Report for Project #3

DEADLINE: 12/15/2016, 14:00
INSTRUCTOR: Yung-Yu Chuang

王冠鈞 b03902027

This homework requires me to implement the algorithm described in the paper given. I'll describe my implementation and the relations or discrepancies with the paper, as well as the performance analysis and results.

My Median Cut Algorithm Implementation

In my implementation, I used a recursive function, whose behaviour is like binary tree traversal, to do median cutting. Given the maximum depth (i.e. the logarithm of the number of samples given by the original scene file), if the current depth doesn't reach the deepest, then it will decide the cut should be chosen by height or by width, depending on which is larger, and then recursively call the same function by passing the new depths, new ranges of heights and depths and (precomputed) sums. When it reaches the deepest, it'll compute the centroid of the final area A with this formula (where u is the "width" axis, and v is the "height" axis, and $(0, 0)$ is on the left-top of the map):

$$u_c = \frac{\sum_{(u,v)\in A} y^2(u,v)\ u}{\sum_{(u,v)\in A} y^2(u,v)}$$

$$v_c = \frac{\sum_{(u,v)\in A} y^2(u,v)\ v}{\sum_{(u,v)\in A} y^2(u,v)}$$

$$y(u,v) = 0.212671R(u,v) + 0.715160G(u,v) + 0.072169B(u,v)$$

Note that the weight of this centroid formula, y, is squared, which will make the final (u_c, v_c) closer to the real lighted region, since the illuminant of light is proportional to the square of the reciprocal of the distance. And finally, the spectrum of the sample point light is simply the sum within the area and the coordinate is converted to the light space, which is on unit sphere:

$$(\theta, \phi) = (\frac{v_c\pi}{h}, \frac{2u_c\pi}{w})$$

$$(x, y, z) = (\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta)$$

Particularly, when an area is already `1x1` in pixels, my implementation will check the current depth, say it k. If the required number of samples is 2^n (i.e required depth is n), then it will add 2^{n-k} identical lights with intensity $\frac{I_{(u_c,v_c)}}{2^{n-k}}$ to the light table. This will make the light sampling remain uniform (i.e. pdf is still a constant), while making the probability of sampling the duplicated light higher.

Above is roughly my implementation, which is basically the same as the descriptions from the paper.

About `MedianCutEnvironmentLight` Class Implementation

This class is in fact a variant of the original `InfiniteAreaLight` class, so as the instructions from the professor and the previous slides, I just modified three of the functions: the constructor, `isDeltaLight()`, and `Sample_L(const Point &p ...)`. As for the others, I've kept them alone anyway, since they are not the core functions in this homework.[1] I'll explain my implementation below.

1. `isDeltaLight()`: Return `true` to avoid BRDF importance sampling.
2. constructor `MedianCutEnvironmentLight(...)`: I added my own code segments in order to do median-cut algorithm. After reading the environment map file, I created a *sumTable* to store the total illuminant value from $(0, 0)$ to (u, v). After such preprocessing, it will call the recursive function `FillPointLightTable()` to do median cutting. The function will compute the final places of each sample lights and store its coordinates in terms of (x, y, z) (which will be on the unit sphere) and their intensities into a table for later sampling.
3. `Sample_L(const Point &p ...)`: Simply a modification on the original version. It first uses `ls.uComponent` to get a sample light from the table, and then fill it into `*wi`. Second it computes the pdf, which can be roughly counted as $\frac{1}{\text{\# of light samples}}$, since all samples has roughly the same intensity. Finally, it will the return radiance value, which is the same as the original version.

Tests, Results and Images

In this project, the running environment I used is the CSIE Workstation server. I used linux3 for execution. linux3 has a CPU of 2.53 GHz clock rate along with 16 cores, and 74 GB of memory. For the execution test, I used 12 of the cores (`-ncores 12`).

The following table is the time for rendering each image, in terms of seconds:

class	`InfiniteAreaLight`			`MedianCutEnvironmentLight`		
time type	pbrt	real	user	pbrt	real	user
`envlight-4`	2.3	10.557	23.950	0.8	8.775	12.410
`envlight-16`	7.5	14.909	1:02.960	1.5	8.633	17.860
`envlight-64`	30.3	37.463	3:33.023	5.0	12.288	40.673
`envlight-256`	112.8	2:00.191	13:38.203	17.7	25.716	2:13.230
`envlight-new-4`	2.6	14.932	30.173	1.1	12.730	17.910
`envlight-new-16`	8.8	20.730	1:13.723	2.5	13.455	27.980
`envlight-new-64`	34.9	46.791	4:09.017	8.3	21.072	1:07.770
`envlight-new-256`	129.8	2:20.968	15:24.663	30.3	45.529	3:44.217

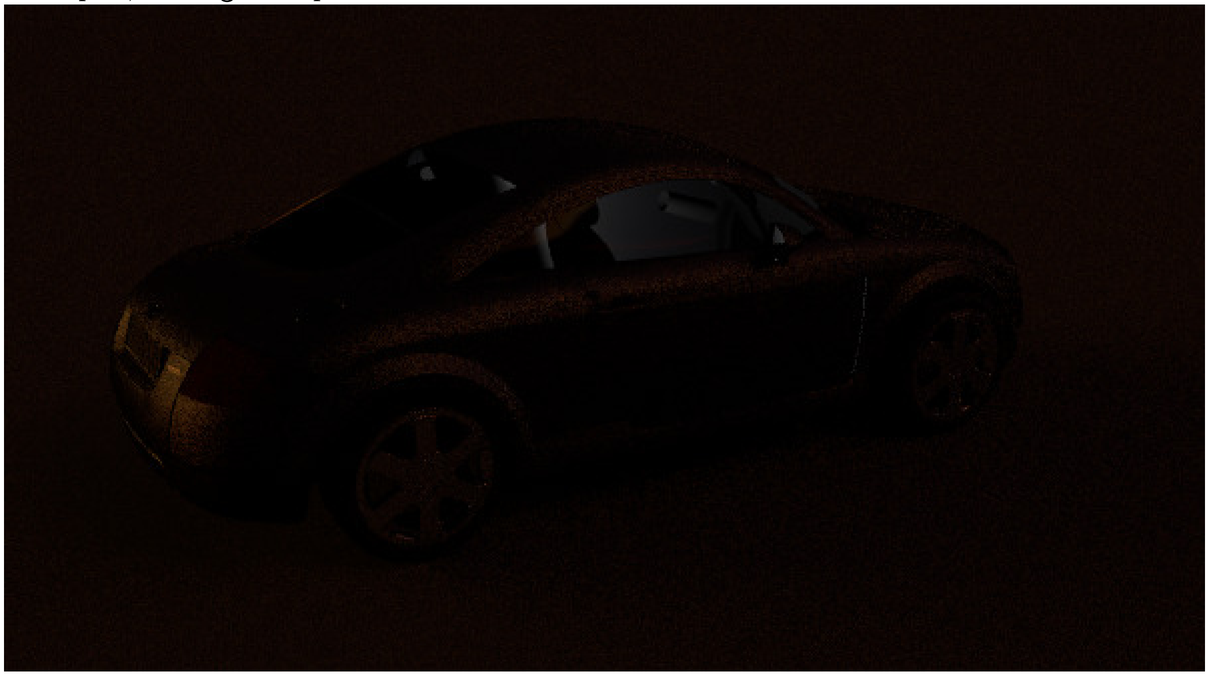
From the results, it's obvious that the alternative method has a better performance than the original implementation. For example, the new class' "`pbrt` time", which roughly measures the rendering time (without preprocessing), is about 3 to 6 times lower than the original class. As for the real time, since the preprocessing time takes longer in the new class than in the original class (because it requires to compute an additional sum table and median cut algorithm), the real time value doesn't has a big difference when the number of samples is low, but when the sample number becomes larger, the differences seem to become significant. This is due to the difference of rendering time growing faster than the difference of preprocessing time.

The following are the rendered images.

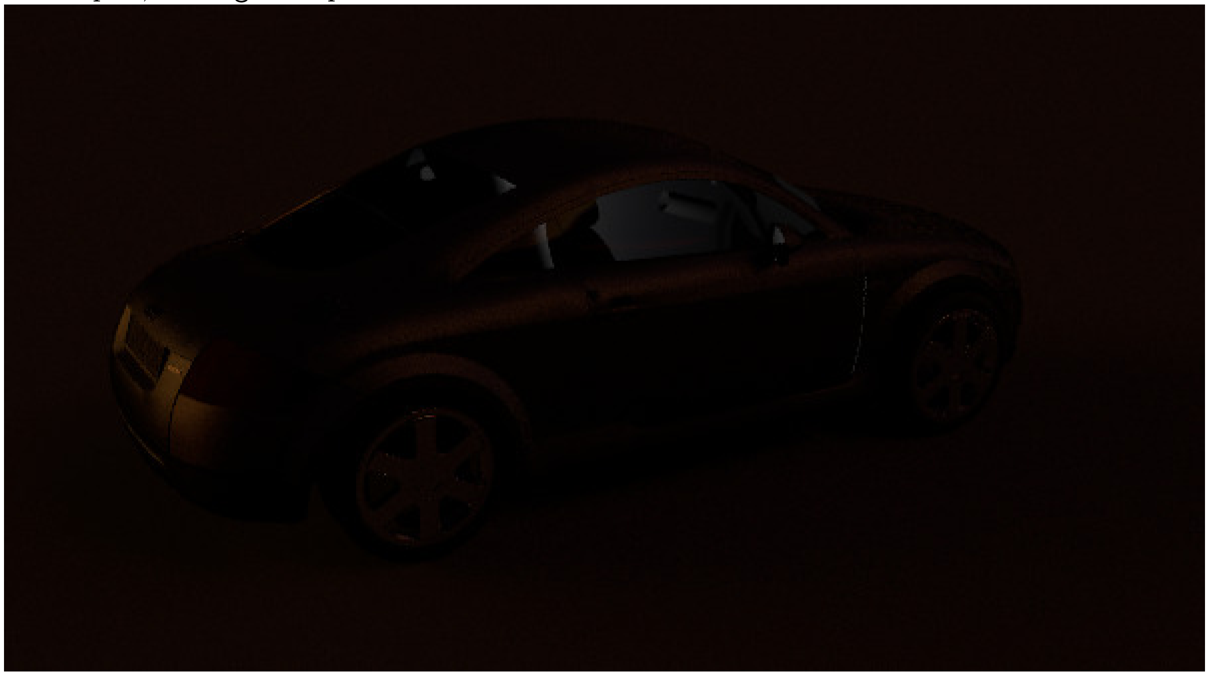
1. Using default settings:

[1] Also referenced from `http://www.csie.ntu.edu.tw/~cyy/courses/rendering/13fall/lectures/handouts/Homework3.pdf`, page 4

