Writing mental ray shaders

by Andy Kopra

Part 3: Light

Lights

Lights

- Light from a single point
- A point light with shadows
- A simple spotlight
    - Acquiring light parameter values from the scene database
    - Using light parameter values in the shader
- A spotlight with a soft edge
- A spotlight with a better soft edge
- Soft shadows using area lights
    - Area light primitives
    - Using area lights in spotlights
- Modifying light intensity based on distance
- Defining good default values for parameters

Lights

Lights

Light element properties

Lights

- Light element properties
  - Determine the type of light

Lights

- Light element properties
  - Determine the type of light
    - Point, directional, spot, area

- Lights
  - Light element properties
    - Determine the type of light
      - Point, directional, spot, area
  - Light shader included in light element

- Lights
  - Light element properties
    - Determine the type of light
      - Point, directional, spot, area
  - Light shader included in light element
    - Determines the color produced by the light

- Lights
  - Light element properties
    - Determine the type of light
      - Point, directional, spot, area
  - Light shader included in light element
    - Determines the color produced by the light
- The material shader "samples" the light

- Lights
  - Light element properties
    - Determine the type of light
      - Point, directional, spot, area
  - Light shader included in light element
    - Determines the color produced by the light
- The material shader "samples" the light
  - The light shader runs to determine the color

- Lights
  - Light element properties
    - Determine the type of light
      - Point, directional, spot, area
  - Light shader included in light element
    - Determines the color produced by the light
  - The material shader "samples" the light
    - The light shader runs to determine the color
    - The light shader runs multiple times for an area light

Lights

Light from a single point

```
declare shader
    color "point_light" (
        color "light_color" default 1 1 1 )
end declare
```

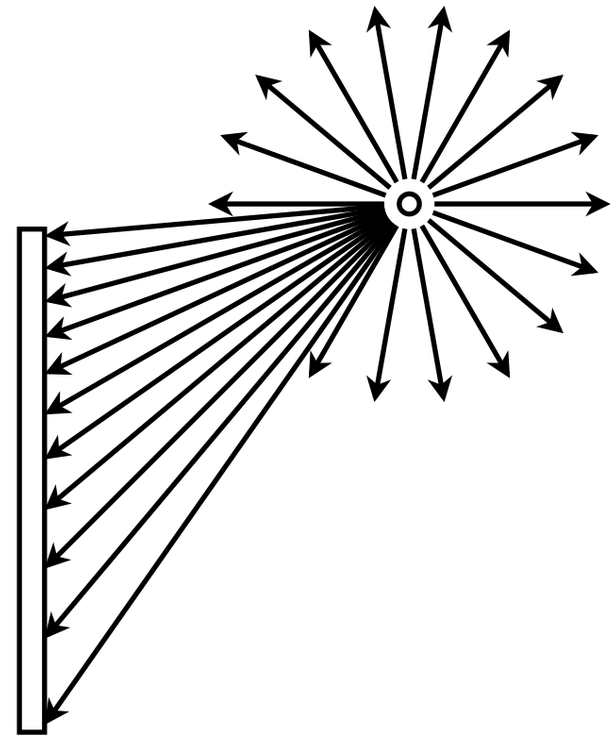
Scene file declaration of shader "point_light"

Light from a single point

```
 1  struct point_light {
 2      miColor light_color;
 3  };
 4
 5  miBoolean point_light (
 6      miColor *result, miState *state, struct point_light *params)
 7  {
 8      *result = *mi_eval_color(&params->light_color);
 9      return miTRUE;
10  }
```

Source code of shader "point_light"

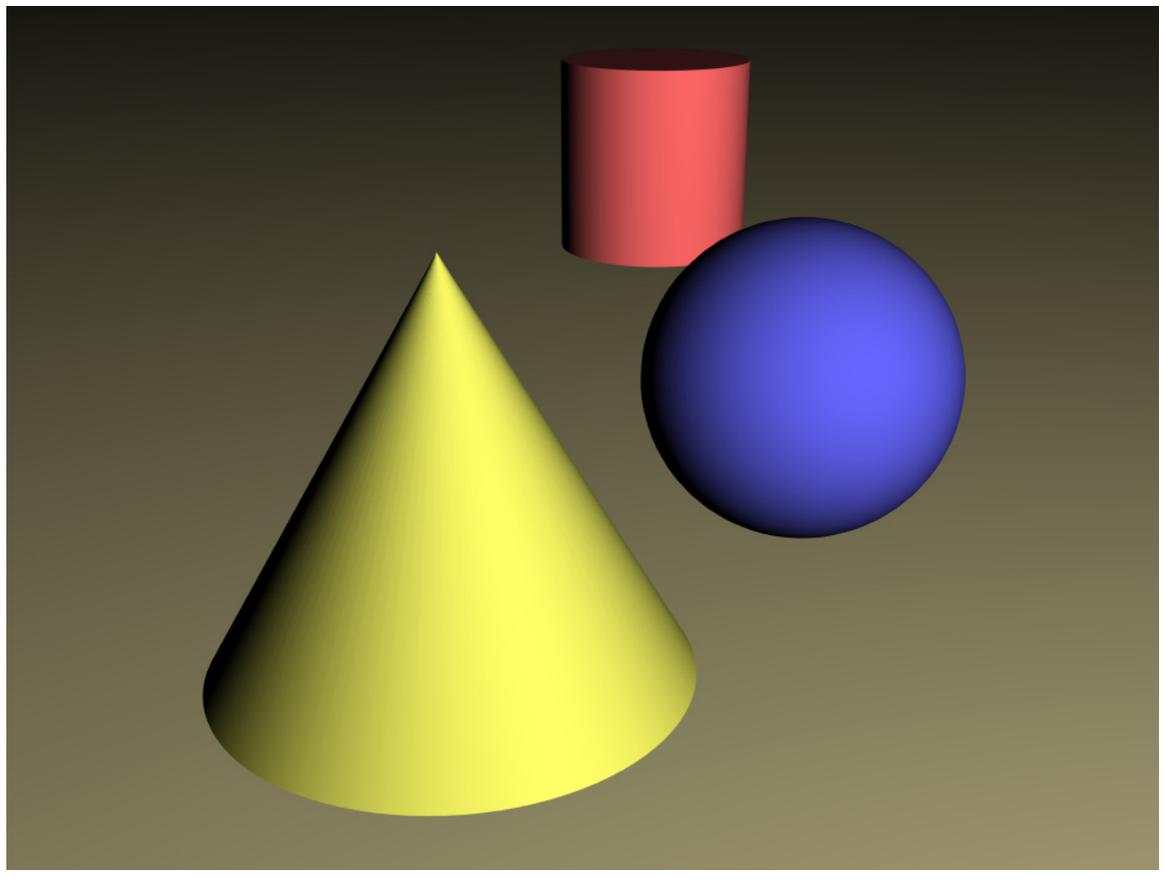
Light from a single point



A point light and a surface it illuminates

Lights

Light from a single point

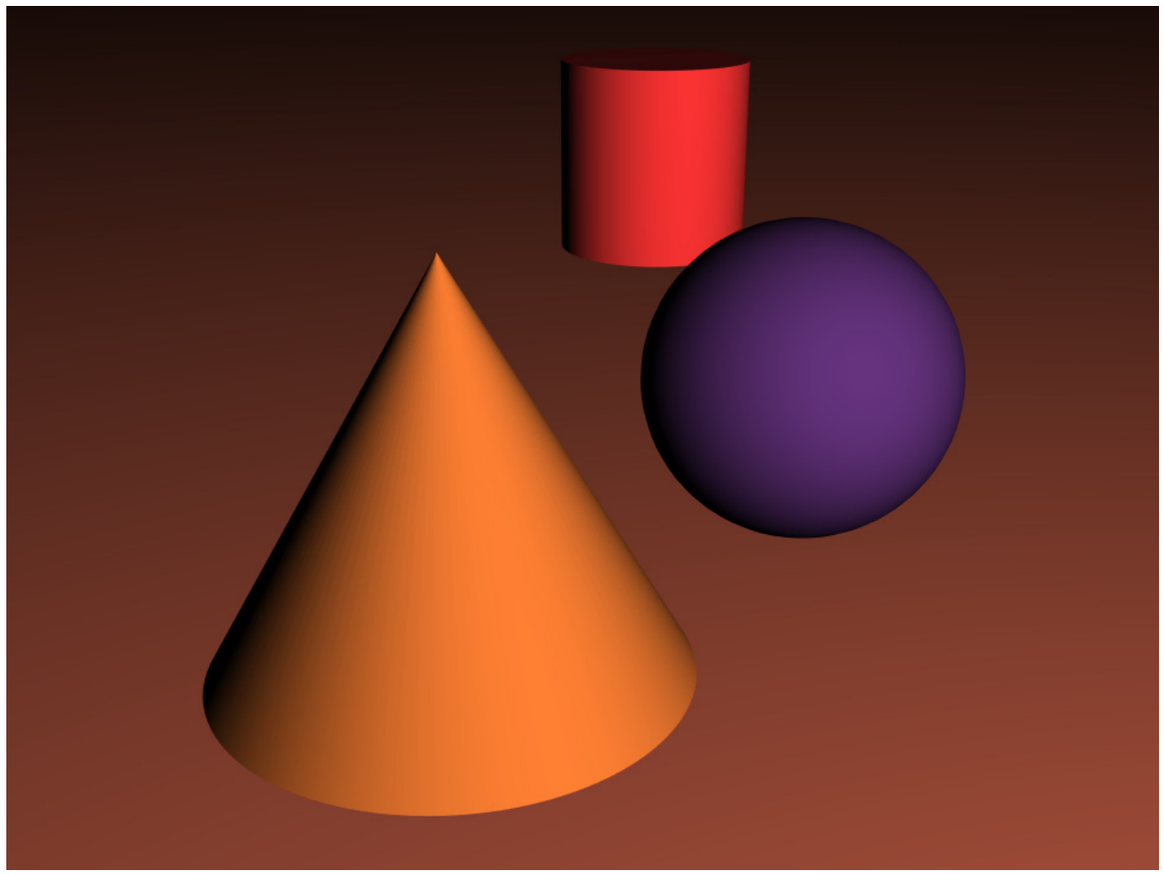


```
light "white_light"
    "point_light" ()
    origin 2 2 5
end light

instance "light_instance"
    "white_light"
end instance

material "yellow"
    "lambert" (
        "diffuse" 1 1 .4,
        "lights" ["light_instance"] )
end material

material "blue"
    "lambert" (
        "diffuse" .4 .4 1,
        "lights" ["light_instance"] )
end material

material "red"
    "lambert" (
        "diffuse" 1 .4 .4,
        "lights" ["light_instance"] )
end material
```

Point light with default white color

Lights

Light from a single point

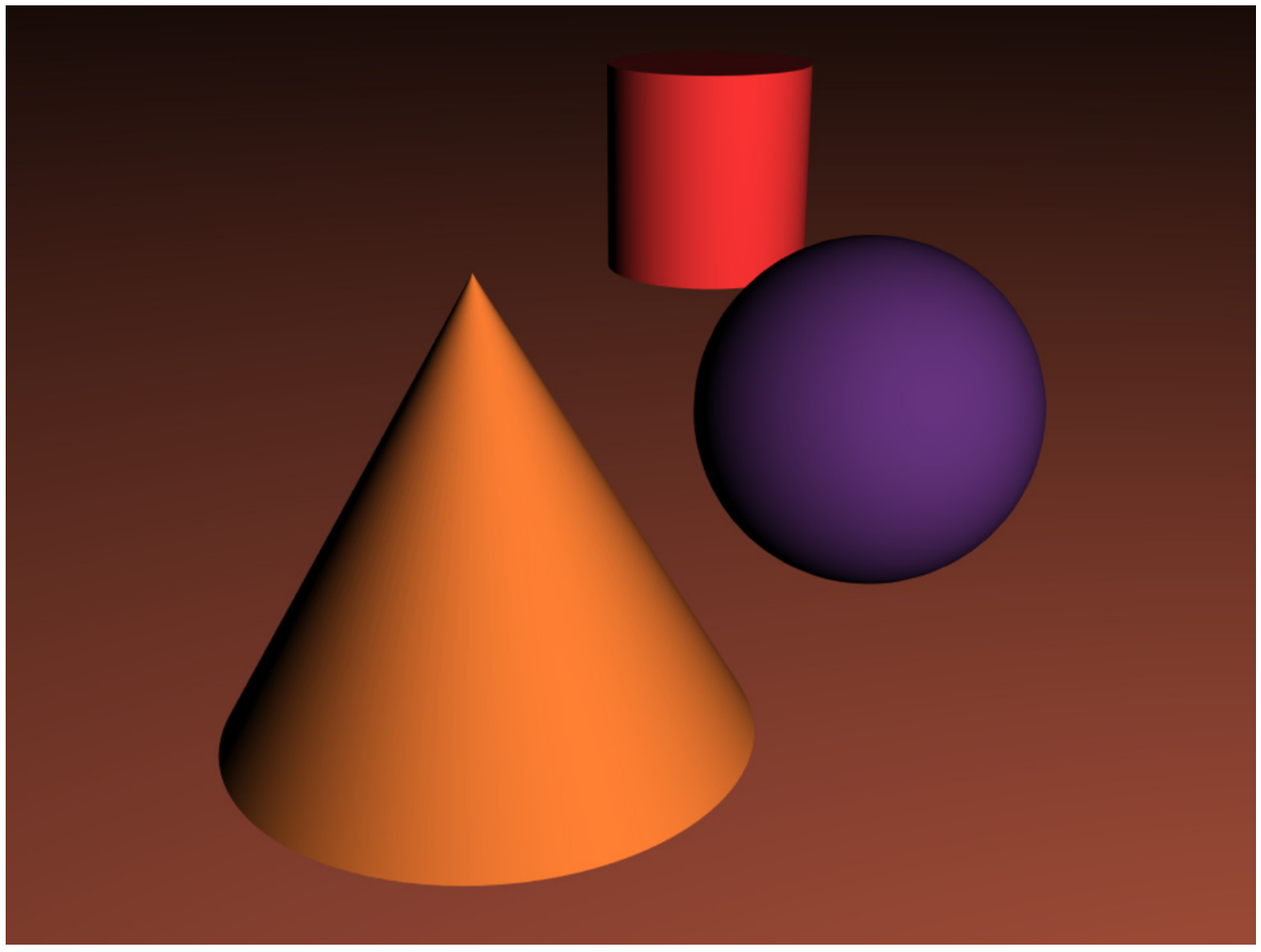


```
light "red_light"
    "point_light" (
        "light_color"  1 .5 .5, )
    origin 2 2 5
end light

instance "light_instance"
    "red_light"
end instance

material "yellow"
    "lambert" (
        "diffuse" 1 1 .4,
        "lights" ["light_instance"] )
end material

material "blue"
    "lambert" (
        "diffuse" .4 .4 1,
        "lights" ["light_instance"] )
end material

material "red"
    "lambert" (
        "diffuse" 1 .4 .4,
        "lights" ["light_instance"] )
end material
```

Point light using color parameter

Lights

Light from a single point



```
light "red_light"
    "one_color" (
        "color"  1 .5 .5, )
    origin 2 2 5
end light

instance "light_instance"
    "red_light"
end instance
```

Point light with one_color as the light shader

A point light with shadows

```
declare shader
    color "point_light_shadow" (
        color "light_color" default 1 1 1 )
end declare
```

Scene file declaration of shader "point_light_shadow"

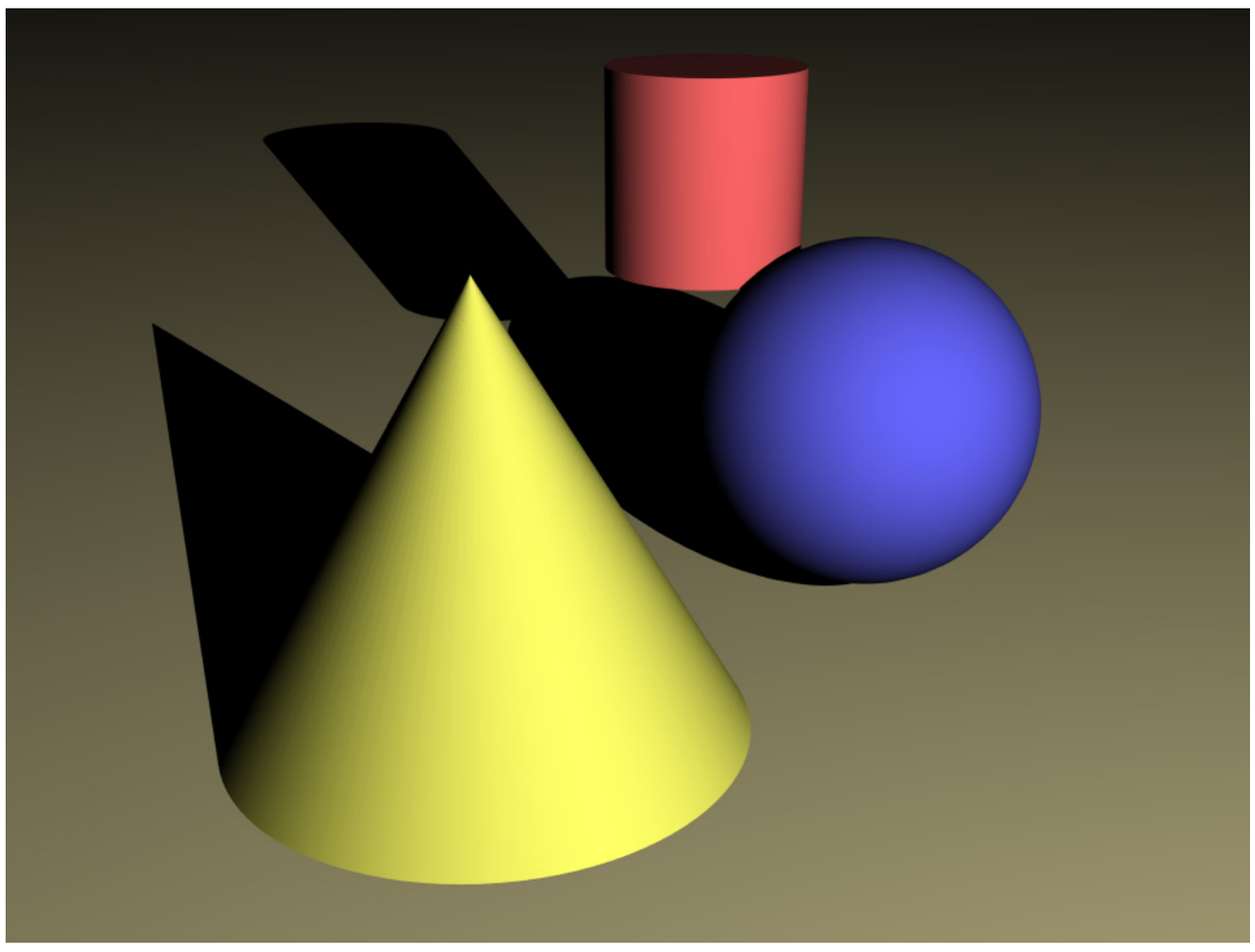
A point light with shadows

```
1  struct point_light_shadow {
2      miColor light_color;
3  };
4
5  miBoolean point_light_shadow (
6      miColor *result, miState *state, struct point_light_shadow *params)
7  {
8      *result = *mi_eval_color(&params->light_color);
9      return mi_trace_shadow(result, state);
10 }
```

Source code of shader "point_light_shadow"

Lights

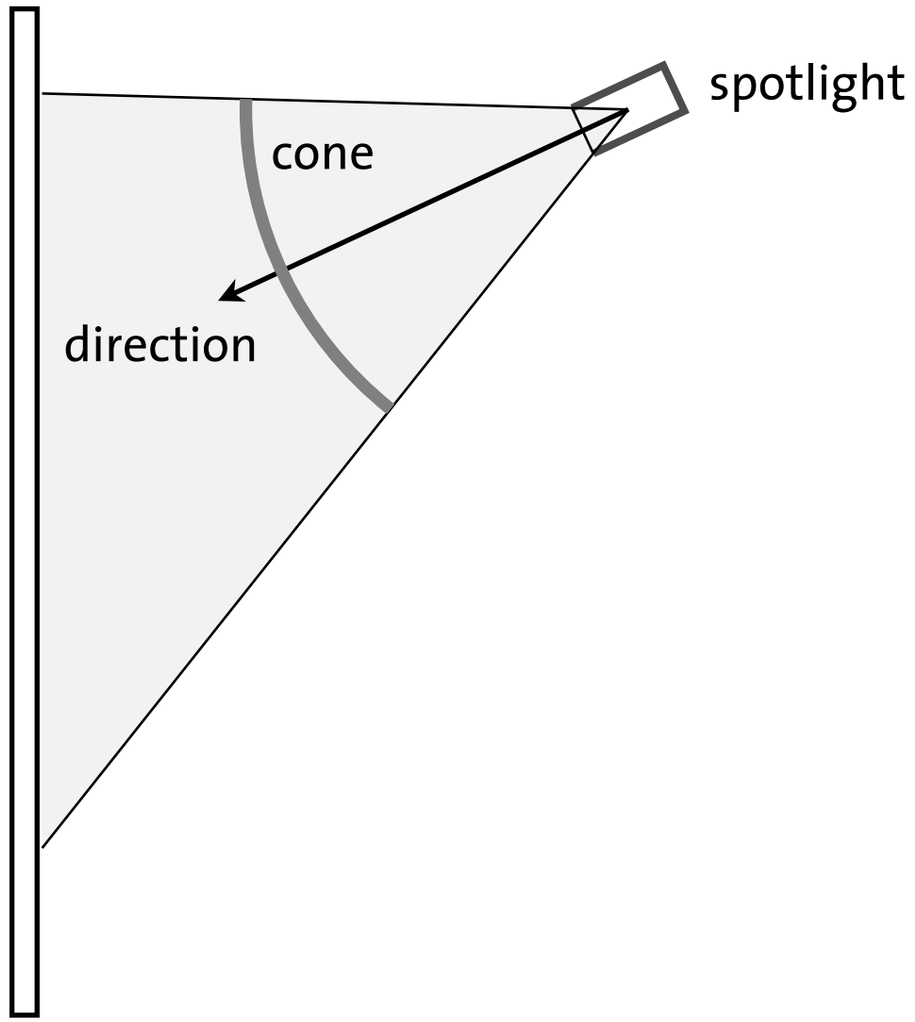
A point light with shadows



```
light "white_light"
    "point_light_shadow" ()
    origin 2 2 5
end light

instance "light_instance"
    "white_light"
end instance
```

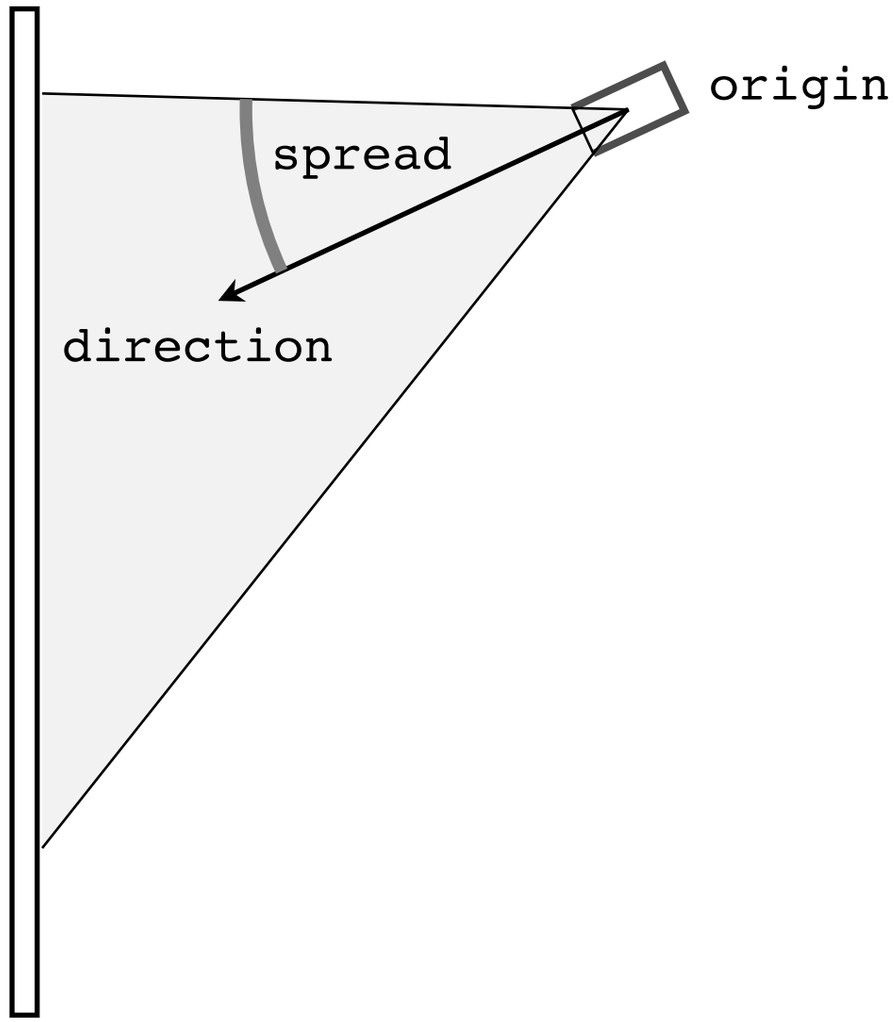
Point light with shadows

A simple spotlight

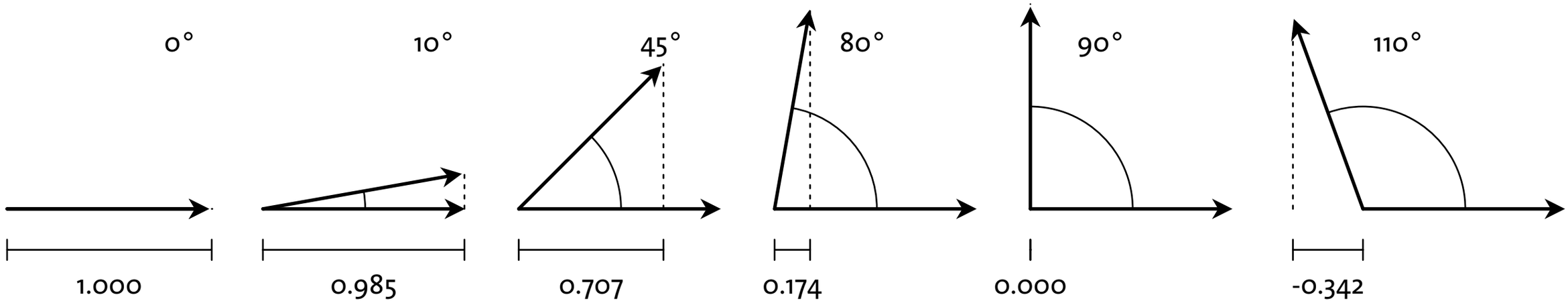


A spotlight with a given cone angle and light direction

A simple spotlight

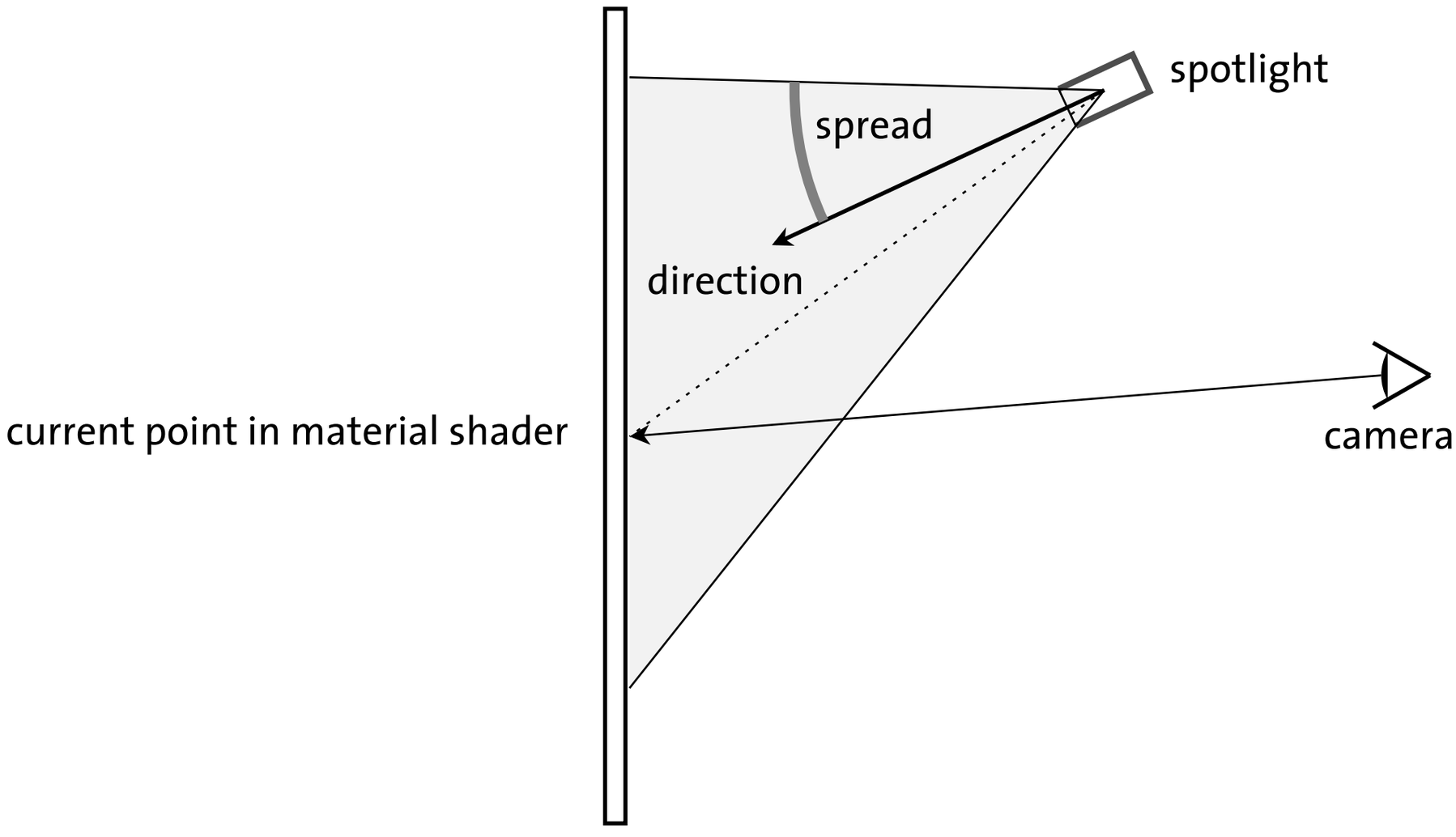


A spotlight labeled by the statements used in the light object



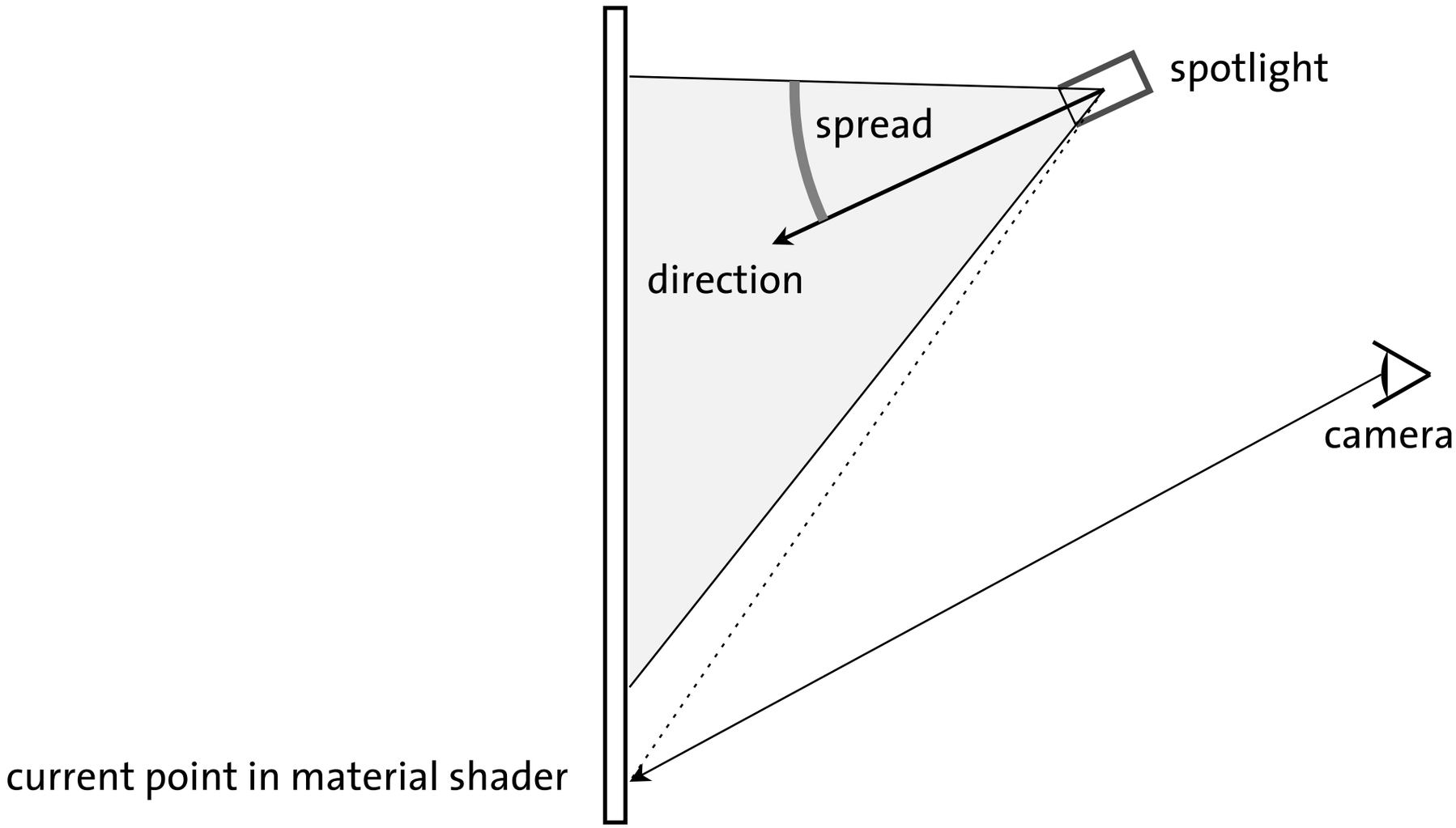
Two normalized vectors, the angle between them, and their dot product

