Implementation of Blender software in C++

Yu-Chi Chu

24 Nov 2024

1. Basic Raytracer Features

- **(a) Image writing:**

  First, I include the **nlohmann_json.hpp** (Fig.1), which is the online resource to help me read the json file.

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <cmath>
#include <algorithm>
#include <nlohmann_json.hpp>
#include <random>
```

Fig.1

The **main** function (Fig.2) serves as the entry point for raytracing:

1. In/Output File Management: A pair of lists is defined to store the paths to the input JSON files and their corresponding output PPM images.
2. In/Output File Consistency Check: Ensure that the number of input and output files is equal to prevent potential errors during processing.
3. Scene Processing Loop: A loop iterates over each in/output file pair, delegating the rendering process to the following function.

```cpp
int main()
{
    std::vector<std::string> inputFiles = {
        "binary_scene.json", "binary_primitives.json", "phong_scen
    std::vector<std::string> outputFiles = {
        "binary_scene.ppm", "binary_primitives.ppm", "phong_scene.

    if (inputFiles.size() != outputFiles.size())
    {
        std::cerr << "Input and output file lists must have the sa
        return 1;
    }
    for (size_t i = 0; i < inputFiles.size(); ++i)
    {
        processScene(inputFiles[i], outputFiles[i]);
    }
    return 0;
}
```

Fig.2

The **readJson** function (Fig.3) is responsible for parsing JSON data:

1. File Access: Open the specified file.
2. JSON Parsing: From the file stream and stored in a JSON object.
3. Error Handling: File cannot be opened; an error message is displayed.

```
json readJson(const std::string &filepath){
    std::ifstream file(filepath);
    json j;
    if (file)    {
        file >> j;
    }
    else    {
        std::cerr << "Could not open file: " << filepath << std::e
    }
    return j;
}
```

Fig.3

- **(b) Virtual pin-hole camera:**

  First, according to the .json file (Fig.4) to implement camera structure:

```
"camera":
    {
        "type":"pinhole",
        "width":1200,
        "height":800,
        "position":[0.0, 0.75, -0.25],
        "lookAt":[0.0, 0.35, 1.0],
        "upVector":[0.0, 1.0, 0.0],
        "fov":45.0,
        "exposure":0.1
    },
```

Fig.4

The **Camera** structure (Fig.5) defines the camera parameters used in the ray tracing process. (Some parameters weren't showed in this json file)

1. position: The position of the camera in 3D space.
2. lookAt: The point in the scene that the camera is looking at.
3. upVector: Defining its orientation.
4. fov: The field of view of the camera.
5. aperture: Affecting depth-of-field.
6. focusDistance: The distance to the point the camera is focused on.

```
struct Camera
{
    Vec3 position;
    Vec3 lookAt;
    Vec3 upVector;
    float fov;
    float aperture;
    float focusDistance;
    Camera(const Vec3 &pos, const Vec3 &look, const Vec3 &up, floa
        : position(pos), lookAt(look), upVector(up), fov(f), apert
};
```

Fig.5

In my **camera** structure, the Vec3 is used to do some arithmetic

calculations (Fig.6), and I only show one part of code:

1. Constructors: Initialize vectors with default values.
2. Arithmetic Operators/Assignment Operators: +, -, *, +=, *=, dot, cross, normalize, [], ==: Default arithmetic operators in C++ only accept same type, so use self-defined function.

```
struct Vec3
{
    float x, y, z;
    Vec3() : x(0), y(0), z(0) {}
    Vec3(float a, float b, float c) : x(a), y(b), z(c) {}
    // Vector addition
    Vec3 operator+(const Vec3 &other) const { return Vec3(x + othe
    // Vector subtraction
    Vec3 operator-(const Vec3 &other) const { return Vec3(x - othe
    // Scalar multiplication
    Vec3 operator*(float scalar) const { return Vec3(x * scalar, y
    // Vector multiplication
    Vec3 operator*(const Vec3 &other) const { return Vec3(x * othe
    // Negation
    Vec3 operator-() const { return Vec3(-x, -y, -z); }
```

Fig.6

Back to **camera** structure, there's **parseCamera** function (Fig.7). It parses the camera data from a JSON object and retrieves camera parameters using their corresponding keys and constructs a Camera object.

```
Camera parseCamera(const json &cameraData)
{
    Vec3 position = parseVec3(cameraData, "position");
    Vec3 lookAt = parseVec3(cameraData, "lookAt");
    Vec3 upVector = parseVec3(cameraData, "upVector");
    float fov = cameraData.at("fov").get<float>();
    float aperture = cameraData.value("aperture", 0.0f);
    float focusDistance = cameraData.value("focusDistance", 1.0f);

    return Camera(position, lookAt, upVector, fov, aperture, focus
}
```

Fig.7

Also in my **parseCamera**, there's **parseVec3** function (Fig.8). It extracts a 3D vector from a JSON object using specified key. This function simplifies the process of retrieving vector data from the scene description files.

```
Vec3 parseVec3(const json &js, const std::string &key)
{
    return Vec3(js[key][0].get<float>(), js[key][1].get<float>(),
}
```

Fig.8

- **(c) Intersection tests:**

  For this part, I implemented the intersection tests to each shape:

Sphere: The ray-sphere intersection uses the quadratic formula. (Fig.9)

1. oc determines the ray's relative position to the sphere.
2. a, b, c then be calculated to the quadratic formula.
3. Calculates the discriminant, which determines the number of intersection points between the ray and the sphere.
4. If discriminant is positive, 2 intersections; If zero, 1 intersection (tangent); otherwise, no intersections.

```
public:
    Sphere(const Vec3 &center, float radius, const Material &mater
        : Shape(material), center(center), radius(radius) {}

    bool intersect(const Ray &ray, float &t) const override
    {
        Vec3 oc = ray.origin - center;
        float a = ray.direction.dot(ray.direction);
        float b = 2.0f * oc.dot(ray.direction);
        float c = oc.dot(oc) - radius * radius;
        float discriminant = b * b - 4 * a * c;
        if (discriminant < 0)
        {
            return false;
        }
        else
        {
            t = (-b - std::sqrt(discriminant)) / (2.0f * a);
            return t > 0;
        }
    }
```

Fig.9

The **computeNormal** function returns normalized vector pointing outwards from the intersection point on the sphere surface.

The **boundingBox** method creates an AABB (Axis-Aligned Bounding Box) around the sphere. (Fig.10)

```
    Vec3 computeNormal(const Vec3 &p) const override
    {
        return (p - center).normalize();
    }

    bool boundingBox(AABB &output_box) const override
    {
        Vec3 rVec(radius, radius, radius);
        output_box = AABB(center - rVec, center + rVec);
        return true;
    }
};
```

Fig.10

Triangle: The intersection logic uses Moller-Trumbore algorithm. (Fig.11)

1. Calculate edge vectors for triangles.
2. Plane normal calculation: It calculates the normal vector to the plane (between the ray's direction vector and edge).
3. Area check (Backface Culling): If **abs(a)** is close to zero, the ray might be parallel to the plane, returning false to avoid calculation.
4. Parameterization (Ray-Plane Intersection):
   s: the vector from the triangle's vertex to the ray's origin.
   u: represent the parameter along edge where the ray intersects the plane containing the triangle.
   This check ensures the intersection point lies within the triangle's bounds along edge. If u is outside the range (0 ~ 1), the ray intersects the plane but misses the triangle, return false.
5. Second Parameterization (Intersection with Triangle): q, v are calculated using the dot product of the ray's direction vector, representing the parameter along edge where the ray intersects the plane containing the triangle.
   This check ensures the intersection point lies within the triangle's bounds along edge. If v < 0 or u + v > 1, the ray intersects the plane but misses the triangle, so the function returns false.
6. Intersection Distance Calculation: If u, v check passed, calculate the actual distance along the ray to the intersection point.
7. The rest (**computeNormal , boundingBox**) have similar logic.

