ORB.S : Open Rendering Benchmark Spec

Ray Tracing Systems

June 17, 2024

Abstract

The standard feature number definition consists of 3 parts. They are material, lights and geometry.

1 How to read and use spec

The specification consists of a list of requirements (see below) which is separated on several groups: (G for geometry, M for materials, L for lights and A for algorithms).

The core requirements assumed to be implemented/fulfilled by renderer developer in order for each group. Thus to determine a finite set of features renderer support you have to specify 3 numbers in subsequent order. For example: «G1/M3/L4» means «1» version for geometry, «3» version for materials and «4» for lights. If such requirement can't be fulfilled, developer may include or exclude specific requirements as i further examples: «G1/M3/L7-(2,3)» which means req. 1,4,5,6,7 are fulfilled and 2,3 are not fulfilled for lights; «G1/M3/L7+(11-13, 15-18)» which means 1-7,11,12,13,15,16,17,18 are fulfilled for lights.

The specification levels are applied either to renderer and scene files. Therefore please read further note.

(!)Note: Requirements are written for render, but they should be applied to 3D content exporter/converter in a kind of an opposite bidirectional way:

1. If rendered **should/must** support some feature, exporter may assume that renderer properly implement this feature and thus may use it during export.

2. If rendered **may assume** some constraint (for example, only uniform scale for instances of spherical lights), exporter **should/must** be sure that such constraint is fulfilled in scene or report error.

3. If rendered **should/must** support some feature only on some concrete level of specification (for example triangle meshes "G2"), exporter must **prepare data** for this level of specification in the way which is described by this specification level and report error if

this is not possible. For example, if scene made with instances, which is not supported by level “G2”, exporter must duplicate all instanced geometry and merge it to single mesh.

Therefore, complex data processing like format loading/conversation, normal and tangent space calculation, explicit making meshes from analytical shapes and e.t.c ... usually is not the renderer’s responsibility. The exporter must do dirty job in the case if render can’t support some feature directly but if it can nevertheless be transformed to supported set of features.

2 Geometry

General requirenments:

(!)REQ: Renderer should not do any geometry preprocessing (such as normals recalculation, vertex splitting, etc.) and use provided data as it is. This is valid for all specification levels unless otherwise specified.

REQ_G1: primitive_format (core)

The geometry should be represented as triangulated indexed meshes. Indices must be specified with uniform winding order for the whole scene - only clockwise or only counter-clockwise. Triangle vertices **must** have the following attributes specified:

- positions in 3d space;
- normals;
- tangents (can be set to zero if not available);
- texture coordinates (can be set to zero if not available).

For each triangle of this mesh some material is assigned — that is, per triangle material is specified.

REQ_G2: single_mesh (core)

At this specification level, the whole scene is defined as a single mesh or several meshes with transformed vertices in world space. All positions of vertices must me specified in meters.

There are five parameters that should be specified for drawing model: (fovY, width, height, zNear, zFar). If they are not specified, renderer may use default values:

- fovY – vertical field of view in degrees, default value is 90.0.
- width – image width in pixels, default value is 1024.
- height – image height in pixels, default value is 1024.
- zNear – near plane in meters, default value is 0.1.
- zFar – far plane in meters, default value is 100.0.

The projection matrix should be obtained from (left,right,bottom,top) in the way OpenGL standart specifies for glFrustum according to further calculations:

$$\begin{aligned} aspect &= \frac{width}{height} \\ ymax &= zNear * tan(fovY * \frac{\pi}{360.0}) \\ xmax &= ymax * aspect \\ left &= -xmax, right = +xmax, bottom = -ymax, top = +ymax \end{aligned}$$

Reference implementation can be found here (perspectiveMatrix)

The camera should be specified in the following way:

- camera position : point in 3D.
- camera lookAt : point in 3D.
- camera upVector : point in 3D, default value is (0,1,0).
- camera distance : scalar, represent distance to focal plane.

The modelView matrix should be calculated from (position,lookAt,upVector) according to gluLookAt specification. Reference implementation can be found here (lookAt).

REQ_G3: inst_matrices (core)

At this specification level, the whole scene is defined as set of instances of meshes. Each mesh instance has its own transform 4x4 matrix. Matrices can contain rotation, translation, scaling (both linear and non-linear). If the mesh was not instanced at all, it should not be drawn.

