Computer Graphics: Rendering CW2

s2192181

Contents

Basic raytracer features

(a) Image Writing

The image writing is implemented in *helpers.cpp* as a function `writePPM`. The function writes to an output file, specifying: ppm type as P3, height, width and colour range to be (0-255). The pixel buffer is then written to the file, outputting r, g and b values for each pixel, making sure to cast rgb values from float to int. This allows complex transformations on float rgb colour during rendering and valid integer values in our final ppm output.

(b) Virtual Pin-hole Camera

The virtual pin-hole camera is implemented in *camera.cpp* as a class hierarchy, with the `PinholeCamera` class inheriting from the base `Camera` class. The camera is defined with its position, the direction it is looking at and the upward vector, along with image resolution, field of view, exposure value and aspect ratio.

The core functionality of the pin-hole camera lies in generating a ray direction for each pixel, which is achieved in the `getRayDirection` method. This method is essential as it is used in all raytracers for shooting rays from the camera and rendering the scene.

(c) Intersection tests

Intersection tests are implemented in *shapes.cpp* as overridden `intersect` methods in the derived classes for spheres, cylinders, and triangles. This determines whether a ray intersects with objects in the scene, and if so, to compute the relevant information such as the intersection point, surface normal, and material properties.

Sphere Intersection: Sphere intersections are calculated by solving the quadratic equation:

$$t^2\vec{d}\cdot\vec{d}+2t(\vec{o}-\vec{c})\cdot\vec{d}+(\vec{o}-\vec{c})\cdot(\vec{o}-\vec{c})-r^2=0$$

where $\vec{o}$ is the ray origin, $\vec{d}$ is the ray direction, $\vec{c}$ is the sphere centre, and r is the radius. The discriminant (b^2-4ac) determines whether an intersection occurs. If positive, the smallest positive root is used to compute the hit point and normal.

Cylinder Intersection: For cylinders, a similar quadratic equation is used:

$$A=\vec{d_\perp}\cdot\vec{d_\perp},\quad B=2(\vec{d_\perp}\cdot\vec{o}_\perp),\quad C=\vec{o}_\perp\cdot\vec{o}_\perp-r^2$$

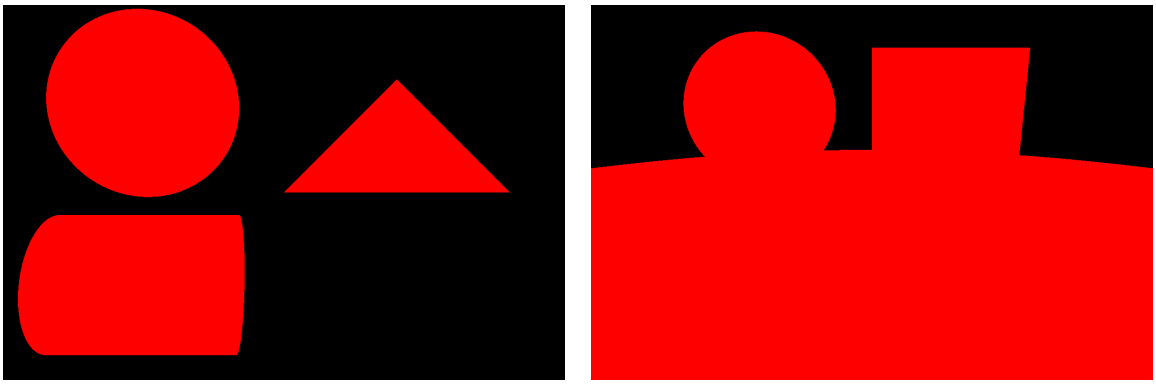
where $\vec{d_\perp}$ and $\vec{o}_\perp$ are the components perpendicular to the cylinder axis. The roots are tested for validity within the cylinder's height, and cap intersections are checked by projecting the ray onto the axis.

Triangle Intersection: Triangle intersections use the Möller-Trumbore algorithm to calculate the intersection point with the triangle's plane:

$$t=\frac{(\vec{v_0}-\vec{o})\cdot\vec{n}}{\vec{d}\cdot\vec{n}}$$

where $\vec{n}$ is the triangle's normal and $\vec{v_0}$ is one vertex. Barycentric coordinates are computed to verify if the point lies within the triangle.

All intersection tests populate an `Intersection` structure with relevant information such as the hit point, normal, and distance (t). As shown in Figure 1, we can see that the intersection tests work well with binary rendering.



(a) Binary primitives (b) Binary scene

Figure 1: Binary render intersections

(d) Blinn-Phong Shading

Blinn-Phong shading is implemented in *raytracer.cpp* for the inherited `PhongTracer` class, specifically in `traceRayRecursive`. This shading model combines ambient, diffuse, and specular components to create lighting effects. The implementation uses a recursive ray tracing approach to calculate the colour of each pixel based on the interaction of rays with the scene's geometry, materials, and lights.

