C++ Raytracer using GLM and SDL

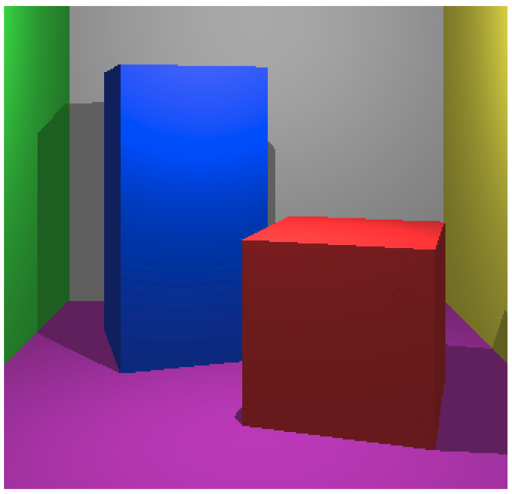
Iwan Pettifer-Cole IP15649   Eleanor Cox EC15618

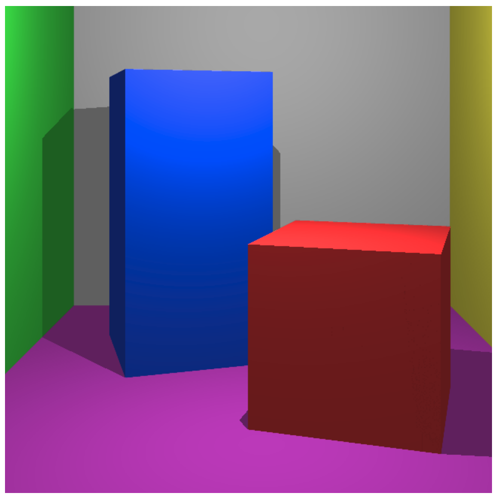
Cramer's Rule

Cramer's rule is used reduce the number of calculations done in finding intersections of rays and surfaces. Instead of inverting the whole matrix containing the ray and surface parameters, we first test if the distance from ray origin to surface is positive. If it is not, the parameters are invalid and we skip to the next point. This speeds up rendering of the initial no extensions cornell box (300x300 pixels) by about 50ms.

Supersampling Anti-Aliasing (SSAA)

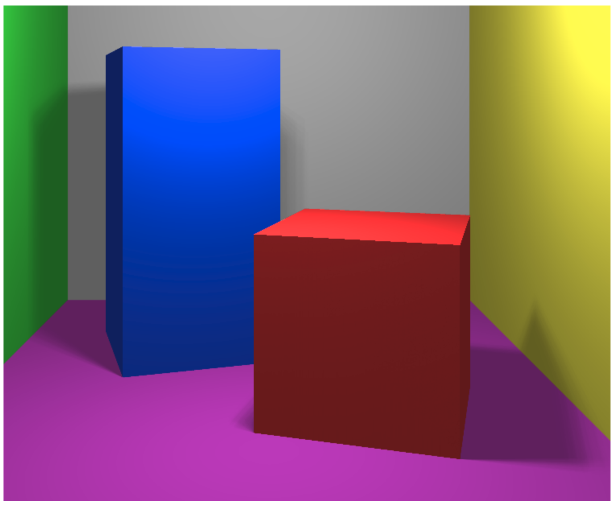
In order to implement smooth edges for solid objects, we used SSAA in a grid pattern. Doing this gives the edges of cubes a much smoother edge against a different colour background. This is implemented by sending out a number of slightly off-center rays for each pixel. We opted for a grid pattern for these additional rays, rather than random / scatter sampling. Our final render uses 8 additional rays per pixel.





