CG extra features report Group 145

Ksawery Radziwiłowicz 6005128, Nicolae Mario-Alexandru 5988543, Shahar Katz 5958318

January 2025

1 Environment maps

1.1 Description

This feature makes it possible to add an environment map to the scene. It's possible to put the cube map between 1 and 50 units away from the center of the scene or at an infinite distance (the infinite distance is used when the distance in the interface is set to the maximum - 50 units). A thing that's relevant for the whole feature and that's important to notice here is that while OpenGL texture space's origin is in the bottom left corner, the `Image` class considers the pixel at coordinates (0, 0) to be the top left pixel. This leads to flipping the value of the v coordinate in the texture space all over the place.

To find texture coordinates for a cube map at a finite distance, I find intersection point between the given ray and an AABB representing the cube map. Then I check which face has been hit and sample the appropriate texture.

For the infinite distance version, I use a method described in chapter 20.4 of Hughes' "Computer graphics: principles and practice" (3rd edition, ISBN 9780133373721); I find the longest component of the ray's direction vector (x, y or z), determine which face to sample from and which axes to use as UV axes in the texture space based on that component and its sign, and finally normalize those axes.

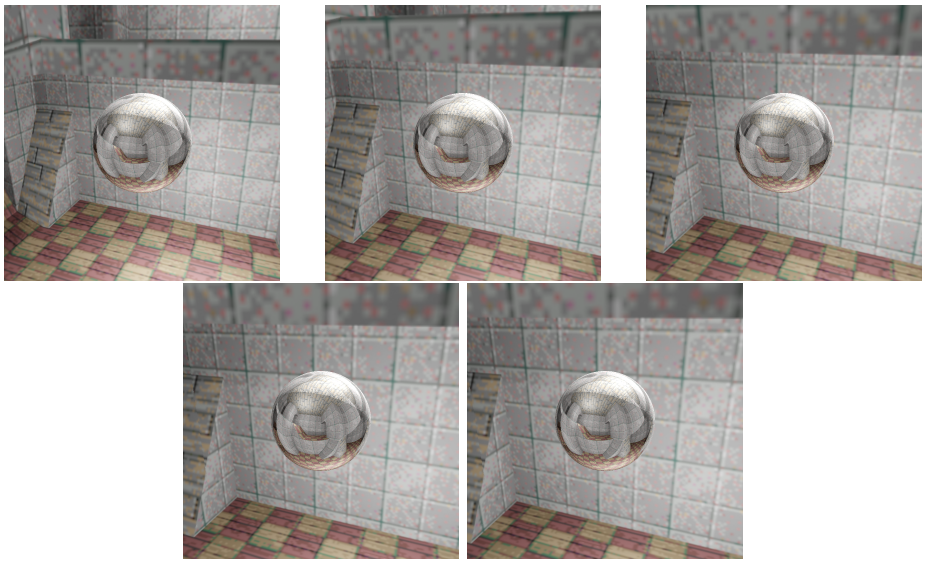


Figure 1: Debug scene with cube map at distances 5, 10, 20, 40 and ∞ respectively. The textures for the cube map were taken from `https://commons.wikimedia.org/wiki/File:Cube_map_level.png`.

1.2 Visual debug & reflection

1.2.1 Debug draw

With debug draw enabled, when a ray that doesn't hit any mesh or sphere is cast, its colour is changed to the colour of the environment map and a cube representing the map is displayed.

