance )  [inline], [virtual]
```

Calculate whether a given ray collides with the ball.

If the ray collides with the ball and the collision is closer than the current smallest distance, the "rayHit" data structure will be updated according to the collision.

Parameters

ray	ray whose collision will be checked
rayHit	address of a Hit data structure
smallestDistance	

Implements Object.

4.2.2.2 printInfo()

```
void Ball::printInfo (
             std::ostream & out ) const  [inline], [virtual]
```

Print ball info to the desired output stream.

Parameters

out	output stream

Returns

std::ostream& the output stream

Implements Object.

The documentation for this class was generated from the following file:

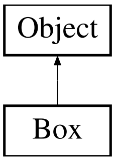
- objects/ball.hpp

4.3 Box Class Reference

Representation of a box (rectangular cuboid) object in the scene.

```
#include <box.hpp>
```

Inheritance diagram for Box:



Public Member Functions

- **Box** (Vector position, float width, float height, float depth, std::shared_ptr< Material > material)
- void collision (Ray &ray, Hit &rayHit, float &smallestDistance)

  Calculate whether a given ray collides with the box.
- float **getWidth** () const
- float **getHeight** () const
- float **getDepth** () const
- void rotate (float angle, Vector axis)

  Rotates the box around a given axis.
- void printInfo (std::ostream &out) const

  Print box info to the desired output stream.

Public Member Functions inherited from Object

- **Object** (Vector position, std::shared_ptr< Material > material)
- Vector **getPosition** () const
- std::shared_ptr< Material > **getMaterial** () const

4.3.1 Detailed Description

Representation of a box (rectangular cuboid) object in the scene.

The position represent the geometric center of the box.

4.3.2 Member Function Documentation

4.3.2.1 collision()

```
void Box::collision (
             Ray & ray,
             Hit & rayHit,
             float & smallestDistance )  [inline], [virtual]
```

Calculate whether a given ray collides with the box.

If the ray collides with the box and the collision is closer than the current smallest distance, the "rayHit" data structure will be updated according to the collision.

Parameters

ray	ray whose collision will be checked
rayHit	address of a Hit data structure
smallestDistance	

Implements Object.

4.3.2.2 printInfo()

```
void Box::printInfo (
             std::ostream & out ) const  [inline], [virtual]
```

Print box info to the desired output stream.

Parameters

out	output stream

Returns

std::ostream& the output stream

Implements Object.

4.3.2.3 rotate()

```
void Box::rotate (
             float angle,
             Vector axis )  [inline]
```

Rotates the box around a given axis.

The center of rotation is the box's center of mass.

Parameters

angle	rotation in radians
axis	axis of rotation (has to be normalized)

The documentation for this class was generated from the following file:

- objects/box.hpp

4.4 Button Class Reference

Implements a class for creating buttons for a SFML window.

```
#include <button.hpp>
```

Public Member Functions

- Button (std::string button_text, sf::Color textColor, int text_size, sf::Vector2f button_size, sf::Color button_↩ color)

  Constructor for the button.
- ∼**Button** ()=default

  Destroy the Button object.
- void setPos (sf::Vector2f pos)

  Sets the position of the button in the window.
- void setColor (sf::Color color)

  Sets the color of the button.
- void setFont (sf::Font &font)

  Sets the font of the text of the button.
- void draw (sf::RenderWindow &window)

  Renders the button on the specified SFML window.
- bool onButton (sf::RenderWindow &window)

  Checks if the user's mouse is on the button.

4.4.1 Detailed Description

Implements a class for creating buttons for a SFML window.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Button()

```
Button::Button (
             std::string button_text,
             sf::Color textColor,
             int text_size,
             sf::Vector2f button_size,
             sf::Color button_color )  [inline]
```

Constructor for the button.

Parameters

button_text	The text inside the button
textColor	The color of the text inside the button
text_size	The size of the text inside the button
button_size	Size of the button
button_color	Color of the button

4.4.3 Member Function Documentation

4.4.3.1 draw()

```
void Button::draw (
             sf::RenderWindow & window )  [inline]
```

Renders the button on the specified SFML window.

Parameters

window	Window to be rendered on

4.4.3.2 onButton()

```
bool Button::onButton (
             sf::RenderWindow & window )  [inline]
```

Checks if the user's mouse is on the button.

Parameters

window	reference to the window the textbox is on

Returns

true value if mouse is on the button

false value if mouse is not on the button

4.4.3.3 setColor()

```
void Button::setColor (
             sf::Color color )  [inline]
```

Sets the color of the button.

Parameters

color	The color to be set to

4.4.3.4 setFont()

```
void Button::setFont (
             sf::Font & font )  [inline]
```

Sets the font of the text of the button.

Parameters

font	The font to be set to

4.4.3.5 setPos()

```
void Button::setPos (
             sf::Vector2f pos )  [inline]
```

Sets the position of the button in the window.

Parameters

pos	Position of the button

The documentation for this class was generated from the following file:

- interface/button.hpp

4.5 BVH Class Reference

Bounding volume hierarchy class for optimising TriangleMesh collisions. Creates a data structure that divides the triangles of the mesh in to groups of Axis Aligned Bounding Boxes (AABB). If a ray misses a box then none of the triangles in the box need to be checked for a collision.

```
#include <bvh.hpp>
```

Public Member Functions

- **BVH** ()

  Default constructor for BVH.
- BVH (std::vector< Triangle > tris)

  Construct a new BVH object.
- void UpdateNodeBounds (std::vector< Node > &bvhNodes, int nodeIdx)

  Updates the bounds of the AABB based on the triangles it contains.
- void Subdivide (std::vector< Node > &bvhNodes, int nodeIdx)

  Divides the traingles of the current node into new AABBs recursively.
- void BVHCollision (Ray &ray, Hit &rayHit, float &smallestDistance, const int nodeIdx)

  Calculate whether a given ray collides with the TriangleMesh contained in the BVH recursively.
- bool AABBCollision (AABB box, Ray ray, float smallestDistance)

Calculate whether a given ray collides with the AABB.

- int getRootNodeIdx () const

  Get the RootNodeIdx of the BVH.

- std::vector< Triangle > getTriangles () const

  Get the triangles vector.

- std::vector< Node > getNodes () const

  Get the nodes vector.

4.5.1 Detailed Description

Bounding volume hierarchy class for optimising TriangleMesh collisions. Creates a data structure that divides the triangles of the mesh in to groups of Axis Aligned Bounding Boxes (AABB). If a ray misses a box then none of the triangles in the box need to be checked for a collision.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 BVH()

```
BVH::BVH (
            std::vector< Triangle > tris )  [inline]
```

Construct a new BVH object.

Parameters

tris	vector containing all the triangles in the mesh

4.5.3 Member Function Documentation

4.5.3.1 AABBCollision()

```
bool BVH::AABBCollision (
            AABB box,
            Ray ray,
            float smallestDistance )  [inline]
```

Calculate whether a given ray collides with the AABB.

Parameters

box	the AABB for which collision is tested
ray	ray whose collision will be checked
smallestDistance	current smallest distance

Returns

true If the ray collides with the box

false If the ray does not collide with the boc

4.5.3.2 BVHCollision()

```
void BVH::BVHCollision (
            Ray & ray,
            Hit & rayHit,
            float & smallestDistance,
            const int nodeIdx )  [inline]
```

Calculate whether a given ray collides with the TriangleMesh contained in the BVH recursively.

Parameters

ray	ray whose collision will be checked
rayHit	address of a Hit data structure
smallestDistance	current smallest distance
nodeIdx	current node index

4.5.3.3 getNodes()

```
std::vector< Node > BVH::getNodes ( ) const  [inline]
```

Get the nodes vector.

Returns

std::vector<Node>

4.5.3.4 getRootNodeIdx()

```
int BVH::getRootNodeIdx ( ) const  [inline]
```

Get the RootNodeIdx of the BVH.

Returns

int

4.5.3.5 getTriangles()

```
std::vector< Triangle > BVH::getTriangles ( ) const  [inline]
```

Get the triangles vector.

Returns

std::vector<Triangle>

4.5.3.6 Subdivide()

```
void BVH::Subdivide (
             std::vector< Node > & bvhNodes,
             int nodeIdx )  [inline]
```

Divides the traingles of the current node into new AABBs recursively.

Parameters

bvhNodes	vector containing the nodes
nodeIdx	current node index

4.5.3.7 UpdateNodeBounds()

```
void BVH::UpdateNodeBounds (
             std::vector< Node > & bvhNodes,
             int nodeIdx )  [inline]
```

Updates the bounds of the AABB based on the triangles it contains.

Parameters

bvhNodes	vector containing the nodes
nodeIdx	the current node index

The documentation for this class was generated from the following file:

- objects/bvh.hpp

4.6 Camera Struct Reference

Struct representing the camera.

```
#include <types.hpp>
```

Public Attributes

- Point **position**
- Vector **lookingAt**
- Vector **direction**
- Vector **up**
- Vector **left**
- float **fov**
- float **focus_distance**
- float **DoF**

4.6.1 Detailed Description

Struct representing the camera.

The documentation for this struct was generated from the following file:

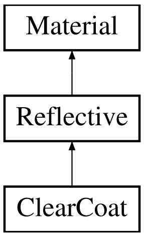
- utils/types.hpp

4.7 ClearCoat Class Reference

Representation of a clear coat material, which can be used for materials that are partially reflective and partially diffusive. Reflective color can be defined independently from diffusive color.

```
#include <material.hpp>
```

Inheritance diagram for ClearCoat:



Public Member Functions

- Color getClearCoatColor ()

  Gets the clear coat color for the material.
- float getClearCoat ()

  Gets clear coat value for the clear coat material.
- ClearCoat (Color color, std::string name, float specularity, float clearcoat, Color clearcoat_color)

  Constructor for ClearCoat class.
- void updateRay (Ray &ray, Hit &hit)

  Updates the ray according to properties of mirror material.

Public Member Functions inherited from Reflective

- Reflective (Color color, std::string name, float specularity)

  Constructor for Reflective class material.
- float getSpecularity ()

  Gets the specularity of the reflective material.
- void updateRay (Ray &ray, Hit &hit)

  Updates the ray according to properties of reflective material.

Public Member Functions inherited from Material

- Color getColor ()

  Gets the color vector for the material.
- std::string getName ()

  Gets the name of the material.
- Material (Color color, std::string name)

  Constructor for material class.
- Vector reflectionDir (Ray &ray, Hit &hit)

  Computes the direction of perfectly reflected ray.
- Vector diffuseDir (Hit &hit)

  Computes the direction of the diffuse ray.
- void updateColor (Ray &ray)

  Updates the color of the ray after interacting with the material.

Additional Inherited Members

Public Attributes inherited from Material

- RandomGenerator **rnd_**

4.7.1 Detailed Description

Representation of a clear coat material, which can be used for materials that are partially reflective and partially diffusive. Reflective color can be defined independently from diffusive color.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 ClearCoat()

```
ClearCoat::ClearCoat (
            Color color,
            std::string name,
            float specularity,
            float clearcoat,
            Color clearcoat_color )  [inline]
```

Constructor for ClearCoat class.

Parameters

color	Material color as RGB value
name	Name of the material
specularity	Specularity of the material
clearcoat	How large proportion of the rays behaves reflectively
clearcoat_color	Color of the reflected rays

4.7.3 Member Function Documentation

4.7.3.1 getClearCoat()

```
float ClearCoat::getClearCoat ( )  [inline]
```

Gets clear coat value for the clear coat material.

Returns

float representing probability of the clear coat bounce

4.7.3.2 getClearCoatColor()

```
Color ClearCoat::getClearCoatColor ( )  [inline]
```

Gets the clear coat color for the material.

Returns

Color that represents the RGB value for the clear coat bounces

4.7.3.3 updateRay()

```
void ClearCoat::updateRay (
             Ray & ray,
             Hit & hit )  [inline], [virtual]
```

Updates the ray according to properties of mirror material.

Parameters

ray	Ray that did hit the material

Implements Material.

The documentation for this class was generated from the following file:

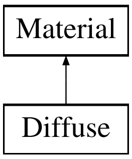
- scenery/material.hpp

4.8 Diffuse Class Reference

Representation of a diffusive material.

```
#include <material.hpp>
```

Inheritance diagram for Diffuse:



Public Member Functions

- Diffuse (Color color, std::string name, float emission_strength, Color emission_color)

  Constructor for Diffuse class.
- Color getEmColor ()

  Gets the emission color for the material.
- float getEmStrength ()

  Gets the emission strength for the material.
- bool isEmitting ()

  Tells if the material is emitting i.e. has emission strength larger than zero.
- void updateRay (Ray &ray, Hit &hit)

  Updates the ray according to properties of diffuse material.

Public Member Functions inherited from Material

- Color getColor ()

  Gets the color vector for the material.
- std::string getName ()

  Gets the name of the material.
- Material (Color color, std::string name)

  Constructor for material class.
- Vector reflectionDir (Ray &ray, Hit &hit)

  Computes the direction of perfectly reflected ray.
- Vector diffuseDir (Hit &hit)

  Computes the direction of the diffuse ray.
- void updateColor (Ray &ray)

  Updates the color of the ray after interacting with the material.

Additional Inherited Members

Public Attributes inherited from Material

- RandomGenerator **rnd_**

4.8.1 Detailed Description

Representation of a diffusive material.

4.8.2 Constructor & Destructor Documentation

4.8.2.1 Diffuse()

```
Diffuse::Diffuse (
             Color color,
             std::string name,
             float emission_strength,
             Color emission_color )  [inline]
```

Constructor for Diffuse class.

Parameters

color	Material color as RGB value
name	Name of the material
emission_strength	Emission strength of the material
emission_color	Emission color of the material

4.8.3 Member Function Documentation

4.8.3.1 getEmColor()

```
Color Diffuse::getEmColor ( )  [inline]
```

Gets the emission color for the material.

Returns

Color vector representing the RGB value for the emission color

4.8.3.2 getEmStrength()

```
float Diffuse::getEmStrength ( )  [inline]
```

Gets the emission strength for the material.

Returns

float representing the emission strength for the material

4.8.3.3 isEmitting()

```
bool Diffuse::isEmitting ( )  [inline]
```

Tells if the material is emitting i.e. has emission strength larger than zero.

Returns

true if material is emitting and zero if it is not emitting

4.8.3.4 updateRay()

```
void Diffuse::updateRay (
            Ray & ray,
            Hit & hit )  [inline], [virtual]
```

Updates the ray according to properties of diffuse material.

Parameters

ray	Ray that did hit the material

Implements Material.

The documentation for this class was generated from the following file:

- scenery/material.hpp

4.9 Environment Class Reference

An object for representing a scene environment.

```
#include <environment.hpp>
```

Public Member Functions

- **Environment** ()

  Construct a new Environment with black void.
- void setSky (Color skyColor=Color(0.2, 0.5, 1.0), Color horizonColor=Color(0.7, 0.8, 0.8), Color groundColor=Color(0.1, 0.1, 0.1))

  Set a sky to the environment.
- Light getLight (Ray &ray)

  Calculate environment light collected by a runaway ray.
- Color **getHorizonColor** ()
- Color **getGroundColor** ()
- Color **getSkyColor** ()

4.9.1 Detailed Description

An object for representing a scene environment.

Currently only solid color or sky environments are supported.

4.9.2 Member Function Documentation

4.9.2.1 getLight()

```
Light Environment::getLight (
            Ray & ray )  [inline]
```

Calculate environment light collected by a runaway ray.

The ray direction determines what the ray should "see" in the environment.

Parameters

ray	a ray whose path doesn't intersect with any objects

Returns

Light collected from the environment by the ray

4.9.2.2 setSky()

```
void Environment::setSky (
             Color skyColor = Color(0.2, 0.5, 1.0),
             Color horizonColor = Color(0.7, 0.8, 0.8),
             Color groundColor = Color(0.1, 0.1, 0.1) )  [inline]
```

Set a sky to the environment.

Parameters are optional, otherwise a default sky is created.

Parameters

skyColor	
horizonColor	
groundColor	

The documentation for this class was generated from the following file:

- scenery/environment.hpp

4.10 FileLoader Class Reference

Implements a class for reading yaml scene files.

`#include <fileloader.hpp>`

Public Member Functions

- FileLoader (std::string filepath)

  Construct a new FileLoader object.
- std::shared_ptr< Scene > loadSceneFile ()

  Loads a scene from the file specified in filepath and returns a shared pointer to this scene.
- ∼**FileLoader** ()

  Destructor for FileLoader object.
- std::string **getFilepath** ()

4.10.1 Detailed Description

Implements a class for reading yaml scene files.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 FileLoader()

```
FileLoader::FileLoader (
            std::string filepath )  [inline]
```

Construct a new FileLoader object.

Parameters

filepath	Filepath to the yaml file that contains properties of the scene

4.10.3 Member Function Documentation

4.10.3.1 loadSceneFile()

```
std::shared_ptr< Scene > FileLoader::loadSceneFile ( )  [inline]
```

Loads a scene from the file specified in filepath and returns a shared pointer to this scene.

Returns

A shared pointer to scene object

The documentation for this class was generated from the following file:

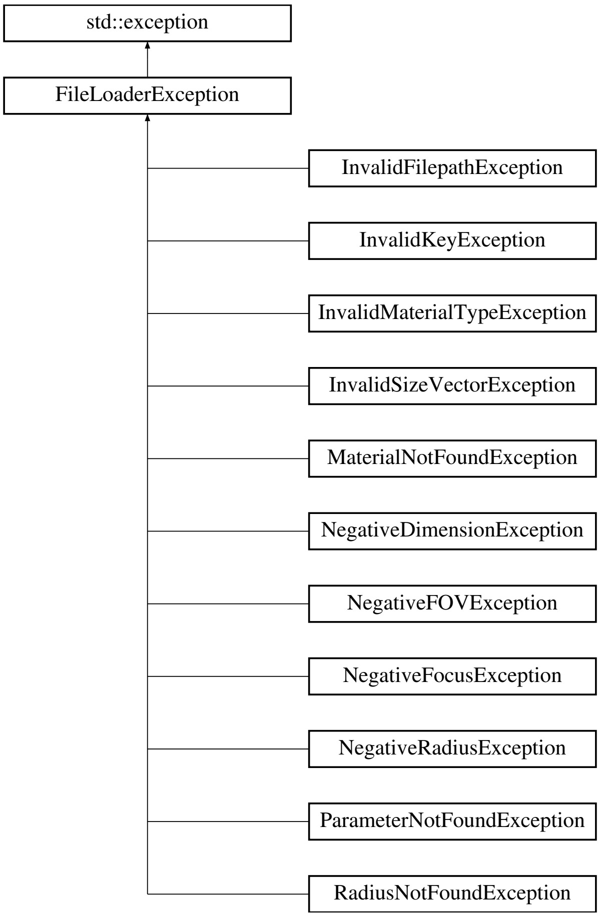
- fileloader/fileloader.hpp

4.11 FileLoaderException Class Reference

An abstract base class for FileLoader exceptions. Exceptions are designed for invalid user inputs in the YAML files.

