KRAY User manual

Index

What is GI?
Lighting
Ambient light
Background
Luminosity
Other lights
Distant lights
Point, Spotlight, Linear, Area
Materials
Luminosity
Balance
Luminosity
Transparency
Balance rule

What is GI?

There is a nice definition of what GI is in Wikipedia:

Global illumination algorithms used in 3D computer graphics are those which, when determining the light falling on a surface, take into account not only the light which has taken a path directly from a light source (*direct illumination*), but also light which has undergone reflection from other surfaces in the world (*indirect illumination*).

Images rendered using global illumination algorithms are more photorealistic than images rendered using local illumination algorithms. However, they are also much slower and more computationally expensive.

Higher computational complexity of GI algorithms is not the only disadvantage compared to local illumination rendering. There are more rendering parameters to control and we must pay bigger attention to scene lighting and materials.

Light and material models used in 3d packages (such as LightWave) are not always physically correct. For example in LightWave you can create material that has diffuse 100% and reflection 100%. Diffuse + reflection = 200% and that means material reflects (in sense of total light reflection, not specular

reflection only) more light that recieves. Physicall incorrectness in non-GI scene helps to hide lack of global illumination and often looks better than correct models, but in GI scene such models can look very wierd and unnatural.

Kray does not force user to use physically correct models. They are sometimes useful to make rendered images look more *dramatic*, it is important to know when the scene matches the laws of physics and what will happend if we break them.

Lighting

If Kray says **Physically incorrect light model in scene with global illumination** one of your light breaks the laws of physics. Read this section to find out why.

Ambient light

Is a basic way to mimic GI for scenes without global illumination. Without it we get completly black shadows.

If we enable GI and use ambient light objects on the scene will glow (because they actually emit light). For best results when you use GI, set **ambient light** to 0%.

Background

Background also act as a light source in global illumination scene. Note that photons tracing does not support background lighting since there is no point from where photons can be emited (use lightmaps instead).

Luminosity

See luminosity in materials section.

Other lights

Distant lights

Has similar limitations to background. It is ok to use them in GI scene, but they are not supported by photon maps.

Intensity of lights that is higher then 100% also has a physical sense and can be used.

Point, Spotlight, Linear, Area

They are physically correct if **Intensity falloff** is set to **Inverse distance ^ 2**. Otherwise they are not physically correct. No matter if they are correct or not, they work with light maps, but they need to have **Inverse distance ^ 2** in order to work with photon maps.

Materials

Luminosity

Object with surface material **Luminosity**>0 material really emits light and with GI enabled can illuminate your scene (this can even work without GI enabled, if you choose **"Compute as direct" Luminosity model**). If you use luminosity to get better shading only your object will glow (similar to glow caused by ambient light). You are not limited to 100% luminosity. You can use values over 100% as well (same way as light intensity). What is nice in luminosity light sources is that unlike regular lights they can have a texture.

Balance

As I mentioned on the begining in LightWave you can create surface that reflects more light than recieves. That gives more freedom to artist and does not make problems in local illumination scenes.

Problem can appear when we use surface that reflects more then it recieves in global illumination scene. This can cause unlimited light bounce, glowing objects, strong artifacts.

Balance rule

General rule: **Diffuse+Reflection+Translucency<100%** . Notice that it is "*less then*" not "*less or equal then*". In reality no material reflects all the light that hits its surface. The same rule is valid if we use gradients or textures for every gradient angle and texture pixel. Also notice that if we have texture color RGB (128,128,128) and *Diffuse* 100% material still fits **Diffuse<100%** rule. In fact it is 50% diffuse because it is the same as RGB (255,255,255) and *Diffuse* 50%. Pure 100% diffuse reflection is when color is RGB (255,255,255) and *Diffuse* amount is 100%.

Luminosity

Balance rule does not affect *Luminosity*. Feel free to use 300% *Luminosity* and 95% *Diffuse* for example.

Transparency

Transparency (the one from *Basic* tab in surface editor) fades *Diffuse* channel (so 90% *Diffuse* + 40% *Transparency* is ok)

because:
90% *diffuse* * (100%-40% *transparency fade*) + 40% *transparency* = 54% *diffuse* + 40% *transparency*

Transparency does not fade *Reflection* (so 90% *Reflection* + 50% *Transparency* is too much).

Additive transparency does not affect *Diffuse* amount (so 90% *Diffuse* + 50% *Addtive transparency* is 140% and is not ok).

KRAY User manual

Introduction

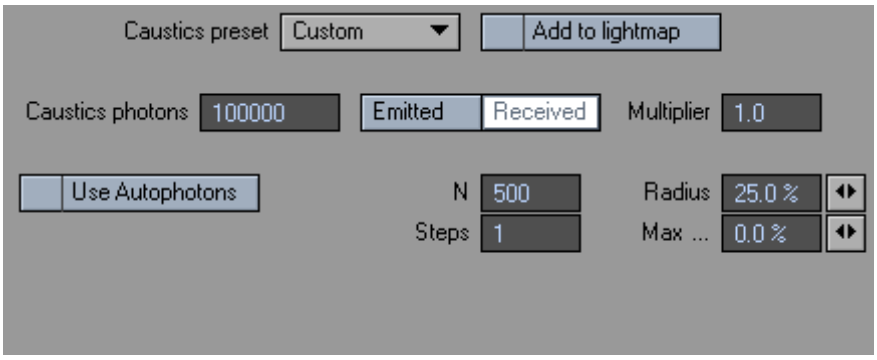
Rendering caustics is a two stage process. At first stage rays (photons) are fired from light sources and stored in *caustics photon map*. Second phase is actual rendering with gathering caustics from caustics map.

