Computer Graphics: Rendering — Coursework Report

Exam No.: B283619

1. Introduction

1.1 Overview

This report primarily concerns the implementation of a ray tracing-based renderer supporting basic geometry, materials, lighting, and camera systems. Course requirements did not mandate external libraries, but during actual development, due to personal computer performance limitations, the Windows environment, and the need for Chinese output during compilation and debugging, I utilized external libraries such as <omp.h>, <windows.h>, and <bemapiset.h> for auxiliary calculations and Windows environment adaptation. Most of these features served solely to assist development and accommodate the development environment. Subsequent attempts to modify the code to eliminate external library usage proved challenging, as most functions were deeply integrated. Ultimately, these libraries were retained. The core code was developed by myself with AI assistance, and it is hoped that external library dependencies will not significantly impact the final grade.

1.2 Build Notes

The current project is being edited in a Windows environment, so SDK support is required. My compilation environment originally used CMake, but when I later attempted to make changes, the migration became impossible due to the large number of related items. However, I used CMake commands to edit the Makefile.
Please locate the Makefile within the build folder and execute the make command from the terminal within the build directory.
To accelerate operational efficiency, I have employed the OpenMP external library for parallel computing acceleration. Please ensure the availability of this library before running the program.

1.3 Course Objective Achievement Rate

	marks	Completion rate
Module 1	6	
Blender exporter	1	95%
Camera Space	4	100%
Image R/W	1	100%
Module 2	8	
Ray intersection	4	90%
Acceleration	4	90%
Module 3	12	
Whitted-style	8	80%
Antialiasing	2	100%
Textures	2	80%
Final raytracer	16	
Sys. Integration	4	90%

Distributed RT	8	80%
Lens effect	4	80%

2. Functionality Implementation Specification

2.1 Blender exporter (95%)

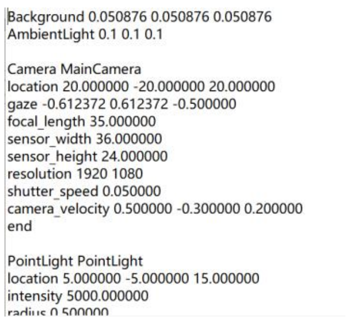
Related code file: Exporter.py

Function Description: The exporter primarily handles exporting objects set within a Blender scene to ASCII format files.

AI assistant provided: In the initial version, only partial information could be exported. Taking cameras as an example, only data such as position, look direction vector, and focal length could be output.

Modifications: In the final version, I added supplementary output for numerous additional properties, such as textures, object specifications, colors, and camera motion blur parameters. The current exporter creates an ASCII folder and generates a scene.txt file within it as the scene output.

Example:



```
Background 0.050876 0.050876 0.050876
AmbientLight 0.1 0.1 0.1

Camera MainCamera
location 20.000000 -20.000000 20.000000
gaze -0.612372 0.612372 -0.500000
focal_length 35.000000
sensor_width 36.000000
sensor_height 24.000000
resolution 1920 1080
shutter_speed 0.050000
camera_velocity 0.500000 -0.300000 0.200000
end

PointLight PointLight
location 5.000000 -5.000000 15.000000
intensity 5000.000000
radius 0.500000
```

Exported File Example

2.2 Camera Space (100%)

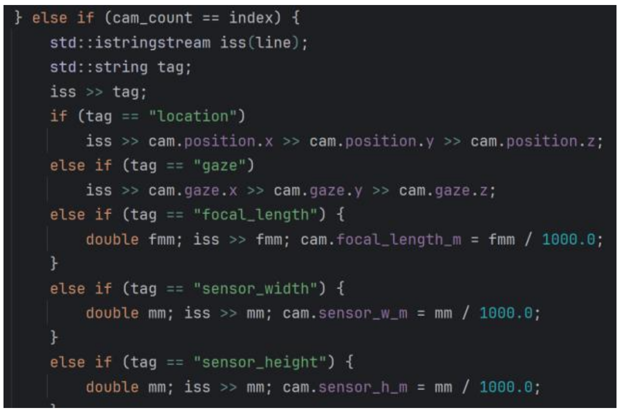
Related code files: camera.h, camera.cpp, Vector3.h, Ray.h

Functionality Description: Primarily implements camera-related capabilities

AI assistant provides: Code fulfilling initial requirements, including basic properties and fundamental methods.