Diffuse Lighting: Diffuse lighting is computed using Lambert's cosine law and for each light source is calculated as:

$$I_{\text{diffuse}} = k_d \cdot \max(0, \vec{n} \cdot \vec{l})$$

where k_d is the diffuse coefficient, $\vec{n}$ is the surface normal, and $\vec{l}$ is the light direction.

Specular Lighting: Specular highlights are calculated using the Blinn-Phong model, with a half-vector ($\vec{h}$), the normalised vector halfway between the light direction and the view direction:

$$I_{\text{specular}} = k_s \cdot (\max(0, \vec{n} \cdot \vec{h}))^{\alpha}$$

where k_s is the specular coefficient, α is the shininess (specular exponent), $\vec{n}$ is the surface normal, and $\vec{h}$ is the half-vector.

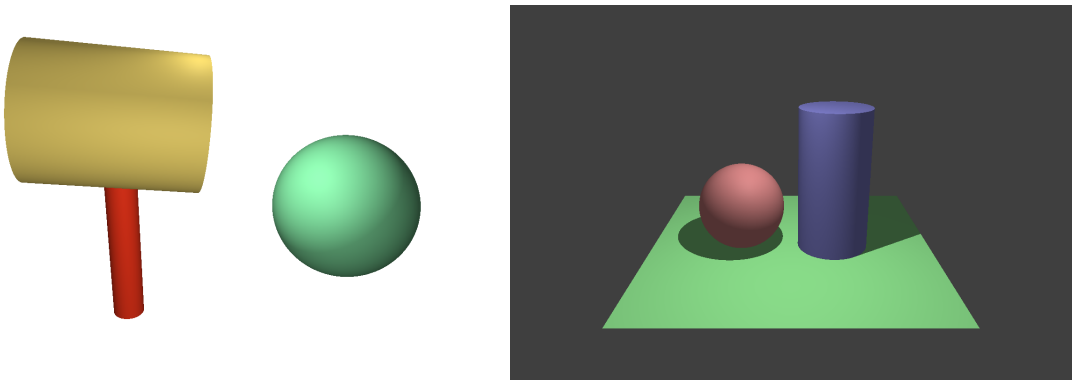
Ambient Lighting: A global ambient term is added to simulate indirect illumination. This is calculated as:

$$I_{\text{ambient}} = k_a \cdot C_{\text{ambient}}$$

where k_a is the ambient coefficient and C_{ambient} is the material or texture colour.

Resulting Pixel Colour: The final pixel colour is the sum of the ambient, diffuse, and specular contributions, scaled by shadow factors.

This implementation of Blinn-Phong shading can be seen in Figure 2, where the shading more accurately models lighting and material effects than binary, enhancing the realism of the rendered scene.



(a) Phong shading (b) Simple phong

Figure 2: Blinn-Phong shading renders

(e) Shadows

Shadows are implemented in the `traceRayRecursive` function within the `PhongTracer` class by using shadow rays to determine whether a light source contributes to the illumination of a point. This approach depicts shadow rendering in the scene by identifying occlusions caused by other objects.

Shadow Ray Creation: For each light source, a shadow ray is cast from the intersection point slightly offset along the surface normal:

$$\text{shadowRay.origin} = \text{hitPoint} + \text{normal} \cdot \epsilon$$

where $\epsilon = 0.00001$ prevents self-intersection errors. The direction of the shadow ray is:

$$\text{shadowRay.direction} = \frac{\text{light.position} - \text{hitPoint}}{|\text{light.position} - \text{hitPoint}|}$$

Occlusion Check: The shadow ray is tested for intersections with other objects in the scene. If a shadow ray intersects an object and the intersection distance (t) is less than the distance to the light source, the point is considered in shadow for that light.

Light Contribution: If the shadow ray does not encounter any occluding objects, the point is illuminated by the light source. The contribution of the light to the final colour includes diffuse and specular lighting, scaled by the shadow factor:

$$\text{colour} += (\text{diffuse} + \text{specular}) \cdot \text{shadowFactor}$$

where shadowFactor accounts for the light's intensity and the absence of shadows.
This ensures that completely shadowed areas are not entirely black but retain some ambient illumination.

Resulting Shadows: The shadows can also be seen in Figure 4, where Figure 4(a) accurately handles shadows for multiple light sources, resulting in overlapping shadows with varying strength.

(f) Tone Mapping

Tone mapping is implemented in *tonemap.cpp* with the functions: `linearToneMap`, `reinhardToneMap` and `ACESFittedToneMap`. Linear tone mapping is used for `PhongTracer` and ACES is used for `PathTracer`.

Linear Tone Mapping: The linear tone mapping technique scales the colour values by an exposure factor and clamps the resulting values to the displayable range. The formula is:

$$\text{scaled} = \text{colour} \cdot \text{exposure}$$

This linear approach to tone mapping seen in Figure 2 and Figure 4 is simple and efficient, making it suitable for scenes where basic brightness control are sufficient.