A simple spotlight



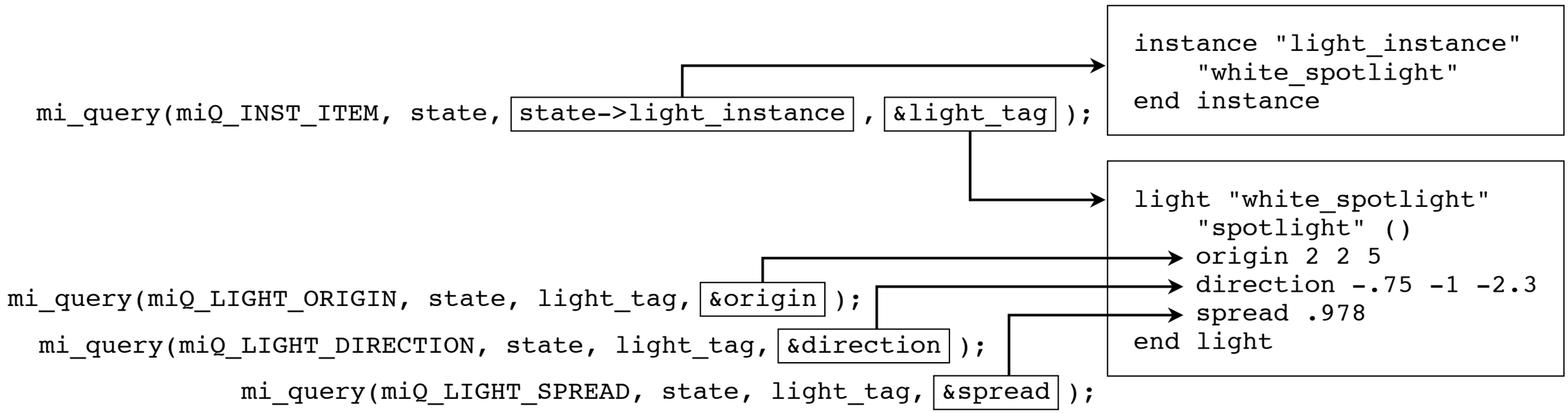
Sampling a point inside the cone of the spotlight

A simple spotlight



Sampling a point outside the cone of the spotlight

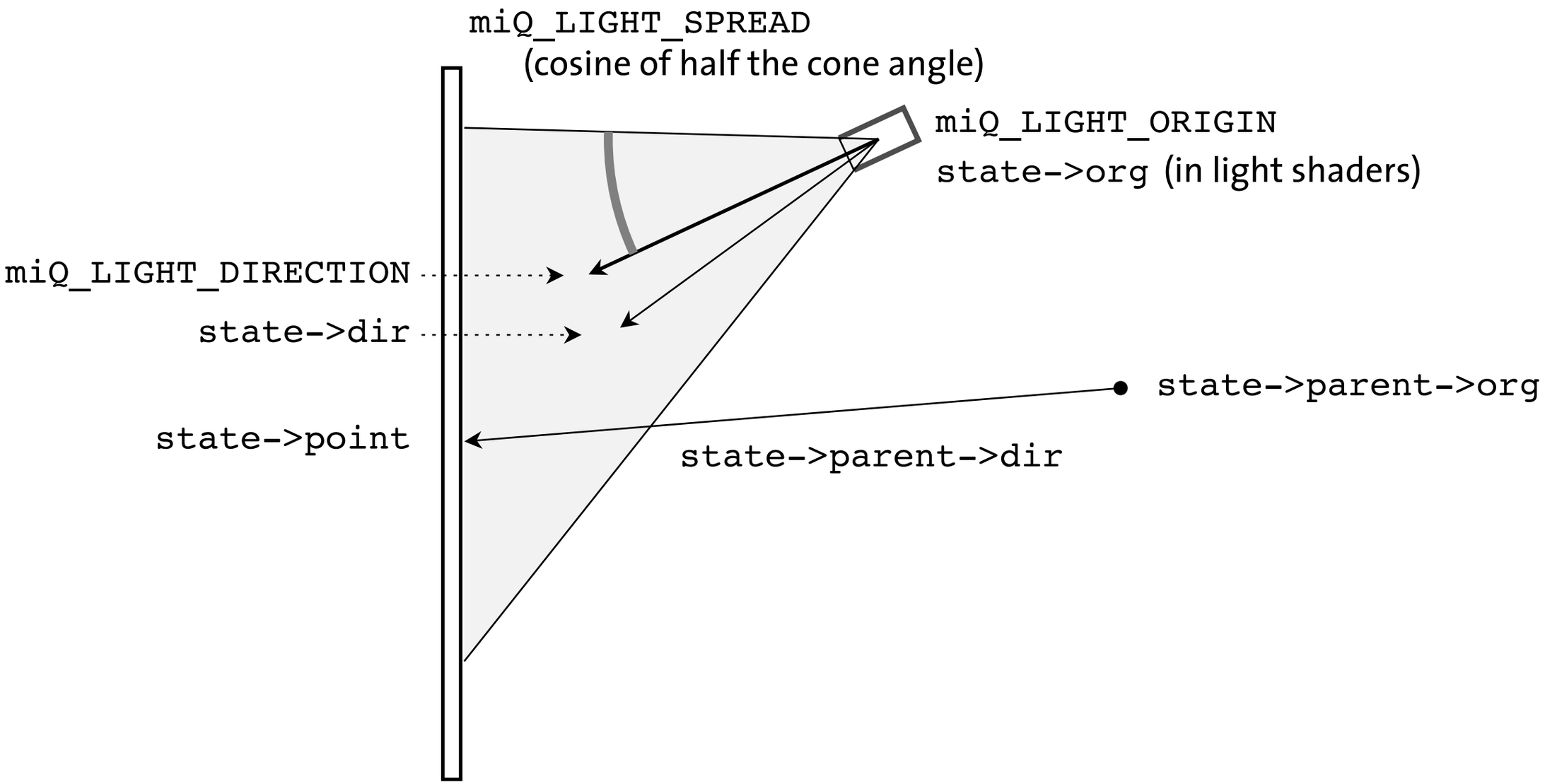
A simple spotlight



Accessing the scene element database with mi_query

Lights

A simple spotlight



State fields and query codes used in light shaders

```
1  miTag miaux_current_light_tag(miState *state)
2  {
3      miTag light_tag;
4      mi_query(miQ_INST_ITEM, state, state->light_instance, &light_tag);
5      return light_tag;
6  }
```

Auxiliary function: miaux_current_light_tag

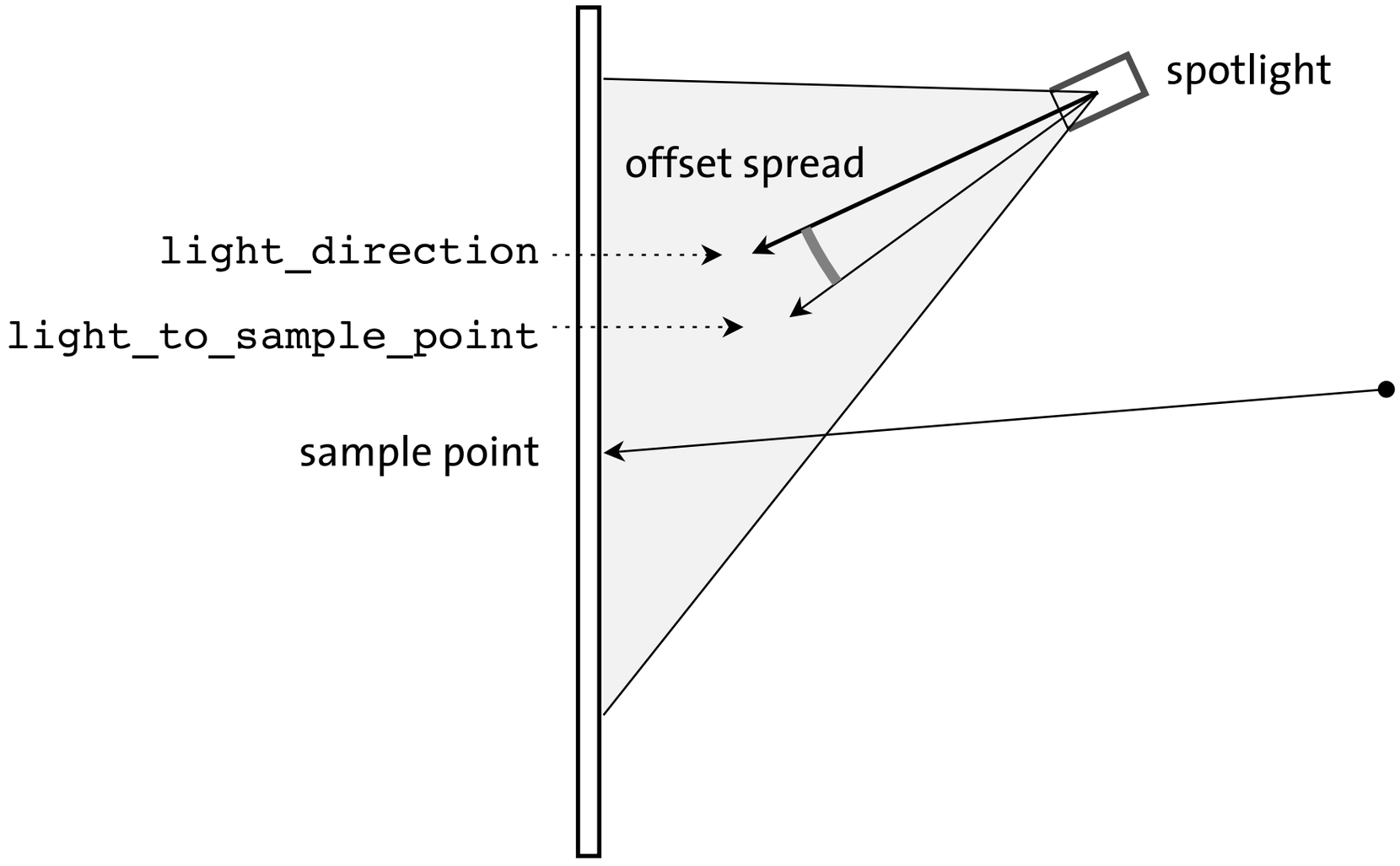
```
1  miScalar miaux_light_spread(miState *state, miTag light_tag)
2  {
3      miScalar light_spread;
4      mi_query(miQ_LIGHT_SPREAD, state, light_tag, &light_spread);
5      return light_spread;
6  }
```

Auxiliary function: miaux_light_spread

```
1  miScalar miaux_offset_spread_from_light(miState *state, miTag light_tag)
2  {
3      miVector light_direction, light_to_sample_point;
4
5      mi_query(miQ_LIGHT_DIRECTION, state, light_tag, &light_direction);
6      mi_vector_normalize(&light_direction);
7
8      mi_vector_to_light(state, &light_to_sample_point, &state->dir);
9      mi_vector_normalize(&light_to_sample_point);
10
11     return mi_vector_dot(&light_to_sample_point, &light_direction);
12 }
```

Auxiliary function: miaux_offset_spread_from_light

A simple spotlight



Vectors compared for the offset spread in function miaux_offset_spread_from_light

A simple spotlight

```
declare shader
    color "spotlight" (
        color "light_color" default 1 1 1, )
end declare
```

Scene file declaration of shader "spotlight"

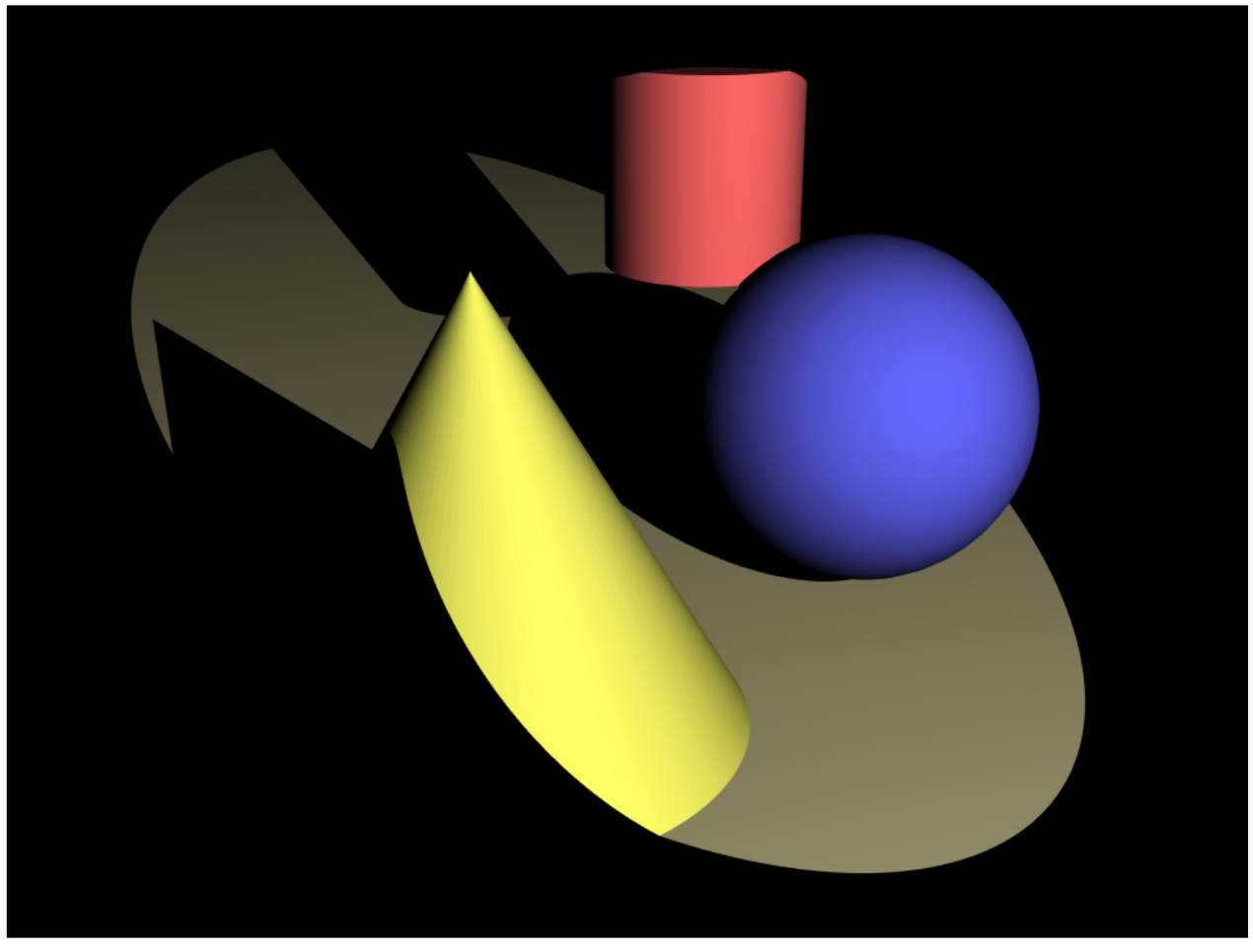
A simple spotlight

```
1   struct spotlight {
2       miColor light_color;
3   };
4
5   miBoolean spotlight (
6       miColor *result, miState *state, struct spotlight *params )
7   {
8       miTag light_tag = miaux_current_light_tag(state);
9
10      if (miaux_offset_spread_from_light(state, light_tag)
11          > miaux_light_spread(state, light_tag)) {
12          *result = *mi_eval_color(&params->light_color);
13          return mi_trace_shadow(result, state);
14      }
15      else
16          return miFALSE;
17  }
```

Source code of shader "spotlight"

Lights

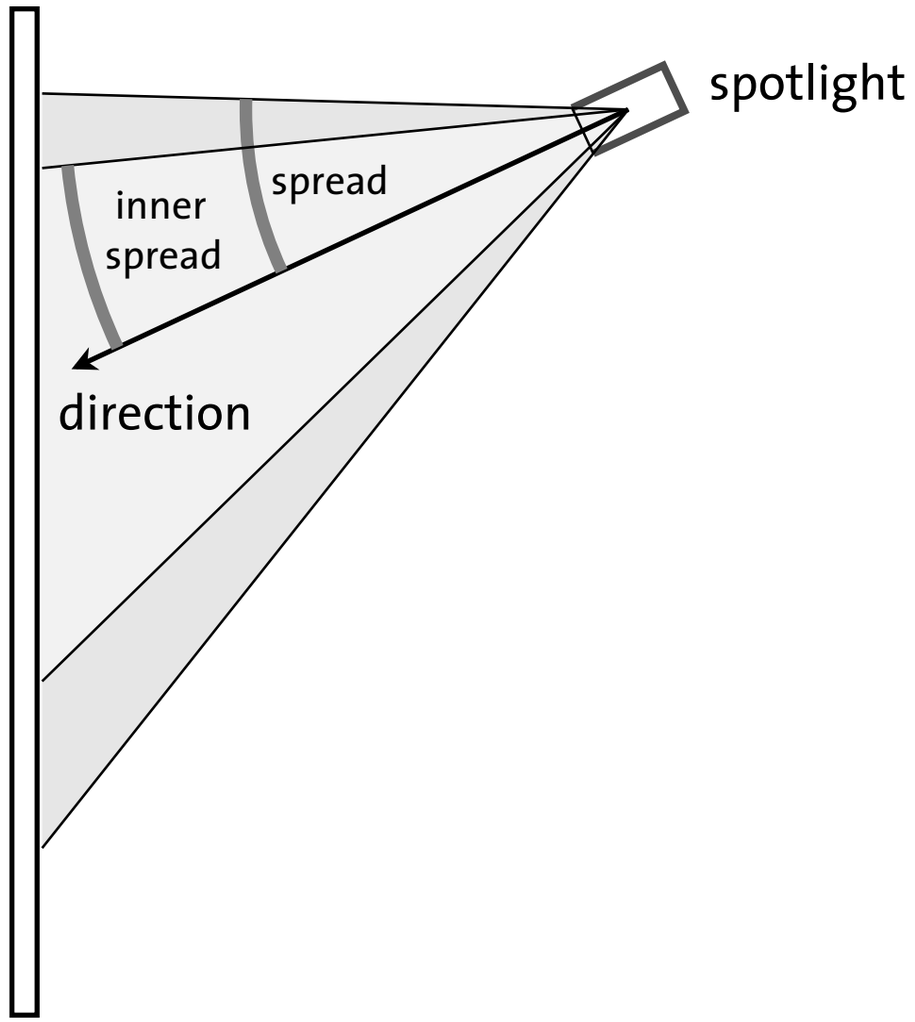
A simple spotlight



```
light "white_spotlight"
    "spotlight" ()
    origin 2 2 5
    direction -.75 -1 -2.3
    spread .978
end light

instance "light_instance"
    "white_spotlight"
end instance
```

Spotlight with cone represented by the cosine value of 0.978 (the ``spread'')

A spotlight with a soft edge



An inner cone of full intensity inside the full cone of the spotlight

A spotlight with a soft edge

```
declare shader
    color "soft_spotlight" (
        color "light_color" default 1 1 1,
        scalar "inner_spread" default 1 )
end declare
```

Scene file declaration of shader "soft_spotlight"

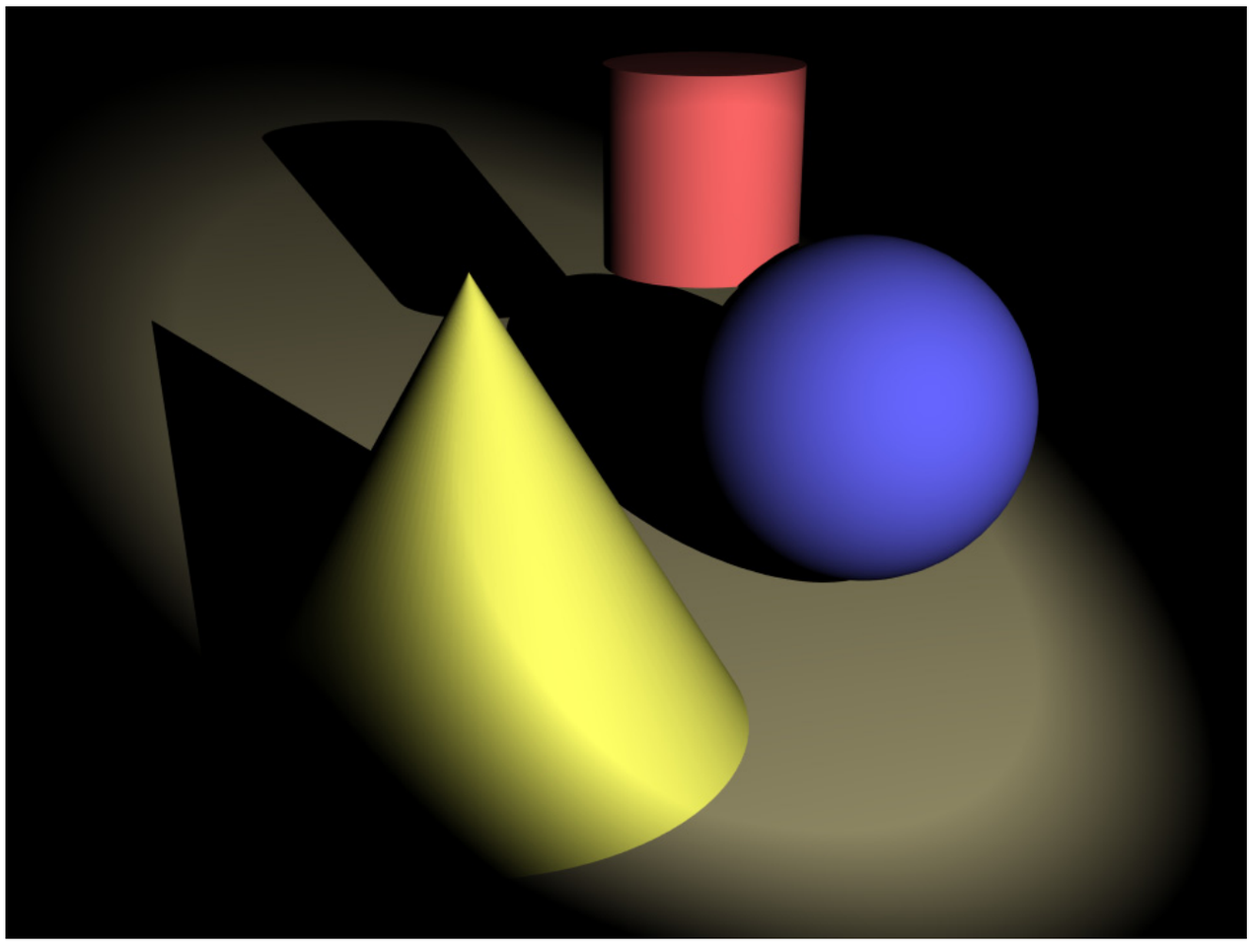
A spotlight with a soft edge

```
 1  struct soft_spotlight {
 2      miColor light_color;
 3      miScalar inner_spread;
 4  };
 5
 6  miBoolean soft_spotlight (
 7      miColor *result, miState *state, struct soft_spotlight *params)
 8  {
 9      miScalar inner_spread, attenuation;
10      miTag light_tag = miaux_current_light_tag(state);
11      miScalar offset_spread = miaux_offset_spread_from_light(state, light_tag);
12      miScalar light_spread = miaux_light_spread(state, light_tag);
13
14      if (offset_spread < light_spread)
15          return miFALSE;
16
17      *result = *mi_eval_color(&params->light_color);
18      inner_spread = *mi_eval_scalar(&params->inner_spread);
19
20      if (offset_spread < inner_spread) {
21          attenuation = miaux_fit(offset_spread, inner_spread, light_spread, 1, 0);
22          miaux_scale_color(result, attenuation);
23      }
24      return mi_trace_shadow(result, state);
25  }
```

Source code of shader "soft_spotlight"

Lights

A spotlight with a soft edge



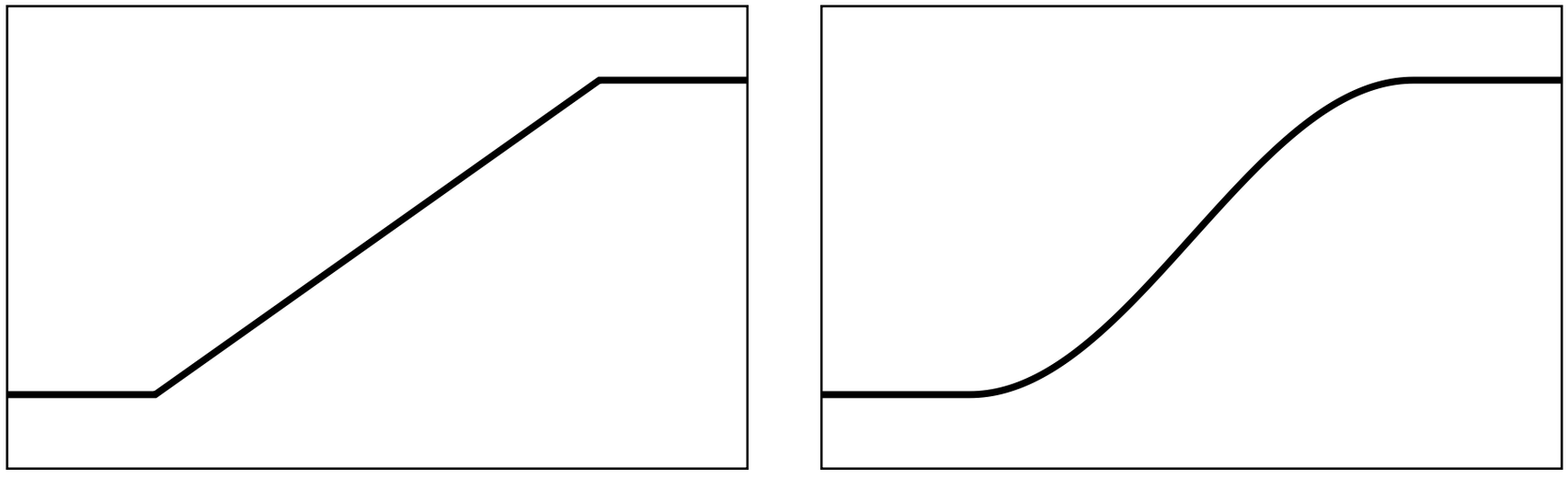
```
light "white_spotlight"
    "soft_spotlight" (
        "inner_spread" .982 )
    origin 2 2 5
    direction -.75 -1 -2.3
    spread .965
end light

instance "light_instance"
    "white_spotlight"
end instance
```

Spotlight with linear transition from inner cone to outer cone

```
1  double miaux_fit(
2      double v, double oldmin, double oldmax, double newmin, double newmax)
3  {
4      return newmin + ((v - oldmin) / (oldmax - oldmin)) * (newmax - newmin);
5  }
```

Auxiliary function: miaux_fit

A spotlight with a better soft edge



Linear transition

Sinusoidal transition

Linear and sinusoidal transitions between two values

```
1  double miaux_sinusoid_fit(
2      double v, double oldmin, double oldmax, double newmin, double newmax)
3  {
4      return miaux_fit(sin(miaux_fit(v, oldmin, oldmax, -M_PI_2, M_PI_2)),
5                       -1, 1,
6                       newmin, newmax);
7  }
```

Auxiliary function: miaux_sinusoid_fit

A spotlight with a better soft edge

```
declare shader
    color "sinusoid_soft_spotlight" (
        color "light_color" default 1 1 1,
        scalar "inner_spread" default 1 )
end declare
```

Scene file declaration of shader "sinusoid_soft_spotlight"

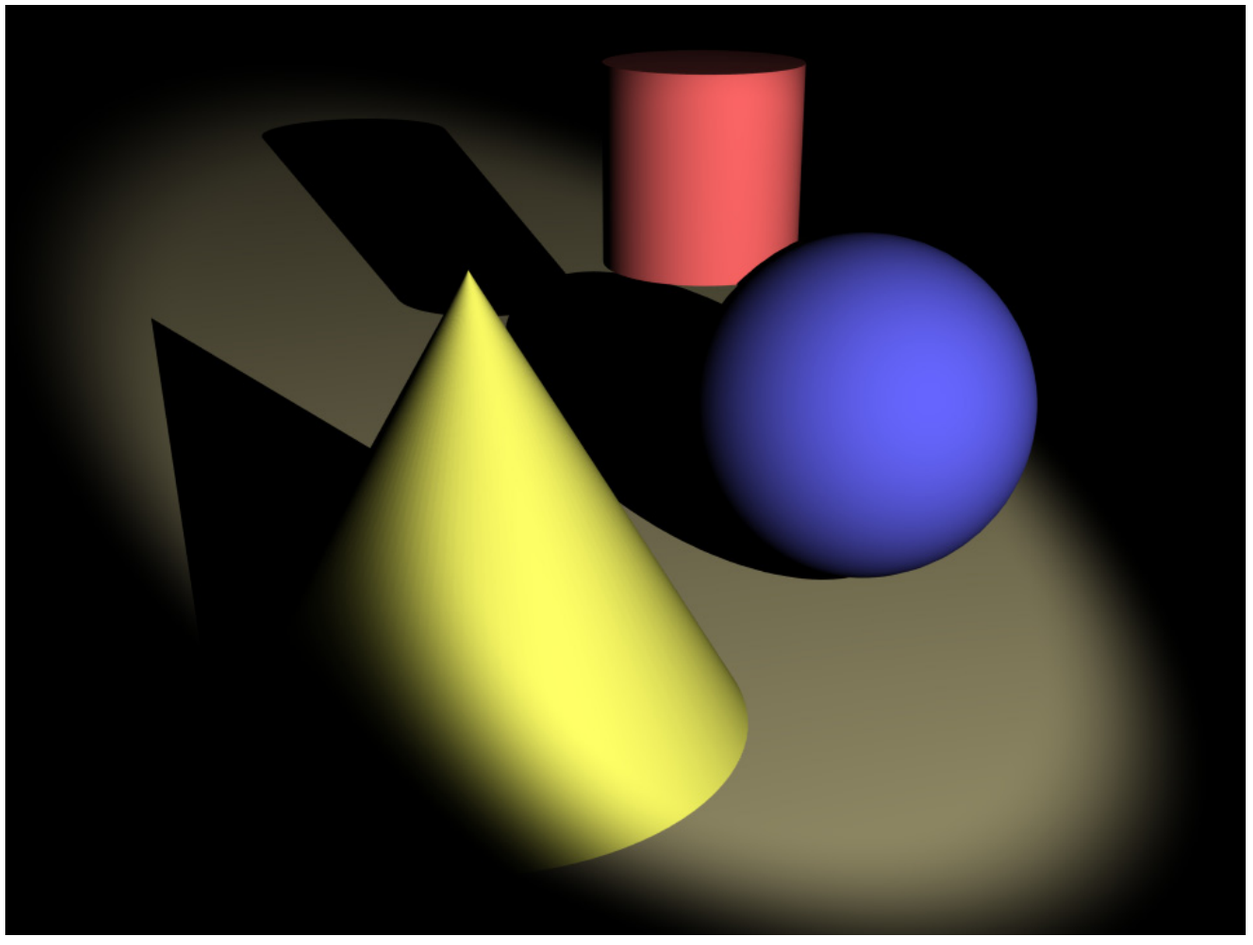
A spotlight with a better soft edge

```
 1  struct sinusoid_soft_spotlight {
 2      miColor light_color;
 3      miScalar inner_spread;
 4  };
 5
 6  miBoolean sinusoid_soft_spotlight(
 7      miColor *result, miState *state, struct sinusoid_soft_spotlight *params)
 8  {
 9      miScalar inner_spread, attenuation;
10      miTag light_tag = miaux_current_light_tag(state);
11      miScalar offset_spread = miaux_offset_spread_from_light(state, light_tag);
12      miScalar light_spread = miaux_light_spread(state, light_tag);
13
14      if (offset_spread < light_spread)
15          return miFALSE;
16
17      *result = *mi_eval_color(&params->light_color);
18      inner_spread = *mi_eval_scalar(&params->inner_spread);
19
20      if (offset_spread < inner_spread) {
21          attenuation =
22              miaux_sinusoid_fit(offset_spread, inner_spread, light_spread, 1, 0);
23          miaux_scale_color(result, attenuation);
24      }
25      return mi_trace_shadow(result, state);
26  }
```

Source code of shader "sinusoid_soft_spotlight"