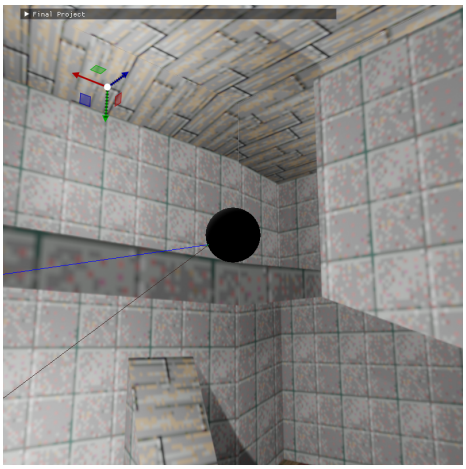


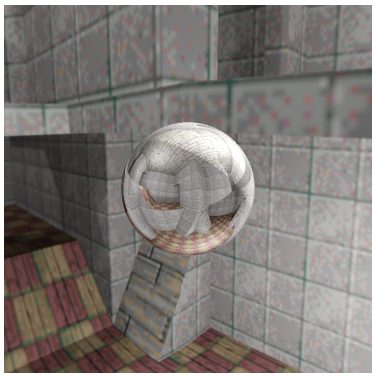
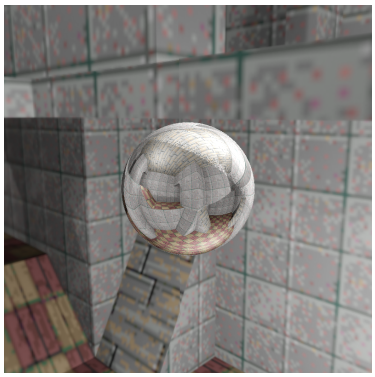
Figure 2: Debug draw with the cube map and coloured reflected ray (brown) visible.

1.2.2 Cube map size comparison

As can be seen in the figure 1, while changing the size of the cube map greatly affects the background, it doesn't affect the reflection in the sphere that much. The biggest difference in the reflections can be seen between sizes 5 and 10, despite the fact that the absolute difference between the sizes of the cube maps there is the smallest.

1.2.3 Finite vs infinite cube map

I've experimented with various cube maps of various sizes and some of the results are shown below. I've noticed that a cube map of finite size generally works better for closed spaces (e.g. rooms), while infinite size works better for open spaces (beaches, parks, etc.).

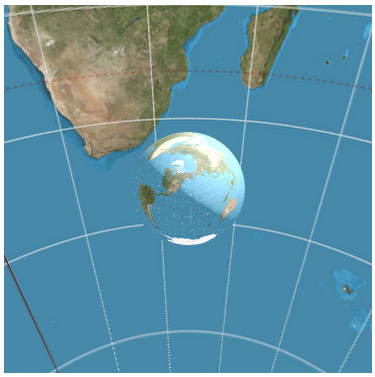
(a) Environment map used earlier at distance 10 and ∞. The finite distance captures a bigger part of the room and therefore makes it look realistically-sized, unlike the infinite distance, which makes it look too big.


(b) Environment map from `https://www.humus.name/index.php?page=Textures&ID=100` at distance 10 and ∞. In the finite distance version it's possible to clearly see corners of the cube map at certain angles; this issue disappears when switching to the map of infinite size.

Figure 3: Comparison of cube maps of finite and infinite sizes.

1.2.4 Alternative environment map shape

A shape alternative to a cube could be a sphere. A suitable application case could be rendering a scene with environment map that's supposed to be a sphere with parallel lines on it. An texture for a sphere map with appropriate projection can actually keep the lines parallel, while textures for a cube map would need to make the lines bent, leading to lower quality with a similar resolution.



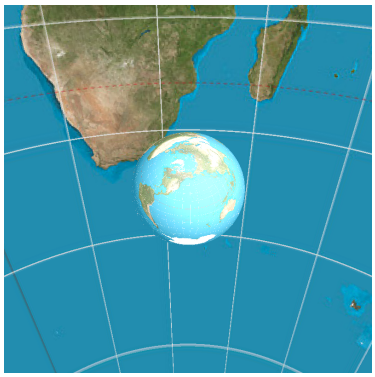


Figure 4: left - cube map, right - sphere map; despite both maps having similar resolution, latitude lines on the sphere map appear much smoother (source of the textures: `https://commons.wikimedia.org/wiki/File:Earth_cube_map.png` and `https://upload.wikimedia.org/wikipedia/commons/8/83/Equirectangular_projection_SW.jpg`).

1.3 Location

This feature is implemented in the file `src/extra.cpp` in functions:

- getCubeMapCoordinates
- intersectRayWithAABBEnvMap
- finiteCubeSampler
- infiniteCubeSampler
- infiniteSphereSampler
- drawSquare
- sampleEnvironmentMap

Additionally, fields `cubeMapSize` and `useSphereMap` were added to `ExtraFeatures` struct in (`src/common.h`) and `cubeMap` and `sphereMap` to `Scene` struct in `src/scene.h` along with a new scene type (`EnvironmentMapDebug`). Finally, a few elements were added to ImGui interface (`main` function in `src/main.cpp`).