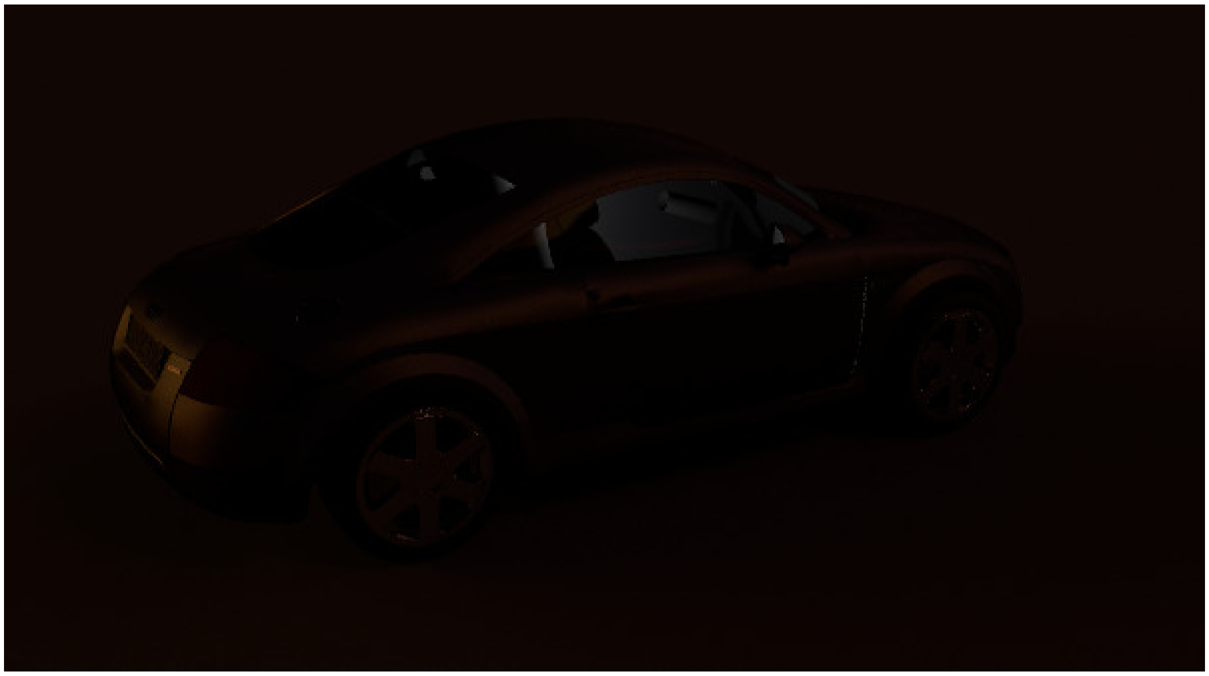
1-a. 4 samples, `envlight` map:



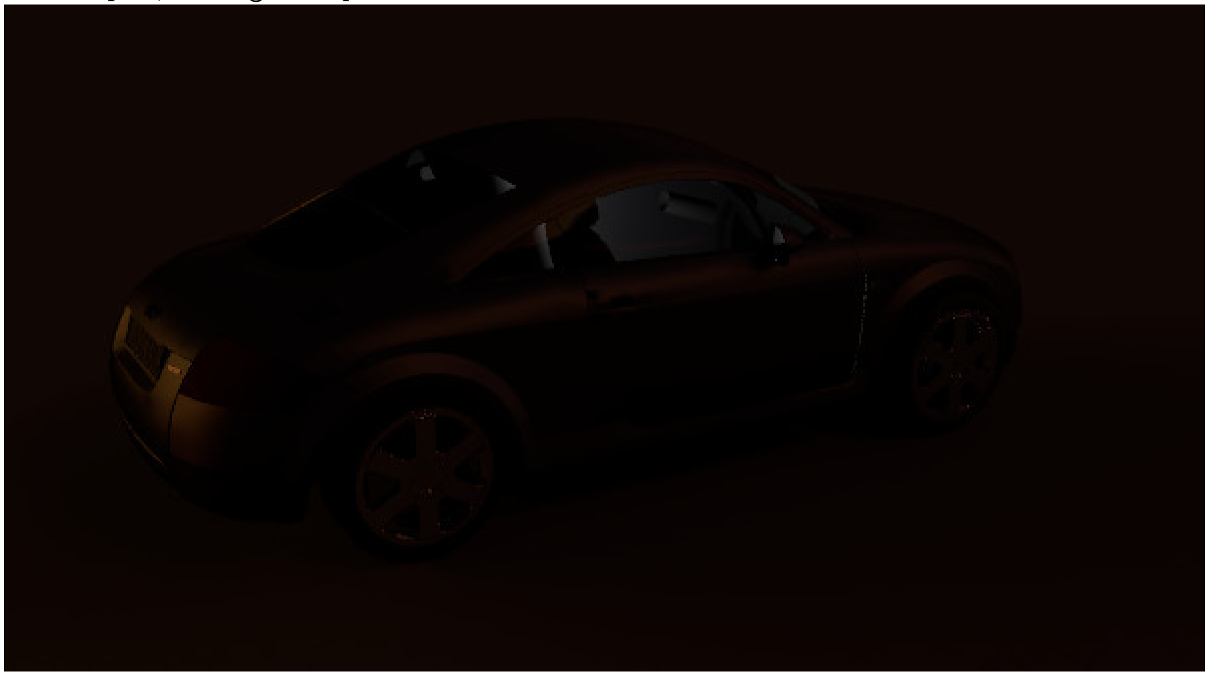
1-b. 16 samples, `envlight` map:



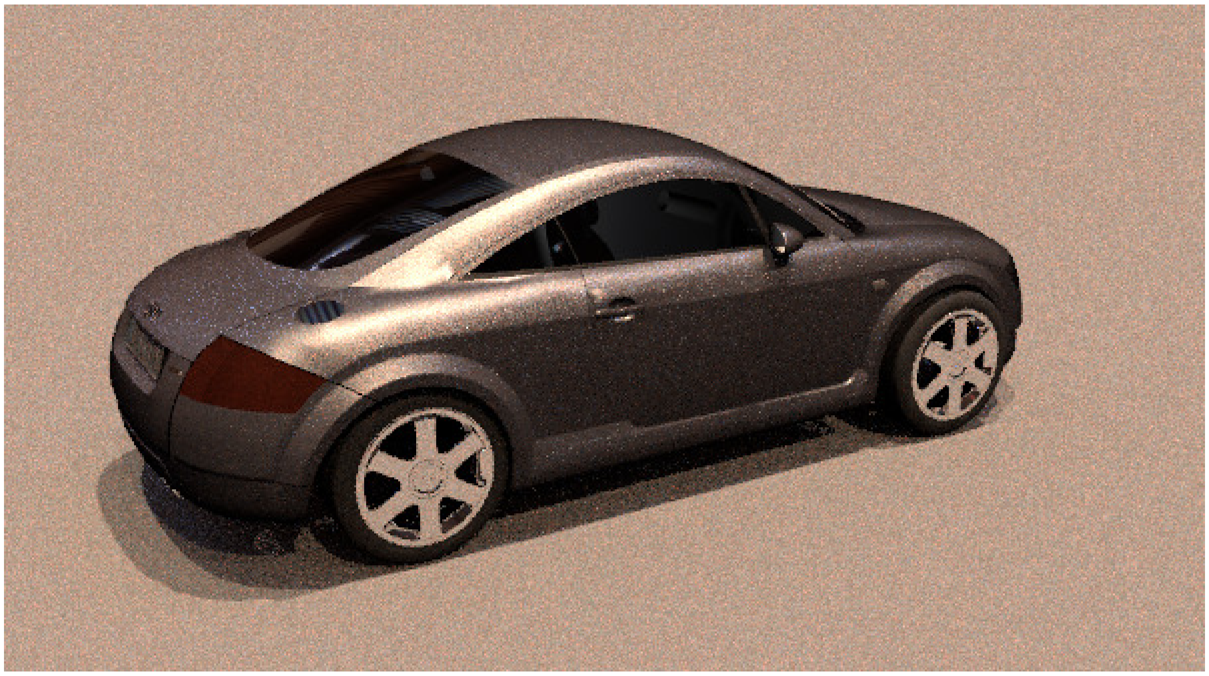
1-c. 64 samples, `envlight` map:



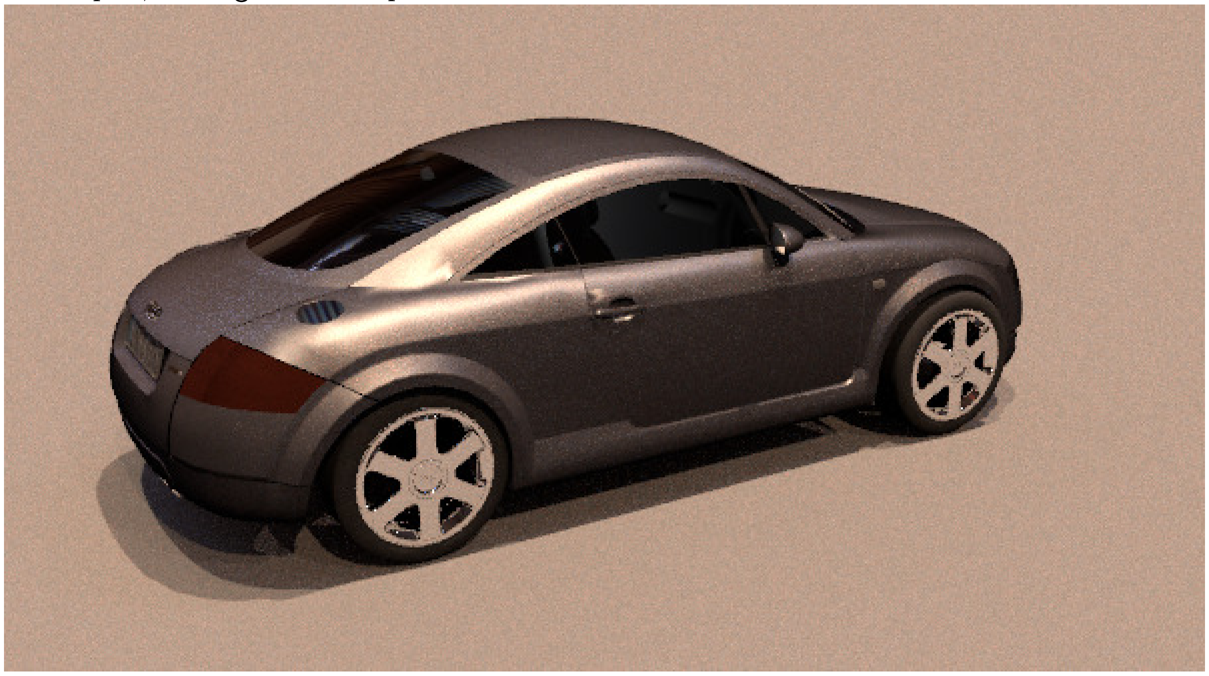
1-d. 256 samples, `envlight` map:



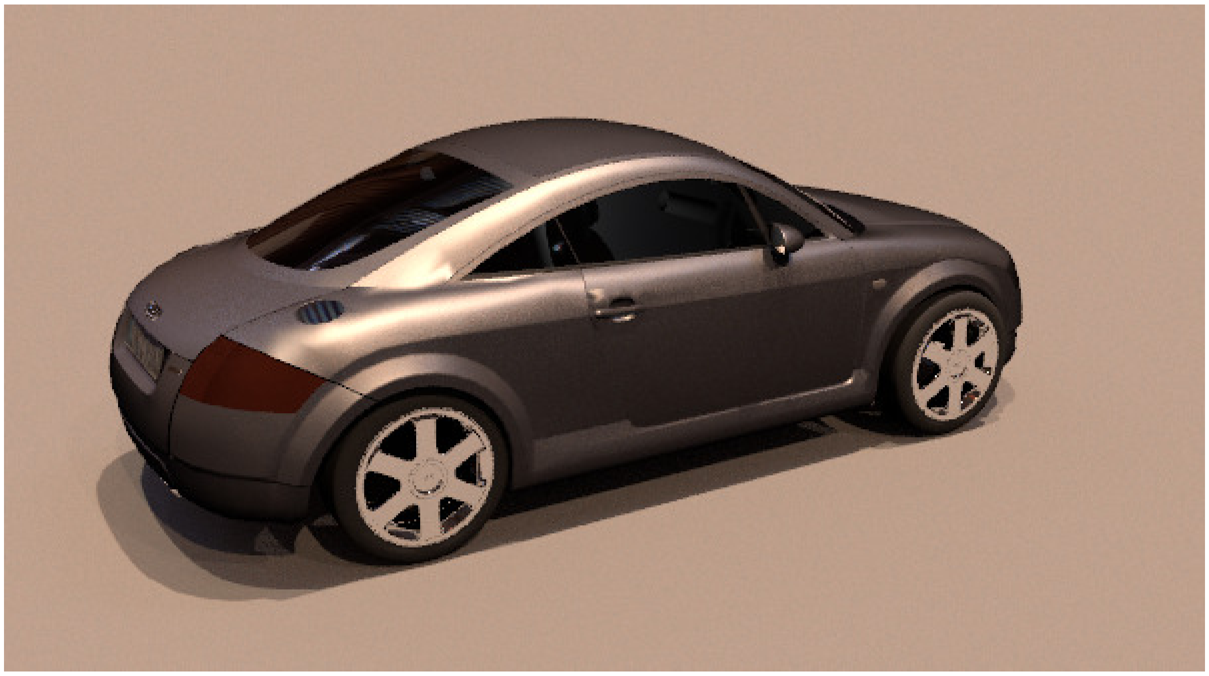
1-e. 4 samples, `envlight-new` map:



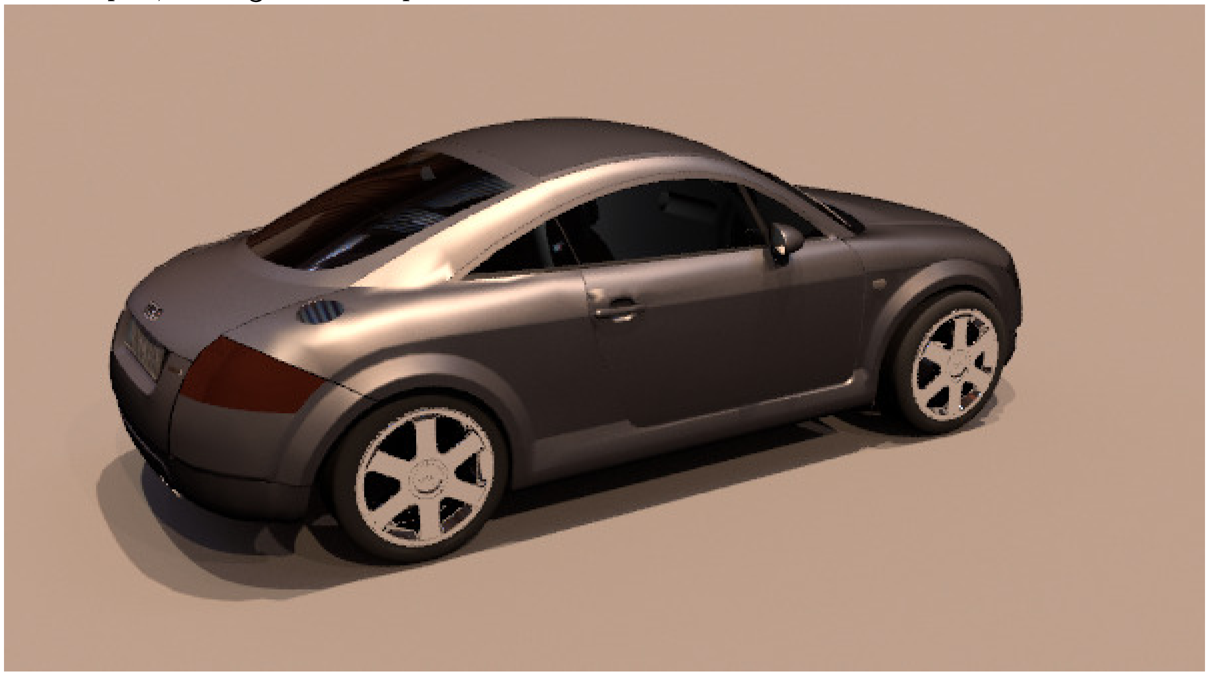
1-f. 16 samples, `envlight-new` map:



1-g. 64 samples, `envlight-new` map:

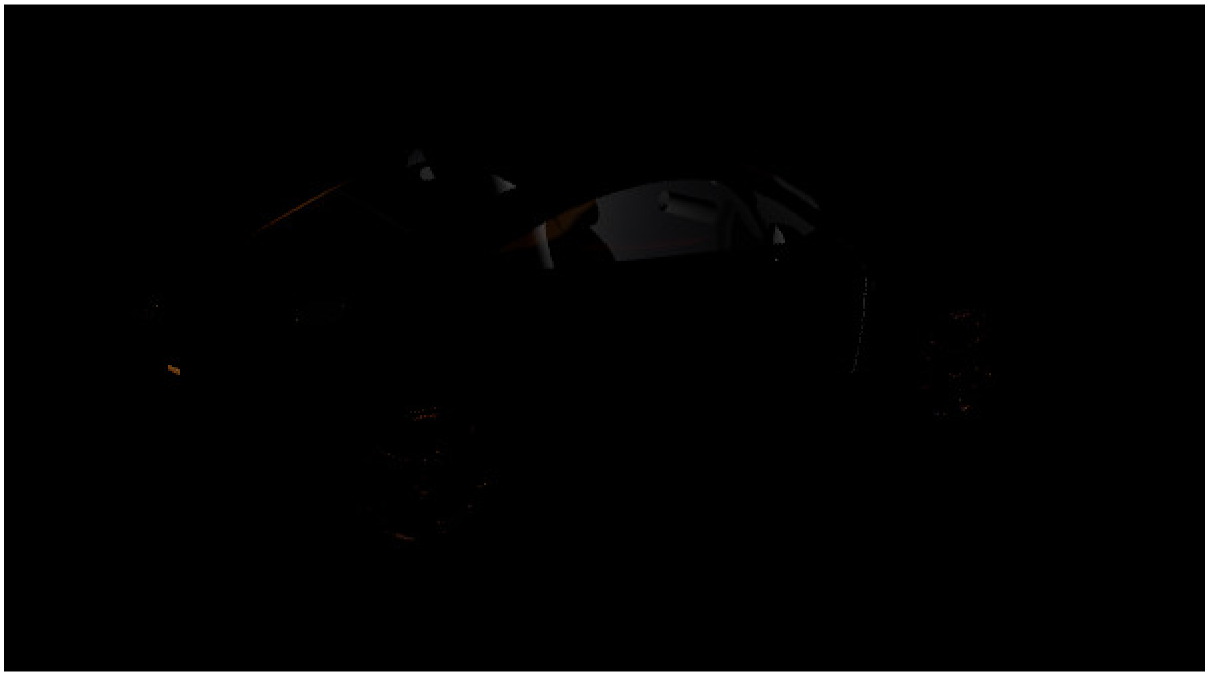


1-h. 256 samples, `envlight-new` map:

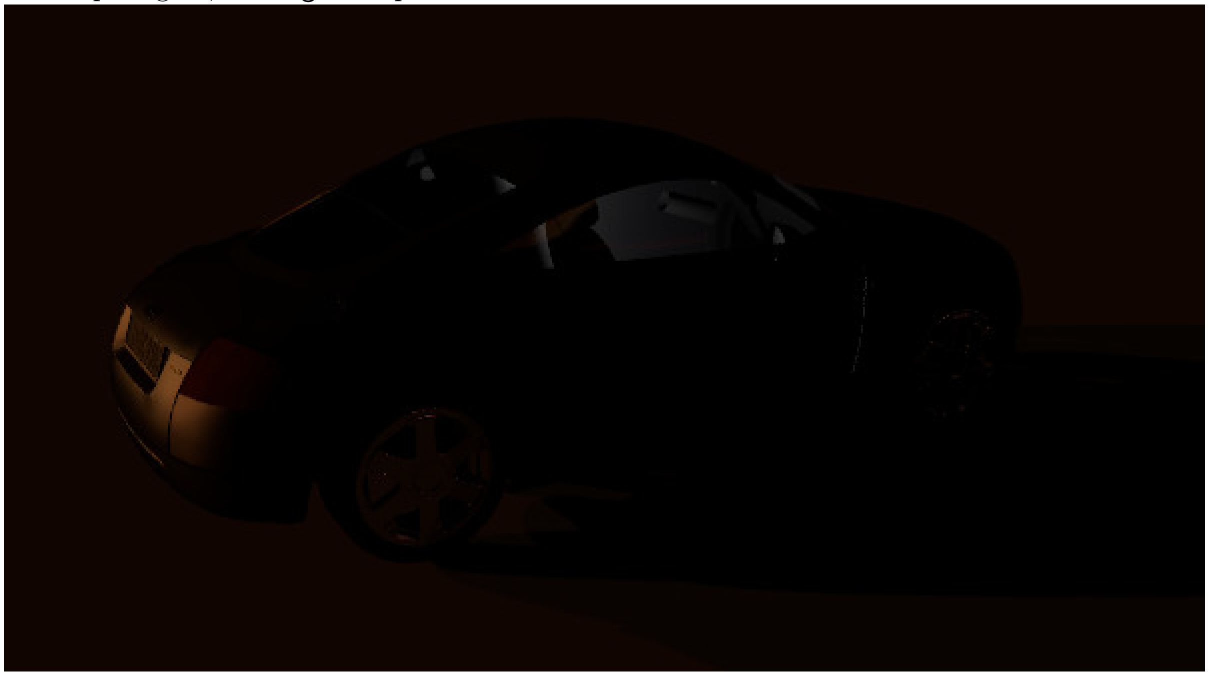


2. Using Median Cut algorithm:

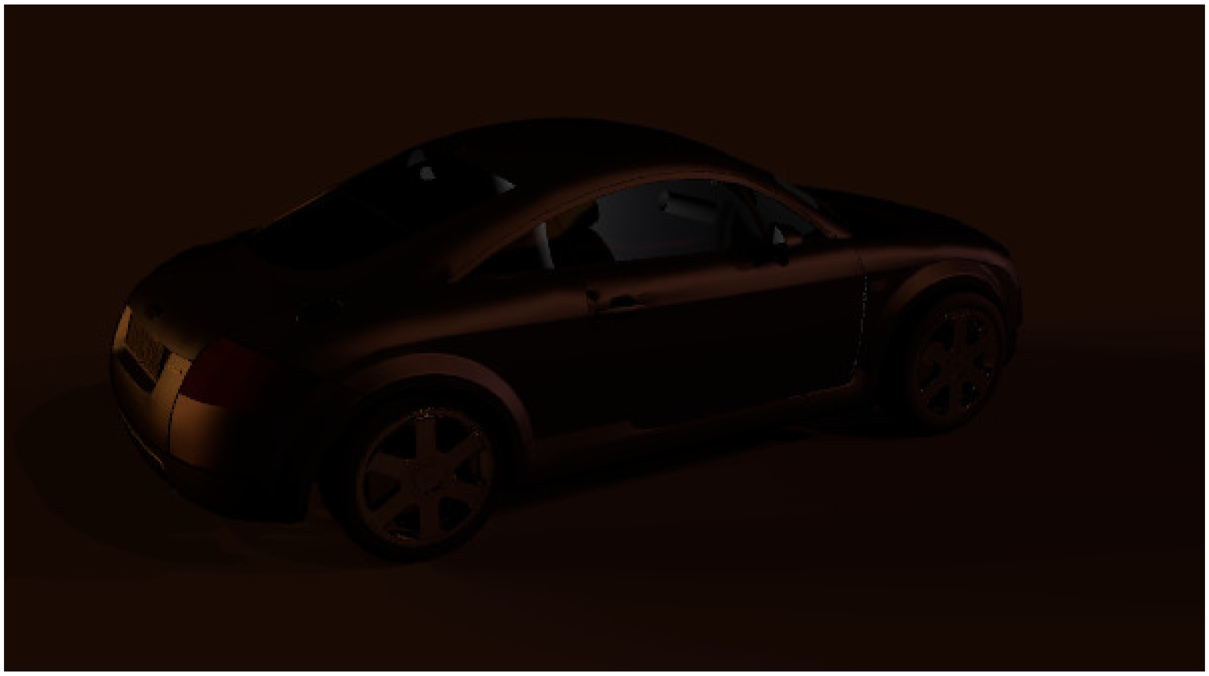
2-a. 4 sample lights, `envlight` map:



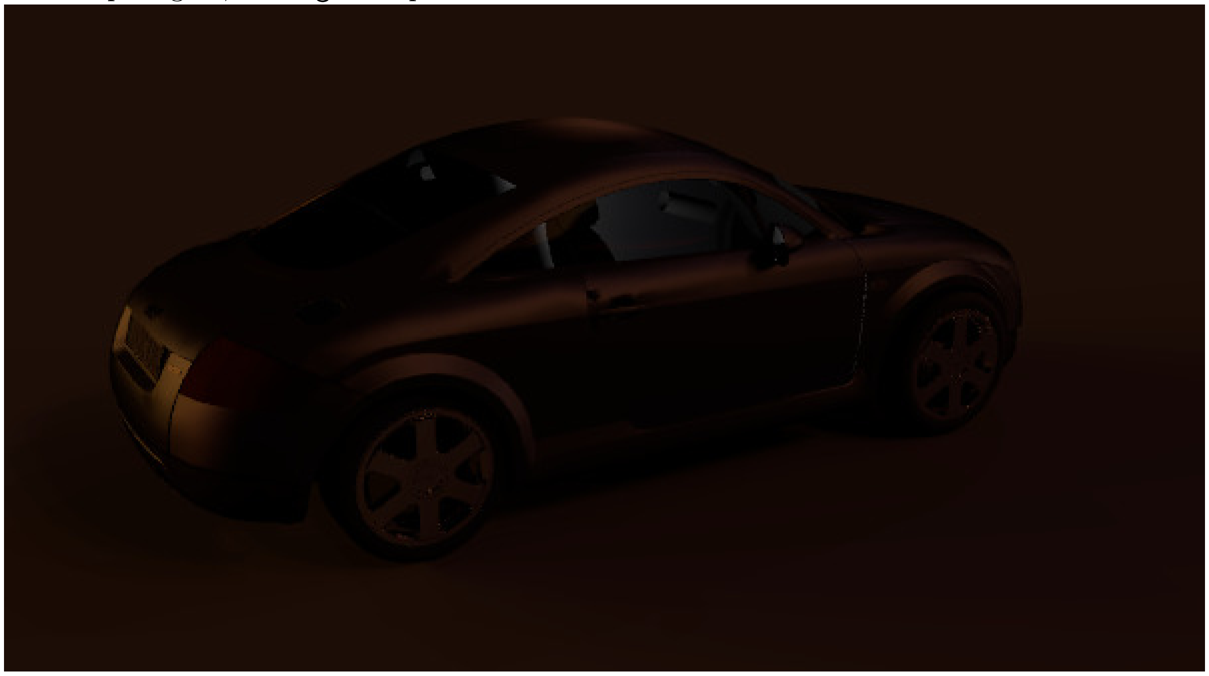
2-b. 16 sample lights, `envlight` map:



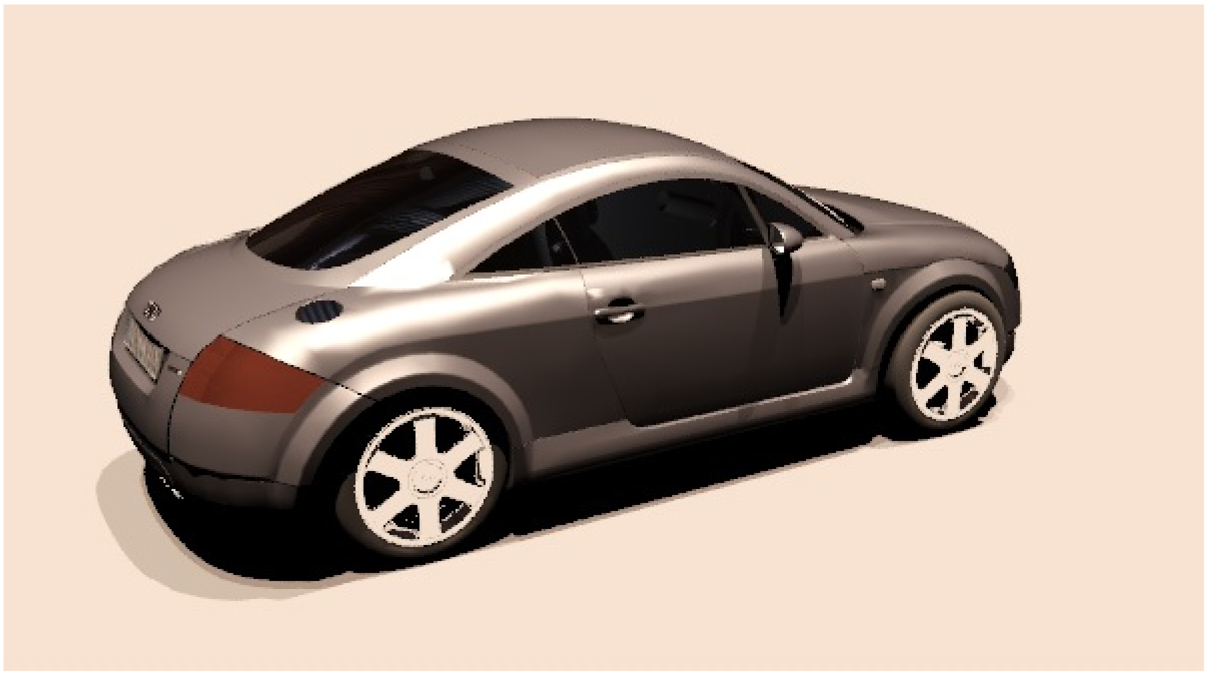
2-c. 64 sample lights, `envlight` map:



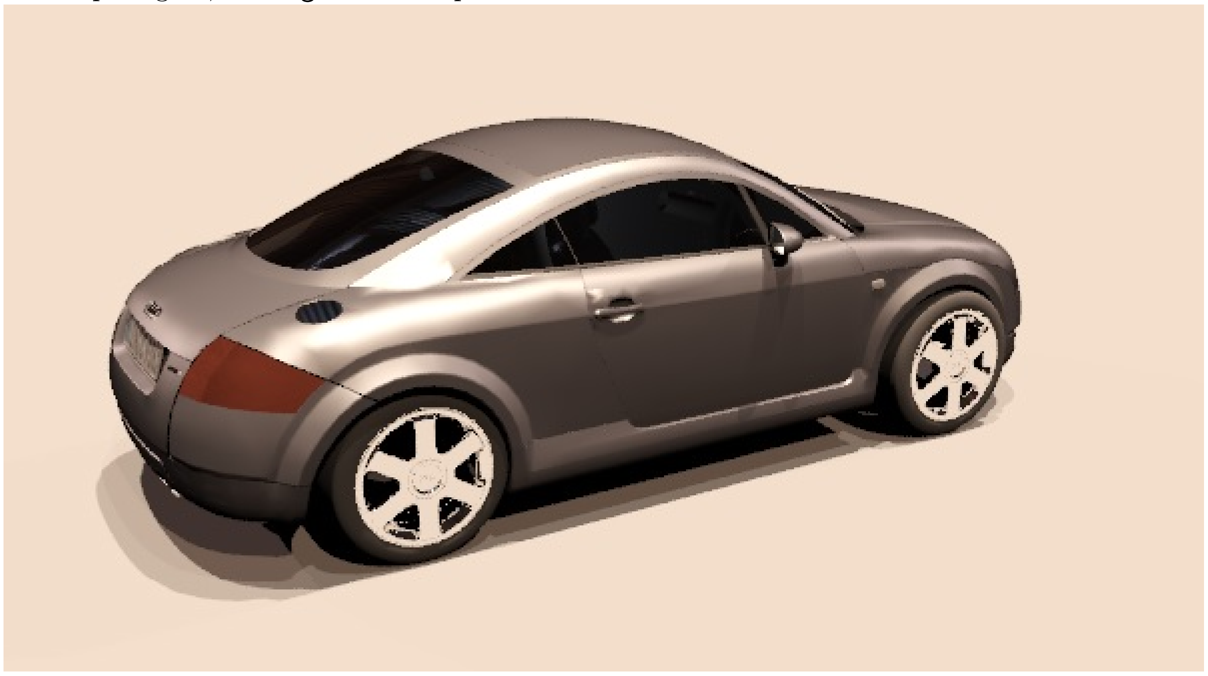
2-d. 256 sample lights, `envlight` map:



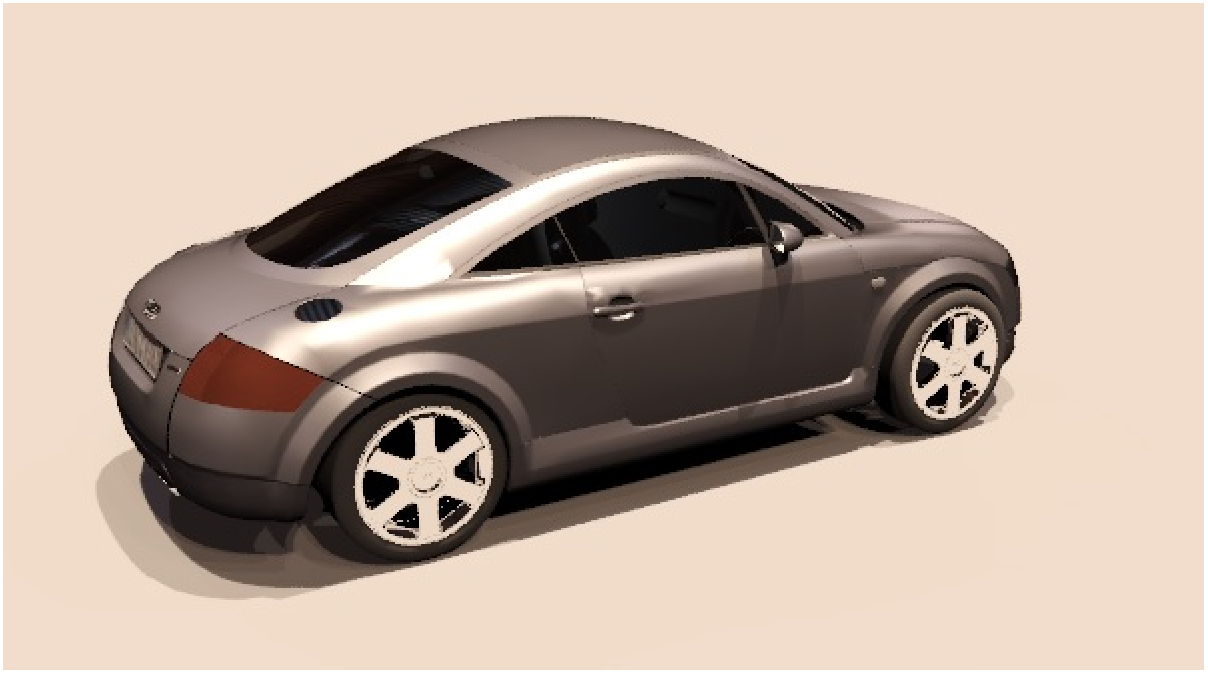
2-e. 4 sample lights, `envlight-new` map:



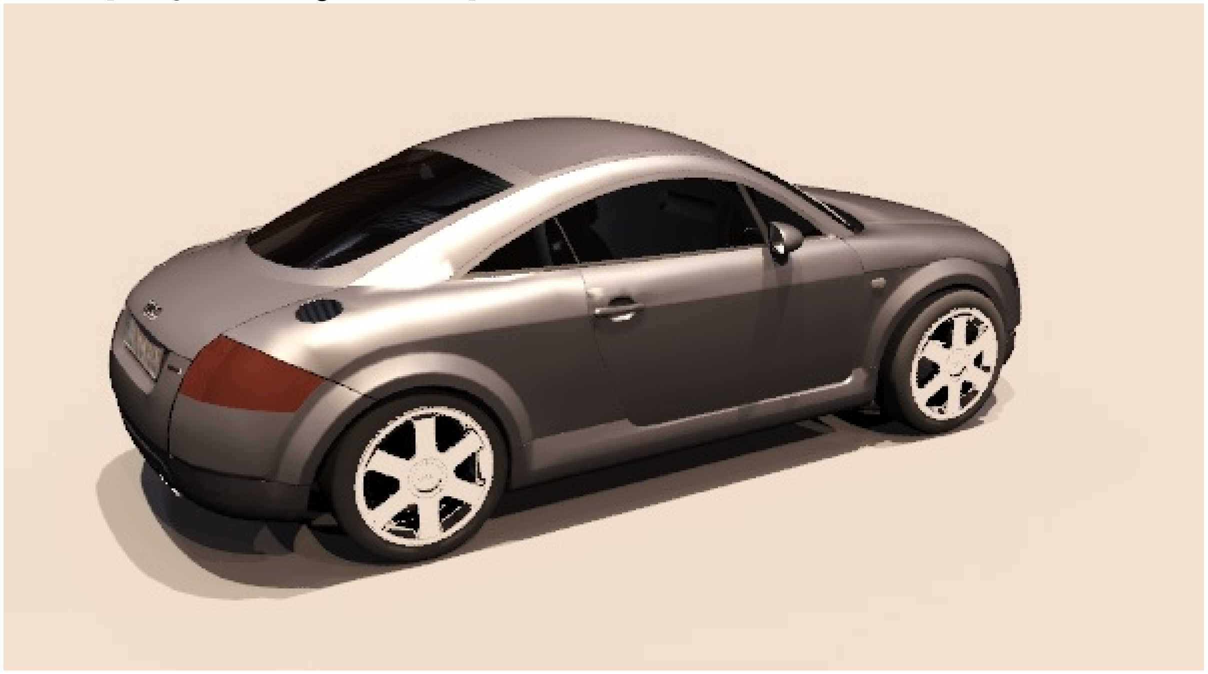
2-f. 16 sample lights, `envlight-new` map:



2-g. 64 sample lights, `envlight-new` map:



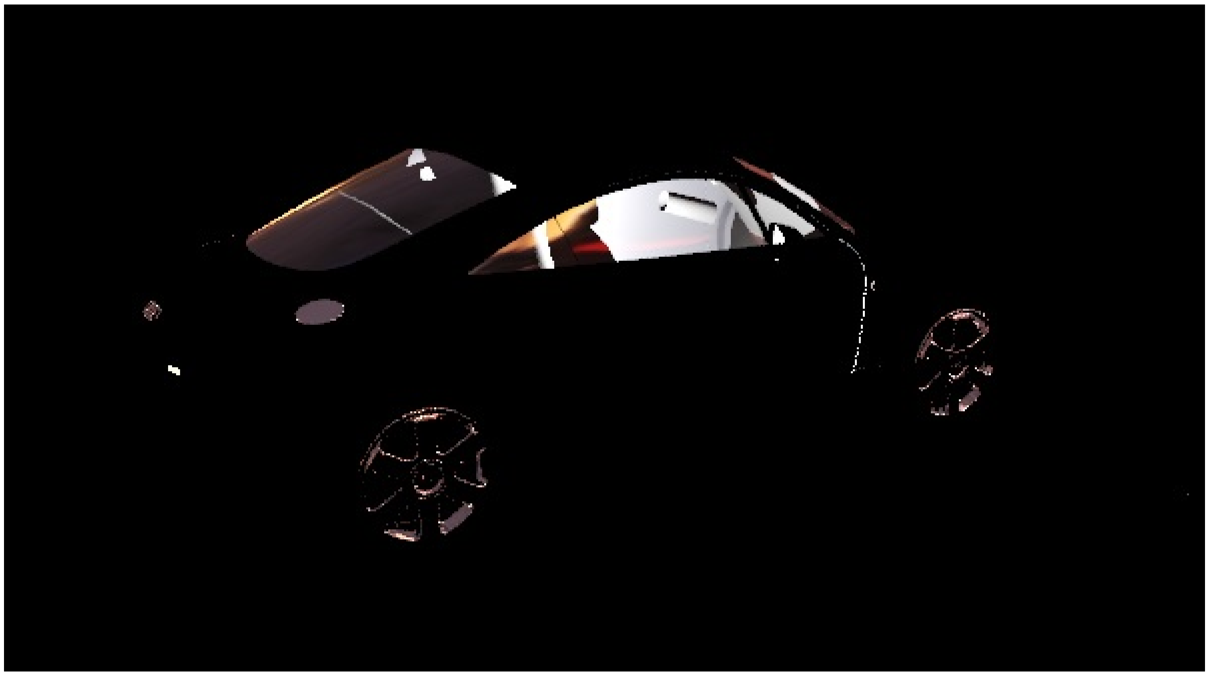
2-h. 256 sample lights, `envlight-new` map:



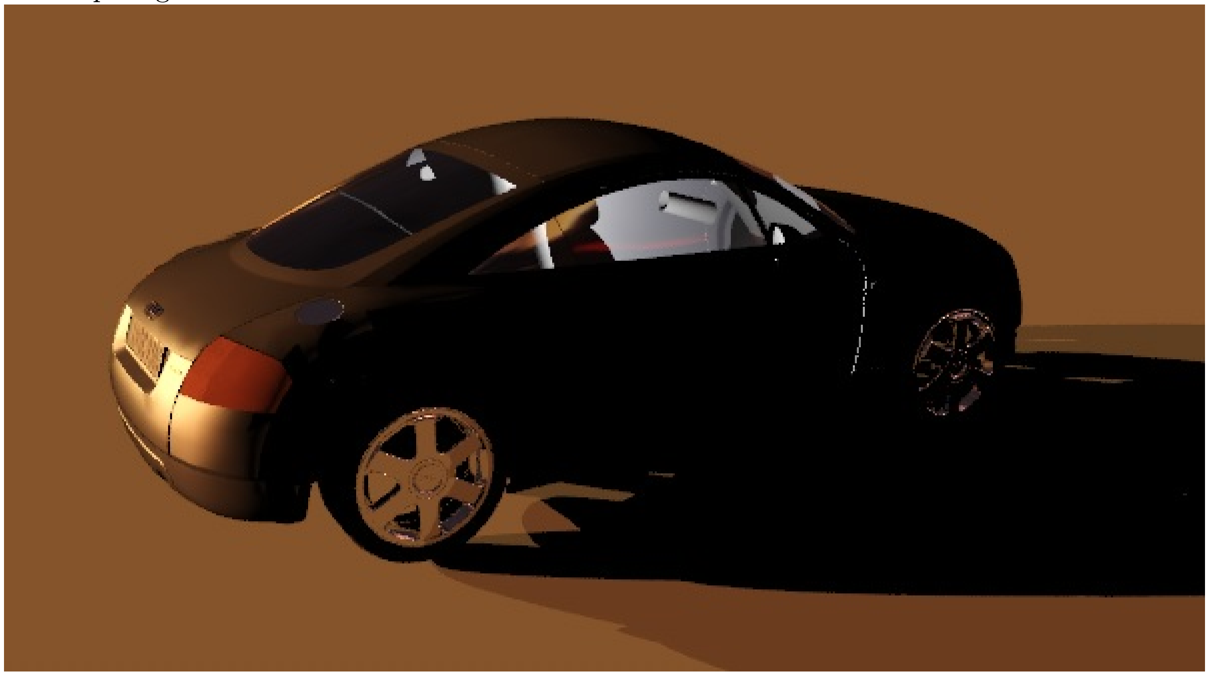
3. Using Median Cut algorithm, a "brighter" version on using `envlight` map[2]:

3-a. 4 sample lights:

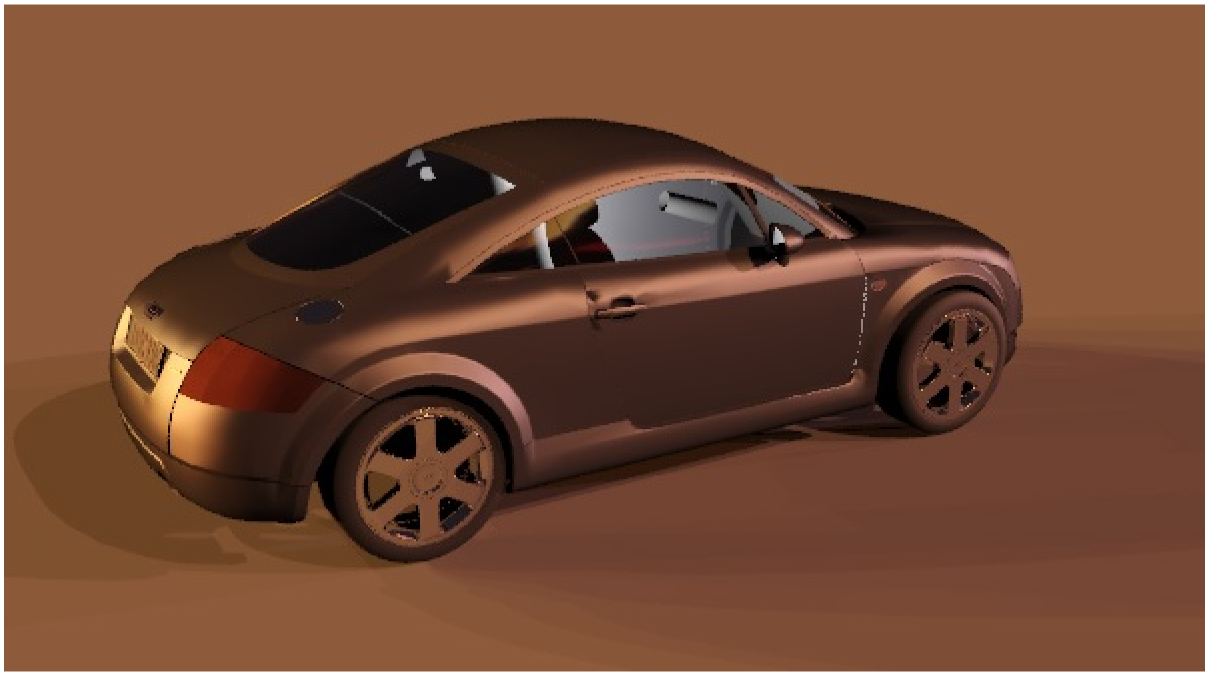
[2]I adjusted the exposure parameter with `exrdisplay` (`http://www.openexr.com/using.html`)



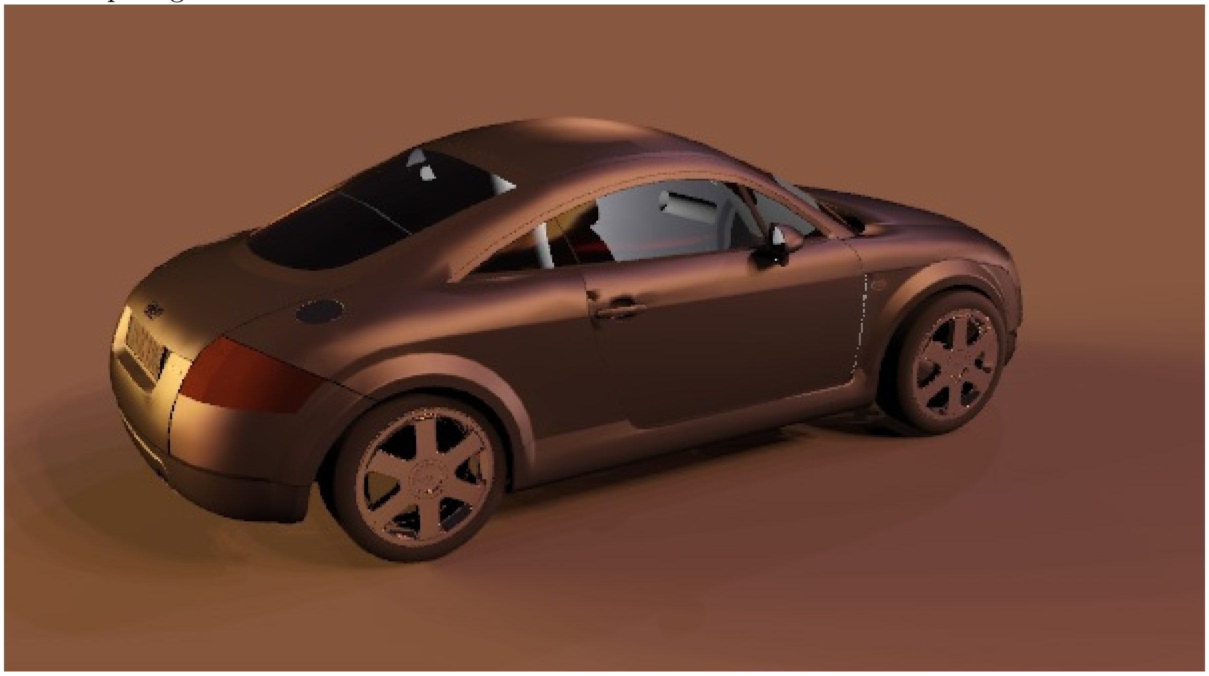
3-b. 16 sample lights:



3-c. 64 sample lights:



3-d. 256 sample lights:



From the results, we can observe some differences between two approaches. For instance, when the number of samples is low, the original approach seems to have more noises. As for the new approach, in the first image (see image 2-a, 3-a), the car is almost entirely dark. This is perhaps due to the chosen sample lights being coincidently on the other side, making the camera side literally be in an entire darkness. (Similarly in 2-b (brighter version in 3-b), the sampled lights are not in front of the right side of the car, making the entire side black.) Also, in image 2-e, the shade beneath the car seems to be unnatural. When the number of lights increases, the images seem to become better, and the unusual regions of darkness decreases. However, something interesting is that in image 2-d, there is an unusual bright area on the top of the car. In general, the new approach will not generate noises, and has a good approximation when the sample lights are many enough, and it's also an acceptable way for light sampling when one would like to make the rendering process faster.

Adjusting the chosen position of the lights: center or centroid In my implementation above, I used the squared-weighted centroid formula. I've stated that it is due to the physical properties of the light, and such implementation will make the final sample position closer to the brighter areas. However, according to the paper, one can either *"place a light source at the center or centroid of each region"*, so I additionally implemented two more kinds of "center/centroid" formulas:

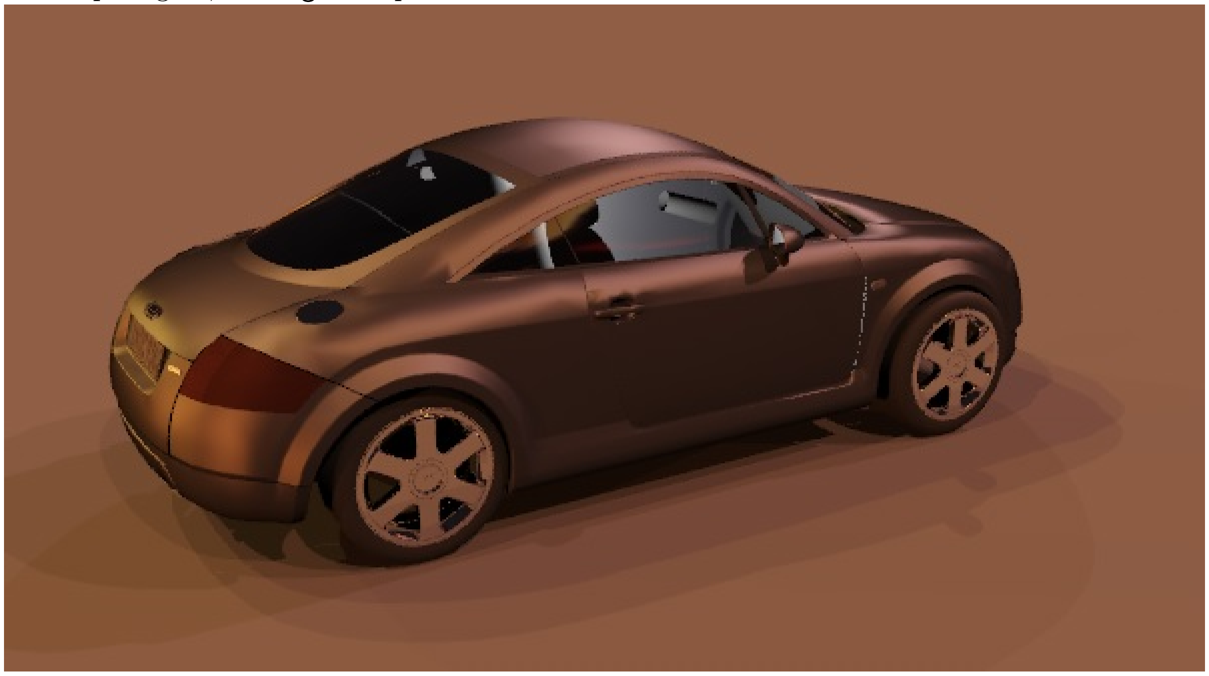
$$\text{center: } (u_c, v_c) = \left(\frac{\min u + \max u}{2}, \frac{\min v + \max v}{2}\right)$$

$$\text{``normal'' centroid: } (u_c, v_c) = \left(\frac{\sum_{(u,v)\in A} y(u,v)\ u}{\sum_{(u,v)\in A} y(u,v)}, \frac{\sum_{(u,v)\in A} y(u,v)\ v}{\sum_{(u,v)\in A} y(u,v)}\right)$$

