Advanced Computer Graphics – Practical Session

Programming Assignment 1

1. Change the scene description to triangles (and triangular patches).

First, we changed all the rectangles to triangles in the scene description (as instances of the `Triangle`-struct). Regarding the patching, we decided to do the following:

We implemented an algorithm, that given a `Triangle`-object, instead of patches, creates 4 new `Triangle`-objects out of the big one in the following way (and removes the big one):

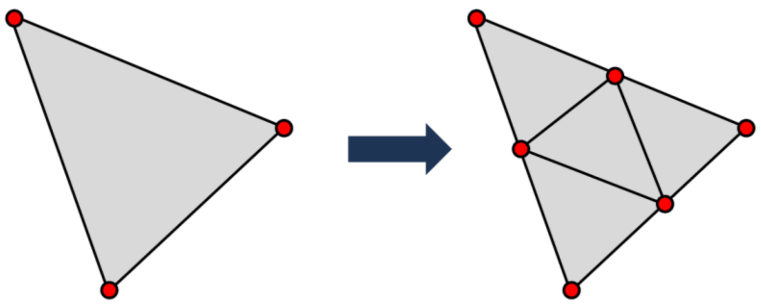

Figure 1: Our triangle-splitting approach.

One can then recursively divide the triangles further. This leads us to the concept of *depth*. A Triangle-object from the scene description has depth 0, since there is no sub-patching yet. Applying our triangle splitting approach once leads to depth 1, applying it twice leads to depth 2, etc.:

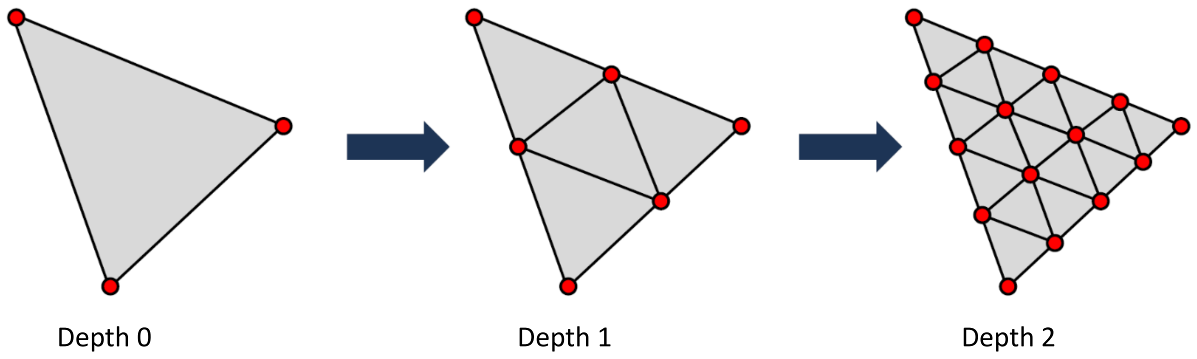

Figure 2: Our concept of depth.

Overall, this means that in our code, the patches are not real patches, but themselves instances of the `Triangle`-struct.

However, our approach has a drawback: it is quite slow. Rendering with depth 0 is no problem, but already depths 1 and 2 become slow and for bigger depths, the rendering becomes really slow. This is because in our approach, we actually *increase* the number of triangles. Starting with n initial triangles in the scene description, with depth d, we get $n \cdot 4^d$ resulting triangles. This explains the decrease in performance.

2. Implement a ray-triangle intersection test.

We chose to implement our ray-triangle intersection test not as described in the practical sessions, but rather we chose the approach described in the following document:

https://courses.cs.washington.edu/courses/csep557/10au/lectures/triangle_intersection.pdf

This approach first tests if the ray and the triangle plane intersect at a point p, and then checks whether this point p actually lies within the triangle. More details on the approach can be found in the document.

3. Change the Monte-Carlo integration to triangles, thus requiring a uniform sampling method in the triangle area.

We implemented this functionality similar to how it was explained in the practical sessions. However, our code still contains a small mistake, as the colors do not seem to be 100% correct in the final rendering.

4. Compare the resulting renderings with those obtained with the original code.

Out of the box, the original code created two rendered images: a patchy image and an interpolated image.