Caustics test scene

Here is our caustics test scene rendered in **Raytrace** mode.

We have reflective ring and refractive ball here, area light with **Inverse square^2** falloff and **Ambient** light set to 0%.

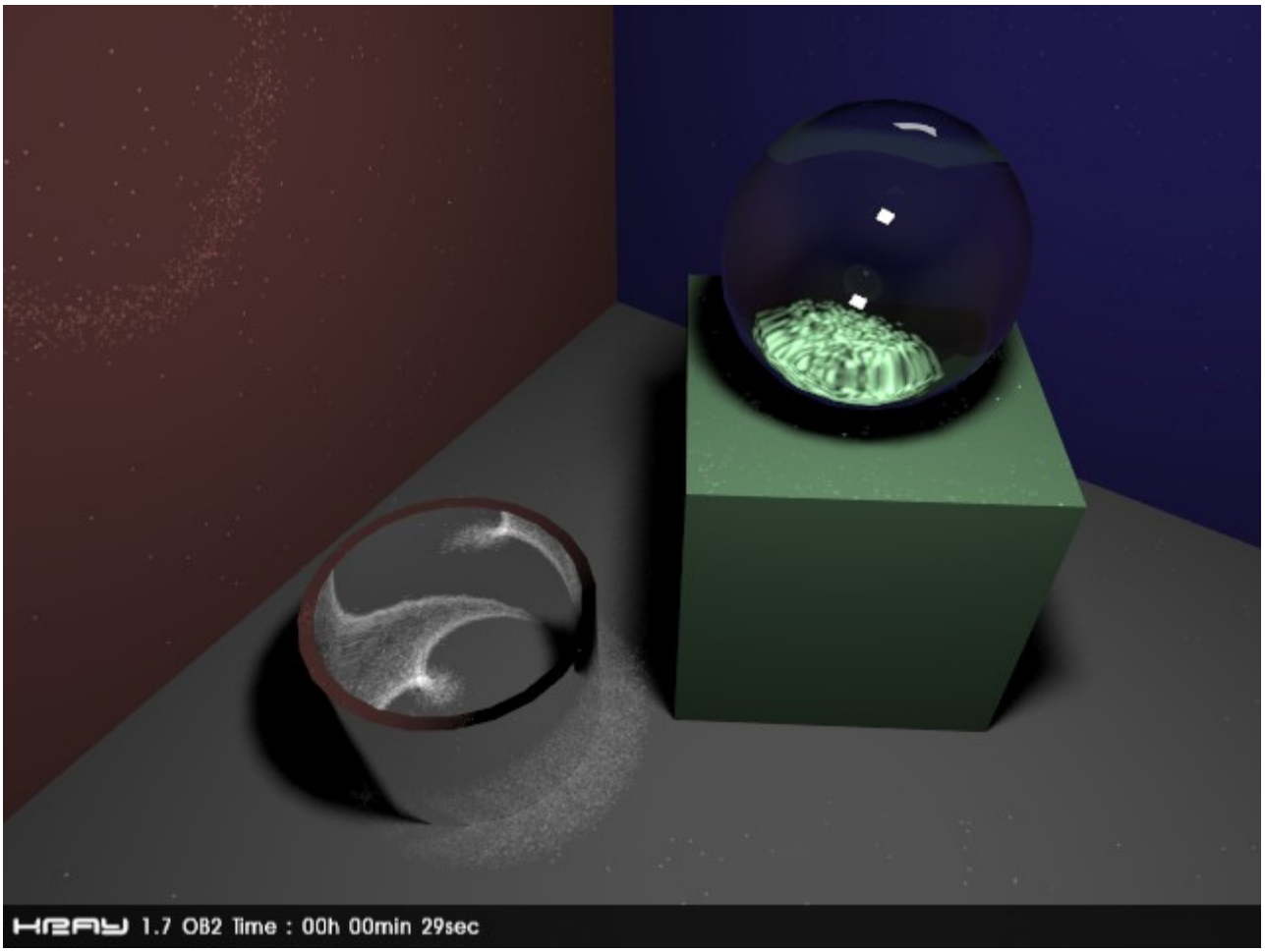
For better understanding how caustics are rendered we will turn on *manual* mode for them. We go to **Photons** tab and choose **Custom** for caustics. Then we turn off **Use autophotons** and set **Max**. We don't bother **N** and **Step** because they are ignored anyway when **Max** is zero. Our initial setup should look like this:



We should also turn off **Auto** for **GI resolution** and set it manually to 100mm.