Lights

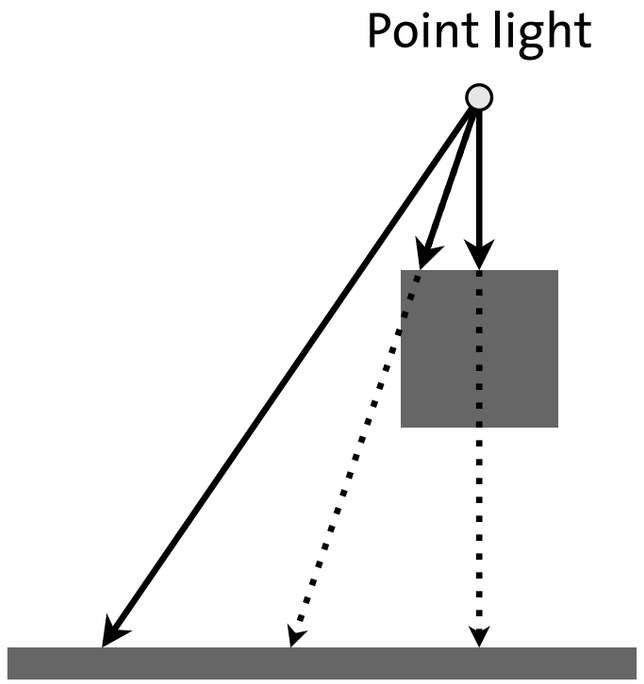
A spotlight with a better soft edge



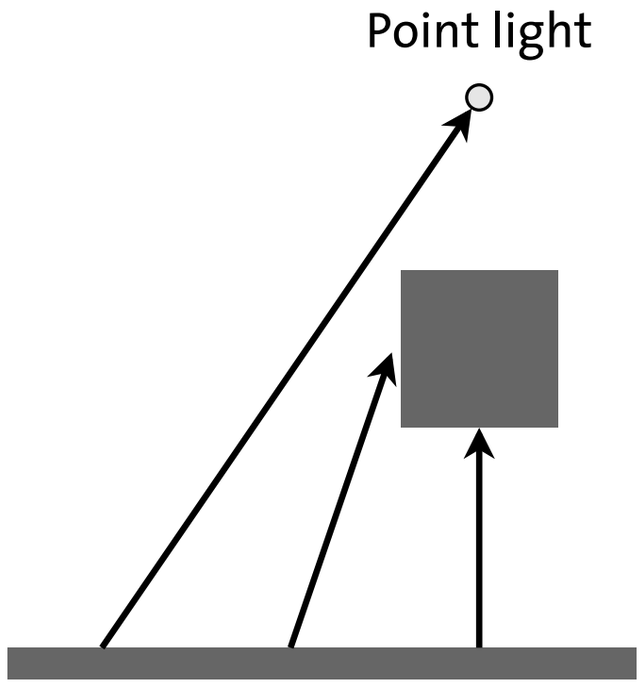
```
light "white_spotlight"
    "sinusoid_soft_spotlight" (
        "inner_spread" .982 )
    origin 2 2 5
    direction -.75 -1 -2.3
    spread .965
end light

instance "light_instance"
    "white_spotlight"
end instance
```

Spotlight with sinusoidal transition from inner cone to outer cone

Soft shadows using area lights

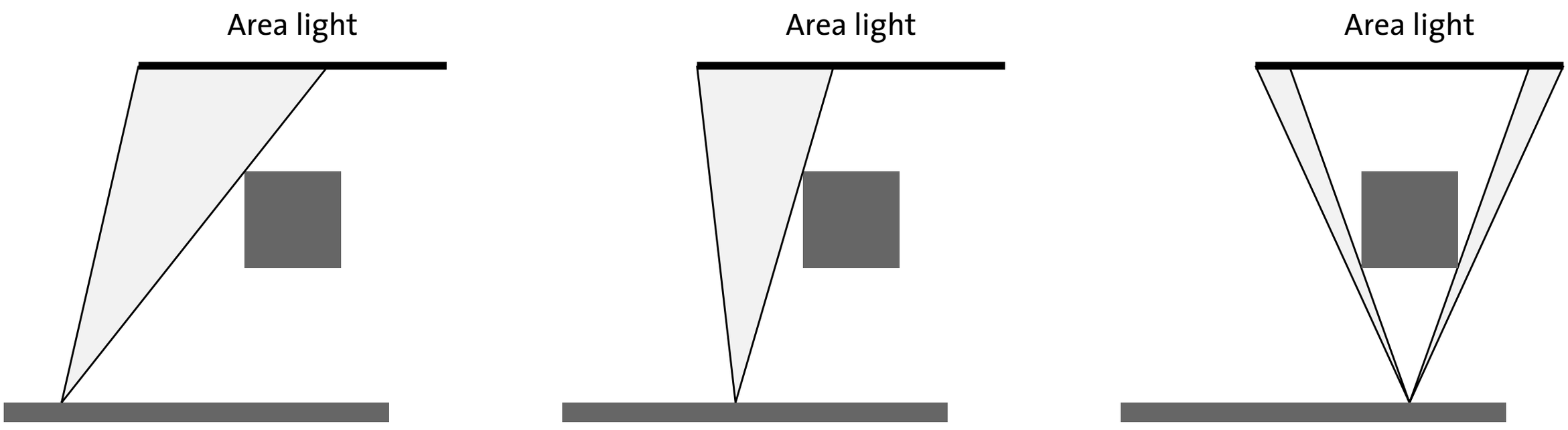


From the light to the surface



From the surface to the light

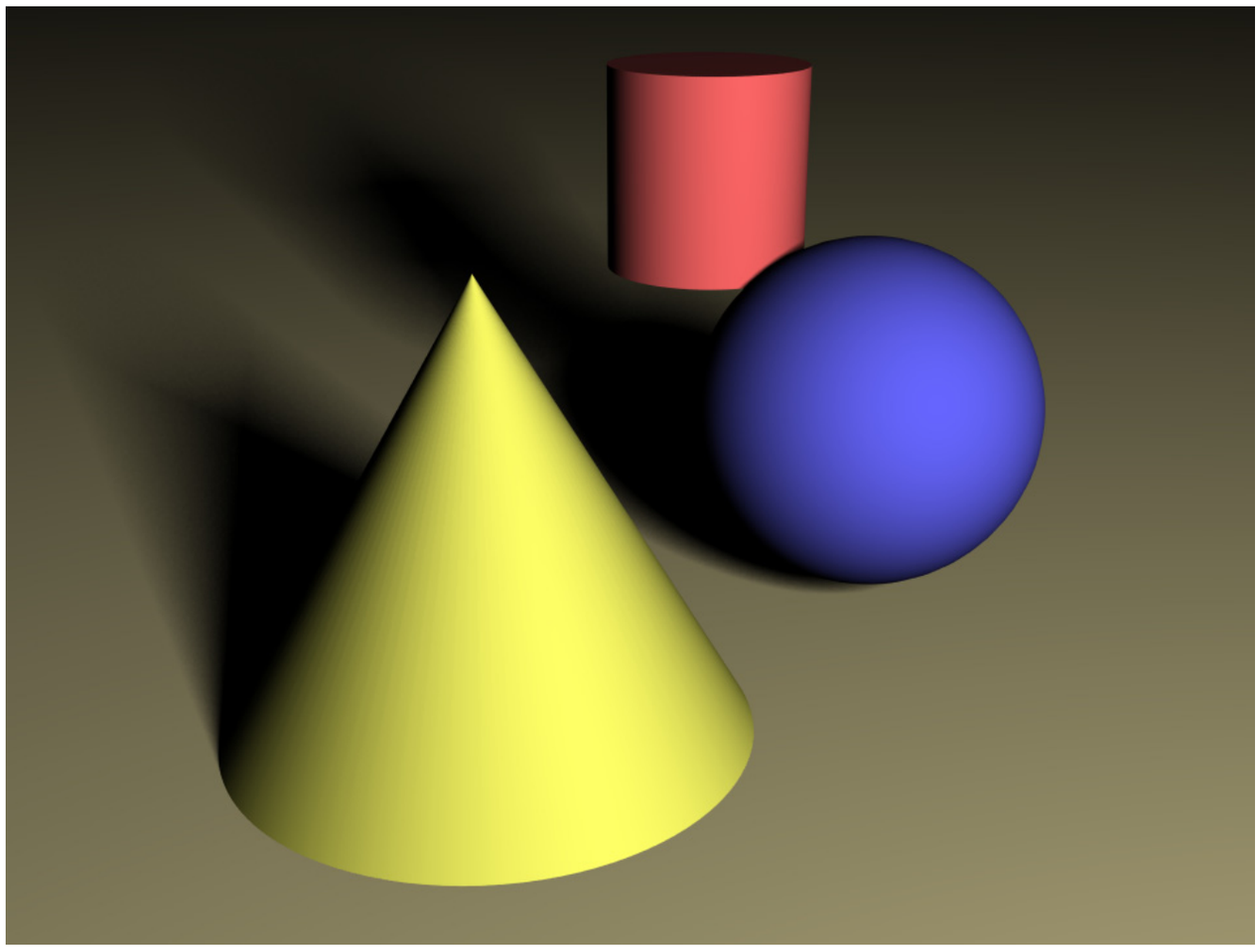
Soft shadows using area lights



Amount of the area light visible to the surface from different positions

Lights

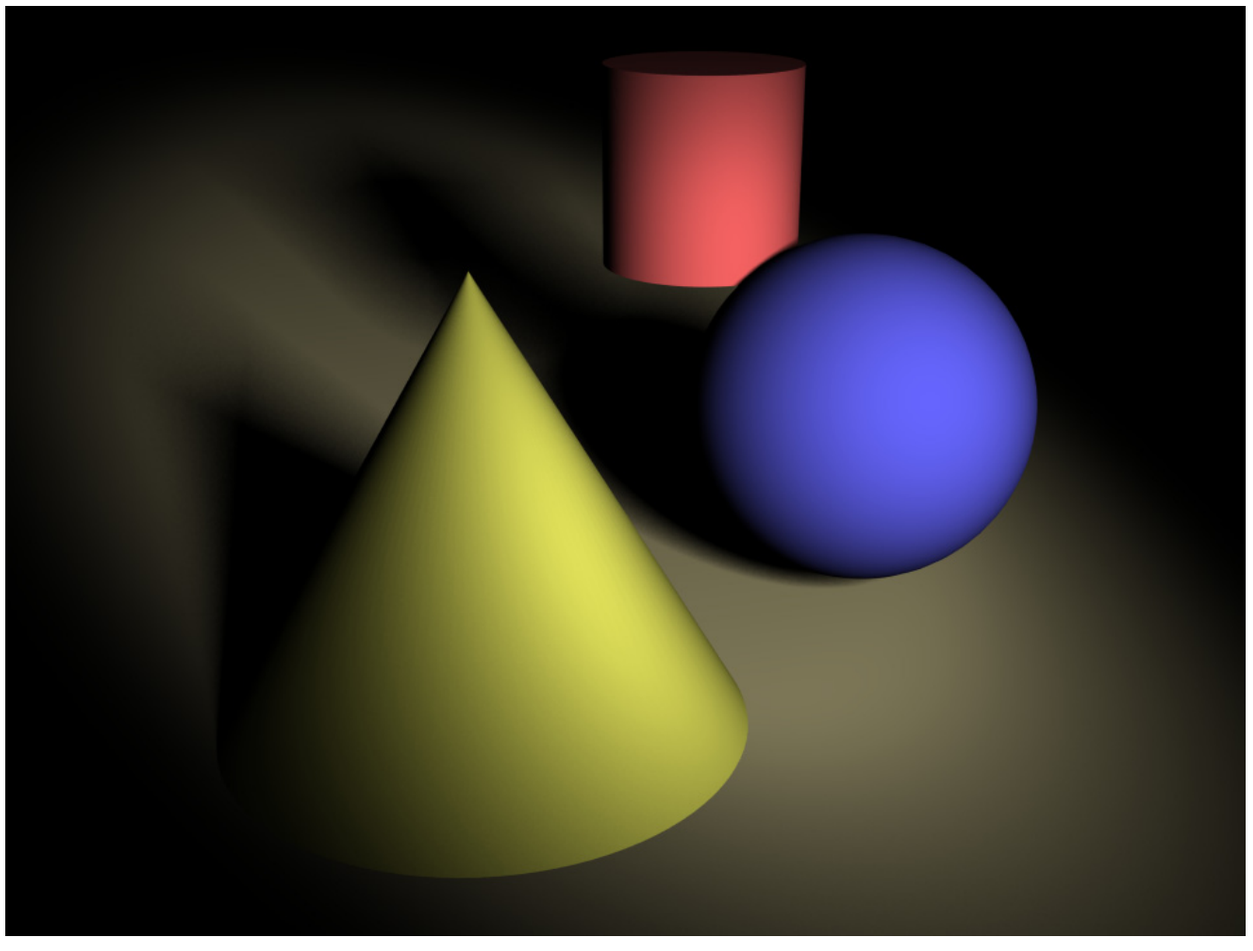
Soft shadows using area lights



```
light "white_light"
    "point_light_shadow" ()
    origin 2 2 5
    sphere .5 8 8
end light

instance "light_instance"
    "white_light"
end instance
```

Point light shader with sphere area light command in light definition

Soft shadows using area lights



```
light "white_spotlight"
    "sinusoid_soft_spotlight" (
        "inner_spread" .982 )
    origin 2 2 5
    direction -.75 -1 -2.3
    spread .965
    sphere .5 8 8
end light

instance "light_instance"
    "white_spotlight"
end instance
```

Spotlight shader with sphere area light command in light definition

Modifying light intensity based on distance

```
1  miScalar miaux_light_exponent(miState *state, miTag light_tag)
2  {
3      miScalar light_exponent;
4      mi_query(miQ_LIGHT_EXPONENT, state, light_tag, &light_exponent);
5      return light_exponent;
6  }
```

Auxiliary function: miaux_light_exponent

Modifying light intensity based on distance

```
declare shader
    color "point_light_falloff" (
        color "light_color" default 1 1 1 )
end declare
```

Scene file declaration of shader "point_light_falloff"

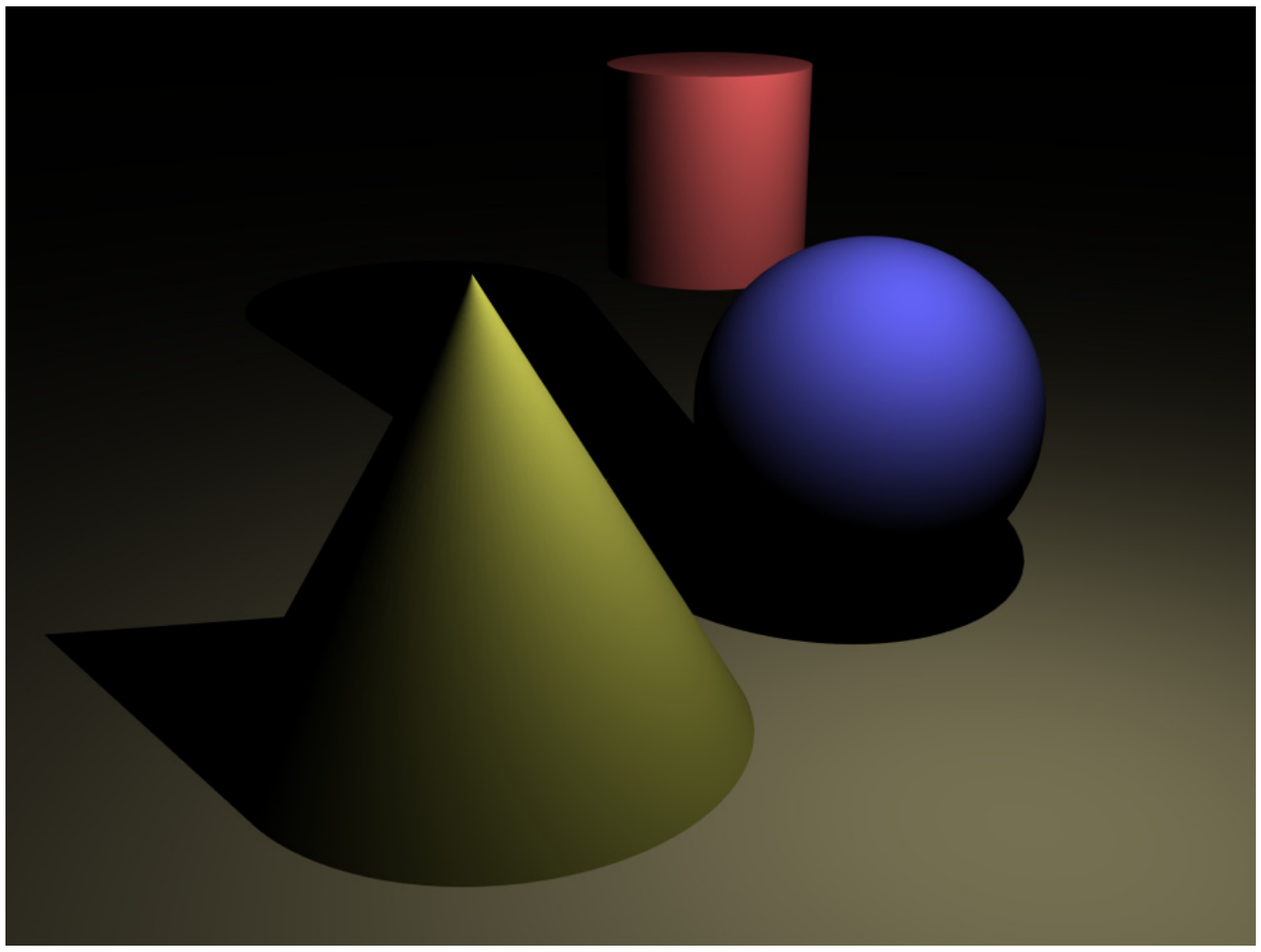
Modifying light intensity based on distance

```
1  struct point_light_falloff {
2      miColor light_color;
3  };
4
5  miBoolean point_light_falloff (
6      miColor *result, miState *state, struct point_light_falloff *params)
7  {
8      miTag light;
9      miScalar exponent;
10     *result = *mi_eval_color(&params->light_color);
11
12     mi_query(miQ_INST_ITEM, state, state->light_instance, &light);
13     mi_query(miQ_LIGHT_EXPONENT, state, light, &exponent);
14
15     miaux_scale_color(result, 1.0 / pow(state->dist, exponent));
16     return mi_trace_shadow(result, state);
17 }
```

Source code of shader "point_light_falloff"

Lights

Modifying light intensity based on distance

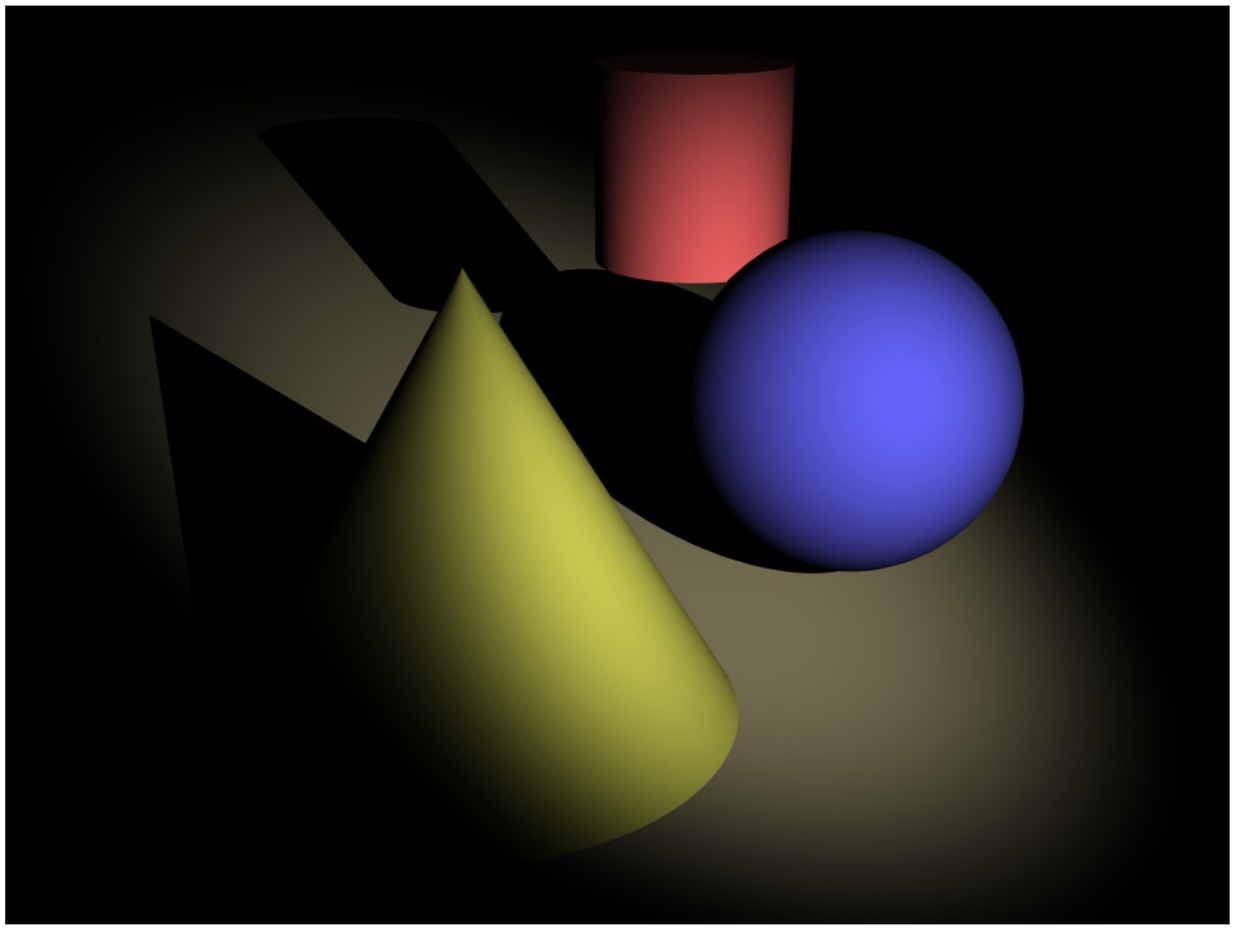


```
light "falloff_light"
    "point_light_falloff" (
        "light_color" 3 3 3 )
    origin 1 1.8 1
    exponent 2
end light

instance "light_instance"
    "falloff_light"
end instance
```

Point light with an exponential falloff 2.0

Lights

Defining good default values for parameters



```
light "white_spotlight"
    "sinusoid_soft_spotlight" ()
    origin 2 2 5
    direction -.75 -1 -2.3
    spread .965
end light

instance "light_instance"
    "white_spotlight"
end instance
```

Spotlight with sinusoidal transition and default inner spread value of 1.0

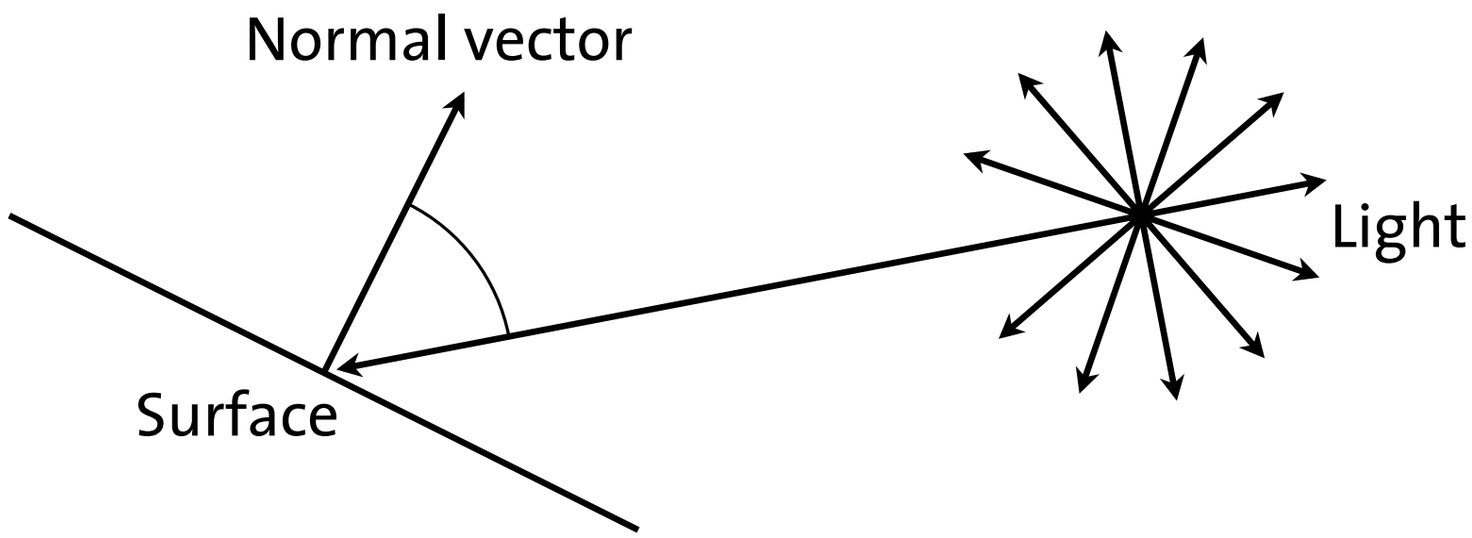
Light on a surface

Light on a surface

- Diffuse reflection: Lambert shading
- Specular reflections
  - The Phong specular model
  - The Blinn specular model
  - The Cook-Torrance specular model
  - The Ward specular model
- Faking the specular component
- Adding arbitrary effects to the Lambert model

Light on a surface

Diffuse reflection: Lambert shading



The angle between the surface normal and the incident light direction

Light on a surface

Diffuse reflection: Lambert shading

```
1  void miaux_add_diffuse_component(
2      miColor *result,
3      miScalar light_and_surface_cosine,
4      miColor *diffuse, miColor *light_color)
5  {
6      result->r += light_and_surface_cosine * diffuse->r * light_color->r;
7      result->g += light_and_surface_cosine * diffuse->g * light_color->g;
8      result->b += light_and_surface_cosine * diffuse->b * light_color->b;
9  }
```

Auxiliary function: miaux_add_diffuse_component

```
1  void miaux_add_scaled_color(miColor *result, miColor *color, miScalar scale)
2  {
3      result->r += color->r * scale;
4      result->g += color->g * scale;
5      result->b += color->b * scale;
6  }
```

Auxiliary function: miaux_add_scaled_color

```
1  void miaux_set_channels(miColor *c, miScalar new_value)
2  {
3       c->r = c->g = c->b = c->a = new_value;
4  }
```

Auxiliary function: miaux_set_channels

Light on a surface

Diffuse reflection: Lambert shading

```
declare shader
    color "lambert" (
        color "ambient" default 0 0 0,
        color "diffuse" default 1 1 1,
        array light "lights" )
end declare
```

Scene file declaration of shader "lambert"

Shader declaration in the scene file

```
array light "lights"
```

Fields in the parameter struct in the C source file

```
int   index_offset_to_first_light;
int   number_of_lights;
miTag light_array[1];
```

Declaration of an array in the .mi file and the three associated C struct parameters

```
1  struct lambert {
2      miColor ambient;
3      miColor diffuse;
4      int     i_light;
5      int     n_light;
6      miTag   light[1];
7  };
```

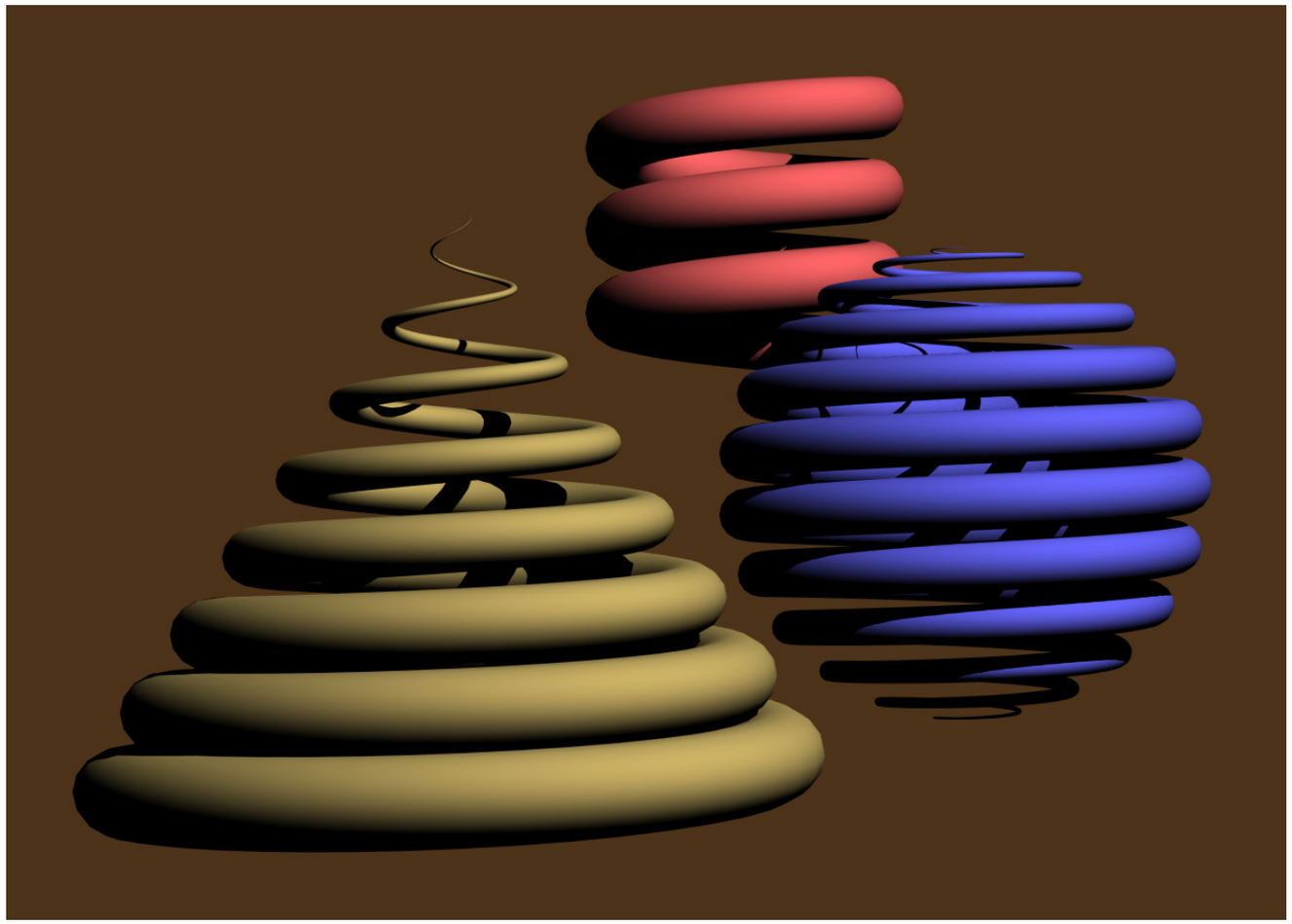
Source code of struct "lambert"

```
1  void miaux_light_array(miTag **lights, int *light_count, miState *state,
2                         int *offset_param, int *count_param, miTag *lights_param)
3  {
4      int array_offset = *mi_eval_integer(offset_param);
5      *light_count = *mi_eval_integer(count_param);
6      *lights = mi_eval_tag(lights_param) + array_offset;
7  }
```

Auxiliary function: miaux_light_array

Light on a surface

Diffuse reflection: Lambert shading



```
material "yellow"
    "lambert" (
        "diffuse" .8 .7 .4,
        "lights" ["light_inst"] )
end material

material "blue"
    "lambert" (
        "diffuse" .4 .4 1,
        "lights" ["light_inst"] )
end material

material "red"
    "lambert" (
        "diffuse" 1 .4 .4,
        "lights" ["light_inst"] )
end material
```

Lambert shading

Light on a surface

Diffuse reflection: Lambert shading

```
 1  struct lambert {
 2      miColor ambient;
 3      miColor diffuse;
 4      int     i_light;
 5      int     n_light;
 6      miTag   light[1];
 7  };
 8
 9  miBoolean lambert (
10      miColor *result, miState *state, struct lambert *params )
11  {
12      int i, light_count, light_sample_count;
13      miColor sum, light_color;
14      miScalar dot_nl;
15      miTag *light;
16
17      miColor *diffuse  = mi_eval_color(&params->diffuse);
18      miaux_light_array(&light, &light_count, state,
19                        &params->i_light, &params->n_light, params->light);
20      *result = *mi_eval_color(&params->ambient);
21
22      for (i = 0; i < light_count; i++, light++) {
23          miaux_set_channels(&sum, 0);
24          light_sample_count = 0;
25          while (mi_sample_light(&light_color, NULL, &dot_nl,
26                                 state, *light, &light_sample_count))
27              miaux_add_diffuse_component(&sum, dot_nl, diffuse, &light_color);
28          if (light_sample_count)
29              miaux_add_scaled_color(result, &sum, 1.0/light_sample_count);
30      }
31      return miTRUE;
32  }
```

Source code of shader "lambert"

```
declare shader
    color "phong" (
        color "ambient"   default 0 0 0,
        color "diffuse"   default 1 1 1,
        color "specular"  default 0 0 0,
        scalar "exponent" default 30,
        array light "lights" )
end declare
```