2 Motion blur

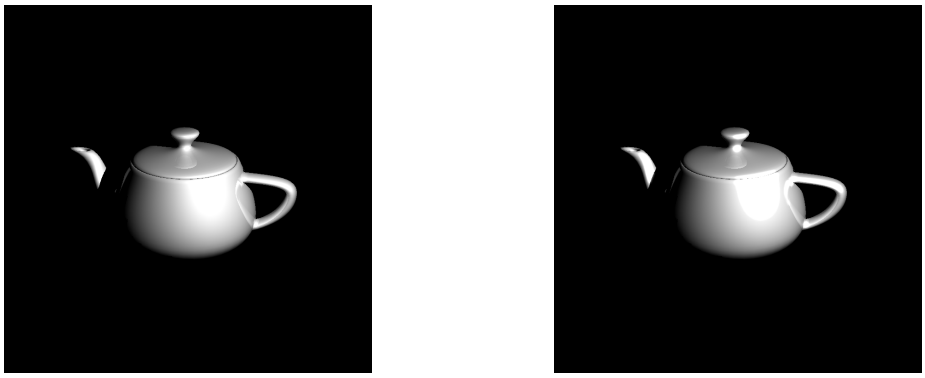
We didn't implement this extra feature.

3 Bloom filter post-processing

3.1 Description

For this feature, we implemented a function that post-processes a rendered image with the bloom filter. Additionally, visual debug functionality to render just bright pixels above the threshold or just the bloom contribution were implemented.

In the basic version, the size of the Gaussian filter used to blur bright pixels can be chosen, as well as lower brightness threshold for pixels to be processed. The brightness is calculated from RGB values according to the formula used in ITU-R Recommendation 709 (item 3.2 of `https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.709-6-201506-I!!PDF-E.pdf`).



(a) Utah teapot rendered with no bloom and with bloom (brightness threshold 0.98, size 9 of Gaussian filter).

(b) Brightness mask and bloom contribution for the image above.

Figure 5: Basic render with bloom filter.

More advanced features include setting upper threshold for brightness of pixels to map, using functions other than binary cutoff to choose bright pixels

and using custom filters loaded from a file instead of a Gaussian filter.

To enable the use of a gray scale image as a bloom filter, a new check box was added to *features*, that if selected, an image could be uploaded. When uploaded, the function `grayScaleFilter` is called. Here, for every pixel of the grayscale the luminance is calculated and normalized. For each pixel of the rendered image, the filter is centered with contributions from the neighboring pixels based on the dimensions of the gray scale image added to it. To enable the use of a non-binary threshold, a few more check boxes were added to *features* that enable different threshold functions. The two functions experimented with were square root mapping and linear mapping. The square root function compressed the high brightness and expanded the low, making it bloom more in darker area as well and more evenly, looking much more realistic than the simple linear mapping.

3.2 Visual debug

Figure 6: Bloom with gray scale filter. Smoother than Gaussian.

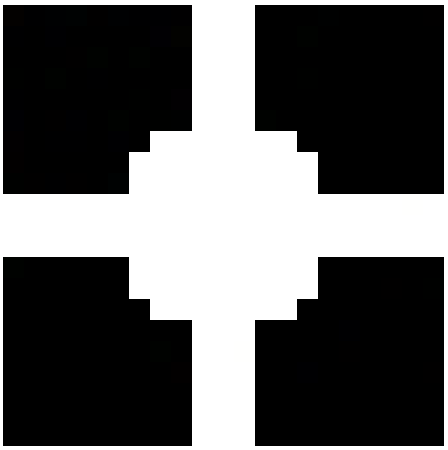


Figure 7: The gray scale filter used.

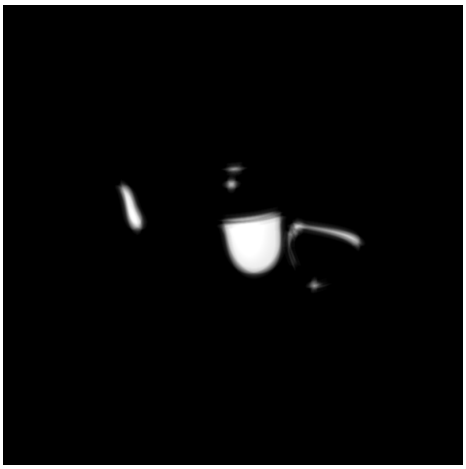


Figure 8: Bloom contribution of gray scale filter.

Figure 9: Bloom with threshold mapping function $y = x$.

Figure 10: Bloom with threshold mapping function $y = \sqrt{x}$.