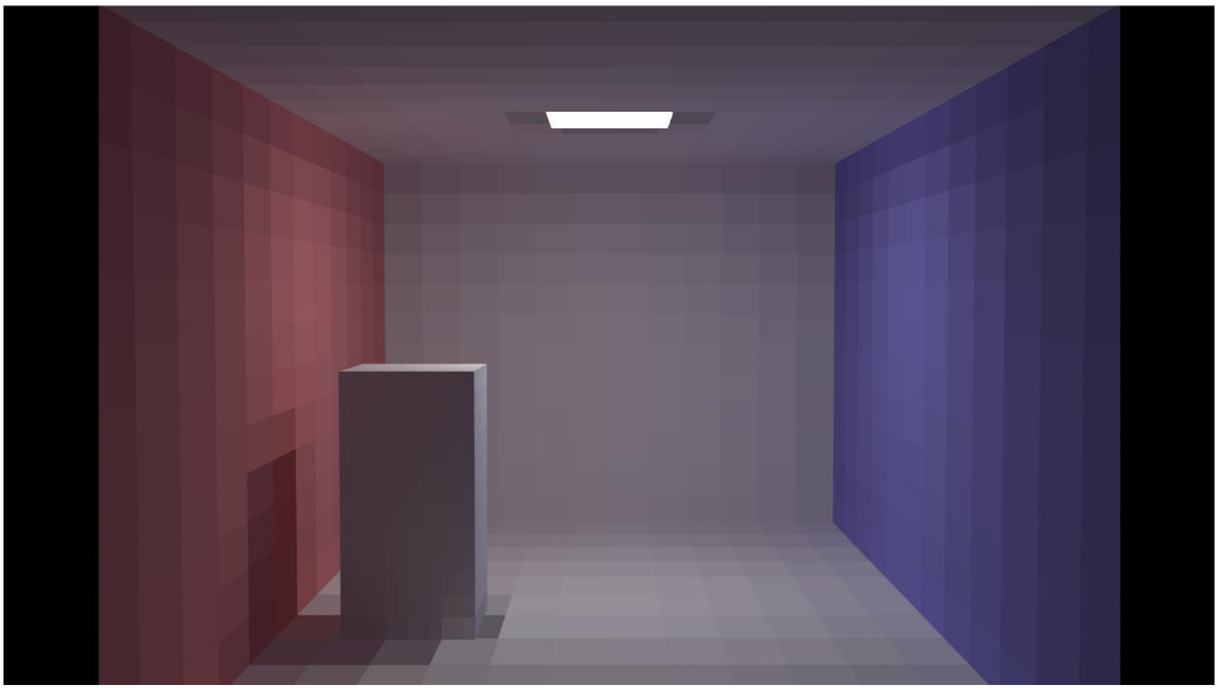

Figure 3: The original, patchy image.

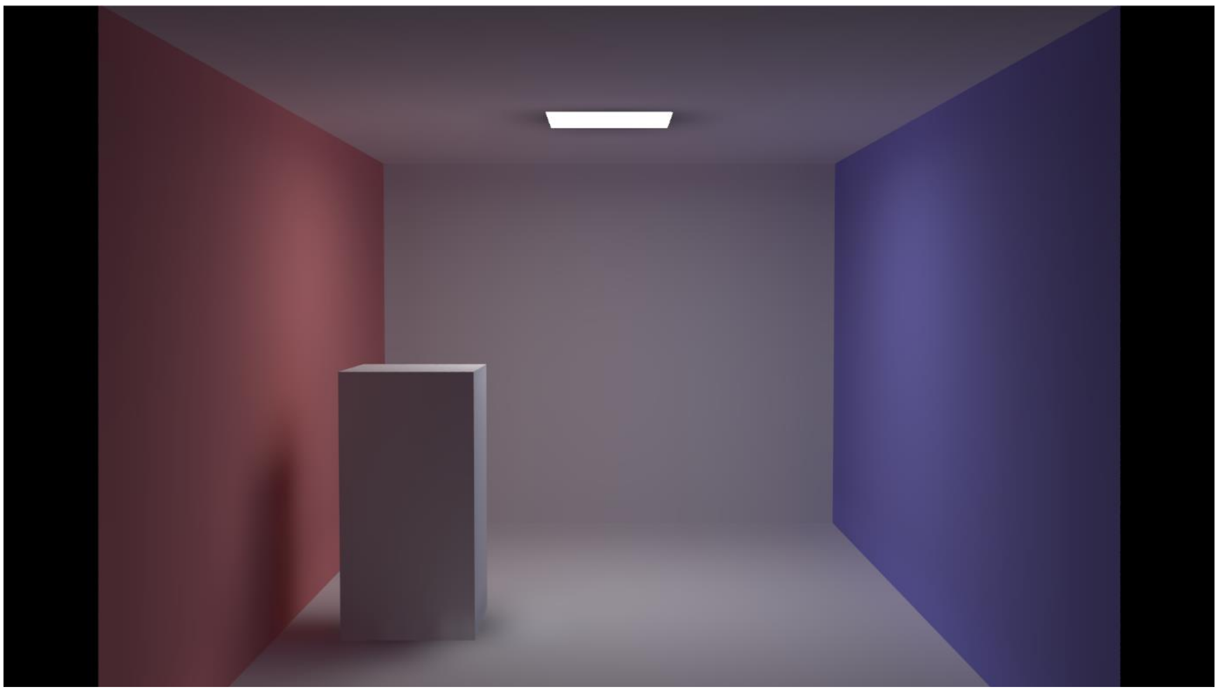

Figure 4: The original, interpolated image.