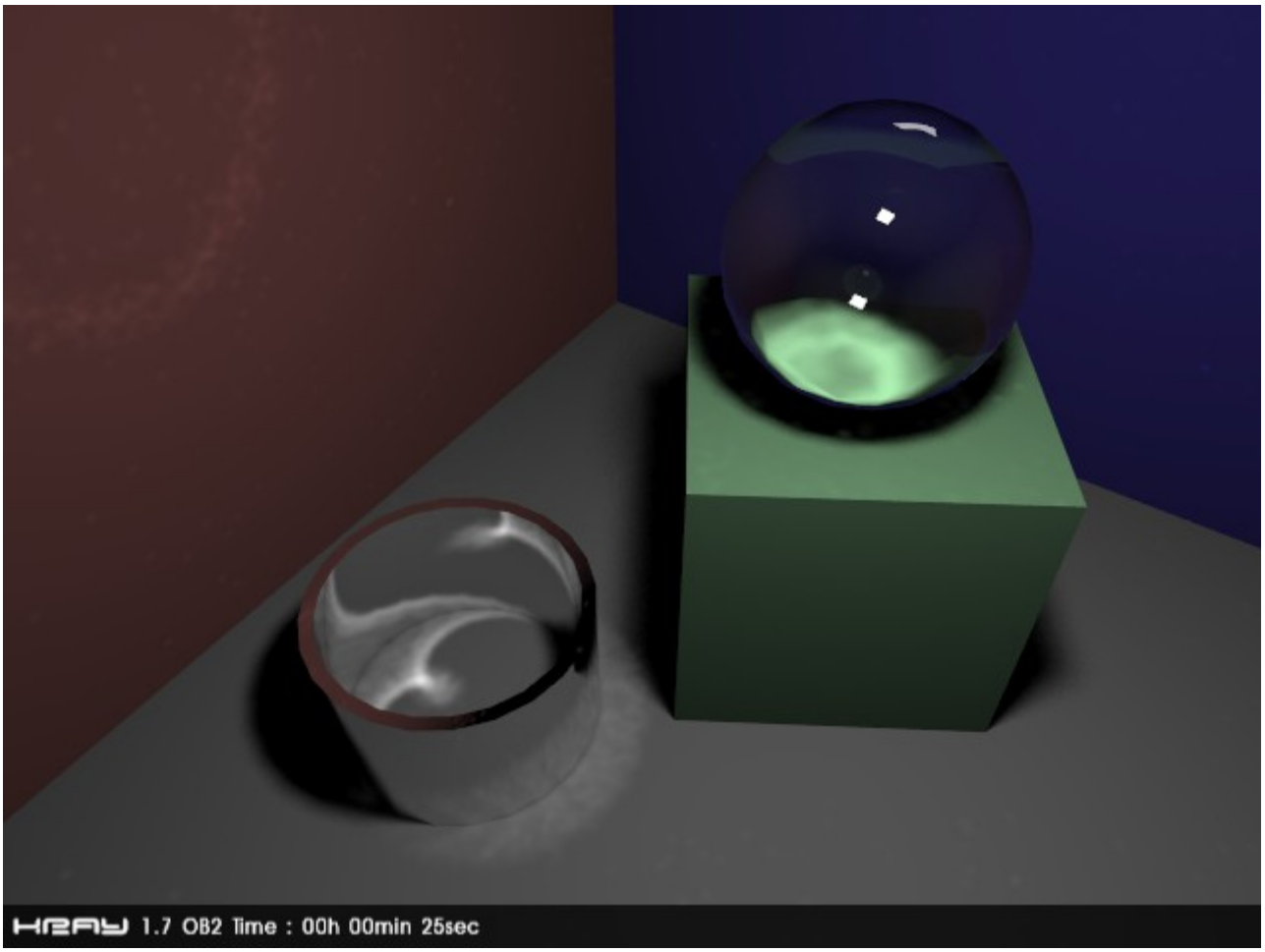
Now we can enable caustics (**General** tab **Diffuse model**)...

... and render.