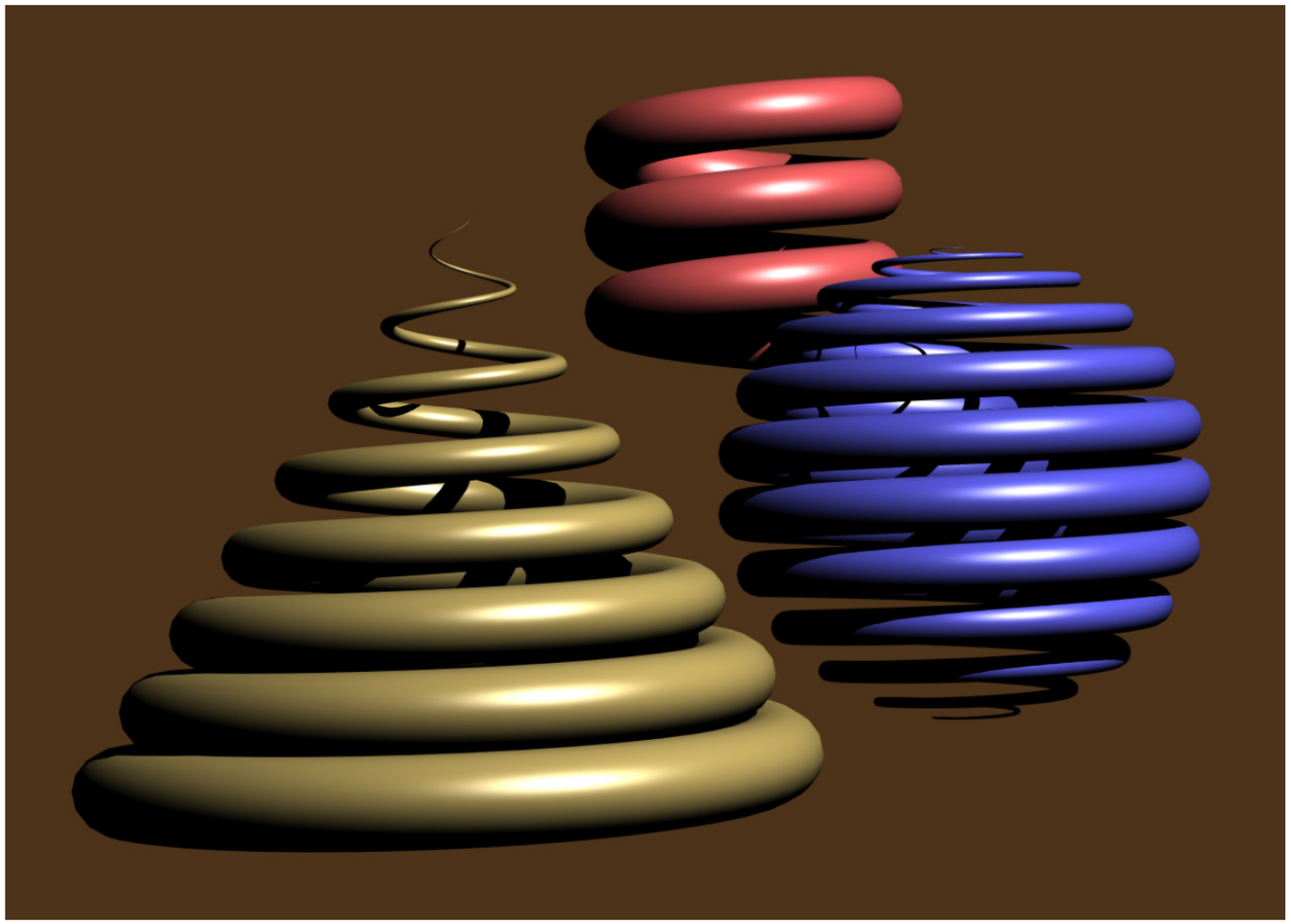
Scene file declaration of shader "phong"

```
1  void miaux_add_phong_specular_component(
2      miColor *result, miState *state, miScalar exponent,
3      miVector *direction_toward_light,
4      miColor *specular, miColor *light_color)
5  {
6      miScalar specular_amount =
7          mi_phong_specular(exponent, state, direction_toward_light);
8      if (specular_amount > 0.0) {
9          result->r += specular_amount * specular->r * light_color->r;
10         result->g += specular_amount * specular->g * light_color->g;
11         result->b += specular_amount * specular->b * light_color->b;
12     }
13 }
```

Auxiliary function: miaux_add_phong_specular_component

Light on a surface

Specular reflections



```
material "yellow"
     "phong" (
        "diffuse" .8 .7 .4,
        "specular" .8 .8 .8,
        "exponent" 40,
        "lights" ["light_inst"] )
end material

material "blue"
    "phong" (
        "diffuse" .4 .4 1,
        "specular" .8 .8 .8,
        "exponent" 40,
        "lights" ["light_inst"] )
end material

material "red"
    "phong" (
        "diffuse" 1 .4 .4,
        "specular" .8 .8 .8,
        "exponent" 40,
        "lights" ["light_inst"] )
end material
```

Phong shading

Light on a surface


```
 1  struct phong {
 2      miColor  ambient;
 3      miColor  diffuse;
 4      miColor  specular;
 5      miScalar exponent;
 6      int      i_light;
 7      int      n_light;
 8      miTag    light[1];
 9  };
10
11  miBoolean phong (
12      miColor *result, miState *state, struct phong *params )
13  {
14      int i, light_count, light_sample_count;
15      miColor sum, light_color;
16      miVector direction_toward_light;
17      miScalar dot_nl;
18      miTag *light;
19
20      miColor *diffuse  = mi_eval_color(&params->diffuse);
21      miColor *specular = mi_eval_color(&params->specular);
22      miScalar exponent = *mi_eval_scalar(&params->exponent);
23      miaux_light_array(&light, &light_count, state,
24                        &params->i_light, &params->n_light, params->light);
25      *result = *mi_eval_color(&params->ambient);
26
27      for (i = 0; i < light_count; i++, light++) {
28          miaux_set_channels(&sum, 0);
29          light_sample_count = 0;
30          while (mi_sample_light(&light_color, &direction_toward_light, &dot_nl,
31                                 state, *light, &light_sample_count)) {
32              miaux_add_diffuse_component(&sum, dot_nl, diffuse, &light_color);
33              miaux_add_phong_specular_component(&sum, state, exponent,
34                                                 &direction_toward_light,
35                                                 specular, &light_color);
36          }
37          if (light_sample_count)
38              miaux_add_scaled_color(result, &sum, 1.0/light_sample_count);
39      }
40      return miTRUE;
41  }
```

Source code of shader "phong"

```
declare shader
    color "blinn" (
        color  "ambient"                default 0 0 0,
        color  "diffuse"                default 1 1 1,
        color  "specular"               default 0 0 0,
        scalar "avg_microfacet_slope" default .2,
        scalar "index_of_refraction"  default 3,
        array light "lights" )
end declare
```

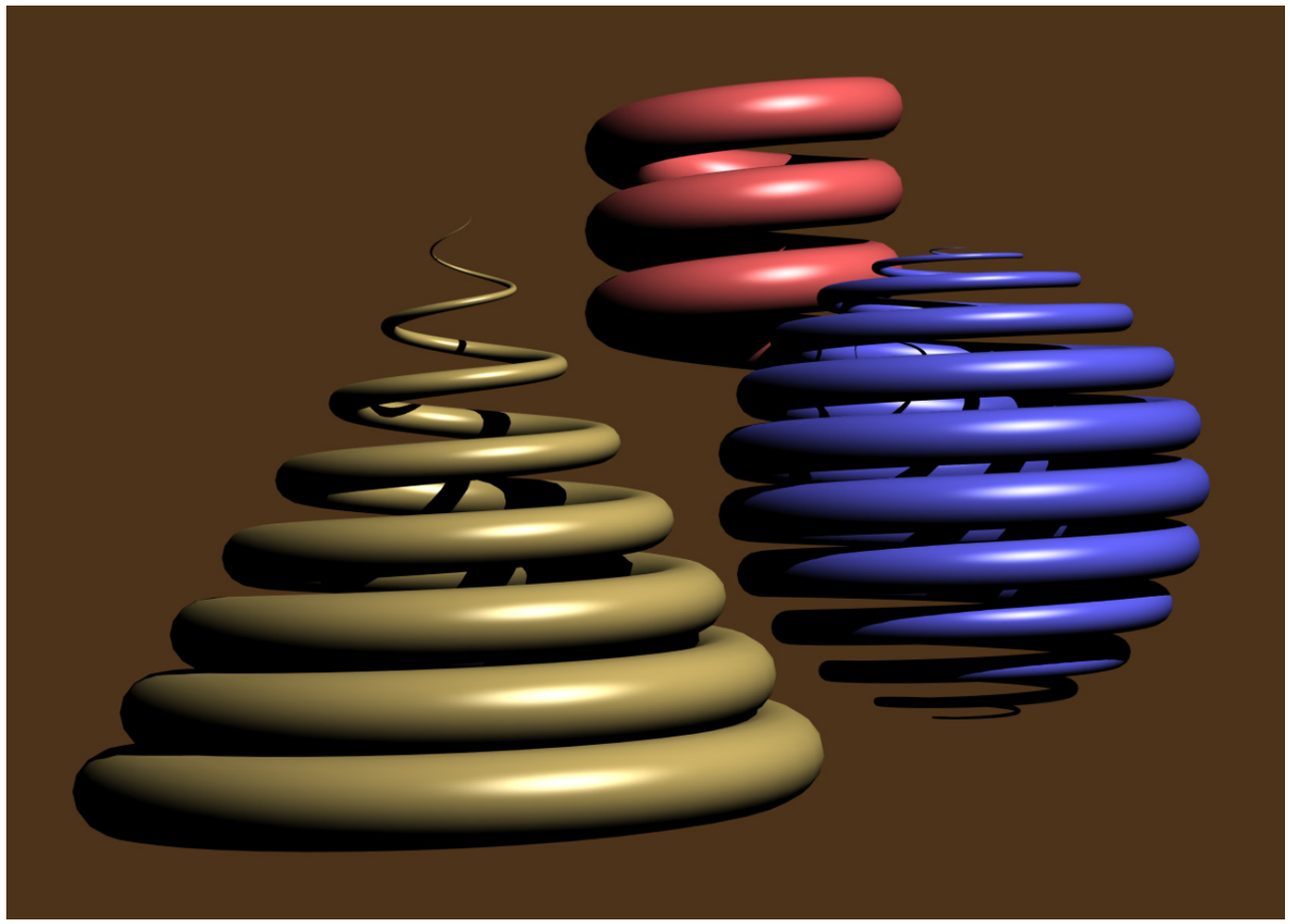
Scene file declaration of shader "blinn"

```
1  void miaux_add_blinn_specular_component(
2      miColor *result, miState *state,
3      miScalar roughness, miScalar ior,
4      miVector direction_toward_light,
5      miColor *specular, miColor *light_color)
6  {
7      miScalar specular_amount =
8          mi_blinn_specular(&state->dir, &direction_toward_light,
9                            &state->normal, roughness, ior);
10     if (specular_amount > 0.0) {
11         result->r += specular_amount * specular->r * light_color->r;
12         result->g += specular_amount * specular->g * light_color->g;
13         result->b += specular_amount * specular->b * light_color->b;
14     }
15 }
```

Auxiliary function: miaux_add_blinn_specular_component

Light on a surface

Specular reflections



```
material "yellow"
    "blinn" (
        "diffuse" .8 .7 .4,
        "specular" 2 2 2,
        "avg_microfacet_slope" .25,
        "index_of_refraction" 20,
        "lights" ["light_inst"] )
end material
```

Blinn shading

```
 1  struct blinn {
 2      miColor  ambient;
 3      miColor  diffuse;
 4      miColor  specular;
 5      miScalar avg_microfacet_slope;
 6      miScalar index_of_refraction;
 7      int      i_light;
 8      int      n_light;
 9      miTag    light[1];
10  };
11
12  miBoolean blinn (
13      miColor *result, miState *state, struct blinn *params )
14  {
     /* ... */
23      miScalar avg_microfacet_slope = *mi_eval_scalar(&params->avg_microfacet_slope);
24      miScalar index_of_refraction  = *mi_eval_scalar(&params->index_of_refraction);
     /* ... */
35              miaux_add_blinn_specular_component(
36                  &sum, state, avg_microfacet_slope, index_of_refraction,
37                  direction_toward_light, specular, &light_color);
     /* ... */
```

Source code of shader "blinn"

```
declare shader
    color "cook_torrance" (
        color  "ambient"                    default 0 0 0,
        color  "diffuse"                    default 1 1 1,
        color  "specular"                   default 0 0 0,
        scalar "average_microfacet_slope" default .2,
        color  "index_of_refraction"        default 1 1 1,
        array light "lights" )
end declare
```

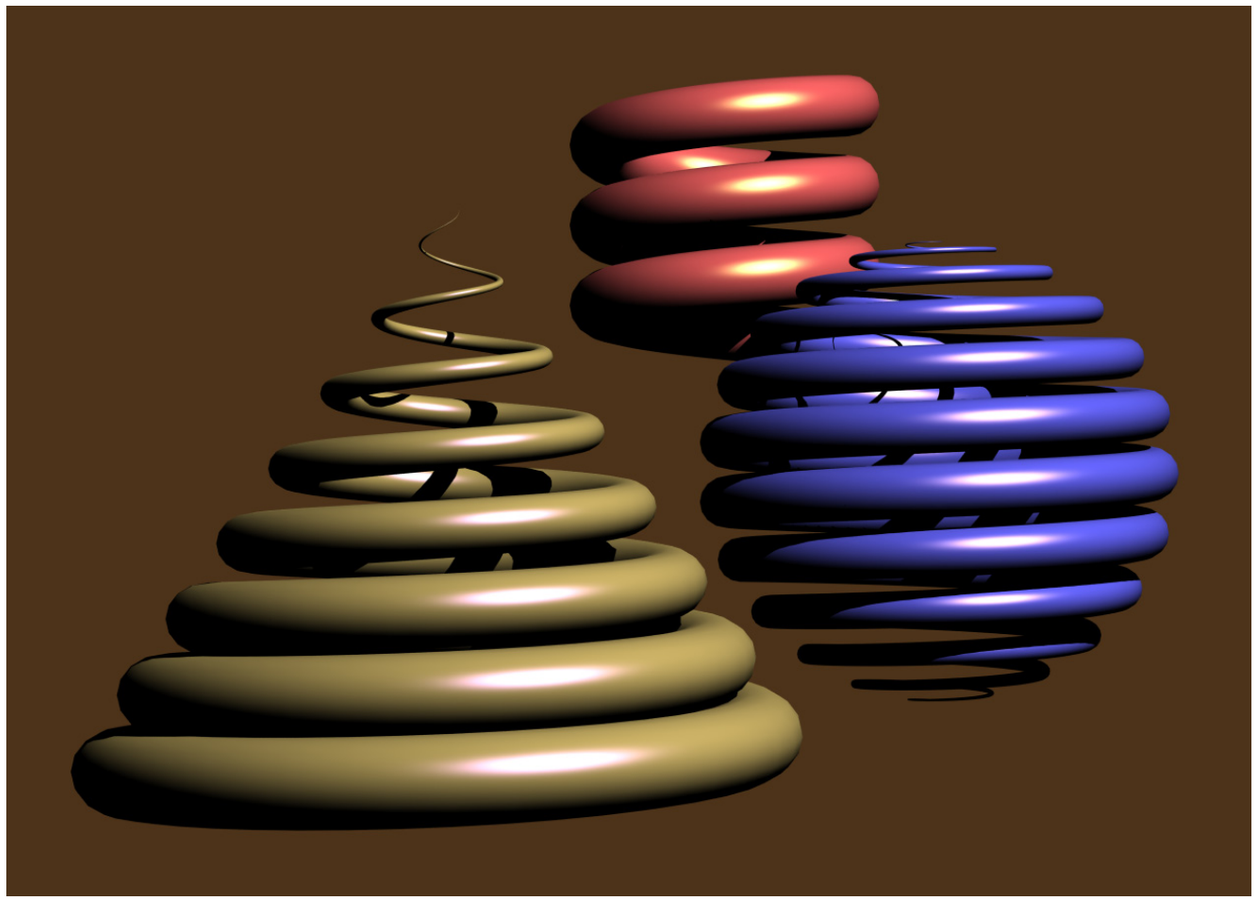
Scene file declaration of shader "cook_torrance"

```
 1  void miaux_add_cook_torrance_specular_component(
 2      miColor *result, miState *state,
 3      miScalar roughness, miColor *ior,
 4      miVector direction_toward_light,
 5      miColor *specular, miColor *light_color)
 6  {
 7      miColor specular_reflection_color;
 8      if (mi_cooktorr_specular(&specular_reflection_color, &state->dir,
 9                               &direction_toward_light,
10                               &state->normal, roughness, ior)) {
11          result->r += specular_reflection_color.r * specular->r * light_color->r;
12          result->g += specular_reflection_color.g * specular->g * light_color->g;
13          result->b += specular_reflection_color.b * specular->b * light_color->b;
14      }
15  }
```

Auxiliary function: miaux_add_cook_torrance_specular_component

Light on a surface

Specular reflections



```
material "yellow"
    "cook_torrance" (
        "diffuse" .8 .7 .4,
        "specular" 2 2 2,
        "index_of_refraction" 10 5 80,
        "lights" ["light_inst"] )
end material
```

Cook-Torrance shading

Light on a surface

Specular reflections

```
 1  struct cook_torrance {
 2      miColor  ambient;
 3      miColor  diffuse;
 4      miColor  specular;
 5      miScalar avg_microfacet_slope;
 6      miColor  index_of_refraction;
 7      int      i_light;
 8      int      n_light;
 9      miTag    light[1];
10  };
11
12  miBoolean cook_torrance (
13      miColor *result, miState *state, struct cook_torrance *params )
14  {
     /* ... */
23      miScalar avg_microfacet_slope = *mi_eval_scalar(&params->avg_microfacet_slope);
24      miColor *index_of_refraction  = mi_eval_color(&params->index_of_refraction);
     /* ... */
35              miaux_add_cook_torrance_specular_component(
36                  &sum, state, avg_microfacet_slope, index_of_refraction,
37                  direction_toward_light, specular, &light_color);
     /* ... */
```

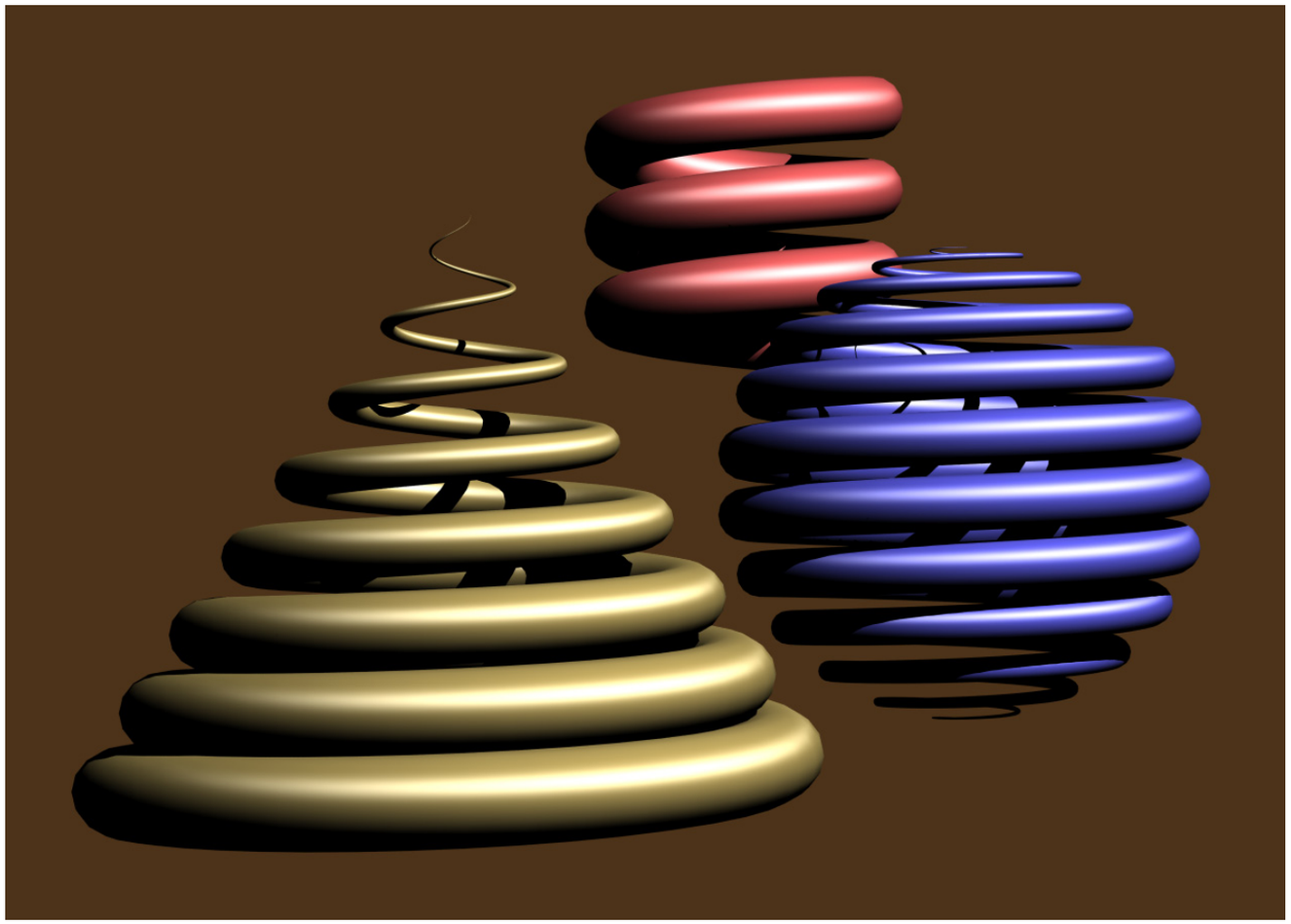
Source code of shader "cook_torrance"

```
declare shader
    color "ward" (
        color  "ambient"        default 0 0 0,
        color  "diffuse"        default 1 1 1,
        color  "glossy"         default 0 0 0,
        scalar "shiny_u_coeff" default 10,
        scalar "shiny_v_coeff" default 10,
        array light "lights" )
end declare
```

Scene file declaration of shader "ward"

Light on a surface

Specular reflections



```
material "yellow"
    "ward" (
        "diffuse" .8 .7 .4,
        "glossy"  2 2 2,
        "shiny_u_coeff" 1,
        "shiny_v_coeff" 4,
        "lights" ["light_inst"] )
end material
```

Ward shading

Light on a surface


```
 1  void miaux_add_ward_specular_component(
 2      miColor *result, miState *state,
 3      miScalar shiny_u, miScalar shiny_v,
 4      miColor *glossy, miScalar normal_dot_light,
 5      miVector direction_toward_light,
 6      miColor *light_color)
 7  {
 8      miScalar specular_reflection_amount;
 9      if (shiny_u == shiny_v)  /* Isotropic */
10          specular_reflection_amount = normal_dot_light *
11              mi_ward_glossy(
12                  &state->dir, &direction_toward_light, &state->normal, shiny_u);
13
14      else {                          /* Anisotropic */
15          miVector u = state->derivs[0], v;
16          float d = mi_vector_dot(&u, &state->normal);
17          u.x -= d * state->normal.x;
18          u.y -= d * state->normal.y;
19          u.z -= d * state->normal.z;
20          mi_vector_normalize(&u);
21          /* Set v to be perpendicular to u (in the tangent plane) */
22          mi_vector_prod(&v, &state->normal, &u);
23          specular_reflection_amount = normal_dot_light *
24              mi_ward_anisglossy(&state->dir, &direction_toward_light,
25                                 &state->normal, &u, &v, shiny_u, shiny_v);
26      }
27      if (specular_reflection_amount > 0.0) {
28          result->r += specular_reflection_amount * glossy->r * light_color->r;
29          result->g += specular_reflection_amount * glossy->g * light_color->g;
30          result->b += specular_reflection_amount * glossy->b * light_color->b;
31      }
32  }
```

Auxiliary function: miaux_add_ward_specular_component

Light on a surface


```
 1  struct ward {
 2      miColor  ambient;
 3      miColor  diffuse;
 4      miColor  glossy;
 5      miScalar shiny_u_coeff;
 6      miScalar shiny_v_coeff;
 7      int      i_light;
 8      int      n_light;
 9      miTag    light[1];
10  };
11
12  miBoolean ward (
13      miColor *result, miState *state, struct ward *params )
14  {
     /* ... */
22      miColor *glossy        = mi_eval_color(&params->glossy);
23      miScalar shiny_u_coeff = *mi_eval_scalar(&params->shiny_u_coeff);
24      miScalar shiny_v_coeff = *mi_eval_scalar(&params->shiny_v_coeff);
     /* ... */
35              miaux_add_ward_specular_component(
36                  &sum, state, shiny_u_coeff, shiny_v_coeff, glossy, dot_nl,
37                  direction_toward_light, &light_color);
     /* ... */
```

Source code of shader "ward"

```
declare shader
    color "mock_specular" (
        color "ambient"  default 0 0 0,
        color "diffuse"  default 1 1 1,
        color "specular" default 1 1 1,
        color "cutoff"   default .95 .95 .95,
        array light "lights" )
end declare
```

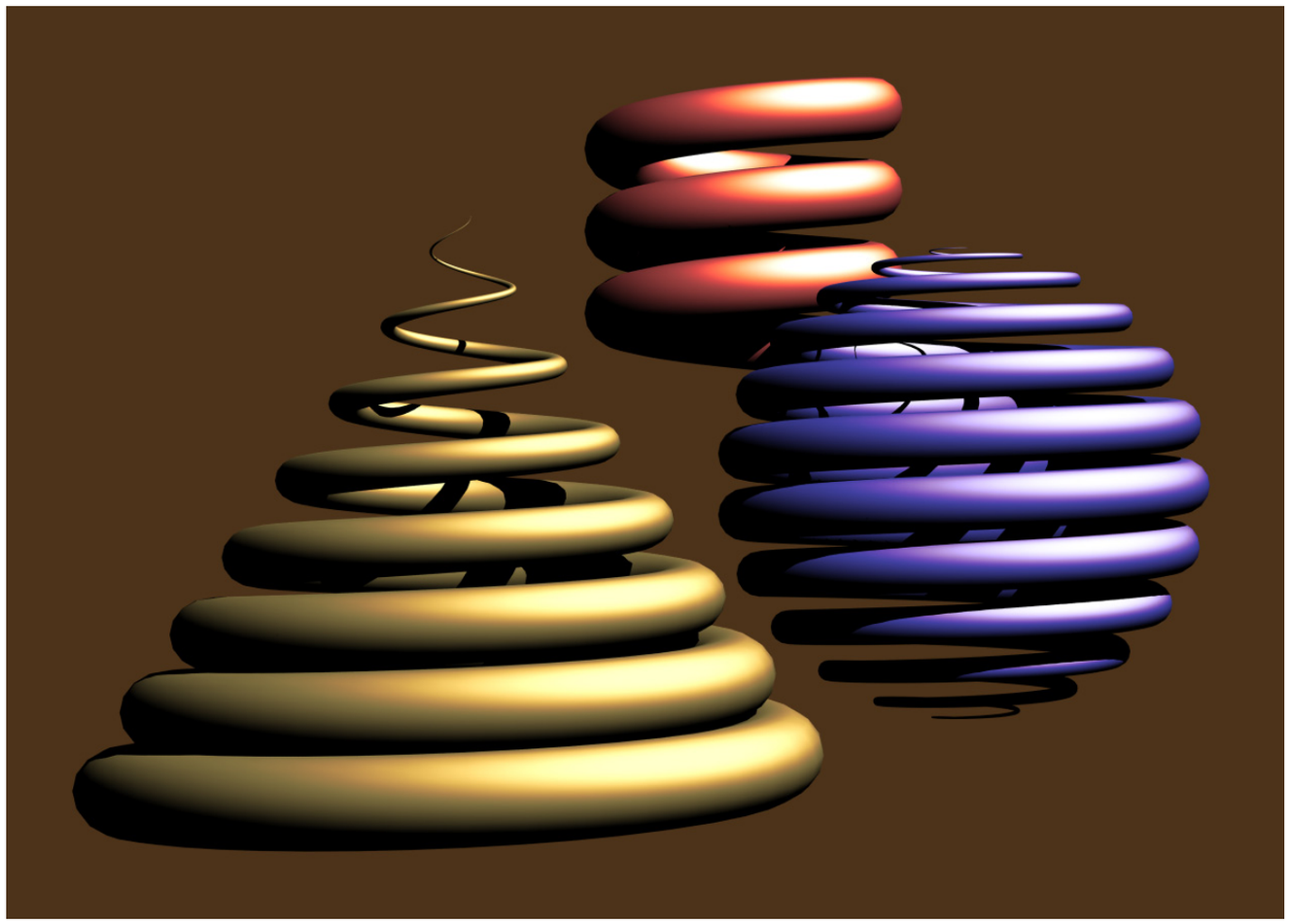
Scene file declaration of shader "mock_specular"

```
 1  void miaux_add_mock_specular_component(
 2      miColor *result, miState *state,
 3      miVector *direction_toward_light,
 4      miColor *specular_color,
 5      miColor *cutoff,
 6      miColor *light_color)
 7  {
 8      miScalar lightdir_offset, r_scale, g_scale, b_scale;
 9      miColor attenuated_specular = {0,0,0,0};
10
11      lightdir_offset = mi_vector_dot(&state->normal, direction_toward_light);
12      r_scale = miaux_sinusoid_fit_clamp(lightdir_offset, cutoff->r, 1.0, 0, 1);
13      g_scale = miaux_sinusoid_fit_clamp(lightdir_offset, cutoff->g, 1.0, 0, 1);
14      b_scale = miaux_sinusoid_fit_clamp(lightdir_offset, cutoff->b, 1.0, 0, 1);
15
16      miaux_scale_channels(&attenuated_specular, specular_color,
17                           r_scale, g_scale, b_scale);
18      miaux_multiply_colors(light_color, light_color, &attenuated_specular);
19      miaux_add_color(result, light_color);
20  }
```

Auxiliary function: miaux_add_mock_specular_component

Light on a surface

Faking the specular component



```
material "yellow"
    "mock_specular" (
        "diffuse" .8 .7 .4,
        "specular" .4 .4 .4,
        "cutoff" .6 .7 .8,
        "lights" ["light_inst"] )
end material

material "blue"
    "mock_specular" (
        "diffuse" .4 .4 1,
        "specular" .6 .6 .6,
        "cutoff" .6 .7 .8,
        "lights" ["light_inst"] )
end material

material "red"
    "mock_specular" (
        "diffuse" 1 .4 .4,
        "specular" .7 .7 .7,
        "cutoff" .6 .7 .8,
        "lights" ["light_inst"] )
end material
```

A specular highlight model that ignores physics

```
 1  struct mock_specular {
 2      miColor ambient;
 3      miColor diffuse;
 4      miColor specular;
 5      miColor cutoff;
 6      int     i_light;
 7      int     n_light;
 8      miTag   light[1];
 9  };
10
11  miBoolean mock_specular (
12      miColor *result, miState *state, struct mock_specular *params )
13  {
     /* ... */
22      miColor *cutoff   = mi_eval_color(&params->cutoff);
     /* ... */
33              miaux_add_mock_specular_component(
34                  &sum, state, &direction_toward_light,
35                  specular, cutoff, &light_color);
     /* ... */
```

Source code of shader "mock_specular"

```
declare shader
    color "rim_bright" (
        color "ambient" default 0 0 0,
        color "diffuse" default 1 1 1 1,
        color "rim"     default 1 1 1,
        array light "lights" )
end declare
```

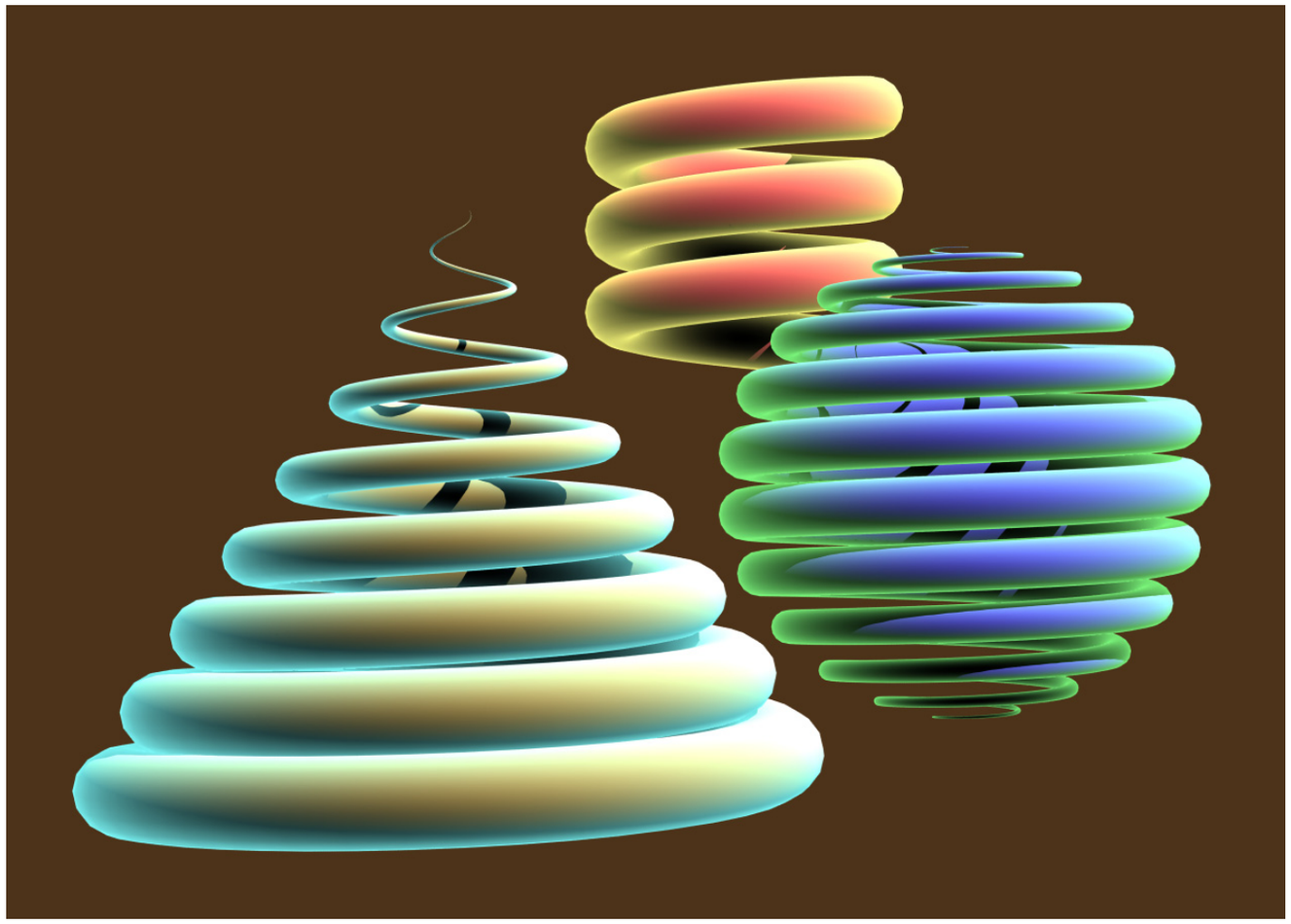
Scene file declaration of shader "rim_bright"

```
1  void miaux_brighten_rim(miColor *result, miState *state,
2                          miColor *rim_color)
3  {
4      miaux_add_scaled_color(result, rim_color, 1.0 + state->dot_nd);
5  }
```

Auxiliary function: miaux_brighten_rim

Light on a surface

Adding arbitrary effects to the Lambert model



```
material "yellow"
    "rim_bright" (
        "diffuse" .8 .7 .4,
        "rim"      .5 1 1,
        "lights" ["light_inst"] )
end material

material "blue"
    "rim_bright" (
        "diffuse" .4 .4 1,
        "rim"      .5 1 .5,
        "lights" ["light_inst"] )
end material

material "red"
    "rim_bright" (
        "diffuse" 1 .4 .4,
        "rim"      1 1 .5,
        "lights" ["light_inst"] )
end material
```

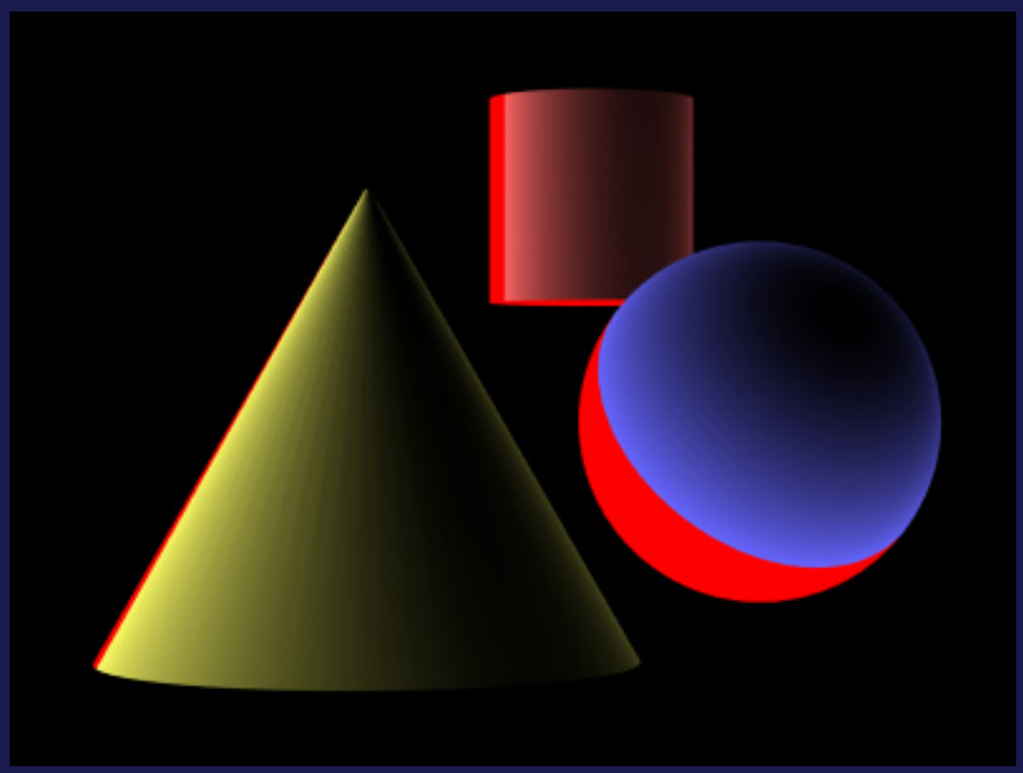
Adding color to the rim of the object

```
 1  struct rim_bright {
 2      miColor ambient;
 3      miColor diffuse;
 4      miColor rim;
 5      int     i_light;
 6      int     n_light;
 7      miTag   light[1];
 8  };
 9
10  miBoolean rim_bright (
11      miColor *result, miState *state, struct rim_bright *params )
12  {
     /* ... */
20      miColor *rim = mi_eval_color(&params->rim);
     /* ... */
35      miaux_brighten_rim(result, state, rim);
36      return miTRUE;
37  }
```

Source code of shader "rim_bright"

Exercise 11: Illumination shaders

1. Copy `shading_1.mi` to `shading.mi`, change output to `shading.tif` and render.
2. Desaturate all the colors in the materials and re-render.
3. Compare `lambert.c` to `phong.c` and find where the specular component is added to the diffuse component.
4. Modify shader file `lambert.c` so that the light value is inverted. If no light is striking the surface, make it red.



