S2666210

CW2 Report - Raytracing Renderer

How to run: `make run ARGS="arg1 arg2 arg3 arg4"`
Where `arg1` specifies the json scene file, and the remaining optional arguments are to enable BVH and/or antialiasing. Please make sure to flag BVH before antialiasing if using both.
Examples:
- BVH: `make run ARGS="path/to/json --bvh"`
- Antialiasing (default is 4 samples per pixel): `make run ARGS="path/to/json --aa"`
- BVH and 16 samples per pixel antialiasing: `make run ARGS="path/to/json --bvh -aa 16"`

1. Basic raytracer features

a) Image writing

Create and write each pixel's colour to `rendered_image.ppm`.

`vector3.h` :
- `vector3` class to represent pixels' colours as RGB values.

`colour.h` :
- `write_colour` function to write a pixel to output file.

b) Virtual pin-hole camera

Parse json scene file to retrieve camera information.

`ray.h` :
- `ray` class: each ray has an origin and a direction.

`camera.h` :
- `Camera` class with a constructor that translates attributes from the json scene file into `vector3` objects required in functions.
- `get_ray` function which creates and returns the ray corresponding to a pixel's location on the image, from the camera.

Note: I used the nlohmann json library to parse files.

c) Intersection tests (binary)

For each pixel in the image, calculate the corresponding ray from the camera and check if it intersects with a shape.

`scene.cpp` :
- `load_from_json` function: amongst others, parses the json scene file and creates `Shape` objects, adding them to the `Scene`.
- `shade_binary` function: return red if intersection detected, black otherwise.
- Each ray tests for intersection with each shape in the scene. Each type of shape has its own `intersects` function implemented based on the shape's dimensions and position (see `sphere.h`, `triangle.h`, `cylinder.h`). The cylinder implementation was quite tricky, since intersection had to be checked with the top and bottom caps, as well as the body.

See binary_primitives.ppm and binary_scene.ppm.

d) Blinn-Phong shading

The same intersection logic is used as in binary rendering above. If intersection occurs, three colours are computed then combined: local, reflection and refraction colours.

`scene.cpp` :

- `load_from_json` function: sets rendermode to binary or Blinn Phong according to the scene json file.
- `shade_surface` function: calls functions for the three colour components and combines the results.
- `compute_blinn_phong` function: for each light in the scene and given a shape, determines the local colour by combining diffuse and specular information of the shape.

See simple_phong.ppm.

e) Shadows

`scene.cpp` :

- `compute_blinn_phong` also checks for shadows by calling `compute_shadow_factor` function, which casts a shadow ray from the shape to the light source and checks for intersection with an obstructing object.
- The shadow is indicated as a factor: 0.1 if in shadow, 1.0 if fully lit.

See simple_phong.ppm.

f) Tone mapping

Default tone mapping is linear and simply uses the exposure coefficient of the scene. Two other tone mapping functions can be applied, and specified in the json file through a "`tone_mapping`" parameter in "`camera`".

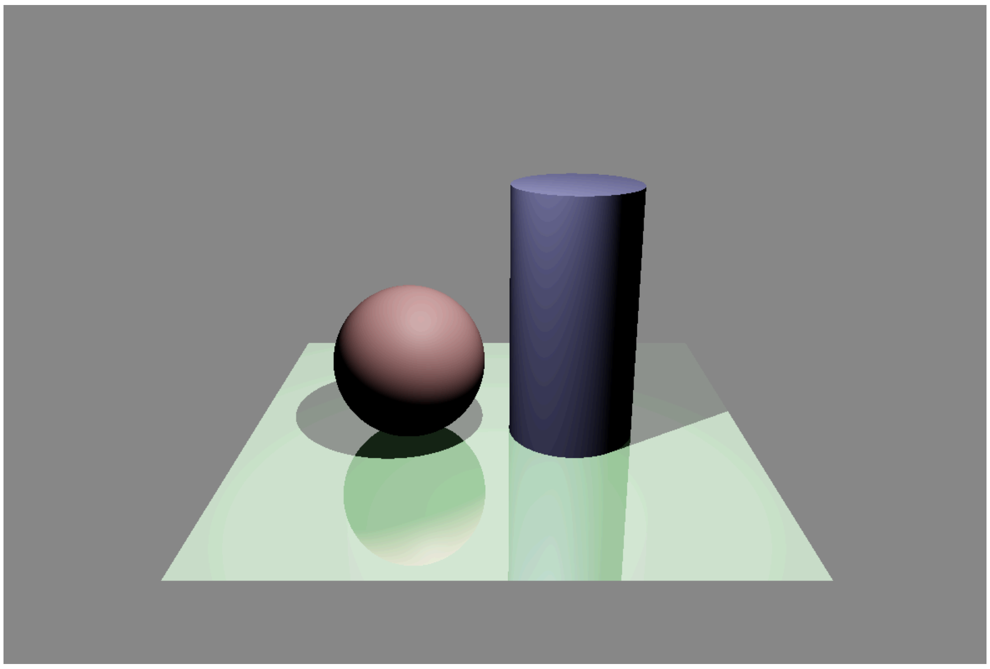


Figure 1: high exposure example (set to 3.0 in simple_phong.json).

