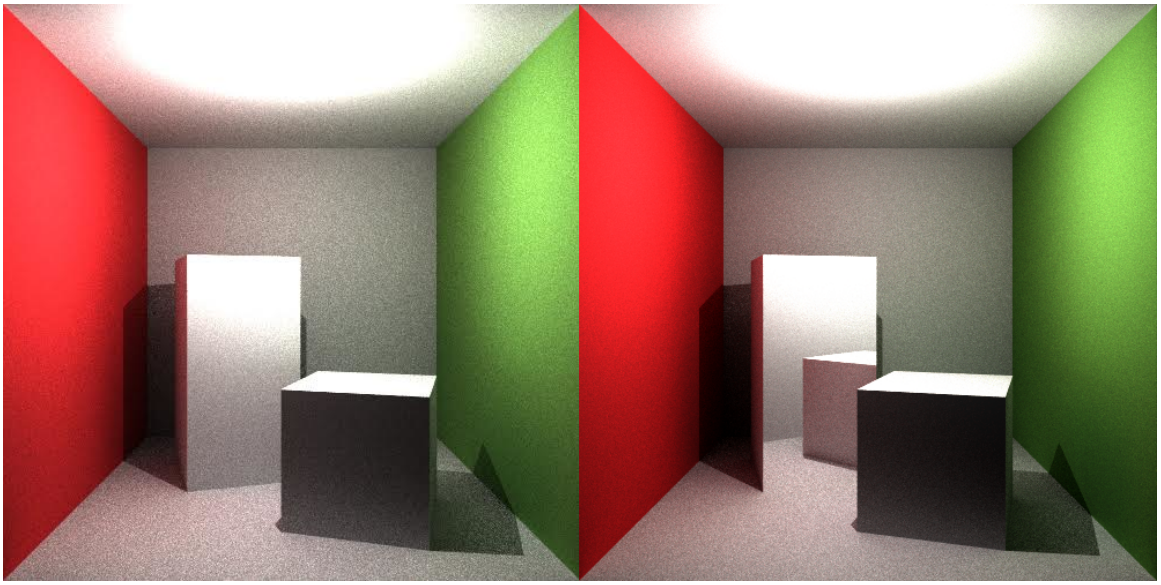
Raytracer



(a) Global illumination by path tracing

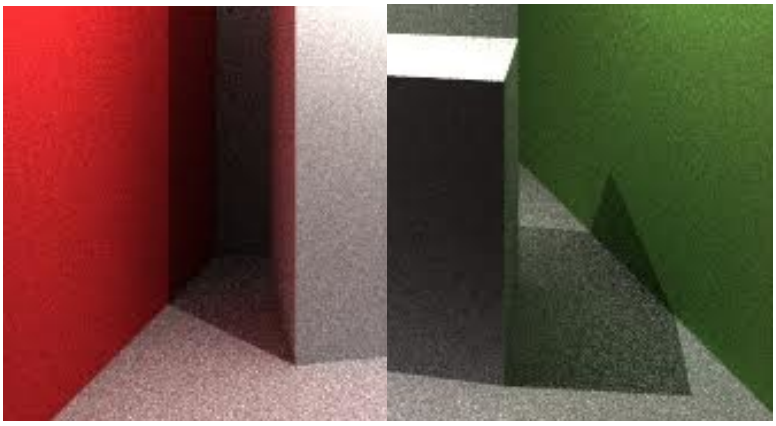
(b) Global illumination by path tracing, mirrored surfaces, Monte Carlo sampling, Markov chain bounces

Figure 1: Raytracer renderings

Path tracing

We implement *global illumination* by path tracing, multiple rays are shot out of the camera through each pixel extending it with **Markov Chain light bouncing**, where, at each potential bounce there is a certain probability of halting the bounces. We also have a maximum number of bounces allowed to limit the runtime of the rendering. We also added **Monte Carlo sampling**, where we shot several rays from the camera through each pixel at random directions to reduce *aliasing* in the image.

Path tracing facilitates *colour bleeding* as can be seen in fig. 2. One of the disadvantages of using path tracing is the large number of samples required to produce noise free renders, one can see the progressive noise reduction of each successive iteration in fig. **??**.



(a) Left cuboid and floor with bleeding from red wall

(b) Right cube and floor with bleeding from green wall

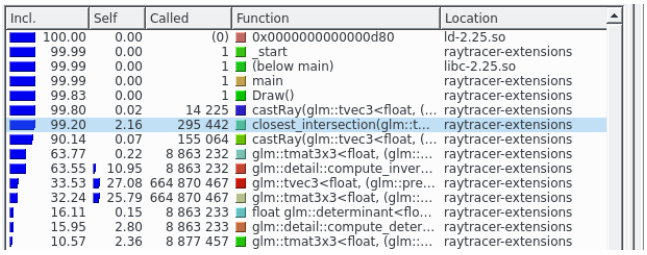
Figure 2: Colour bleeding crops

Mirrors

We implemented mirrors, where it is possible to set zero or more triangles as perfect mirrors. We added a threshold for the maximum number of bounces on mirrors to prevent getting into an infinite loop if there are a lot of mirrors.