Modifications: In the final version, the camera retains its original functionality while modifying other related methods. For example, the initialization function optimizes data reading details, aligns with Blender's output values, and incorporates adjustments.

Example:



```cpp
} else if (cam_count == index) {
    std::istringstream iss(line);
    std::string tag;
    iss >> tag;
    if (tag == "location")
        iss >> cam.position.x >> cam.position.y >> cam.position.z;
    else if (tag == "gaze")
        iss >> cam.gaze.x >> cam.gaze.y >> cam.gaze.z;
    else if (tag == "focal_length") {
        double fmm; iss >> fmm; cam.focal_length_m = fmm / 1000.0;
    }
    else if (tag == "sensor_width") {
        double mm; iss >> mm; cam.sensor_w_m = mm / 1000.0;
    }
    else if (tag == "sensor_height") {
        double mm; iss >> mm; cam.sensor_h_m = mm / 1000.0;
```

Camera.h File Example

2.3 Image R/W (100%)

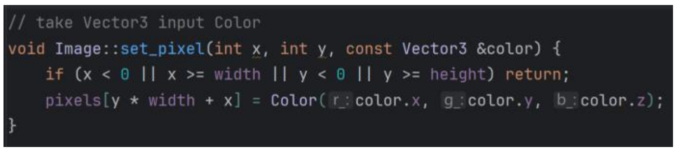
Relevant code files: Vector3.h, Image.h, Image.cpp

Functionality Description: Develop a C++ image class capable of reading and writing .ppm format image files.

AI Assistant Provided: Initial implementation of file read/write functionality exists, but it cannot effectively adjust actual pixel color data.

Modifications: Conveyed RGB color data using vector representation, implemented reasonable code reuse, modified pixel return handling to also utilize vectors.

Example:



```cpp
// take Vector3 input Color
void Image::set_pixel(int x, int y, const Vector3 &color) {
    if (x < 0 || x >= width || y < 0 || y >= height) return;
    pixels[y * width + x] = Color( r: color.x,  g: color.y,  b: color.z);
}
```

Image File Example

2.4 Ray intersection (90%)

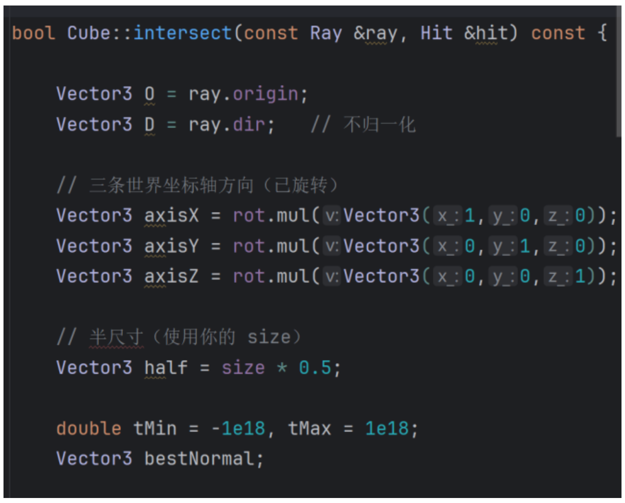
Relevant code files: Ray.h, SceneUtils.h, SceneUtils.cpp, Shape.h

Functionality Description: Implements ray intersection feedback for objects, returning intersection points and related information.

AI assistant provided: The basic approach for intersection detection has been implemented, but the actual execution remains somewhat unstable.

Modifications: Adjusted object transformation methods, particularly for the cube component. Local-to-world coordinate transformation was temporarily suspended in the final version in favor of simplified intersection calculations. The transformed intersection normal and ray return direction remained uncorrected until the end. Aim to optimize and implement this in future iterations.

Example:



```
bool Cube::intersect(const Ray &ray, Hit &hit) const {

    Vector3 O = ray.origin;
    Vector3 D = ray.dir;    // 不归一化

    // 三条世界坐标轴方向（已旋转）
    Vector3 axisX = rot.mul(v:Vector3(x:1, y:0, z:0));
    Vector3 axisY = rot.mul(v:Vector3(x:0, y:1, z:0));
    Vector3 axisZ = rot.mul(v:Vector3(x:0, y:0, z:1));

    // 半尺寸（使用你的 size）
    Vector3 half = size * 0.5;

    double tMin = -1e18, tMax = 1e18;
    Vector3 bestNormal;
```