ACES Fitted Tone Mapping: The ACES (Academy Color Encoding System) fitted tone mapping technique applies a non-linear transformation to colours, aiming to compress their luminance while maintaining detail and colour fidelity. This method seen in Figure 3(a) is more advanced than linear tone mapping and better suited for realistic rendering.

$$\text{hdrColor} = \text{color} \cdot \text{exposure}$$

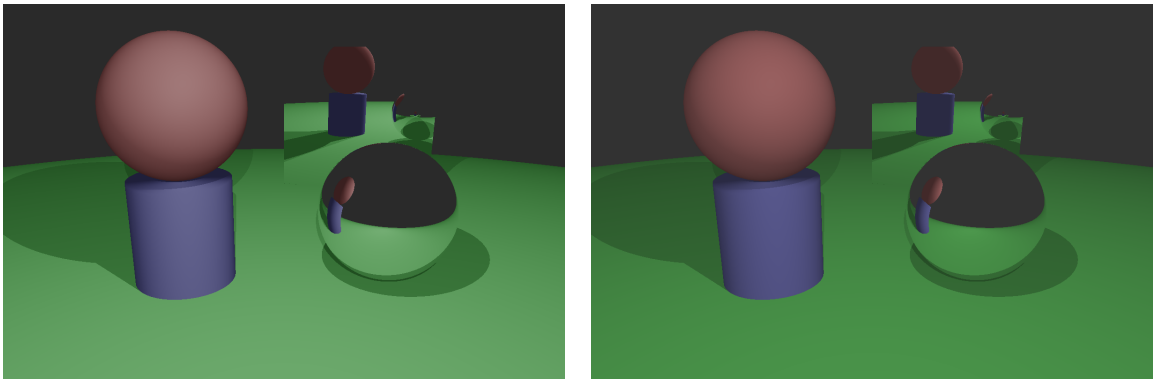
The input HDR colour is first scaled by an exposure factor to control brightness, similar to linear tone mapping.

Tone Mapping Formula: The core of ACES tone mapping is a non-linear transformation that balances highlights and shadows. The tone-mapped colour is computed as:

$$\text{tonemappedColor} = \frac{\text{hdrColor} \cdot (\text{hdrColor} + A) - B}{\text{hdrColor} \cdot (\text{hdrColor} \cdot C + D) + E}$$

where the constants A, B, C, D, E are empirically derived to achieve it's tone mapping:

- A adjusts the curve's contrast.
- B offsets the intensity for highlights.
- C and D define the steepness of the curve.
- E acts as a lower bound to prevent division by zero.



(a) ACES tonemapping

(b) Reinhard tonemapping

Figure 3: Alternative Tonemappers

(g) Reflection

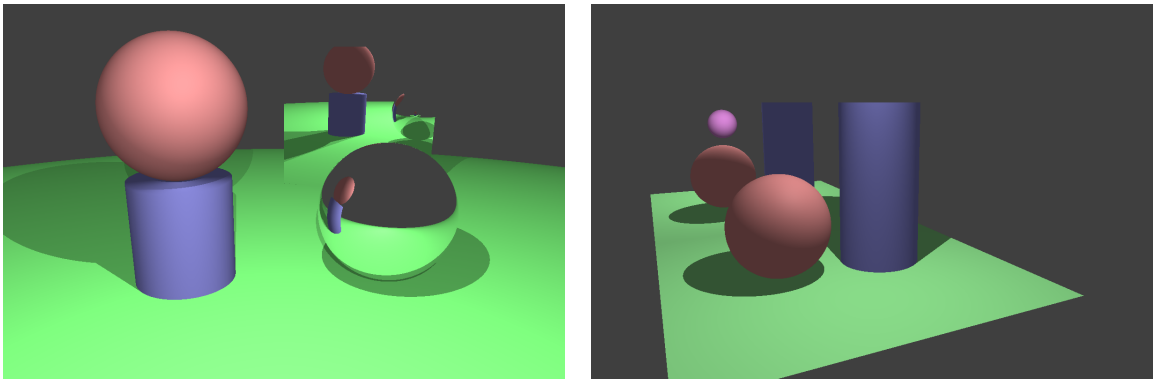
Reflections are handled in the `traceRayRecursive` method. Materials with reflective properties determine how light interacts with their surfaces, contributing to effects such as mirrors.

Reflection: If the material is reflective, the direction of the reflected ray is calculated using the reflection equation:

$$\vec{r} = \vec{d} - 2(\vec{d} \cdot \vec{n})\vec{n}$$

where $\vec{r}$ is the reflected ray direction, $\vec{d}$ is the incoming ray direction, and $\vec{n}$ is the surface normal. The reflected ray is traced recursively to calculate the colour contributed by the reflection. Fresnel reflectance is also calculated to determine the intensity of the reflection based on the viewing angle.

Reflection can be seen in Figure 4, with both images contain a mirror that fully reflects the scene. Furthermore, we can see that in Figure 4(a), the mirror comprised of two triangles reflects the sphere mirror's reflection in itself!