Test: test_305_instancing.

Figure 1: Example of instancing.

REQ_G4: inst_change_material (core)

This requirement defines the ability to instantiate same objects with different materials. However, only single material for the whole mesh (and thus fo the whole instance) is allowed. Though, rasterization based traditional renderers assume to change their shaders once per instance and the ray tracing based solutions should substitute single material id for each instance.

(!)Note: In other words, each mesh has strictly 1 material, and if any of them has 2 different materials, the exporter must divide this mesh into two different meshes.

(!)Note: Here we can see a unobvious thing for content exporter developers. On both previous and subsequent levels of specification for a single mesh it is allowed to have several materials. But current specification level directly forbids this: export/preparation software must split meshes to have only single material per mesh. It is important to emphasize that there is no contradiction here, because our specification is created for renderers and thus exporters must adapt to this. For render systems without full instancing support (geom: **??**, 3) in general there is no difference if object was spitted to several meshes or not, as well remap list (see next spec. level) size may be equal to 1 when we go to subsequent levels.

REQ_G5: inst_remaplist_(MaxMaterials == 4) (core)

This requirement defines the ability to instantiate same objects with different materials. That is, different instances of the same object can have different materials. For this, we define such a thing as «remap list». The remap list is a list of pairs of material indices. In each pair, the first index defines the material to be replaced, and the second index determines the final material. For example, consider further remap lists (listing 1):

```
<remap_lists>
  <remap_list id="0" size="4" val="0 5 3 1 " />
  <remap_list id="1" size="8" val="1 0 2 0 3 0 4 0 " />
</remap_lists>
```

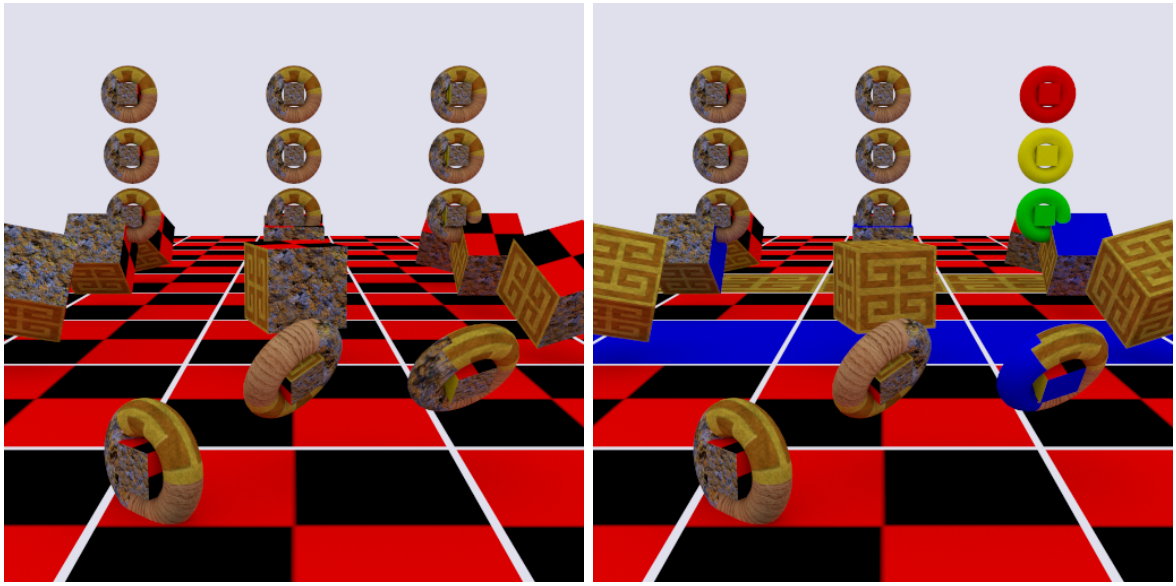
Listing 1: Remap list example

The remap list with id «0» will replace material 0 to 5 and material 3 to 1. The remap list with id «1» will replace all its input materials (1,2,3,4) to 0.

So, each instance can assign an existing remap list. If this happends, for this instance on the displayed model the replacements specified in remap list should be applied (fig. 5).

Test: test78_material_remap_list1.