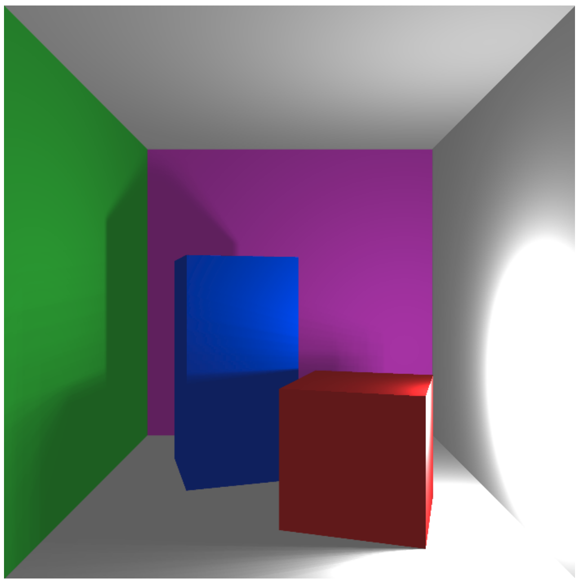
Smooth Shadows

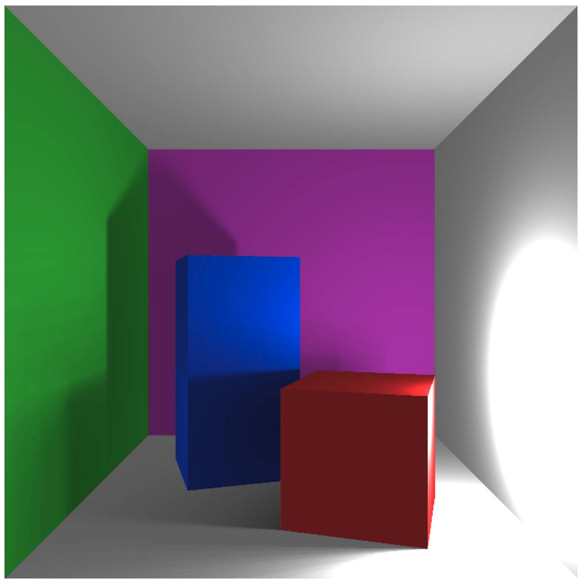
In order to implement smooth shadows, we increased the number of light sources in the scene. By setting each light point's origin to a slightly randomised position around a specified center, we created an orb of light points. The existing DirectLight function was used to calculate if there are objects obstructing the ray from pixel to each light source, and the result is averaged to produce a smooth gradient shadow.



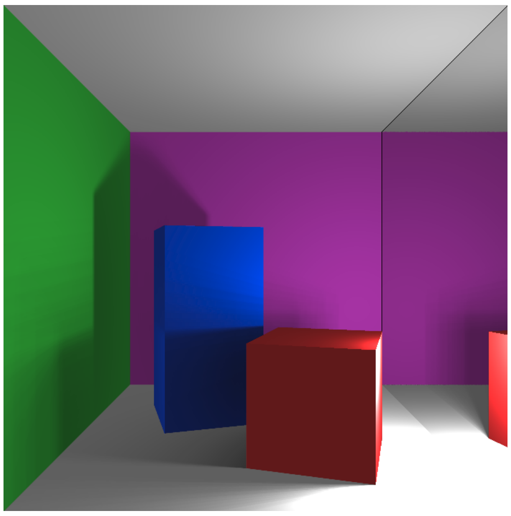
Dark Shadows

Even with smooth shadows, the darkness of the shadow only varied on the edges where some light sources were still reaching the intersection. "Dark shadows" are implemented by calculating the number of objects between the intersection and light source, and the darkness of the shadow is based on this. This means the "secondary" shadow behind two objects will be darker than a "primary" shadow behind one object. This differs from the previous shadows which only check if the light is blocked, rather than how many times the light is blocked. Dark shadows are a very hacky way of giving a slight impression of global illumination.