(a) Phong Scene

(b) Mirror image

Figure 4: Shadows and reflection renders

(h) Refraction

Refractions are handled in the `traceRayRecursive` method. Refractions occur when light passes through a transparent material, altering its direction due to the change in refractive index, contributing to effects such as see through glass.

Refraction: For refractive materials, Snell's Law is used to calculate the refracted ray:

$$\eta = \frac{n_1}{n_2}, \quad \vec{t} = \eta\vec{d} + (\eta \cos\theta_i - \cos\theta_t)\vec{n}$$

where η is the ratio of refractive indices, and $\cos\theta_t$ is computed using:

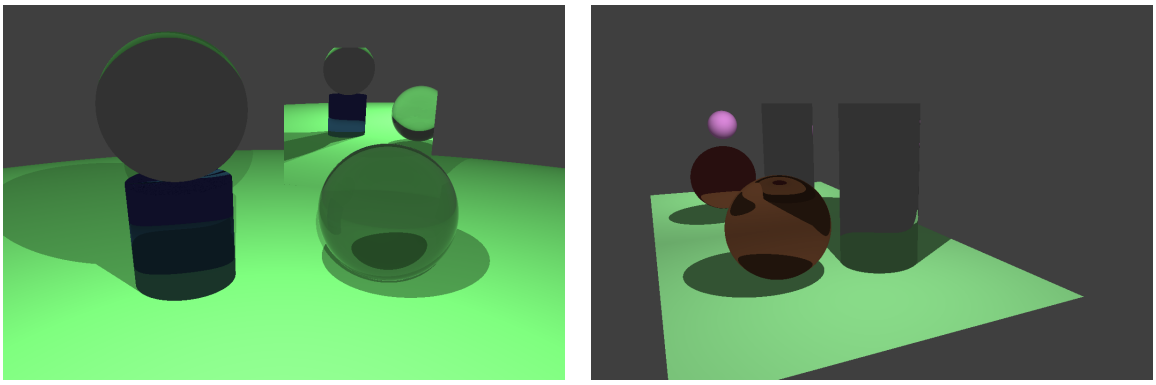
$$\cos\theta_t = \sqrt{1 - \eta^2(1 - \cos^2\theta_i)}$$

If total internal reflection occurs ($\sin^2\theta_t > 1$), the material is fully reflective. Otherwise, the refracted ray is traced recursively.

Blending Reflected and Refracted Rays: For materials with both reflective and refractive properties, the final colour is a weighted combination of the reflected and refracted rays:

$$\text{colour} = R \cdot \text{reflectedColour} + (1 - R) \cdot \text{refractedColour}$$

Refraction can be seen in Figure 5, with both images containing refractive objects, with varying refractive indexes. Furthermore, we can see that in Figure 5(a), the first sphere refracts and distorts the ground as well as the second sphere showing both reflective and refractive properties blended together using fresnel's.



(a) Refractive scene

(b) Mirror with refractive shapes

Figure 5: refraction renders

Intermediate raytracer features

(a) Textures

Textures are implemented in *texture.cpp* with a `Texture` class to add surface details to objects by mapping 2D images onto their surfaces. The texture data is loaded from PPM files and sampled using UV coordinates for colour interpolation.

Texture Loading: The constructor of the `Texture` class loads texture data from PPM files stored in a predefined `textures/` directory:

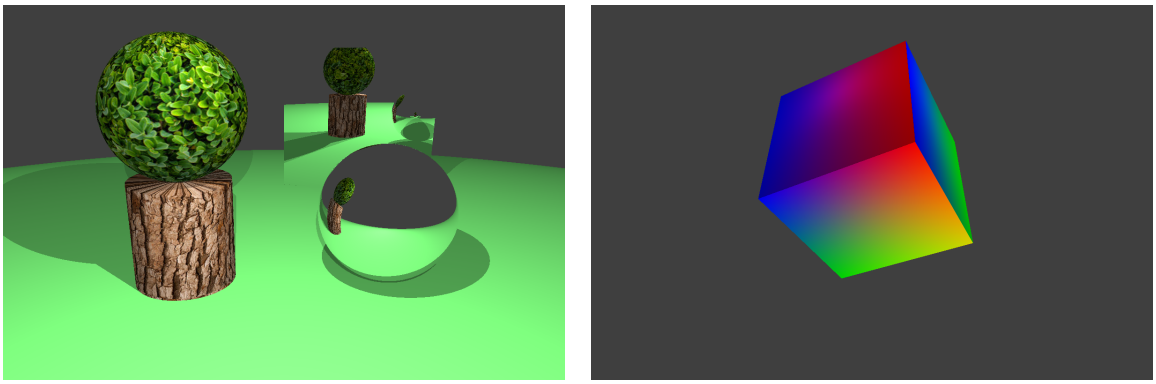
Texture Sampling: To retrieve the texture's colour at specific coordinates, the `sample` method uses UV mapping:

- **UV Coordinates:** Floating-point values in the range [0,1] representing a point on the texture. UV mapping aligns these coordinates with the surface of a 3D object, allowing textures to conform to the object's shape.
- **Tiling:** UV coordinates are wrapped using modulo between [0,1] to correctly tile textures.
- **Bilinear Interpolation:** The sampling process performs bilinear interpolation to smoothly blend between four adjacent pixels:

$$\text{Colour} = c_{00} \cdot (1 - dx) \cdot (1 - dy) + c_{10} \cdot dx \cdot (1 - dy) + c_{01} \cdot (1 - dx) \cdot dy + c_{11} \cdot dx \cdot dy$$

  where $c_{00}, c_{10}, c_{01}, c_{11}$ are the colours of the four surrounding pixels, and dx, dy are the fractional distances from the top-left pixel derived from UV coordinates.
- **Pixel Access:** The `getPixel` method retrieves a specific pixel from the texture using UV values.

Example textures are displayed in Figure 6, with a tree made from a sphere + cylinder texture combination and a rainbow cube made from triangles!



(a) Tree scene (b) Rainbow cube!

Figure 6: Textures on display

(b) Acceleration hierarchy

The acceleration hierarchy is implemented in *bvh.cpp* through a `BoundingVolume` class, `BVHNode` struct and a `BVH` class (Bounding Volume Hierarchy). The BVH groups objects into a hierarchical tree structure, where each node contains a bounding volume (AABB) that encompasses its child nodes or objects.

Bounding Volume: Each node in the BVH contains an Axis-Aligned Bounding Box (AABB) that defines the minimum and maximum extents of the objects it encompasses. The `BoundingVolume` class provides:

- **Expansion:** Expands an AABB to include another bounding volume.
- **Intersection Test:** Determines if a ray intersects the AABB by comparing ray extents along each axis using slab testing. Early termination occurs if the ray misses the bounding volume.

BVH Construction: The BVH is constructed recursively by dividing the scene into smaller subsets of objects:

- **Bounding Volume Computation:** The bounding volume for a node is calculated using the extents of all objects within the range.
- **Partitioning:** Objects are partitioned along the largest axis of the bounding volume, determined by comparing its extents.
- **Leaf Nodes:** If the number of objects in a node is below a threshold (e.g. 2), the node is marked as a leaf, and objects are stored directly.

BVH Traversal: Ray traversal through the BVH is implemented recursively:

- **Intersection Test with AABB:** Each node is tested for ray intersection with its bounding volume. If the ray does not intersect, traversal stops for that branch.
- **Leaf Node Intersections:** For leaf nodes, the ray is tested against all objects in the node, and the closest intersection is recorded.
- **Recursive Traversal:** For internal nodes, the ray is tested against both child nodes. If either child intersects, the traversal continues.

The BVH reduces the complexity of intersection tests by limiting ray-object checks to only the objects within potentially intersecting bounding volumes. This hierarchical approach ensures scalability for scenes with large numbers of objects, as irrelevant portions of the scene are culled early in the traversal process. Table 1 tabulates the speed differences between BVH and non-BVH.

Table 1: Comparison of Phong render times with and without BVH

Image	Render Time (with BVH)	Render Time (without BVH)
400_spheres.ppm	2.90312s	9.08831s
1600_shapes.ppm	15.5755s	54.5966s
6400_shapes.ppm	57.4737s	233.9919s
12800_shapes.ppm	108.841s	453.877s

Advanced raytracer features

(a) Pixel sampling

Pixel sampling (anti-aliasing) is implemented in *raytracer.cpp* as a method `tracePixel`, of the `PathTracer` class. The method employs multiple samples per pixel and combines their results to calculate the final pixel colour.

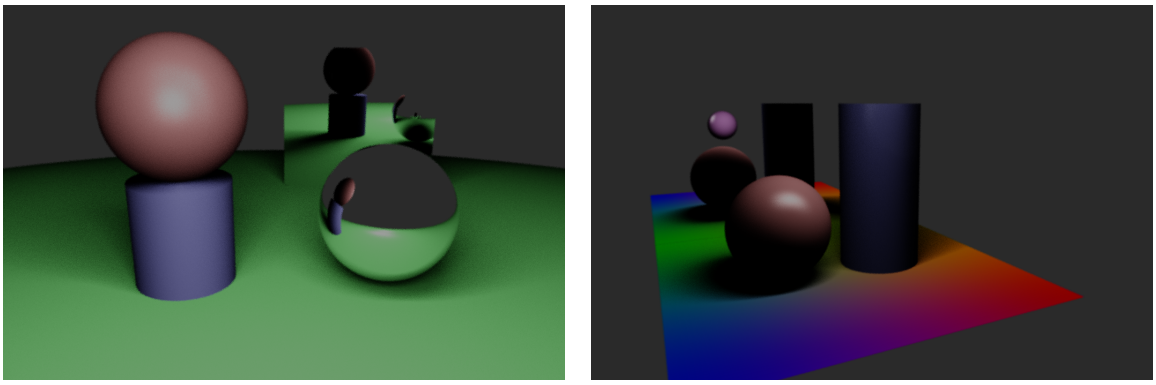
Anti-alias smoothing: Pixel sampling works by generating multiple rays through each pixel, with slight offsets (jitter), and averaging the results. The random jitter introduces variation, effectively mimicking real-world light behaviour and resulting in higher-quality renders.