(!)Note: For efficiency reasons renderers may restrict remap list size (MaxMaterials == 4) in several specific cases. This is essential to objects with opacity maps (leaves, grass and e.t.c). If remapped material has opacity property, during ray-triangle intersection we have to look up in the opacity texture. However, material id remapping in general case is implemented via binary search which is too expensive if performed for each triangle. In this way renderer may restrict remap list size to 4 or 8 elements (and thus 2 or 4 materials, for example) to be sure that remapping will be cheap enough. Please note that default maximum material amount in remap list is equal to 4 and thus exporter must split single mesh to several meshes for all instances that have greater than 4 materials in their remap list if original or remapped material has opacity.



Original material mapping Appliyng remap list

Figure 2: Remap list example.

2 Materials

Image 3 and equation 1 show lighting integral and related angles. These notation will be used further in the document. They assume reverse (i.e. Path Tracing) direction of light by default:

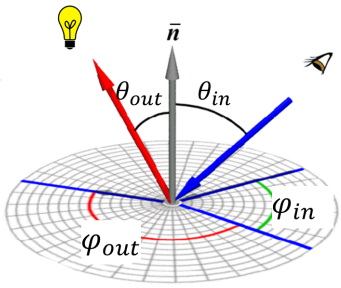


Figure 3: BRDF and angles.

$$I(\phi_{in}, \theta_{in}) = \int_{\phi_{out}} \int_{\theta_{out}} L(\phi_{out}, \theta_{out}) BRDF(\phi_{out}, \theta_{out}, \phi_{in}, \theta_{in}) cos(\theta_{out}) \mathrm{d}\phi_{out} \mathrm{d}\theta_{out} \tag{1}$$

It should be noted that the adopted agreement of the backward propagation of light (from eye to light) does not limit anything and can be applied also for calculation of forward propagation (from light to eye).

(!) Implementation REQ. : All BRDF and BTDF should be implemented without $cos(\theta_{out})$ term. For mirrors cosine shouldn't be applied, thus you should pre-divide $cos(\theta_{out})$ instead (so that the cosines cancel each other). Also note that we do not include color in BRDF/BTDF equations. The color should be multiplied with the BRDF value.

(!) Implementation REQ. : Note that final color of the image should be first **clamped** (i.e. (2,1.5,1) is clamped to (1,1,1) which in fact leads to color loss) and then transformed from HDR to LDR space with gamma correction according to [1].

$$C_{LDR} = \begin{cases} 12.92 \cdot C_{HDR}, & \text{if } C_{HDR} \leq 0.0031308 \\ 1.055 \cdot {C_{HDR}}^{1/2.4} - 0.055, & \text{if } C_{HDR} > 0.0031308 \end{cases}$$

where C_{LDR} in color value in LDR space and C_{HDR} is color value in HDR space.

(!) Implementation REQ. : Note that texture color (that comes from PNG, JPG, BMP and othe LDR formats) should be transformed to linear space with gamma transform according to [1]. Unlike textures, values that are **specified explicitly** in XML (or other) assumed to be **already in linear space**.

$$C_{HDR} = \begin{cases} \frac{C_{LDR}}{12.92}, & \text{if } C_{LDR} \leq 0.04045 \\ \left(\frac{C_{LDR}+0.055}{1.055}\right)^{2.4}, & \text{if } C_{LDR} > 0.04045 \end{cases}$$

(!) Implementation REQ. : For all tests PSNR value between renderer and reference should be greater than 35 for 512x512 LDR image if otherwise is not specified.

(!) Symbols:

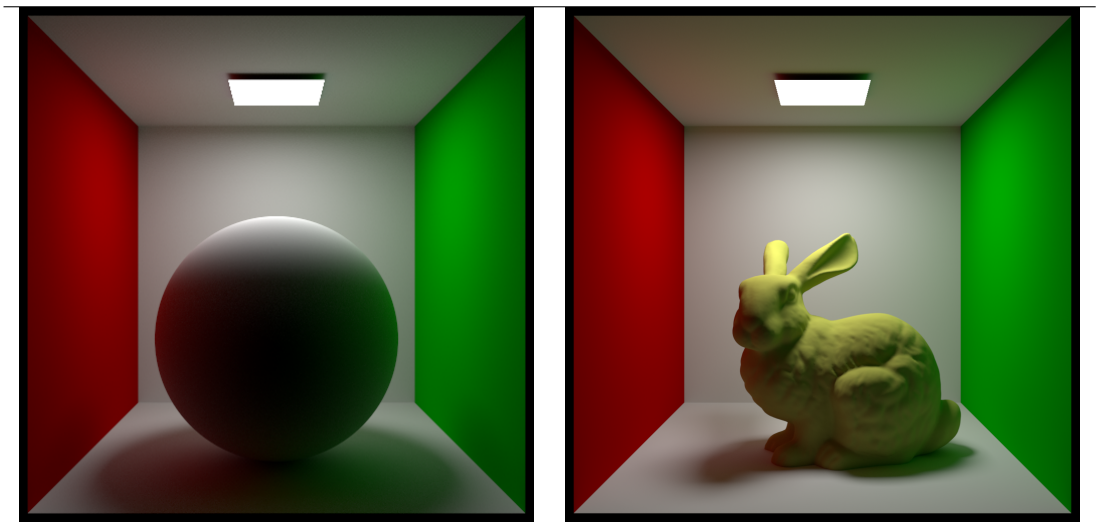
- $\omega_i = v$: in, or view vector (note that on image 3 inverse view vector (i.e. $-v$) is drawn with blue arrow);
- $\omega_o = l$: out, or light vector;
- $\omega_h = \frac{l+v}{2}$, half vector;
- n, normal vector.

REQ_M1: mat_lambert (core)

Basic support for Lambertian reflections. The BRDF function **must** be implemented according to equation 2. See fig 1.

Implementation:

$$BRDF_{lambert} = \frac{1}{\pi} \tag{2}$$



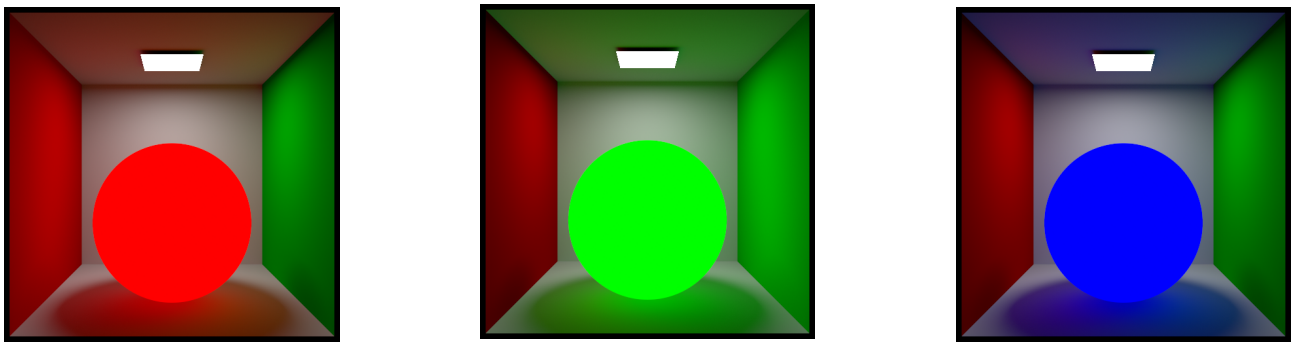
TestHy: 101_diffuse_lambert **TestMi:** Lambert

Table 1: Example of lambertian material. The grey ball on left image have color equal to (0.5, 0.5, 0.5). The yeallow rabbit on the right image have color equal to (0.5, 0.5, 0.1).

REQ_M2: mat_emission (core)

Basic support for emissive material. On this level of specification emissive materials don't combined with others. Renderer may suppose mutually exclusive: (emissive or 'some other'). Please note that emissive materials have not be considered as lights (i.e. they don't have to be sampled with any explicit sampling strategy, however, of cource, they could), but they must cast GI.

Test: 123_emission



red emission green emission blue emission

Figure 4: Phong for different cosine power and color (0.5, 0.5, 0.5).

(!) Implementation REQ. : Emission color value **must** specify **brightness** value. It is easy to ckeck this requirenment. Directly visible emissive surface on rendered image (stored in linear HDR framebuffer) must have exactly same value that was set for material. For example if you set emission color to (0.5, 0.5, 0) and then check directly visible surfaces (when primary

rays immediately hit surface), you shoulg get exactly same yellow color (0.5, 0.5, 0) in the rendered image (in linear space, i.e. HDR image).