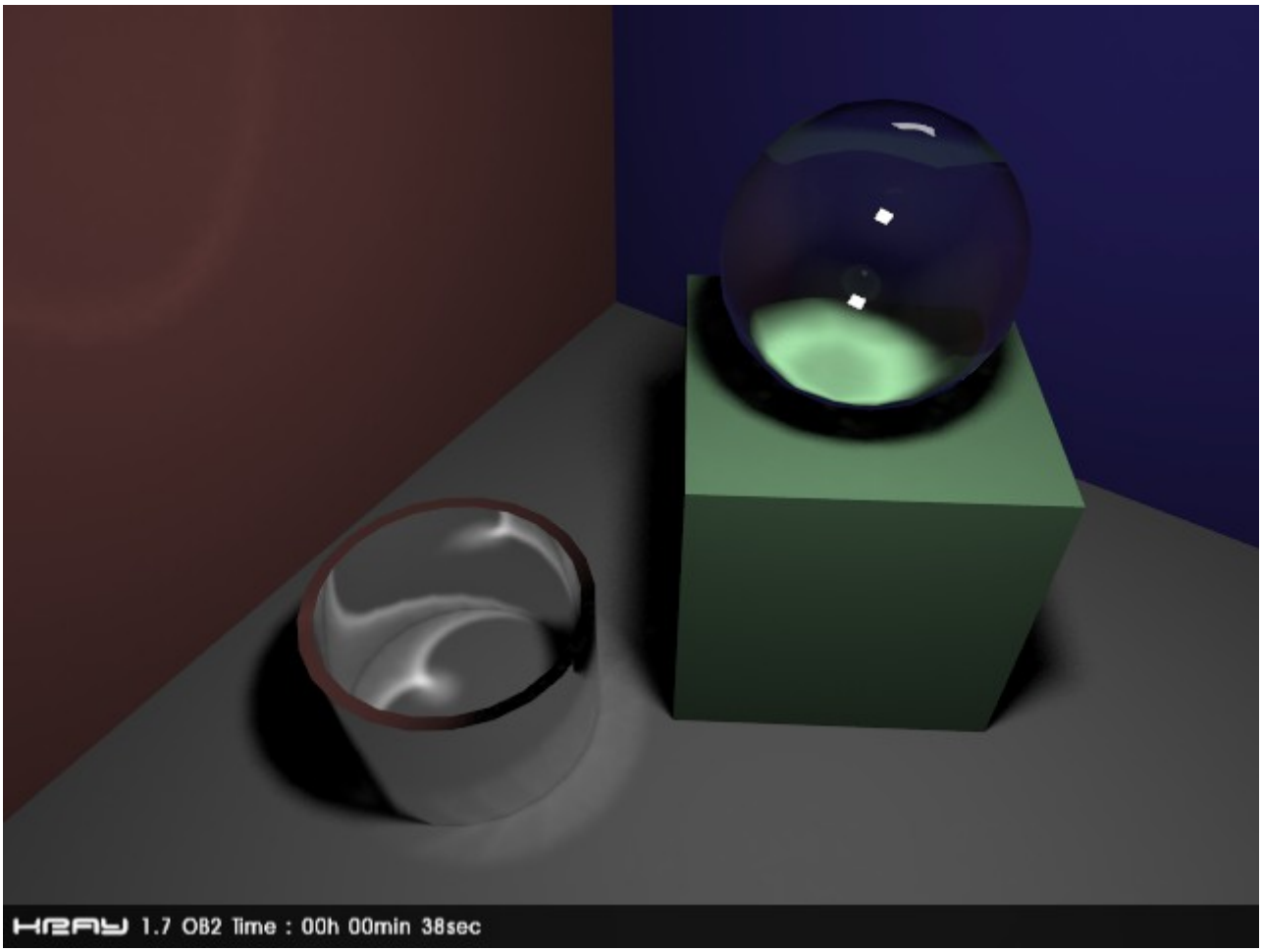
Caustics filter

Caustics rendered, but looks noisy. We can see every particular caustic photon. We can avoid this by increasing caustics filter radius. This is **Radius** parameter in caustics section of **Photons** tab. Why is it expressed in % not in meters? **Radius** like many other parameters in Kray is expressed relative to **GI resolution**. This is helpful when we want to scale all settings for another scene. We don't need to change every size related parameter, just **GI resolution**. In our scene we don't use neighter global photons nor final gathering so it does not matter what we will change **GI resolution** or **Radius**. If we have full GI scene changing **GI resolution** will affect all size related GI parameters while **Radius** will only affect caustics.
I have changed **Radius** from 25% to 100% (so it is now 100mm, equal to **GI resolution**).



Caustics photons number

As we can see caustics are less noisy then before, but they are also little blurred so details are not so well visible. If we want better quality increasing filter even more will not help (will blur caustics to much). Instead of increasing filter we can increase number of **Caustics photons**. Lets see what happens when I increase **Caustics photons** 10 times (from 100000 to 1000000).



Quality has improved a lot. Of course not without speed impact, but I think it was worth it.

For this render we have used 1000000 photons fired from light source. Not all fired photons were stored on photon map. Some of them did not hit neighter reflective nor reflective surface. Such photons does not cause caustics and are not stored in photon map. If we check Kray log, we can see how many photons actually were stored on the map.

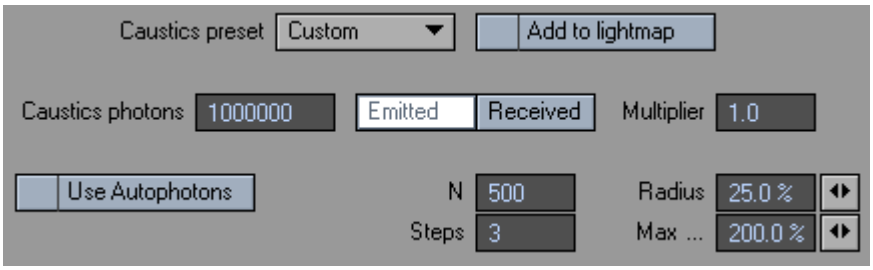
```
*** Caustics PM tracing
*** Building caustics KD 436486/1000001 (43%)
```

We can see that Kray fired actually 1000001 photons (difference of one is because of how photons are fired) and 436486 photons were stored on photon map (43% of fired photons). Instead of controlling number of fired photons (emitted) we can control number of stored photons (received). This is what **Emitted/ Received** switch does.