```
#include <fileloader_ex.hpp>
```

Inheritance diagram for FileLoaderException:



Public Member Functions

- virtual const char $*$ what () const noexcept=0

  Pure virtual function overload for exception messages.

4.11.1 Detailed Description

An abstract base class for FileLoader exceptions. Exceptions are designed for invalid user inputs in the YAML files.

4.11.2 Member Function Documentation

4.11.2.1 what()

```
virtual const char * FileLoaderException::what ( ) const  [pure virtual], [noexcept]
```

Pure virtual function overload for exception messages.

Implemented in InvalidFilepathException, NegativeRadiusException, InvalidSizeVectorException, NegativeFOVException, NegativeFocusException, InvalidKeyException, MaterialNotFoundException, RadiusNotFoundException, ParameterNotFoundExcepti NegativeDimensionException, and InvalidMaterialTypeException.

The documentation for this class was generated from the following file:

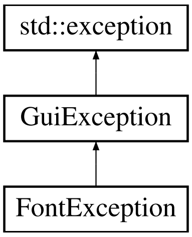
- fileloader/fileloader_ex.hpp

4.12 FontException Class Reference

Exception for not finding the given font.

`#include <gui_ex.hpp>`

Inheritance diagram for FontException:



Public Member Functions

- **FontException** (std::string fontPath)
- virtual const char ∗ what () const noexcept

4.12.1 Detailed Description

Exception for not finding the given font.

4.12.2 Member Function Documentation

4.12.2.1 what()

```
virtual const char * FontException::what ( ) const  [inline], [virtual], [noexcept]
```

Implements GuiException.

The documentation for this class was generated from the following file:

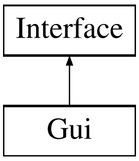
- interface/gui_ex.hpp

4.13 Gui Class Reference

Implements the Graphical user interface class.

`#include <gui.hpp>`

Inheritance diagram for Gui:



Public Member Functions

- std::shared_ptr< Scene > titleScreen ()

  Opens a titlescreen where the user can input a filepath. If the filepath contains a valid yaml file, a shared pointer to a scene constructed with this file is returned. An invalid filepath will result in an error being shown to the user.
- void openSettings (std::shared_ptr< Scene > loadedScene)

  Opens a settings menu where the user can preview and influence a loaded scene.
- void **openRender** (int resX, int resY, std::shared_ptr< Scene > loadedScene, int sampleSize, float dof, int bounces)

Public Member Functions inherited from Interface

- void createImg (std::vector< std::vector< Color > > pixels)

  Create a sf::Image from the matrix of RGB values.
- bool saveImage (const std::string &filename)

  Save the image with given filename.
- sf::Image **getImage** ()

4.13.1 Detailed Description

Implements the Graphical user interface class.

4.13.2 Member Function Documentation

4.13.2.1 openSettings()

```
void Gui::openSettings (
             std::shared_ptr< Scene > loadedScene )  [inline]
```

Opens a settings menu where the user can preview and influence a loaded scene.

The setting menu contains a "preview" button and textboxes for parameter. There are two types of parameters that the user can change. First are the parameters that influence only the final render the user can initiate with pressing enter. The other parameters, in addition to changing the final render, affect the preview image.

Parameters

loadedScene	the scene to be influenced

4.13.2.2 titleScreen()

```
std::shared_ptr< Scene > Gui::titleScreen ( )  [inline]
```

Opens a titlescreen where the user can input a filepath. If the filepath contains a valid yaml file, a shared pointer to a scene constructed with this file is returned. An invalid filepath will result in an error being shown to the user.

Returns

std::shared_ptr<Scene> The shared pointer to the scene created with the inputted yaml file

The documentation for this class was generated from the following file:

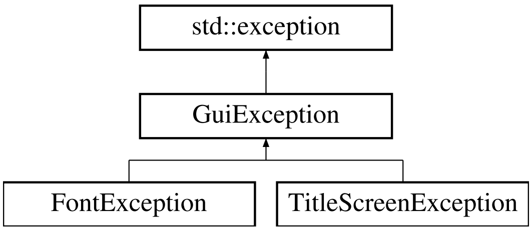
- interface/gui.hpp

4.14 GuiException Class Reference

Abstact class for a GUI exception.

```
#include <gui_ex.hpp>
```

Inheritance diagram for GuiException:



Public Member Functions

- virtual const char * **what** () const noexcept=0

4.14.1 Detailed Description

Abstact class for a GUI exception.

The documentation for this class was generated from the following file:

- interface/gui_ex.hpp

4.15 Hit Struct Reference

Struct containing information about a ray hitting an object.

```
#include <types.hpp>
```

Public Attributes

- bool **did_hit** = false
- std::shared_ptr< Material > **material**
- Vector **normal**
- Point **point**
- float **distance**

4.15.1 Detailed Description

Struct containing information about a ray hitting an object.

The documentation for this struct was generated from the following file:

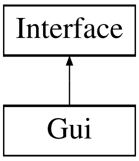
- utils/types.hpp

4.16 Interface Class Reference

Class that creates an image from a raw RGB matrix and saves the image.

```
#include <interface.hpp>
```

Inheritance diagram for Interface:



Public Member Functions

- void createImg (std::vector< std::vector< Color > > pixels)

  Create a sf::Image from the matrix of RGB values.
- bool saveImage (const std::string &filename)

  Save the image with given filename.
- sf::Image **getImage** ()

4.16.1 Detailed Description

Class that creates an image from a raw RGB matrix and saves the image.

4.16.2 Member Function Documentation

4.16.2.1 createImg()

```
void Interface::createImg (
            std::vector< std::vector< Color > > pixels )  [inline]
```

Create a sf::Image from the matrix of RGB values.

Parameters

pixels	matrix of RGB values

4.16.2.2 saveImage()

```
bool Interface::saveImage (
            const std::string & filename )  [inline]
```

Save the image with given filename.

Parameters

filename	Image filename

The documentation for this class was generated from the following file:

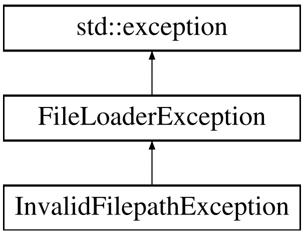
- interface/interface.hpp

4.17 InvalidFilepathException Class Reference

Representation of invalid filepath exception.

`#include <fileloader_ex.hpp>`

Inheritance diagram for InvalidFilepathException:



Public Member Functions

- InvalidFilepathException (std::string filepath)

  Constructor for InvalidFilePathException.
- virtual const char ∗ what () const noexcept

  Function that creates an exception message for invalid filepath.

4.17.1 Detailed Description

Representation of invalid filepath exception.

4.17.2 Constructor & Destructor Documentation

4.17.2.1 InvalidFilepathException()

```
InvalidFilepathException::InvalidFilepathException (
            std::string filepath )  [inline]
```

Constructor for InvalidFilePathException.

Parameters

filepath	Filepath of the given YAML file

4.17.3 Member Function Documentation

4.17.3.1 what()

```
virtual const char * InvalidFilepathException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for invalid filepath.

Returns

Pointer to the first char of the exception message.

Implements FileLoaderException.

The documentation for this class was generated from the following file:

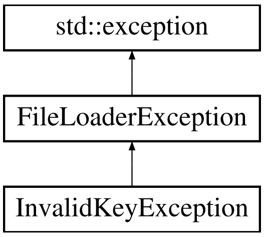
- fileloader/fileloader_ex.hpp

4.18 InvalidKeyException Class Reference

Representation of invalid key exception.

```
#include <fileloader_ex.hpp>
```

Inheritance diagram for InvalidKeyException:



Public Member Functions

- InvalidKeyException (std::string filepath, std::string key)

  Constructor for InvalidKeyException.
- virtual const char $*$ what () const noexcept

  Function that creates an exception message for invalid key value in the YAML file.

4.18.1 Detailed Description

Representation of invalid key exception.

4.18.2 Constructor & Destructor Documentation

4.18.2.1 InvalidKeyException()

```
InvalidKeyException::InvalidKeyException (
            std::string filepath,
            std::string key )  [inline]
```

Constructor for InvalidKeyException.

Parameters

filepath	Filepath of the YAML file
key	String of the key that was not found from the YAML file

4.18.3 Member Function Documentation

4.18.3.1 what()

```
virtual const char * InvalidKeyException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for invalid key value in the YAML file.

Returns

Pointer to the first char of the exception message

Implements FileLoaderException.

The documentation for this class was generated from the following file:

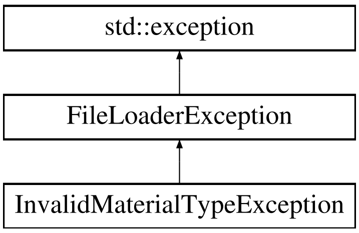
- fileloader/fileloader_ex.hpp

4.19 InvalidMaterialTypeException Class Reference

Representation of invalid material type exception.

```
#include <fileloader_ex.hpp>
```

Inheritance diagram for InvalidMaterialTypeException:



Public Member Functions

- InvalidMaterialTypeException (std::string filepath, int line)

  Constructor for InvalidMaterialTypeException.
- virtual const char ∗ what () const noexcept

  Function that creates an exception message for invalid material type in the YAML file.

4.19.1 Detailed Description

Representation of invalid material type exception.

4.19.2 Constructor & Destructor Documentation

4.19.2.1 InvalidMaterialTypeException()

```
InvalidMaterialTypeException::InvalidMaterialTypeException (
            std::string filepath,
            int line )  [inline]
```

Constructor for InvalidMaterialTypeException.

Parameters

filepath	Filepath of the YAML file
line	Line number where material is defined

4.19.3 Member Function Documentation

4.19.3.1 what()

```
virtual const char * InvalidMaterialTypeException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for invalid material type in the YAML file.

Returns

Pointer to the first char of the exception message

Implements FileLoaderException.

The documentation for this class was generated from the following file:

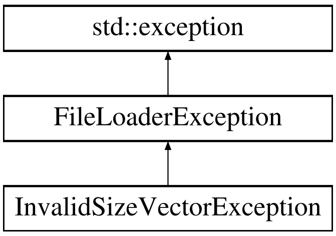
- fileloader/fileloader_ex.hpp

4.20 InvalidSizeVectorException Class Reference

Representation of invalid size vector exception.

```
#include <fileloader_ex.hpp>
```

Inheritance diagram for InvalidSizeVectorException:



Public Member Functions

- InvalidSizeVectorException (std::string filepath, size_t size, int line)

  Constructor for InvalidSizeVectorException.
- virtual const char ∗ what () const noexcept

  Function that creates an exception message for invalid size vector.

4.20.1 Detailed Description

Representation of invalid size vector exception.

4.20.2 Constructor & Destructor Documentation

4.20.2.1 InvalidSizeVectorException()

```
InvalidSizeVectorException::InvalidSizeVectorException (
              std::string filepath,
              size_t size,
              int line )  [inline]
```

Constructor for InvalidSizeVectorException.

Parameters

filepath	Filepath of the YAML file
size	Size of the invalid size vector
line	Line number where the invalid size vector is defined

4.20.3 Member Function Documentation

4.20.3.1 what()

```
virtual const char * InvalidSizeVectorException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for invalid size vector.

Returns

Pointer to the first char of the exception message

Implements FileLoaderException.

The documentation for this class was generated from the following file:

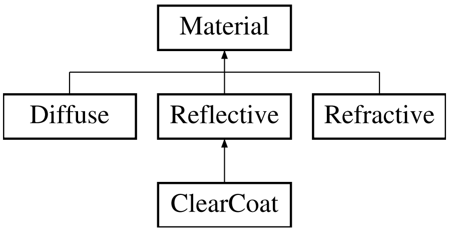
- fileloader/fileloader_ex.hpp

4.21 Material Class Reference

An abstract base class for any type of material object can have.

```
#include <material.hpp>
```

Inheritance diagram for Material:



Public Member Functions

- Color getColor ()

  Gets the color vector for the material.
- std::string getName ()

  Gets the name of the material.
- Material (Color color, std::string name)

  Constructor for material class.
- Vector reflectionDir (Ray &ray, Hit &hit)

  Computes the direction of perfectly reflected ray.
- Vector diffuseDir (Hit &hit)

  Computes the direction of the diffuse ray.
- void updateColor (Ray &ray)

  Updates the color of the ray after interacting with the material.
- virtual void updateRay (Ray &ray, Hit &hit)=0

  Pure virtual function that updates the ray according to the properties of the material.

Public Attributes

- RandomGenerator **rnd_**

4.21.1 Detailed Description

An abstract base class for any type of material object can have.

4.21.2 Constructor & Destructor Documentation

4.21.2.1 Material()

```
Material::Material (
             Color color,
             std::string name )  [inline]
```

Constructor for material class.

Parameters

color	Material color as RGB value
name	Name of the material

4.21.3 Member Function Documentation

4.21.3.1 diffuseDir()

```
Vector Material::diffuseDir (
             Hit & hit )  [inline]
```

Computes the direction of the diffuse ray.

randomDirection() and ray.normal are both normalized so this gives a random vector whose probability distribution is weighted towards the ray.normal.

Parameters

ray	Ray that did hit the material

Returns

vector towards the direction of the diffused ray

4.21.3.2 getColor()

```
Color Material::getColor ( )  [inline]
```

Gets the color vector for the material.

Returns

Color vector representing the RGB value for the material

4.21.3.3 getName()

```
std::string Material::getName ( )  [inline]
```

Gets the name of the material.

Returns

std::string representing the name of the material

4.21.3.4 reflectionDir()

```
Vector Material::reflectionDir (
            Ray & ray,
            Hit & hit )  [inline]
```

Computes the direction of perfectly reflected ray.

Parameters

ray	Ray that did hit the material

Returns

vector towards the direction of the reflected ray

4.21.3.5 updateColor()

```
void Material::updateColor (
            Ray & ray )  [inline]
```

Updates the color of the ray after interacting with the material.

Component wise product with the original color and the color of the interaction material.

Parameters

ray	Ray that did hit the material

4.21.3.6 updateRay()

```
virtual void Material::updateRay (
          Ray & ray,
          Hit & hit )  [pure virtual]
```

Pure virtual function that updates the ray according to the properties of the material.

Parameters

ray	Ray that did hit the material

Implemented in Diffuse, Reflective, ClearCoat, and Refractive.

The documentation for this class was generated from the following file:

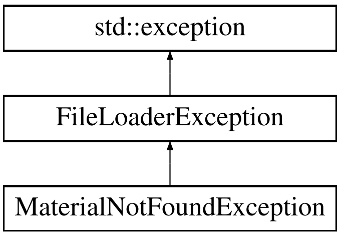
- scenery/material.hpp

4.22 MaterialNotFoundException Class Reference

Representation of material not found exception.

`#include <fileloader_ex.hpp>`

Inheritance diagram for MaterialNotFoundException:



Public Member Functions

- MaterialNotFoundException (std::string filepath, int line)

    Constructor for MaterialNotFoundException.
- virtual const char ∗ what () const noexcept

    Function that creates an exception message for objects that are missing the material definition in the YAML file.

4.22.1 Detailed Description

Representation of material not found exception.

4.22.2 Constructor & Destructor Documentation

4.22.2.1 MaterialNotFoundException()

```
MaterialNotFoundException::MaterialNotFoundException (
          std::string filepath,
          int line )  [inline]
```

Constructor for MaterialNotFoundException.

Parameters

filepath	Filepath of the YAML file
line	Line number of the object that do not have material defined

4.22.3 Member Function Documentation

4.22.3.1 what()

```
virtual const char * MaterialNotFoundException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for objects that are missing the material definition in the YAML file.