REQ_M3: perfect_mirror (core)

Pure specular reflection. Note that mixing of mirror and lambertian reflections is not needed at this level of specification. Renderer may suppose mutually exclusive: (phong, mirror or lambert). **Implementation:** It is assumed that you will pre-divide to $cos(\theta_{out})$ inside BRDF and later cancel this cosine in lighting integral (equation 1) instead of considering pure specular reflection as a special case during lighting integral evaluation. Though, you will probably have to consider pure specular reflection as special cases in many other places.

$$BRDF_{mirror} = \frac{1}{cos(\theta_{out})} \tag{3}$$

(!) Implementation Note: You are **allowed** to clamp denominator to avoid dividing by zero:

$$BRDF_{mirror} = \frac{1}{\max(cos(\theta_{out}), 10^{-6})} \tag{4}$$

Test: test_102_mirror_orbspec_mat03.

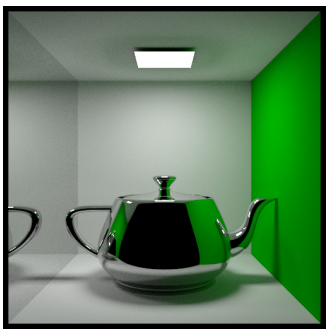


Figure 5: Example of mirror material with color equal to (0.75, 0.75, 0.75).

REQ_M4: lambert_texture (core)

Adding LDR textures for Lambertian materials. When reading from LDR texture, gamma transform **should** be applied. The resulting color must be the product of texture color and material color. Texture matrices support is not needed on this level of specification.

Test: test_103_diffuse_texture.

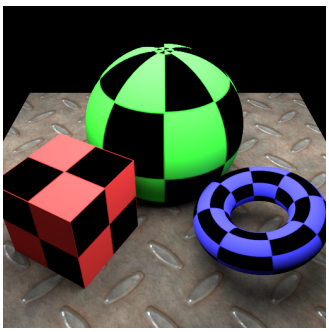


Figure 6: Example of lambertian material with textures.

REQ_M5: texture_matrices (core)

The sampler object must support the storage and use of texture matrices. Texture matrices are aplied to texture coordinates before actual texture fetch is performed. Initially, the matrices should be 4x4, but since in most cases the texture coordinates are two-dimensional, at this level of the specification, the render can save only the first 2 rows of the matrix.

Test: 110_texture_sampler

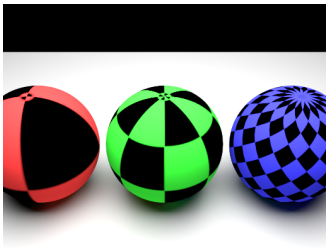


Figure 7: Test for texture matrices.

REQ_M6: emission_texture_ldr (core)

Adding LDR textures to emission color. When reading from LDR texture, gamma transform **should** be applied. The resulting color must be the product of texture color and material color. Texture matrices support is not needed on this level of specification.

Test: test_124_emission_texture.

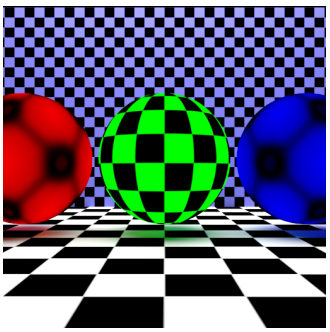


Figure 8: Emissive textures.

REQ_M7: mat_diffuse_Oren-Nayar (core)

Basic support for Oren-Nayar reflections. The BRDF function **must** be implemented according to equation 5. See fig 9.

Implementation:

$$BRDF_{oren-nayar} = \frac{\rho}{\pi}(A + Bmax(0, cos(\theta_i - \theta_r))sin(\alpha)tan(\beta)) \tag{5}$$

where ρ is the Lambert diffusion of the surface, $A = 1 - 0.5\frac{\sigma^2}{\sigma^2+0.33}$, $B = 0.45\frac{\sigma^2}{\sigma^2+0.09}$, $\alpha = max(\theta_i, \theta_r)$, $\beta = min(\theta_i, \theta_r)$, θ_i and θ_r are the angles between the normal and the directions to the light source and camera, respectively, and σ is the surface roughness parameter.

Test: test_100_diffuse_orennayar.