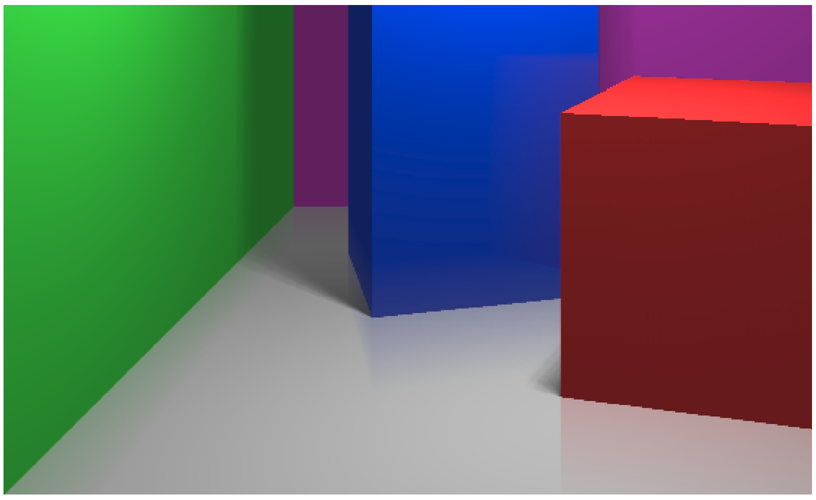




Mirrors
To make reflective surfaces, we added material types to the Triangle class. A ray intersecting with a surface of Mirror type is reflected (using the $R = I - 2(N.I)N$ equation) and the new intersection is returned. If the reflected ray collides with another mirror, it is reflected again for a maximum of 3 times. Reflected pixels are darkened slightly for added realism, as real mirrors do not reflect 100% of the energy of the incident photon.



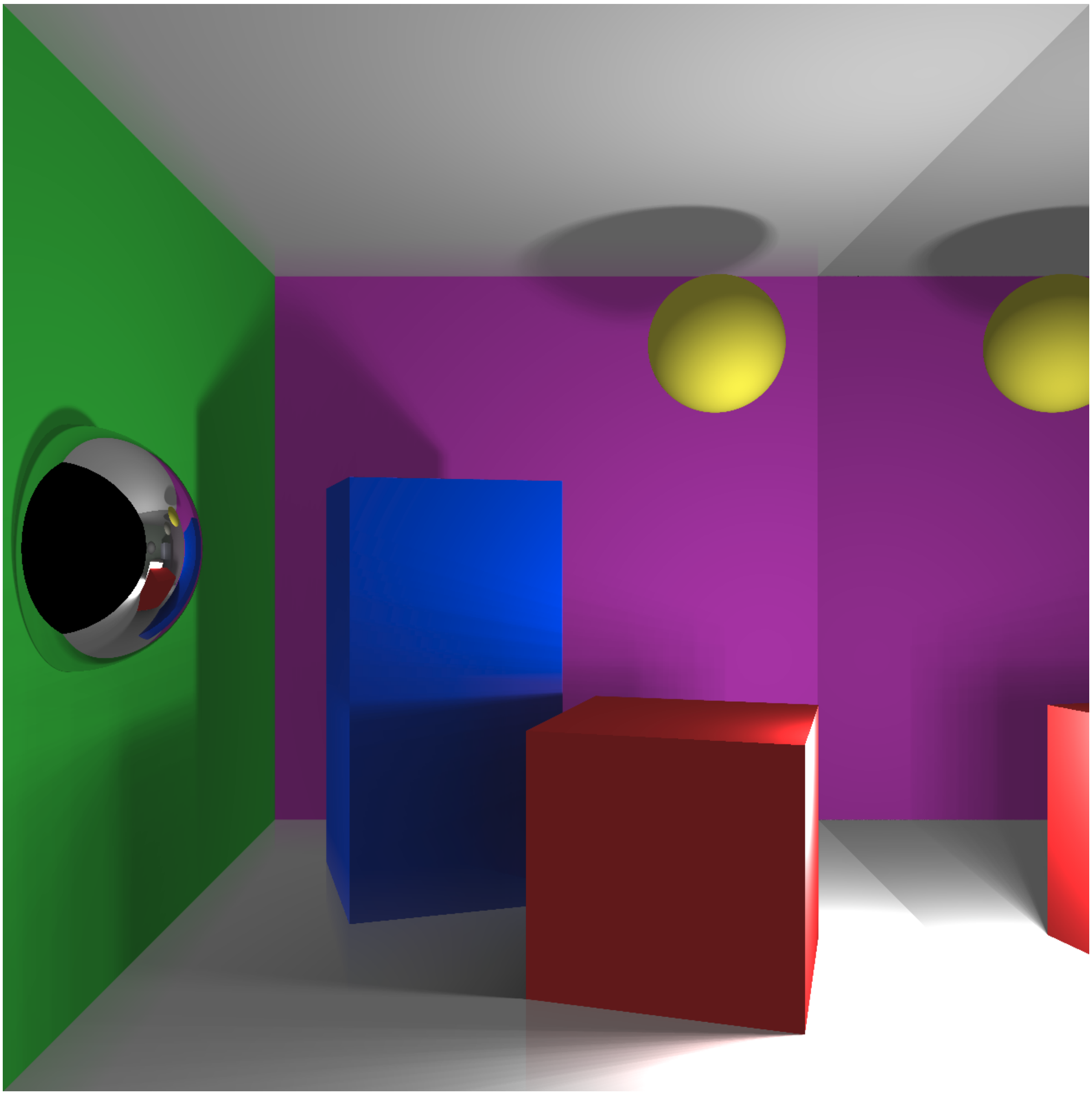
Colour Bleeding (Glossy materials and Global Illumination)
Global Illumination is the model that light bounces off many surfaces, changing colour slightly, before finally hitting the camera. One common result of GI is colour bleeding, that a strong colour on one object can cast that colour onto close neighbouring objects. Colour bleeding was first implemented for all objects. To do this, every ray cast from the camera is reflected once from its intersection, and the new intersection’s colour and distance is stored. If this distance is below a threshold (i.e the reflected intersection is close enough to the original intersection) then the “bleed colour” is added to the pixel, proportionally to the distance of the reflected intersection. This is shown clearly on the edges of white and coloured walls. We then added a new Matte and Glossy material. Glossy materials would accept colour bleed, while Matte would not. By making the cubes Matte and the floor Glossy, we created a nice diffuse reflection of the cubes in the floor. This is shown in the image below. Also in the image, the tall blue box is set to glossy, so a slight diffuse reflection of the red cube can be seen in it. Colour bleeding from the walls can also be seen in the floor.