```
class Triangle : public Shape
    bool intersect(const Ray &ray, float &t) const override
    {
        Vec3 edge1 = v1 - v0;
        Vec3 edge2 = v2 - v0;
        Vec3 h = ray.direction.cross(edge2);
        float a = edge1.dot(h);
        if (std::abs(a) < 1e-5)
            return false;

        float f = 1.0f / a;
        Vec3 s = ray.origin - v0;
        float u = f * s.dot(h);
        if (u < 0.0f || u > 1.0f)
            return false;

        Vec3 q = s.cross(edge1);
        float v = f * ray.direction.dot(q);
        if (v < 0.0f || u + v > 1.0f)
            return false;

        t = f * edge2.dot(q);
        return t > 1e-5;
    }
```

Fig.11

Cylinder: This logic involves solving a quadratic equation and checking intersection points against the cylinder's height. (Fig.12)

1. Calculate edge vectors, direction vector of the ray, pre-defined vector representing the cylinder's axis.
2. Project the direction vector onto a plane perpendicular to the axis. This removes any component of the direction that's aligned with the axis, effectively treating the cylinder as a rectangular prism along the non-axis directions.
3. Setting Up the Quadratic Equation, calculating the squared magnitude of the direction along the relevant plane, the squared distance from the modified center point to the origin
   This essentially checks if the modified origin point is within the cylinder's radius along the relevant plane.
4. Checking for Intersection (discriminant): If the discriminant is negative, there's no intersection, return false.
5. Finding Intersection Points (t0 & t1): Calculate the square root of the discriminant, two distance along the ray where it intersects the object (based on the quadratic formula).
6. Checking Intersection: Calculate the actual intersection point, project the intersection point onto the cylinder's axis.
   This checks if the intersection falls within the valid height range of the cylinder.
7. Check for the same height for the second intersection point. If both t0, t1 fall outside the height range, the function returns false.
8. The rest (computeNormal , boundingBox) are similar logic.

```
class Cylinder : public Shape
    bool intersect(const Ray &ray, float &t) const override
        Vec3 oc = ray.origin - center;
        Vec3 d = ray.direction;
        Vec3 a = axis;

        Vec3 d_prime = d - a * d.dot(a);
        Vec3 oc_prime = oc - a * oc.dot(a);

        float A = d_prime.dot(d_prime);
        float B = 2.0f * d_prime.dot(oc_prime);
        float C = oc_prime.dot(oc_prime) - radius * radius;

        float discriminant = B * B - 4 * A * C;
        if (discriminant < 0)
        {
            return false;
        }

        float sqrt_discriminant = std::sqrt(discriminant);
        float t0 = (-B - sqrt_discriminant) / (2.0f * A);
        float t1 = (-B + sqrt_discriminant) / (2.0f * A);
```

```
        if (t0 > t1)
            std::swap(t0, t1);

        Vec3 p0 = ray.origin + d * t0;
        float y0 = (p0 - center).dot(a);

        if (y0 < 0 || y0 > height)
        {
            Vec3 p1 = ray.origin + d * t1;
            float y1 = (p1 - center).dot(a);
            if (y1 < 0 || y1 > height)
                return false;
            t0 = t1;
        }

        if (t0 < 0)
            return false;
        t = t0;
        return true;
    }
```

Fig.12

AABB: It represents an Axis-Aligned Bounding Box defined by its min and max corner points. The intersect method checks if a ray intersects the AABB within a specified **tmin** and **tmax** range. (Fig.13)

1. Iterate over x, y, and z axes, and find out the inverse of the ray's direction component along the current axis, the time at which the ray enters/exits the AABB along the current axis.
2. Handle Negative Direction: If the ray's direction is negative along the current axis, the entry/exit times need to be swapped.
3. Update Intersection Interval: Updates **tmin** to ensure that the intersection must occur after the previous intersection.
4. If **tmax** <= **tmin**, no intersection along this axis; complete without returning false, the ray intersects the AABB along all axes.

```
struct AABB
{
    // Check intersection with a ray
    bool intersect(const Ray &ray, float tmin, float tmax) const
    {
        for (int i = 0; i < 3; ++i)
        {
            float invD = 1.0f / ray.direction[i];
            float t0 = (min[i] - ray.origin[i]) * invD;
            float t1 = (max[i] - ray.origin[i]) * invD;
            if (invD < 0.0f)
                std::swap(t0, t1);
            tmin = t0 > tmin ? t0 : tmin;
            tmax = t1 < tmax ? t1 : tmax;
            if (tmax <= tmin)
                return false;
        }
        return true;
    }
};
```

Fig.13

As I already have intersection tests & image writing & virtual pin-hole camera, I can render the binary mode now. **There's a important thing to mention that due to pin-hole camera, the image will be opposite.**

Read in **"binary_scene.json"**. Open **"binary_scene.ppm"** file. (Fig.14)

Fig.14 **("binary_scene.ppm")**

Read in **"binary_primitives.json"**. Open **"binary_primitives.ppm"** file. (Fig.15)

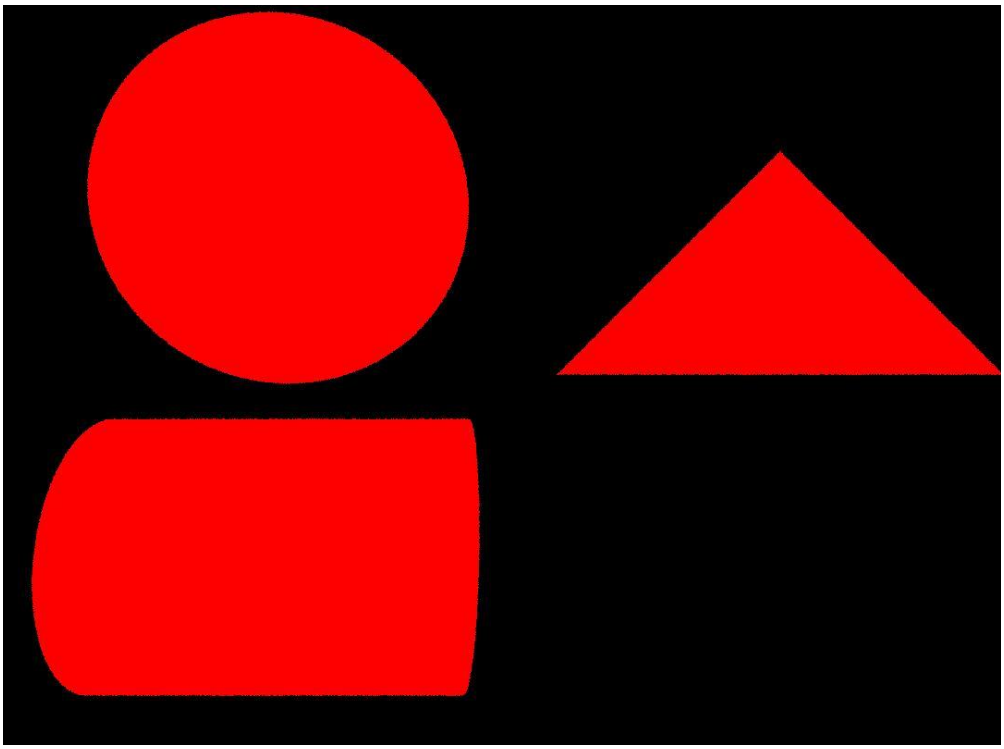


Fig.15 **("binary_primitives.ppm")**

- **(d) Blinn-Phong shading:**

  This function implements the Blinn-Phong shading model to calculate the final color of a point on a 3D object, not just red and black. (Fig.16)

  1. Setting Ambient Light: Define a static ambient light color with a low intensity. Initialize a color variable color by multiplying the ambient light with the material's diffuse color. This represents the base color influenced by the ambient light.
  2. Light Direction and Shadow Check: Calculate the direction vector from the light source's position to the point on the object. This is normalized to get a unit vector.
     This function checks for any objects between the light and the point, blocking the light.
  3. Halfway Vector Calculation: The view direction vector points from the camera towards the point on the object.
  4. Diffuse Light Calculation: Calculate the diffuse contribution by taking dot product of the surface normal and the light direction. The **max** function ensures non-negative result (light hitting the surface from the "back" would have a negative dot product).
  5. Specular Light Calculation: Calculate the specular contribution based on the halfway vector. Determine how well the light reflects off the surface to the viewer. It is raised to the power of material's specular exponent, which controls the shininess of the material.
  6. Adding Contributions & Returning Color: Accumulate the diffuse & specular contributions from this light source to the overall color.

```
Vec3 blinnPhongShading(const Vec3 &point, const Vec3 &normal, cons
{
    Vec3 ambient(0.1, 0.1, 0.1); // Static ambient light
    Vec3 color = ambient * material.diffuseColor;
    for (const auto &light : lights)
    {
        Vec3 lightDir = (light->position - point).normalize();
        if (!isInShadow(point, lights, shapes))
        {
            Vec3 halfwayDir = (lightDir + viewDir).normalize();

            float diff = std::max(normal.dot(lightDir), 0.0f);
            Vec3 diffuse = material.kd * diff * material.diffuseCo

            float spec = std::pow(std::max(normal.dot(halfwayDir),
            Vec3 specular = material.ks * spec * material.specular

            color += diffuse + specular;
        }
    }
    return color;
}
```

Fig.16

BlinnPhongShading takes parameters from JSON file. (Fig.17)

```
"shapes":[
    {
        "type":"sphere",
        "center": [0, -25.0, 0],
        "radius":25.1,
        "material":
            {
                "ks":0.1,
                "kd":0.9,
                "specularexponent":10,
                "diffusecolor":[0.5, 1, 0.5],
                "specularcolor":[1.0,1.0,1.0],
                "isreflective":false,
                "reflectivity":1.0,
                "isrefractive":false,
                "refractiveindex":1.0
            }
    },
```

Fig.17

The parameters will then go to **struct Material** to run the logic. (Fig.18)

1. ks/kd: 0~1 that determines the amount of specular/diffuse reflection.
2. specularExponent: Control the sharpness of specular highlights.
3. reflectivity: 0~1 that determines the amount of reflection.
4. refractiveIndex: Used for refraction calculations.
5. diffuseColor: Color of the diffuse reflection.
6. specularColor: Color of the specular reflection.
7. isReflective: Boolean flag indicating whether the material is reflective.
8. isRefractive: Boolean flag indicating whether the material is refractive.