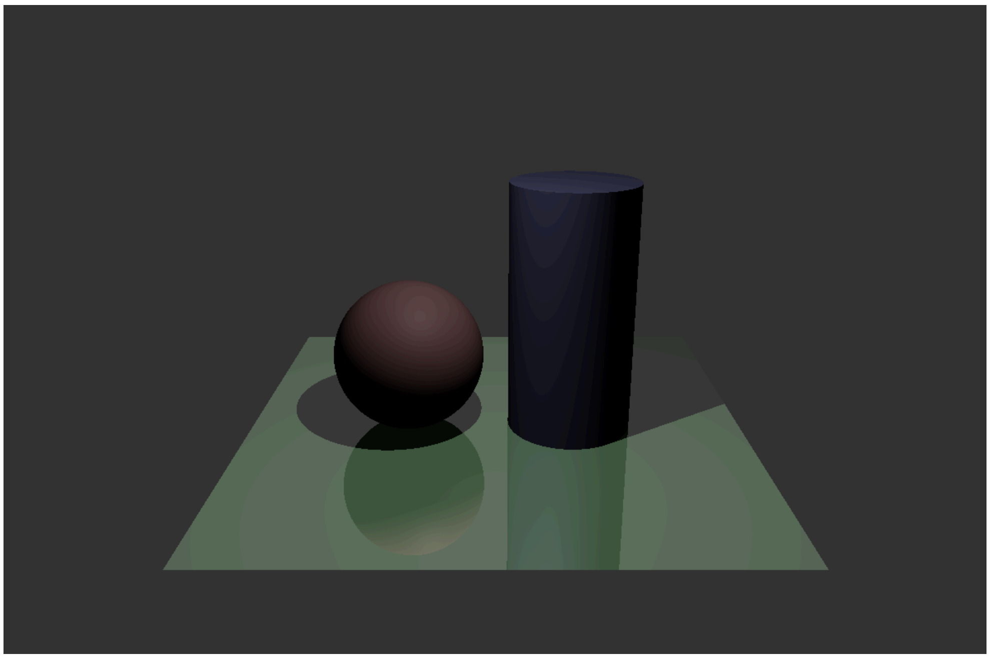


Figure 2: Reinhard tone mapping (“tone_mapping”: “reinhard”)

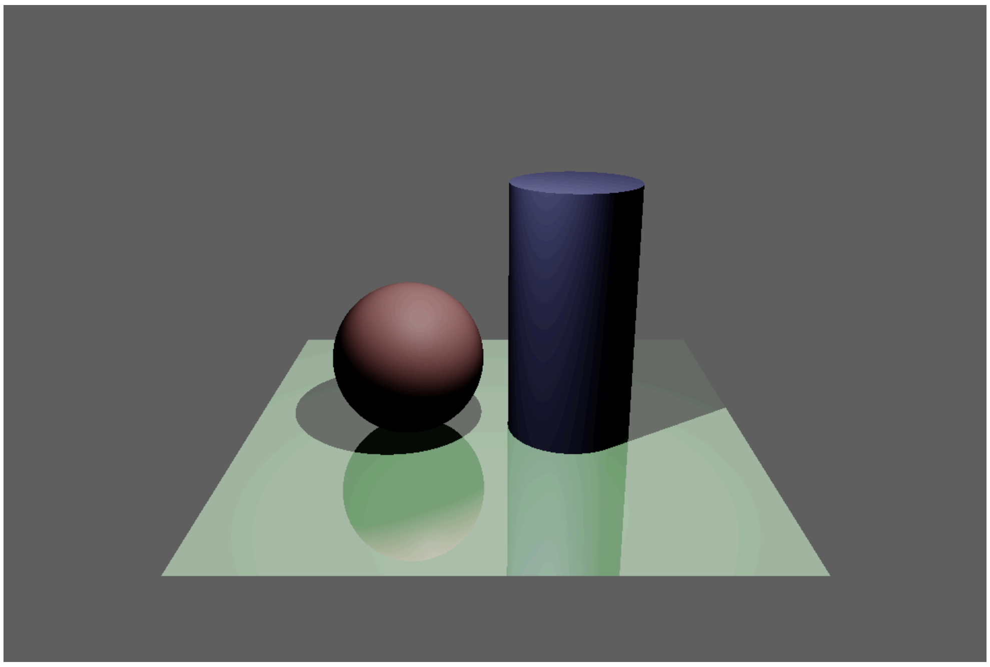


Figure 3: Aces tone mapping for sharper contrasts and a film-like aspect (“tone_mapping”: “aces”)

Note: the default 0.1 exposure in the provided json scene files resulted in underexposed images, so I had to increase it in most cases (e.g. 0.1 to 0.5 in scene.json).

g) Reflection

`scene.cpp` :

- `compute_reflection` function: called in `compute_blinn_phong`, recursively shades the reflected ray `nbounces` times to compute reflection colour at a point.
- Specularity is taken into account for more or less diffuse reflections.