Example of the Intersecting Cubes Algorithm

2.5 Acceleration (90%)

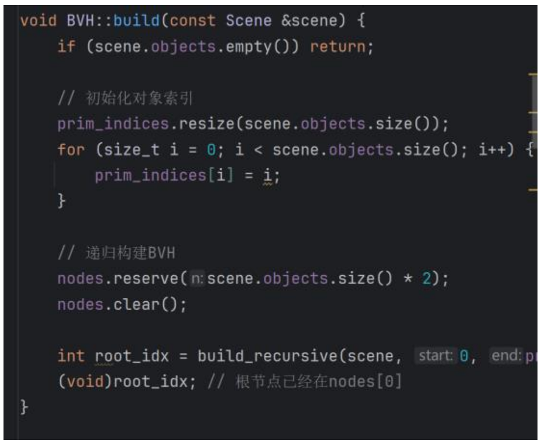
Relevant code files: BVH.h, BVH.cpp, SceneUtils.h, SceneUtils.cpp

Functionality Description: Constructs a hierarchical data structure to enhance intersection testing efficiency.

AI Assistant Provided: Basic BVH node and AABB collision box framework, with numerous adjustments made during subsequent practical implementation.

Modifications: Implemented changes to leaf node and object index verification, optimized decision structures, achieving high completion. Maintains consistent acceleration effects even after subsequent feature development.

Example:



```
void BVH::build(const Scene &scene) {
    if (scene.objects.empty()) return;

    // 初始化对象索引
    prim_indices.resize(scene.objects.size());
    for (size_t i = 0; i < scene.objects.size(); i++) {
        prim_indices[i] = i;
    }

    // 递归构建BVH
    nodes.reserve(n:scene.objects.size() * 2);
    nodes.clear();

    int root_idx = build_recursive(scene, start:0, end:p
    (void)root_idx; // 根节点已经在nodes[0]
}
```

Example of BVH Construction Method

2.6 Whitted-style (80%)

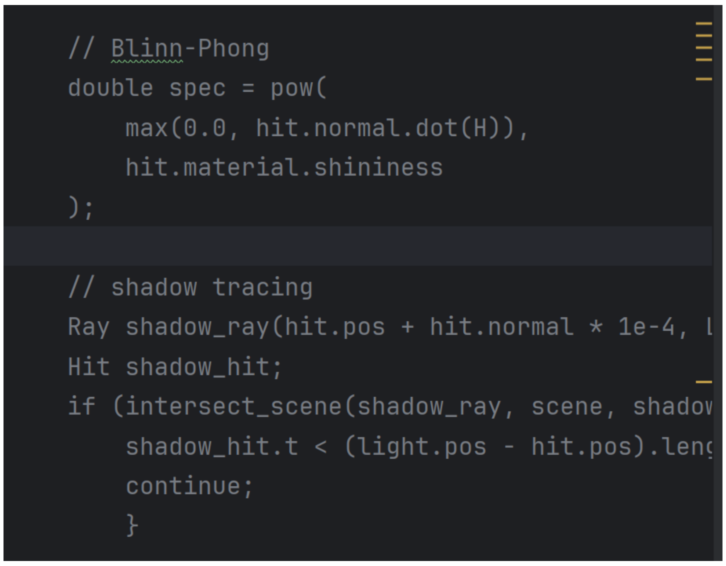
Relevant code file: main.cpp

Functionality description: Implements Whitted-style ray tracing.

AI assistant provided: Integrates previous single-ray tracing effects, modifies reflection and refraction effects based on the Blinn-Pong model for intersection point shading.

Modifications: The AI-provided version contained numerous computational errors, resulting in significant discrepancies between the returned color values and actual outcomes. Subsequent investigation revealed inconsistencies between object intersection calculations and current implementation. These issues were subsequently corrected. The provided example code has been commented out, as it will be merged into the code for implementing distributed ray tracing later.