3.3 Location

This feature is implemented in the file `src/extra.cpp` in functions:

- binomialCoefficient
- gaussianFilter
- luminance
- grayScaleFilter

- postprocessImageWithBloom

Additionally, fields `bloomFilterSize`, `bloomRenderType`, `enableGrayScale`, `bloomGrayscaleFilter`, `thresholdMin`, `thresholdMax` and `bloomFunctionType` were added to `ExtraFeatures` struct in (`src/common.h`). Additionally, a few elements were added to ImGui interface (`main` function in `src/main.cpp`).

4 Glossy reflections

4.1 Description

the function `renderRayGlossyComponent` generates a reflected ray. Then it finds the orthonormal basis to it by calling `calculateBasis`. After that, the function makes a glossy reflected ray, which is an offset of the original reflected ray in a random place on a disk made from the basis. This disk is made by calculating a random value for the angle and radius of the sampled location. It then recursively does this by calling `renderRay`. Doing this for a given number of samples. Lastly it adds the avarage of all the colored samples to hit color after multiplying it by ks. The function `calculateBasis` works by finding the smallest value in the basisW and changing it to 1. Using the normalized cross product of this vector with the original basisW to get another basis vector, a method given from the book.

4.2 Debug

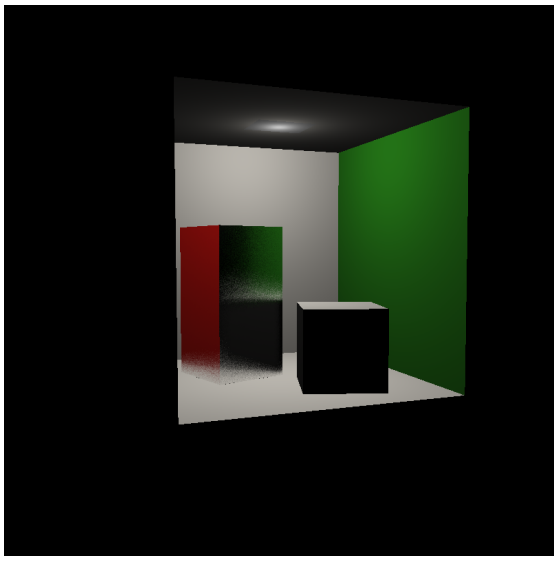


Figure 11: Glossy with 30 samples

The function `allFourVerificationsVisualDebug` helps us verify all correctness of the implementation.

1. Orthogonal Basis Verification

- The orthogonal basis vectors (U, V, W) are visualized at the intersection point. The lines are colored with red, green and blue.
- We can see from the picture that these vectors form a valid right-handed coordinate system orthogonal to the reflection direction.

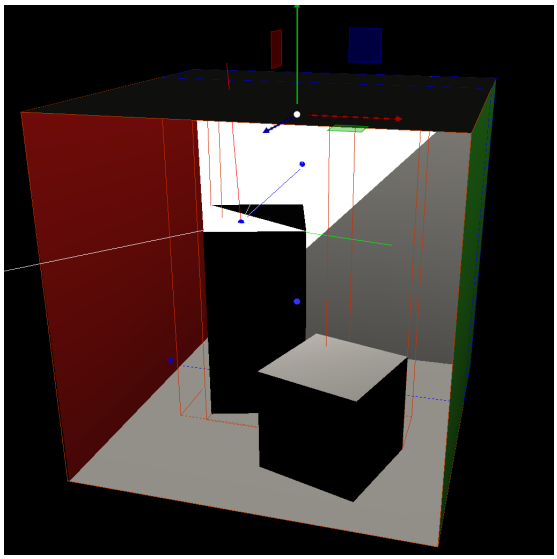


Figure 12: Orthogonal basis

2. Disk Sample Distribution

- The samples on the reflection disk are visualized as blue points, ensuring uniform distribution.
- Sampling is also achieved by randomizing radius and angle to simulate a natural spread.