See phong_scene.ppm and mirror_image.ppm.

h) Refraction

`scene.cpp` :

- `compute_refraction` function: called in `compute_blinn_phong`, recursively shades the refracted ray `nbounces` times to compute reflection colour at a point, using the material's refractive index and Snell's law to adjust the refracted ray's direction.

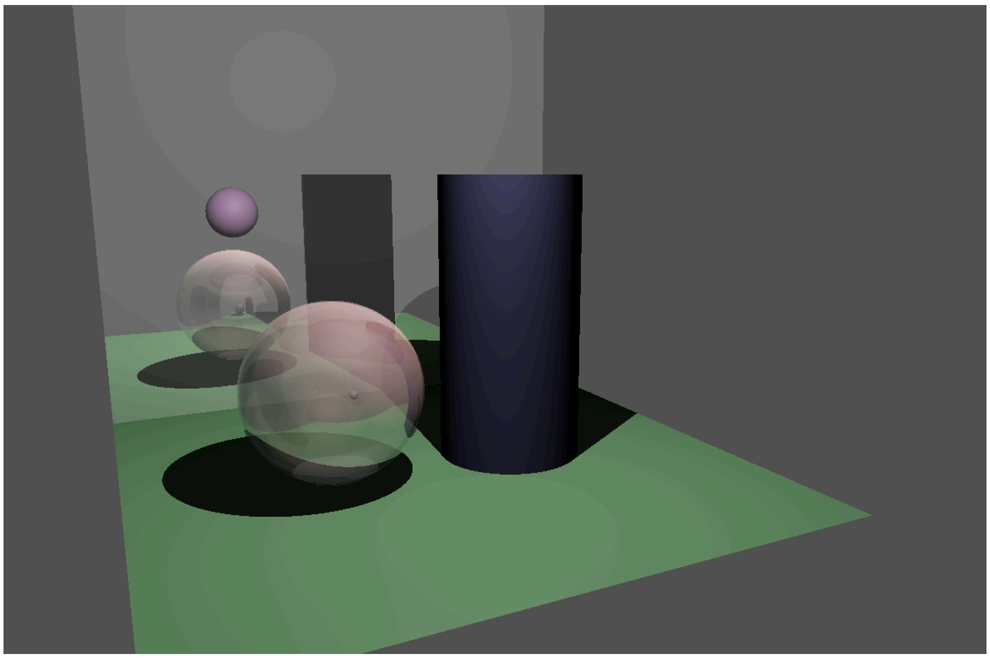


Figure 4: mirror_image with a fully refractive sphere and specular exponent of 20.

Evaluation: despite the implementation of Snell's law, ray distortion is not very apparent. Still, the sphere in Figure 4 looks quite similar to glass.

2. Intermediate raytracer features

a) Textures

`scene.cpp` :

- `load_from_json` function: loads each shape's texture as an `Image`, if any `texture_file` is provided in the scene json file.
- `compute_blinn_phong` function: calls `get_uv` to map a 3D point on a given shape's surface to a 2D position, which allows mapping to a texture. If a texture exists, `get_color_at_uv` is called and the result is used as material colour.

`image.cpp` :

- `load_image` function: creates a BMP image file to store texture from a given file path.
- `get_color_at_uv` function: returns the colour of a pixel at given coordinates in the image.

Note that for triangles, mapping was done using barycentric coordinates (a weighted combination of the UV coordinates of vertices). This ensured consistent texture alignment by applying the same UV space to all triangles on a surface. As seen in Figure XX, the grid is preserved despite the plane consisting of two triangles.

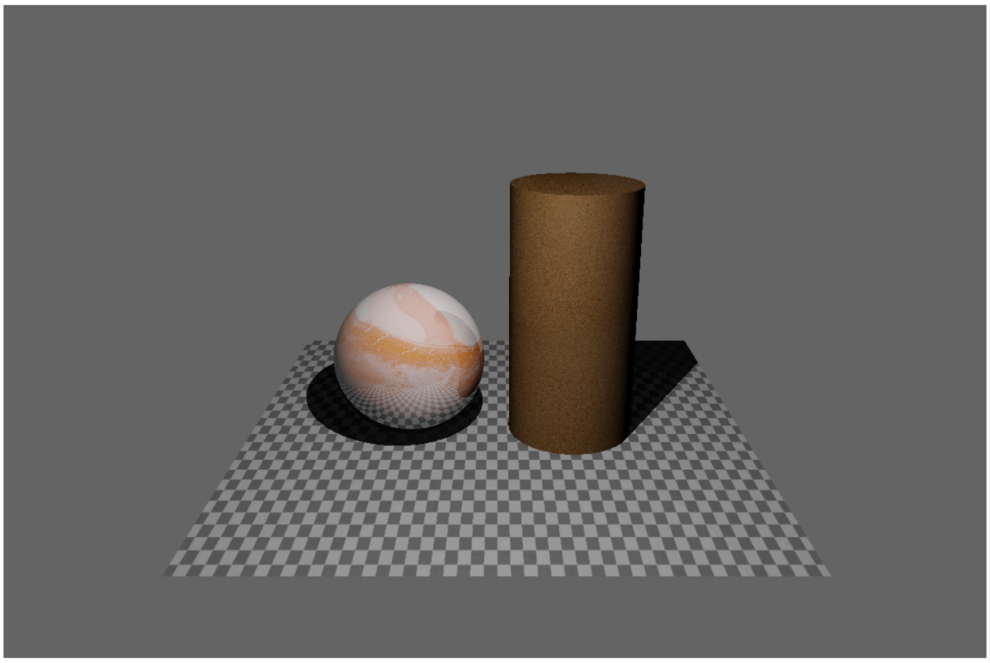


Figure 5: marble, cork and checker textures.