Example:



```
// Blinn-Phong
double spec = pow(
    max(0.0, hit.normal.dot(H)),
    hit.material.shininess
);

// shadow tracing
Ray shadow_ray(hit.pos + hit.normal * 1e-4, l
Hit shadow_hit;
if (intersect_scene(shadow_ray, scene, shadow
    shadow_hit.t < (light.pos - hit.pos).leng
    continue;
    }
```

Ray Tracing Example

2.7 Antialiasing（100%）

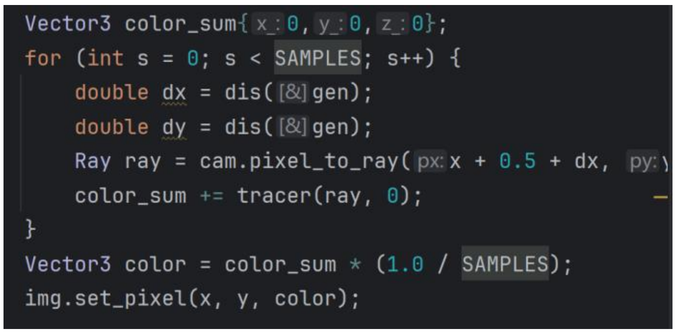
Relevant code file: main.cpp

Function Description: Implement anti-aliasing by calculating the average contribution of different points to a single pixel.

AI Assistant Provided: Add multiple samples at the same position within the original rendering function to compute the average value.

Modification: Minimal changes were made, focusing solely on sample quantity at the point calculation level.

Example:



```
Vector3 color_sum{ x: 0, y: 0, z: 0};
for (int s = 0; s < SAMPLES; s++) {
    double dx = dis( [&] gen);
    double dy = dis( [&] gen);
    Ray ray = cam.pixel_to_ray( px: x + 0.5 + dx,  py:
    color_sum += tracer(ray, 0);
}
Vector3 color = color_sum * (1.0 / SAMPLES);
img.set_pixel(x, y, color);
```

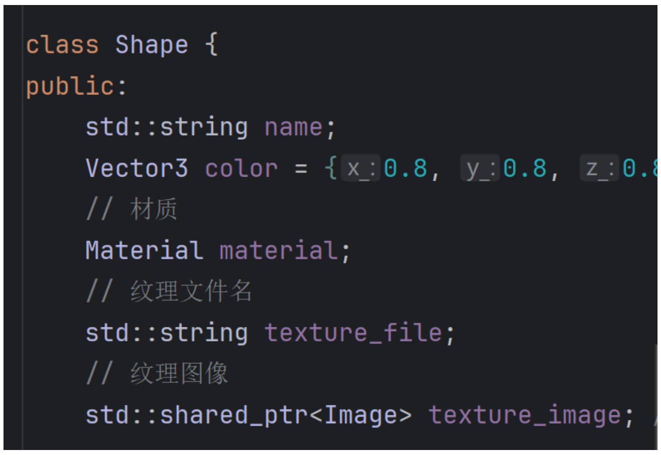
Antialiasing Code Example

2.8 Textures（80%）

Relevant code files: Shape.h, Scene.cpp, Exporter.py

Functionality description: Implemented texture functionality, currently enabling object shading based on texture files.

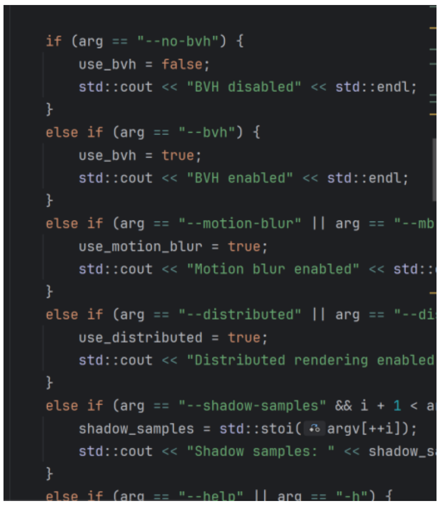
AI assistant provided: Implemented simple texture generation and texture file reading.
Modifications: Implemented recognition and output of texture-related properties within the exporter. Added corresponding acceptance logic in the scene loading function. For intersecting functions, relevant code files should also include modifications to each object's intersecting function, incorporating the texture coordinates used (typically u,v).
Example:



```
class Shape {
public:
    std::string name;
    Vector3 color = { x: 0.8,  y: 0.8,  z: 0.
    // 材质
    Material material;
    // 纹理文件名
    std::string texture_file;
    // 纹理图像
    std::shared_ptr<Image> texture_image;
```

Example of Texture Properties in Shape Class

2.9 Sys. Integration （90%）
Relevant code file: main.cpp
Functionality description: Integrates the system, consolidating and optimizing all the aforementioned modules.
AI assistant provides: Simple implementation of output and command-line invocation.
Modifications: Modified the available command-line parameters and controllable features. Now supports control over BVH, distributed ray tracing, lens shake effects, and the number of shadow sampling points in distributed ray tracing.
Example:

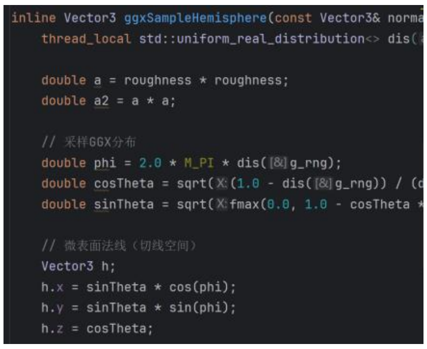


```
if (arg == "--no-bvh") {
    use_bvh = false;
    std::cout << "BVH disabled" << std::endl;
}
else if (arg == "--bvh") {
    use_bvh = true;
    std::cout << "BVH enabled" << std::endl;
}
else if (arg == "--motion-blur" || arg == "--mb
    use_motion_blur = true;
    std::cout << "Motion blur enabled" << std::
}
else if (arg == "--distributed" || arg == "--di
    use_distributed = true;
    std::cout << "Distributed rendering enabled
}
else if (arg == "--shadow-samples" && i + 1 < a
    shadow_samples = std::stoi( argv[++i]);
    std::cout << "Shadow samples: " << shadow_s
}
else if (arg == "--help" || arg == "-h") {
```

System Integration Examples

2.10 Distributed RT （80%）
Relevant code files: main.cpp, Sampling.h

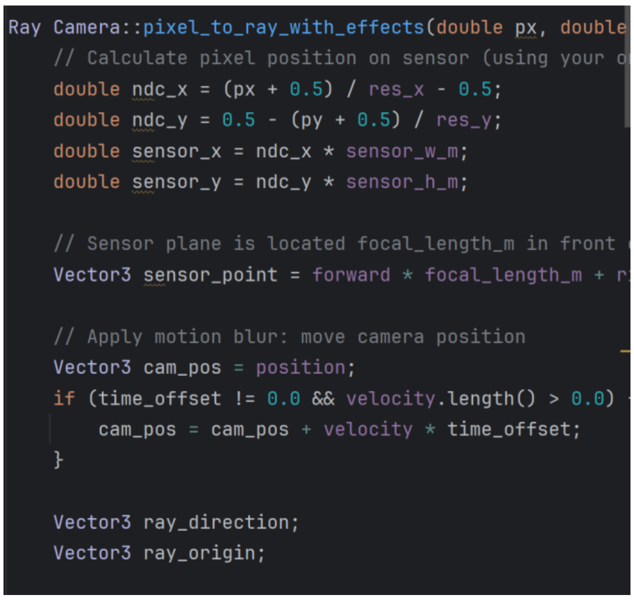
Functionality description: Implements soft shadows and specular reflections as distributed ray tracing effects.
AI assistant provided: Initially combined soft shadows and reflections, but failed to effectively provide sampling methods.
Modifications: Added sampling files and related functions, enabling modifications to sampling points.
Example:



```cpp
inline Vector3 ggxSampleHemisphere(const Vector3& norma
    thread_local std::uniform_real_distribution<> dis(

    double a = roughness * roughness;
    double a2 = a * a;

    // 采样GGX分布
    double phi = 2.0 * M_PI * dis(g_rng);
    double cosTheta = sqrt((1.0 - dis(g_rng)) / (c
    double sinTheta = sqrt(fmax(0.0, 1.0 - cosTheta *

    // 微表面法线（切线空间）
    Vector3 h;
    h.x = sinTheta * cos(phi);
    h.y = sinTheta * sin(phi);
    h.z = cosTheta;
```

Distributed Ray Tracing Sampling Example

2.11 Lens effect（80%）
Relevant code files: Camera.h, main.cpp
Functionality description: Implements motion blur and blur effects caused by out-of-focus (depth of field).
AI assistant provided: Supplied basic code implementation and added motion blur-related properties.
Modifications: Revised scene loading functions, exporters, and related functions to incorporate camera effects into computations. While the rendering performance improvement may not be as pronounced as distributed ray tracing, it achieves a more realistic implementation for edge blurring scenarios.
Example:



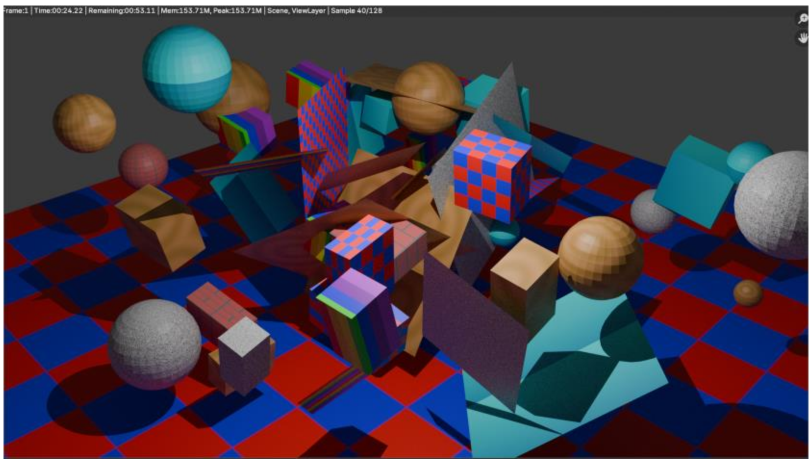
```cpp
Ray Camera::pixel_to_ray_with_effects(double px, double
    // Calculate pixel position on sensor (using your o
    double ndc_x = (px + 0.5) / res_x - 0.5;
    double ndc_y = 0.5 - (py + 0.5) / res_y;
    double sensor_x = ndc_x * sensor_w_m;
    double sensor_y = ndc_y * sensor_h_m;

    // Sensor plane is located focal_length_m in front
    Vector3 sensor_point = forward * focal_length_m + r

    // Apply motion blur: move camera position
    Vector3 cam_pos = position;
    if (time_offset != 0.0 && velocity.length() > 0.0)
        cam_pos = cam_pos + velocity * time_offset;
    }

    Vector3 ray_direction;
    Vector3 ray_origin;
```

Lens Effect Examples

3. Sample Output and Comparison

Test Scenario: text.blend

Blender Output:


rame:1 | Time:00:24.22 | Remaining:00:53.11 | Mem:153.71M, Peak:153.71M | Scene, ViewLayer | Sample 40/128

Blender Output Sample

3.1 Test Feature: BVH Acceleration

Output Files: output_bvh.ppm, output_no_bvh.ppm

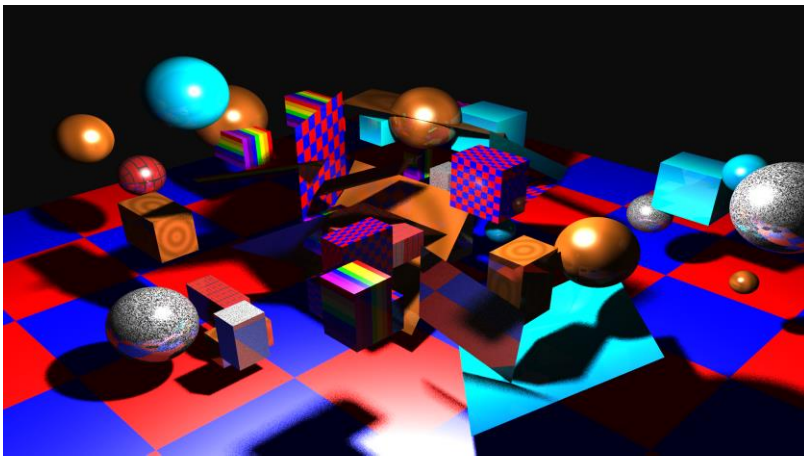
Similarity to Blender: 75%

Differences: More noticeable variations in highlights and shadows, with reduced detail

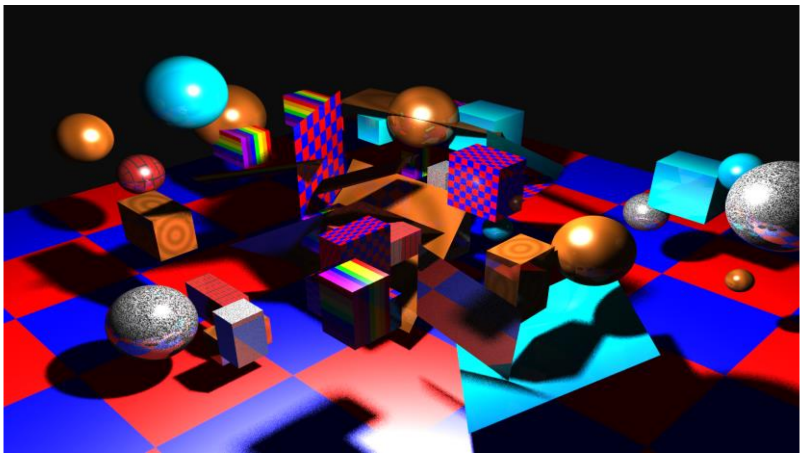
Rendering Time Comparison:

Here we uniformly analyze the effects of using BVH acceleration. My personal build utilizes OpenMP for parallel computing acceleration; if unavailable, runtime differences may occur.

Without distributed computing or camera effects, the non-accelerated runtime is 252.153 seconds, while the accelerated runtime is 183.944 seconds. In contrast, with distributed ray tracing and interstellar effects enabled: Accelerated time: 362.129 seconds Unaccelerated time: 440.653 seconds Other results show consistent trends. The gap between using and not using BVH acceleration is now quite noticeable. However, regardless of effects enabled, the required time remains significantly longer than Blender's performance. The project still requires substantial optimization.



BVH examples



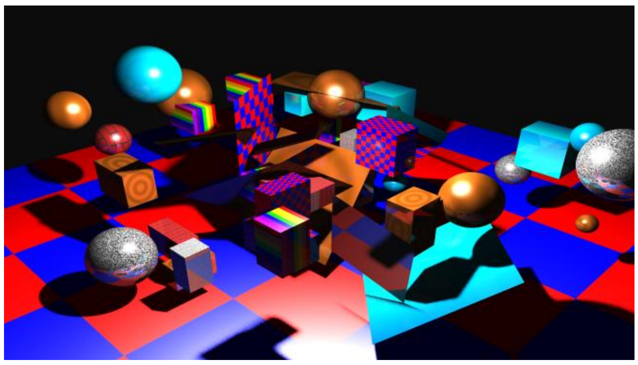
No BVH examples

3.2 Test Feature: Distributed Ray Tracing

Output File: output_distributed_bvh_ss4_ps16.ppm

Similarity to Blender: 90%

Differences: Shadow handling is highly similar to Blender; increasing shadow sampling may yield greater similarity.



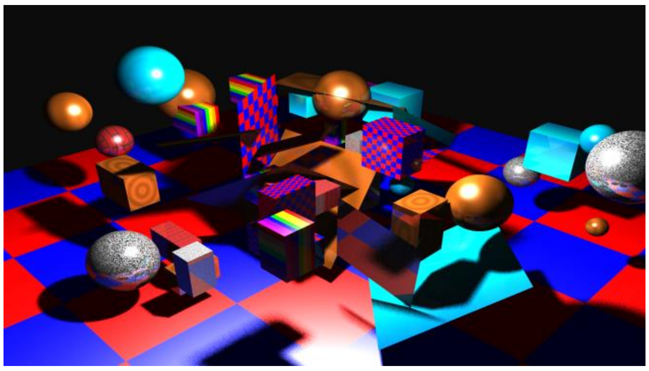
Distributed Ray Tracing Output Example

3.3 Test Function: Lens Effects

Output File: output_distributed_bvh_ss4_ps16.ppm

Similarity to Blender: 80%

Differences: Partial camera shake effects are present, but the handling of depth-of-field blur differs significantly from Blender and requires improvement.

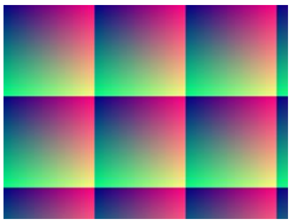


Sample Lens Effect Output

4. Timeliness bonus

Model 1: Only the initial section of the code has been implemented, with numerous functional shortcomings. For instance, the camera-related code fails entirely to focus on objects correctly in this first iteration. However, as only the camera, read/write, and exporter functionalities were realised in this initial version, the output cannot be used to assess the completeness of this feature. Consequently, it can only be termed a preliminary implementation.