Exercise 11: Illumination shaders (part 2)

Modify shader file `lambert.c` so that the light value is inverted. If no light is striking the surface, make it red.

Old:

```
while (mi_sample_light(&light_color, NULL, &dot_nl,
                       state, *light, &light_sample_count))
    miaux_add_diffuse_component(&sum, dot_nl, diffuse, &light_color);
if (light_sample_count)
    miaux_add_scaled_color(result, &sum, 1.0/light_sample_count);
```

New:

```
while (mi_sample_light(&light_color, NULL, &dot_nl,
                       state, *light, &light_sample_count))
    miaux_add_diffuse_component(&sum, 1.0-dot_nl, diffuse, &light_color);
if (light_sample_count)
    miaux_add_scaled_color(result, &sum, 1.0/light_sample_count);
else
    result->r = 1.0;
```

Shadows

Shadows

Color shadows without full transparency
Midrange transparency control
Other parameters for the breakpoint function
A shadow shader with continuous transparency change
Combining the material and shadow shader

Shadows

```
 1  struct point_light_shadow {
 2      miColor light_color;
 3  };
 4
 5  miBoolean point_light_shadow (
 6      miColor *result, miState *state, struct point_light_shadow *params)
 7  {
 8      *result = *mi_eval_color(&params->light_color);
 9      return mi_trace_shadow(result, state);
10  }
```

Point light with shadowing

Shadows

```
     /* ... */
25           while (mi_sample_light(&light_color, NULL, &dot_nl,
26                                  state, *light, &light_sample_count))
27               miaux_add_diffuse_component(&sum, dot_nl, diffuse, &light_color);
28           if (light_sample_count)
29               miaux_add_scaled_color(result, &sum, 1.0/light_sample_count);
     /* ... */
```

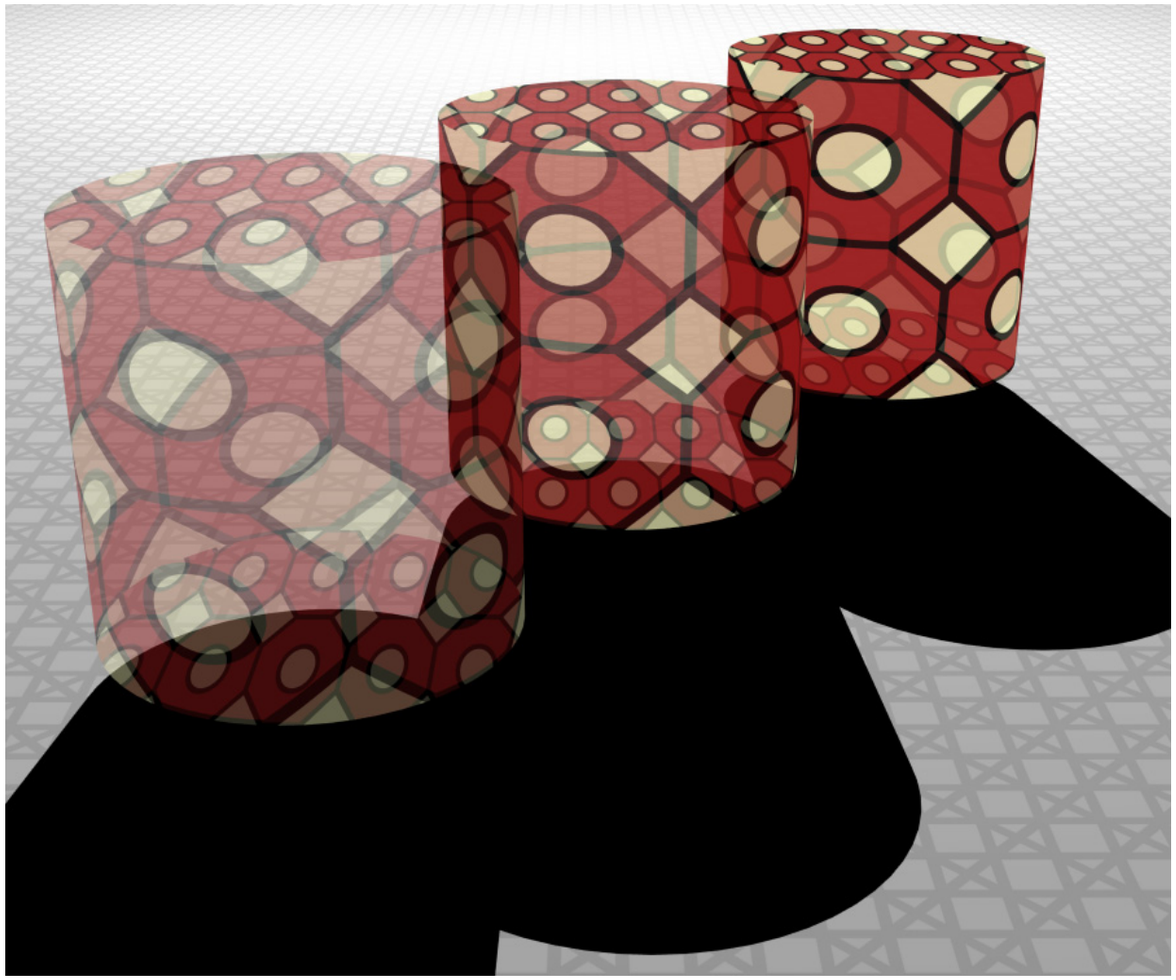
Function mi_sample_light in the context of the light loop in the lambert shader

Shadows

```
1  miBoolean shadow_default (
2      miColor *result, miState *state, void *params )
3  {
4      result->r = result->g = result->b = 0.0;
5      return miFALSE;
6  }
```

Source code of shader "shadow_default"

Shadows



```
material "transparent_75"
    "transparent" (
        "color" = "tile_shader",
        "transparency" .75 .75 .75 )
    shadow
        "shadow_default" ()
end material
```

Material description code{transparent_75} for cylinder on left

Color shadows without full transparency

```
declare shader
    color "shadow_color" (
        color "color" default 1 1 1,
        color "transparency" default .5 .5 .5 )
end declare
```

Scene file declaration of shader "shadow_color"

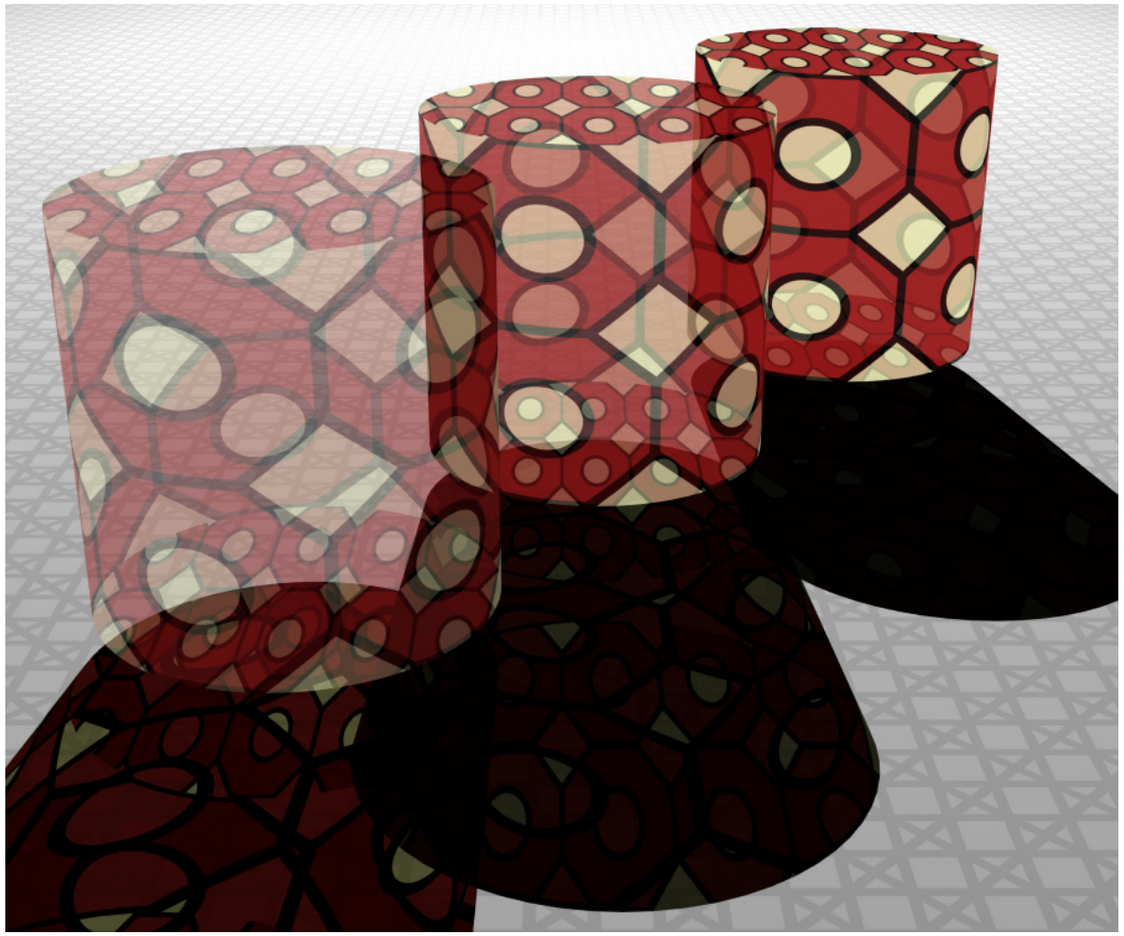
Color shadows without full transparency

```
 1  /*
 2     This is an incorrect calculation of the shadow color -- you can't
 3     just multiply the original light color by the transparency and
 4     surface colors.
 5  */
 6
 7  #include <shader.h>
 8  #include "miaux.h"
 9
10  struct shadow_color {
11      miColor color;
12      miColor transparency;
13  };
14
15  miBoolean shadow_color (
16      miColor *result, miState *state, struct shadow_color *params )
17  {
18      miColor *color = mi_eval_color(&params->color);
19      miColor *transparency = mi_eval_color(&params->transparency);
20
21      result->r *= color->r * transparency->r;
22      result->g *= color->g * transparency->g;
23      result->b *= color->b * transparency->b;
24
25      return miaux_all_channels_equal(result, 0.0) ? miFALSE : miTRUE;
26  }
```

Source code of shader "shadow_color"

Shadows

Color shadows without full transparency

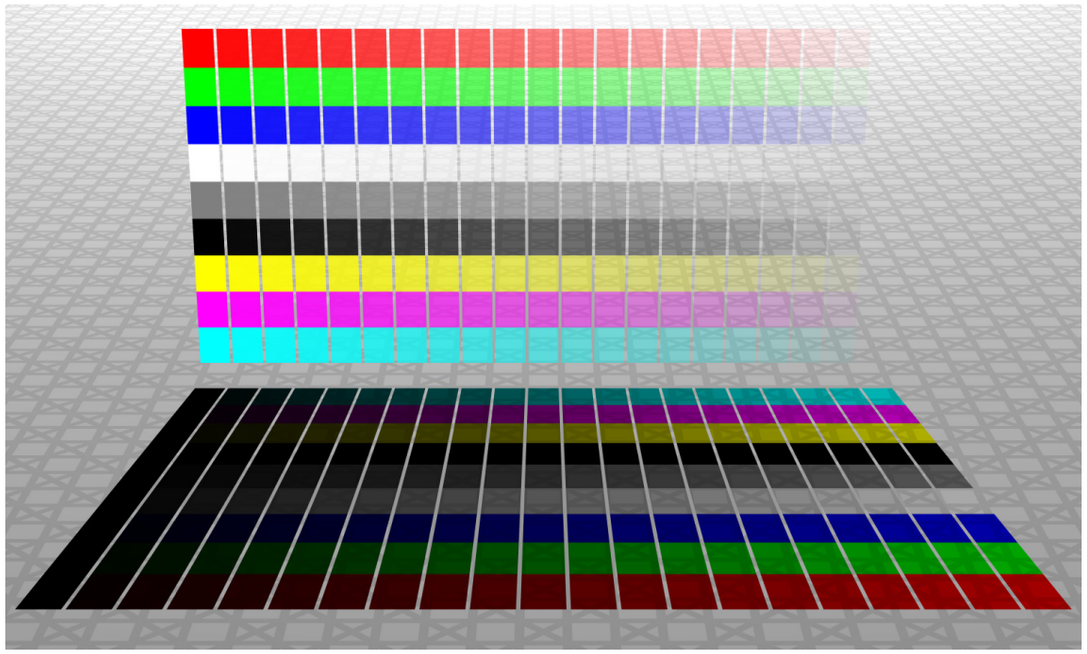


```
material "transparent_25"
    "transparent" (
        "color" = "tile_shader",
        "transparency" .25 .25 .25 )
    shadow
        "shadow_color" (
            "color" = "tile_shader",
            "transparency" .25 .25 .25 )
end material

material "transparent_50"
    "transparent" (
        "color" = "tile_shader",
        "transparency" .5 .5 .5 )
    shadow
        "shadow_color" (
            "color" = "tile_shader",
            "transparency" .5 .5 .5 )
end material

material "transparent_75"
    "transparent" (
        "color" = "tile_shader",
        "transparency" .75 .75 .75 )
    shadow
        "shadow_color" (
            "color" = "tile_shader",
            "transparency" .75 .75 .75 )
end material
```

Transparent shadows with apparent inconsistency in higher transparency values

Shadows

Color shadows without full transparency

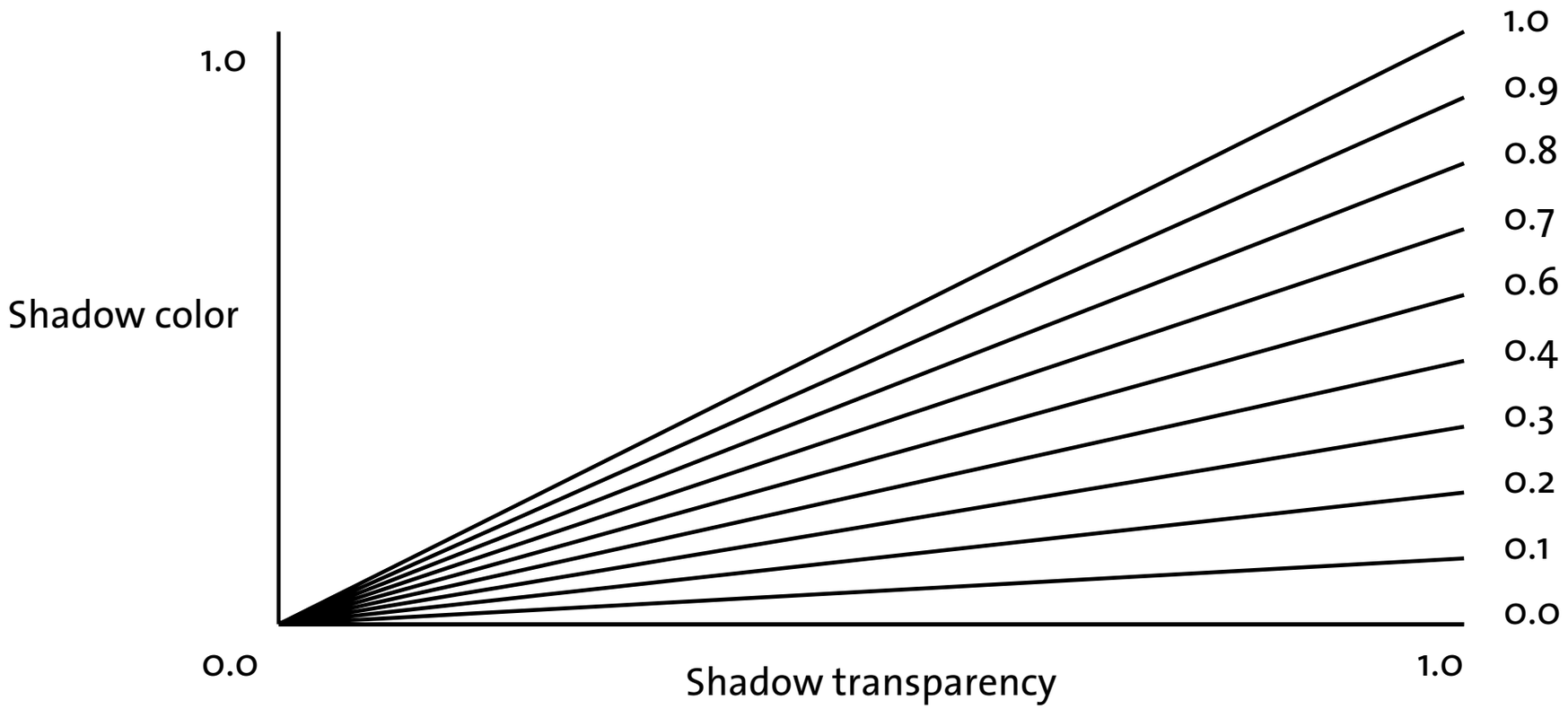


```
material "shadow_transparent_45"
    "transparent" (
        "color" = "grid",
        "transparency" 0.45 0.45 0.45 )
    shadow
        "shadow_color" (
            "color" = "grid",
            "transparency" 0.45 0.45 0.45 )
end material
```

Creating an incremental ramp of transparency values to determine the accuracy of a shader

Shadows

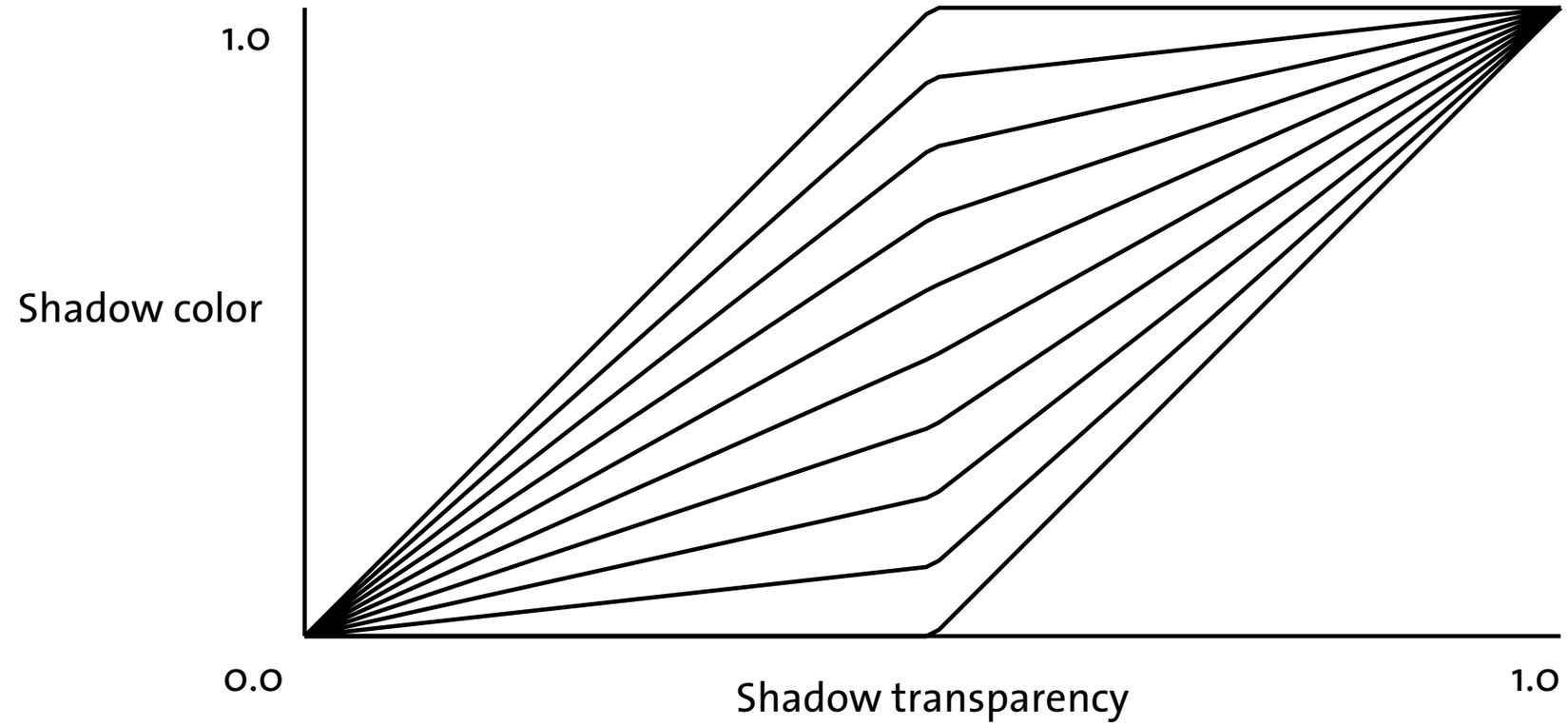
Color shadows without full transparency



Product of the shadow color and shadow transparency

Shadows

Midrange transparency control



Light attenuation with transition at breakpoint

```
1  double miaux_shadow_breakpoint (
2      double color, double transparency, double breakpoint )
3  {
4      if (transparency < breakpoint)
5          return miaux_fit(transparency, 0, breakpoint, 0, color);
6      else
7          return miaux_fit(transparency, breakpoint, 1, color, 1);
8  }
```

Auxiliary function: miaux_shadow_breakpoint

Midrange transparency control

```
declare shader
    color "shadow_breakpoint" (
        color "color" default 1 1 1,
        color "transparency" default .5 .5 .5,
        scalar "breakpoint" default .5 )
end declare
```

Scene file declaration of shader "shadow_breakpoint"

Shadows

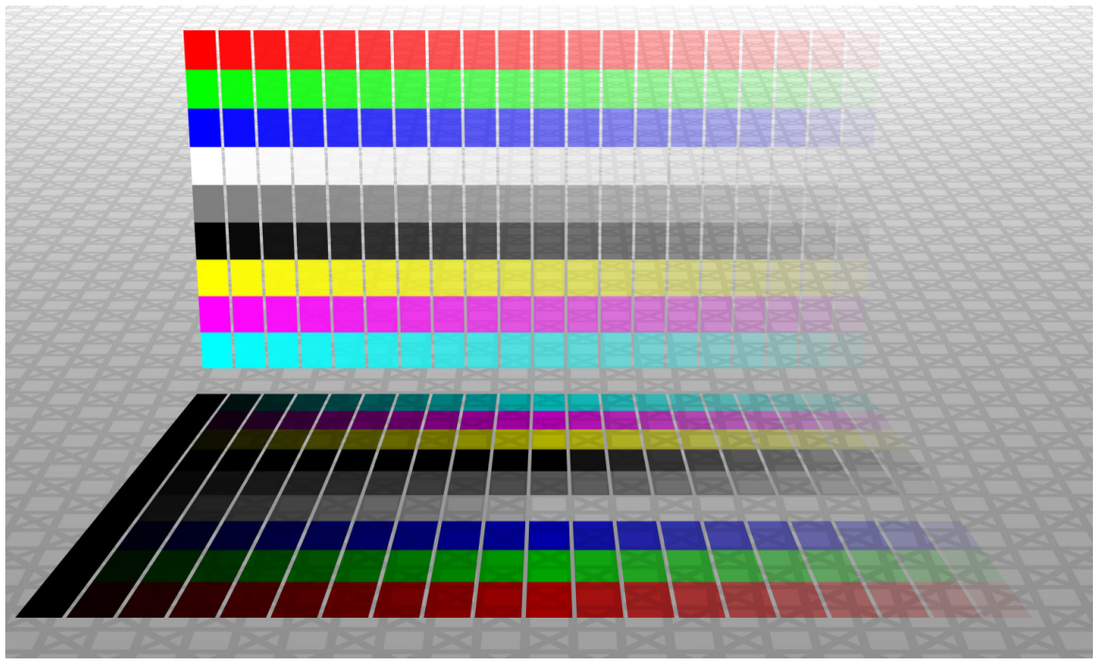
Midrange transparency control

```
 1  struct shadow_breakpoint {
 2      miColor color;
 3      miColor transparency;
 4      miScalar breakpoint;
 5  };
 6
 7  miBoolean shadow_breakpoint (
 8      miColor *result, miState *state, struct shadow_breakpoint *params )
 9  {
10      miColor *color = mi_eval_color(&params->color);
11      miColor *transparency = mi_eval_color(&params->transparency);
12      miScalar breakpoint = *mi_eval_scalar(&params->breakpoint);
13
14      result->r *= miaux_shadow_breakpoint(color->r, transparency->r, breakpoint);
15      result->g *= miaux_shadow_breakpoint(color->g, transparency->g, breakpoint);
16      result->b *= miaux_shadow_breakpoint(color->b, transparency->b, breakpoint);
17
18      return miaux_all_channels_equal(result, 0.0) ? miFALSE : miTRUE;
19  }
```

Source code of shader "shadow_breakpoint"

Shadows

Midrange transparency control

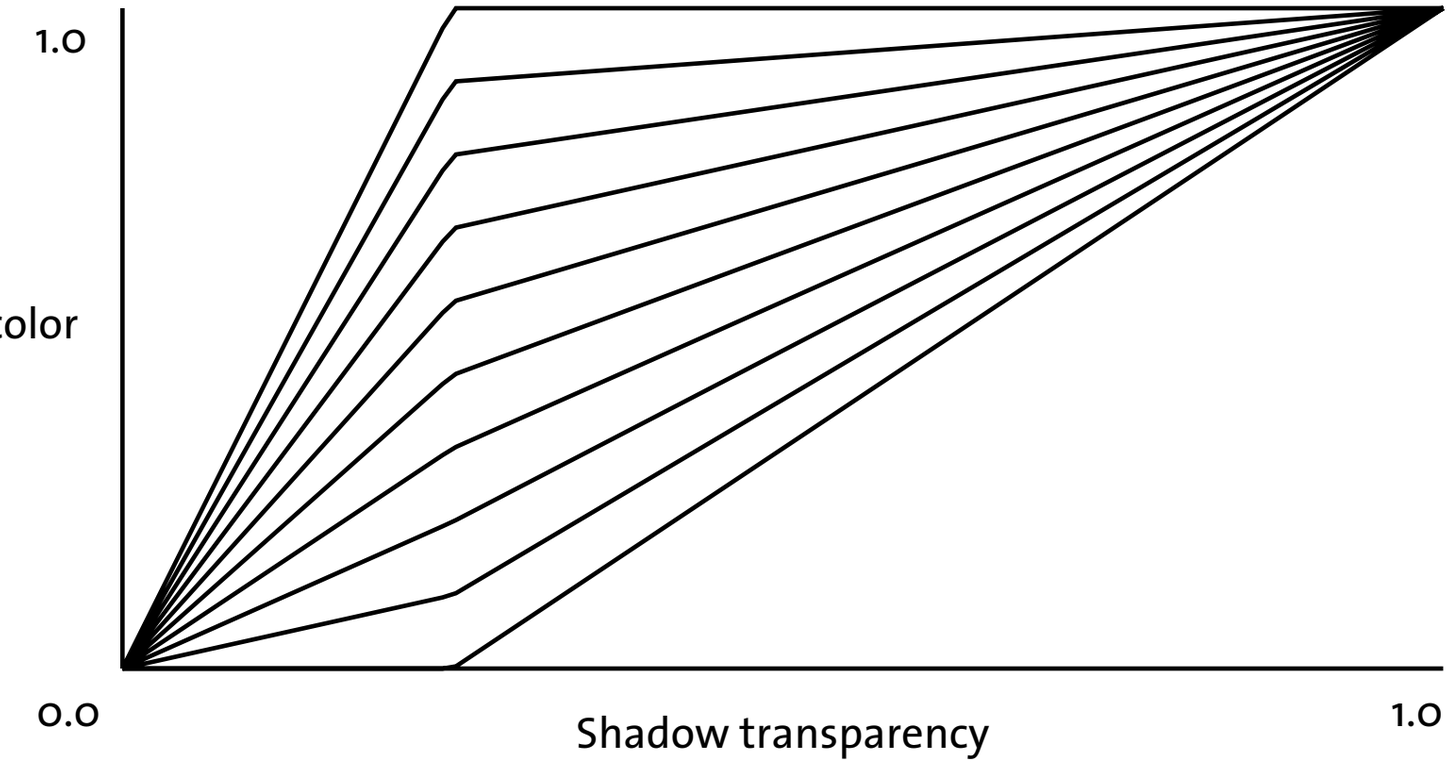


```
material "shadow_transparent_45"
    "transparent" (
        "color" = "grid",
        "transparency" 0.45 0.45 0.45 )
    shadow
        "shadow_breakpoint" (
            "color" = "grid",
            "transparency" 0.45 0.45 0.45,
            "breakpoint" 0.5 )
end material
```

