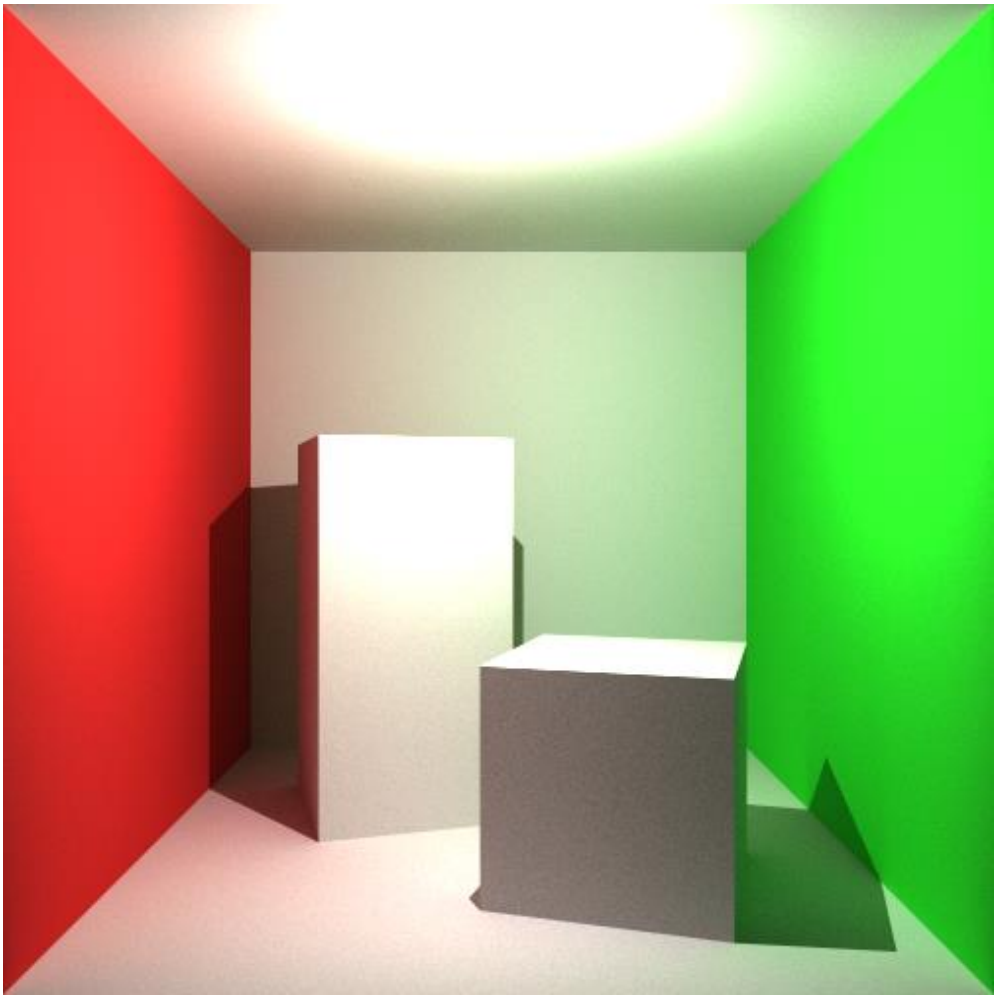
COMS30115 Coursework one: Raytracer



Lingyu Si 1611852

Muyu Yang 1652347

Contribution table

Name:	**Contribution percentage**: (Total is 100%)
Lingyu Si	50%
Muyu Yang	50%

Compiling and running the code.

Rather than doing all things in one script, we have split the key components into different files. The Intersection class contains the algorithm to find the closest intersection. The lighting class contains the implementation of lighting. The camera class oversees controlling the camera translation and rotation. The Triangle has been extracted from TestModel.h and becomes class to benefit further modifications. The makefile has been modified to adapt the change. A -03 flag has been set so that the code is compiled in most optimized way. And -std=c++11 flag has been added to allow the usage of Uniform Distribution for generate the random number.