Evaluation: the chosen textures are quite convincing on these objects.

b) Acceleration hierarchy

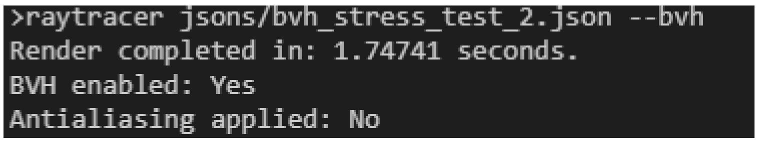
BVH acceleration can be enabled using a "`--bvh`" flag at runtime.

The BVH tree is first built before rendering, and then used when testing for intersections.

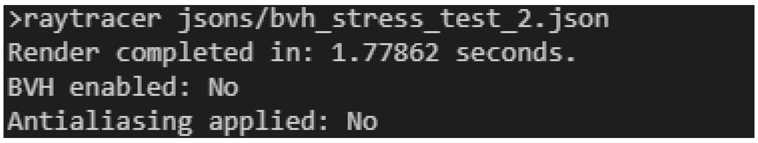
`bvh.cpp` :

- Initialization: a `BVH` is initialised with a list of shapes, and the function `build_tree` recursively builds the tree. Recursion ends and a leaf node is created when there are 2 or less shapes. Leaf nodes contain a list of primitives and their combined bounding box, while internal nodes have pointers to left and right children.
- Splitting and partitioning of the scene is done at the midpoint of the axis with the largest gap between bounding boxes.

- Intersection tests: starts at the root of the `BVH` tree and recursively checks for intersection throughout the tree (`intersects_node`). If a ray intersects a node's bounding box, the function either recurses (if internal node) or tests the ray against all shapes in the node to find the closest intersection (if leaf node).



```
>raytracer jsons/bvh_stress_test_2.json --bvh
Render completed in: 1.74741 seconds.
BVH enabled: Yes
Antialiasing applied: No
```



```
>raytracer jsons/bvh_stress_test_2.json
Render completed in: 1.77862 seconds.
BVH enabled: No
Antialiasing applied: No
```

Figure 5: render times with and without BVH.

Evaluation:

- In theory, this implementation should provide significant optimisation from splitting over the axis of largest extent, the fallback median split (ensures balanced partitions when shapes are clustered on one side), and prioritising the closest hit among left and right subtrees.
- However, there was little difference in render time when BVH was applied. This could be due to the fact that both child nodes are always checked for intersection, and all shapes get tested sequentially in leaf nodes. Splitting could also be optimised, for instance using a Surface Area Heuristic (SAH).
- Rendered images are exactly the same regardless of whether BVH is used or not.

3. Advanced raytracer features

a) Antialiasing via multi pixel sampling

Antialiasing can be enabled using a "`--aa`" flag at runtime, optionally with the number of samples per pixel (default is 4). If enabled, multi-sample antialiasing is applied by sampling multiple jittered points around each pixel, performing shading as previously done, and averaging their values. This can be seen in the `render` function in `main.cpp`.

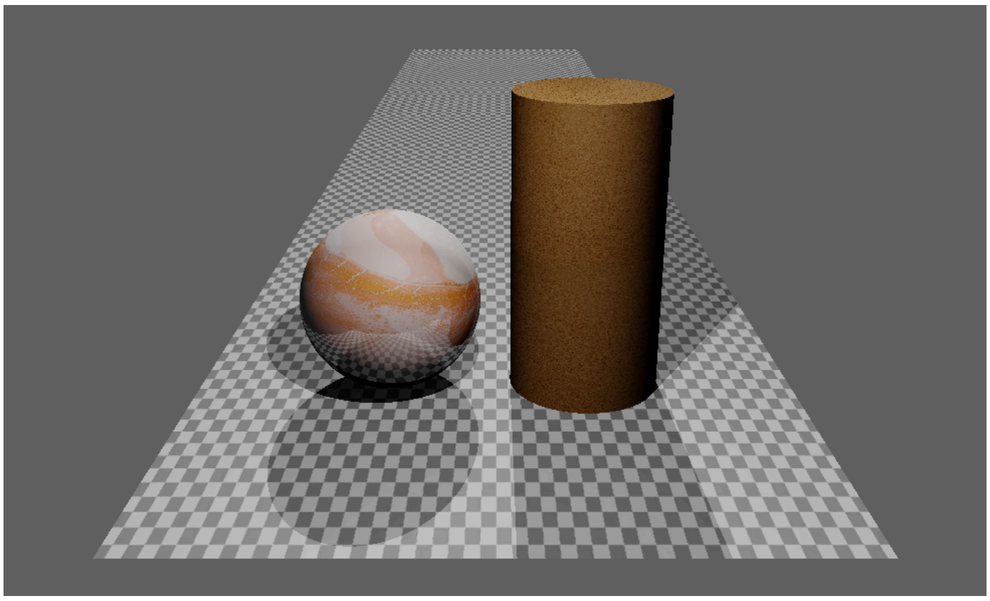


Figure 6: no antialiasing.