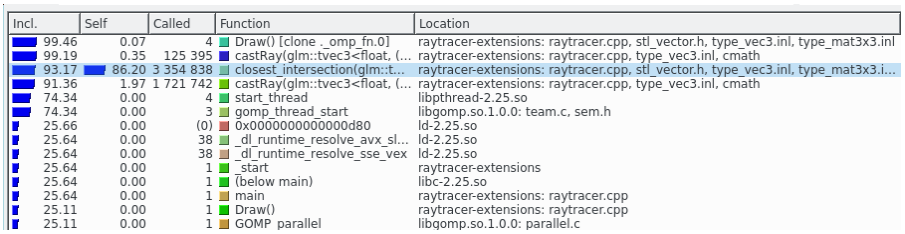
Parallelisation

We profile the raytracer under cachegrind, a sampling profiler part of the valgrind project. The samples collected were visualised using KCacheGrind for graphical analysis, screenshots of the tool pre and post parallelisation and optimisation can be seen in fig. **??**



Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000000d80	ld-2.25.so
99.99	0.00	1	_start	raytracer-extensions
99.99	0.00	1	(below main)	libc-2.25.so
99.99	0.00	1	main	raytracer-extensions
99.83	0.00	1	Draw()	raytracer-extensions
99.80	0.02	14 225	castRay(glm::tvec3<float, (...	raytracer-extensions
99.20	2.16	295 442	closest_intersection(glm::t...	raytracer-extensions
90.14	0.07	155 064	castRay(glm::tvec3<float, (...	raytracer-extensions
63.77	0.22	8 863 232	glm::tmat3x3<float, (glm::...	raytracer-extensions
63.55	10.95	8 863 232	glm::detail::compute_inver...	raytracer-extensions
33.53	27.08	664 870 467	glm::tvec3<float, (glm::pre...	raytracer-extensions
32.24	25.79	664 870 467	glm::tmat3x3<float, (glm::...	raytracer-extensions
16.11	0.15	8 863 233	float glm::determinant<flo...	raytracer-extensions
15.95	2.80	8 863 233	glm::detail::compute_deter...	raytracer-extensions
10.57	2.36	8 877 457	glm::tmat3x3<float, (glm::...	raytracer-extensions

Figure 3: Pre optimisation



Incl.	Self	Called	Function	Location
99.46	0.07	4	Draw() [clone ._omp_fn.0]	raytracer-extensions: raytracer.cpp, stl_vector.h, type_vec3.inl, type_mat3x3.inl
99.19	0.35	125 395	castRay(glm::tvec3<float, (...	raytracer-extensions: raytracer.cpp, type_vec3.inl, cmath
93.17	86.20	3 354 838	closest_intersection(glm::t...	raytracer-extensions: raytracer.cpp, stl_vector.h, type_vec3.inl, type_mat3x3.i...
91.36	1.97	1 721 742	castRay(glm::tvec3<float, (...	raytracer-extensions: raytracer.cpp, type_vec3.inl, cmath
74.34	0.00	4	start_thread	libpthread-2.25.so
74.34	0.00	3	gomp_thread_start	libgomp.so.1.0.0: team.c, sem.h
25.66	0.00	(0)	0x0000000000000d80	ld-2.25.so
25.64	0.00	38	_dl_runtime_resolve_avx_sl...	ld-2.25.so
25.64	0.00	38	_dl_runtime_resolve_sse_vex	ld-2.25.so
25.64	0.00	1	_start	raytracer-extensions
25.64	0.00	1	(below main)	libc-2.25.so
25.64	0.00	1	main	raytracer-extensions: raytracer.cpp
25.11	0.00	1	Draw()	raytracer-extensions: raytracer.cpp
25.11	0.00	1	GOMP_parallel	libgomp.so.1.0.0: parallel.c

Figure 4: Post optimisation

We **parallelise** the path tracing loop over each pixel using **OpenMP**. This gives about a 2–3 factor speedup on a dual core machine with hyper threading.