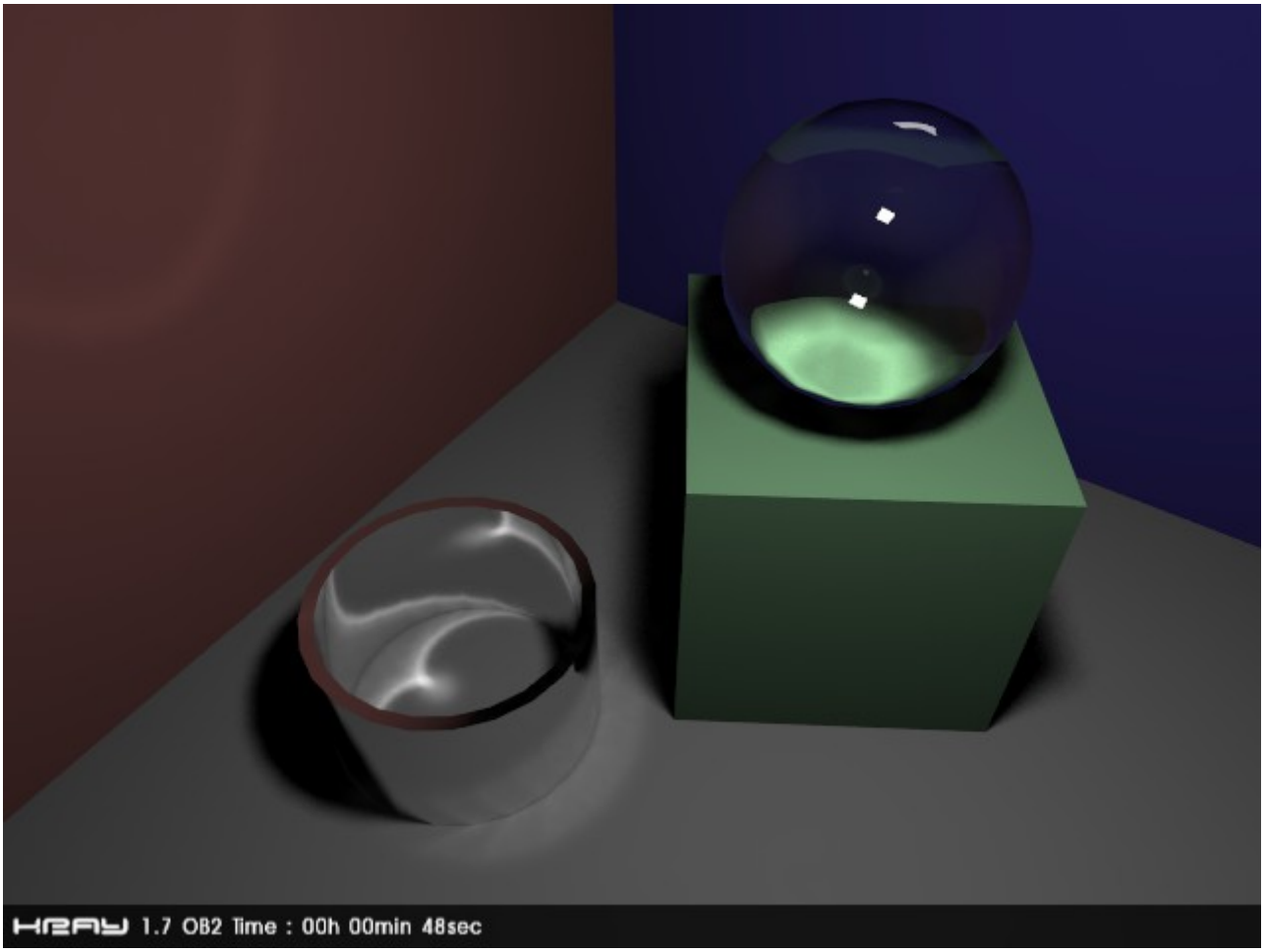
Max, Steps and N

One of the problems with adjusting photons filter size is that their distribution is not even on the whole scene. In one part of scene there are plenty of them and we need small filter there to avoid too much blurring details. On the other side of scene there are few bright dots only and we need larger filter to blur them.

We can tell Kray to adjust filter size depending on photons density on the scene. Instead of expressing filter size in meters, we can set number of photons (**N**) that should fit in filter size. These are settings I've used:



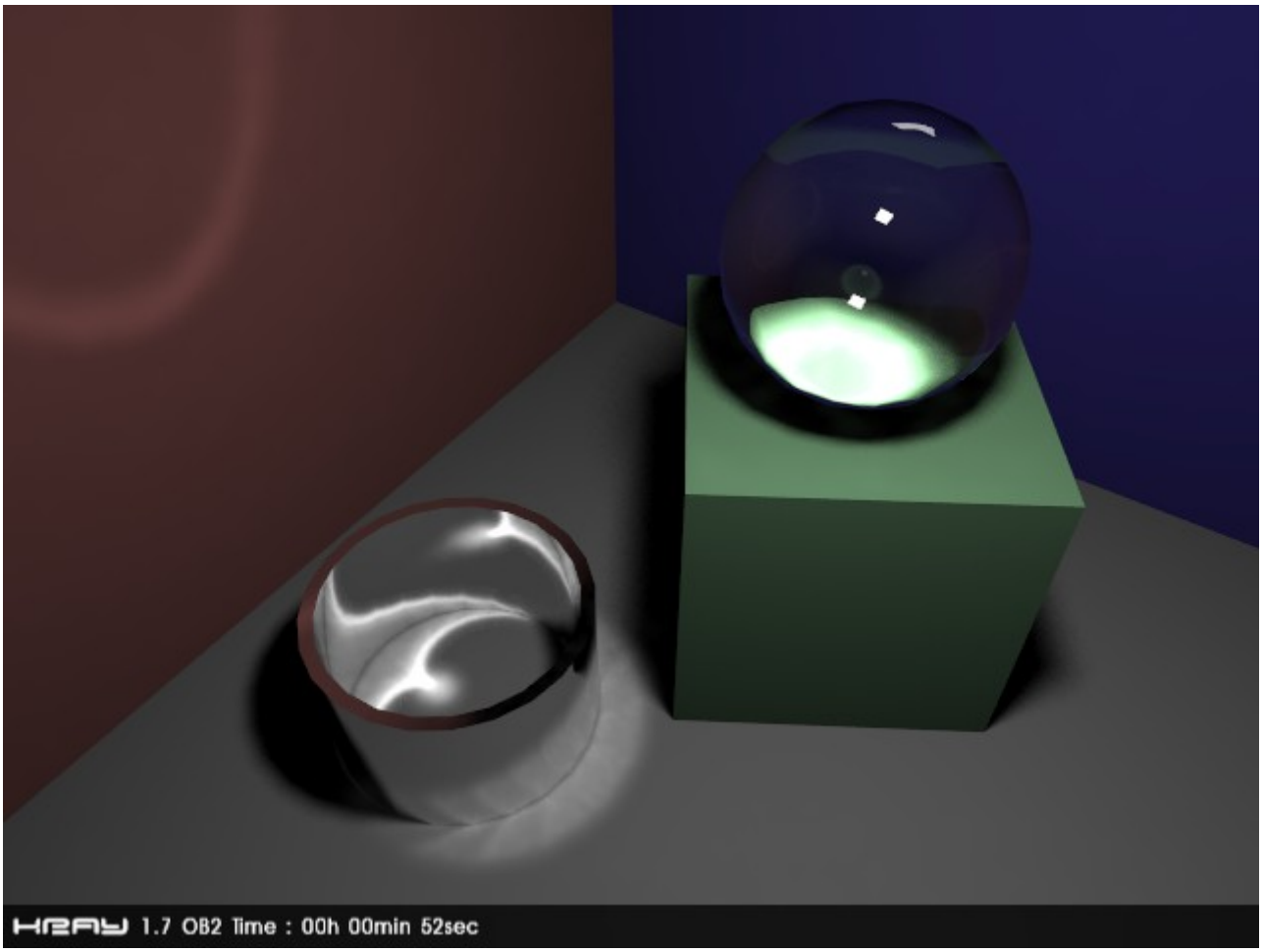
This means for Kray *"Try 20% filter size, if number of photons in filter is smaller then 500 increase filter, but not use larger filter then 200%. Use maximum 3 steps of filter incrementation between 20% and 200%"*. Here is a result:



Note that caustic inside the rings is less blurred while caustic on red wall is less noisy.

Multiplier

Multiplier is just caustic brightness. Default value of 1 means physical accurate computing. We can violate laws of physics to achieve more dramatic effect. Here is our scene with **Multiplier**=3.



Autophotons

Setting **Radius**, **Max** and **Step** manually is not very comfortable. We never know how many photons will fit in given filter radius. That is where **Autophotons** can help.

In preprocessing phase of rendering, **Autophotons** computes distribution of photons on the scene and sets according to this distribution. **Low** and **High** percentages are equivalents of **Min** and **Max** radius, but they are expressed in relation to the real photon density of the scene. If **Low** value is too big, regions in scene with a high photon density (low radius) may contain too few

(less than **N**) photons. If **High** value is too low, areas with a low photon density (big radius) will contain much more photons than **N**. Setting the Low value too low and **High** value too high will increase the rendering time. **Dynamic** is equivalent of **Steps**, but it automatically adapts to the real distance between regions of high and low photons density. That means if for example **Dynamic** is set to 10 and density of photons of the scene is even, the autophoton system will set **Step** to 1. But in case the photon density varies in different scene locations the **Steps** value may be higher, but not bigger than 10. Too low a **Dynamic** value may appear as noise on caustics. Too high a value may slow down the rendering process.

KRAY User manual

I n d e x

Header commands
- var __blendss
- var __undersample_edge

Tailer commands
- recurse
- limitdr

Other commands
- about
- cam_singleside
- debug
- echo
- include
- lmimportance
- logfile
- lwtexturesystem
- lwtransoptions
- mipmap
- octstats
- octsmallcube
- photons_singleside
- postprocess
  - *add*
  - *autoexp*
  - *filter3*
  - *filter5*
  - *gblur*
  - *mult*
- rem
- renderinfo
- renderinfosize
- showcornersamples
- showphstats
- smoothprecachescale
- surface_flags

K r a y c o m m a n d s

Kray has a built in script language for scene definition. It allows user to define the scene in a text file. LightWave's version of Kray uses script to pass parameters from GUI to Kray render core (you can see it if you open .lws file with Kray settings in a text editor). Some Kray options are not exposed in GUI

and are available only from Kray script.

You can run Kray script commands from Kray GUI level. On the misc Tab there are 2 fields: **Header commands** and **Tailer commands**. They correspond to Kray script commands fired before importing LW scene (Header) and after import (Tailer). Some commands should be fired before importing LW's scene, some after and some works no matter where they are placed.

Header commands

var __blendss,*value*

Determines how polygons casts shadows (for direct illumination).
value = 0 - polygon cast shadow no matter if it is front sided or back sided to the light source.
value = 1 - only front side of polygon casts shadow (like in LW).
value = 2 - only back side of polygon casts shadow.
Example:

```
var __blendss,0; // default value
```

var __undersample_edge,*value*

Sets undersampling mode. *value* = 1 means edge undersampling (slower, but more accurate). *value* = 0 is corner undersampling.

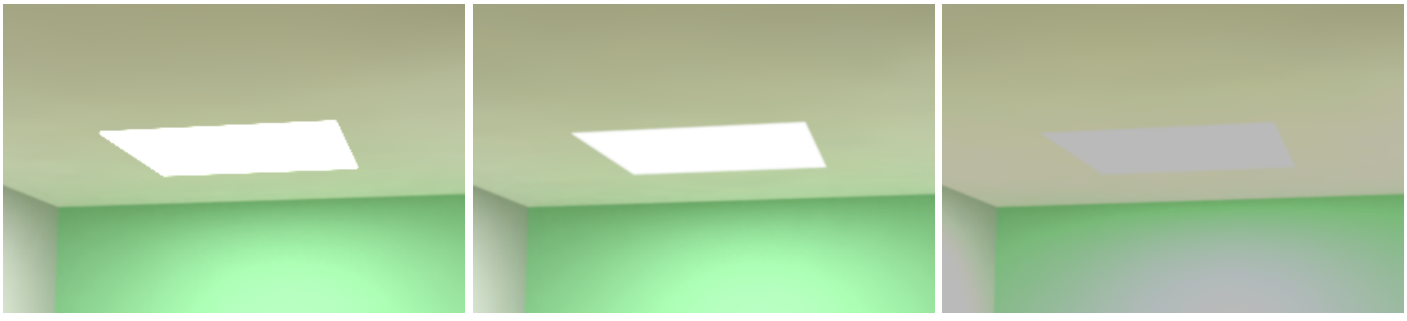
Example:

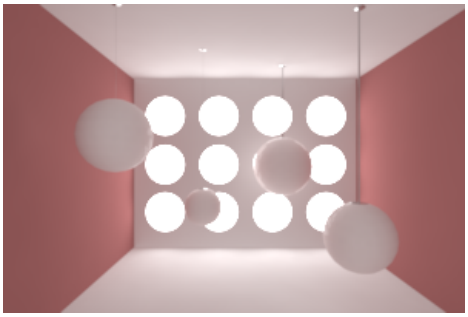
```
var __undersample_edge,1; // default value
```

Tailer commands

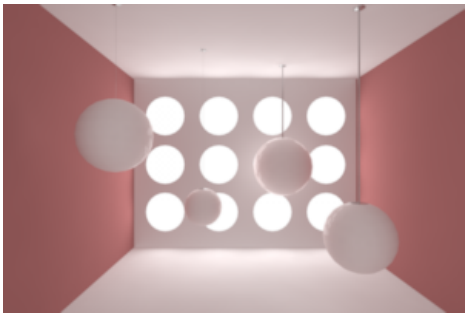
limitdr *value*

Limits dynamic range of primary rays (those fired from camera). If a RGB component of a camera ray is bigger then *value* it is truncated. This have importance with antialiasing overblown parts of the image. Luminosity of the white areas on the image is much higher then 100%.

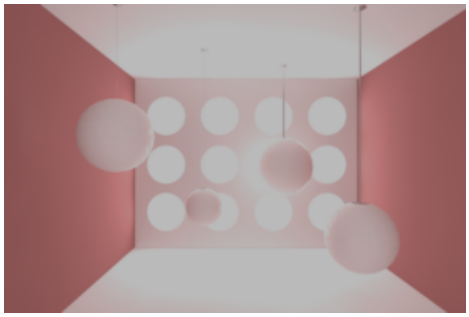




limitdr 0;
no dynamic range limit
aliasing visible



limitdr 1;
limitdr equals image exposure
aliasing reduced, but small bright details washed out



limitdr 0.5;
not very useful, just example

By default **limitdr** setting depends on output image format. If it is HDR **limitdr** is set to zero (no dynamic range limit). If output format is LDR (low dynamic range image format like PNG, TGA, BMP) **limitdr** is equal to output exposure. This improves antialiasing of overblown areas. Note that for LDR images, image dynamic range is always limited to output exposure. The only difference with **limitdr** is that this is done per camera ray, not per pixel.

recurse *ray recursion depth*

Sets ray recursion depth and overrides LightWave's setting. Allows to set higher recursion depth then avaiable in LW.

Example:

```
recurse 100;
```

Other commands

about

Displays info about rendering engine.

Example:

```
about;
```

cam_singleside *mode*

If *mode* = 1 camera rays intersects front side of polygon only. When *mode* = 0 camera rays intersects both sides of polygon.

Example:

```
cam_singleside 1; // default value
```

debug *debug_flags*

Displays debug infos in rendering log. *debug_flags* determines what information will be displayed.

debug_flags	**Meaning**
0	No debug info
1	Luminosity lights
2	Objects import
4	Rendering finalization
8	Log to file (Default filename is *kray_log.txt*. It can be changed with **logfile**)

You can sum *debug_flags*. For example *debug_flags* = 2+8 will show object import info and save render log to a file. By default *debug_flags* is set to 0.

Example:

```
debug 1+2+4+8;
```

echo *text*

Displays *text* on Kray console.

Example:

```
echo "This text will be displayed in Kray console.";
```

include *filename*

Allows to load set of Kray script commands from external file.

Example:

```
include "myfile.txt";
```

lmimportance *importance*

Controls how accurate direct light and caustics are computed during lightmap building. *importance* = 1 computes direct light and caustics with best quality (slowest). *importance* = 0 gives fastest, but most inaccurate result.

Example:

```
lmimportance 0.5;
lmimportance 0; // default value
```

logfile *filename*

Enables loging to a file and sets log *filename*. See **debug** for more loging options.

Example:

```
logfile 'render_log1.txt';
```

lwtexturesystem

Forces Kray to use LW's texturing system instead of converting LW textures into Kray textures. Off by default.

Example:

```
lwtexturesystem;
```

lwtransoptions sideness_mode,color

sideness_mode determines how transparency/refraction works.
sideness_mode = 0 transparency/refraction ray leaves object before next intersection
sideness_mode = 1 transparency/refraction ray intersects both sides of a polygon
sideness_mode = 2 transparency/refraction ray intersects front hit front side of polygon only

If *color* = 1 transparency and refraction amount is computed separately for r,g,b components if there is a refraction/transparency texture. *color* = 0 is LW compatible setting, transparency and refraction uses grayscaled refraction/transparency texture.

Example:

```
lwtransoptions 1,1; // default values
```

mipmap *filter_size_shift*,*additional_samples*,*minsamples*,*maxsamples*

Controls anisotropic filtering of textures (texture antialiasing).

With *filter_size_shift* you can control texture blurring. Negative values decreases blurring, positive increases. Typical values are between -1 and 1. Values < -1 can cause texture aliasing. Values > 1 can look too blurred.

additional_samples,*minsamples* and *maxsamples* controls number of samples Kray uses for anisotropic filtering. More samples - slower rendering, but better quality of texture filtering. Kray computes number of samples from texture anisotropy level and than adds *additional_samples* to this value. *minsamples*,*maxsamples* are minimum and maximum limits.

Example:

```
mipmap 0.2,4,5,30;
mipmap 0,0,0,30; // default values
```

octstats

Displays information about octree used for rays intersection optimization. Information for advanced users and debugging purposes.