- **Random Jitter:** For each sample, a random offset (`jitterX`, `jitterY`) is generated within the pixel boundaries:
$$\text{jitterX}, \text{jitterY} \in [-0.5, 0.5]$$
This ensures that the rays are distributed uniformly within the pixel, avoiding sharp transitions.
- **Ray Direction:** The offset is added to the pixel coordinates, and the camera's `getRayDirection` method generates the ray direction for the jittered coordinates:
$$\vec{r} = \text{camera.getRayDirection}(x + \text{jitterX}, y + \text{jitterY})$$
- **Recursive Ray Tracing:** Each jittered ray is traced using the `traceRayRecursive` method, returning a sample colour based on the scene geometry, materials, and lighting.
- **Accumulation and Averaging:** The colours of all samples are accumulated and averaged to compute the final pixel colour:
$$\text{finalColour} = \frac{1}{N} \sum_{i=1}^{N} \text{sampleColour}_i$$
where N is the number of samples per pixel (e.g., 30 in this implementation).

We can see visibly smoother edges in Figure 7 with the pathtracer renders, especially when compared to the jagged pixel edges that the phong raytracer produces.

(b) Lens sampling

Lens sampling is implemented in *camera.cpp* within the `getRayDirection` method of the `ApertureCamera` class. This simulates depth of field effects, where objects at different distances from the camera appear in or out of focus. It is achieved by randomly sampling points on the lens aperture and calculating rays that converge at the focal plane.



(a) Pathtracer scene

(b) Mirror scene with rainbow floor

Figure 7: Pixel, lens, light sampling (soft shadows)

Depth of field: Lens sampling models a camera with a finite aperture size, introducing depth of field. Instead of a pinhole, rays are traced from random points on the lens aperture toward a specific focal point.

- **Base Ray Direction:** The method first computes the ray direction as if the camera were a pinhole camera:
$$\vec{r}_{\text{dir}} = \text{forward} + \text{right} \cdot \text{ndcX} + \text{cameraUp} \cdot \text{ndcY}$$
where ndcX and ndcY are the normalised device coordinates of the pixel. This ensures the ray passes through the pixel on the image plane.
- **Focal Point:** The ray direction is used to calculate the focal point on the focal plane, at a distance equal to the focal length:
$$\text{focalPoint} = \text{position} + \vec{r}_{\text{dir}} \cdot \text{focalDistance}$$
- **Random Point on Lens Aperture:** A random point on the lens aperture is sampled to introduce variation:
$$r = \text{lensRadius} \cdot \sqrt{\text{random}(0,1)}$$
$$\theta = 2\pi \cdot \text{random}(0,1)$$
The random polar coordinates (r, θ) are converted to Cartesian coordinates to get the lens offset:
$$\text{lensOffset} = (r \cdot \cos\theta, r \cdot \sin\theta, 0)$$
- **New Ray Origin:** The ray origin is shifted to the sampled lens position:
$$\text{newOrigin} = \text{position} + \text{lensOffset}$$
- **Final Ray Direction:** The new ray direction is calculated as the vector from the lens position to the focal point:
$$\vec{r}_{\text{final}} = (\text{focalPoint} - \text{newOrigin}).\text{normalise}()$$

Lens sampling accurately simulates depth of field effects, adding realism to the rendered images in Figure 7 by mimicking the behaviour of real-world cameras with finite apertures.

(c) BRDF sampling

BRDF (Bidirectional Reflectance Distribution Function) sampling is implemented in the `traceRayRecursive` method of the `pathtracer` class. This simulates indirect lighting and surface interactions with realistic reflections and rough surfaces. The implementation uses the Cook-Torrance BRDF model combined with GGX sampling for the distribution of microfacet normals, allowing for accurate representation of glossy and rough surfaces.

Sampling Components:

- **Roughness and GGX Sampling:** The GGX distribution is used to model surface roughness. The roughness parameter is derived from the material's specular exponent:

$$\text{roughness} = \sqrt{\frac{2}{\text{specularExponent} + 2}}$$

  Multiple samples of the halfway vector ($\vec{h}$) are generated using the GGX sampling method.

- **Reflected Direction:** The reflected ray direction ($\vec{r}$) is computed using the reflection equation:

$$\vec{r} = \vec{d} - 2(\vec{d} \cdot \vec{n})\vec{n}$$

  where $\vec{d}$ is the incoming ray direction, and $\vec{n}$ is the surface normal. The reflected ray is traced recursively to calculate the contribution from indirect light.