We focus on the patchy version, since only the patchy version was updated in this assignment.

One can see that each rectangle in the scene is subdivided into 12x12 smaller, rectangular patches. At each patch, the radiosity is constant.

Now, we look at our results. We only look at the patchy images for the reason described above.

For depth 0, we get:

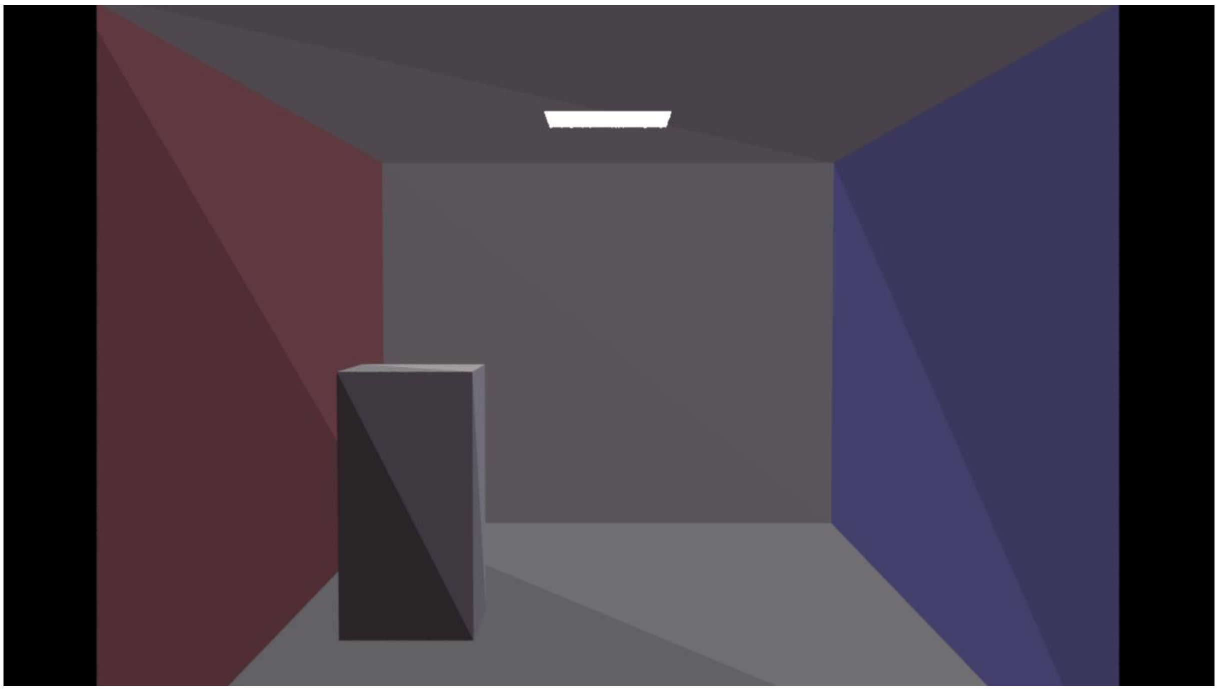

Figure 5: Our final result for depth 0 – each triangle consists of 1 triangular patch.

For depth 1, we get:

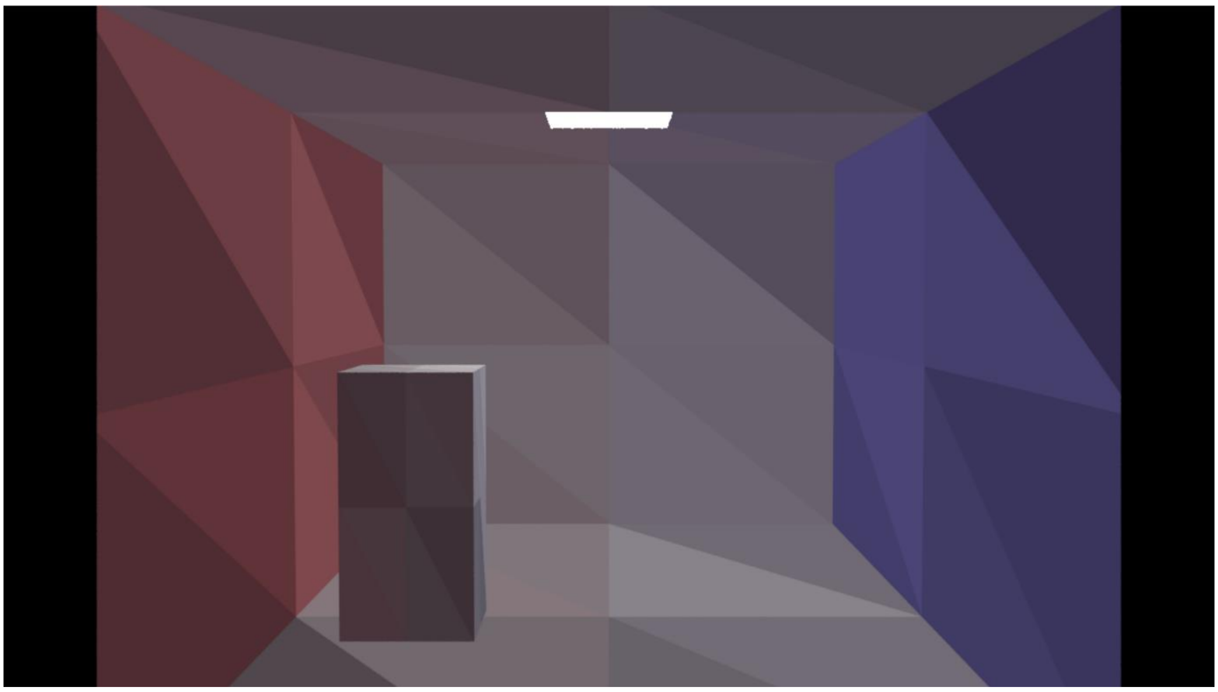

Figure 6: Our final result for depth 1 – each triangle consists of 4 triangular patches.

For depth 2, we get:

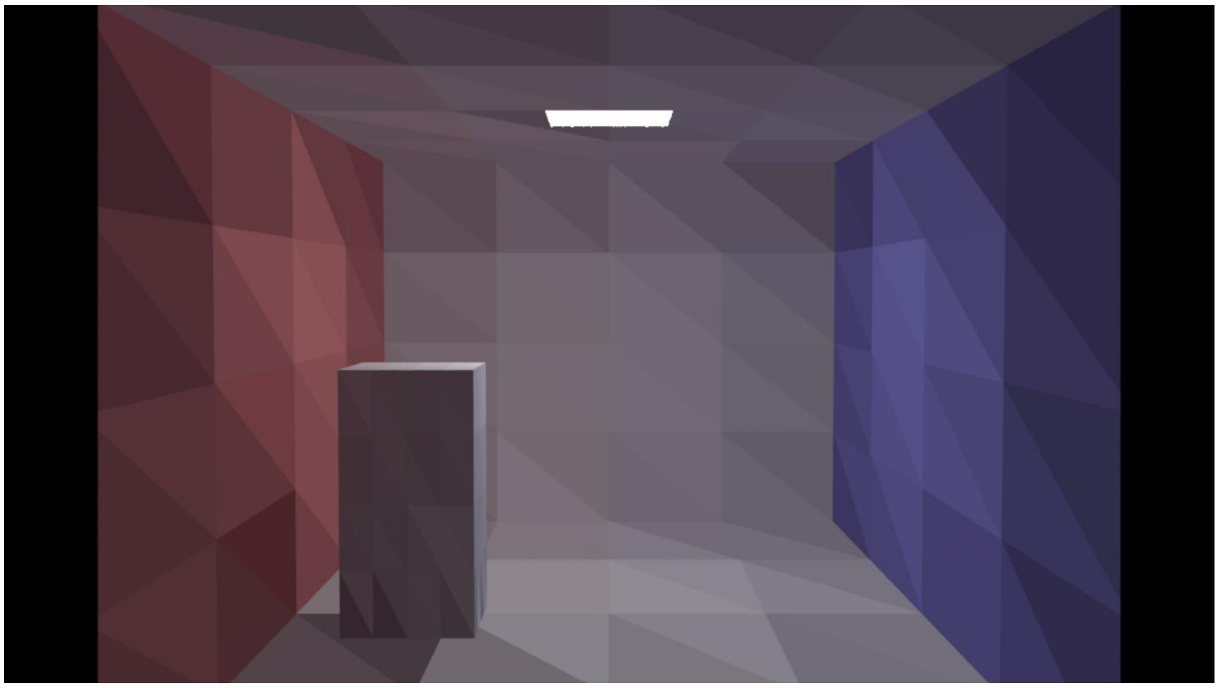

Figure 7: Our final result for depth 2 – each triangle consists of 16 triangular patches.

We did not compute images for bigger depths in the interest of (computation) time.

However, one can easily see that the scene now contains triangles (instead of rectangles) that are themselves subdivided into smaller triangles, the triangular patches (instead of rectangular patches). Each patch has a constant radiosity color.

Our small mistake in the Monte Carlo implementation is visible in the final renderings: the colors of the patches do not seem as coherent as in the original version.