Example:

```
octstats;
```

octsmallcube *ratio*

Additional parameter for octree generation. It is a *ratio* between size of polygon and smallest octree leaf. For advanced users.

Example:

```
octsmallcube 2; // default value
```

photons_singleside

Enables front side intersection only for photons fired from light sources. By default photons hits both sides of polygon.

Example:

```
photons_singleside;
```

postprocess

Adds post processing filter to Kray's native post processing system.

Note: Some examples are multiline. You can use multiline syntax in Kray script files (see include command), or write a command in single line and directly place in Header/Tailer field.

postprocess add,(*r*,*g*,*b*)

Adds *r*,*g*,*b* values to every pixel of renderd image.

Example:

```
postprocess add,(0.5,0.5,0.5);
```

postprocess autoexp,*strength*

Compensates exposure of an image. *Strength* is a value between 0 and 1. *Strength*=0 no compensation. *Strength*=1 full compensation.

Example:

```
postprocess autoexp,1;
```

postprocess filter3

3x3 convolution filter.

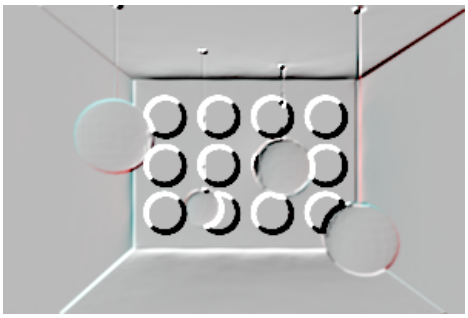
Example 1:

```
postprocess filter3,
-1,-1,0,
-1,0 ,1,
0 ,1 ,1;
```

Example 2:

```
limitdr 0;
postprocess filter3,-1,-1,0,-1,0,1,0,1,1;
postprocess add,(0.5,0.5,0.5);
```

Result:



postprocess filter5

5x5 convolution filter.

Example:

```
postprocess filter5,
0 ,-1,-1,0 ,0,
-1,-1,-1,0 ,0,
-1,-1,0 ,1 ,1,
0 ,0 ,1 ,1 ,1,
0 ,0 ,1 ,1 ,0;
```

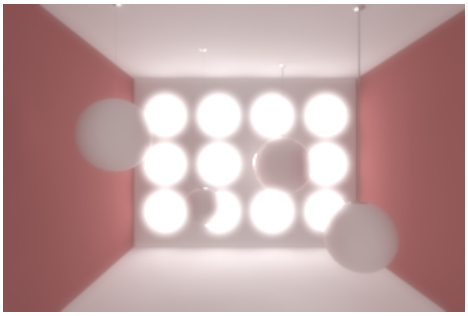
postprocess gblur,*blur radius*,*blend*

Gaussian blur blended with original image. *Blur radius* controls amount of gaussian blurring. *Blend* is a balance between original image and blurred image. *Blend*=0 , only original image (filter does nothing). *Blend*=1 , only bluring, no blending with original. *Blend*=0.5 , 50% blurred image 50% original image. This filter works nice with overblown areas on the images. See example below and compare it with **limitdr** examples.

Example:

```
limitdr 0;postprocess gblur,15,0.4;
```

Result:



postprocess mult,(*r*,*g*,*b*)

Multiplies every pixel of the reneder image by *r*,*g*,*b* values.

Example:

```
postprocess mult,(0.5,0.5,0.5);
```

rem *comment*

Comment in Kray script. Not very useful in **Header commands** and **Tailer commands** fields, but can be handy in Kray script files (see **include**).

Example:

```
rem This is comment. Kray will ignore it.
```

renderinfo *info text*

Adds information bar to rendered image. Info text can contain following symbols:

Symbol	Meaning
%kray	Kray logo.
%ver	Kray version number.
%time	Rendering time in ***hours*:*minutes*:*seconds*.*centiseconds*** format.
%days	Render time - days.
%hours	Render time - hours.
%mins	Render time - minutes.
%secs	Render time - seconds.
%centisecs	Render time - 1/10 seconds.
%sectime	Total render time in seconds.
%file	Output image filename
%width	Image width.
%height	Image height.

Example:

```
renderinfo "%kray %ver | Rendering time : %time";
```

Adds bar like this:

renderinfosize *text scale*,*border size in pixels*

Scales render info font size and sets border size.

Example:

```
renderinfosize 0.66,22; // default setting
```

showcornersamples (*r*,*g*,*b*)

Shows irradiance samples in corners with desired color (those who matches **Thin geometry distance**). Works only if irradiance caching is on.

Example:

```
showcornersamples (10,0,0);
```

showphstats

Shows photon statistics.

Example:

```
showphstats;
```

smoothprecachescale *ratio*

Sets the *ratio* between **precached** and **precached filtered** search range. **Precached filtered** interpolates between neighbour irradiance samples and require biger search range (**precached filtered** needs at least 2 samples when for **precached** one sample is enough). High values gives better filtering, but needs more computing power.

Example:

```
smoothprecachescale 1.2; // default value
```

surface_flags *flags*

Set flags for diffuse surfaces. Flags allow to enable/disable direct lighting computation and texturing (and output irradiance).

flags	**Meaning**
1	Enables direct lighting computation
2	Enables textures

Surface flags can be added. *flags* = 1+2 means that both direct lights and textures must be computed (default setting). Disabling textures and direct lighting is very useful combined with texture baker. This allows to bake irradiance maps in low resolution without loosing details of texturing and shadows.

Example:

```
surface_flags 1+2; // default value
```