- **BRDF Evaluation:** For each sampled direction, the Cook-Torrance BRDF is evaluated using the halfway vector ($\vec{h}$) and includes Fresnel, geometry, and distribution terms:

$$\text{BRDF} = \text{Fresnel} \cdot \text{Geometry} \cdot \text{Distribution}$$

  The material's specular colour (F_0) and roughness are used in the evaluation.

- **PDF Weighting:** The contribution of each sample is weighted by the GGX Probability Density Function (PDF) and the cosine-weighted Lambertian term:

$$\text{weight} = \frac{\text{BRDF} \cdot (\vec{n} \cdot \vec{r})}{\text{PDF}}$$

  where $(\vec{n} \cdot \vec{r})$ is the cosine term between the surface normal and the reflected direction.

- **Accumulation and Averaging:** The contributions from all samples are accumulated and averaged:

$$\text{reflectedColour} = \frac{1}{N} \sum_{i=1}^{N} \text{sampleColour} \cdot \text{weight}$$

  where N is the number of samples. This ensures the final reflected colour is energy-conserving and accounts for the roughness and reflectance properties of the material.

Fresnel Reflectance: The Fresnel-Schlick approximation is used to blend reflection and refraction contributions based on the viewing angle:

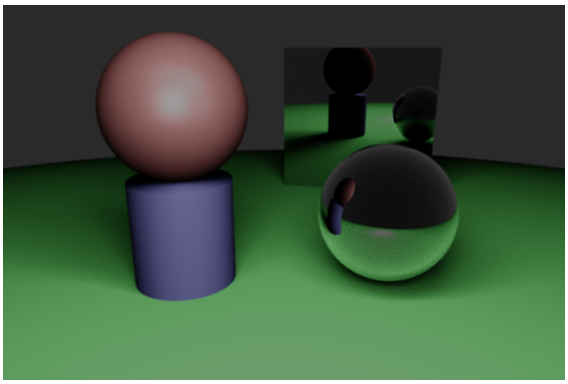
$$R = R_0 + (1 - R_0)(1 - \cos\theta_i)^5$$

where R_0 is the reflectance at normal incidence, and θ_i is the angle between the view direction and the surface normal.

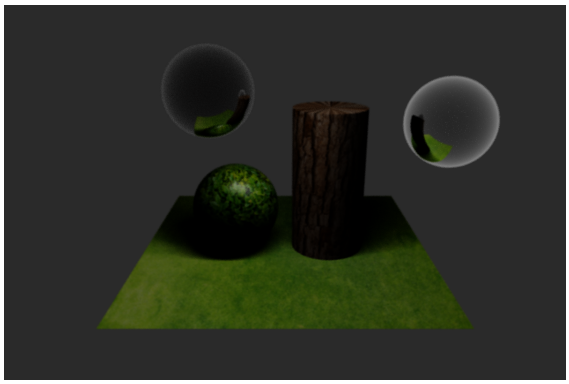
BRDF sampling enables accurate representation of glossy and rough surfaces, allowing for realistic indirect lighting effects. The use of GGX sampling provides efficient handling of roughness, while the Cook-Torrance model ensures physically based reflections. The results of Cook-Torrance fancy BRDF can be seen in Figure 8, where the flat and sphere mirror look much more realistic, with the outline of the flat mirror being clearly visible which was not the case beforehand.

(d) Light sampling

Light sampling is implemented in the`traceRayRecursive` method of the `pathtracer` class. This computes the contribution of area lights to the illumination of a surface point. By sampling multiple points on the light surface, the implementation achieves soft shadows and realistic lighting effects.



(a) Pathtracer scene



(b) Tree with a haircut

Figure 8: Fancy BRDF sampling

Sampling Steps:

- **Light Sampling:** A random point on the light surface is sampled using the `samplePoint` function provided by the `AreaLight` class. This generates a point on the light's rectangular area.
- **Shadow Ray Testing:** For each sampled point, a shadow ray is cast from the intersection point (`hitPoint`) to the light sample point. A small offset (ϵ) is applied to avoid self-intersection.

  If the shadow ray intersects any object before reaching the light sample, the point is considered in shadow, and no contribution from this sample is added.
- **Diffuse Contribution:** If the point is not in shadow, the diffuse reflection is calculated using Lambert's cosine law:

  $$I_{\text{diffuse}} = k_d \cdot (\vec{n} \cdot \vec{l}) \cdot \text{lightIntensity}$$

  where k_d is the diffuse coefficient, $\vec{n}$ is the surface normal, $\vec{l}$ is the light direction, and `lightIntensity` is the intensity of the light.
- **Specular Contribution:** The specular reflection is calculated using the Blinn-Phong model:

  $$I_{\text{specular}} = k_s \cdot (\max(0, \vec{v} \cdot \vec{r}))^{\alpha} \cdot \text{lightIntensity}$$

  where k_s is the specular coefficient, $\vec{v}$ is the view direction, $\vec{r}$ is the reflection direction, and α is the specular exponent.
- **Light Area Normalization:** The contribution from each sample is normalized by the area of the light:

  $$\text{sampleContribution} = \frac{I_{\text{diffuse}} + I_{\text{specular}}}{\text{lightArea}}$$
- **Accumulation and Averaging:** Contributions from all samples are accumulated and averaged to compute the total contribution of the light:

  $$\text{lightContribution} = \frac{1}{N} \sum_{i=1}^{N} \text{sampleContribution}$$

  where N is the number of samples.