Returns

Pointer to the first char of the exception message

Implements FileLoaderException.

The documentation for this class was generated from the following file:

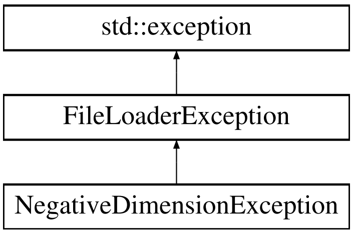
- fileloader/fileloader_ex.hpp

4.23 NegativeDimensionException Class Reference

Representation of negative dimension exception.

```
#include <fileloader_ex.hpp>
```

Inheritance diagram for NegativeDimensionException:



Public Member Functions

- NegativeDimensionException (std::string filepath, float value, int line)

  Constructor for NegativeDimensionException.
- virtual const char ∗ what () const noexcept

  Function that creates an exception message for negative dimension value.

4.23.1 Detailed Description

Representation of negative dimension exception.

4.23.2 Constructor & Destructor Documentation

4.23.2.1 NegativeDimensionException()

```
NegativeDimensionException::NegativeDimensionException (
            std::string filepath,
            float value,
            int line )  [inline]
```

Constructor for NegativeDimensionException.

Parameters

filepath	Filepath of the YAML file
value	Value of the negative dimension
line	Line number where the negative dimension is defined

4.23.3 Member Function Documentation

4.23.3.1 what()

```
virtual const char * NegativeDimensionException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for negative dimension value.

Returns

Pointer to the first char of the exception message

Implements FileLoaderException.

The documentation for this class was generated from the following file:

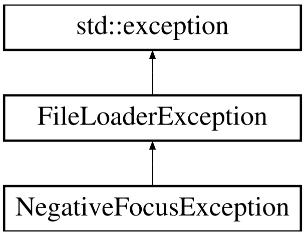
- fileloader/fileloader_ex.hpp

4.24 NegativeFocusException Class Reference

Representation of negative focus exception.

`#include <fileloader_ex.hpp>`

Inheritance diagram for NegativeFocusException:



Public Member Functions

- NegativeFocusException (std::string filepath, float focus, int line)

    Constructor for NegativeFocusException.
- virtual const char ∗ what () const noexcept

    Function that creates an exception message for negative focus distance.

4.24.1 Detailed Description

Representation of negative focus exception.

4.24.2 Constructor & Destructor Documentation

4.24.2.1 NegativeFocusException()

```
NegativeFocusException::NegativeFocusException (
             std::string filepath,
             float focus,
             int line )  [inline]
```

Constructor for NegativeFocusException.

Parameters

filepath	Filepath of the YAML file
focus	Value of the negative focus distance
line	Line number where the negative dimension is defined

4.24.3 Member Function Documentation

4.24.3.1 what()

```
virtual const char * NegativeFocusException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for negative focus distance.

Returns

Pointer to the first char of the exception message

Implements FileLoaderException.

The documentation for this class was generated from the following file:

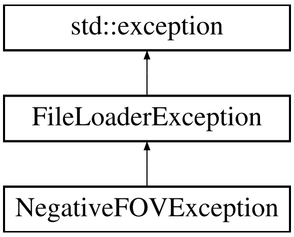
- fileloader/fileloader_ex.hpp

4.25 NegativeFOVException Class Reference

Representation of negative field of view exception.

```
#include <fileloader_ex.hpp>
```

Inheritance diagram for NegativeFOVException:



Public Member Functions

- NegativeFOVException (std::string filepath, float fow, int line)

  Constructor for NegativeFOVException.
- virtual const char ∗ what () const noexcept

  Function that creates an exception message for negative field of view.

4.25.1 Detailed Description

Representation of negative field of view exception.

4.25.2 Constructor & Destructor Documentation

4.25.2.1 NegativeFOVException()

```
NegativeFOVException::NegativeFOVException (
             std::string filepath,
             float fow,
             int line )  [inline]
```

Constructor for NegativeFOVException.

Parameters

filepath	Filepath of the YAML file
fow	Value of the negative field of view
line	Line number where the negative fow is defined

4.25.3 Member Function Documentation

4.25.3.1 what()

```
virtual const char * NegativeFOVException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for negative field of view.

Returns

Pointer to the first char of the exception message

Implements FileLoaderException.

The documentation for this class was generated from the following file:

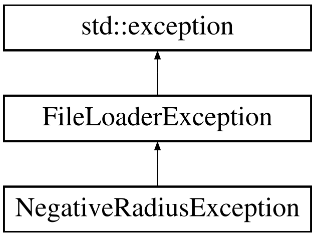
- fileloader/fileloader_ex.hpp

4.26 NegativeRadiusException Class Reference

Representation of negative radius exception.

```
#include <fileloader_ex.hpp>
```

Inheritance diagram for NegativeRadiusException:



Public Member Functions

- NegativeRadiusException (std::string filepath, float radius, int line)

  Constructor for NegativeRadiusException.
- virtual const char ∗ what () const noexcept

  Function that creates an exception message for negative radius.

4.26.1 Detailed Description

Representation of negative radius exception.

4.26.2 Constructor & Destructor Documentation

4.26.2.1 NegativeRadiusException()

```
NegativeRadiusException::NegativeRadiusException (
            std::string filepath,
            float radius,
            int line )  [inline]
```

Constructor for NegativeRadiusException.

Parameters

filepath	Filepath of the YAML file
radius	Value of the negative radius
line	Line number where the negative radius is defined

4.26.3 Member Function Documentation

4.26.3.1 what()

```
virtual const char * NegativeRadiusException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for negative radius.

Returns

Pointer to the first char of the exception message.

Implements FileLoaderException.

The documentation for this class was generated from the following file:

- fileloader/fileloader_ex.hpp

4.27 Node Struct Reference

Struct representing a node in the bounding volume hiererchy.

```
#include <bvh.hpp>
```

Public Member Functions

- bool **isLeaf** ()

Public Attributes

- AABB **box**
- int **leftChild**
- int **firstTriIdx**
- int **triCount**

4.27.1 Detailed Description

Struct representing a node in the bounding volume hiererchy.

The documentation for this struct was generated from the following file:

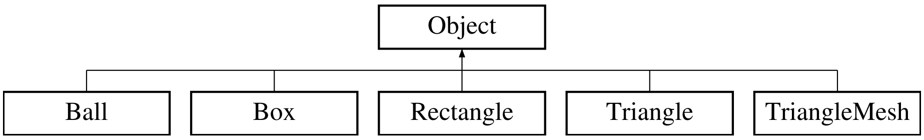
- objects/bvh.hpp

4.28 Object Class Reference

An abstract base class for any type of visible object in a scene.

```
#include <object.hpp>
```

Inheritance diagram for Object:



Public Member Functions

- **Object** (Vector position, std::shared_ptr< Material > material)
- Vector **getPosition** () const
- std::shared_ptr< Material > **getMaterial** () const
- virtual void collision (Ray &ray, Hit &rayHit, float &smallestDistance)=0

  Calculate whether a given ray collides with the object.
- virtual void printInfo (std::ostream &out) const =0

  Print object info to the desired output stream.

4.28.1 Detailed Description

An abstract base class for any type of visible object in a scene.

4.28.2 Member Function Documentation

4.28.2.1 collision()

```
virtual void Object::collision (
            Ray & ray,
            Hit & rayHit,
            float & smallestDistance )  [pure virtual]
```

Calculate whether a given ray collides with the object.

If the ray collides with the object and the collision is closer than the current smallest distance, the "rayHit" data structure will be updated according to the collision.

Parameters

ray	ray whose collision will be checked
rayHit	address of a Hit data structure
smallestDistance	

Implemented in Ball, Box, Rectangle, Triangle, and TriangleMesh.

4.28.2.2 printInfo()

```
virtual void Object::printInfo (
             std::ostream & out ) const  [pure virtual]
```

Print object info to the desired output stream.

Parameters

out	output stream
object	object to be printed

Returns

std::ostream& the output stream

Implemented in Ball, Box, Rectangle, Triangle, and TriangleMesh.

The documentation for this class was generated from the following file:

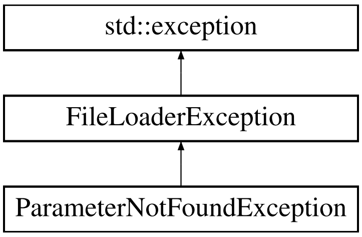
- objects/object.hpp

4.29 ParameterNotFoundException Class Reference

Representation of parameter not found exception.

`#include <fileloader_ex.hpp>`

Inheritance diagram for ParameterNotFoundException:



Public Member Functions

- ParameterNotFoundException (std::string filepath, int line)

  Constructor for ParameterNotFoundException.
- virtual const char ∗ what () const noexcept

  Function that creates an exception message for missing parameters.

4.29.1 Detailed Description

Representation of parameter not found exception.

4.29.2 Constructor & Destructor Documentation

4.29.2.1 ParameterNotFoundException()

```
ParameterNotFoundException::ParameterNotFoundException (
            std::string filepath,
            int line )  [inline]
```

Constructor for ParameterNotFoundException.

Parameters

filepath	Filepath of the YAML file
line	Line number where the parameter is missing

4.29.3 Member Function Documentation

4.29.3.1 what()

```
virtual const char * ParameterNotFoundException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for missing parameters.

Returns

Pointer to the first char of the exception message

Implements FileLoaderException.

The documentation for this class was generated from the following file:

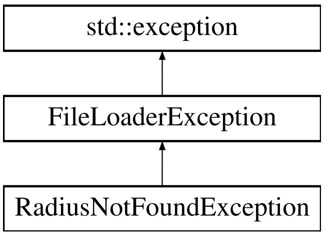
- fileloader/fileloader_ex.hpp

4.30 RadiusNotFoundException Class Reference

Representation of radius not found exception.

```
#include <fileloader_ex.hpp>
```

Inheritance diagram for RadiusNotFoundException:



Public Member Functions

- RadiusNotFoundException (std::string filepath, int line)

  Constructor for RadiusNotFoundException.
- virtual const char ∗ what () const noexcept

  Function that creates an exception message for ball object that do not have radius defined in the YAML file.

4.30.1 Detailed Description

Representation of radius not found exception.

4.30.2 Constructor & Destructor Documentation

4.30.2.1 RadiusNotFoundException()

```
RadiusNotFoundException::RadiusNotFoundException (
              std::string filepath,
              int line )  [inline]
```

Constructor for RadiusNotFoundException.

Parameters

filepath	Filepath of the YAML file
line	Line number of the ball that do not have radius defined

4.30.3 Member Function Documentation

4.30.3.1 what()

```
virtual const char * RadiusNotFoundException::what ( ) const  [inline], [virtual], [noexcept]
```

Function that creates an exception message for ball object that do not have radius defined in the YAML file.

Returns

Pointer to the first char of the exception message

Implements FileLoaderException.

The documentation for this class was generated from the following file:

- fileloader/fileloader_ex.hpp

4.31 RandomGenerator Class Reference

An object for creating random numbers and directions.

```
#include <randomgenerator.hpp>
```

Public Member Functions

- Vector randomDirection ()

  Creates a random direction, i.e., a random point on the unit sphere.
- float randomZeroToOne ()

  Returns a random real number between 0 and 1.
- Vector2 randomInCircle ()

  Creates a random point inside the unit disk.

4.31.1 Detailed Description

An object for creating random numbers and directions.

4.31.2 Member Function Documentation

4.31.2.1 randomDirection()

```
Vector RandomGenerator::randomDirection ( )  [inline]
```

Creates a random direction, i.e., a random point on the unit sphere.

Returns

3-dimensional vector pointing to a random direction

4.31.2.2 randomInCircle()

```
Vector2 RandomGenerator::randomInCircle ( )  [inline]
```

Creates a random point inside the unit disk.

Returns

2-dimensional vector inside of the unit disk.

4.31.2.3 randomZeroToOne()

```
float RandomGenerator::randomZeroToOne ( )  [inline]
```

Returns a random real number between 0 and 1.

Returns

float

The documentation for this class was generated from the following file:

- utils/randomgenerator.hpp

4.32 Ray Struct Reference

Struct representing a ray.

```
#include <types.hpp>
```

Public Attributes

- Point **origin**
- Vector **direction**
- bool **inside_material** = false
- Color **color** = Color(1.0, 1.0, 1.0)
- Light **light** = Color(0.0, 0.0, 0.0)

4.32.1 Detailed Description

Struct representing a ray.

The documentation for this struct was generated from the following file:

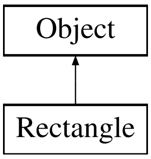
- utils/types.hpp

4.33 Rectangle Class Reference

Representation of a box (rectangular cuboid) object in the scene.

```
#include <rectangle.hpp>
```

Inheritance diagram for Rectangle:



Public Member Functions

- **Rectangle** (Vector position, float width, float height, std::shared_ptr< Material > material)
- void collision (Ray &ray, Hit &rayHit, float &smallestDistance)

  Calculate whether a given ray collides with the rectangle.
- float **getWidth** () const
- float **getHeight** () const
- void rotate (float angle, Vector axis)

  Rotates the box around a given axis.
- void printInfo (std::ostream &out) const

  Print box info to the desired output stream.

Public Member Functions inherited from Object

- **Object** (Vector position, std::shared_ptr< Material > material)
- Vector **getPosition** () const
- std::shared_ptr< Material > **getMaterial** () const

4.33.1 Detailed Description

Representation of a box (rectangular cuboid) object in the scene.

The position represent the geometric center of the box.

4.33.2 Member Function Documentation

4.33.2.1 collision()

```
void Rectangle::collision (
            Ray & ray,
            Hit & rayHit,
            float & smallestDistance )  [inline], [virtual]
```

Calculate whether a given ray collides with the rectangle.

If the ray collides with the rectangle and the collision is closer than the current smallest distance, the "rayHit" data structure will be updated according to the collision.

Parameters

ray	ray whose collision will be checked
rayHit	address of a Hit data structure
smallestDistance	

Implements Object.

4.33.2.2 printInfo()

```
void Rectangle::printInfo (
            std::ostream & out ) const  [inline], [virtual]
```

Print box info to the desired output stream.

Parameters

out	output stream

Returns

std::ostream& the output stream

Implements Object.

4.33.2.3 rotate()

```
void Rectangle::rotate (
            float angle,
            Vector axis )  [inline]
```

Rotates the box around a given axis.

The center of rotation is the box's center of mass.

Parameters

angle	rotation in radians
axis	axis of rotation (has to be normalized)

The documentation for this class was generated from the following file:

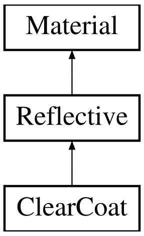
- objects/rectangle.hpp

4.34 Reflective Class Reference

Representation of a reflective material that can be used for mirror or metal like materials.

```
#include <material.hpp>
```

Inheritance diagram for Reflective:



Public Member Functions

- Reflective (Color color, std::string name, float specularity)

  Constructor for Reflective class material.
- float getSpecularity ()

  Gets the specularity of the reflective material.
- void updateRay (Ray &ray, Hit &hit)

  Updates the ray according to properties of reflective material.

Public Member Functions inherited from Material

- Color getColor ()

  Gets the color vector for the material.
- std::string getName ()

  Gets the name of the material.
- Material (Color color, std::string name)

  Constructor for material class.
- Vector reflectionDir (Ray &ray, Hit &hit)

  Computes the direction of perfectly reflected ray.
- Vector diffuseDir (Hit &hit)

  Computes the direction of the diffuse ray.
- void updateColor (Ray &ray)

  Updates the color of the ray after interacting with the material.

Additional Inherited Members

Public Attributes inherited from Material

- RandomGenerator **rnd_**

4.34.1 Detailed Description

Representation of a reflective material that can be used for mirror or metal like materials.

4.34.2 Constructor & Destructor Documentation

4.34.2.1 Reflective()

```
Reflective::Reflective (
             Color color,
             std::string name,
             float specularity )  [inline]
```

Constructor for Reflective class material.

Parameters

color	Material color as RGB value
name	Name of the material
specularity	Specularity of the material

4.34.3 Member Function Documentation

4.34.3.1 getSpecularity()

```
float Reflective::getSpecularity ( ) [inline]
```

Gets the specularity of the reflective material.

Returns

specularity of the object

4.34.3.2 updateRay()

```
void Reflective::updateRay (
            Ray & ray,
            Hit & hit ) [inline], [virtual]
```

Updates the ray according to properties of reflective material.

Parameters

ray	Ray that did hit the material

Implements Material.

The documentation for this class was generated from the following file:

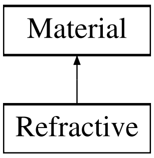
- scenery/material.hpp

4.35 Refractive Class Reference

Representation of a refractive material, which can be used for glass or diamond like materials.