```
struct Material
{
    float ks, kd, specularExponent, reflectivity, refractiveIndex;
    Vec3 diffuseColor, specularColor;
    bool isReflective, isRefractive;

    Material()
        : ks(0), kd(0), specularExponent(1), reflectivity(1),
          refractiveIndex(1), diffuseColor(1, 1, 1), specularColor
          isReflective(false), isRefractive(false) {}

    Material(float ks, float kd, float specExp, Vec3 diffColor, Ve
             bool isRefl, bool isRefr, float refl, float refrIdx)
        : ks(ks), kd(kd), specularExponent(specExp), reflectivity(
          refractiveIndex(refrIdx), diffuseColor(diffColor), specu
          isReflective(isRefl), isRefractive(isRefr) {}
};
```

Fig.18

- **(e) Shadows:**

  This function helps determine the amount of light that reaches a surface point by considering the presence of occluding objects between the point and the light source. This information is then used to calculate the final color of the pixel corresponding to that point. (Fig.19)

  1. Calculate the direction vector from the point to the light source, normalize the vector to ensure it has a unit length.
  2. Create a shadow ray originating slightly offset (prevents self-shadowing) from the point in the direction of the light source.
  3. Store the intersection distance if a shadow-casting object is found.
  4. Iterates over each shape in the scene.
  5. Checks if the shadow ray intersects the current shape. The condition ensures that the intersection point is in front of the point being tested. If an intersection occurs, partial shadowing; otherwise, full illumination.

```
bool isInShadow(const Vec3 &point, const std::vector<Light *> &lig
{
    for (const auto &light : lights)
    {
        Vec3 lightDir = (light->position - point).normalize();
        Ray shadowRay(point + lightDir * 0.001f, lightDir);
        float t;

        for (const auto &shape : shapes)
        {
            if (shape->intersect(shadowRay, t) && t > 0.001f)
            {
                return true;
            }
        }
    }
    return false;
}
```

Fig.19

With **shadow** function being implemented, I can then do the Blinn-Phong mode for **"simple_phong.json"**. Here's **"simple_phong.ppm"**. (Fig.20)

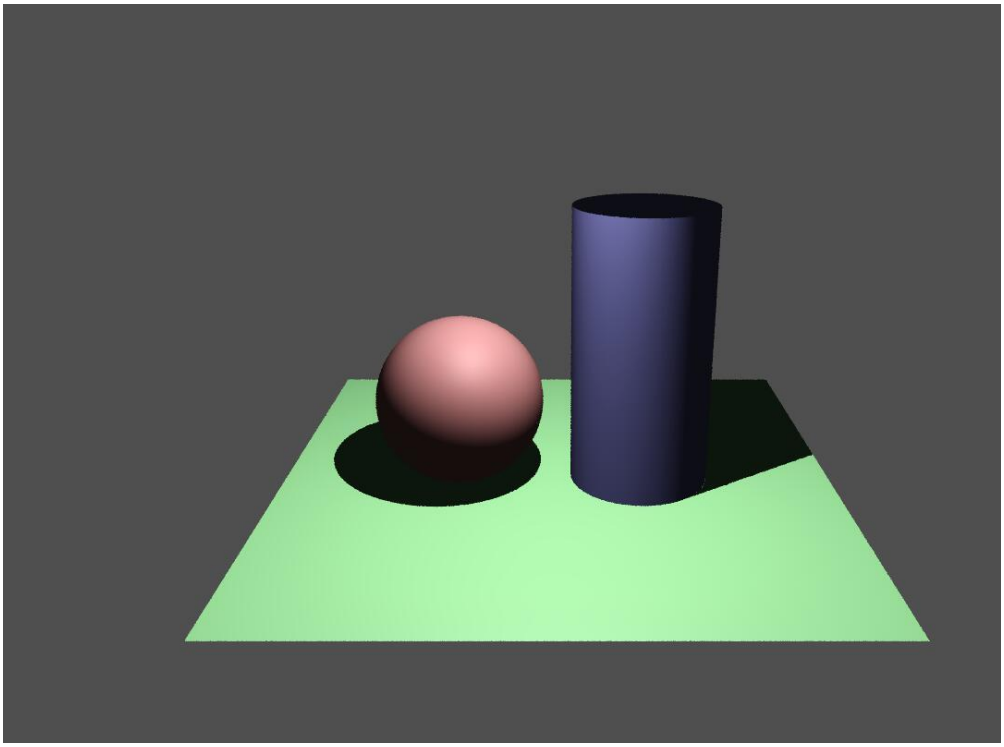


Fig.20 **("simple_phong.ppm")**

- **(f) Tone mapping:**

  The function is done by clamping the pixel values to a specific range, ensuring that no values exceed the displayable range. This prevents clipping and loss of detail in both highlights and shadows. (Fig.21)

  1. Define a simple structure to represent a pixel with red, green, and blue color components.
  2. Take vector of Pixel structures representing an HDR image as input, return the tone-mapped LDR image.
  3. Clamp the component of the pixel to the range [0, 1], and add the clamped pixel to the **ldr_image** vector.

```
struct Pixel {
    float r, g, b;
};

std::vector<Pixel> linear_tone_mapping(const std::vector<Pixel>& h
    std::vector<Pixel> ldr_image;
    for (const Pixel& pixel : hdr_image) {
        Pixel clamped;
        clamped.r = std::min(std::max(pixel.r, 0.0f), 1.0f);
        clamped.g = std::min(std::max(pixel.g, 0.0f), 1.0f);
        clamped.b = std::min(std::max(pixel.b, 0.0f), 1.0f);
        ldr_image.push_back(clamped);
    }
    return ldr_image;
}
```

Fig.21

Originally, we were asked to implement linear tone mapping. However, the effect created by this method is not that natural. Therefore, I implement another method that contains more complexity in calculating tone mapping parameters. It can be found in the **Exceptionalism** part.

- **(g) Reflection:**

  Designed based on "angle of incidence = angle of reflection." (Fig.22)

  1. I is a constant 3D vector representing the incident vector.
  2. N is a constant 3D vector representing the normal of the surface.
  3. Calculate final reflected vector, pointing in the opposite direction as incident vector but lies on the plane defined by normal vector.

```
Vec3 reflect(const Vec3 &I, const Vec3 &N)
{
    return I - N * 2.0f * I.dot(N);
}
```

Fig.22

- **(h) Refraction:**

  It calculates refracted direction of ray passing through surface. (Fig.23)

  1. Clamping cosine of incident angle: Store the cosine of the incident angle, measuring how directly the ray hits the surface, and ensure the cosine value stays within its valid range.
  2. Setting up refractive indices: Store the refractive index of the medium the light is coming from/entering.
  3. Handling light entering/exiting the surface: If negative, flips the sign to make calculations work for both cases (light entering or exiting); if positive (light entering), swaps the refractive indices.
  4. Calculate refraction ratio, coefficient & check internal reflection.
  5. Refracted ray direction: Implement Snell's law of refraction.