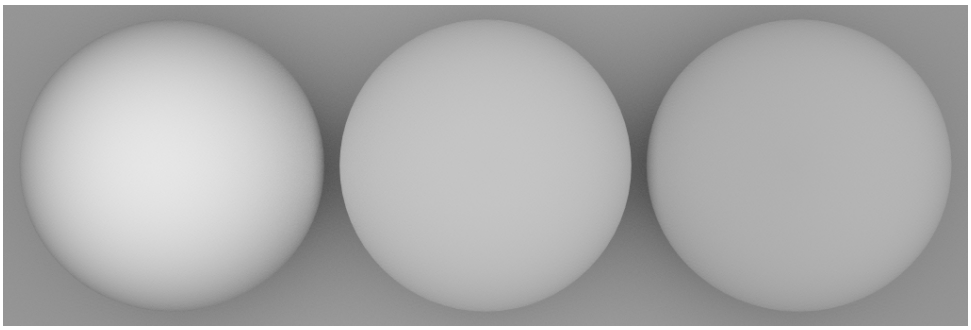


Figure 9: Diffuse Oren-Nayar BRDF.

REQ_M8: mat_smooth_glass (core)

Basic support for smooth glass. See fig 10.

Implementation: The ray is stochastically selected based on the Fresnel formula, determining the direction in reflection or refraction (eq. 6).

$$\text{dielectric} = \begin{cases} reflect, & \text{if } x \in [0,1] < fresnel \\ refract, & \text{otherwise} \end{cases} \tag{6}$$

where x is the random number.

The Fresnel function calculates the light reflection coefficient in the direction of the incoming ray, the normal to the surface and the refractive index (eq. 7).

$$\text{fresnel}(\vec{v}, \vec{n}, \text{ior}) = \begin{cases} 1, & \text{if } \sin_t > 1 \\ \frac{1}{2}\left(\left(\frac{\text{ior}\cos_i - \cos_t}{\text{ior}\cos_i + \cos_t}\right)^2 + \left(\frac{\cos_i - \text{ior}\cos_t}{\cos_i + \text{ior}\cos_t}\right)^2\right), & \text{otherwise} \end{cases} \tag{7}$$

where $\vec{v}$ is the vector from the light source, $\vec{n}$ is the normal to the surface, ior is the absolute refractive index, $\cos_i$ is the dot product of vectors $\vec{v}$ and the $\vec{n}$, $\sin_t = \frac{\sqrt{1-\cos_i^2}}{\text{ior}}$ and $\cos_t = \sqrt{1-\sin_t^2}$.

The reflect function calculates the direction of the reflected ray in the direction of the incoming ray and the normal to the surface (eq. 8).

$$\text{reflect}(\vec{dir}, \vec{n}) = \vec{dir} - 2(\vec{dir} \cdot \vec{n})\vec{n} \tag{8}$$

where $\vec{dir}$ is the vector from the light source, $\vec{n}$ is the normal to the surface.

The refraction function calculates the direction of the refracted ray in the direction of the incoming ray, the normal to the surface and the relative index of refraction (eq. 9).

$$\text{refract}(\vec{dir}, \vec{n}, \eta) = \begin{cases} \text{reflect}(\vec{dir}, \vec{n}), & \text{if } k < 0 \\ \eta'\vec{dir} - (\eta'\cos(\theta_i) + \sqrt{k})\vec{n}, & \text{otherwise} \end{cases} \tag{9}$$

where $\vec{dir}$ is the vector from the light source, $\vec{n}$ is the normal, η is the relative refractive index, η' is the inverted η, since the incoming vector and the normal are directed in the same direction. θ_i is the angle between the vector $\vec{dir}$ and the normal $\vec{n}$, and $k = 1 - \eta'^2(1-\cos^2(\theta_i))$.

(!) Implementation Note: In this implementation, in all formulas, the normal looks at the incoming ray.

Test: test_113_transparency_ior.

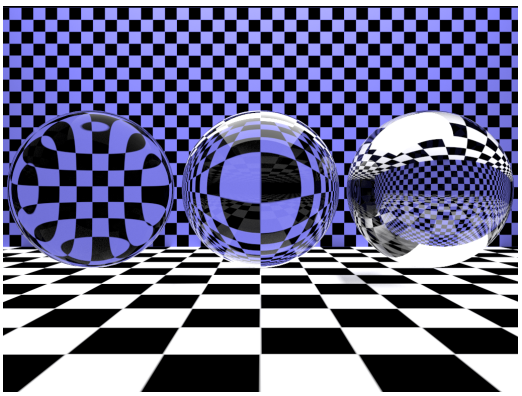


Figure 10: Smooth glass with different IOR.