Shadows with a correct handling of transparency value of 1.0

Shadows

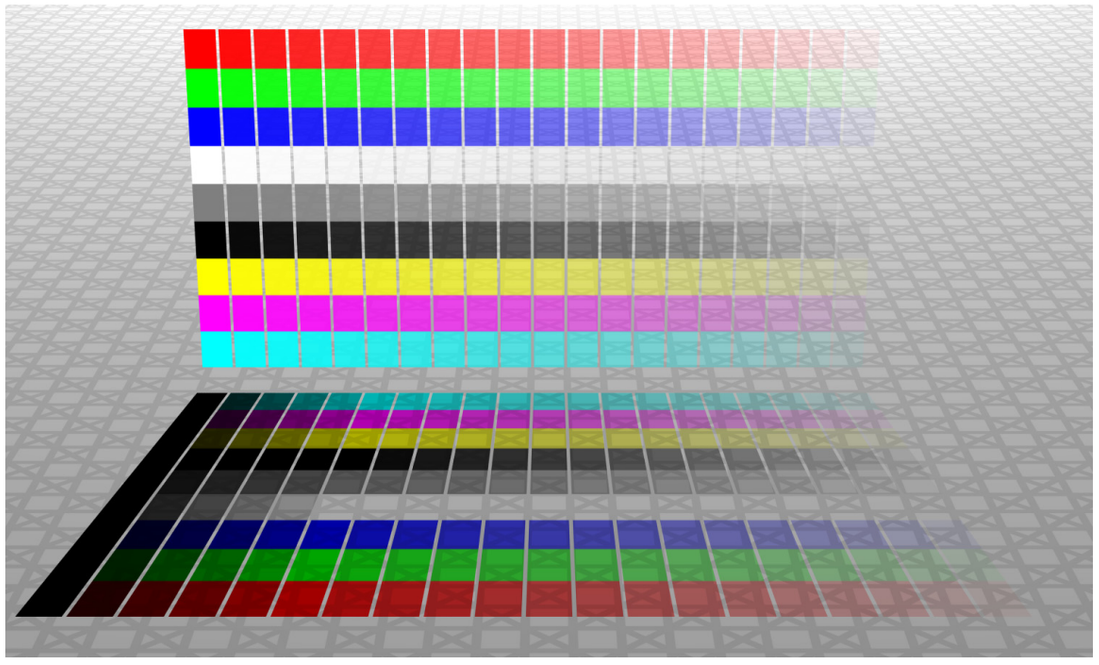
Midrange transparency control



Light attenuation with breakpoint at 25% transparency

Shadows

Midrange transparency control



```
material "shadow_transparent_45"
    "transparent" (
        "color" = "grid",
        "transparency" 0.45 0.45 0.45 )
    shadow
        "shadow_breakpoint" (
            "color" = "grid",
            "transparency" 0.45 0.45 0.45,
            "breakpoint" .25 )
end material
```

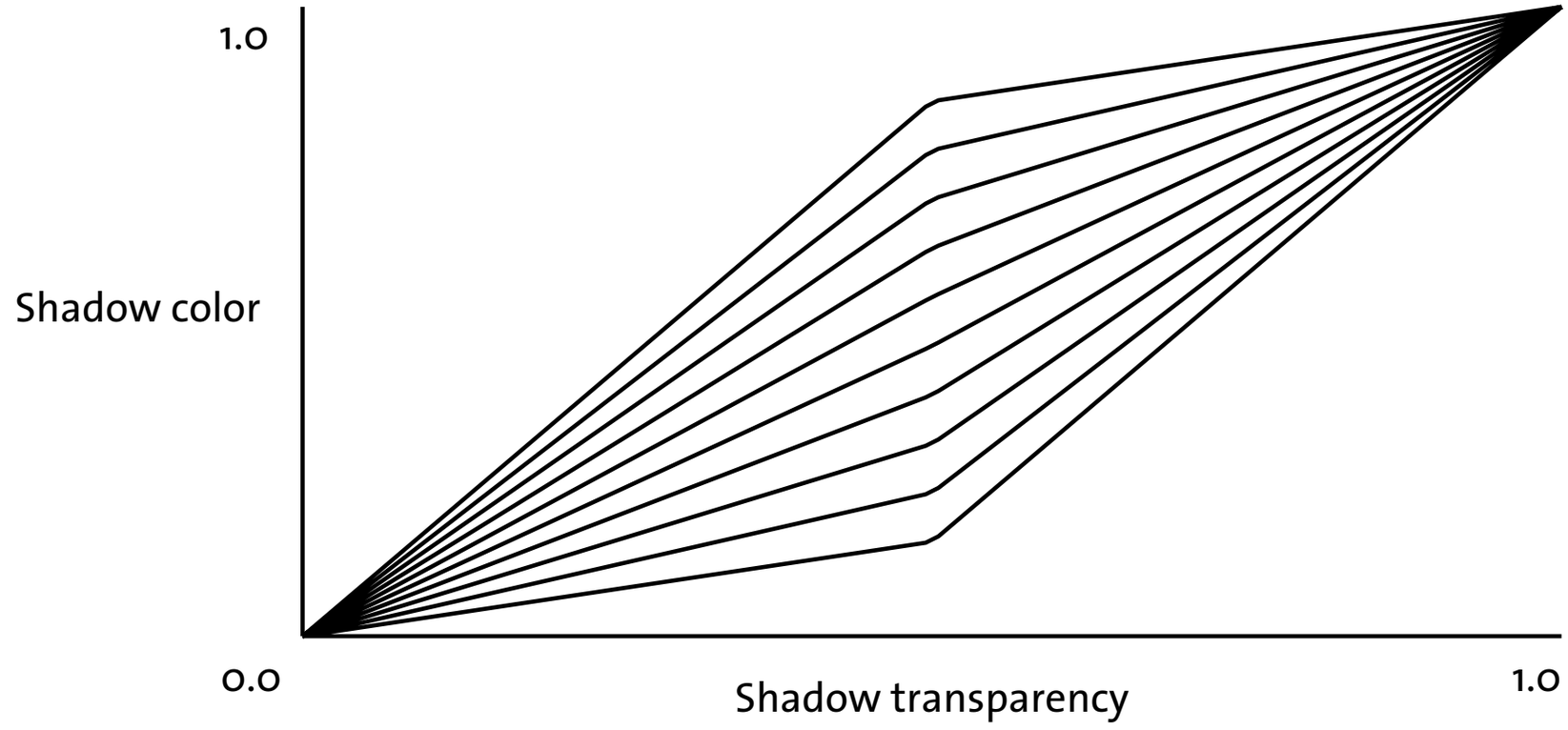
Adjusting the breakpoint of the transition to transparency value 1.0

Other parameters for the breakpoint function

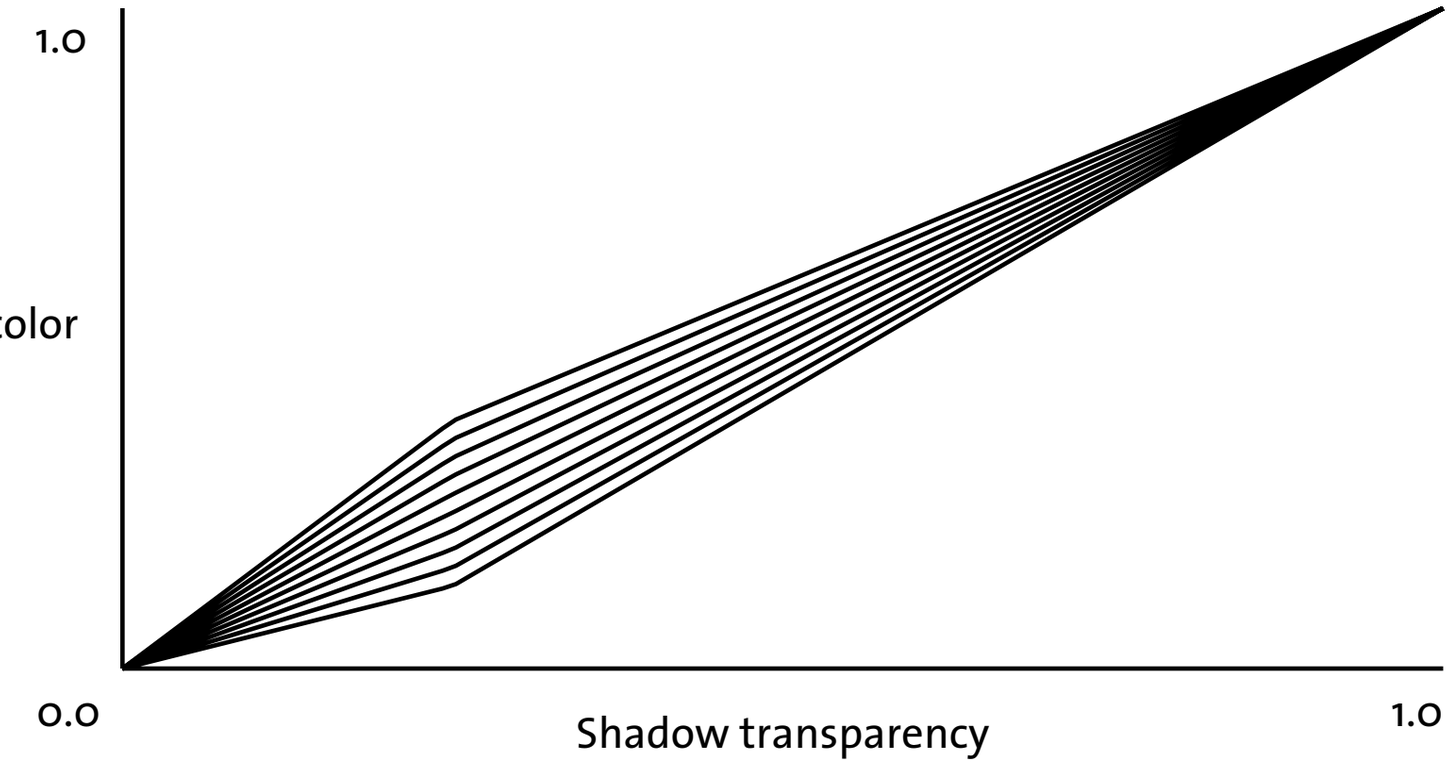
```
1  double miaux_shadow_breakpoint_scale (
2      double color, double transparency, double breakpoint,
3      double center, double extent )
4  {
5      double scaled_color =
6          miaux_fit(color, 0, 1, center - extent/2.0, center + extent/2.0);
7      if (transparency < breakpoint)
8          return miaux_fit(transparency, 0, breakpoint, 0, scaled_color);
9      else
10         return miaux_fit(transparency, breakpoint, 1, scaled_color, 1);
11 }
```

Auxiliary function: miaux_shadow_breakpoint_scale

Other parameters for the breakpoint function



Light attenuation with breakpoint=.5, center=.5, extent=.7



Light attenuation with breakpoint=.25, center=.25, extent=.25

Other parameters for the breakpoint function

```
declare shader
    color "shadow_breakpoint_scale" (
        color "color" default 1 1 1,
        color "transparency" default .5 .5 .5,
        scalar "breakpoint" default .5,
        scalar "center" default .5,
        scalar "extent" default 1 )
end declare
```

Scene file declaration of shader "shadow_breakpoint_scale"

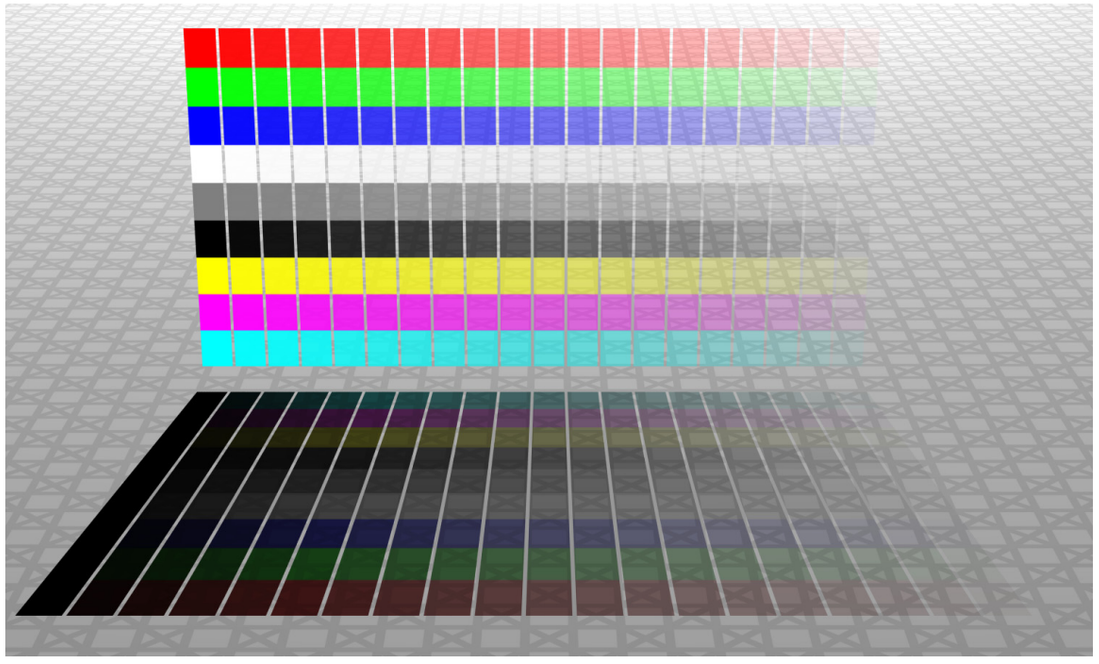
Other parameters for the breakpoint function

```
 1  struct shadow_breakpoint_scale {
 2      miColor color;
 3      miColor transparency;
 4      miScalar breakpoint;
 5      miScalar center;
 6      miScalar extent;
 7  };
 8
 9  miBoolean shadow_breakpoint_scale (
10      miColor *result, miState *state, struct shadow_breakpoint_scale *params )
11  {
12      miColor *color = mi_eval_color(&params->color);
13      miColor *transparency = mi_eval_color(&params->transparency);
14      miScalar breakpoint = *mi_eval_scalar(&params->breakpoint);
15      miScalar center = *mi_eval_scalar(&params->center);
16      miScalar extent = *mi_eval_scalar(&params->extent);
17
18      result->r *= miaux_shadow_breakpoint_scale(color->r, transparency->r,
19                                                 breakpoint, center, extent);
20      result->g *= miaux_shadow_breakpoint_scale(color->g, transparency->g,
21                                                 breakpoint, center, extent);
22      result->b *= miaux_shadow_breakpoint_scale(color->b, transparency->b,
23                                                 breakpoint, center, extent);
24
25      return miaux_all_channels_equal(result, 0.0) ? miFALSE : miTRUE;
26  }
```

Source code of shader "shadow_breakpoint_scale"

Shadows

Other parameters for the breakpoint function



```
material "shadow_transparent_45"
    "transparent" (
        "color" = "grid",
        "transparency" 0.45 0.45 0.45 )
    shadow
        "shadow_breakpoint_scale" (
            "color" = "grid",
            "transparency" 0.45 0.45 0.45,
            "breakpoint" .25,
            "center" .25,
            "extent" .25 )
end material
```

Setting the breakpoint to 0.25

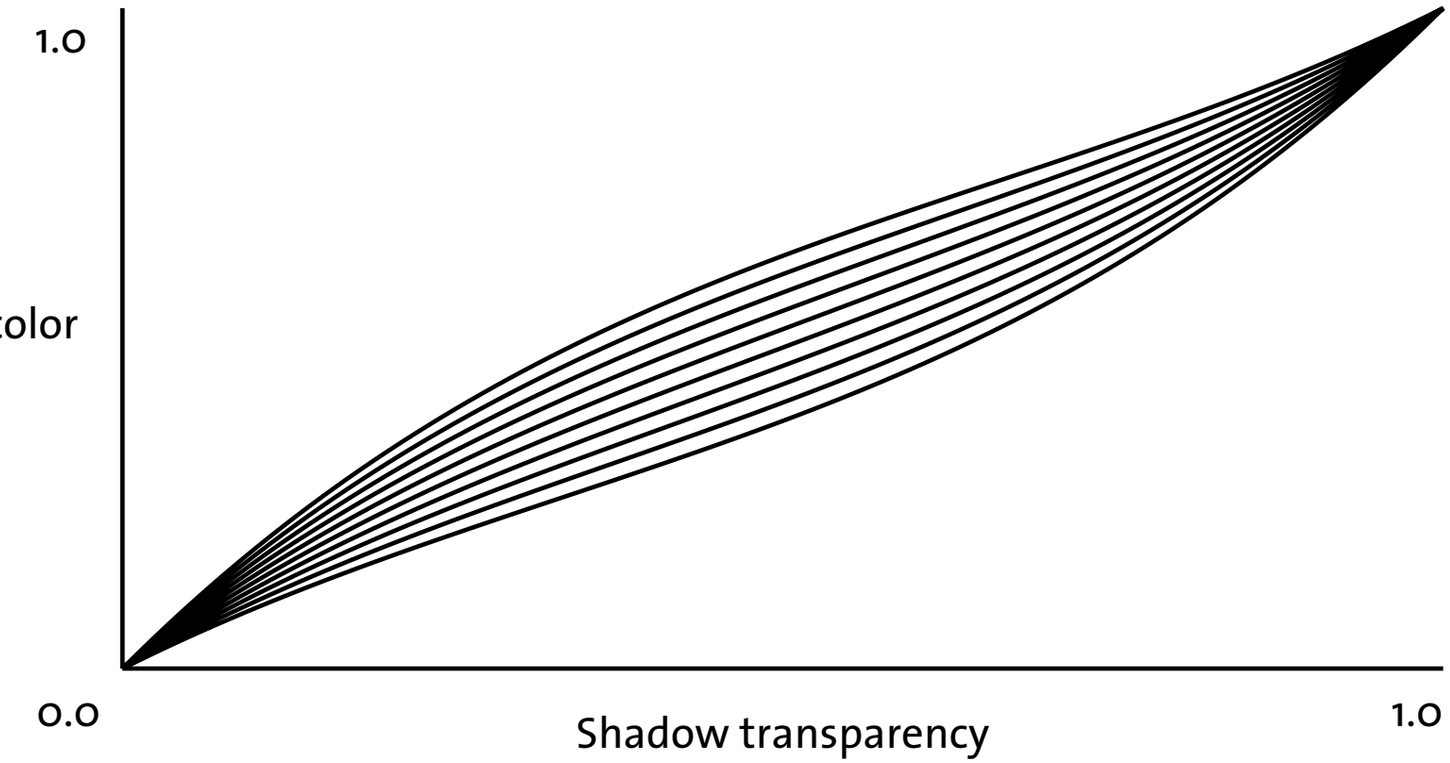
A shadow shader with continuous transparency change

```
1  double miaux_shadow_continuous (
2      double color, double transparency, double expansion )
3  {
4      return transparency
5          + miaux_fit(transparency,
6                      0, 1,
7                      expansion * transparency * (color - transparency), 0);
8  }
```

Auxiliary function: miaux_shadow_continuous

Shadows

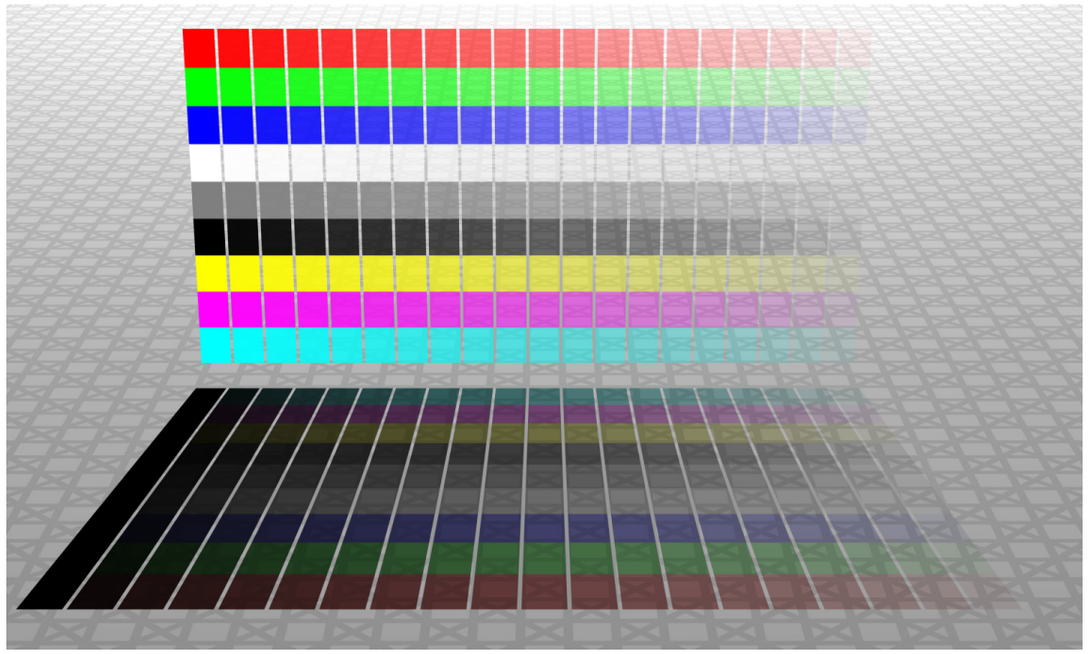
A shadow shader with continuous transparency change



Light attenuation with continuous change

Shadows

A shadow shader with continuous transparency change



```
material "shadow_transparent_45"
    "transparent" (
        "color" = "grid",
        "transparency" 0.45 0.45 0.45 )
    shadow
        "shadow_continuous" (
            "color" = "grid",
            "transparency" 0.45 0.45 0.45 )
end material
```

Shadows with continuous function for transparency

A shadow shader with continuous transparency change

```
declare shader
    color "shadow_continuous" (
        color "color" default 1 1 1,
        color "transparency" default .5 .5 .5,
        scalar "expansion" default 1 )
end declare
```

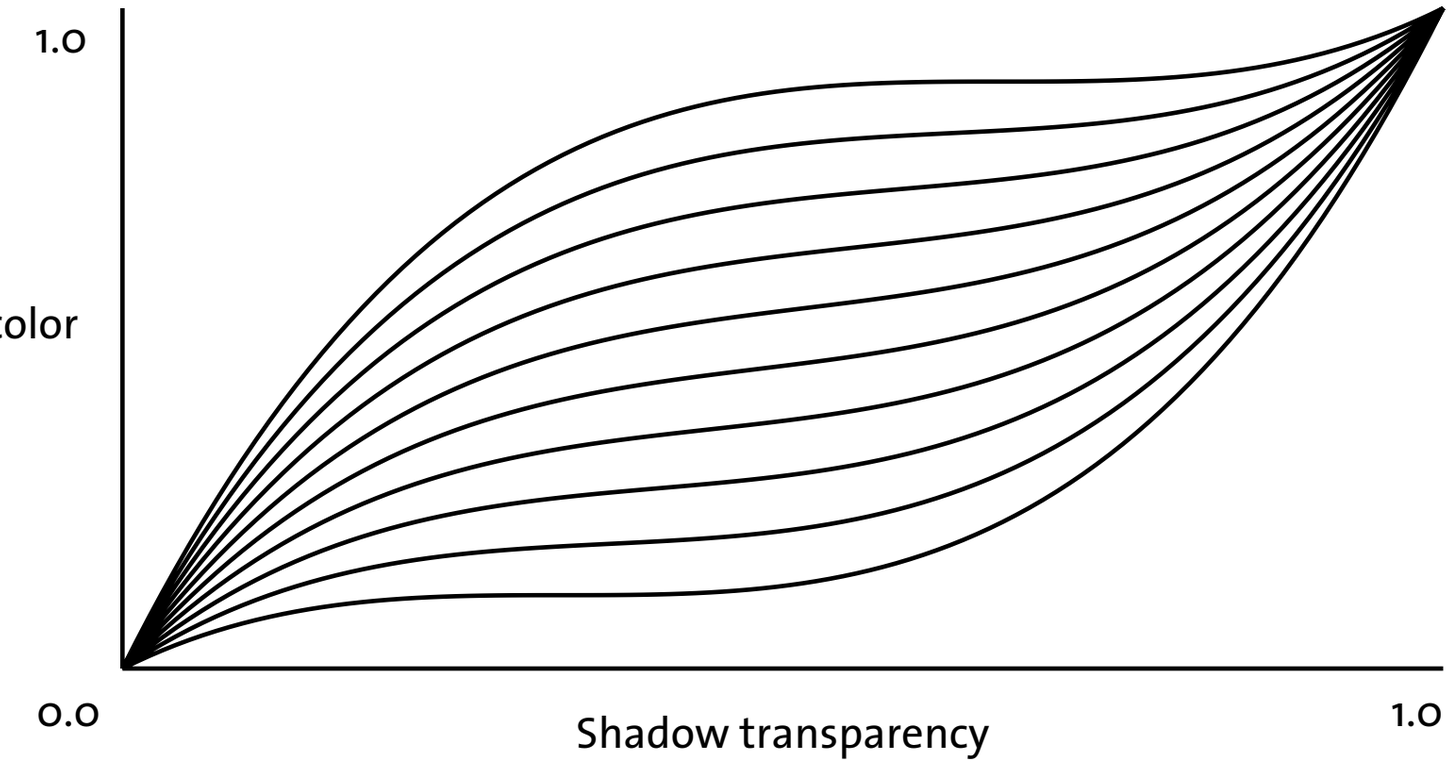
Scene file declaration of shader "shadow_continuous"

A shadow shader with continuous transparency change

```
 1  struct shadow_continuous {
 2      miColor color;
 3      miColor transparency;
 4      miScalar expansion;
 5  };
 6
 7  miBoolean shadow_continuous (
 8      miColor *result,
 9      miState *state,
10      struct shadow_continuous *params )
11  {
12      miColor *color = mi_eval_color(&params->color);
13      miColor *transparency = mi_eval_color(&params->transparency);
14      miScalar expansion = *mi_eval_scalar(&params->expansion);
15
16      result->r *= miaux_shadow_continuous(color->r, transparency->r, expansion);
17      result->g *= miaux_shadow_continuous(color->g, transparency->g, expansion);
18      result->b *= miaux_shadow_continuous(color->b, transparency->b, expansion);
19
20      return miaux_all_channels_equal(result, 0.0) ? miFALSE : miTRUE;
21  }
```

Source code of shader "shadow_continuous"

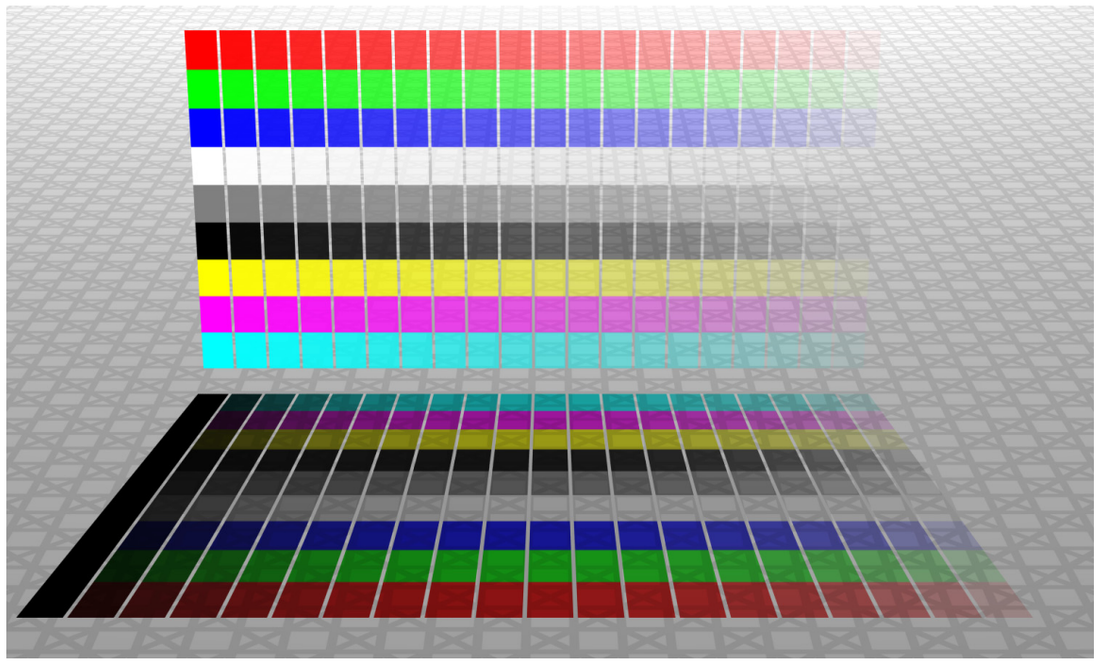
A shadow shader with continuous transparency change



Light attenuation with continuous change with the midrange expanded by 3.0

Shadows

A shadow shader with continuous transparency change



```
material "shadow_transparent_45"
    "transparent" (
        "color" = "grid",
        "transparency" 0.45 0.45 0.45 )
    shadow
        "shadow_continuous" (
            "color" = "grid",
            "transparency" 0.45 0.45 0.45,
            "expansion" 3 )
end material
```

Higher contrast in midrange for transparent shadows

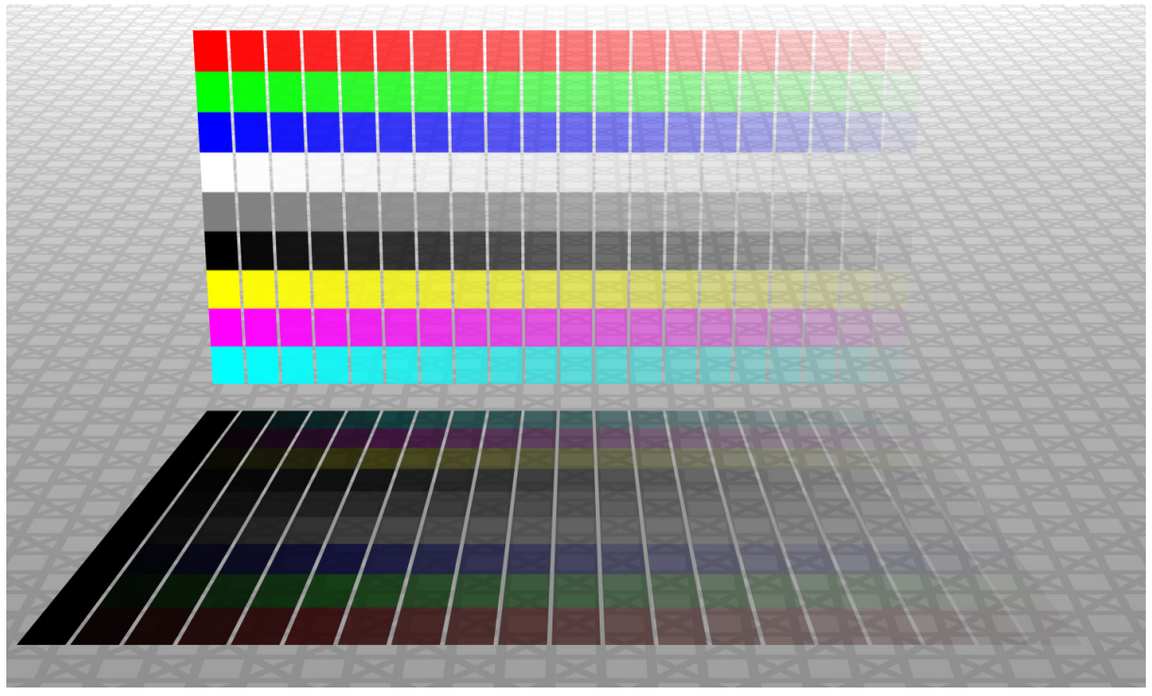
Combining the material and shadow shader

```
declare shader
    color "transparent_shadow" (
        color "color",
        color "transparency",
        scalar "breakpoint",
        scalar "center",
        scalar "extent" )
end declare
```

Scene file declaration of shader "transparent_shadow"

Shadows

Combining the material and shadow shader



```
material "shadow_transparent_45"
    "transparent_shadow" (
        "color" = "grid",
        "transparency" 0.45 0.45 0.45,
        "breakpoint" .25,
        "center" .25,
        "extent" .25 )
    shadow
        "transparent_shadow" ()
end material
```