```
bool refract(const Vec3 &I, const Vec3 &N, float eta, Vec3 &refrac
    float cosi = std::clamp(I.dot(N), -1.0f, 1.0f);
    float etai = 1, etat = eta;
    Vec3 n = N;
    if (cosi < 0)    {
        cosi = -cosi;
    }
    else    {
        std::swap(etai, etat);
        n = -N;
    }
    float etaRatio = etai / etat;
    float k = 1 - etaRatio * etaRatio * (1 - cosi * cosi);
    if (k < 0)
        return false;
    refracted = I * etaRatio + n * (etaRatio * cosi - std::sqrt(k)
    return true;
}
```

Fig.23

Until now, I have enough function to do Blinn-Phong on other two pictures (Fig.24~25). The shadow is a bit offset to avoid self-shadowing.

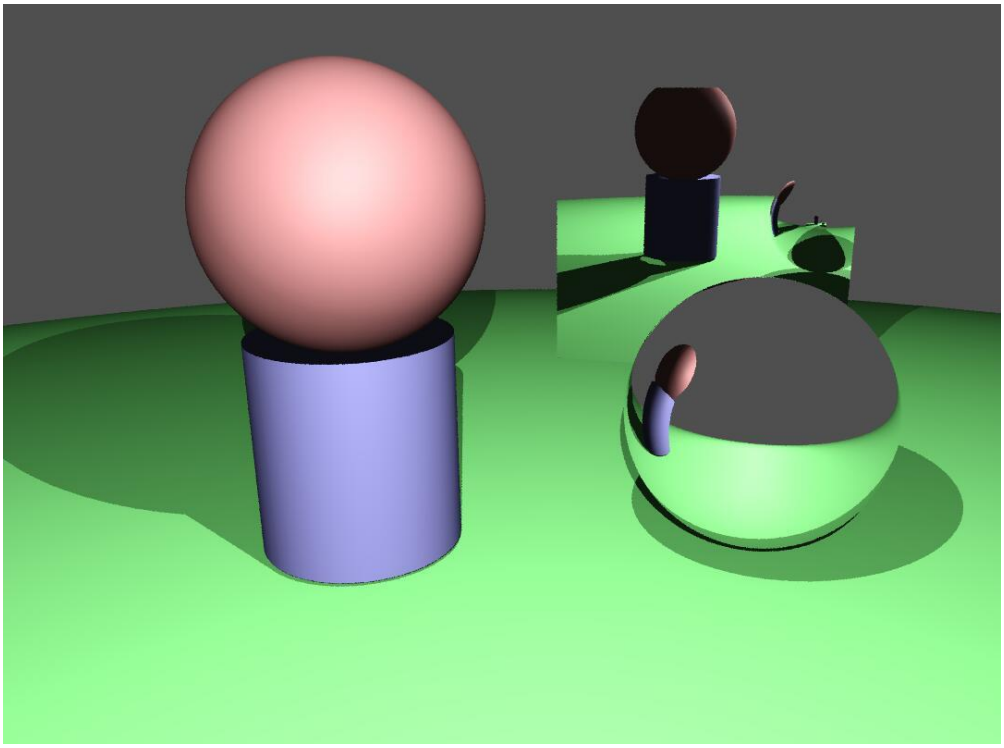


Fig.24 (**"phong_scene.ppm")**

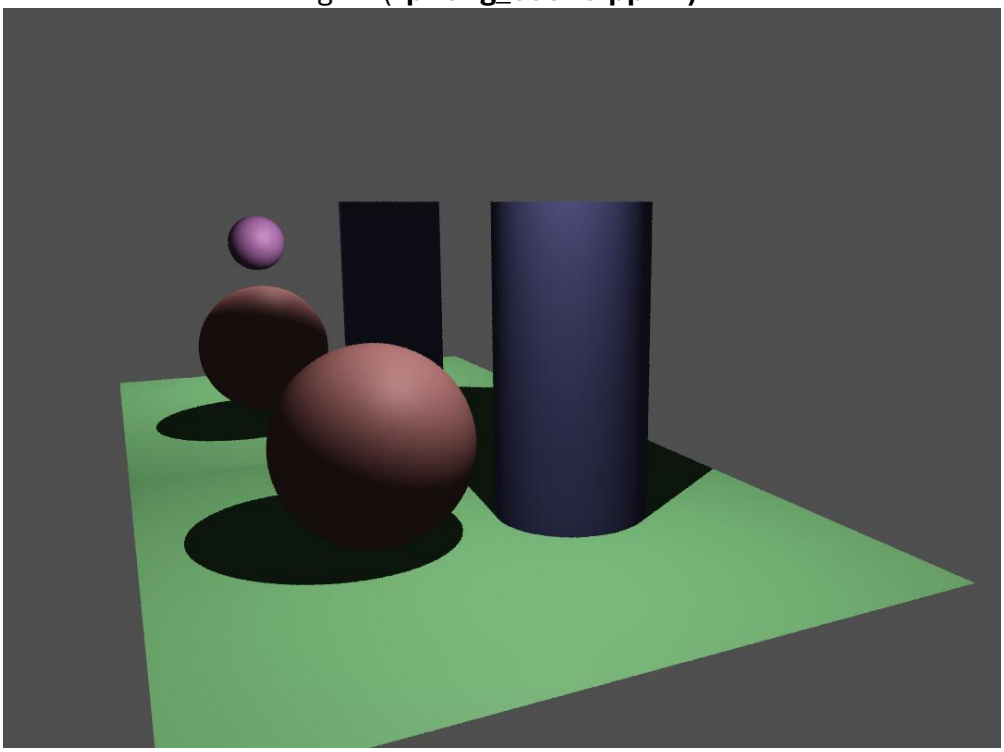


Fig.25 (**"mirror_image.ppm")**

2. Intermediate Raytracer Features

- **(a) Textures:**

  This is a part of code to represent an image texture, including storing the texture's dimensions and pixel data. (Fig.26)

1. Load data: If the data pointer isn't null, it was loaded successfully.
2. UV Mapping: Sample the texture at specified UV coordinates.
3. If there's no texture file, do not apply this function.
4. Wrap the u, v coordinate to the range [0, 1), and calculate the x, y coordinate of the pixel to sample, clamping it to the valid range.
5. Extract each color component from the pixel data and scales it to the range [0, 1], and finish this function calculation.

```
struct Texture

    ~Texture()
    {
        if (data)
            stbi_image_free(data);
    }

    Vec3 sample(float u, float v) const
    {
        if (!data)
            return Vec3(1.0f, 1.0f, 1.0f); // Return white if no t

        u = u - std::floor(u); // Wrap texture coordinates
        v = v - std::floor(v);

        int x = std::min(static_cast<int>(u * width), width - 1);
        int y = std::min(static_cast<int>(v * height), height - 1)
        int offset = (y * width + x) * channels;

        float r = data[offset] / 255.0f;
        float g = data[offset + 1] / 255.0f;
        float b = data[offset + 2] / 255.0f;

        return Vec3(r, g, b);
```

Fig.26

For input the texture file, I modified the **JSON** file for **"texture"**. (Fig.27)

```
        "material":
            {
                "ks":0.1,
                "kd":0.9,
                "specularexponent":20,
                "diffusecolor":[0.5, 0.7, 0.9],
                "specularcolor":[1.0,1.0,1.0],
                "texture": "texture3.jpeg",
                "isreflective":false,
                "reflectivity":1.0,
                "isrefractive":false,
                "refractiveindex":1.0
            }
```

Fig.27

Then, update the **getColor** in each shape. It combines base color with the texture color sampled at UV coords. (Take sphere for example, Fig.28)

1. Calculate the base color of the point p, considering factors like lighting, shading, and material properties.
2. If the sphere has a texture applied to it, then do UV mapping.
3. Sample the texture to obtain the texture color at that point and multiply base color by the texture color to modulate the final color.