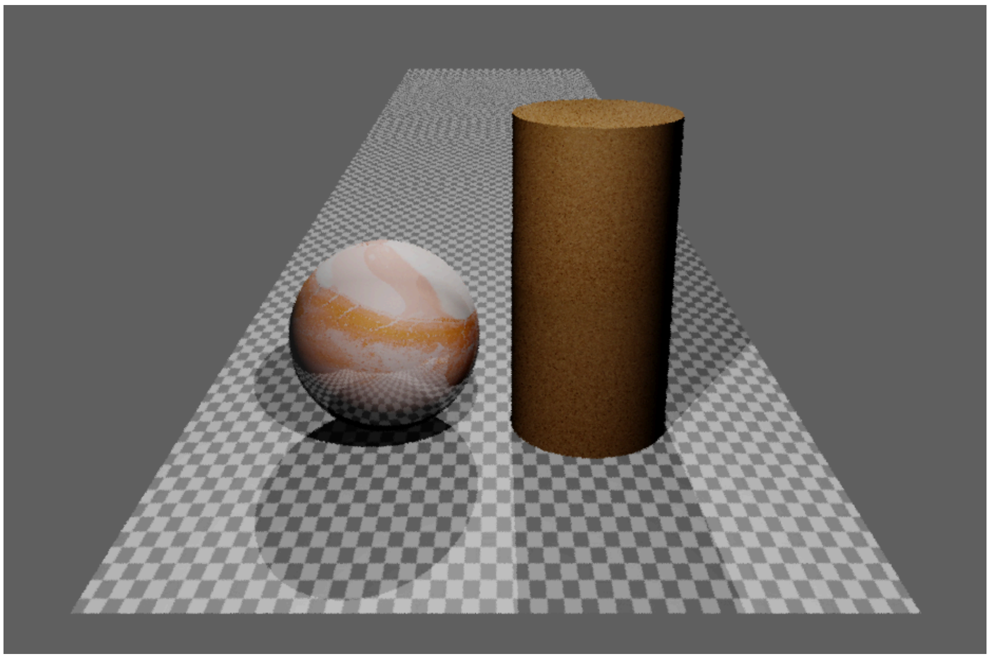


Figure 7: antialiasing with 4 samples per pixel.

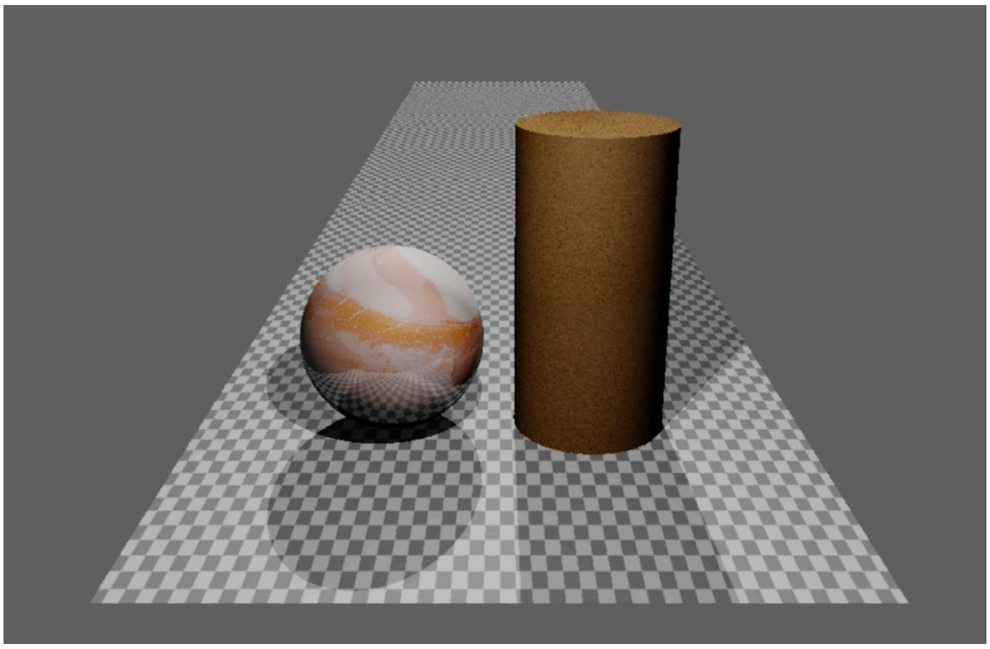
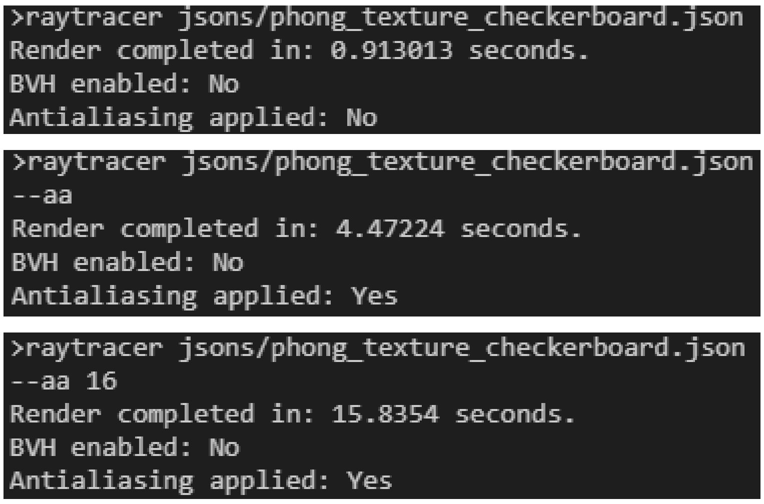