```
#include <material.hpp>
```

Inheritance diagram for Refractive:



Public Member Functions

- Refractive (Color color, std::string name, float refraction_ratio)

  Constructor for Refractive class.
- void updateRay (Ray &ray, Hit &hit)

  Updates the ray according to properties of Refractive material.

Public Member Functions inherited from Material

- Color getColor ()

  Gets the color vector for the material.
- std::string getName ()

  Gets the name of the material.
- Material (Color color, std::string name)

  Constructor for material class.
- Vector reflectionDir (Ray &ray, Hit &hit)

  Computes the direction of perfectly reflected ray.
- Vector diffuseDir (Hit &hit)

  Computes the direction of the diffuse ray.
- void updateColor (Ray &ray)

  Updates the color of the ray after interacting with the material.

Additional Inherited Members

Public Attributes inherited from Material

- RandomGenerator **rnd_**

4.35.1 Detailed Description

Representation of a refractive material, which can be used for glass or diamond like materials.

4.35.2 Constructor & Destructor Documentation

4.35.2.1 Refractive()

```
Refractive::Refractive (
              Color color,
              std::string name,
              float refraction_ratio )  [inline]
```

Constructor for Refractive class.

Parameters

color	Material color as RGB value
name	Name of the material
refraction_ratio	Refraction ratio from outside to inside the material

4.35.3 Member Function Documentation

4.35.3.1 updateRay()

```
void Refractive::updateRay (
             Ray & ray,
             Hit & hit )  [inline], [virtual]
```

Updates the ray according to properties of Refractive material.

Parameters

ray	Ray that did hit the material
hit	Information about the hit point

Implements Material.

The documentation for this class was generated from the following file:

- scenery/material.hpp

4.36 Renderer Class Reference

Implements the ray tracing algorithm.

`#include <renderer.hpp>`

Public Member Functions

- Renderer (int res_x, int res_y, std::shared_ptr< Scene > sceneToRender)

    Construct a new Renderer object and initialize all necessary values.
- void setMaxBounces (int bounces)

    Function to set maximum bounces for rays.
- auto parallelRender (int samples)

    Rendering function that uses all available CPU cores.
- void setDof (float dof)

    Set the depth of field for the camera in the renderer.

4.36.1 Detailed Description

Implements the ray tracing algorithm.

4.36.2 Constructor & Destructor Documentation

4.36.2.1 Renderer()

```
Renderer::Renderer (
             int res_x,
             int res_y,
             std::shared_ptr< Scene > sceneToRender )  [inline]
```

Construct a new Renderer object and initialize all necessary values.

Parameters

res_x	horizontal resolution of the rendering area
res_y	vertical resolution of the rendering area
sceneToRender	Scene object to be renderer

4.36.3 Member Function Documentation

4.36.3.1 parallelRender()

```
auto Renderer::parallelRender (
             int samples )  [inline]
```

Rendering function that uses all available CPU cores.

Parameters

samples	Amount of samples that will be taken for each pixel

4.36.3.2 setDof()

```
void Renderer::setDof (
             float dof )  [inline]
```

Set the depth of field for the camera in the renderer.

Parameters

dof	Value for depth of field

4.36.3.3 setMaxBounces()

```
void Renderer::setMaxBounces (
             int bounces )  [inline]
```

Function to set maximum bounces for rays.

Parameters

bounces	Number of maximum bounces ray can take

The documentation for this class was generated from the following file:

- rendering/renderer.hpp

4.37 Scene Class Reference

Representation of a 3D-scene.

```
#include <scene.hpp>
```

Public Member Functions

- **Scene** ()=default

    Construct an empty scene.
- Scene (Camera camera, std::list< std::shared_ptr< Object > > objects)

    Construct a new Scene object with a camera and some objects.
- ∼**Scene** ()

    Destructor for Scene object. Deletes all the object in the scene.
- Scene & **operator=** (const Scene &that)=default
- **Scene** (const Scene &that)=default
- Camera **getCamera** () const
- Environment & **getEnvironment** ()
- std::list< std::shared_ptr< Object > > **getObjects** () const
- void **setFov** (float fov)
- void **setFocusDist** (float dof)

Friends

- std::ostream & operator<< (std::ostream &out, const Scene &scene)

    Print scene info to the desired output stream.

4.37.1 Detailed Description

Representation of a 3D-scene.

4.37.2 Constructor & Destructor Documentation

4.37.2.1 Scene()

```
Scene::Scene (
            Camera camera,
            std::list< std::shared_ptr< Object > > objects )  [inline]
```

Construct a new Scene object with a camera and some objects.

Parameters

camera	
objects	

4.37.3 Friends And Related Symbol Documentation

4.37.3.1 operator<<

```
std::ostream & operator<< (
            std::ostream & out,
            const Scene & scene )  [friend]
```

Print scene info to the desired output stream.

Parameters

out	output stream
scene	scene to be printed

Returns

std::ostream& the output stream

The documentation for this class was generated from the following file:

- scenery/scene.hpp

4.38 Textbox Class Reference

Implements a class for creating textboxes that take inputs into SFML-window.

```
#include <textbox.hpp>
```

Public Member Functions

- Textbox (int size, sf::Color color, bool isSelected, int limit, std::string preText)

  Construct a new Textbox object.
- ∼**Textbox** ()=default

  Destroy the Textbox object.
- void setFont (sf::Font &font)

  Set the font of the texbox text.
- void **setSelected** ()

  Set the object to be selected.
- void **setUnselected** ()

  Set the object to be unselected.
- void setPos (sf::Vector2f pos)

  Sets the position of the box in the SFML window.
- void setColor (sf::Color textColor)

  Set the color of the text inside the box.
- void draw (sf::RenderWindow &window)

  Renders a box on a specified window.
- bool onButton (sf::RenderWindow &window)

  Checks if the users mouse is on the textbox.
- std::string getInput ()

  Get the text typed into the textbox.
- void typedOn (sf::Event input)

  Takes an event, that is supposed to be a keypress, and acts accrodingly: checks if the box is selected and if it is the input is checked. If the input is semi-logical (unicode < 128) and the character limit is not exceeded, the character is added to the box. Backspace key deletes the previous inputted character instead of adding it to the box.

4.38.1 Detailed Description

Implements a class for creating textboxes that take inputs into SFML-window.

4.38.2 Constructor & Destructor Documentation

4.38.2.1 Textbox()

```
Textbox::Textbox (
             int size,
             sf::Color color,
             bool isSelected,
             int limit,
             std::string preText )  [inline]
```

Construct a new Textbox object.

Parameters

size	Size of the text
color	Color of the text
isSelected	A boolean value telling if the box is selected
limit	Limit to the the user can input into the box
preText	A non modifiable text in the box before the input space

4.38.3 Member Function Documentation

4.38.3.1 draw()

```
void Textbox::draw (
             sf::RenderWindow & window )  [inline]
```

Renders a box on a specified window.

Parameters

window	a reference to the window

4.38.3.2 getInput()

```
std::string Textbox::getInput ( )  [inline]
```

Get the text typed into the textbox.

Returns

std::string of the input text

4.38.3.3 onButton()

```
bool Textbox::onButton (
             sf::RenderWindow & window )  [inline]
```

Checks if the users mouse is on the textbox.

Parameters

window	reference to the window the textbox is on

Returns

true value if mouse is on the box

false value if mouse is not on the box

4.38.3.4 setColor()

```
void Textbox::setColor (
             sf::Color textColor )  [inline]
```

Set the color of the text inside the box.

Parameters

textColor	Color that the text is to be set to

4.38.3.5 setFont()

```
void Textbox::setFont (
             sf::Font & font )  [inline]
```

Set the font of the texbox text.

Parameters

font	a loaded font for the text

4.38.3.6 setPos()

```
void Textbox::setPos (
             sf::Vector2f pos )  [inline]
```

Sets the position of the box in the SFML window.

Parameters

pos	Position of the box

4.38.3.7 typedOn()

```
void Textbox::typedOn (
             sf::Event input )  [inline]
```

Takes an event, that is supposed to be a keypress, and acts accrodingly: checks if the box is selected and if it is the input is checked. If the input is semi-logical (unicode < 128) and the character limit is not exceeded, the character is added to the box. Backspace key deletes the previous inputted character instead of adding it to the box.

Parameters

input	a sf::Event of what the user is doing eg. clicking a button or moving mouse

The documentation for this class was generated from the following file:

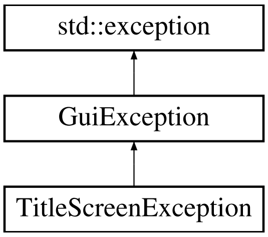
- interface/textbox.hpp

4.39 TitleScreenException Class Reference

Exception for unexpected behaviour in the title screen.

`#include <gui_ex.hpp>`

Inheritance diagram for TitleScreenException:



Public Member Functions

- virtual const char * what () const noexcept

4.39.1 Detailed Description

Exception for unexpected behaviour in the title screen.

4.39.2 Member Function Documentation

4.39.2.1 what()

```
virtual const char * TitleScreenException::what ( ) const  [inline], [virtual], [noexcept]
```

Implements GuiException.

The documentation for this class was generated from the following file:

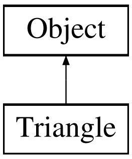
- interface/gui_ex.hpp

4.40 Triangle Class Reference

Representation of a mathematical triangle object in the scene.

```
#include <triangle.hpp>
```

Inheritance diagram for Triangle:



Public Member Functions

- Triangle (Vector v0, Vector v1, Vector v2, std::shared_ptr< Material > m)

  Construct a new Triangle object.
- void collision (Ray &ray, Hit &rayHit, float &smallestDistance)

  Calculate whether a given ray collides with the triangle using the Möller-Trumbore algorithm.
- void printInfo (std::ostream &out) const

  Print triangle info to the desired output stream.
- std::vector< Vector > getVertexPos () const

  Get the vertex positions of the triangle.
- std::vector< Vector > getPlaneVec () const

  Get the plane vectors of the triangle.
- Vector getNormal () const

  Get the normal of the triangle.
- Vector getCentroid () const

  Get the centroid of the triangle.

Public Member Functions inherited from Object

- **Object** (Vector position, std::shared_ptr< Material > material)
- Vector **getPosition** () const
- std::shared_ptr< Material > **getMaterial** () const

4.40.1 Detailed Description

Representation of a mathematical triangle object in the scene.

4.40.2 Constructor & Destructor Documentation

4.40.2.1 Triangle()

```
Triangle::Triangle (
            Vector v0,
            Vector v1,
            Vector v2,
            std::shared_ptr< Material > m )  [inline]
```

Construct a new Triangle object.

Parameters

v0	Vertex 1
v1	Vertex 2
v2	Vertex 3
m	Material of the triangle

4.40.3 Member Function Documentation

4.40.3.1 collision()

```
void Triangle::collision (
            Ray & ray,
            Hit & rayHit,
            float & smallestDistance )  [inline], [virtual]
```

Calculate whether a given ray collides with the triangle using the Möller-Trumbore algorithm.

If the ray collides with the triangle and the collision is closer than the current smallest distance, the "rayHit" data structure will be updated according to the collision.

Parameters

ray	ray whose collision will be checked
rayHit	address of a Hit data structure
smallestDistance	current smallest distance

Implements Object.

4.40.3.2 getCentroid()

```
Vector Triangle::getCentroid ( ) const  [inline]
```

Get the centroid of the triangle.

Returns

The centroid vector of the triangle

4.40.3.3 getNormal()

```
Vector Triangle::getNormal ( ) const  [inline]
```

Get the normal of the triangle.

Returns

The normal vector of the triangle

4.40.3.4 getPlaneVec()

```
std::vector< Vector > Triangle::getPlaneVec ( ) const  [inline]
```

Get the plane vectors of the triangle.

Returns

std::vector<Vector> containing the plane vectors

4.40.3.5 getVertexPos()

```
std::vector< Vector > Triangle::getVertexPos ( ) const  [inline]
```

Get the vertex positions of the triangle.

Returns

std::vector<Vector> containing the vertice vectors

4.40.3.6 printInfo()

```
void Triangle::printInfo (
             std::ostream & out ) const  [inline], [virtual]
```

Print triangle info to the desired output stream.

Parameters

out	output stream

Implements Object.

The documentation for this class was generated from the following file:

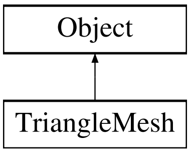
- objects/triangle.hpp

4.41 TriangleMesh Class Reference

Trianglemesh object consiting of Triangles, loaded from .obj file.

```
#include <trianglemesh.hpp>
```

Inheritance diagram for TriangleMesh:



Public Member Functions

- TriangleMesh (std::string obj_filepath, Vector scenePos, std::shared_ptr< Material > m, Vector rotation, double scale)

  Construct a new Triangle Mesh object.
- void collision (Ray &ray, Hit &rayHit, float &smallestDistance)

  Calclute whether a given ray collides with the TriangleMesh.
- void printInfo (std::ostream &out) const

  Print TriangleMesh info to the desired output stream.
- BVH getBVH () const

  Get the BVH of the trianglemesh object.
- std::string getName () const

  Get the name of the trianglemesh object.

Public Member Functions inherited from Object

- **Object** (Vector position, std::shared_ptr< Material > material)
- Vector **getPosition** () const
- std::shared_ptr< Material > **getMaterial** () const

4.41.1 Detailed Description

Trianglemesh object consiting of Triangles, loaded from .obj file.

4.41.2 Constructor & Destructor Documentation

4.41.2.1 TriangleMesh()

```
TriangleMesh::TriangleMesh (
            std::string obj_filepath,
            Vector scenePos,
            std::shared_ptr< Material > m,
            Vector rotation,
            double scale )  [inline]
```

Construct a new Triangle Mesh object.

Parameters

obj_filepath	filepath string
scenePos	position of the object in the scene
m	material of the object
scale	scaling of the object size
angle	counterclockwise rotation of the object in radians

4.41.3 Member Function Documentation

4.41.3.1 collision()

```
void TriangleMesh::collision (
            Ray & ray,
            Hit & rayHit,
            float & smallestDistance )  [inline], [virtual]
```

Calclute whether a given ray collides with the TriangleMesh.

Parameters

ray	ray whose collision will be checked
rayHit	address of a Hit data structure
smallestDistance	current smallest distance

Implements Object.

4.41.3.2 getBVH()

```
BVH TriangleMesh::getBVH ( ) const  [inline]
```

Get the BVH of the trianglemesh object.

Returns

BVH

4.41.3.3 getName()

```
std::string TriangleMesh::getName ( ) const  [inline]
```

Get the name of the trianglemesh object.

Returns

std::string

4.41.3.4 printInfo()

```
void TriangleMesh::printInfo (
            std::ostream & out ) const  [inline], [virtual]
```

Print TriangleMesh info to the desired output stream.

Parameters

out	output stream

Implements Object.

The documentation for this class was generated from the following file:

- objects/trianglemesh.hpp

Chapter 5

File Documentation

5.1 fileloader.hpp

```
#pragma once

#include <yaml-cpp/yaml.h>
#include <string>
#include <exception>
#include <memory>
#include <sys/stat.h>

#include "trianglemesh.hpp"
#include "box.hpp"
#include "rectangle.hpp"
#include "ball.hpp"
#include "scene.hpp"
#include "types.hpp"
#include "fileloader_ex.hpp"
#include "material.hpp"

class FileLoader {
    public:

        FileLoader(std::string filepath) {
            // Check the existence of the given file
            struct stat buf;
            if (stat(filepath.c_str(), &buf) == 0)
            {
                filepath_ = filepath;
            }
            else
            {
                throw InvalidFilepathException(filepath);
            }
        }

        std::shared_ptr<Scene> loadSceneFile() {
            Camera camera = LoadCamera();
            std::list<std::shared_ptr<Object» objects = LoadObjects();
            scene_ = std::make_shared<Scene>(camera, objects);
            LoadEnvironment(scene_);
            return scene_;
        }

        ~FileLoader() {}

        // Getter for filepath
        std::string getFilepath() { return filepath_; }


    private:

        void LoadEnvironment(std::shared_ptr<Scene> scene) {
            if(YAML::LoadFile(filepath_)["Environment"]) {
                YAML::Node environment_node = loadParams("Environment");
                Eigen::Vector3d skyColor = LoadVector(environment_node, "SkyColor");
                Eigen::Vector3d horizonColor = LoadVector(environment_node, "HorizonColor");
                Eigen::Vector3d groundColor = LoadVector(environment_node, "GroundColor");
                (*scene).getEnvironment().setSky(skyColor, horizonColor, groundColor);
            }else {
                (*scene).getEnvironment().setSky();
            }
        }

