**Report**

Both are implementations of a ray tracing algorithm, but they have some differences in functionality and structure. Let's go through the main differences:

Here the first script is old\_render.py, and the second script is recursice\_render.py

1. **Ray Bounce and Sampling**: The first script (separated by the line of asterisks) performs a straightforward ray tracing algorithm. It calculates the colour of each pixel by tracing a single primary ray and bouncing it multiple times in the scene. The `bounce\_limit` variable determines the number of ray bounces, and the number of rays per pixel is set to `num\_ray\_per\_pixels`. Each pixel's colour is determined by averaging the incoming light from multiple rays.

In contrast, the second script incorporates more advanced features. It introduces the concept of primary and secondary rays. The primary rays are traced for each pixel, and then for each intersection, secondary rays are traced in different directions to account for diffuse and specular reflections. The number of primary ray bounces is set to `num\_ray\_primary\_obj\_bounce`, and the number of secondary ray bounces is set to `num\_ray\_secondary\_obj\_bounce`. The resulting colour for each pixel is determined by averaging the contributions of the secondary rays.

2. **Frame Averaging**: The second script extends the functionality to render multiple frames. It introduces the `num\_frames` variable to specify the number of frames to render. Instead of directly storing the colour for each pixel, it accumulates the colours of each pixel over multiple frames in the `averaged\_frame\_image` array. After rendering all frames, it averages the accumulated colours to get the final colour for each pixel. This technique helps to reduce noise and produce smoother images.

3. **Parallelisation**: Both scripts use MPI (Message Passing Interface) to enable parallelisation and distribute the rendering process across multiple processes. The `comm`, `size`, and `rank` variables are used to handle the communication and coordination between the processes. Each process is responsible for rendering a portion of the image defined by `rank\_image\_height`. Parallelisation allows for faster rendering by utilising the resources of multiple processors or nodes.

4. **Image Output**: The second script saves the final image after rendering all frames, using the `Image.fromarray` function from the PIL (Python Imaging Library) module. It also plays a sound notification using the `playsound` module after the rendering is completed. These additions provide visual and auditory feedback to the user.

Overall, the second script introduces more advanced features such as primary and secondary rays, frame averaging, and parallelisation using MPI, which can result in higher-qualityand realistic images compared to the first script.

Pros and cons of both code implementations:

**First Script:**

Pros:

1. Simplicity: The first script is relatively simple and straightforward. It follows a basic ray tracing algorithm, making it easier to understand and modify.

2. Parallelisation: The use of MPI allows for parallel processing and distributing the rendering workload across multiple processes, resulting in faster rendering times and improved efficiency.

3. Fast Rendering: With parallelisation and tracing a limited number of rays per pixel, the first script can render images relatively quickly.

Cons:

1. Limited Realism: The first script may not capture advanced lighting effects accurately, such as multiple bounces, reflections, and refractions, which can limit the realism of the rendered images.

2. Noisy Output: With a limited number of rays per pixel (`num\_ray\_per\_pixels`), the first script may produce images with more noise or graininess, especially in areas with complex lighting or materials.

3. Lack of Advanced Features: The first script does not incorporate advanced features like frame averaging or fine-tuning of the ray-tracing parameters, which can limit its flexibility and the quality of the rendered images.

**Second Script:**

Pros:

1. Enhanced Realism: By incorporating primary and secondary rays, the second script can produce more realistic images with accurate reflections, refractions, and lighting effects.

2. Reduced Noise: The use of multiple primary and secondary rays per pixel (`num\_ray\_primary\_obj\_bounce` and `num\_ray\_secondary\_obj\_bounce`) helps to reduce noise and produce smoother images, especially when averaging multiple frames.

3. Frame Averaging: The second script introduces frame averaging, which can further enhance the quality of the rendered images by reducing noise and artifacts over multiple frames.

4. Parallelisation: The second script can distribute the rendering workload across multiple processes by utilising MPI for parallel processing, resulting in faster rendering times and improved efficiency.

Cons:

1. Increased Complexity: The second script is more complex than the first one, incorporating advanced features and additional ray-tracing techniques. This complexity can make it harder to understand, modify, and debug.

2. Longer Rendering Time: The use of multiple primary and secondary rays, along with frame averaging, can significantly increase the rendering time, especially for complex scenes or high-resolution images.

3. Potential Memory Usage: Accumulating the colour values for each pixel over multiple frames (`averaged\_frame\_image`) can require significant memory, especially if rendering a large number of frames or high-resolution images.