Figure 8: antialiasing with 16 samples per pixel.

Evaluation: antialiasing seems to help in the farther part of the checker pattern background for lower numbers of samples per pixel, but the overall image appears blurrier. Antialiasing significantly increases render time as more samples per pixel are taken, as expected.



```
>raytracer jsons/phong_texture_checkerboard.json
Render completed in: 0.913013 seconds.
BVH enabled: No
Antialiasing applied: No
```

```
>raytracer jsons/phong_texture_checkerboard.json
--aa
Render completed in: 4.47224 seconds.
BVH enabled: No
Antialiasing applied: Yes
```

```
>raytracer jsons/phong_texture_checkerboard.json
--aa 16
Render completed in: 15.8354 seconds.
BVH enabled: No
Antialiasing applied: Yes
```

Figure 9: render time with no antialiasing, and antialiasing with 4 and 16 samples per pixel.

b) **Lens sampling** - not implemented

c) **BRDF sampling** - not implemented

d) **Area light sampling for soft shadows**
When a scene has an area light, `compute_blinn_phong` (in `scene.cpp`) uses the same methodology to compute shadows as with point lights, except that it performs it multiple times, by randomly sampling points on the area light for each point on the shape's surface. This essentially uses multiple shadow rays to aggregate multiple jittered shadow points.

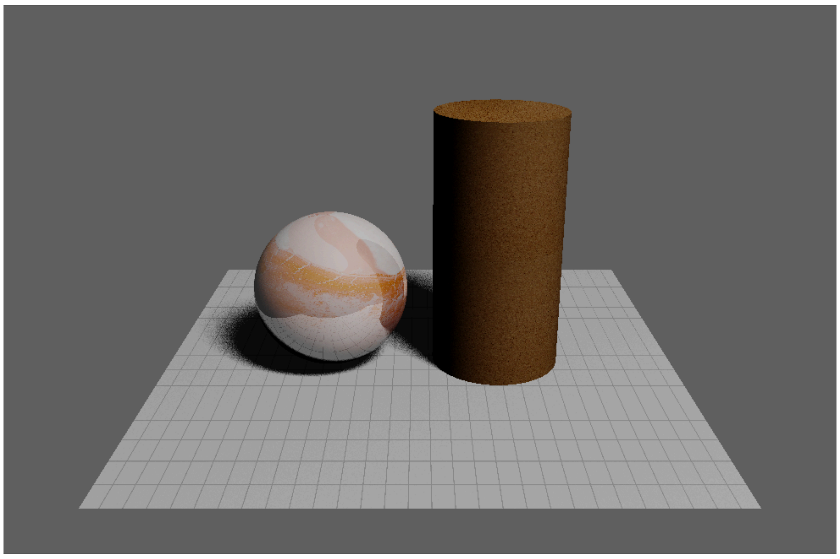


Figure 10: soft shadows with area light sampling (16 samples).

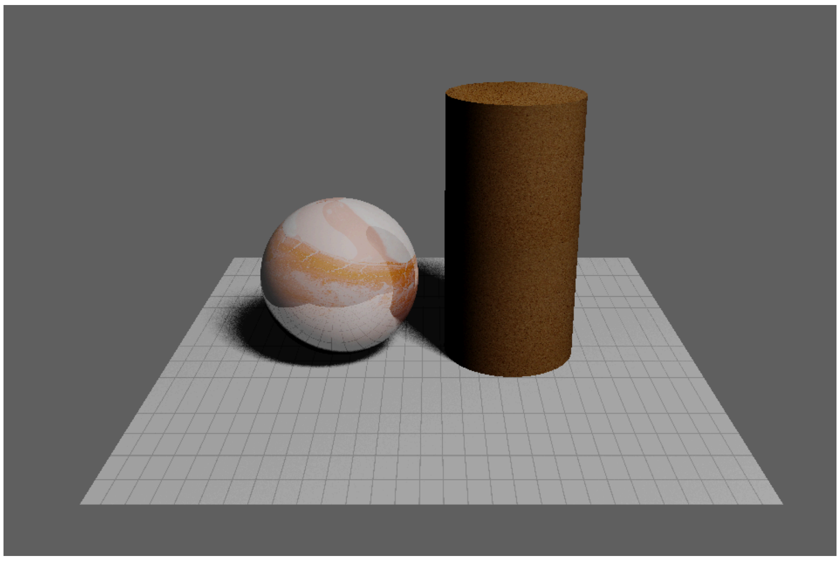


Figure 11: soft shadows with area light sampling (32 samples).