        Eigen::Vector3d LoadVector(YAML::Node node, std::string key) {
            Eigen::Vector3d vector;
            YAML::Node coords = node[key];
            // Throw exception if incorrect size or if not defined
            if (!coords.IsDefined()) {
                throw InvalidKeyException(filepath_, key);
            }
            size_t size = coords.size();
            if (size != 3) {
                throw InvalidSizeVectorException(filepath_, size, coords.Mark().line);
            }
            int i = 0;
            for (YAML::const_iterator it=coords.begin(); it!=coords.end(); ++it) {
                vector[i] = it->as<float>();
                i++;
            }
            return vector;
        }

        std::shared_ptr<Material> LoadMaterial(YAML::Node node) {
            //Material material
            YAML::Node material_node = node["Material"];
            if (!material_node.IsDefined()) {
                throw MaterialNotFoundException(filepath_, node.Mark().line);
            }

            // Initialize default values
            Color color = Color(1.0, 1.0, 1.0);
            Color emission_color = Color(1.0, 1.0, 1.0);
            float emission_strength = 0.0;
            std::string name = "UNDEFINED";
            float specularity = 1.0;
            float clearcoat = 0.5;
            float refraction_ratio = 1.0;
            Color clearCoat_color = Color(1.0, 1.0, 1.0);

            // Update the parameters based on .yaml file if the parameters are defined
            if(material_node["Color"]) {
                color = LoadVector(material_node, "Color");
            }
            if(material_node["EmissionColor"]) {
                emission_color = LoadVector(material_node, "EmissionColor");
            }
            if(material_node["EmissionStrength"]) {
                emission_strength = material_node["EmissionStrength"].as<float>();
            }
            if(material_node["Specularity"]) {
                specularity = material_node["Specularity"].as<float>();
            }
            if(material_node["Name"]) {
                name = material_node["Name"].as<std::string>();
            }
            if(material_node["ClearCoat"]) {
                clearcoat = material_node["ClearCoat"].as<float>();
            }
            if(material_node["ClearCoatColor"]) {
                clearCoat_color = LoadVector(material_node, "ClearCoatColor");
            }
            if(material_node["RefractionRatio"]) {
                refraction_ratio = material_node["RefractionRatio"].as<float>();
            }

            // Return shared pointer to the correct material type or throw exception
            std::string type = material_node["Type"].as<std::string>();
            if (type == "Diffuse") {
                return std::make_shared<Diffuse>(color, name, emission_strength, emission_color);
            }
            else if (type == "Reflective") {
                return std::make_shared<Reflective>(color, name, specularity);
            }
            else if (type == "ClearCoat") {
                return std::make_shared<ClearCoat>(color, name, specularity, clearcoat,
clearCoat_color);
            }
            else if (type == "Refractive") {
                return std::make_shared<Refractive>(color, name, refraction_ratio);
            }
            else {
                throw InvalidMaterialTypeException(filepath_, material_node.Mark().line);
            }
        }

        std::shared_ptr<Ball> LoadBall(YAML::Node ball) {
            YAML::Node radius_node = ball["Radius"];
            if (!radius_node.IsDefined()) {
                throw RadiusNotFoundException(filepath_, ball.Mark().line);
            }
            float radius = radius_node.as<float>();
            if (radius < 0)
            {
                throw NegativeRadiusException(filepath_, radius, radius_node.Mark().line);
            }
            else
            {
                std::shared_ptr<Ball> ball_ptr = std::make_shared<Ball>(LoadVector(ball, "Position"), radius, LoadMaterial(ball));
                return ball_ptr;
            }
        }

        std::shared_ptr<Box> LoadBox(YAML::Node box) {
            YAML::Node width_node = box["Width"];
            YAML::Node height_node = box["Height"];
            YAML::Node depth_node = box["Depth"];

            if (!width_node.IsDefined() || !height_node.IsDefined() || !depth_node.IsDefined()) {
                throw ParameterNotFoundException(filepath_, box.Mark().line);
            }

            float width = width_node.as<float>();
            float height = height_node.as<float>();
            float depth = depth_node.as<float>();

            if (width < 0)
            {
                throw NegativeDimensionException(filepath_, width, width_node.Mark().line);
            }

            if (height < 0)
            {
                throw NegativeDimensionException(filepath_, height, height_node.Mark().line);
            }

            if (depth < 0)
            {
                throw NegativeDimensionException(filepath_, depth, depth_node.Mark().line);
            }

            std::shared_ptr<Box> box_ptr = std::make_shared<Box>(LoadVector(box, "Position"), width, height, depth, LoadMaterial(box));

            Vector rotation;
            try
            {
                rotation = LoadVector(box, "Rotation");
            }
            catch(const InvalidKeyException& e)
            {
                std::cout « e.what() « std::endl;
                std::cout « "Standard rotation (0, 0, 0) will be used." « std::endl;
                rotation = Vector(0, 0, 0);
            }
            rotation = (rotation / 180.0) * M_PI;

            box_ptr->rotate(rotation[0], Vector::UnitX());
            box_ptr->rotate(rotation[1], Vector::UnitY());
            box_ptr->rotate(rotation[2], Vector::UnitZ());

            return box_ptr;
        }

        std::shared_ptr<TriangleMesh> LoadTriangleMesh(YAML::Node tmesh) {
            YAML::Node filepath_node = tmesh["Filepath"];
            YAML::Node scale_node = tmesh["Scale"];

            if (!filepath_node.IsDefined() || !scale_node.IsDefined()) {
                throw ParameterNotFoundException(filepath_, tmesh.Mark().line);
            }

            std::string tmesh_filepath = filepath_node.as<std::string>();

            struct stat buf;
            if (stat(tmesh_filepath.c_str(), &buf) != 0)
            {
                throw InvalidFilepathException(tmesh_filepath);
            }

            double scale = scale_node.as<double>();

            if (scale < 0)
            {
                throw NegativeDimensionException(filepath_, scale, scale_node.Mark().line);
            }

            Vector rotation;
            try
            {
                rotation = LoadVector(tmesh, "Rotation");
            }
            catch(const InvalidKeyException& e)
            {
                std::cout « e.what() « std::endl;
                std::cout « "Standard rotation (0, 0, 0) will be used." « std::endl;
                rotation = Vector(0, 0, 0);
            }
            rotation = (rotation / 180.0) * M_PI;

            std::shared_ptr<TriangleMesh> tmesh_ptr = std::make_shared<TriangleMesh>(tmesh_filepath, LoadVector(tmesh, "Position"), LoadMaterial(tmesh), rotation, scale);

            return tmesh_ptr;
        }

        std::shared_ptr<Rectangle> LoadRectangle(YAML::Node rect) {
            YAML::Node width_node = rect["Width"];
            YAML::Node height_node = rect["Height"];

            if (!width_node.IsDefined() || !height_node.IsDefined()) {
                throw ParameterNotFoundException(filepath_, rect.Mark().line);
            }

            float width = width_node.as<float>();
            float height = height_node.as<float>();

            if (width < 0)
            {
                throw NegativeDimensionException(filepath_, width, width_node.Mark().line);
            }

            if (height < 0)
            {
                throw NegativeDimensionException(filepath_, height, height_node.Mark().line);
            }

            std::shared_ptr<Rectangle> rect_ptr = std::make_shared<Rectangle>(LoadVector(rect, "Position"), width, height, LoadMaterial(rect));

            Vector rotation;
            try
            {
                rotation = LoadVector(rect, "Rotation");
            }
            catch(const InvalidKeyException& e)
            {
                std::cout « e.what() « std::endl;
                std::cout « "Standard rotation (0, 0, 0) will be used." « std::endl;
                rotation = Vector(0, 0, 0);
            }
            rotation = (rotation / 180.0) * M_PI;

            rect_ptr->rotate(rotation[0], Vector::UnitX());
            rect_ptr->rotate(rotation[1], Vector::UnitY());
            rect_ptr->rotate(rotation[2], Vector::UnitZ());

            return rect_ptr;
        }

        std::list<std::shared_ptr<Object» LoadObjects() {
            YAML::Node objects = loadParams("Objects");
            std::list<std::shared_ptr<Object» object_list;
            for (YAML::const_iterator it=objects.begin(); it!=objects.end(); ++it) {
                if((*it)["Object"]["Type"].as<std::string>() == "Ball") {
                    object_list.push_back(LoadBall((*it)["Object"]));
                }
                if((*it)["Object"]["Type"].as<std::string>() == "Box") {
                    object_list.push_back(LoadBox((*it)["Object"]));
                }
                if((*it)["Object"]["Type"].as<std::string>() == "TriangleMesh") {
                    object_list.push_back(LoadTriangleMesh((*it)["Object"]));
                }
                if((*it)["Object"]["Type"].as<std::string>() == "Rectangle") {
                    object_list.push_back(LoadRectangle((*it)["Object"]));
                }
            }
            return object_list;
        }

        Camera LoadCamera() {
            YAML::Node params = loadParams("Camera");

            YAML::Node angle_node = params["Angle"];
            float angle = angle_node.IsDefined() ? angle_node.as<float>() : 0; // Set angle to zero if it not defined in yaml

            YAML::Node DoF_node = params["DepthOfField"];
            float DoF = DoF_node.IsDefined() ? DoF_node.as<float>() : 0; // Set depth of field to zero if not defined in yaml

            float fow = params["Fov"].as<float>();
            float focus = params["FocusDistance"].as<float>();
            if (fow < 0)
            {
                throw NegativeFOVException(filepath_, fow, params["Fov"].Mark().line);
            }
            else if (focus < 0)
            {
                throw NegativeFocusException(filepath_, focus, params["FocusDistance"].Mark().line);
            }
            Camera camera;
            camera.position = LoadVector(params, "Position");
            camera.lookingAt = LoadVector(params, "LookingAt");
            camera.direction = (camera.lookingAt - camera.position).normalized();
            Eigen::AngleAxisd rotation(angle*M_PI/180, camera.direction); // Rotation around the axis of camera looking direction
            Vector left = Vector(-camera.direction[1], camera.direction[0], 0).normalized(); // Vector towards left of the image plane (90 degrees with respect to cam dir)
            camera.left = rotation * left; // Rotate the camera's left direction according to rotation
            camera.up = camera.direction.cross(camera.left); // Up direction dynamically determined from camera direction and left direction
            camera.fov = M_PI * fow;
            camera.focus_distance = focus;
            camera.DoF = DoF;
            return camera;
        }

        YAML::Node loadParams(std::string key) {
            YAML::Node params = YAML::LoadFile(filepath_)[key];
            if (!params.IsDefined()) {
                throw InvalidKeyException(filepath_, key);
            }
            else
            {
                return params;
            }
        }



        std::string filepath_;
        std::shared_ptr<Scene> scene_;
};
```

5.2 fileloader_ex.hpp

```
#ifndef FILELOADER_EXCEPTION
#define FILELOADER_EXCEPTION

#include <exception>
#include <iostream>
#include <string>

class FileLoaderException : public std::exception {
    public:
        FileLoaderException() {}

        virtual const char* what() const noexcept = 0;

        virtual ~FileLoaderException() = default;

};

class InvalidFilepathException : public FileLoaderException {
    public:
        InvalidFilepathException(std::string filepath) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nInvalid filepath: " + filepath;
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;

};

class NegativeRadiusException : public FileLoaderException {
    public:
        NegativeRadiusException(std::string filepath, float radius, int line) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nNegative radius (" +
                    std::to_string(radius) + ") in file: " +
                    filepath + ", on line: " + std::to_string(line);
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;

};

class InvalidSizeVectorException : public FileLoaderException {
    public:
        InvalidSizeVectorException(std::string filepath, size_t size, int line) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nInvalid vector of size " +
                    std::to_string(size) + " in file: " + filepath +
                    ", on line: " + std::to_string(line);
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;

};

class NegativeFOVException : public FileLoaderException {
    public:
        NegativeFOVException(std::string filepath, float fow, int line) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nNegative FOW: " + std::to_string(fow) +
                    ", in file: " + filepath + ", on line: " + std::to_string(line);
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;

};

class NegativeFocusException : public FileLoaderException {
    public:
        NegativeFocusException(std::string filepath, float focus, int line) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nNegative focus distance: " +
                    std::to_string(focus) + " in file: " + filepath +
                    ", on line: " + std::to_string(line);
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;

};

class InvalidKeyException : public FileLoaderException {
    public:
        InvalidKeyException(std::string filepath, std::string key) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nInvalid key: " + key + ", for file: " + filepath;
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;

```

```
};

class MaterialNotFoundException : public FileLoaderException {
    public:
        MaterialNotFoundException(std::string filepath, int line) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nMaterial not defined for object at line: " +
                    std::to_string(line) + ", in file: " + filepath + ".";
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;
};

class RadiusNotFoundException : public FileLoaderException {
    public:
        RadiusNotFoundException(std::string filepath, int line) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nRadius not defined for ball at line: " +
                    std::to_string(line) + ", in file: " + filepath + ".";
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;
};

class ParameterNotFoundException : public FileLoaderException {
    public:
        ParameterNotFoundException(std::string filepath, int line) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nMissing parameters in line: " +
                    std::to_string(line) + ", in file: " + filepath + ".";
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;
};

class NegativeDimensionException : public FileLoaderException {
    public:
        NegativeDimensionException(std::string filepath, float value, int line) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nNegative dimension: " +
                    std::to_string(value) + " in file: " + filepath +
                    ", on line: " + std::to_string(line);
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;

};

class InvalidMaterialTypeException : public FileLoaderException {
    public:
        InvalidMaterialTypeException(std::string filepath, int line) : FileLoaderException() {
            msg_ = "FileLoader exception caught:\nInvalid material type in file: " +
                    filepath + ", for material starting on line: " + std::to_string(line) + ".";
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;
};


#endif
```

5.3 button.hpp

```cpp
#pragma once

#include <iostream>
#include <SFML/Graphics.hpp>
class Button {
    public:

        Button(std::string button_text, sf::Color textColor, int text_size, sf::Vector2f button_size, sf::Color button_color)  {
            button_text_.setString(button_text);
            button_text_.setFillColor(textColor);
            button_text_.setCharacterSize(text_size);

            button_.setFillColor(button_color);
            button_.setSize(button_size);
        }

        ~Button() = default;


        void setPos(sf::Vector2f pos) {
            button_.setPosition(pos);

            float text_pos_x = (pos.x + button_.getLocalBounds().width / 2) - (button_text_.getGlobalBounds().width / 2);
            float text_pos_y = (pos.y + button_.getLocalBounds().height / 2) - (button_text_.getGlobalBounds().height / 2);
            button_text_.setPosition({text_pos_x, text_pos_y});
        }

        void setColor(sf::Color color) {
            button_.setFillColor(color);
        }

        void setFont(sf::Font &font) {
            button_text_.setFont(font);
        }

        void draw(sf::RenderWindow &window) {
            window.draw(button_);
            window.draw(button_text_);
        }

        bool onButton(sf::RenderWindow &window) {
            float mousePos_x = sf::Mouse::getPosition(window).x;
            float mousePos_y = sf::Mouse::getPosition(window).y;
            float butPos_x = button_.getPosition().x;
            float butPos_y = button_.getPosition().y;
            float butSize_x = button_.getSize().x;
            float butSize_y = button_.getSize().y;

            bool onButton_x = (mousePos_x >= butPos_x) && (mousePos_x <= (butSize_x + butPos_x));
            bool onButton_y = (mousePos_y >= butPos_y) && (mousePos_y <= (butSize_y + butPos_y));

            return (onButton_x && onButton_y);

        }


    private:
    sf::Text button_text_;
    sf::RectangleShape button_;
};
```

5.4 gui.hpp

```cpp
#pragma once

#include <SFML/Graphics.hpp>
#include "renderer.hpp"
#include "button.hpp"
#include "textbox.hpp"
#include "fileloader.hpp"
#include "types.hpp"
#include "gui_ex.hpp"

class Gui : public Interface {
public:

    std::shared_ptr<Scene> titleScreen() {
        sf::Font arial;
        std::string arialpath = "../fonts/Arial.ttf";
        if(!arial.loadFromFile(arialpath)) {
            throw FontException(arialpath);
        }
        sf::Font comic;
        std::string comicpath = "../fonts/ComicSansMS.ttf";
        if(!comic.loadFromFile(comicpath)) {
            throw FontException(comicpath);
        }

        selectedBox = nullptr;

        sf::RenderWindow window(sf::VideoMode(700, 400), "Path Tracer", sf::Style::Titlebar | sf::Style::Close);

        //Creates all UI elements
        sf::Text title;
        title.setFillColor(sf::Color::White);
        title.setFont(comic);
        title.setCharacterSize(50);
        title.setString("Path Tracer");
        title.setPosition({175, 150});

        sf::Text invalidPath;
        invalidPath.setFillColor(sf::Color::Red);
        invalidPath.setFont(arial);
        invalidPath.setCharacterSize(25);
        invalidPath.setString("Invalid filepath");
        invalidPath.setPosition({175, 250});

        bool error = false;