```
    Vec3 getColor(
        const Vec3 &p, const Vec3 &viewDir,
        const std::vector<Shape *> &shapes, const std::vector<Ligh
    {
        Vec3 baseColor = Shape::getColor(p, viewDir, shapes, light
        if (material.texture)
        {
            float u = 0.5 + atan2(p.z - center.z, p.x - center.x)
            float v = 0.5 - asin((p.y - center.y) / radius) / M_PI
            Vec3 textureColor = material.texture->sample(u, v);
            baseColor *= textureColor;
        }
        return baseColor;
    }
};
```

Fig.28

The rest of code is just some slight changes, like adding texture definition.

Since I have already created **texture** function, I can now apply texture on the objects. Take **"simple_texture.json"** for example: (Fig.29)

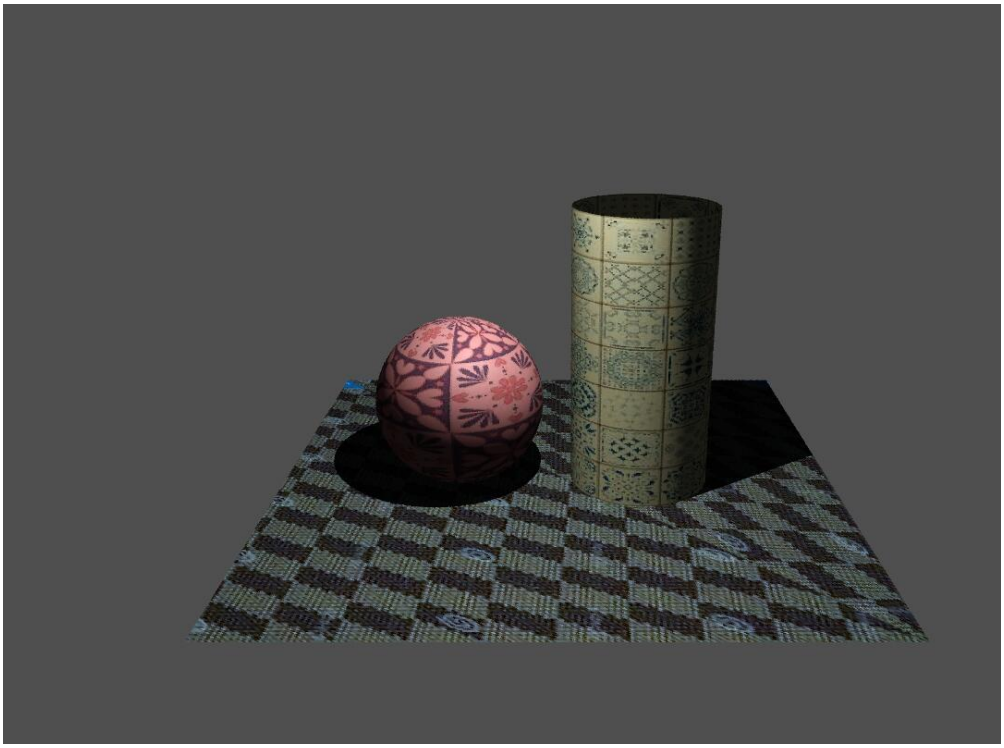


Fig.29 (**"simple_texture.ppm"**)

Here’s my self-created texture figure. (Fig.30~32)

Fig.30 (For sphere)

Fig.31 (For cylinder)

Fig.32 (For plane)

- **(b) Acceleration hierarchy:**

  Accelerate using Bounding Volume Hierarchy(BVH) by building its function, optimizing intersection tests, and verify its performance.

  1. **Surroundingbox:** Calculate the smallest Axis-Aligned Bounding Box (AABB) that encompasses two given AABBs. It does this by finding the minimum and maximum coordinates along each axis and constructing a new AABB using these values. (Fig.33)

```
struct AABB
{
    static AABB surroundingBox(const AABB &box0, const AABB &box1
    {
        Vec3 small(
            std::fmin(box0.min.x, box1.min.x),
            std::fmin(box0.min.y, box1.min.y),
            std::fmin(box0.min.z, box1.min.z));
        Vec3 big(
            std::fmax(box0.max.x, box1.max.x),
            std::fmax(box0.max.y, box1.max.y),
            std::fmax(box0.max.z, box1.max.z));
        return AABB(small, big);
    }
};
```

Fig.33

  2. **Intersection test:** This function checks if a given ray intersects the BVH node.
     First, check if the ray intersects the left and right leaf nodes.
     If the node has child nodes, the function recursively checks if the ray intersects the left and right child nodes.
     If the ray intersects either the left or right subtree, indicating a potential intersection with a shape within the BVH. (Fig.34)

```
struct BVHNode
{
    AABB box;
    Shape *left;
    Shape *right;
    BVHNode *leftNode;
    BVHNode *rightNode;

    BVHNode() : left(nullptr), right(nullptr), leftNode(nullptr),

    bool intersect(const Ray &ray, float &t) const
    {
        if (!box.intersect(ray, 0.001f, t))
            return false;

        bool hitLeft = left && left->intersect(ray, t);
        bool hitRight = right && right->intersect(ray, t);

        if (leftNode && leftNode->intersect(ray, t))
            hitLeft = true;
        if (rightNode && rightNode->intersect(ray, t))
            hitRight = true;

        return hitLeft || hitRight;
```

Fig.34

3. **Build up BVH tree:**

   Single Shape: If there's only one shape, it becomes a leaf node. The shape is assigned to the left pointer of the node. The bounding box of the shape is calculated and assigned to the node's box.

   Two Shapes: If there are two shapes, they become direct child nodes. The shapes are assigned to the left and right pointers. The surrounding box is assigned to the node's box. (Fig.35)

```
BVHNode *buildBVH(std::vector<Shape *> &shapes, size_t start, siz
{
    BVHNode *node = new BVHNode();
    size_t shapeCount = end - start;

    if (shapeCount == 1)
    {
        node->left = shapes[start];
        AABB box;
        shapes[start]->boundingBox(box);
        node->box = box;
    }
    else if (shapeCount == 2)
    {
        node->left = shapes[start];
        node->right = shapes[start + 1];

        AABB boxLeft, boxRight;
        shapes[start]->boundingBox(boxLeft);
        shapes[start + 1]->boundingBox(boxRight);
        node->box = AABB::surroundingBox(boxLeft, boxRight);
    }
```

Fig.35

4. **Partitioning**: The shapes are sorted based on the minimum x-coordinate of their bounding boxes. The left and right subtrees are constructed recursively. The bounding box of the current node is calculated as the surrounding box of the left and right child nodes' bounding boxes. (Fig.36)

```
else
{
    std::nth_element(shapes.begin() + start, shapes.begin() +
                     [](Shape *a, Shape *b)
                     {
                         AABB boxA, boxB;
                         a->boundingBox(boxA);
                         b->boundingBox(boxB);
                         return boxA.min.x < boxB.min.x;
                     });

    size_t mid = start + shapeCount / 2;
    node->leftNode = buildBVH(shapes, start, mid);
    node->rightNode = buildBVH(shapes, mid, end);

    node->box = AABB::surroundingBox(node->leftNode->box, nod
}

return node;
```

Fig.36

Lastly, create a function to destroy the BVH to free the space and memory. (Fig.37)

```
void destroyBVH(BVHNode *node)
{
    if (!node)
        return;
    if (node->leftNode)
        destroyBVH(node->leftNode);
    if (node->rightNode)
        destroyBVH(node->rightNode);
    delete node;
}
```

Fig.37

3. Advanced Raytracer Features

- **(a) Pixel sampling:**

    1. Anti-Aliasing: I Increased the **samplesPerPixel** value for better sampling and antialiasing.
    2. Random Sampling: I use a random distribution to jitter the rays within each pixel, averaging results for smoother edges.
    3. The rest of the code is also modified, but the main idea is above mentioned. (Fig.38)

```
void render(const std::string &renderMode, const Camera &camera, ir
{
    float brightness = 2.2f;

    const int samplesPerPixel = 16;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 1.0);
```

Fig.38

We can see that in the figure **"mirror_image.ppm"** (Fig.39~40), especially the shadow part, it looks more aliasing before pixel sampling applied.

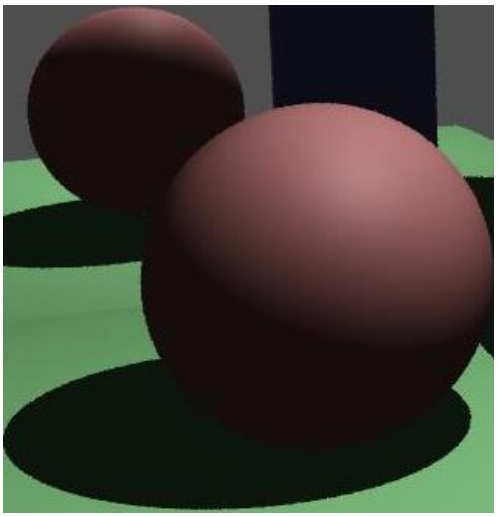


Fig.39 (Before pixel sampling)

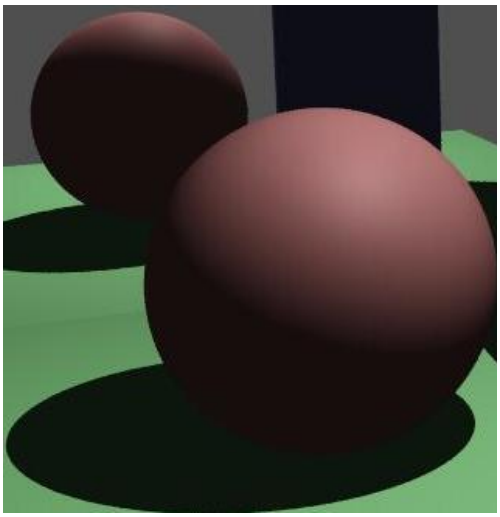


Fig.40 (After pixel sampling)

- **(b) Lens sampling:**

Implement a depth of field effect by simulating aperture in a camera lens.

1. Randomly sample points within this aperture to simulate light rays through different parts of lens. Then, adjust light ray direction:
2. Calculate the direction of each light ray based on sampled points on the aperture, the focus distance, and the size of the aperture.
3. Rays passing through the center of the aperture will be focused sharply, while those passing through the edges will be focused on different distances, achieving foreground and background blur:

However, I've done this part in the previous camera setting, so it looks simple in this part of coding. (Fig.41)