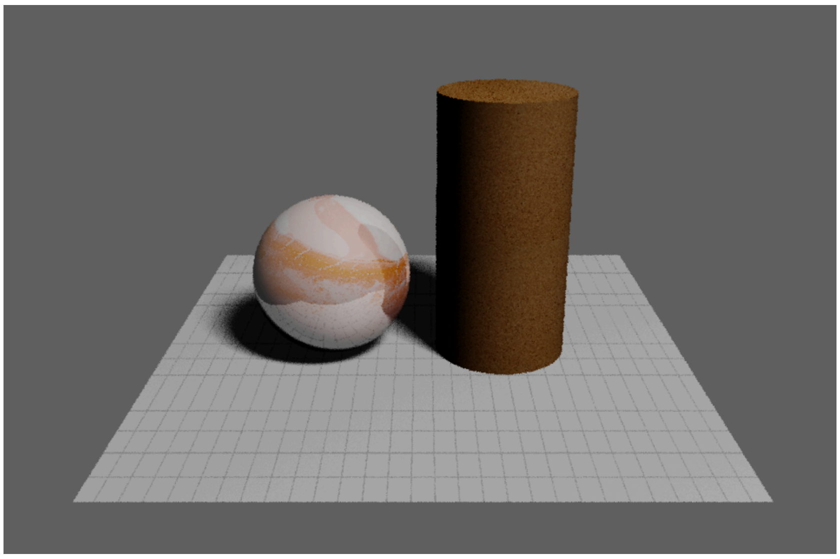


Figure 12: soft shadows with area light sampling and antialiasing (16 samples).

Evaluation: the shadow is quite notably diffuse. Increasing the number of samples softens it further, and so does antialiasing.

e) Multi-bounce path tracing

Reflections and refractions are computed recursively, and pass decrementing number of bounces to `shade_blinn_phong` with each recursion.

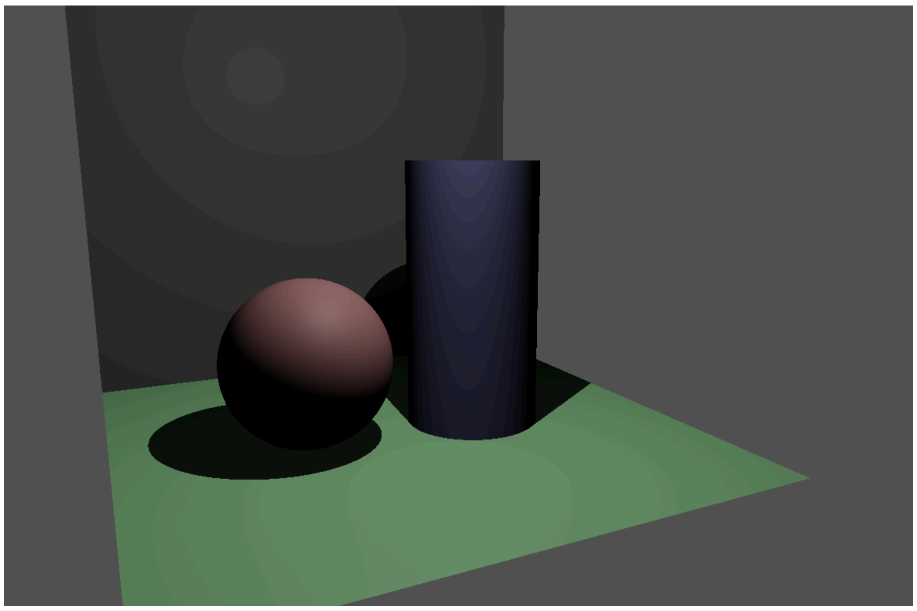


Figure 13: `"nbounces":1`.