If we took the number of reflections further, we could fully implement global illumination. However we decided not to do that, due to time constraints, and the rendered image looking good enough.

Spheres

To make the scene more varied, we added spheres. To do this, we abstracted the Triangle class to inherit a new “Object” class. By moving shape-specific functions inside the Sphere and Triangle class, the rest of the raytracer did not have to be refactored. This included the intersect function, and the compute normal function. Mirror materials also work with spheres, as shown.



Runtime Flags for enabling features

Runtime flags were added to toggle different extensions on the fly. These include:

--*soft4*	to enable SSAA with a 4-sample grid
--*soft8*	to enable SSAA with a 8-sample grid
--*smooth*	to enable smooth shadows
--*dark*	to enable dark shadows
--*mirror*	to enable mirrors
--*bleed*	to enable colour bleeding (GI)
--*all-flags*	to enable all of the above, with SSAA set to 8-sample grid pattern.

To generate our final render, please run the program using --all-flags

Small Optimisations

In order to reduce memory usage, we removed as many temporary variables and variable declarations from within FOR loops that are used the most. We also limited the number of conditional statements within loop. Conditionals in loops cause branching in the processor pipeline, and prevent the compiler from vectorising sections of the code.

Another small optimisation was being careful of the order of expressions within an IF statement's condition. Many IFs contained expressions that evaluated the values of objects, or used the return value of a function called within the conditional. For example:

```
if (bleed && global_lum > 0 && global_lum < 3 && Gloss ==
    objects[intersection.objectIndex]->material)
```

If the "bleed" flag is not enabled, then there is no need to use processing power and memory allocations to evaluate whether or not a surface is Glossy, as we do not care.

OpenMP

We added OpenMP to the compiler, and added a pragma in the Draw() function to instruct the compiler to vectorise as much of the loops as possible. This is most effective when expressions and variables in each iteration of the loop are independent of each other. When the data and memory access is independent, the loop can be parallelised massively, providing roughly a 35% speedup in practise.