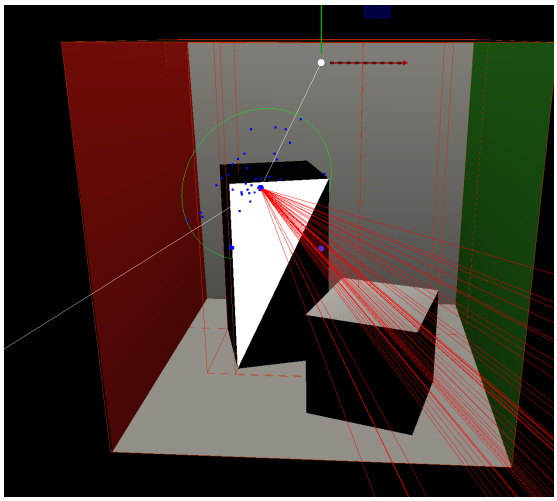


Figure 13: Disk Sample

3. Reflection Ray Bundle

- The bundle of secondary reflection rays is visualized using white lines originating from the calculated intersection point.
- These rays represent the recursive evaluation of glossy reflections.

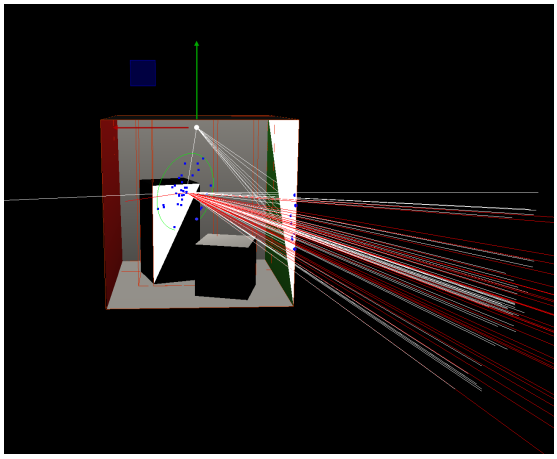


Figure 14: Reflection Ray

4. Perfect Reflection Direction

- I drew a pink line to indicate the perfect reflection direction, to evaluate the bundle's centering.

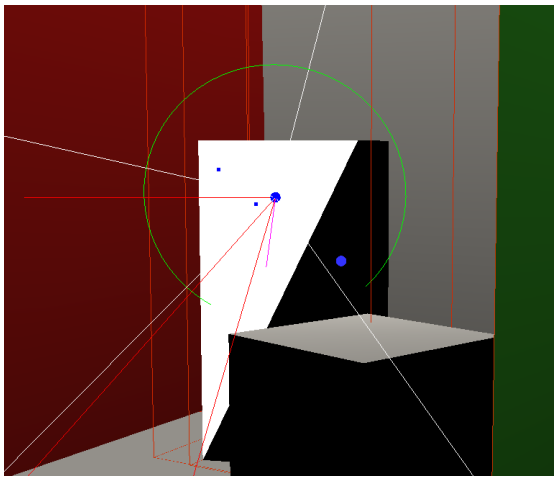


Figure 15: Perfect Reflection

5. Spread Evaluation

- The green circle represents the disk visually showing its size.
- Higher shininess values shrink the disk radius, narrowing the spread of the reflection ray bundle. The Choice For Reducing Disk Size: The disk radius is inversely proportional to the material's shininess parameter. This approach ensures that higher shininess results in tighter reflections, as in the real world.

4.3 Location

This feature is implemented in the file `src/extra.cpp` in functions:

- `renderRayGlossyComponent`
- `calculateBasis`
- `allFourVerificationsVisualDebug`

5 Depth of field

5.1 Description

The Depth of Field feature implements a thin less camera model to simulate objects in and out of focus. Each pixel generates multiple rays sampled through lens, which converge at a focal point on the focal plane. The lens radius, focal distance and focal length are adjustable. Debugging includes visualizing focal points, lens points, image plane points and rays to see that everything works.

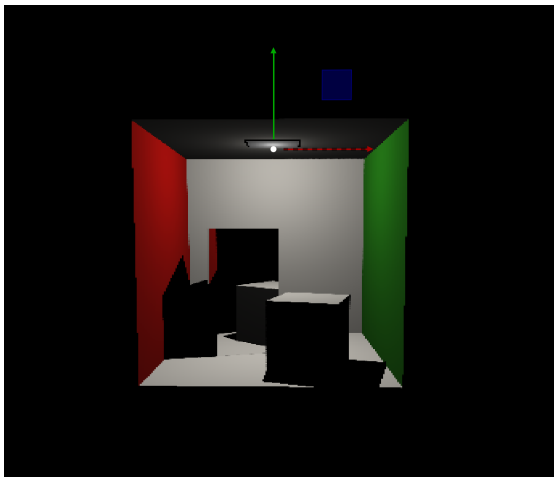


Figure 16: Dof

5.2 Debug

You are best able to see the Debug Visualization, if select BVH, Reflections, Shading with Phong and Shadows, as well as Depth of Field. After selecting these, you can press render to file, wait for it to render and then move the camera. You would be able to see the pink rays. At the bottom section of the window Final Project you can click on the Focal Points button, so that the spheres would appear. I also calculated and verified that the points are on the grid and it works, but I couldn't make it appear due to some issues. I also added some Adjustable parameters with UI Sliders in main.cpp.