The effects of light sampling can be seen in the soft shadows in Figures 7 and 8. This creates a visible improvement in rendered shadows when compared to the Blinn-Phong shading.

Exceptional raytracer features

(a) Cook-Torrance BRDF

The **Cook-Torrance Bidirectional Reflectance Distribution Function (BRDF)** extends the basic microfacet approach by incorporating realistic physical effects that accurately simulates the way light interacts with rough surfaces. It is a significant improvement over simpler models like Phong or Blinn-Phong, as it incorporates key physical phenomena, enabling a more realistic rendering of materials.

Key Features of Cook-Torrance:

- **Microfacet Theory:** The model assumes a surface is composed of countless tiny "microfacets," each acting as a perfect mirror. The surface roughness determines how these microfacets are oriented, influencing how light reflects.
- **Energy Conservation:** Cook-Torrance ensures that the total reflected light never exceeds the incoming light, a critical aspect of physically accurate rendering.
- **Three Components:** The BRDF is computed as the product of:
  - **Fresnel Term:** Describes how light reflects at grazing angles, enhancing reflectivity at shallow viewing angles.
  - **Geometric Attenuation:** Models shadowing and masking effects, accounting for how some microfacets block light from reaching others.
  - **Distribution Term:** Controls the density of microfacets oriented to reflect light towards the viewer, based on the surface roughness.

Effects:

- **Dynamic Highlights:** Highlights change dynamically with the viewing angle and roughness, creating visually engaging effects that make objects appear more lifelike and three-dimensional.
- **Realism Without Complexity:** While grounded in physics, the model balances accuracy and computational efficiency, making it suitable for real-time applications like games.
- **Scientific Basis:** The model is rooted in the physics of light interaction, making it both an artistic tool and a scientifically accurate method for simulating real-world behaviour.

The inclusion of the Cook-Torrance BRDF in a raytracer showcases its ability to bridge the gap between art and physics, providing a realistic and sophisticated representation of surface details. The effects of Cook-Torrance BRDF can be seen in Figure 8, with the realistic mirror outlining, glass balls and material roughness making the same tree textures used in Figure 6 far more realistic.

(b) Caustics (Partially working)

Caustics are implemented in *photon.cpp* and *raytracer.cpp*. They occur when light rays interact with materials such as glass or water, creating intricate patterns of concentrated light on nearby surfaces. In the pathtracer, caustics are implemented using photon mapping techniques to capture and render these effects.

Photon Emission: The `emitPhotons` method generates photons from area lights in the scene:

- **Photon Direction:** Each photon is assigned a direction sampled from a hemisphere oriented around the surface normal of the light source:

$$\vec{d} = \text{sampleHemisphere}(\vec{n})$$

  Randomised directions ensure an even distribution of photons.

- **Photon Energy:** Photons carry energy proportional to the light source's intensity, ensuring their contribution to the scene is physically accurate.
- **Photon Tracing:** Each photon is traced through the scene using the `tracePhoton` method. Interactions with reflective or refractive surfaces determine whether the photon is stored in the photon map or continues tracing.

Photon Storage and Caustics Gathering:

- **Photon Map:** When photons interact with diffuse surfaces, their position, direction, and energy are stored in the photon map. This data is later used to gather and render caustics.
- **Query for Caustics:** In the `gatherCaustics` method, photons within a given radius of the intersection point are queried. Their contribution is weighted based on proximity:

$$\text{weight} = \max\left(0, 1 - \frac{\text{distance}^2}{\text{radius}^2}\right)$$

  This ensures closer photons have a stronger influence, simulating the natural falloff of light.
- **Normalisation:** The accumulated photon energy is normalised by the query area:

$$\text{caustics} = \frac{\text{photonEnergy}}{\pi \cdot \text{radius}^2}$$

Caustics in the Final Render: Caustics are integrated into the final colour for diffuse surfaces by adding the gathered caustic contribution:

- **Indirect Illumination:** For non-reflective, non-refractive materials, caustics are added as an indirect light component:

$$\text{finalColour} = \text{directIllumination} + \text{indirectIllumination}$$

- **Clamping and Limits:** The caustic contribution is clamped to ensure valid intensity levels, preventing over-brightening or artefacts.

Visual Results: Figure 8 demonstrates the slight impact of photon-mapped caustics in the render. Caustic patterns from refractive spheres and reflective surfaces appear mildly on objects.