        Textbox filepathBox(25, sf::Color::White, false, 30, "Enter filename: ");
        filepathBox.setFont(arial);
        filepathBox.setPos({175, 225});

        while(window.isOpen()) {
            sf::Event event;
            //Checks user created events constantly
            while(window.pollEvent(event)) {
                switch(event.type) {
                    case sf::Event::Closed:
                        window.close();
                        break;
                    case sf::Event::MouseButtonPressed:
                        if(filepathBox.onButton(window)) {
                            filepathBox.setSelected();
                            filepathBox.setColor(sf::Color::Green);
                        }
                        break;
                    case sf::Event::TextEntered:
                        filepathBox.typedOn(event);
                        break;
                    case sf::Event::KeyPressed:
                        if(event.key.code == sf::Keyboard::Enter){
                            try
                            {
                                std::string filepath = filepathBox.getInput().substr();
                                FileLoader loader(filepath);
                                window.close();
                                std::shared_ptr<Scene> loadedScene = loader.loadSceneFile();
                                return loadedScene;
                            }
                            catch(FileLoaderException& ex) {
                                error = true;
                            }
                        } else if(event.key.code == sf::Keyboard::Escape){
                            filepathBox.setColor(sf::Color::White);
                        }
                        break;
                    default:
                        break;
                }
            }

            window.clear();
            window.draw(title);
            if(error) {
                window.draw(invalidPath);
            }
            filepathBox.draw(window);
            window.display();

        }
        throw TitleScreenException();
    }


    void openSettings(std::shared_ptr<Scene> loadedScene) {
        sf::Image image;
        sf::Sprite sprite;
        sf::Texture texture;
        sf::Font arial;
        std::string arialpath = "../fonts/Arial.ttf";
        if(!arial.loadFromFile(arialpath)) {
            throw FontException(arialpath);
        }

        sf::RenderWindow window(sf::VideoMode(700, 400), "Path Tracer", sf::Style::Titlebar | sf::Style::Close);

        //Creates all the UI elements
        sf::Vector2f size(200, 100);
        Button preview("Preview", sf::Color::Black, 20, size, sf::Color::Green);
        preview.setFont(arial);
        preview.setPos({0, 300});

        Textbox resXbox(20, sf::Color::White, false, 4, "ResX: ");
        resXbox.setFont(arial);
        resXbox.setPos({0, 0});

        Textbox resYbox(20, sf::Color::White, false, 4, "ResY: ");
        resYbox.setFont(arial);
        resYbox.setPos({0, 25});

        Textbox sampleBox(20, sf::Color::White, false, 4, "Samples: ");
        sampleBox.setFont(arial);
        sampleBox.setPos({0, 50});

        Textbox bounceBox(20, sf::Color::White, false, 4, "Light bounces: ");
        bounceBox.setFont(arial);
        bounceBox.setPos({0, 75});

        Textbox fovBox(20, sf::Color::White, false, 4, "Fov: ");
        fovBox.setFont(arial);
        fovBox.setPos({0, 125});

        Textbox dofBox(20, sf::Color::White, false, 4, "Depth of field: ");
        dofBox.setFont(arial);
        dofBox.setPos({0, 150});

        Textbox focusBox(20, sf::Color::White, false, 4, "Focus distance: ");
        focusBox.setFont(arial);
        focusBox.setPos({0, 175});

        sf::Text errorText;
        errorText.setFont(arial);
        errorText.setFillColor(sf::Color::Red);
        errorText.setCharacterSize(20);
        errorText.setPosition({0, 225});

        selectedBox = nullptr;

        while (window.isOpen())
        {
            sf::Event event;
            //Checks user created events constantly
            while (window.pollEvent(event))
            {
                switch(event.type) {
                    case sf::Event::Closed:
                        window.close();
                        break;
                    case sf::Event::MouseMoved:
                        if(preview.onButton(window)) {
                            preview.setColor(sf::Color::White);
                        }else {
                            preview.setColor(sf::Color::Green);
                        }
                        break;
                    case sf::Event::MouseButtonPressed:
                        //Checks if preview button can be clicked
                        if(preview.onButton(window)) {
                            if(checkIfPosFloat(fovBox.getInput()) && fovBox.getInput() != ""){
                                loadedScene->setFov((M_PI * std::stof(fovBox.getInput()) / 180));
                            }

                            if(checkIfPosFloat(focusBox.getInput()) && focusBox.getInput() != ""){
                                loadedScene->setFocusDist(std::stof(focusBox.getInput()));
                            }

                            Renderer previewCreator(500, 400, loadedScene);

                            if(checkIfPosFloat(dofBox.getInput()) && dofBox.getInput() != ""){
                                previewCreator.setDof(std::stof(dofBox.getInput()));
                            }
                            createImg(previewCreator.parallelRender(1));
                            saveImage("preview.png");
                            image.loadFromFile("preview.png");
                            texture.loadFromImage(image);
                            sprite.setTexture(texture);
                            sprite.setPosition(200, 0);
                            }
                        //checks if any textboxes can be clicked
                        clickBox(resXbox, window);
                        clickBox(resYbox, window);
                        clickBox(sampleBox, window);
                        clickBox(bounceBox, window);
                        clickBox(fovBox, window);
                        clickBox(dofBox, window);
                        clickBox(focusBox, window);
                        break;

                    case sf::Event::TextEntered:
                        fovBox.typedOn(event);
                        resXbox.typedOn(event);
                        resYbox.typedOn(event);
                        dofBox.typedOn(event);
                        sampleBox.typedOn(event);
                        focusBox.typedOn(event);
                        bounceBox.typedOn(event);
                        break;

                    case sf::Event::KeyPressed:
                        if(event.key.code == sf::Keyboard::Enter){
                            int resX;
                            int resY;
                            int sampleSize;
                            int bounceAmount;
                            float dof = 0;

                            if(checkIfPosFloat(fovBox.getInput()) && fovBox.getInput() != ""){
                                loadedScene->setFov((M_PI * std::stof(fovBox.getInput()) / 180));
                            }

                            if(checkIfPosFloat(focusBox.getInput()) && focusBox.getInput() != ""){
                                loadedScene->setFocusDist(std::stof(focusBox.getInput()));
                            }

                            if(checkIfPosFloat(dofBox.getInput()) && dofBox.getInput() != ""){
                                dof = std::stof(dofBox.getInput());
                            }

                            if(checkIfPosNum(resXbox.getInput()) && resXbox.getInput() != ""){
                                resX = std::stof(resXbox.getInput());
                            }else {
                                errorText.setString("Invalid resX");
                                break;
                            }

                            if(checkIfPosNum(resYbox.getInput()) && resYbox.getInput() != ""){
                                resY = std::stof(resYbox.getInput());
                            }else {
                                errorText.setString("Invalid resY");
                                break;
                            }

                            if(checkIfPosNum(sampleBox.getInput()) && sampleBox.getInput() != ""){
                                sampleSize = std::stoi(sampleBox.getInput());
                            }else {
                                errorText.setString("Invalid sample amount");
                                break;
                            }

                            if(checkIfPosNum(bounceBox.getInput()) && bounceBox.getInput() != ""){
                                bounceAmount = std::stoi(bounceBox.getInput());
                            }else {
                                errorText.setString("Invalid bounce amount");
                                break;
                            }

                            selectedBox = nullptr;
                            window.close();
                            openRender(resX, resY, loadedScene, sampleSize, dof, bounceAmount);
                            break;
                        }
                    default:
                        break;

                }
                }
```

```

            window.clear();
            preview.draw(window);
            window.draw(sprite);
            window.draw(errorText);
            fovBox.draw(window);
            resXbox.draw(window);
            resYbox.draw(window);
            focusBox.draw(window);
            dofBox.draw(window);
            sampleBox.draw(window);
            bounceBox.draw(window);
            window.display();

            }

        }

    void openRender(int resX, int resY, std::shared_ptr<Scene> loadedScene, int sampleSize, float dof, int bounces) {
        sf::RenderWindow window(sf::VideoMode(resX, resY), "Path Tracer", sf::Style::Close);
        window.setSize(sf::Vector2u(resX, resY));
        window.setFramerateLimit(0);
        sf::Image image;
        sf::Sprite sprite;
        sf::Texture texture;
        Renderer sceneRenderer(resX, resY, loadedScene);
        sceneRenderer.setMaxBounces(bounces);
        sceneRenderer.setDof(dof);
        std::vector<std::vector<Color» combinedSamples;

        int i = 0;

            while (window.isOpen())
            {
                sf::Event event;
                while (window.pollEvent(event))
                {
                    switch(event.type) {
                        case sf::Event::Closed:
                            window.close();
                            break;
                        default:
                            break;
                    }
                }
                while(i < sampleSize) {
                    auto result = sceneRenderer.parallelRender(1);
                    float weight = 1.0 / (1 + i);
                    if(i == 0) {
                        combinedSamples = result;
                    } else {
                        for (int pixel = 0; pixel < resX * resY; ++pixel)
                        {
                            int x = pixel % resX;
                            int y = pixel / resX;

                            combinedSamples[x][y] = clamp(combinedSamples[x][y] * (1 - weight) + weight * result[x][y]);

                        }
                    }
                    createImg(combinedSamples);
                    saveImage("image.png");
                    image.loadFromFile("image.png");
                    texture.loadFromImage(image);
                    sprite.setTexture(texture);
                    window.clear();
                    window.draw(sprite);
                    window.display();
                    i++;


                }
        }

        }

private:
    Textbox* selectedBox = nullptr;

    bool checkIfPosNum(std::string text) {
        for(auto i : text) {
            if(!std::isdigit(i)) {
                return false;
            }
        }
        return true;
    }

    bool checkIfPosFloat(std::string text) {
        int j = text.length();
        for(auto i : text) {
            if(!std::isdigit(i)) {
                if(j != 2 && std::to_string(text[text.length()-2]) != ".") {
                    return false;
                }
            }
            j--;
        }
        return true;
    }

    void clickBox(Textbox &box, sf::RenderWindow &window) {
        if(box.onButton(window)) {
             if(selectedBox) {
                selectedBox->setUnselected();
                selectedBox->setColor(sf::Color::White);
                }
                box.setSelected();
                box.setColor(sf::Color::Green);
                selectedBox = &box;
                }
    }

    Color clamp(Color input) {
        float R = input(0) > 1 ? 1 : input(0);
        float G = input(1) > 1 ? 1 : input(1);
        float B = input(2) > 1 ? 1 : input(2);
        return Color(R, G, B);
    }
};
```

5.5 gui_ex.hpp

```
#ifndef GUI_EXCEPTION
#define GUI_EXCEPTION

#include <exception>
#include <iostream>
#include <string>

class GuiException : public std::exception {
    public:
        GuiException() {}

        virtual const char* what() const noexcept = 0;

        virtual ~GuiException() = default;
};

class TitleScreenException : public GuiException {
    public:
        TitleScreenException() : GuiException() {
            msg_ = "Title window closed";
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;
};

class FontException : public GuiException {
    public:
        FontException(std::string fontPath) : GuiException() {
            msg_ = "Did not find font in path: " + fontPath;
        }

        virtual const char* what() const noexcept {
            return msg_.c_str();
        }

    private:
        std::string msg_;
};

#endif
```

5.6 interface.hpp

```
#pragma once

#include <SFML/Graphics.hpp>
#include <vector>

#include "types.hpp"

class Interface {
private:
    sf::Image img;

    sf::Uint8 scale(float x) {
        return floor(255*x);
    }

public:

    void createImg(std::vector<std::vector<Color» pixels) {
        int width = pixels.size();
        int height = pixels[0].size();
        img.create(width, height);
        for (int i = 0; i < width; ++i) {
            for (int j = 0; j < height; ++j) {
                const sf::Uint8 R = scale(pixels[i][j](0));
                const sf::Uint8 G = scale(pixels[i][j](1));
                const sf::Uint8 B = scale(pixels[i][j](2));
                img.setPixel(i, j, sf::Color(R, G, B));
            }
        }
    }

    bool saveImage(const std::string &filename) {
        return img.saveToFile(filename);
    }

    sf::Image getImage() { return img; }

};
```

5.7 textbox.hpp

```
#pragma once

#include <iostream>
#include <sstream>
#include <SFML/Graphics.hpp>

#define BACKSPACE_KEY 8
#define ESCAPE_KEY 27
#define TAB_KEY 9
#define ENTER_KEY 11

class Textbox {
public:

    Textbox(int size, sf::Color color, bool isSelected, int limit, std::string preText) {
        textbox_.setCharacterSize(size);
        size_ = size;
        textbox_.setFillColor(color);
        textbox_.setString(preText);
        isSelected_ = isSelected;
        limit_ = limit + preText.length() - 1;
        preTextLength = preText.length();
        text_ « preText;
    }

    ~Textbox() = default;

    void setFont(sf::Font &font) {
        textbox_.setFont(font);
    }

    void setSelected() {
        isSelected_ = true;
    }

    void setUnselected() {
        isSelected_ = false;
    }

    void setPos(sf::Vector2f pos) {
        textbox_.setPosition(pos);
    }

    void setColor(sf::Color textColor) {
        textbox_.setFillColor(textColor);
    }

    void draw(sf::RenderWindow &window) {
        window.draw(textbox_);
    }

    bool onButton(sf::RenderWindow &window) {
            float mousePos_x = sf::Mouse::getPosition(window).x;
            float mousePos_y = sf::Mouse::getPosition(window).y;
            float boxPos_x = textbox_.getPosition().x;
            float boxPos_y = textbox_.getPosition().y;
            float boxSize_x = (size_/1.618)*limit_;
            float boxSize_y = size_;

            bool onBox_x = (mousePos_x >= boxPos_x) && (mousePos_x <= (boxSize_x + boxPos_x));
            bool onBox_y = (mousePos_y >= boxPos_y) && (mousePos_y <= (boxSize_y + boxPos_y));

            return (onBox_x && onBox_y);

        }

    std::string getInput() {
            return text_.str().substr(preTextLength);
        }

    void typedOn(sf::Event input) {
        if(isSelected_ && input.key.code != sf::Keyboard::Enter) {
            int inputChar = input.text.unicode;
            if(inputChar < 128 && inputChar != 0 && inputChar != TAB_KEY && inputChar != ENTER_KEY) {
                if(text_.str().length() <= limit_) {
                    checkInput(inputChar);
                }else if(inputChar == BACKSPACE_KEY) {
                    deleteLastChar();
                }
            }
        }
    }

private:
    sf::Text textbox_;
    std::ostringstream text_;
    bool isSelected_ = false;
    int limit_;
    int preTextLength;
    int size_;

    void checkInput(int inputChar) {
        if(inputChar != BACKSPACE_KEY && inputChar != ESCAPE_KEY) {
            text_ « static_cast<char>(inputChar);
        }
        else if(inputChar == BACKSPACE_KEY) {
            deleteLastChar();
        }
        else if(inputChar == ESCAPE_KEY && isSelected_ == true) {
            setUnselected();
            textbox_.setFillColor(sf::Color::White);
        }
        textbox_.setString(text_.str());
    }

    void deleteLastChar() {
        if(text_.str().length() > preTextLength){
            std::string newTxt = text_.str();
            newTxt.pop_back();
            text_.str(std::string());
            text_ « newTxt;
            textbox_.setString(text_.str());
        }
    }

};
```

5.8 ball.hpp

```
#pragma once

#include "object.hpp"
#include "fileloader_ex.hpp"
```

```
#include "material.hpp"

#include <memory>

class Ball : public Object
{
private:
    float radius_;

public:
    Ball(Vector position, float radius, std::shared_ptr<Material> material) : Object(position, material), radius_(radius) {}

    void collision(Ray& ray, Hit &rayHit, float& smallestDistance) {

        Vector toBall = ray.origin - this->getPosition();

        float a = ray.direction.dot(ray.direction);
        float b = 2 * ray.direction.dot(toBall);
        float c = toBall.dot(toBall) - radius_ * radius_;

        float discriminant = b*b - 4*a*c;

        if (discriminant >= 0)
        {
            float distance = (-b - sqrt(discriminant)) / (2*a);

            if (distance > 0 && distance < smallestDistance)
            {
                smallestDistance = distance;
                rayHit.distance = distance;
                rayHit.material = this->getMaterial();
                rayHit.did_hit = true;
                rayHit.point = ray.origin + ray.direction * distance;
                rayHit.normal = (rayHit.point - this->getPosition()).normalized();
            }
        }

        return;
    }

    float getRadius() const { return radius_; }

    void printInfo(std::ostream& out) const {
        out « "Ball at: (" « this->getPosition().transpose() « ") with radius: " « this->getRadius() « ", with material: " « (*getMaterial()).getName() « std::endl;
    }