```
Camera parseCamera(const json &cameraData)
{
    Vec3 position = parseVec3(cameraData, "position");
    Vec3 lookAt = parseVec3(cameraData, "lookAt");
    Vec3 upVector = parseVec3(cameraData, "upVector");
    float fov = cameraData.at("fov").get<float>();
    float aperture = cameraData.value("aperture", 0.0f);
    float focusDistance = cameraData.value("focusDistance", 1.0f);

    return Camera(position, lookAt, upVector, fov, aperture, focusD
}
```

Fig.41

Now, I only have to modify JSON file, to give the initial parameters and change from pinhole camera to thin lens camera. (Fig.42)

```
"camera":
    {
        "type":"thinlens",
        "width":1200,
        "height":800,
        "position":[-1, 0.5, -1.5],
        "lookAt":[0.25, -0.05, 1.0],
        "upVector":[0.0, 1.0, 0.0],
        "fov":45.0,
        "focusDistance": 3.0,
        "aperture": 0.2,
        "exposure":0.1
    },
```

Fig.42

Comparison between lens sampling for **"mirror_image.ppm"**. (Fig.43~44)

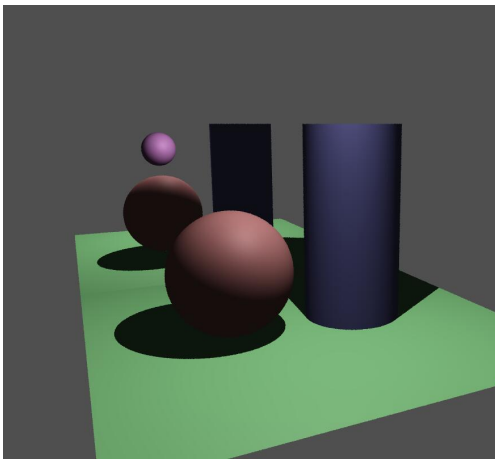


Fig.43 (Before lens sampling)

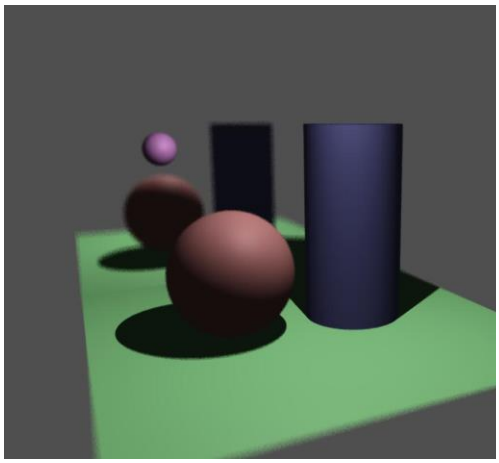


Fig.44 (After lens sampling)

- **(c) BRDF sampling:**

  Here's how the function implements:

  1. Define reflection properties based on material. Characterize the reflective properties of different materials. Determine the direction of the reflected ray through random sampling.
  2. Generate a random direction vector to represent the potential direction of the reflected ray. The sampling distribution should match the shape of the BRDF function to ensure that the sampled results align with the material's reflective properties.
  3. Adjust the weights of sampled results to simulate material effects.
  4. Calculate a weight based on the sampled direction and the value of the BRDF function at that direction. This weight represents the probability density of the sampled direction.
  5. By adjusting the weights, we can simulate the reflective effects of different materials. For example, for a specular material, most of

the weight will be concentrated in the specular reflection direction. (Fig.45)

```
Vec3 sampleBRDF(const Material &material, const Vec3 &normal, cor
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 1.0);

    float u1 = dis(gen);
    float u2 = dis(gen);
    float sinTheta = std::sqrt(1.0f - u1);
    float cosTheta = std::sqrt(u1);

    float phi = 2.0f * M_PI * u2;
    float x = sinTheta * std::cos(phi);
    float y = sinTheta * std::sin(phi);
    float z = cosTheta;

    Vec3 h(x, y, z);
    Vec3 halfVector = (h + viewDir).normalize();

    // Reflect the view direction using the sampled half vector
    return reflect(viewDir, halfVector);
}
```

Fig.45

Updated to include the **useBRDFsampling** property that decides whether to use BRDF sampling for a shape in JSON file. (Fig.46)

```
            "material":
                {
                    "ks":0.2,
                    "kd":0.8,
                    "specularexponent":10,
                    "diffusecolor":[0.94, 0.87, 0.74],
                    "specularcolor":[0.9, 0.9, 0.9],
                    "texture": "texture4.jpeg",
                    "isreflective":false,
                    "reflectivity":0.1,
                    "isrefractive":false,
                    "refractiveindex":1.0
                },
            "useBRDFsampling": true
        },
```

Fig.46

Other functions like **Shape::Sphere** have also be adjusted to align with BRDF sampling. However, all are slightly different from initial rendering code, so I don't paste to here to make explanation, as they're trivial.

Here's the object that applies BRDF sampling. (Fig.47)

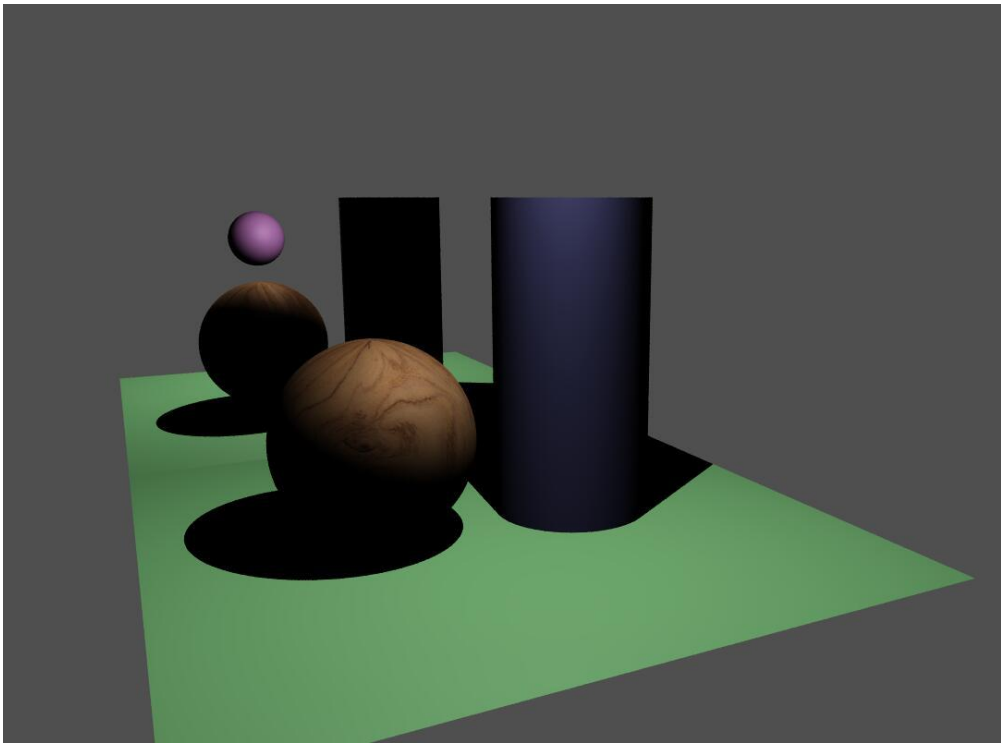


Fig.47

- **(d) Light sampling:**

  By implement the function, I can get:

  1. Higher Efficiency: Directly performing importance sampling on light sources and calculate the contribution of light sources to specific points in the scene.
  2. Noise Reduction: Capture more direct lighting contributions, especially when the direct light source is very bright or small.
  3. More realistic shadows, making shadow edges softer.
  4. Better Light Transitions: Handle the influence of different light sources on the scene, achieving more natural light transitions.

  For **directLightSampling** function: (Fig.48)

  1. Initialize a Mersenne Twister 19937 using the seed from random number generator. Create a uniform integer distribution object that generates random integers.
  2. Dereference the light pointer at the randomly chosen index to access the specific light object. Calculate the direction vector from the point to the light source's position and normalize it.
  3. Calculate the halfway direction vector, the diffuse lighting contribution, diffuse lighting color. This ensures that only light hitting the surface directly contributes to diffuse reflection.
  4. Calculate the specular reflection contribution (based on the Phong model), the specular lighting color like diffuse but use specular material properties and the calculated specular contribution. Finally, accumulate them into the final color vector.