When compiling the code, make sure you are current in template folder, type make to compile the code and type ./Build/Raytracing to run the code.

Basic implementation

The basic implementation of this assignment is followed the guidance from the pdf file on module webpage. The given skeleton file is used as the foundation of the assignment. It contained the draw and update functions and it allows us to see the output images and save the images. Followed the pdf file, we learnt about how to cast ray and how to find the intersection. For each ray casted by the camera, we calculated if it has intersection with triangles. We stored and sorted all the intersections of this ray. The surface that has the closest intersection should be visible to the camera at the pixel's location. Then the colour of this surface could be the colour of this pixel.

According to the pdf, the camera could be able to move around the scene and rotate. We managed to implement this requirement by add a new class called camera. Both camera.h and camera.cpp have been edited. The camera class takes the initial location of the camera and the focal length as input parameter and calculate the rotation and translation of the camera. However, after we added the extensions, the computational cost is significantly

increased, therefore the transformation of the camera is not necessary. We refactored our code so that the Raytracer will only render once.

To benefit the further extensions, we have extracted the calculations of light into a new script. For basic implementation, the light class only contains DirectLight function. This function calculates the power of the light by considering it as a sphere around its position, the radius of the sphere is determined by the distance between light and pixels on surface. The angle between surface and light is needed to see if the surface is face the light or not. To do this we calculated the dot product between the normal of the surface and the radius of the sphere. The final value of the pixel is the production of light intensity calculate by DirectLight function and the colour of the pixel.

The direct shadow was drawn by casting a additional ray from the surface to the light source in DirectLight function. The distance between light source and surface called d1, and distance between surface and its closest intersection along the same direction called d2, are compared to find the shadow. If there are a closest intersection and d1 is greater than d2, it means the pixel is blocked by other surfaces and this pixel should have no illumination.

The indirect illumination in the basic implementation is a global constant value. We added this constant to the light to achieve this implementation.

Extension

Multithreading

Since ray tracing has great computational cost, we decided to use multithread to reduce the rendering time. We have included pthread and create 8 worker threads to calculate and draw the pixel on screen. The screen was divided into 8 pieces. Each piece is processed by a worker thread. Since PutPixelSDL function could not been called in worker threads, we have implemented a buffer to store the colour value of each pixel. Then we called PutPixelSDL in main threads to draw the pixels on the screen using the buffer. When drawing the buffer onto screen, we created a DrawBuffer function to put the pixel on the screen. We lock before the call PutPixelSDL and unlock afterwards. This extension increases the efficiency of the code significantly, it is not several times faster than before the extension (depending on how many cores the computer has).

Super sampling

In real world, the edge of object is continuous from the start to the end. However, when we project the object onto the screen, the edges are presented by discrete pixels. The pixels are considered as the minimum element of the image. If a line is partly overlay the pixel, the pixel will only be considered as a part of the line when it passes the centre of the pixel. In this case, aliasing occurs.

To overcome this problem, we used super sampling techniques to calculate the average value of a pixel. The pixel is conceptually divide into grids. The size of the grid determines the quality of super sampling. The larger the grid is, for example, 32*32 compare to 8*8, the better the anti-aliasing will be. For each element on the grid, we calculated the colour by calling function from lighting class. Since the computational cost is linear increased with the increase of grid size, therefore the multithreading extension is critical to this assignment.