};
```

5.9 box.hpp

```
#pragma once

#include "object.hpp"
#include <vector>

class Box : public Object
{
private:
    float width_; /* Associated with y direction */
    float height_; /* Associated with z direction */
    float depth_; /* Associated with x direction */

    std::vector<Vector> corners_;
    /*
    *       4.......7
    *     / :     / :
    *   0-------3   :
    *   |   :   |   :
    *   |   5...|...6
    *   | /     | /
    *   1-------2
    *
    * When looking towards positive x-direction
    */

   std::list<std::vector<int» sides_ = {   {0, 1, 2},
                                            {3, 2, 6},
                                            {7, 6, 5},
                                            {4, 5, 1},
                                            {4, 0, 3},
                                            {1, 5, 6}};

public:
    Box(Vector position, float width, float height, float depth, std::shared_ptr<Material> material)
            : Object(position, material), width_(width), height_(height), depth_(depth) {

        corners_.push_back(Vector(-depth_/2, width_/2, height_/2) + position);
        corners_.push_back(Vector(-depth_/2, width_/2, -height_/2) + position);
        corners_.push_back(Vector(-depth_/2, -width_/2, -height_/2) + position);
        corners_.push_back(Vector(-depth_/2, -width_/2, height_/2) + position);

        corners_.push_back(Vector(depth_/2, width_/2, height_/2) + position);
        corners_.push_back(Vector(depth_/2, width_/2, -height_/2) + position);
        corners_.push_back(Vector(depth_/2, -width_/2, -height_/2) + position);
        corners_.push_back(Vector(depth_/2, -width_/2, height_/2) + position);
    }

    void collision(Ray& ray, Hit &rayHit, float& smallestDistance) {

        // Check first whether ray collides with the bounding ball
        Vector toBall = ray.origin - this->getPosition();
        float radius = width_ * width_ + height_ * height_ + depth_ * depth_;
        float a = ray.direction.dot(ray.direction);
        float b = 2 * ray.direction.dot(toBall);
        float c = toBall.dot(toBall) - radius;
        float discriminant = b*b - 4*a*c;
        if (discriminant < 0) return;

        // If inside ball, check sides
        for (auto side : sides_) {

            Vector topLeft = corners_[side[0]];
            Vector bottomLeft = corners_[side[1]];
            Vector bottomRight = corners_[side[2]];

            Vector s1 = -bottomLeft + bottomRight;
            Vector s2 = -bottomLeft + topLeft;

            Vector normal = s1.cross(s2).normalized();

            float distance = (bottomLeft - ray.origin).dot(normal) / ray.direction.dot(normal);
            Vector intersection = -bottomLeft + ray.origin + ray.direction * distance;

            if (intersection.dot(s1) <= s1.squaredNorm()
                && intersection.dot(s1) >= 0
                && intersection.dot(s2) <= s2.squaredNorm()
                && intersection.dot(s2) >= 0
            )
            {
                if (distance > 0 && distance < smallestDistance)
                {
                    smallestDistance = distance;
                    rayHit.distance = distance;
                    rayHit.material = this->getMaterial();
                    rayHit.did_hit = true;
                    rayHit.point = ray.origin + ray.direction * distance;
                    rayHit.normal = normal;
                }
            }
        }
    }

    float getWidth() const { return width_; }
    float getHeight() const { return height_; }
    float getDepth() const { return depth_; }

    void rotate(float angle, Vector axis) {
        Eigen::AngleAxisd rotation(angle, axis);

        for (auto& corner : corners_) {
            corner = (rotation * (corner - this->getPosition())) + this->getPosition();
        }
    }

    void printInfo(std::ostream& out) const {
        out « "Box at: (" « this->getPosition().transpose() « ") with dimensions: " « width_ « "x" « height_ « "x" « depth_ « ", with material: " « (*getMaterial()).getName() « std::endl;
    }

};
```

5.10 bvh.hpp

```
#pragma once
```

```
#include "types.hpp"
#include "triangle.hpp"
#include <vector>

struct AABB
{
    Vector min, max;
};

struct Node
{
    AABB box;
    int leftChild;
    bool isLeaf() { return triCount > 0; }
    int firstTriIdx, triCount;
};

class BVH
{
public:
    BVH() { }

    BVH(std::vector<Triangle> tris) {
        //Initialize variables
        n = tris.size();
        triangles = tris;
        std::vector<Node> bvhNodes(n*2 - 1);
        rootNodeIdx = 0;
        nodesUsed = 1;
        for(int i = 0; i < n; i++) {
            triIdx.push_back(i);
        }
        Node& root = bvhNodes[rootNodeIdx];
        root.leftChild = 0;
        root.triCount = n;

        //Create the structure
        UpdateNodeBounds(bvhNodes, rootNodeIdx);
        Subdivide(bvhNodes, rootNodeIdx);

        //Copy structure to a private variable
        for(int i = 0; i < bvhNodes.size(); i++) {
            nodes.push_back(bvhNodes[i]);
        }
    }

    void UpdateNodeBounds(std::vector<Node>& bvhNodes, int nodeIdx) {
        Node& currentNode = bvhNodes[nodeIdx];

        //Set bounds to "infinity" in the beginning
        currentNode.box.min = Vector(1e30f, 1e30f, 1e30f);
        currentNode.box.max = Vector(-1e30f, -1e30f, -1e30f);

        //Loop over triangles and set new bounds accordingly
        for(int first = currentNode.firstTriIdx, i=0; i < currentNode.triCount; i++) {
            int leafTriIdx = triIdx[first + i];
            Triangle &leafTriangle = triangles[leafTriIdx];
            std::vector<Vector> vertices = leafTriangle.getVertexPos();
            currentNode.box.min = currentNode.box.min.cwiseMin(vertices[0]);
            currentNode.box.min = currentNode.box.min.cwiseMin(vertices[1]);
            currentNode.box.min = currentNode.box.min.cwiseMin(vertices[2]);
            currentNode.box.max = currentNode.box.max.cwiseMax(vertices[0]);
            currentNode.box.max = currentNode.box.max.cwiseMax(vertices[1]);
            currentNode.box.max = currentNode.box.max.cwiseMax(vertices[2]);
        }
    }

    void Subdivide(std::vector<Node>& bvhNodes, int nodeIdx) {
        //Terminates recursion
        Node &currentNode = bvhNodes[nodeIdx];
        if(currentNode.triCount <= 2) return;

        //Determine the split
        Vector extent = currentNode.box.max - currentNode.box.min;
        int axis = 0;
        if(extent(1) > extent(0)) axis = 1;
        if(extent(2) > extent(axis)) axis = 2;
        float splitPos = currentNode.box.min(axis) + extent(axis)*0.5f;

        //Partition triangles
        int i = currentNode.firstTriIdx;
        int j = i + currentNode.triCount - 1;
        while(i <= j) {
            if(triangles[triIdx[i]].getCentroid()(axis) < splitPos) {
                i++;
            }else {
                std::swap(triIdx[i], triIdx[j--]);
            }
        }

        //Check if one of the sides is empty
        int leftCount = i - currentNode.firstTriIdx;
        if(leftCount == 0 || leftCount == currentNode.triCount) return;

        //create child nodes
        int leftChildIdx = nodesUsed++;
        int rightChildIdx = nodesUsed++;
        bvhNodes[leftChildIdx].firstTriIdx = currentNode.firstTriIdx;
        bvhNodes[leftChildIdx].triCount = leftCount;
        bvhNodes[rightChildIdx].firstTriIdx = i;
        bvhNodes[rightChildIdx].triCount = currentNode.triCount - leftCount;
        currentNode.leftChild = leftChildIdx;
        currentNode.triCount = 0;

        //Update node bounds
        UpdateNodeBounds(bvhNodes, leftChildIdx);
        UpdateNodeBounds(bvhNodes, rightChildIdx);
        //Recursion
        Subdivide(bvhNodes, leftChildIdx);
        Subdivide(bvhNodes, rightChildIdx);
    }

    void BVHCollision(Ray& ray, Hit& rayHit, float& smallestDistance, const int nodeIdx) {
        Node& currentNode = nodes[nodeIdx];
        if(!AABBCollision(currentNode.box, ray, smallestDistance)) return;
        if(currentNode.isLeaf()) {
            for(int i = 0; i < currentNode.triCount; i++) {
                triangles[triIdx[currentNode.firstTriIdx + i]].collision(ray, rayHit, smallestDistance);
            }
        }else {
            BVHCollision(ray, rayHit, smallestDistance, currentNode.leftChild);
            BVHCollision(ray, rayHit, smallestDistance, currentNode.leftChild + 1);
        }
    }

    bool AABBCollision(AABB box, Ray ray, float smallestDistance) {
        float tx1 = (box.min(0) - ray.origin(0))/ray.direction(0);
        float tx2 = (box.max(0) - ray.origin(0))/ray.direction(0);
        float tmin = std::min(tx1, tx2);
        float tmax = std::max(tx1, tx2);

        float ty1 = (box.min(1) - ray.origin(1))/ray.direction(1);
        float ty2 = (box.max(1) - ray.origin(1))/ray.direction(1);
        tmin = std::max(tmin, std::min(ty1, ty2));
        tmax = std::min(tmax, std::max(ty1, ty2));

        float tz1 = (box.min(2) - ray.origin(2))/ray.direction(2);
        float tz2 = (box.max(2) - ray.origin(2))/ray.direction(2);
        tmin = std::max(tmin, std::min(tz1, tz2));
        tmax = std::min(tmax, std::max(tz1, tz2));

        return tmax >= tmin && tmin < smallestDistance && tmax > 0;
    }

    int getRootNodeIdx() const {
        return rootNodeIdx;
    }

    std::vector<Triangle> getTriangles() const {
        return triangles;
    }

    std::vector<Node> getNodes() const {
        return nodes;
    }


private:
    std::vector<Triangle> triangles;
    std::vector<int> triIdx;
    std::vector<Node> nodes;
    int rootNodeIdx;
    int nodesUsed;
    int n;
};
```

5.11 object.hpp

```
#pragma once
```

```

#include "types.hpp"
#include "material.hpp"

#include <memory>

class Object
{
private:
    Vector position_;
    std::shared_ptr<Material> material_;
    std::string name_;

public:
    Object(Vector position, std::shared_ptr<Material> material)
                    : position_(position), material_(material) {}

    virtual ~Object() { }

    Vector getPosition() const { return position_; }
    std::shared_ptr<Material> getMaterial() const { return material_; }

    virtual void collision(Ray& ray, Hit &rayHit, float& smallestDistance) = 0;

    virtual void printInfo(std::ostream& out) const = 0;
};

std::ostream &operator«(std::ostream& out, const Object& object) {
    object.printInfo(out);
    return out;
}
```

5.12 rectangle.hpp

```
#pragma once

#include "object.hpp"
#include <vector>
#include <memory>

class Rectangle : public Object
{
private:
    float width_; /* Associated with y direction */
    float height_; /* Associated with z direction */

    std::vector<Vector> corners_;
    /*
    *
    *   0-------3
    *   |       |
    *   |       |
    *   |       |
    *   1-------2
    *
    * When looking towards positive x-direction
    */

public:
    Rectangle(Vector position, float width, float height, std::shared_ptr<Material> material)
            : Object(position, material), width_(width), height_(height) {

        corners_.push_back(Vector(0, width_/2, height_/2) + position);
        corners_.push_back(Vector(0, width_/2, -height_/2) + position);
        corners_.push_back(Vector(0, -width_/2, -height_/2) + position);
        corners_.push_back(Vector(0, -width_/2, height_/2) + position);
    }

    void collision(Ray& ray, Hit &rayHit, float& smallestDistance) {

        Vector topLeft = corners_[0];
        Vector bottomLeft = corners_[1];
        Vector bottomRight = corners_[2];

        Vector s1 = -bottomLeft + bottomRight;
        Vector s2 = -bottomLeft + topLeft;

        Vector normal = s1.cross(s2).normalized();

        float distance = (bottomLeft - ray.origin).dot(normal) / ray.direction.dot(normal);
        Vector intersection = -bottomLeft + ray.origin + ray.direction * distance;

        if (intersection.dot(s1) <= s1.squaredNorm()
            && intersection.dot(s1) >= 0
            && intersection.dot(s2) <= s2.squaredNorm()
            && intersection.dot(s2) >= 0
        )
        {
            if (distance > 0 && distance < smallestDistance)
            {
                smallestDistance = distance;
                rayHit.distance = distance;
                rayHit.material = this->getMaterial();
                rayHit.did_hit = true;
                rayHit.point = ray.origin + ray.direction * distance;
                rayHit.normal = normal;
            }
        }
    }

    float getWidth() const { return width_; }
    float getHeight() const { return height_; }

    void rotate(float angle, Vector axis) {
        Eigen::AngleAxisd rotation(angle, axis);

        for (auto& corner : corners_) {
            corner = (rotation * (corner - this->getPosition())) + this->getPosition();
        }
    }

    void printInfo(std::ostream& out) const {
        out « "Rectangle at: (" « this->getPosition().transpose() « ") with dimensions: " « width_ « "x" « height_ « ", with material: " « getMaterial()->getName() « std::endl;
    }

};
```

5.13 triangle.hpp

```
#pragma once

#include "types.hpp"
#include "object.hpp"
#include <vector>

class Triangle : public Object
{
public:
    Triangle(Vector v0, Vector v1, Vector v2, std::shared_ptr<Material> m) : Object(v0, m) {
        //Vectors pointing at the vertices
        a = v0;
        b = v1;
        c = v2;

        //Vectors on the triangle plane
        e1 = b-a;
        e2 = c-a;

        //Normal vector of the triangle
        n = e1.cross(e2).normalized();

        //Centroid of the triangle
        centroid = Vector(v0.mean(), v1.mean(), v2.mean());
    }

    void collision(Ray& ray, Hit &rayHit, float& smallestDistance) {
        //Compute determinant
        Vector p = ray.direction.cross(e2);
        float det = e1.dot(p);

        //Check if the ray is on the same plane as the triangle
        if(det == 0) return;

        float invDet = 1 / det;

        //Compute beta
        Vector s = ray.origin - a;
        float beta = s.dot(p)*invDet;

        //If beta < 0 or beta > 1 the ray does not intersect the triangle
        if(beta < 0 || beta > 1) return;

        //Compute gamma
        Vector q = s.cross(e1);
        float gamma = ray.direction.dot(q)*invDet;

        //If gamma < 0 or beta + gamma > 1 the ray does not intersect the triangle
        if(gamma < 0 || beta + gamma > 1) return;

        //Compute t
        float t = e2.dot(q)*invDet;

        //If t < smallestDistance the ray intersects the triangle
        if(t < smallestDistance) {
            smallestDistance = t;
            rayHit.distance = t;
            rayHit.material = this->getMaterial();
            rayHit.did_hit = true;
            rayHit.point = ray.origin + ray.direction*t;
            rayHit.normal = this->n;
        }

        return;
    }

    void printInfo(std::ostream& out) const {
        std::vector<Vector> v = getVertexPos();
        out « "Triangle at: (" « v[0].transpose() « "), (" « v[1].transpose()
        « "), (" « v[2].transpose() « "), with normal: (" « getNormal().transpose()
        « "), with material: " « getMaterial()->getName() « std::endl;
    }

    std::vector<Vector> getVertexPos() const {
        std::vector<Vector> v;
        v.push_back(a);
        v.push_back(b);
        v.push_back(c);
        return v;
    }

    std::vector<Vector> getPlaneVec() const {
        std::vector<Vector> v;
        v.push_back(e1);
        v.push_back(e2);
        return v;
    }

    Vector getNormal() const {
        return n;
    }

    Vector getCentroid() const {
        return centroid;
    }

private:
    Vector a, b, c;
    Vector e1, e2;
    Vector n;
    Vector centroid;
};


```

5.14 trianglemesh.hpp

```
#pragma once

#include "triangle.hpp"
#include "bvh.hpp"
#include "../libs/tiny_obj_loader/tiny_obj_loader.cc"

#include <iostream>
#include <vector>
#include <string>

class TriangleMesh : public Object {
public:
    TriangleMesh(std::string obj_filepath, Vector scenePos, std::shared_ptr<Material> m, Vector rotation, double scale) : Object(scenePos, m) {
        unsigned long pos = obj_filepath.find_last_of("/");
        std::string basepath = obj_filepath.substr(0, pos+1);
        std::string obj_name = obj_filepath.substr(pos+1, obj_filepath.length());

        //Name of the object wihthout .obj
        name = obj_name.substr(0, obj_name.length()-4);

        tinyobj::attrib_t attributes;
        std::vector<Triangle> triangles;
        std::vector<tinyobj::shape_t> objects;
        std::vector<tinyobj::material_t> materials;
        std::string warnings;
        std::string errors;
        std::vector<Vector> vertices;

        //Load data from object file
        bool r = tinyobj::LoadObj(&attributes, &objects, &materials, &warnings, &errors, obj_filepath.c_str(), basepath.c_str());

        //Check for errors in loading the data
        if(!errors.empty()) {
            std::cout « "Error: " « errors « std::endl;
        }
        if(!r) {
            std::cout « "Failed to load object file." « std::endl;
            exit(1);
        }

        //Loop over objects (shapes)
        for(size_t o = 0; o < objects.size(); o++) {
            size_t o_offset = 0;
            //Loop over triangles (faces)
            for(size_t t = 0; t < objects[o].mesh.num_face_vertices.size(); t++) {
                int fv = objects[o].mesh.num_face_vertices[t];

                //Looping over vertices in this face
                for(size_t v = 0; v < fv; v++) {
                    tinyobj::index_t idx = objects[o].mesh.indices[o_offset + v];
                    tinyobj::real_t vx = attributes.vertices[3*idx.vertex_index+0];
                    tinyobj::real_t vz = attributes.vertices[3*idx.vertex_index+1];
                    tinyobj::real_t vy = attributes.vertices[3*idx.vertex_index+2];