```
Vec3 directLightSampling(const Vec3 &point, const Vec3 &normal, cor
{
    Vec3 color(0, 0, 0);
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, lights.size() - 1);
    int lightIndex = dis(gen);
    const Light &light = *lights[lightIndex];
    Vec3 lightDir = (light.position - point).normalize();
    float shadowFactor;
    if (!isInShadow(point, light, shapes, shadowFactor))
    {
        Vec3 halfwayDir = (lightDir + viewDir).normalize();
        float diff = std::max(normal.dot(lightDir), 0.0f);
        Vec3 diffuse = material.kd * diff * material.diffuseColor *
        float spec = std::pow(std::max(normal.dot(halfwayDir), 0.0f
        Vec3 specular = material.ks * spec * material.specularColor
        color += diffuse + specular;
    }
    return color * static_cast<float>(lights.size());
```

Fig.48

In view of the changes, now each shape's **getcolor** should also change to **dorectLightSampling**. (Fig.49)

```
    virtual Vec3 getColor(
        const Vec3 &p, const Vec3 &viewDir,
        const st
    {
        Vec3 normal = computeNormal(p);
        return directLightSampling(p, normal, material, viewDir, li
```

virtual Vec3 Shape::computeNormal(const Vec3 &p)

Fig.49

Here's the image applied light sampling. The effect of light and shadow looks more realistic. (Fig.50)

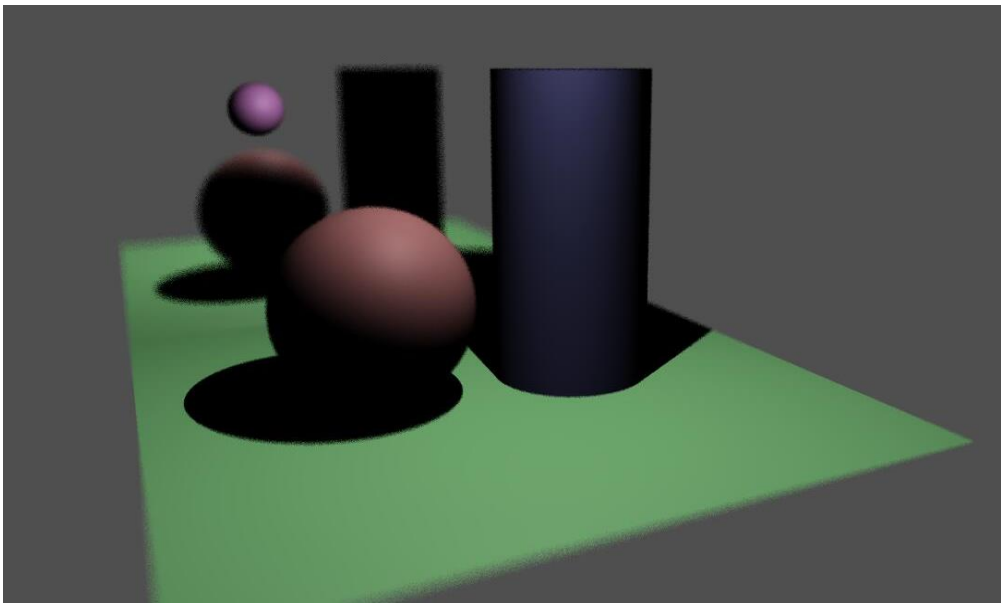


Fig.50

4. Exceptionalism

- **Tone Mapping:**

  This code implements Filmic Tone Mapping, making the figures appear more natural and visually appealing on display devices. It simulates the way a camera captures a scene. (Fig.51)

  1. Filmic Tone Mapping: Using a specific set of parameters (A, B, C, D, E, F) to achieve a more film-like appearance.
  2. Mapping Equation: Employ a filmic lambda function to map color values, derived from tone mapping formula.
  3. White Point Adjustment: The white point is adjusted to ensure that the output colors are within the typical HDR input range.

```
Vec3 toneMap(const Vec3 &hdrColor)
{
    // Parameters for Uncharted 2 tone mapping
    const float A = 0.22f;
    const float B = 0.30f;
    const float C = 0.10f;
    const float D = 0.20f;
    const float E = 0.01f;
    const float F = 0.30f;
    auto filmic = [&](float x)
    {
        return ((x * (A * x + C * B) + D * E) / (x * (A * x + B) +
    };
    Vec3 mappedColor(
        filmic(hdrColor.x),
        filmic(hdrColor.y),
        filmic(hdrColor.z));

    // Use a white point that corresponds to typical HDR input
    const float whitePoint = filmic(11.2f);
    mappedColor *= 1.0f / whitePoint;

    return mappedColor;
```

Fig.51

- **Self-Created Texture Photos:**

  Use **Canva** to create multiple texture figures which were used to apply on objects by myself (Fig.52). Created texture can be seen in Fig.53~55.

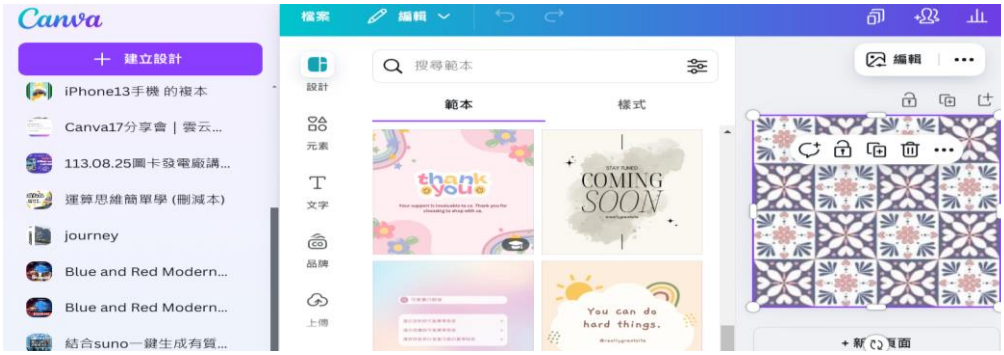


Fig.52

Here’s my self-created texture figure. (Fig.53~55)

Fig.53

Fig.54

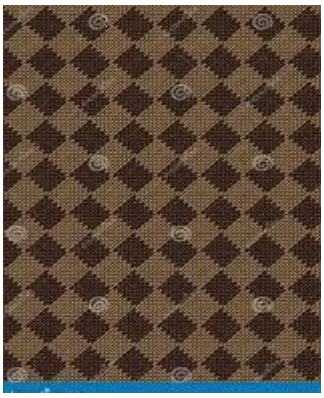


Fig.55

- **Self-Created Scene:**

  Below is my self-created scene, and it includes all of the functions implemented. (Fig.56)

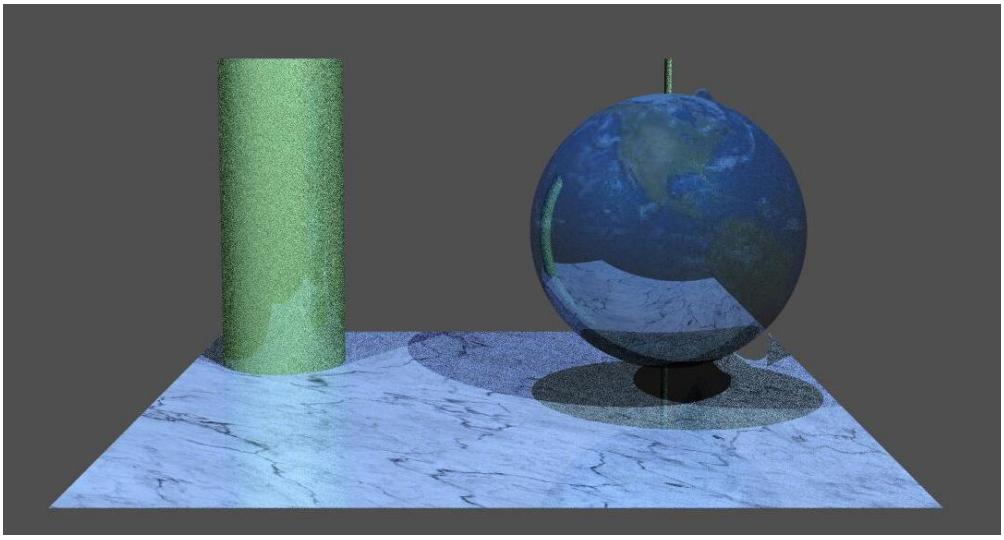


Fig.56

When we look at the tellurion, I applied the earth texture on it. (Fig.57)