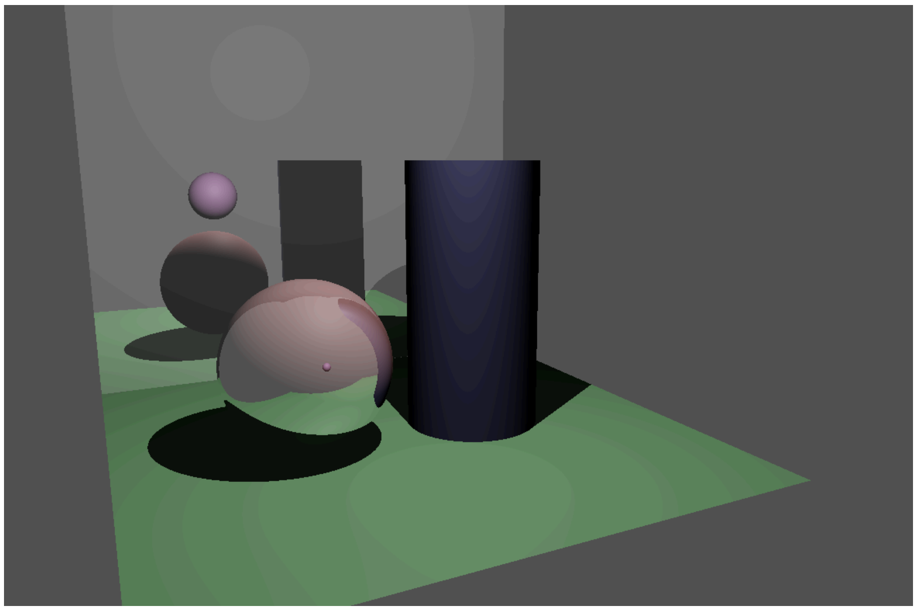


Figure 14: `"nbounces":2`.

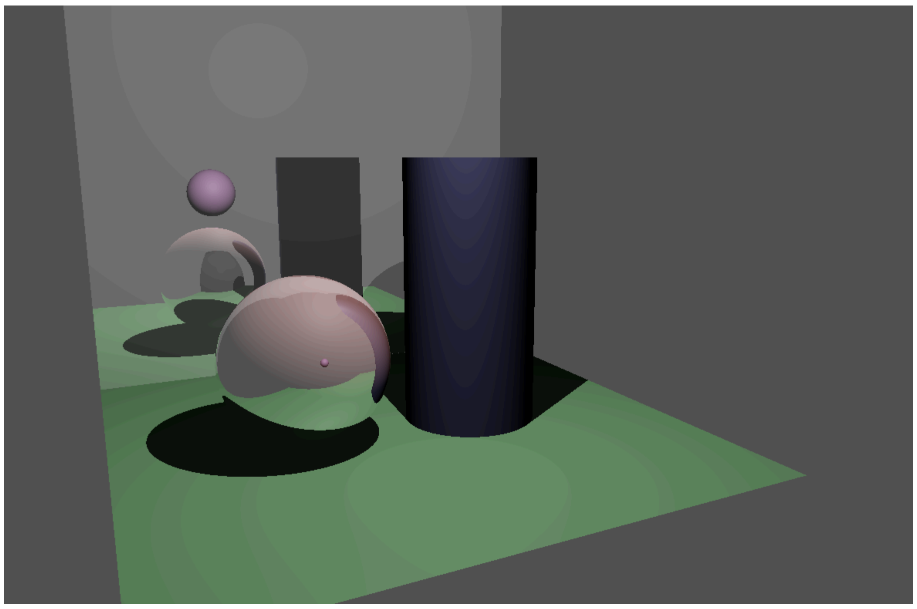


Figure 15: `"nbounces":4`.

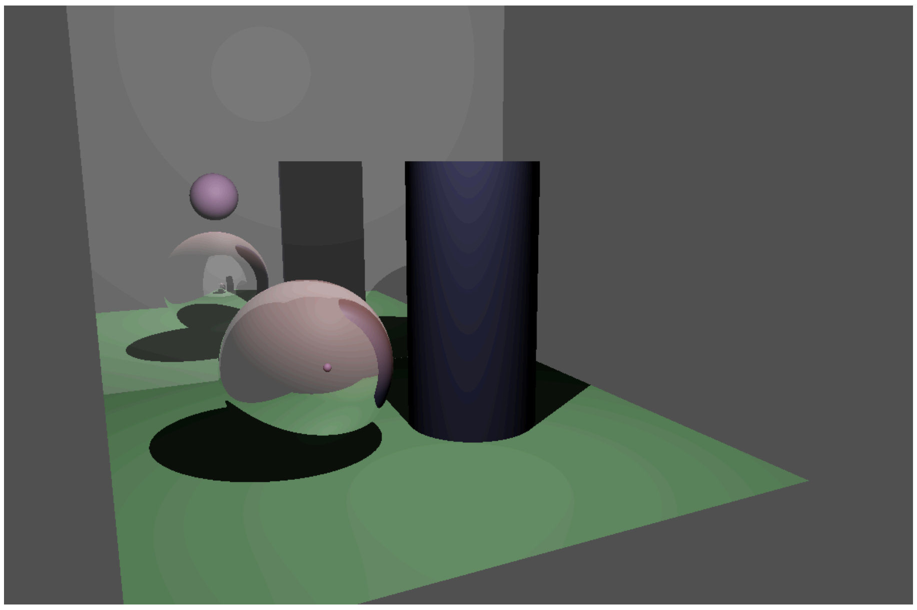


Figure 16: `"nbounces":8`.

Evaluation: we can distinctly see increased depth of reflection as the number of bounces increases.