And the following are the results of some scenes[3]:

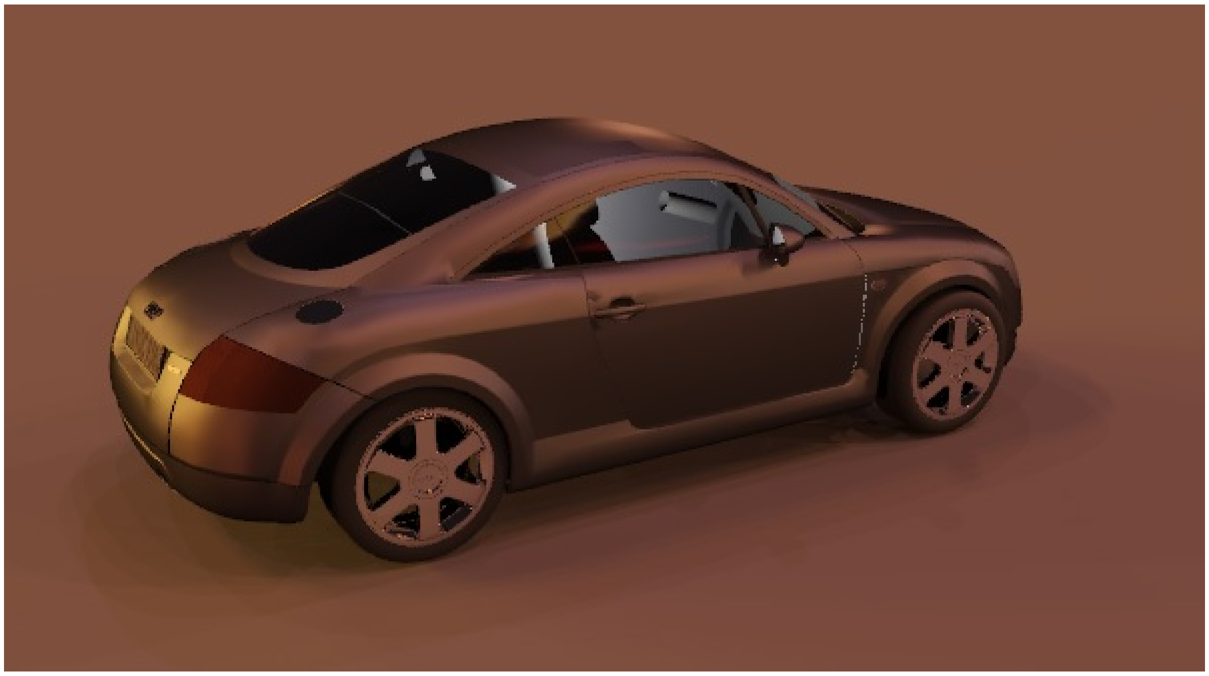
4. center:

   4-a. 64 sample lights, `envlight` map:

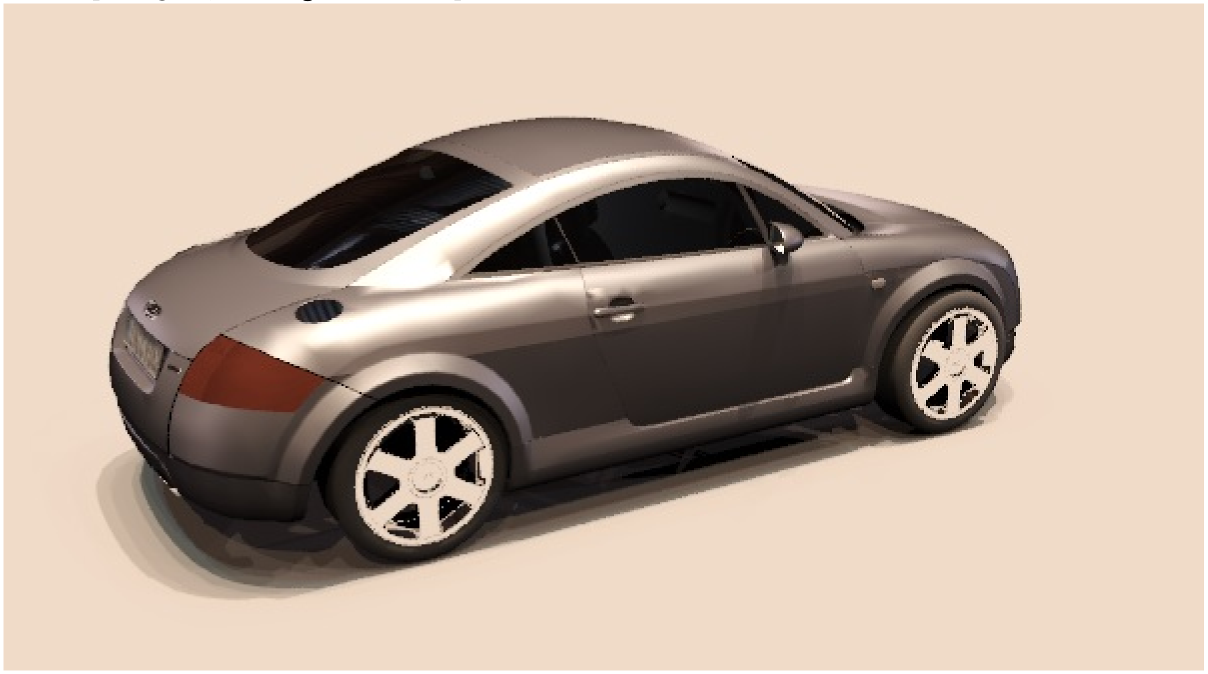


   4-b. 256 sample lights, `envlight` map:

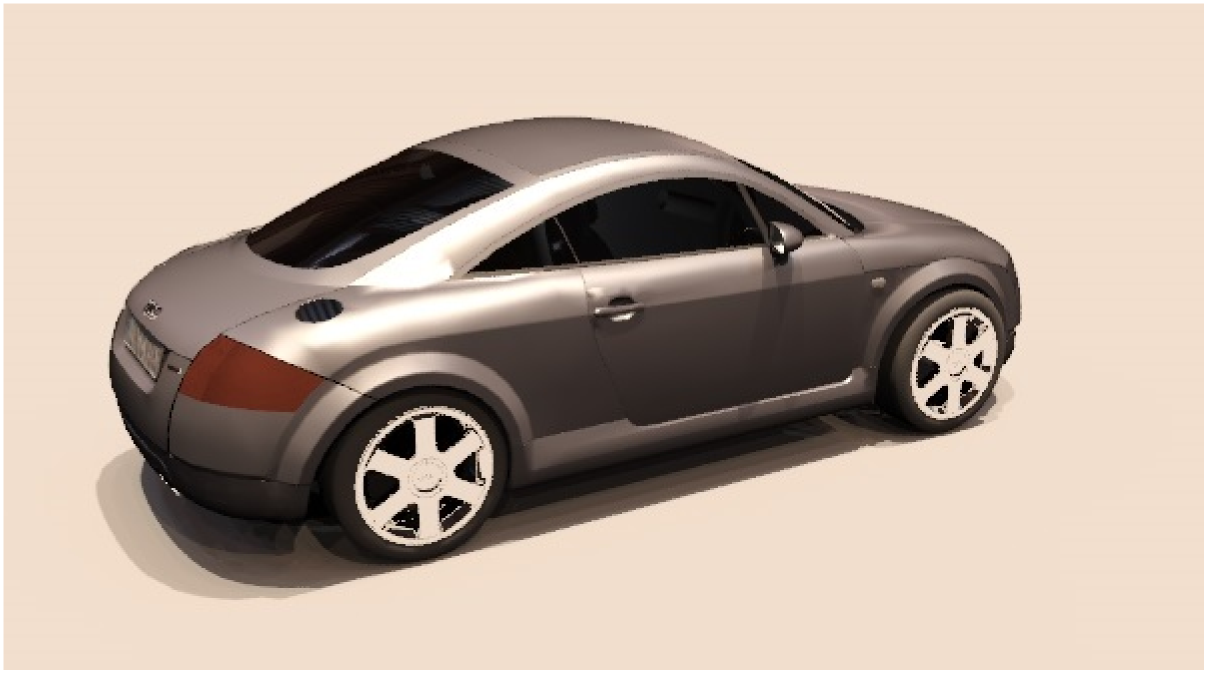
[3] Exposure are all adjusted if necessary.



4-c. 64 sample lights, `envlight-new` map:

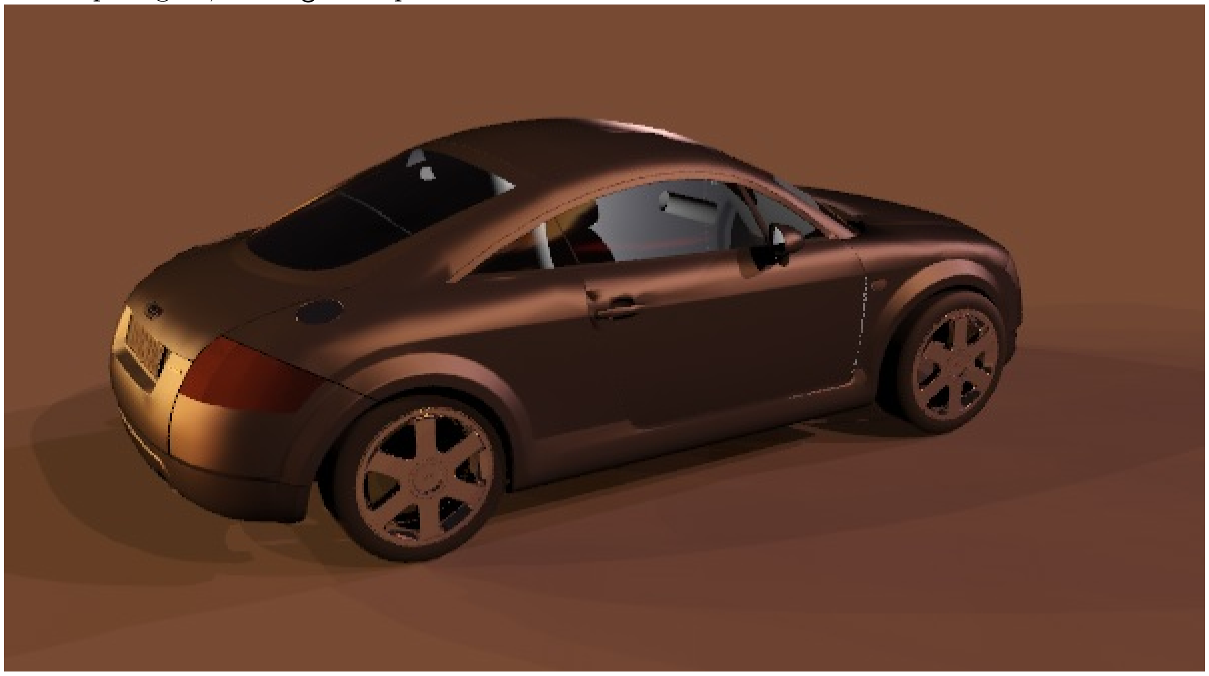


4-d. 256 sample lights, `envlight-new` map:

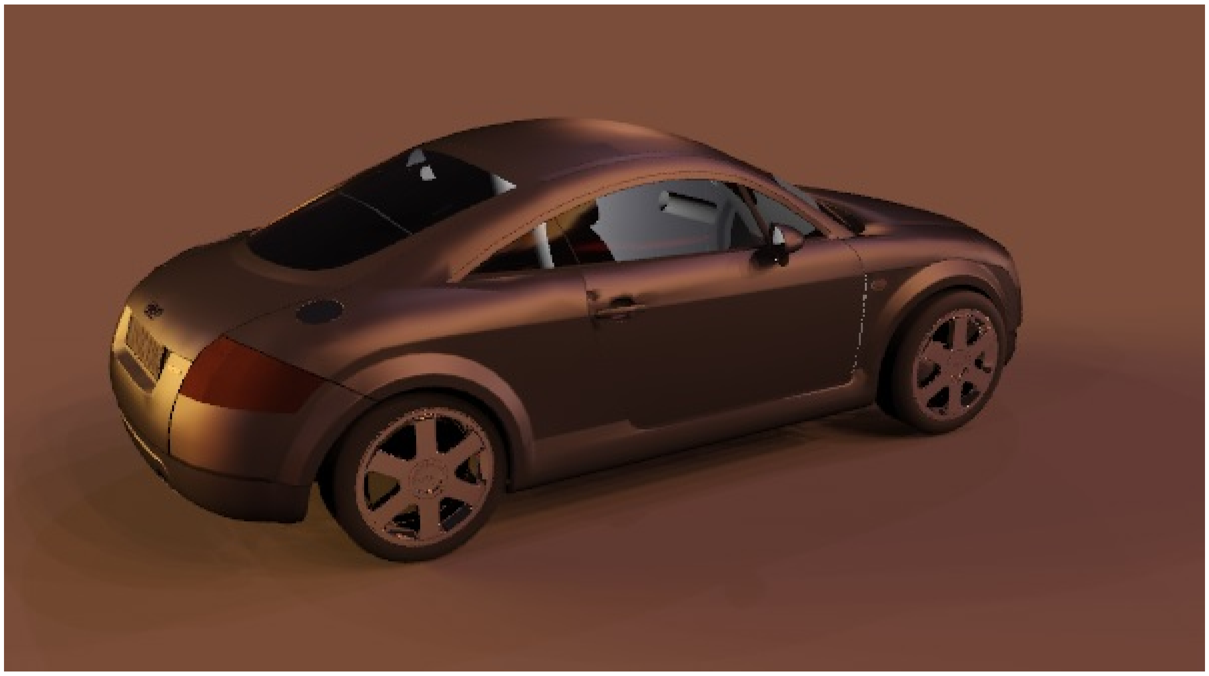


5. “normal” centroid:

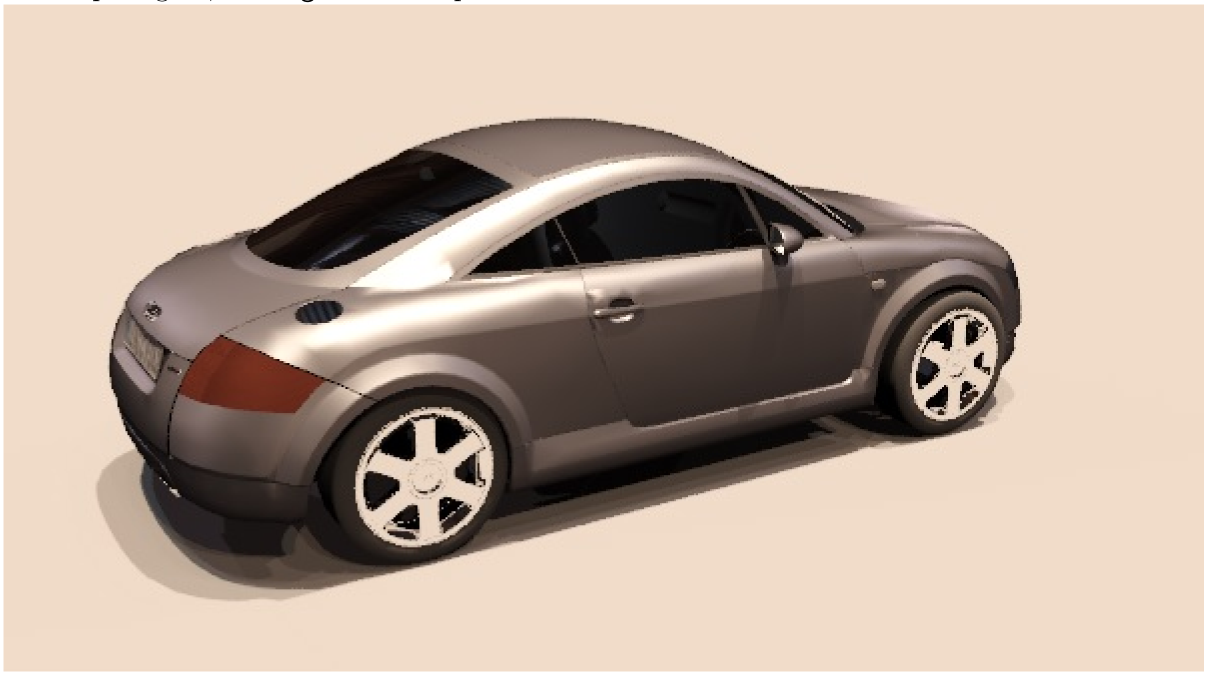
   5-a. 64 sample lights, `envlight` map:



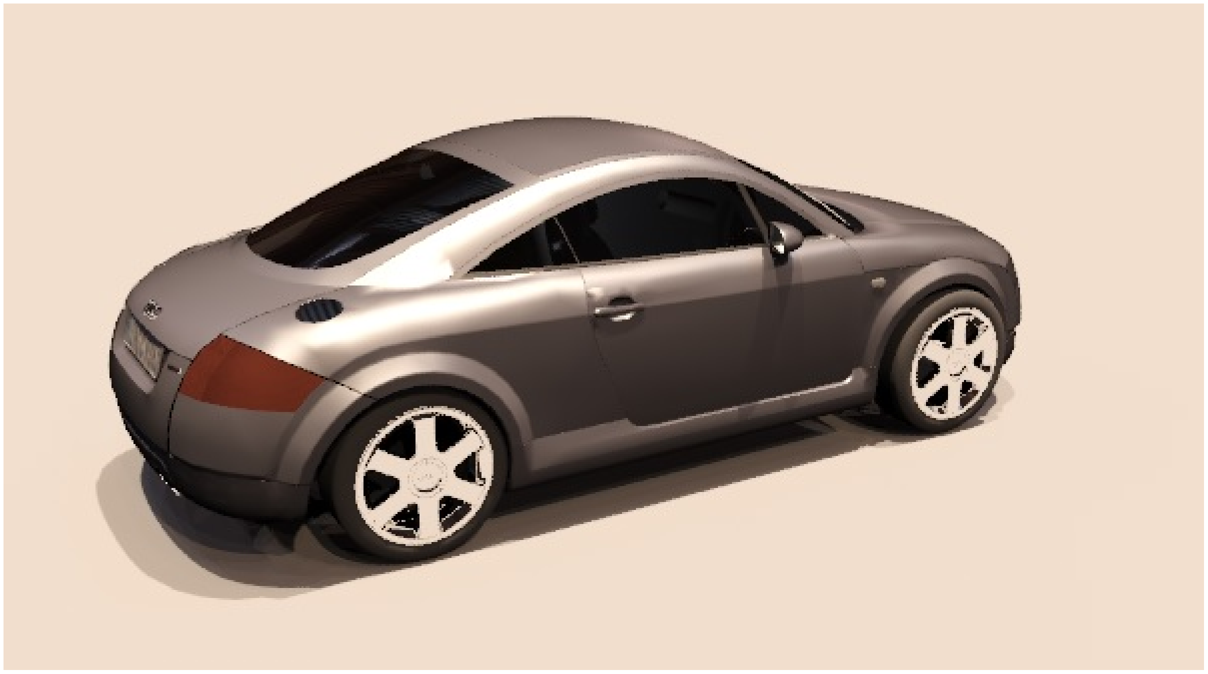
   5-b. 256 sample lights, `envlight` map:



5-c. 64 sample lights, `envlight-new` map:



5-d. 256 sample lights, `envlight-new` map:



For the "center" implementation, when we examine image 4-a, there is an unusual light area on top of the car (and the front engine), but no apparent light on the back side of the car. Image 4-b is better. This is perhaps that the "center" method may make the real bright area be not chosen, but when the number of cut areas increases, the center of each region will eventually become closer to the real bright area.

As for the "normal centroid" implementation, it is more similar than the "square-weighted centroid" implementation then the "center" implementation does, but in image 5-a, the intensity of the "unusual" car roof light seem to be larger than other versions.

In conclusion, all methods has some slight discrepancies with the original sampling method, but in some cases they do can render images that are faster and satisfiable. In my opinion, the "center" method has easier calculation, but the results are good enough only when the number of lights is large enough. The two "centroids" may be similar, and it may be acceptable to apply the normal one.

Other notes

During the implementation, I've been confused that when I just return the original summed intensity in `Sample_L()`, the entire result will be extremely bright. Although according to the homework description, it states, *"Note that your images might be darker or brighter overall than pbrt's. It is fine because it can usually be overcome by tone mapping. Use programs such as Photomatix could solve the problem"*, I just divide the summed intensity with a constant (in my implementation: $\frac{wh}{40}$) such that the brightness directly rendered will be in the acceptable range.

Furthermore, since the images have been converted to `.jpg` with some converter, their brightnesses may be somewhat different from the originally rendered images.