Rasteriser

Controls

Mouse control stuff

Clipping

We implement clipping and culling to reduce rendering times by removing triangles outside of the view frustrum (culling) and straddling the frustrum (clipping). This is performed by converting the following pipeline process:

- Convert world coordinates into homogenous coordinates
- Applying the camera transform to move the vertices into the camera coordinates frame.
- Squash the view frustrum into a cube by application of the clip space transform encoding perspective into the w coordinate

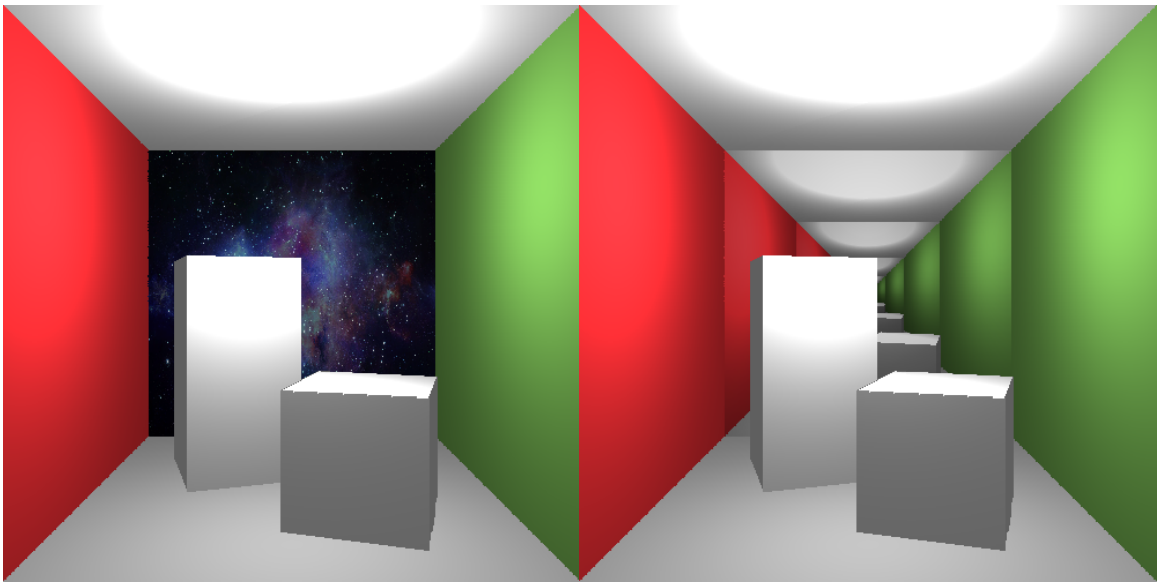
- Clip the edges of the polygons crossing the planes defining the cube in clip space ($w = D$, $w = -D$, etc for each dimension D of x, y, z), we also clip against the $w = 0$ plane to ensure that we don't get infinitely long non-contiguous lines in normalised device coordinate (NDC) space and to avoid division by zero errors.
- The vertices defining the clipped polygon are transformed into NDC space (a cube containing the world with bounds [-1, 1] in all dimensions) using the perspective divide.

The actual clipping algorithm is that proposed by Ken Joy, it utilises the insight that the side of a plane a point lies on can be determined by the definition of the plane and the point. The plane is defined to have normal $\boldsymbol{n}$ with a point P lying on the plane, we wish to determine with side a point Q lies on, to do construct a vector from P to Q and use the definition of the dot product to determine whether the point Q is on the same side the normal $\boldsymbol{n}$ is facing, or on the opposing side. We can determine the angle between the vector from P to Q and the normal $\boldsymbol{n}$ by $\theta = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}||\boldsymbol{b}}$ since $\boldsymbol{a} \cdot \boldsymbol{b} = |\boldsymbol{a}||\boldsymbol{b}| \cos(\theta)$. We now can classify the point Q as 'in' if $\theta > 0$, 'on' if $\theta = 0$ and 'out' if $\theta < 0$, the terminology of the side of the plane the point lies ('in', 'on') is chosen as we define normals of the bounding planes to point into our world, and so a point that is 'in' is inside the bounding plane and should not be clipped.

The plane 'in'/'out' test is utilised in an algorithm used to clip a polygon, the polygon is clipped against each plane in turn. We give an example clipping against the bounding plane $w = x$ which corresponds to bounding plane $x = 1$ in NDC space. For each vertex v we determine whether v is 'in' or 'out', we keep track of the dot product between the vector from the surface to v with the normal of the plane that tells us this (negative implies 'out' and positive implies 'in') for both the current vertex and the previous vertex. We then compute the product of the previous dot product with the current dot product, if the result is negative then either the previous vertex was out and the current in or vica versa and so we compute the intersection of the line with the bounding plane. We add the intersection point to an 'in' list which will form the vertex definition of our clipped polygon. Finally we add the vertex v to the in list if the current dot product is greater than 0, i.e. if it is in.

Textures

We implemented simple texture mapping. It loads a BMP image file and allows any triangles to contain all or part of the image.



(a) Space back plane texture

(b) Infinity Mirror like texture mapping of the back plane

Figure 5: Rasteriser renderings