- Lens radius (lr): Larger radius increases blur for out-of-focus ares.
- Focal distance (fd): Sets the distance to the focal plane.
- Focal length (fl): Modulates the focus characteristics, affecting how object appear.

1. Focal Plane Points

- This is verified with debug visualization, red spheres at intersection points
- I calculated the points to lie on the focal plane grid.

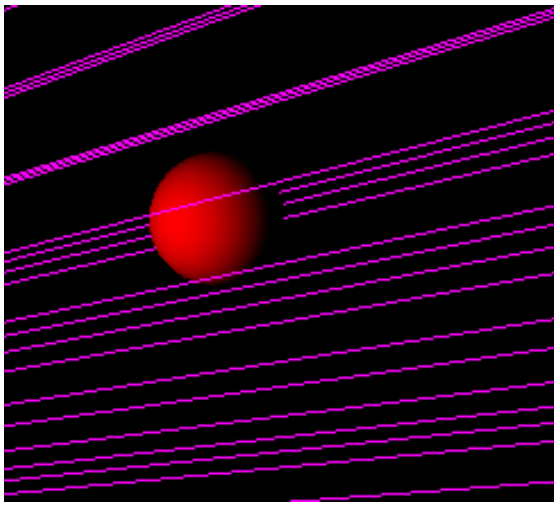


Figure 17: Focal Sphere

2. Lens Sampling Points

- A blue sphere illustrates sampling on the lens disk.
- Uniform distribution ensures consistent dof.

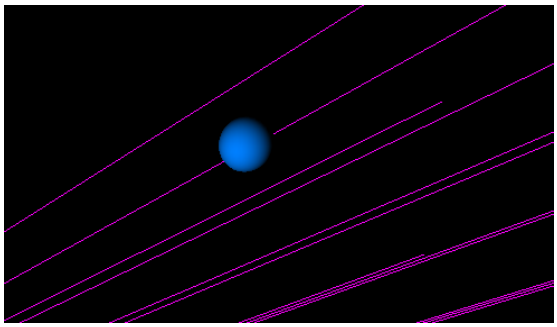
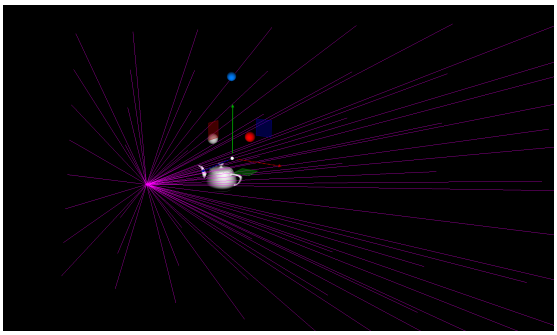


Figure 18: Lens Sphere

3. Image Points

- Points on the image plane help verify the relationship between the lens, rays and focal points.



4. Rays

- Visualized as lines from the lens to focal points for debugging ray convergence.

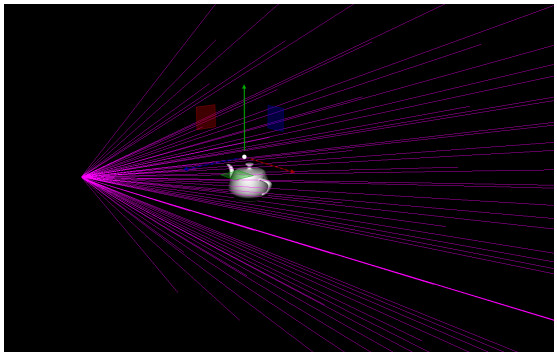


Figure 19: Computed Rays

5.3 Location

This feature is implemented in the file `src/extra.cpp` in functions:

- `renderImageWithDepthOfField`
- `takefp`
- `takelens`
- `takeimage`
- `takerays`

I also added some parameters in ExtraFeatures in the file `common.h`.

- `lr`

- `fd`
- `fl`

This feature is implemented in the file `src/main.cpp` in functions:

- `createSphere`
- `drawing`
- `main`