4 Light

REQ_L1: area_light (core)

This level introduce basic support for area lights;

1. Area light should support two geometric form – "rect" (rectangle) and "disk".
   (a) If rectangular form is set, separate **half-width** (half size along 'X') axis and **half-length** (half size along 'Z') should be supported. **half-width** corresponds to the radius of disk inscribed in a square (when **half-width** is equal to **half-length**).
   (b) If rectangular form is set, disk **radius** should be set
2. The default light distribution is Lambert;
3. By default the light source is located at (0,0,0) and shine with direction (0,-1,0).
4. Renderer may support only translation for light instance matrix when light is instanced.
5. The intensity/brightness is a screen brightness (same implementation requirement as in 2).

TestHy: 101_diffuse_lambert

REQ_L2: sphere_light (core)

This level introduce basic support for spherical lights.

1. Renderer should support explicit change of light radius.
2. Renderer may support only translation for light instance matrix when light is instanced.

3. The intensity/brightness is a screen brightness (same implementation requirement as in 2).

Test: test_201_sphere.

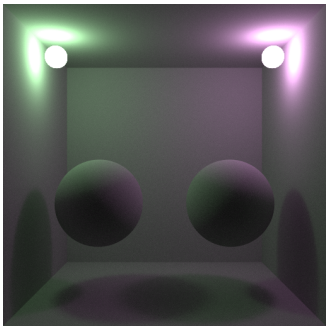


Figure 11: Spherical light source. Please note that bright areas (lights surface itself for example is clamped when transformed to LDR color space according to 2.)

REQ_L3: light_instance_matrices (core)

This level of spec add support for arbitrary matrices applied to lights. By default the light source is located at (0,0,0) and shine with direction (0,-1,0). Transform matrix may rotate, scale and translate light source. However, some assumptions can be made by renderer.

1. Renderer may assume that only uniform scale is applied to spherical lights.
2. When scale is applied to light source, its size, area and total flux changes (i.e. the whole scene becomes brighter or darker); Screen brightness does not changes.

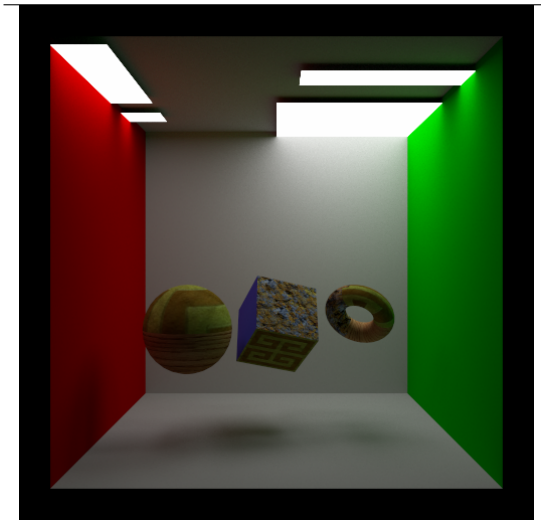
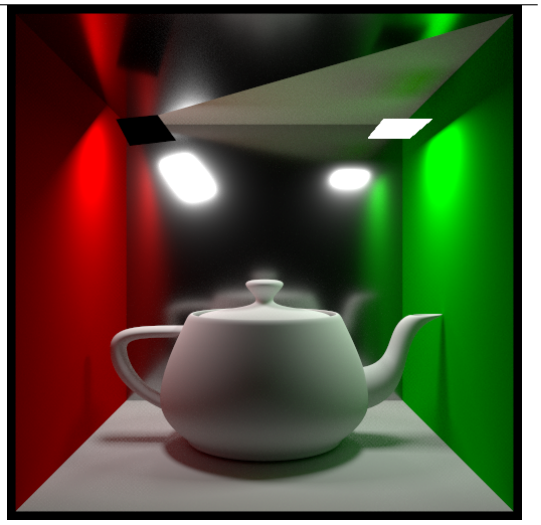
	
TestHy: test_215 (4 inst. of light)	**TestHy:** test_224 (2 inst. of light)

Table 2: Example of different matrices applied to lights.