Using the same shader for both the material and shadow shaders

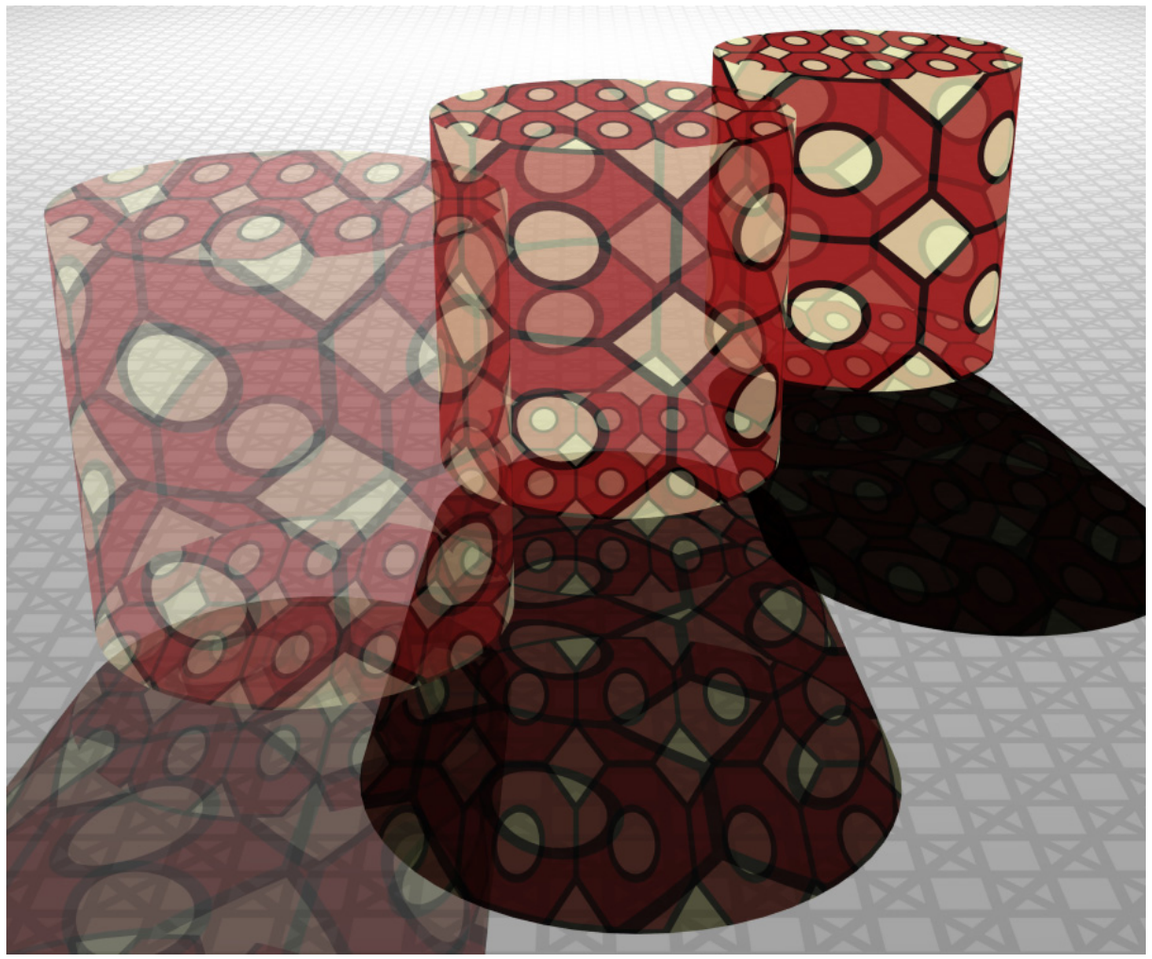
Combining the material and shadow shader

```
 1  struct transparent_shadow {
 2      miColor color;
 3      miColor transparency;
 4      miScalar breakpoint;
 5      miScalar center;
 6      miScalar extent;
 7  };
 8
 9  miBoolean transparent_shadow (
10      miColor *result, miState *state, struct transparent_shadow *params )
11  {
12      miColor *transparency = mi_eval_color(&params->transparency);
13
14      if (state->type == miRAY_SHADOW) {
15          miColor *color = mi_eval_color(&params->color);
16          miScalar breakpoint = *mi_eval_scalar(&params->breakpoint);
17          miScalar center = *mi_eval_scalar(&params->center);
18          miScalar extent = *mi_eval_scalar(&params->extent);
19
20          result->r *= miaux_shadow_breakpoint_scale(color->r, transparency->r,
21                                                     breakpoint, center, extent);
22          result->g *= miaux_shadow_breakpoint_scale(color->g, transparency->g,
23                                                     breakpoint, center, extent);
24          result->b *= miaux_shadow_breakpoint_scale(color->b, transparency->b,
25                                                     breakpoint, center, extent);
26          return miaux_all_channels_equal(result, 0.0) ? miFALSE : miTRUE;
27      }
28      if (miaux_all_channels_equal(transparency, 0.0))
29          *result = *mi_eval_color(&params->color);
30      else {
31          mi_trace_transparent(result, state);
32          if (!miaux_all_channels_equal(transparency, 1.0)) {
33              miColor *color = mi_eval_color(&params->color);
34              miColor opacity;
35              miaux_invert_channels(&opacity, transparency);
36              mi_opacity_set(state, &opacity);
37              miaux_blend_channels(result, color, transparency);
38          }
39      }
40      return miTRUE;
41  }
```

Source code of shader "transparent_shadow"

Shadows

Combining the material and shadow shader



```
material "transparent_25"
    "transparent_shadow" (
        "color" = "tile_shader",
        "transparency" 0.25 0.25 0.25,
        "breakpoint" .5,
        "center" .5,
        "extent" .5 )
    shadow
        "transparent_shadow" ()
end material
```

Shader transparent_shadow with scene from Figure~ref{render-shadows-2}

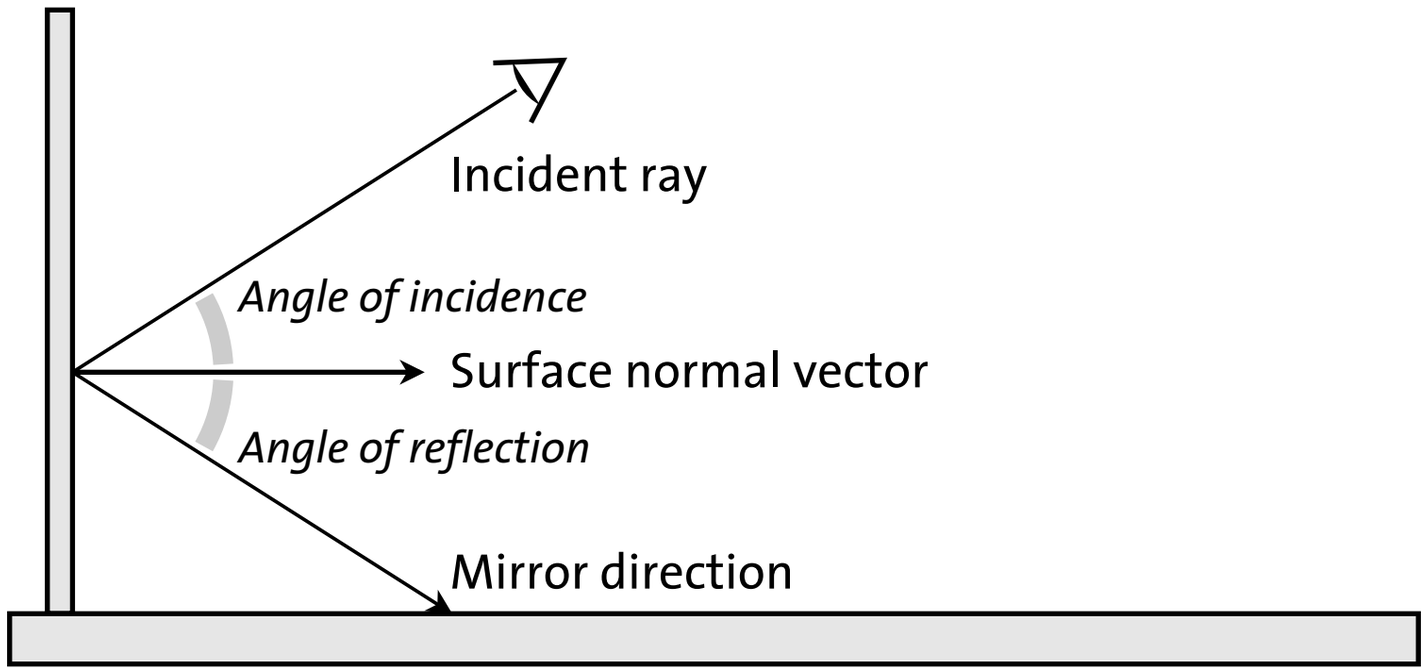
Reflection

Reflection

- Specular reflections
- Reflections and trace depth
- Glossy reflections
- Glossy reflection with multiple samples in the shader
- Glossy reflection with varying sample counts for reflections
- Glossy reflections and object geometry

Reflection

Specular reflections



Specular reflection from a single ray

```
declare shader
    color "specular_reflection" ()
end declare
```

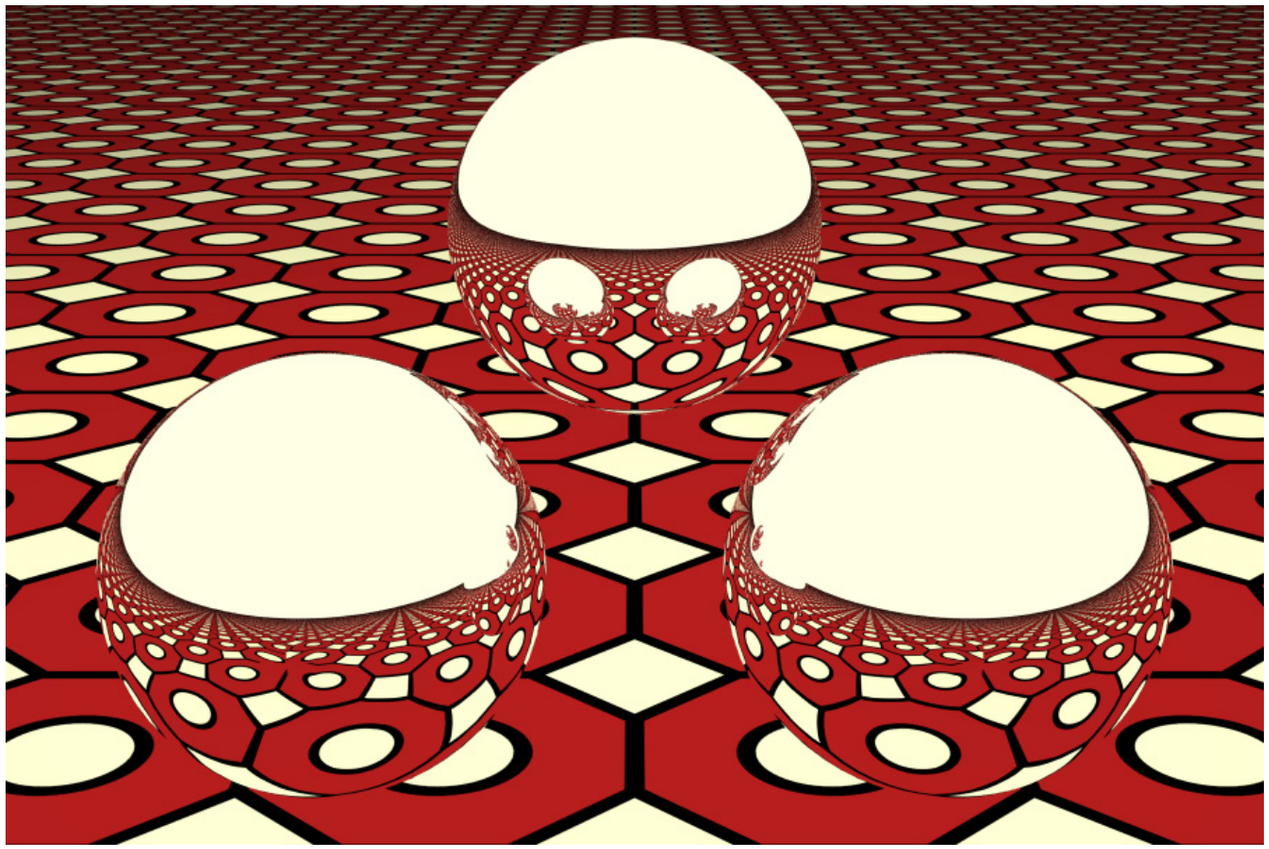
Scene file declaration of shader "specular_reflection"

```
1  miBoolean specular_reflection (
2      miColor *result, miState *state, void *params )
3  {
4      miVector reflection_direction;
5      mi_reflection_dir(&reflection_direction, state);
6
7      if (!mi_trace_reflection(result, state, &reflection_direction))
8          mi_trace_environment(result, state, &reflection_direction);
9
10     return miTRUE;
11 }
```

Source code of shader "specular_reflection"

Reflection

Specular reflections

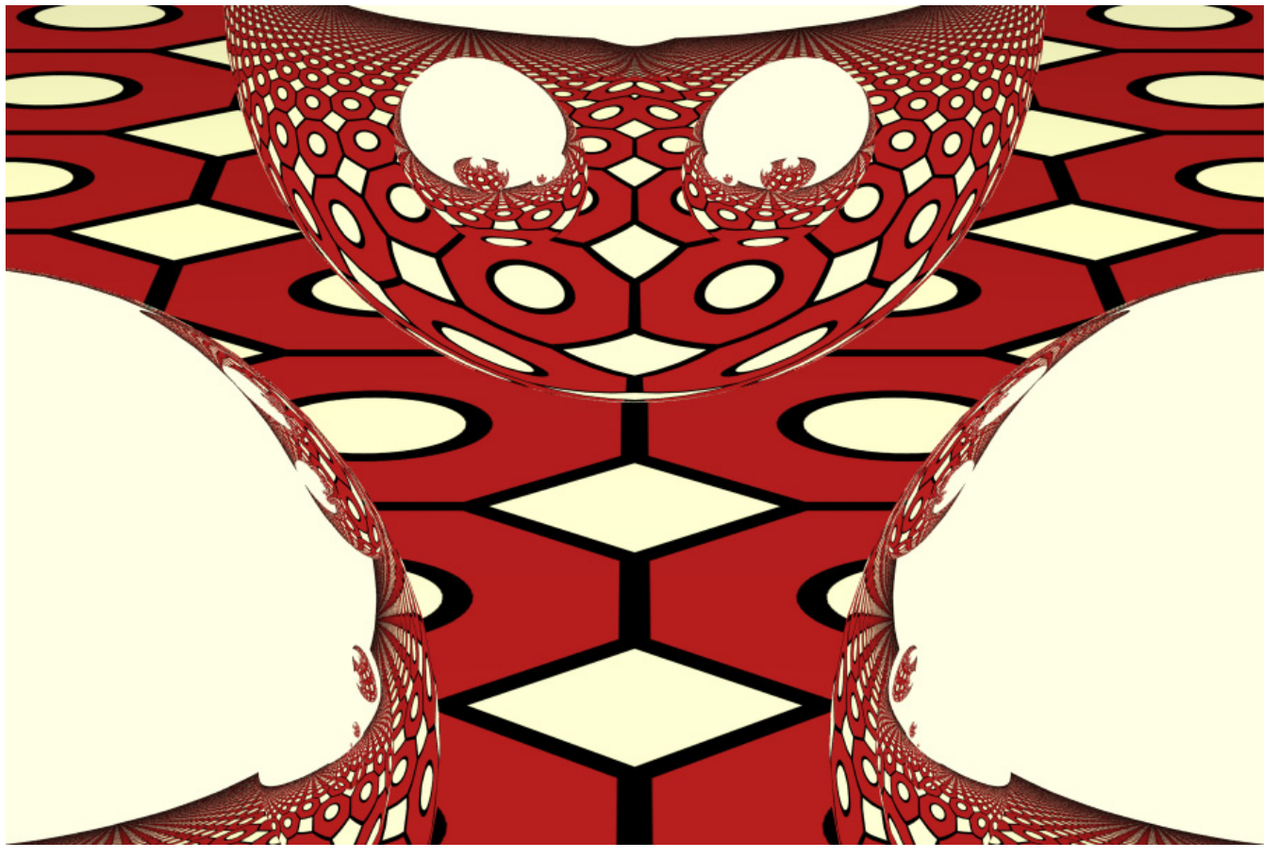


```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5
end options

camera "cam"
    output "rgba" "tif"
        "reflection_1.tif"
    focal 50
    aperture 33.3
    aspect 1.5
    resolution 300 200
    environment
        "one_color" (
            "color" 1 1 .9 )
end camera

material "reflect"
    "specular_reflection" ()
end material
```

Specular reflection

Reflection

Reflections and trace depth

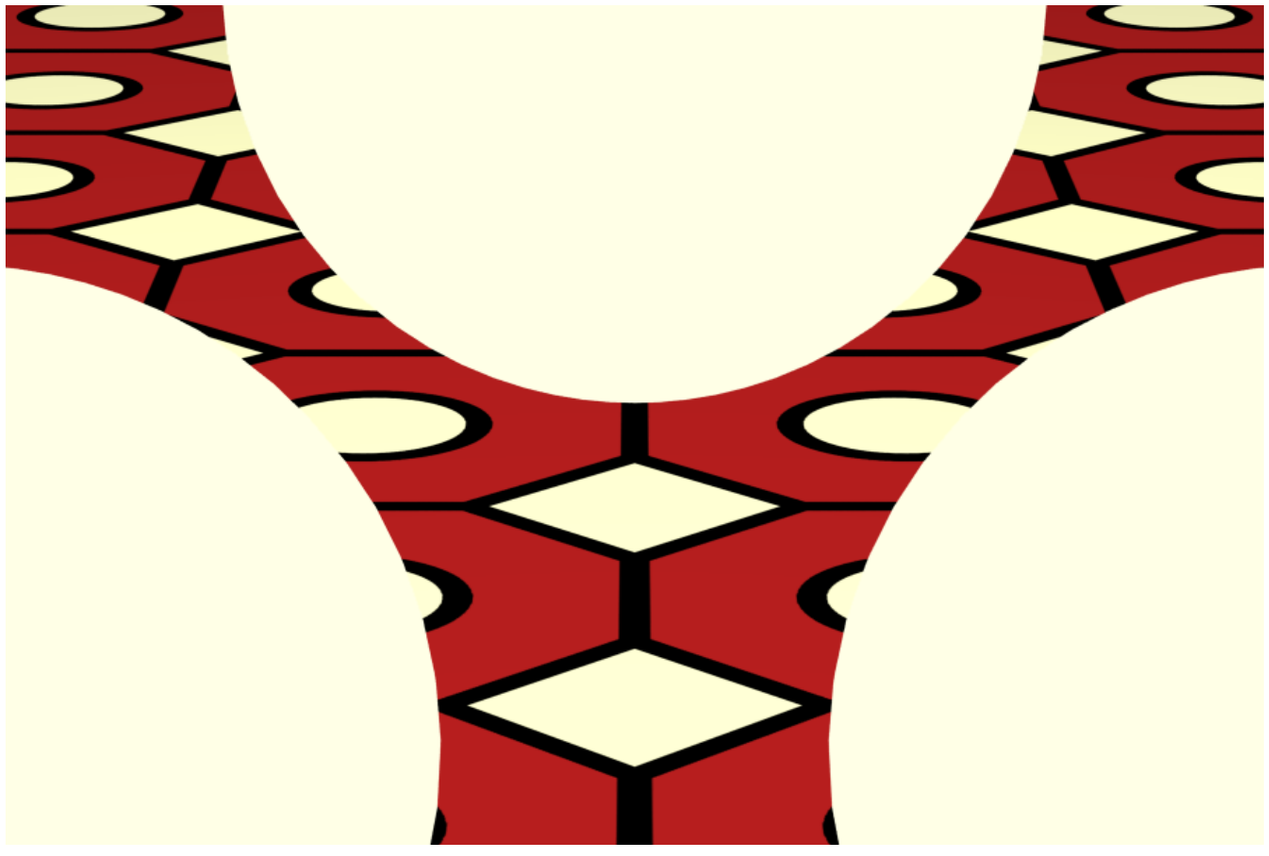


```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5
end options

material "reflect"
    "specular_reflection" ()
end material
```

Specular reflection between three objects

Reflection

Reflections and trace depth

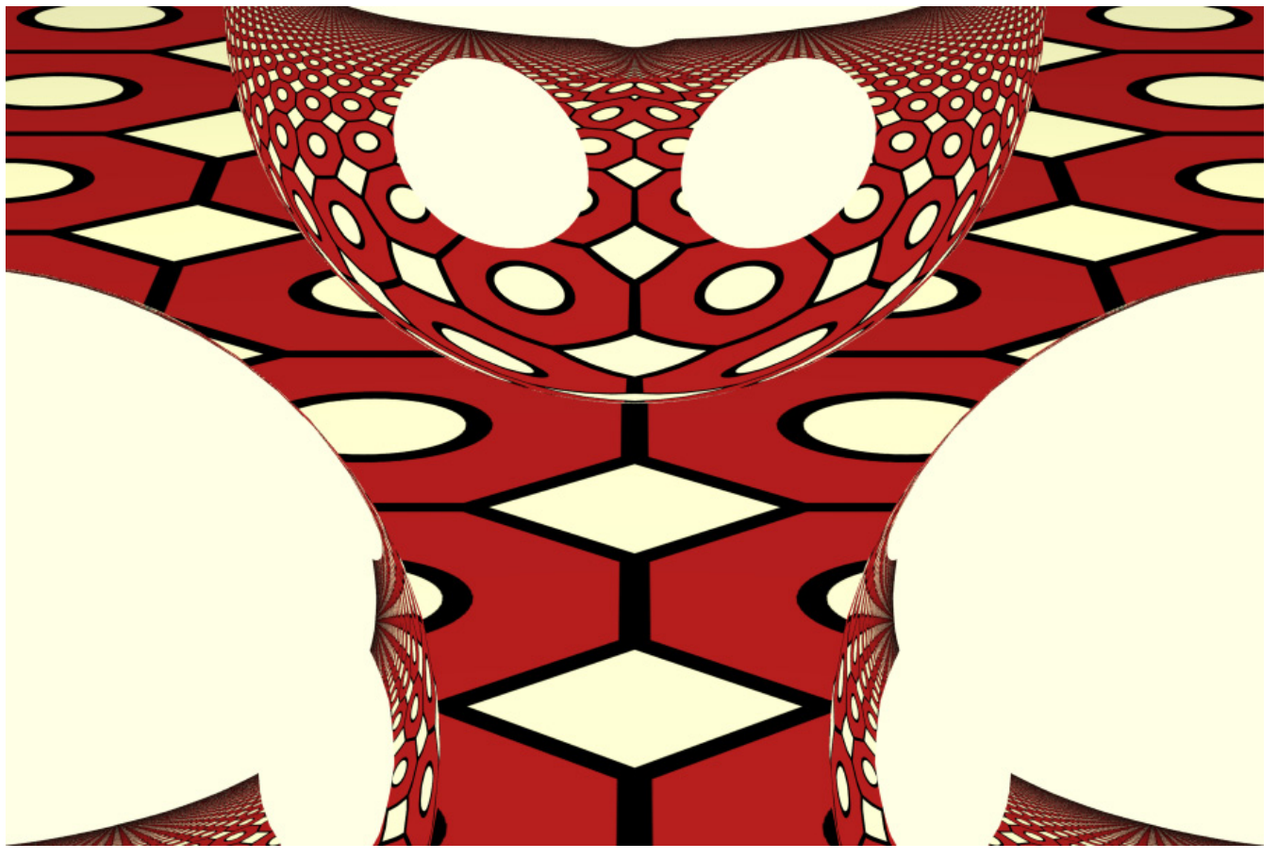


```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 0
end options

material "reflect"
    "specular_reflection" ()
end material
```

Specular reflection disabled from zero trace depth

Reflection

Reflections and trace depth

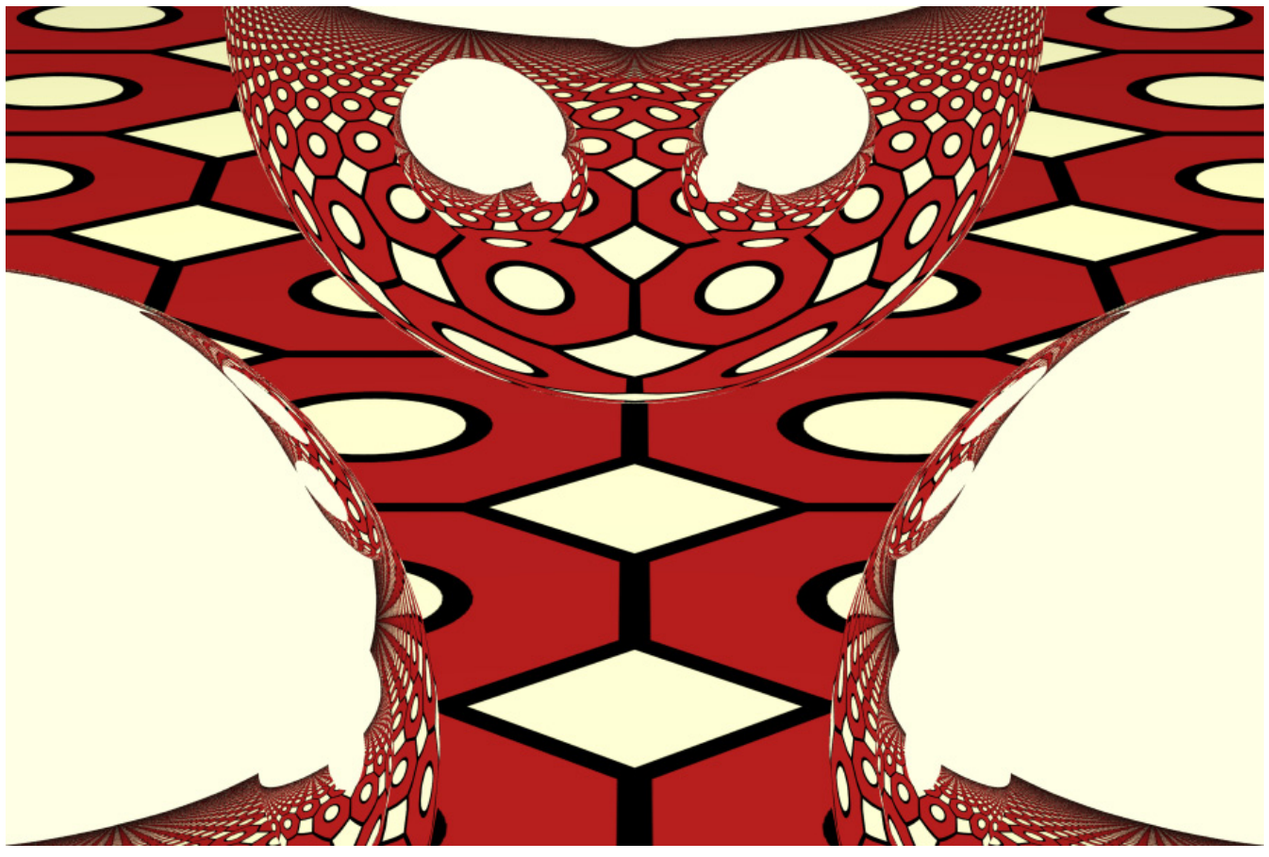


```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 1
end options

material "reflect"
    "specular_reflection" ()
end material
```

Specular reflection with trace depth value of 1: no inter-reflection

Reflection

Reflections and trace depth

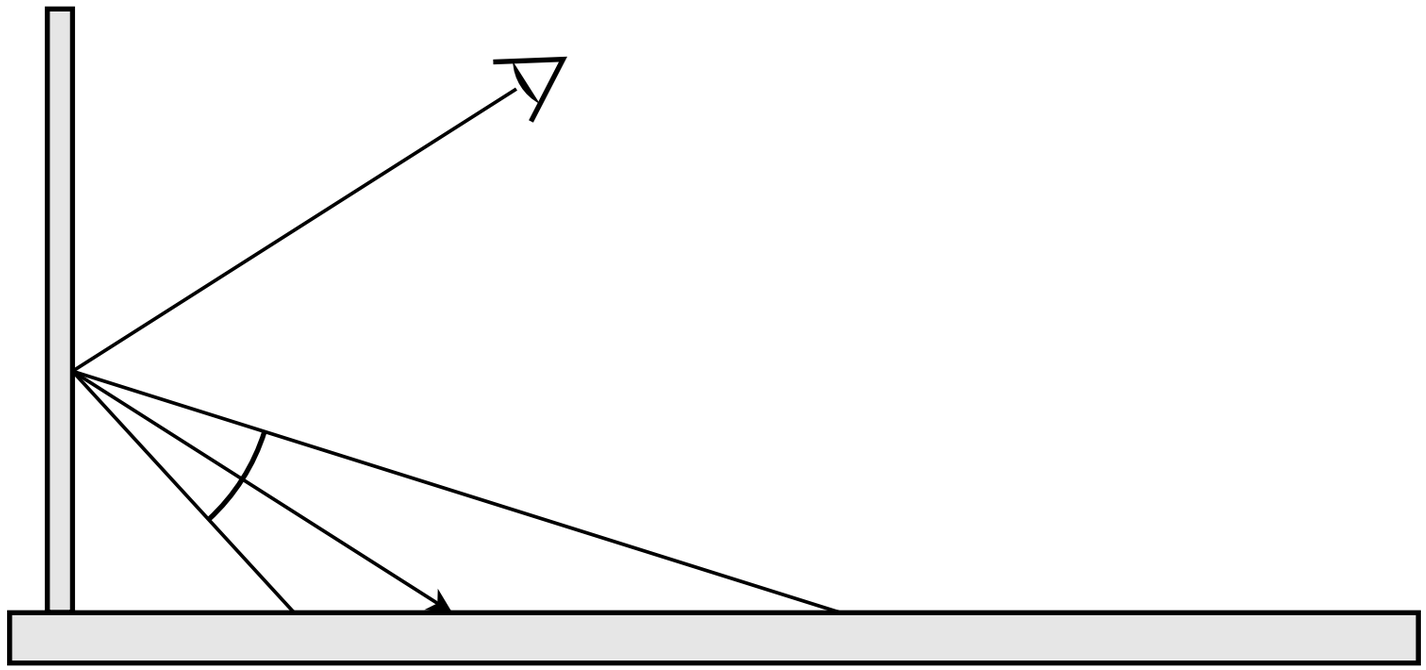


```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 2
end options

material "reflect"
    "specular_reflection" ()
end material
```

Specular reflection with trace depth value of 2: single inter-reflection

Reflection

Glossy reflections



Reflection directions chosen within a cone for a glossy appearance

```
declare shader
    color "glossy_reflection" (
         scalar "shiny" default 5 )
end declare
```

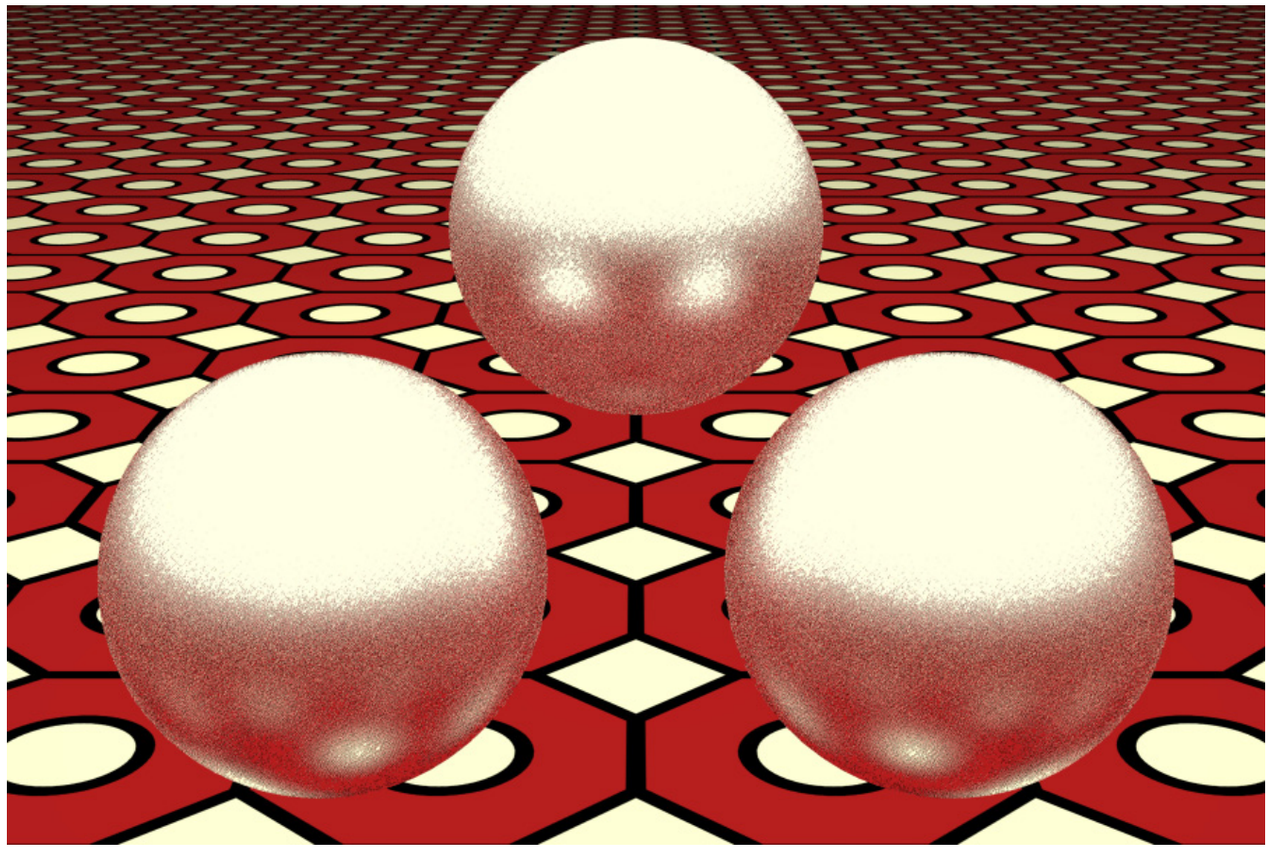
Scene file declaration of shader "glossy_reflection"

```
1  struct glossy_reflection {
2      miScalar shiny;
3  };
4
5  miBoolean glossy_reflection (
6      miColor *result, miState *state, struct glossy_reflection *params )
7  {
8      miVector reflection_dir;
9      miScalar shiny = *mi_eval_scalar(&params->shiny);
10     mi_reflection_dir_glossy(&reflection_dir, state, shiny);
11
12     if (!mi_trace_reflection(result, state, &reflection_dir))
13         mi_trace_environment(result, state, &reflection_dir);
14
15     return miTRUE;
16 }
```

Source code of shader "glossy_reflection"