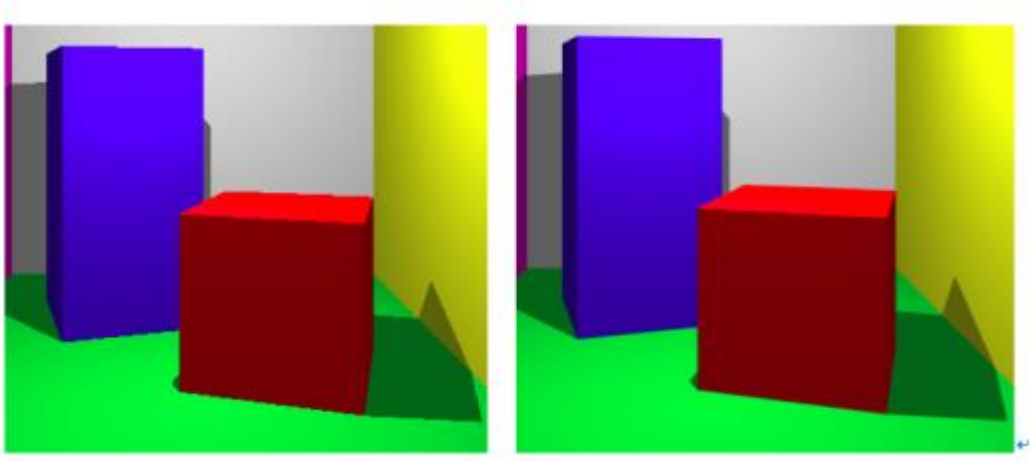


FIGURE 1. BEFORE AND AFTER ANTI-ALIASING

Global illumination

In the real world, the indirect light is not a constant as the pdf gives, the light will bounce between objects and the reflected light from different objects will affect the overall lighting of these objects and therefore cause shadows, reflection and refractions.

Since we already knew how to cast ray, it is not hard to implement the global illumination. The method we used is recursive path tracing. Basically, for every pixel on the surfaces, we consider it as the new starting point. We casted number of rays at this new points to different directions. The rays we casted bounced between pixels. After the bouncing

finished, we calculate the light intensity for each pixel by adding the direct light and bounced light (which is the indirect light) together.

In lighting class, we have already implemented a method called CombineColour, which takes the effect of lighting and then calculates the over colour. This function allows us to implement the recursive ray tracing with minor modifications. For every pixel, we calculated the indirect light by calculate the average of the incoming lights from different direction.

The directions of the rays the casted on the new start point are crucial to the quality of global illumination. The casted rays should have similar directions as its normal, therefore, we defined a hemisphere centred at the start point with the radius of 1. We should get the point on the hemisphere that oriented about the normal of the start point. To do this, we implemented the uniform random number generator and obtained two numbers ranged from 0 to 2PI using this method. The obtained number is used to calculate point on the hemisphere. We initialised a vector with value (0,1,0), rotated it on Z-axis first and then rotated on X-axis using the random numbers. We checked the angle between the normal of the start point and the vector we obtained. Only the vector that within the hemisphere could be used.

The images below should the effect of global illumination. On the left image, the colour from the red wall bleeds on the ceiling. On the right image, the green wall affect the colour of the right side of the block.

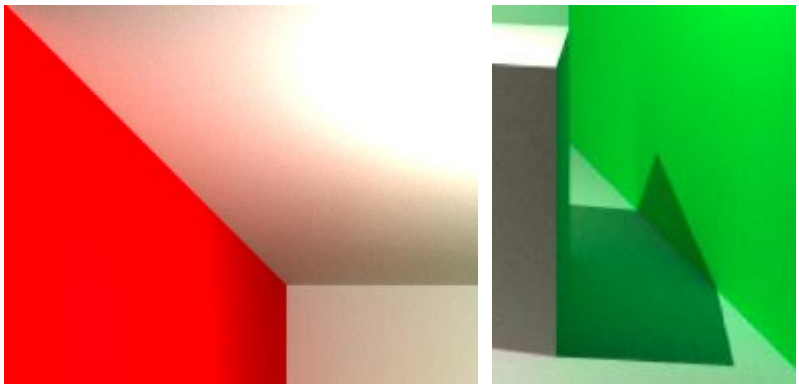


FIGURE 2. EXAMPLE OF COLOUR BLEEDING