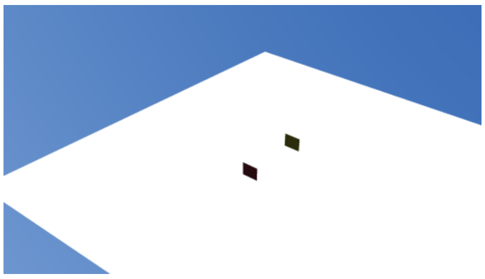
Output example:



Module One Version Output Example

Model 2: This implementation fulfils both requirements for ray intersection and BVH acceleration. However, the submitted content may prove challenging to discern in practical implementation. This stems from an initial mismatch between the exporter and reader functions, where point light intensity failed to render correctly on both sides. Consequently, the project's rendered output appeared predominantly white. Subsequent optimisations to the reader involved dividing light intensity by a constant to attenuate its data rather than employing normalisation. This approach prevents the intensity representation from being entirely overridden.

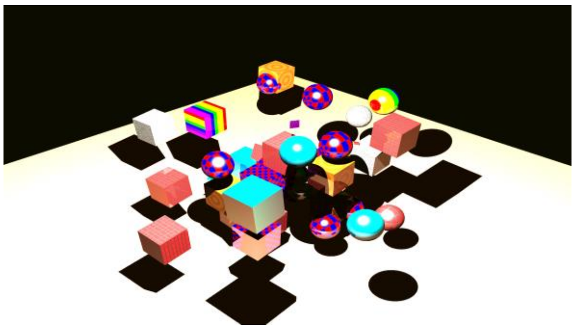
Output example:



Module Two Version Output Example

Model 3: Implemented Whitted-style ray tracing, but encountered issues with the transformation between local and world coordinates for cubes. Consequently, a temporary optimisation was applied, omitting the use of this transformation method in the code. Regarding anti-aliasing, the current output utilizes 16-point sampling for anti-aliasing implementation. Texture functionality is currently limited. Texture files are in PPM format. Having searched numerous websites without finding a method for rapid retrieval of PPM files, I generated PPM texture files myself, though the styles are extremely basic. The current read function only recognizes texture file names within the current folder. Should additional files be added later, modifications to exporting, reading, and related components will be required.

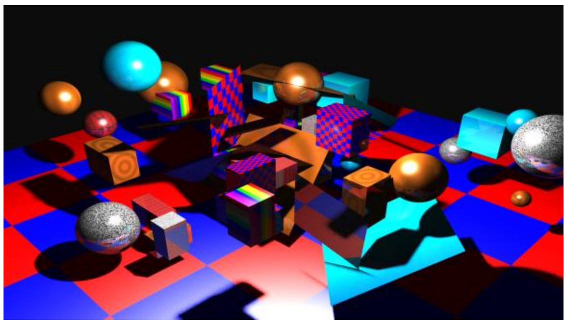
Output example:



Module Three Version Output Example

Final Ray Tracer: System integration has been completed, consolidating the original output and code usage. While the code currently lacks comprehensive modularisation, it possesses a degree of scalability. Command-line parameters now enable the activation or deactivation of various functions, and the export script has been modified to support the required parameters. Modularised control has been implemented for distributed ray tracing and enabling lens effects, though standalone file modularisation remains incomplete.

Output example:



Final version output example

5. Programming Assistant Usage Summary

Using programming assistants in this assignment offers numerous advantages: First, they rapidly provide foundational algorithm implementations, such as ray-geometry intersection detection, saving significant time on basic implementations. Additionally, AI can simplify explanations of complex concepts (like BVH construction), aiding comprehension of core principles. When encountering specific errors, the assistant offers step-by-step debugging suggestions.

However, programming assistants also have several drawbacks. Sometimes the code provided is too basic to solve specific problems, and due to the large number of files, it cannot understand complex interrelated calls. Furthermore, the assistant has limited understanding of domain-specific knowledge. Achieving the same functionality may require multiple iterations before a preliminary implementation, and some features might ultimately prove unimplementable. The code snippets provided often require substantial integration work.

In terms of personal work assistance, given the tight schedule during the semester, programming assistants have become indispensable. They enable individuals to rapidly transform foundational knowledge into functional code prototypes and offer some degree of support for code optimization. However, the final results fall far short of achieving a state where AI can autonomously generate code that fully meets all system requirements. Overall, programming assistants are powerful tools for boosting productivity but cannot replace systematic learning and professional expertise. They are best suited for rapid prototyping, algorithm interpretation, and debugging assistance. For complex graphics projects, long-term personal study and experience remain essential for identifying and resolving related issues.