```
"type": "sphere",
"center": [0.5, 0, 1],
"radius": 0.4,
"material": {
    "ks": 0.3,
    "kd": 0.7,
    "specularexponent": 50,
    "diffusecolor": [0.2, 0.5, 0.8],
    "specularcolor": [1.0, 1.0, 1.0],
    "texture": "earth_texture.jpeg",
    "isreflective": true,
    "reflectivity": 0.5,
    "isrefractive": false,
    "refractiveindex": 1.0
},
"useBRDFsampling": false
```

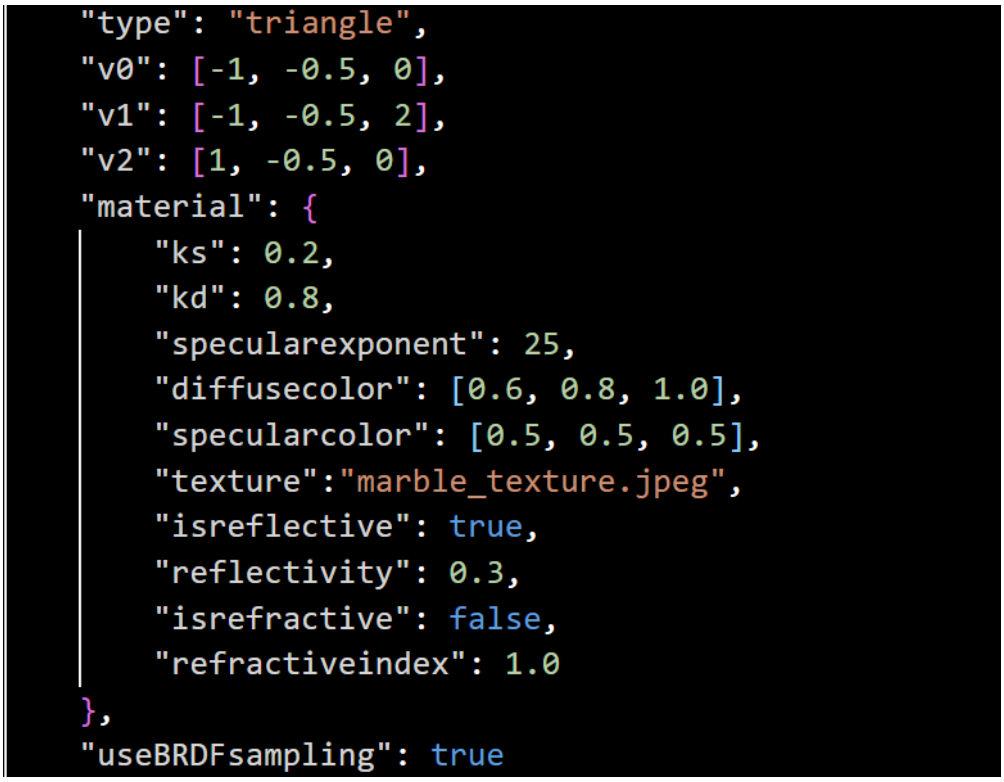
Fig.57

If we looked more specifically, we could see there's an axis of rotation in the middle of earth, just like the real tellurion. (Fig.58)

```
"type": "cylinder",
"center": [0.7, -0.5, 1.5],
"axis": [0, 1, 0],
"radius": 0.2,
"height": 1.0,
"material": {
    "ks": 0.1,
    "kd": 0.9,
    "specularexponent": 20,
    "diffusecolor":[0.5, 0.8, 0.5],
    "specularcolor":[1.0,1.0,1.0],
    "isreflective": true,
    "reflectivity": 1.0,
    "isrefractive": false,
    "refractiveindex": 1.0
},
"useBRDFsampling": false
```

Fig.58

For the plane, I applied the marble texture on it, with BRDF sampling to make it softer and more realistic. (Fig.59)



```
"type": "triangle",
"v0": [-1, -0.5, 0],
"v1": [-1, -0.5, 2],
"v2": [1, -0.5, 0],
"material": {
    "ks": 0.2,
    "kd": 0.8,
    "specularexponent": 25,
    "diffusecolor": [0.6, 0.8, 1.0],
    "specularcolor": [0.5, 0.5, 0.5],
    "texture":"marble_texture.jpeg",
    "isreflective": true,
    "reflectivity": 0.3,
    "isrefractive": false,
    "refractiveindex": 1.0
},
"useBRDFsampling": true
```

Fig.59

Moreover, I have created another scene for this coursework.

For this one **“funny_face.ppm”**, I want to create a cute face using my own creating shape. With a ambient light background as robot’s head, two marble-like eyes, entirely-shadowed as mouth, and a hollow cylinder to be his nose. (Fig.60)

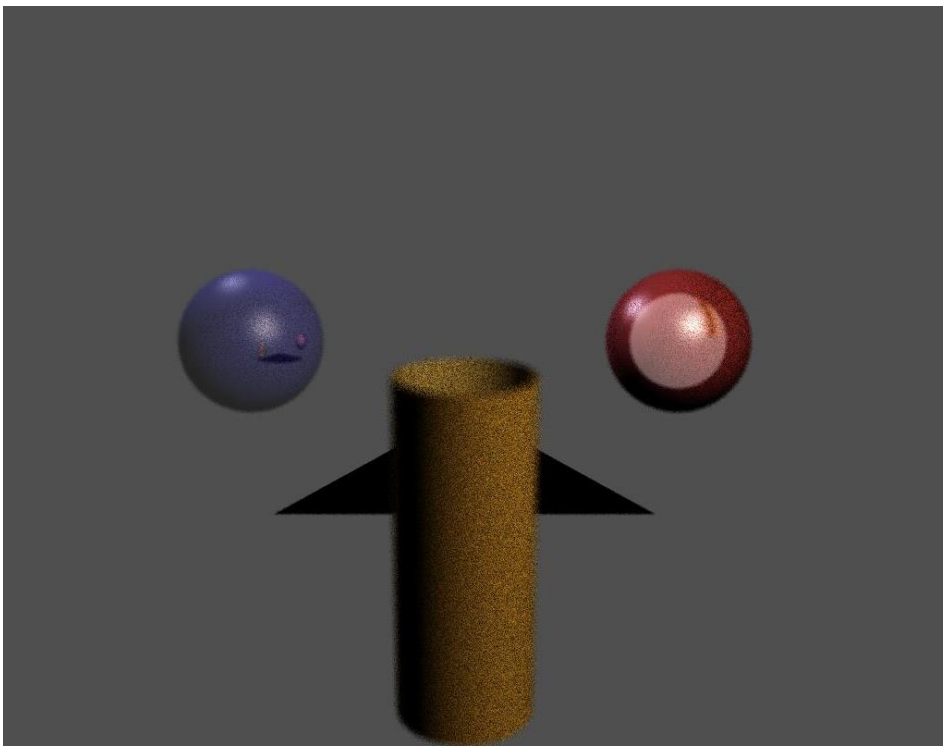


Fig.60

- **Light Sources:**

  Except for the designated point light sources, I create spotlight sources and its related functions. (Fig.61)

```
"lightsources": [
    {
        "type": "pointlight",
        "position": [3, 5, -5],
        "intensity": [1.0, 0.9, 0.8]
    },
    {
        "type": "spotlight",
        "position": [-3, 4, 3],
        "direction": [0, -1, -1],
        "angle": 45,
        "intensity": [0.8, 0.8, 1.0]
    }
],
```

Fig.61

Some things to mention:

1. **All code captured in this report are written without comments, you can see the completed version in my cpp file.**
2. **Please run the JSON file provided by me, as I modified some parameters that original JSON may not work in my MakeFile.**
3. **How to run?**
   First, run **make**. (Fig.62)

```
chu@DESKTOP-0LL52DB:/mnt/c/Users/user/Downloads/Computer Graphics Renderin
s2749500/Code$ make
```

Fig.62

Then it'll automatically produce o file for all modules. (Fig.63)

```
g++ -Wall -O3 -std=c++20 -I. -I/path/to/json/include -c s2749500.cpp -o s2
.o
g++ -Wall -O3 -std=c++20 -I. -I/path/to/json/include -c Vec3.cpp -o Vec3.o
g++ -Wall -O3 -std=c++20 -I. -I/path/to/json/include -c Texture.cpp -o Tex

g++ -Wall -O3 -std=c++20 -I. -I/path/to/json/include -c Material.cpp -o Ma
```

Fig.63

Until finished, it'll list all o files. (Fig.64)

```
g++ -Wall -O3 -std=c++20 -I. -I/path/to/json/include -o raytracer s2749500
3.o Texture.o Material.o Light.o Ray.o AABB.o Camera.o parseCamera.o sampl
o Shape.o BVHNode.o Sphere.o Triangle.o Cylinder.o readJson.o generateRay.
eMaterial.o isInShadow.o blinnPhongShading.o directLightSampling.o reflect
ract.o castRay.o loadLights.o toneMap.o render.o processScene.o
```

Fig.64

Finally, run **./raytracer**, about 1 minute, ppm files produce. (Fig.65)

```
chu@DESKTOP-0LL52DB:/mnt/c/Users/user/Downloads/Computer Graphics Renderin
s2749500/Code$ ./raytracer
```

Fig.65