REQ_L4: light_ies (core)

Simulation of goniograms (IES files) for light is accounted as if ray always hit center of light. I.e. point-area approximation (fig. 12, right).

(!)Note: Modeling light distribution using goniograms (IES files) for area light sources is **not a physically correct** way to set light distribution properties for at least two reasons: (1) IES distribution assumes light source is point; (2) IES distribution does not account that area light sources does not emit light to the side at 90 degrees. Therefore tests with IES distribution set for area lights have to account slight discrepancy in implicit and explicit sampling for area lights close to the edge of the light source (by masking such areas or decrease requirenment for PSNR).

Test: test_232_point_area.

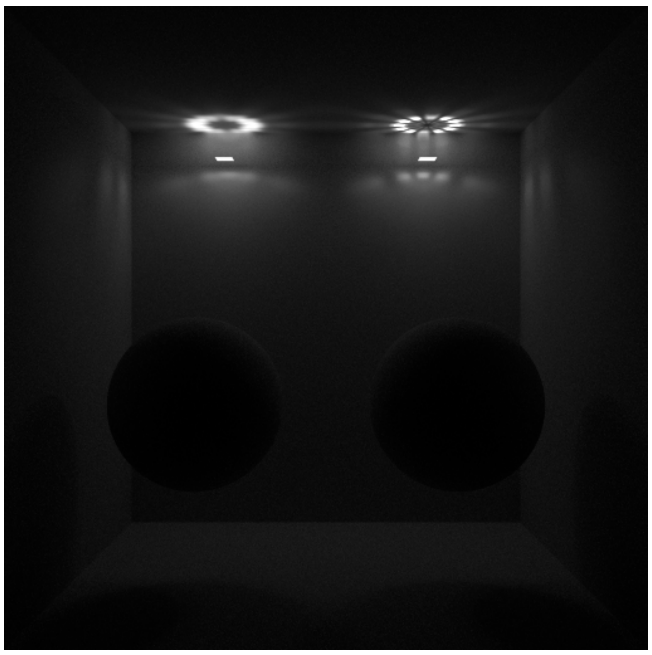


Figure 12: Left: a light surface with infinite number of points with IES distribution. Right: point-area approximation.

2 Integrators

REQ_A.1: trace_depth (core)

A tracing depth equal to 1 corresponds to the direct illumination.

REQ_A.2: paths_basic (core)

Scenes should be tested in which the following types of paths contribute significantly to the image:

1. ESDL, EDSL, ESDSL (R) — mirror wall that is also reflect single reflected caustic;
2. EGDL, EDGL, EGDGL (R) — glossy wall that is also reflect single reflected caustic;
3. ESSL, EDSSL — double perfect reflection.
4. EGGL, EDGGL — double glossy reflection.

REQ_A.3: paths_glass (core)

Scenes should be tested in which the following types of paths contribute significantly to the image:

1. EDSL, EDSL (T) — spherical light inside glass ball to check light transport symmetry;
2. ESDSL (T) — single refracted caustic to check light transport symmetry (water scene);
3. ESSL (T) — single refracted caustic to check light transport symmetry (glass ball);

REQ_A.4: paths_glossy_glass (core)

Scenes should be tested in which the following types of paths contribute significantly to the image:

1. EDGL, EDGL (T) — spherical light inside glass ball to check light transport symmetry;
2. EGDGL (T) — single refracted caustic to check light transport symmetry (water scene);
3. EGGL (T) — single refracted caustic to check light transport symmetry (glass ball);

References

[1] Jason Summers. A close look at the sRGB formula. 2012. URL=https://entropymine.com/imageworsener/srgbformula/

[2] Eric P. Lafortune and Yves D. Willems. *Using the Modified Phong Reflectance Model for Physically Based Rendering.* 1994.

[3] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. *Microfacet models for refraction through rough surfaces.* In Proceedings of the 18th Eurographics conference on Rendering Techniques (EGSR'07). 2007. Eurographics Association, Goslar, DEU, 195–206.

[4] Eric Heitz. *A Simpler and Exact Sampling Routine for the GGX Distribution of Visible Normals.* [Research Report] Unity Technologies. 2017. hal-01509746

[5] Joe Schutte. *Importance Sampling techniques for GGX with Smith Masking-Shadowing: Part 2.* March 7, 2018. Github pages, accessed 22.05.2020.