                    //Orientating the object according to the rotation vector
                    Eigen::AngleAxisd xRotation(rotation[0], Vector::UnitX());
                    Eigen::AngleAxisd yRotation(rotation[1], Vector::UnitY());
                    Eigen::AngleAxisd zRotation(rotation[2], Vector::UnitZ());

                    Vector relVertex = Vector(vx, vy, vz);

                    relVertex = xRotation * relVertex;
                    relVertex = yRotation * relVertex;
                    relVertex = zRotation * relVertex;

                    //Creating one vertex
                    Vector vertex = relVertex*scale+scenePos;
                    vertices.push_back(vertex);
                }
                o_offset += fv;
            }
        }

        //Loops over vertices and creates triangles
        for(int i = 0; i < vertices.size()/3; ++i) {
            triangles.push_back(Triangle(vertices[i*3], vertices[i*3+1], vertices[i*3+2], m));
        }

        bvh = BVH(triangles);

        std::cout « "Object file: " « obj_name « ", succesfully opened!" « std::endl;

        triangles.clear();
        objects.clear();
        materials.clear();
    }

    void collision(Ray& ray, Hit& rayHit, float& smallestDistance) {

        bvh.BVHCollision(ray, rayHit, smallestDistance, bvh.getRootNodeIdx());

        return;
    }

    void printInfo(std::ostream& out) const {
        out « "TriangleMesh object: " « name « ", at :" « this->getPosition().transpose() « ", with material: " « (*getMaterial()).getName() « std::endl;
    }

    BVH getBVH() const {
        return bvh;
    }

    std::string getName() const {
        return name;
    }

```

```
private:
    BVH bvh;
    std::string name;
};
```

5.15 renderer.hpp

```
#pragma once

#include "types.hpp"
#include <vector>
#include "randomgenerator.hpp"
#include <iostream>
#include <omp.h>
#include <chrono>
#include <memory>

class Renderer
{
private:

    std::shared_ptr<Scene> scene_;
    Camera camera_;
    RandomGenerator rnd_;

    int resolution_x;
    int resolution_y;
    std::vector<std::vector<Color» result;

    float anti_alias_radius = 1;

    int max_bounces = 5; // Default value if anyone do not change

    float view_width;
    float view_height;

    Vector topleft_pixel;
    Vector pixel_x;
    Vector pixel_y;

    int progressBarWidth = 100;

    Color clamp(Color input) {
        float R = input(0) > 1 ? 1 : input(0);
        float G = input(1) > 1 ? 1 : input(1);
        float B = input(2) > 1 ? 1 : input(2);
        return Color(R, G, B);
    }

    Ray createRay(int x, int y) {

        // Depth of field effect randomizes the origin
        Vector2 jiggle = rnd_.randomInCircle() * camera_.DoF;
        Vector randomShift = jiggle(0) * pixel_x + jiggle(1) * pixel_y;
        Point origin = camera_.position + randomShift;

        // Anti-aliasing randomizes the target
        jiggle = rnd_.randomInCircle() * anti_alias_radius;
        randomShift = jiggle(0) * pixel_x + jiggle(1) * pixel_y;
        Vector target = topleft_pixel + pixel_y * y + pixel_x * x + randomShift;

        Vector direction = (target-origin).normalized();

        return Ray{.origin = origin, .direction = direction};
    }

    Hit rayCollision(Ray& ray) {

        float closestHit = INFINITY;
        Hit rayHit = { .did_hit = false };

        for (auto object : (*scene_).getObjects()) {
            object->collision(ray, rayHit, closestHit);
        }

        return rayHit;
    }

    Light trace(Ray& ray) {
        for (int bounce = 0; bounce < max_bounces; ++bounce)
        {
            Hit hit = rayCollision(ray);

            if (hit.did_hit && hit.distance > 0.0001) {
                // Update ray according to material properties
                (*hit.material).updateRay(ray, hit);
            }
            else
            {
                ray.light += (*scene_).getEnvironment().getLight(ray).cwiseProduct(ray.color);
                break;
            }
        }
        return ray.light;
    }

    void progressBar(int sample, int samples) {
        std::cout « "[";
        int pos = progressBarWidth * (sample + 1)/samples;
        for (int i = 0; i < progressBarWidth; ++i) {
            if (i < pos) std::cout « "=";
            else if (i == pos) std::cout « ">";
            else std::cout « " ";
        }
        std::cout « "] " « std::round((sample + 1) * 100.0 / samples) « " %\r";
        std::cout.flush();
    }

public:

    Renderer(int res_x, int res_y, std::shared_ptr<Scene> sceneToRender) {
        resolution_x = res_x;
        resolution_y = res_y;
        result = std::vector<std::vector<Color»(resolution_x, std::vector<Color> (resolution_y));
        scene_ = sceneToRender;
        camera_ = (*scene_).getCamera();
        view_width = camera_.focus_distance * tan(camera_.fov / 2);
        view_height = view_width * (resolution_y - 1) / (resolution_x - 1);
        pixel_x = -2 * view_width / (resolution_x - 1) * camera_.left;
        pixel_y = - 2 * view_height / (resolution_y - 1) * camera_.up;
        topleft_pixel = camera_.position + camera_.focus_distance * camera_.direction + view_width * camera_.left + view_height * camera_.up;
    }

    void setMaxBounces(int bounces) {
        max_bounces = bounces;
    }

    auto parallelRender(int samples) {

        auto startTime = std::chrono::high_resolution_clock::now();
        std::vector<std::vector<Color» result(resolution_x, std::vector<Color> (resolution_y));

        std::cout « "Rendering started..." « std::endl;

        for (int sample = 0; sample < samples; ++sample)
        {
            float weight = 1.0 / (sample + 1);

            #pragma omp parallel for num_threads(omp_get_max_threads())
            for (int x = 0; x < resolution_x; ++x)
            {
                for (int y = 0; y < resolution_y; ++y)
                {
                    Ray ray = createRay(x, y);
                    Light totalLight = trace(ray);
                    result[x][y] = clamp(result[x][y] * (1 - weight) + weight * totalLight.cwiseSqrt());
                }
            }
            progressBar(sample, samples);
        }

        std::cout « std::endl;
        auto endTime = std::chrono::high_resolution_clock::now();
        std::chrono::duration<float> duration = endTime - startTime;
        std::cout « "Used " « omp_get_max_threads() « " threads.\n" « std::endl;
        std::cout « "Rendering completed in " « duration.count() « " seconds.\n" « std::endl;

        return result;
    }

    void setDof(float dof) {
        camera_.DoF = dof;
    }
};
```

5.16 environment.hpp

```
#pragma once

#include "types.hpp"

class Environment
{
private:
    Color skyColor_;
    Color horizonColor_;
    Color groundColor_;
public:

    Environment() : skyColor_(Color(0, 0, 0)), horizonColor_(Color(0, 0, 0)), groundColor_(Color(0, 0, 0)) {}

    void setSky(Color skyColor = Color(0.2, 0.5, 1.0), Color horizonColor = Color(0.7, 0.8, 0.8), Color groundColor = Color(0.1, 0.1, 0.1)) {
        skyColor_ = skyColor;
        horizonColor_ = horizonColor;
        groundColor_ = groundColor;
    }

    Light getLight(Ray& ray) {

        if (ray.direction(2) >= 0)
        {
            return horizonColor_ + (skyColor_ - horizonColor_) * pow(abs(ray.direction(2)), 0.8);
        }
        else
        {
            return horizonColor_ + (groundColor_ - horizonColor_) * pow(abs(ray.direction(2)), 0.1);
        }
    }

    // Getters
    Color getHorizonColor() { return horizonColor_; }
    Color getGroundColor() { return groundColor_; }
    Color getSkyColor() { return skyColor_; }
};
```

5.17 material.hpp

```
#ifndef MATERIAL_CLASS
#define MATERIAL_CLASS

#include "types.hpp"
#include "randomgenerator.hpp"

#include <vector>

class Material {
    private:
        Color color_;
        std::string name_;

    public:
        RandomGenerator rnd_;

        Color getColor() { return color_; }

        std::string getName() { return name_; }

        Material(Color color, std::string name) : color_(color), name_(name) {}

        Vector reflectionDir(Ray& ray, Hit& hit) {
            return ray.direction - 2 * ray.direction.dot(hit.normal) * hit.normal;
        }

        Vector diffuseDir(Hit& hit) {
            return (rnd_.randomDirection() + hit.normal).normalized();
        }

        void updateColor(Ray& ray) {
            ray.color = ray.color.cwiseProduct(getColor());
            return;
        }

        virtual void updateRay(Ray& ray, Hit& hit) = 0;
};

class Diffuse : public Material {
    private:

        bool emitting_;
        Color emission_color_;
        float emission_strength_;

        void diffuseEmission(Ray& ray) {
            if (!isEmitting()) { return; } // If not emitting, do nothing
            else {
                ray.light += (getEmStrength() * getEmColor()).cwiseProduct(ray.color);
                return;
            }
        }

    public:

        Diffuse(Color color, std::string name, float emission_strength, Color emission_color) :
        Material(color, name), emission_strength_(emission_strength), emission_color_(emission_color) {
            // Boolean value, to determine if material is emitting
            (emission_strength > 0) ? (emitting_ = true) : (emitting_ = false);
        }

        Color getEmColor() { return emission_color_; }

        float getEmStrength() { return emission_strength_; }

        bool isEmitting() { return emitting_; }

        void updateRay(Ray& ray, Hit& hit) {
            ray.origin = hit.point;
            Vector diffused_dir = diffuseDir(hit);
            ray.direction = diffused_dir;
            updateColor(ray);
            diffuseEmission(ray);
            return;
        }
};

class Reflective : public Material {
    private:

        float specularity_;

    public:

        Reflective(Color color, std::string name, float specularity) :
            Material(color, name), specularity_(specularity) {}

        float getSpecularity() { return specularity_; }

        void updateRay(Ray& ray, Hit& hit) {
            ray.origin = hit.point;
            updateColor(ray);
            Vector reflectedRay = reflectionDir(ray, hit);
            Vector diffusedRay = diffuseDir(hit);
            // Weight direction of the reflection based on specularity
            ray.direction = diffusedRay + getSpecularity() * (reflectedRay - diffusedRay);
            return;
        }
};

class ClearCoat : public Reflective {
    private:

        float clearcoat_;
        Color clearcoat_color_;

        bool clearCoatBounce() { return (clearcoat_ >= rnd_.randomZeroToOne()); }

        void updateColor(Ray& ray, bool bounce) {
            ray.color = ray.color.cwiseProduct(bounce ? getClearCoatColor() : getColor());
            return;
        }


    public:

        Color getClearCoatColor() { return clearcoat_color_; }

        float getClearCoat() { return clearcoat_; }

        ClearCoat(Color color, std::string name, float specularity, float clearcoat, Color clearcoat_color) :
            Reflective(color, name, specularity), clearcoat_(clearcoat), clearcoat_color_(clearcoat_color) {}

        void updateRay(Ray& ray, Hit& hit) {
            bool bounce = clearCoatBounce();
            ray.origin = hit.point;
            updateColor(ray, bounce);
            Vector diffused_dir = diffuseDir(hit);
            if (bounce) {
                ray.direction = diffused_dir + getSpecularity() * (reflectionDir(ray, hit) - diffused_dir);
            }
            else {
                ray.direction = diffused_dir;
            }
            return;
        }
};

class Refractive : public Material {
    private:
        float refraction_ratio_; // From outside to inside the material

        // Theta is angle between incoming ray and surface normal (vectors has to be normalized)
        float cosTheta(Vector a, Vector b) {
            return a.dot(b);
        }

        Vector refractionPerpendicular(Vector in, Vector normal, float ref_ratio) {
            float cos_theta = cosTheta(-in, normal);
            return ref_ratio * (in + cos_theta * normal);
        }

        Vector refractionDir(Ray& ray, Hit& hit, float ref_ratio) {
            Vector perpendicular = refractionPerpendicular(ray.direction, hit.normal, ref_ratio);
            Vector parallel = -sqrt(1 - perpendicular.dot(perpendicular)) * hit.normal;
            return perpendicular + parallel;
        }

        float reflectance(float cos_theta, float ref_ratio) {
            float r0 = (1 - ref_ratio) / (1 + ref_ratio);
            r0 = r0 * r0;
            return r0 + (1 - r0) * pow((1 - cos_theta), 5);
        }

    public:

        Refractive(Color color, std::string name, float refraction_ratio) :
            Material(color, name), refraction_ratio_(refraction_ratio) {}

        void updateRay(Ray& ray, Hit& hit) {
            ray.origin = hit.point;
            updateColor(ray);

            // Real refraction ratio depends on the direction
            float ref_ratio = ray.inside_material ? 1 / refraction_ratio_ : refraction_ratio_;

            // Real glass reflects depending on the intersection angle and refraction ratio
            float cos_theta = cosTheta(-ray.direction, hit.normal);
            float sin_theta = sqrt(1 - cos_theta*cos_theta);
            bool must_reflect = ref_ratio * sin_theta > 1;
            float reflectanceProb = reflectance(cos_theta, ref_ratio);

            if (must_reflect || reflectanceProb > rnd_.randomZeroToOne()) {
                // Reflects
                Vector reflectedDir = reflectionDir(ray, hit);
                ray.direction = reflectedDir;
            } else {
                // Refracts
                Vector refractedDir = refractionDir(ray, hit, ref_ratio);
                ray.inside_material = !(ray.inside_material);
                ray.direction = refractedDir;
            }
            return;
        }
};

#endif
```

5.18 scene.hpp

```
#pragma once

#include <memory>
#include <list>
#include <iostream>
```

```
#include "ball.hpp"
#include "types.hpp"
#include "environment.hpp"

class Scene
{
private:
    std::list<std::shared_ptr<Object» objects_;
    Camera camera_;
    Environment environment_;

public:
    Scene() = default;

    Scene(Camera camera, std::list<std::shared_ptr<Object» objects) : camera_(camera), objects_(objects) {}


    ~Scene() {}

    // Default copying for now - list is copied (are the objects copied as well?)
    Scene& operator=(const Scene& that) = default;
    Scene(const Scene& that) = default;

    Camera getCamera() const { return camera_; }

    Environment& getEnvironment() { return environment_; }

    std::list<std::shared_ptr<Object» getObjects() const { return objects_; }

    void setFov(float fov) {
        camera_.fov = fov;
    }

    void setFocusDist(float dof) {
        camera_.focus_distance = dof;
    }

    friend std::ostream &operator«(std::ostream& out, const Scene& scene) {
        out « "\n======================================== SCENE INFORMATION ========================================" « std::endl;

        for (auto object : scene.objects_)
        {
            out « (*object);
        }

        out « std::endl;

        out « "Camera at: (" « scene.camera_.position.transpose() « ") looking at point: ("
            « scene.camera_.lookingAt.transpose() « ") with FOV: " « scene.camera_.fov / M_PI * 180
            « " degrees, DOF: " « scene.camera_.DoF « " and focus distance; " « scene.camera_.focus_distance « "." « std::endl;

        out « "==================================================================================================\n" « std::endl;

        return out;
    }
};
```

5.19 randomgenerator.hpp

```
#pragma once

#include <random>
#include "types.hpp"

class RandomGenerator
{
private:

    std::random_device randomDevice; /* Random numbers provided by the OS */
    std::mt19937 randomInt; /* Uniformly distributed random integers */
    std::uniform_real_distribution<float> randZeroToOne; /* Uniformly distributed reals between 0 and 1 */
    std::normal_distribution<float> normal;  /* Random numbers satisfying the standard normal distribution */

public:
    RandomGenerator() : randomInt(randomDevice()), randZeroToOne(0, 1), normal(0, 1) {}
    ~RandomGenerator() = default;

    Vector randomDirection() {
        return Vector(normal(randomInt), normal(randomInt), normal(randomInt)).normalized();
    }

    float randomZeroToOne() { return randZeroToOne(randomInt); }

    Vector2 randomInCircle() {
        float angle = randZeroToOne(randomInt) * 2 * M_PI;
        float distance = randZeroToOne(randomInt);

        return Vector2(cos(angle), sin(angle)) * sqrt(distance);
    }
};
```

5.20 types.hpp

```
#pragma once

#include <Eigen/Dense>
#include <string>
#include <memory>

typedef Eigen::Vector3d Vector;
typedef Eigen::Vector3d Point;
typedef Eigen::Vector3d Color;
typedef Eigen::Vector3d Light;
typedef Eigen::Vector2d Vector2;
typedef Eigen::Matrix<double, 3, 3> Matrix;

// Forward declaration for Material class, such that the Hit struct knows the existence
class Material;

struct Camera
{
    Point position;
    Vector lookingAt;
    Vector direction;
    Vector up;
    Vector left;
    float fov;
    float focus_distance;
    float DoF;
};

struct Ray
{
    Point origin;
    Vector direction;
    bool inside_material = false;
    Color color = Color(1.0, 1.0, 1.0);
    Light light = Color(0.0, 0.0, 0.0);
};

struct Hit
{
    bool did_hit = false;
    std::shared_ptr<Material> material; // Has to be pointer, since compiler do not yet know anything about Material class
    Vector normal;
    Point point;
    float distance;
};
```

Index