Reflection

Glossy reflections

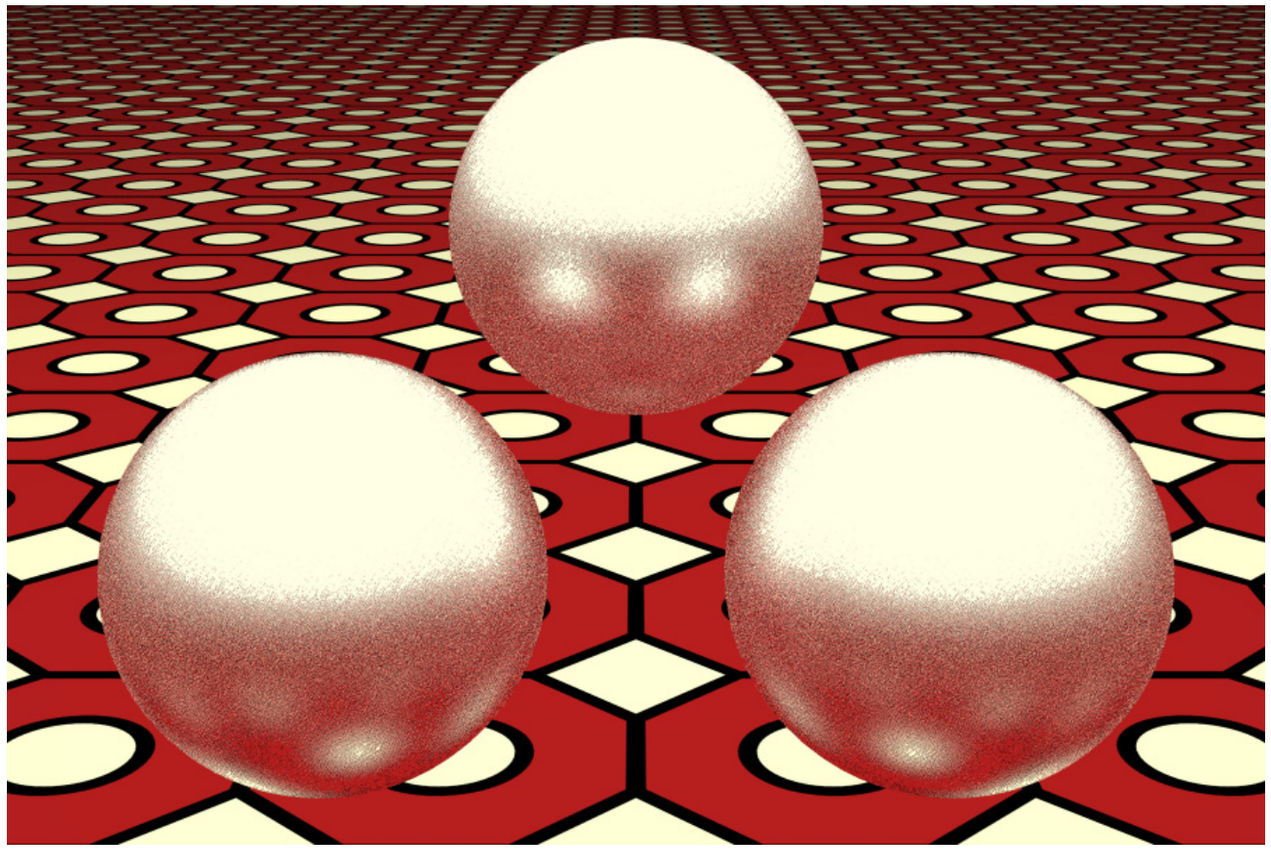


```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5
end options

material "reflect"
    "glossy_reflection" (
        "shiny" 3 )
end material
```

Glossy reflection

Reflection

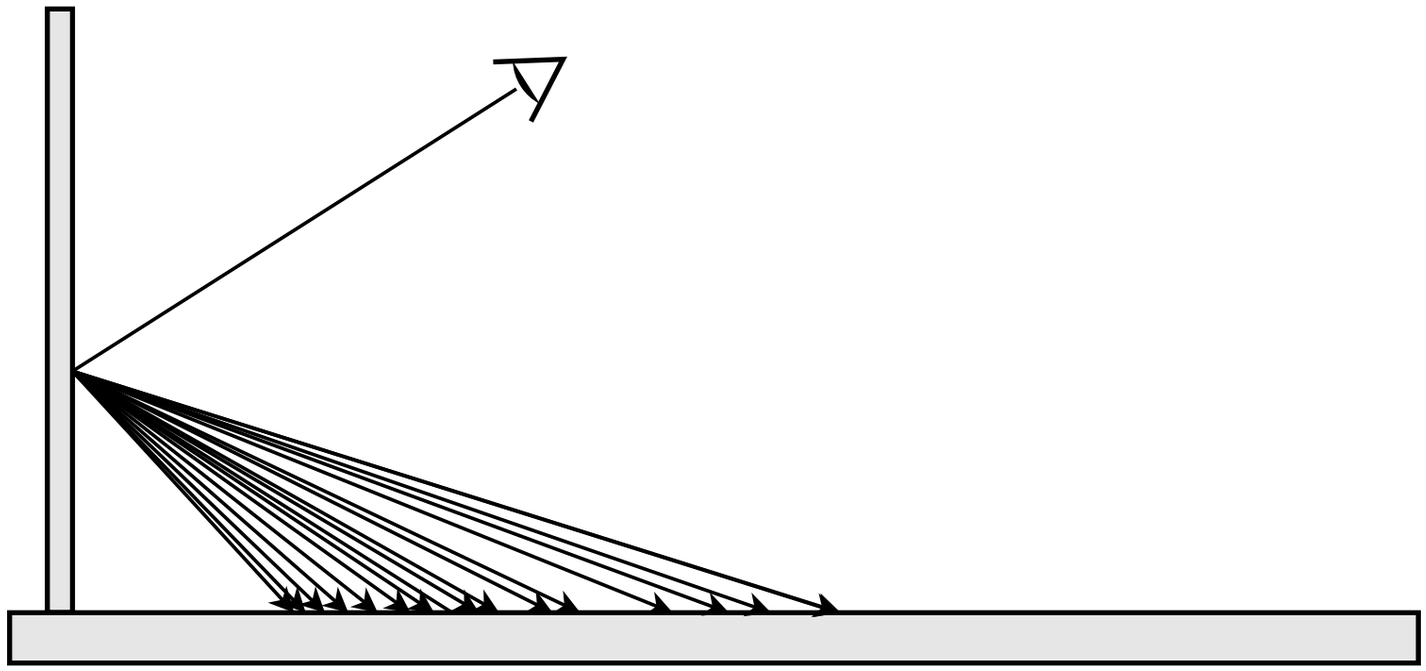
Glossy reflections



```
options "opt"
    object space
    samples 0 2
    contrast .01 .01 .01
    trace depth 5
end options

material "reflect"
    "glossy_reflection" (
        "shiny" 3 )
end material
```

Glossy reflection with increased contrast to improve quality

Glossy reflection with multiple samples in the shader



Multiple reflection directions averaged to produce glossy reflections

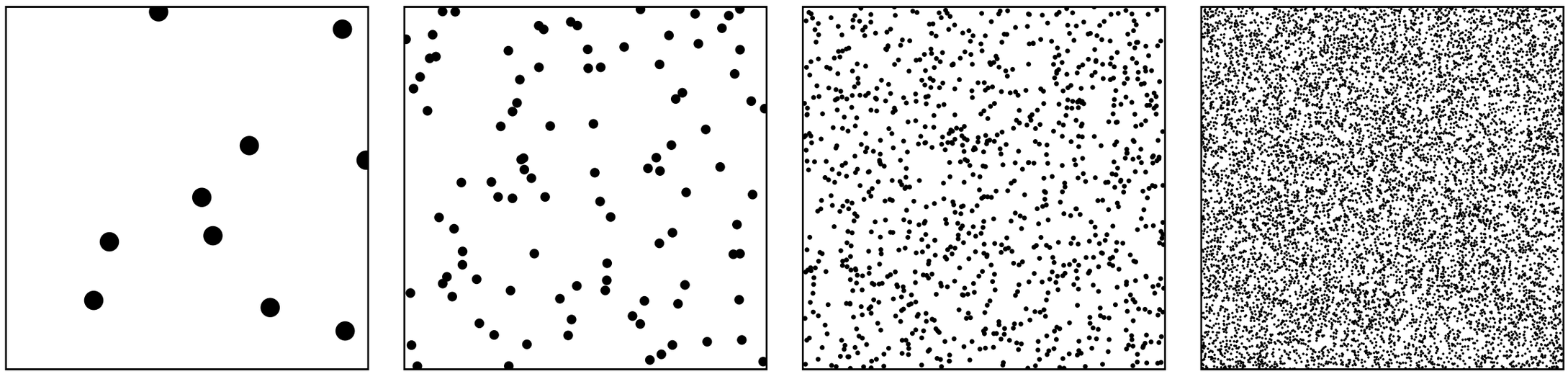
Glossy reflection with multiple samples in the shader

```
declare shader
    color "glossy_reflection_sample" (
        scalar "shiny" default 5,
        integer "samples" default 8 )
end declare
```

Scene file declaration of shader "glossy_reflection_sample"

Reflection

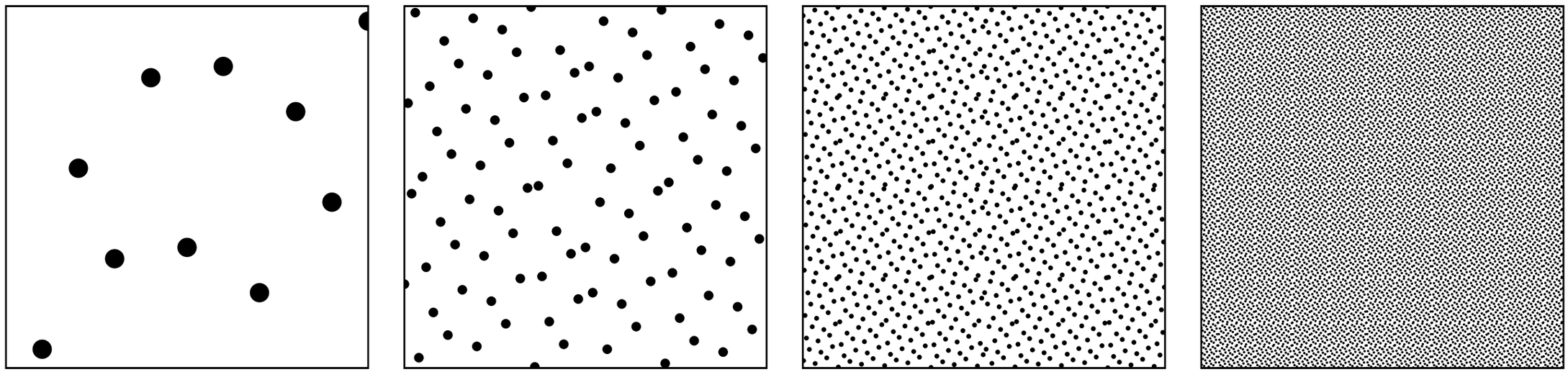
Glossy reflection with multiple samples in the shader



Increasing the number of random sample points (10, 100, 1,000, 10,000)

Reflection

Glossy reflection with multiple samples in the shader



Increasing the number of quasi-Monte Carlo sample points (10, 100, 1,000, 10,000)

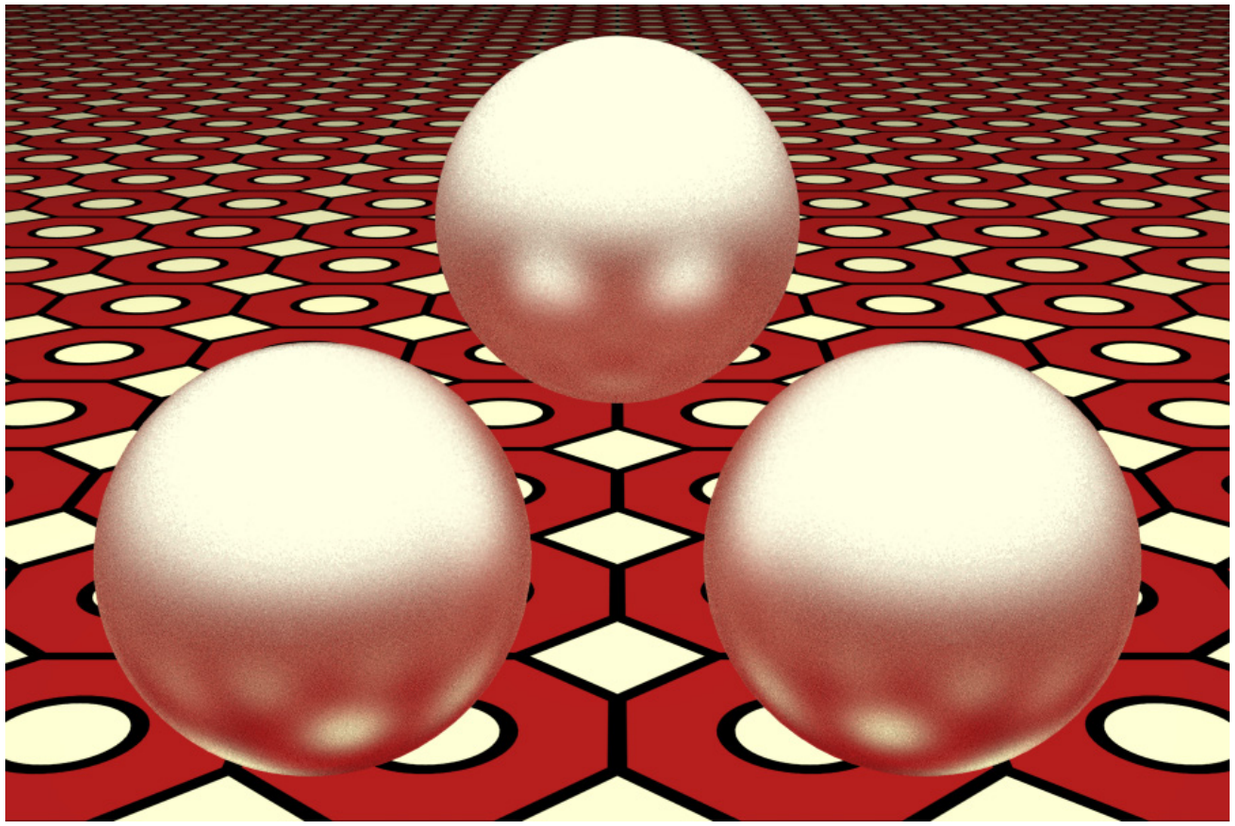
Glossy reflection with multiple samples in the shader

```
 1  struct glossy_reflection_sample {
 2      miScalar shiny;
 3      miInteger samples;
 4  };
 5
 6  miBoolean glossy_reflection_sample (
 7      miColor *result, miState *state, struct glossy_reflection_sample *params )
 8  {
 9      miScalar shiny = *mi_eval_scalar(&params->shiny);
10      miUint samples = *mi_eval_integer(&params->samples);
11      miVector reflect_dir;
12      miColor reflect_color;
13      int sample_number = 0;
14      double sampled_dir[2];
15
16      result->r = result->g = result->b = 0.0;
17      while (mi_sample(sampled_dir, &sample_number, state, 2, &samples)) {
18          mi_reflection_dir_glossy_x(&reflect_dir, state, shiny, sampled_dir);
19          if (!mi_trace_reflection(&reflect_color, state, &reflect_dir))
20              mi_trace_environment(&reflect_color, state, &reflect_dir);
21          miaux_add_color(result, &reflect_color);
22      }
23      miaux_scale_color(result, 1.0 / (double)samples);
24
25      return miTRUE;
26  }
```

Source code of shader "glossy_reflection_sample"

Reflection

Glossy reflection with multiple samples in the shader

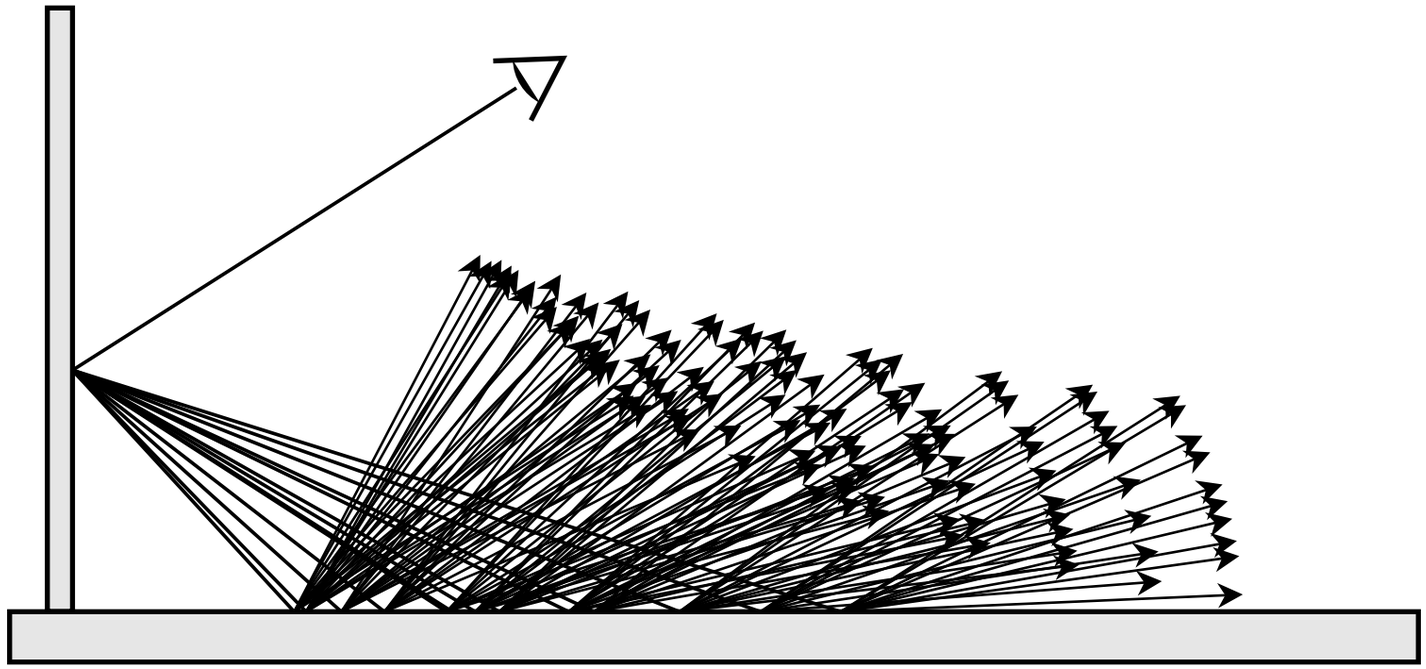


```
options "opt"
    object space
    samples 0 2
    contrast .05 .05 .05
    trace depth 5
end options

material "reflect"
    "glossy_reflection_sample" (
        "shiny" 3,
        "samples" 8 )
end material
```

Glossy reflection with additional rays generated at reflection point

Reflection

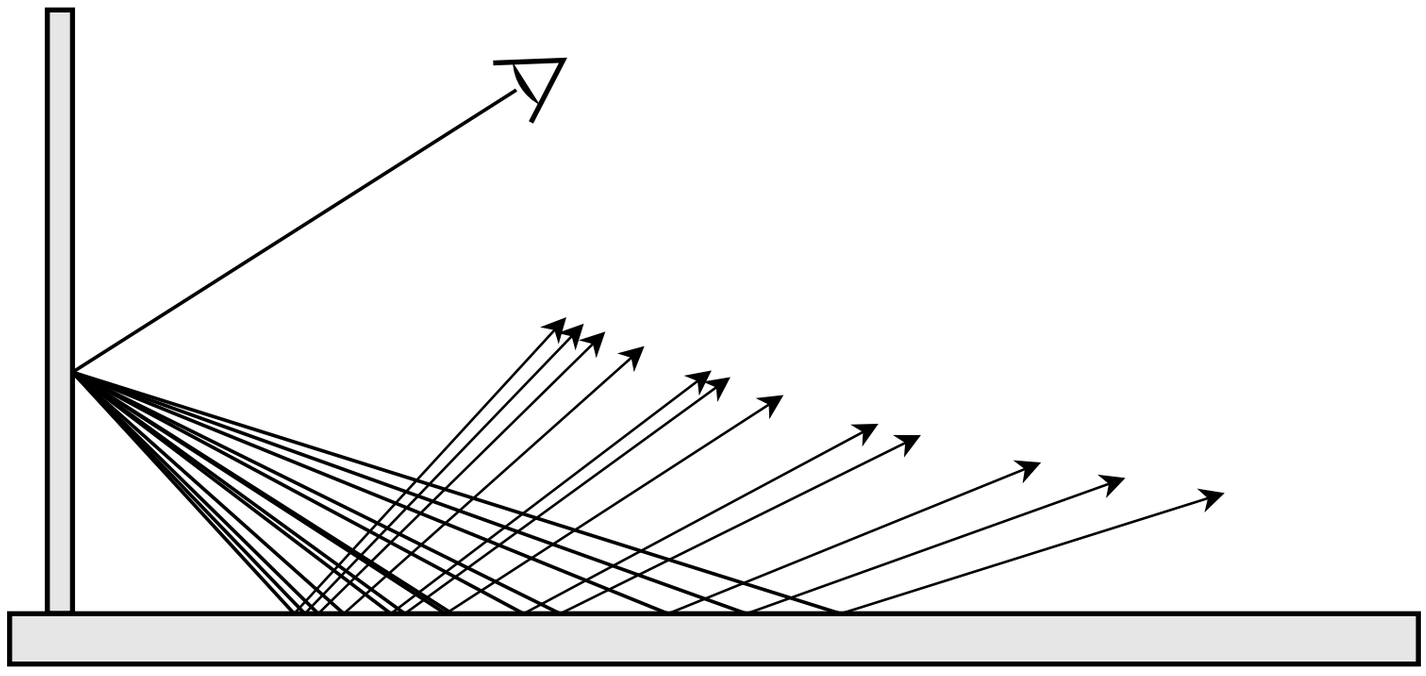
Glossy reflection with varying sample counts for reflections



Subsequent reflections increasing the ray count exponentially

Reflection

Glossy reflection with varying sample counts for reflections



Varying the number of sample rays at each reflection level

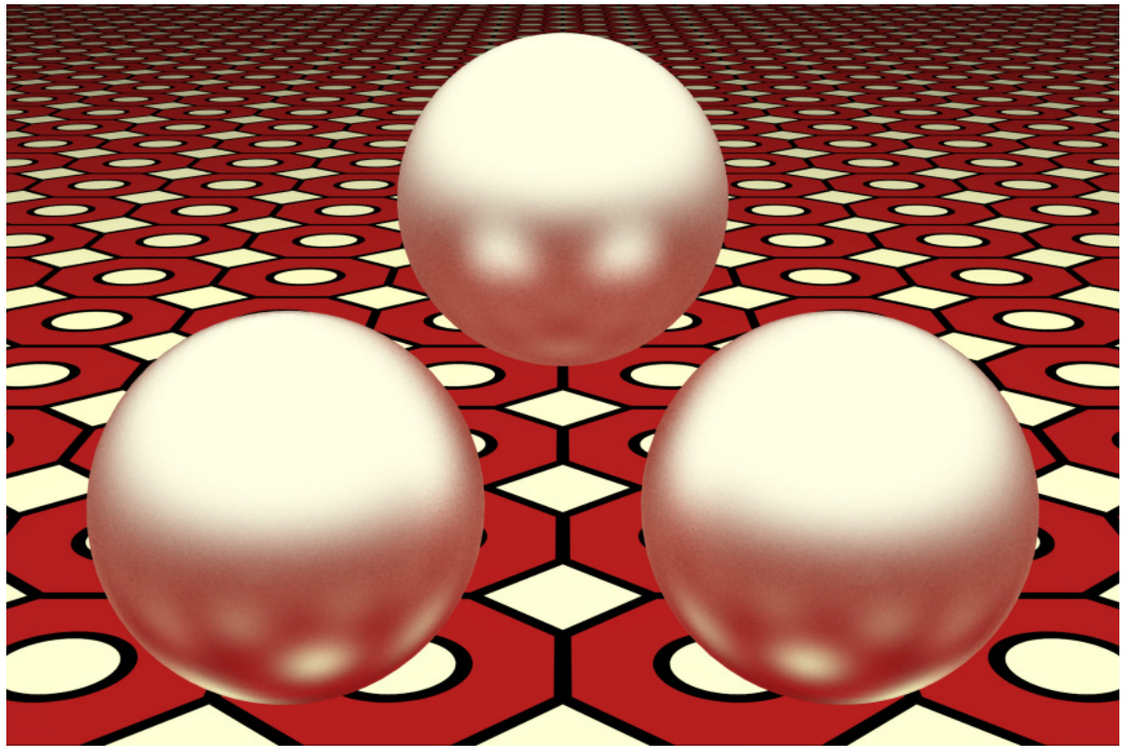
Glossy reflection with varying sample counts for reflections

```
declare shader
    color "glossy_reflection_sample_varying" (
        scalar "shiny" default 5,
        array integer "samples" )
end declare
```

Scene file declaration of shader "glossy_reflection_sample_varying"

Reflection

Glossy reflection with varying sample counts for reflections



```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5
end options

material "reflect"
    "glossy_reflection_sample_varying" (
        "shiny" 3,
        "samples" [200, 1] )
end material
```

Glossy reflection with 200 rays at first reflection but only one for the second

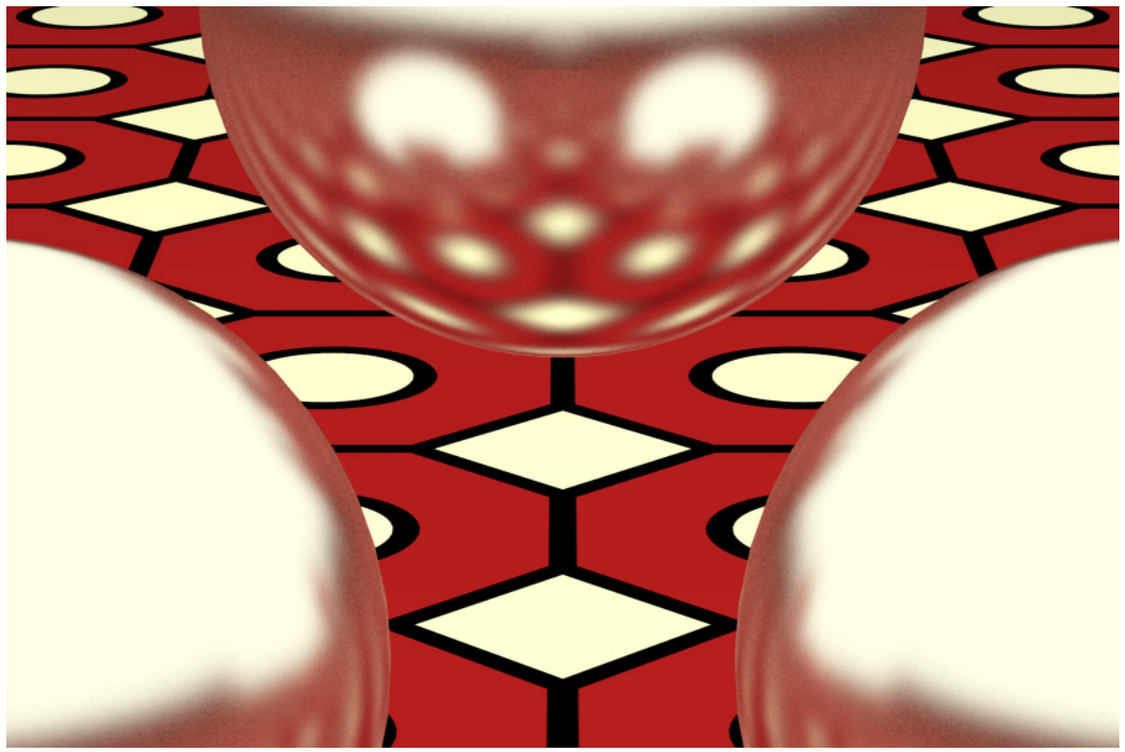
Glossy reflection with varying sample counts for reflections

```
 1  struct glossy_reflection_sample_varying {
 2      miScalar shiny;
 3      int i_samples;
 4      int n_samples;
 5      int samples[1];
 6  };
 7
 8  miBoolean glossy_reflection_sample_varying (
 9      miColor *result, miState *state,
10      struct glossy_reflection_sample_varying *params)
11  {
12      miScalar shiny = *mi_eval_scalar(&params->shiny);
13      int i_samples  = *mi_eval_integer(&params->i_samples);
14      int n_samples  = *mi_eval_integer(&params->n_samples);
15      int *samples   = mi_eval_integer(params->samples) + i_samples;
16      miVector reflect_dir;
17      miColor reflect_color;
18      double sampled_dir[2];
19      int level = state->reflection_level, sample_number = 0;
20      miUint sample_count = samples[level >= n_samples ? n_samples - 1 : level];
21
22      if (sample_count == 1) {
23          mi_reflection_dir_glossy(&reflect_dir, state, shiny);
24          if (!mi_trace_reflection(result, state, &reflect_dir))
25              mi_trace_environment(result, state, &reflect_dir);
26      } else {
27          result->r = result->g = result->b = 0.0;
28          while (mi_sample(sampled_dir, &sample_number,
29                           state, 2, &sample_count)) {
30              mi_reflection_dir_glossy_x(&reflect_dir, state, shiny,
31                                         sampled_dir);
32              if (!mi_trace_reflection(&reflect_color, state, &reflect_dir))
33                  mi_trace_environment(&reflect_color, state, &reflect_dir);
34              miaux_add_color(result, &reflect_color);
35          }
36          miaux_scale_color(result, 1.0 / (double)sample_count);
37      }
38      return miTRUE;
39  }
```

Source code of shader "glossy_reflection_sample_varying"

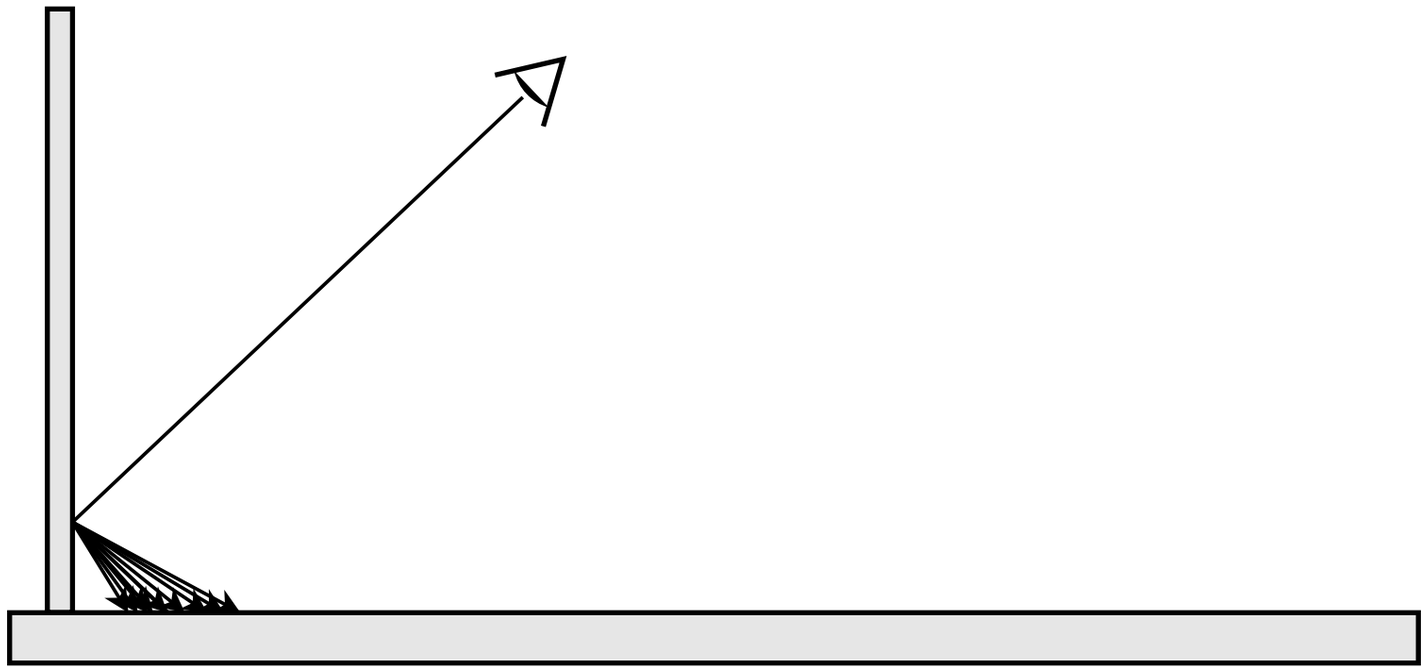
Reflection

Glossy reflection with varying sample counts for reflections



```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5
end options

material "reflect"
    "glossy_reflection_sample_varying" (
        "shiny" 10,
        "samples" [100, 4, 1] )
end material
```

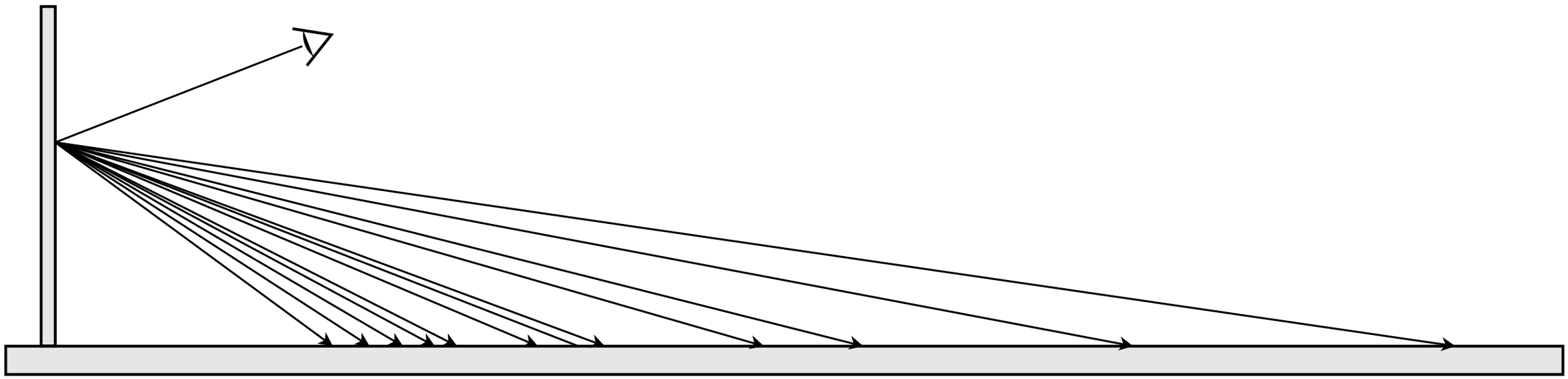
Glossy reflection with different number of samples for first three reflections



Effect of object geometry on degree of glossy reflection

Reflection

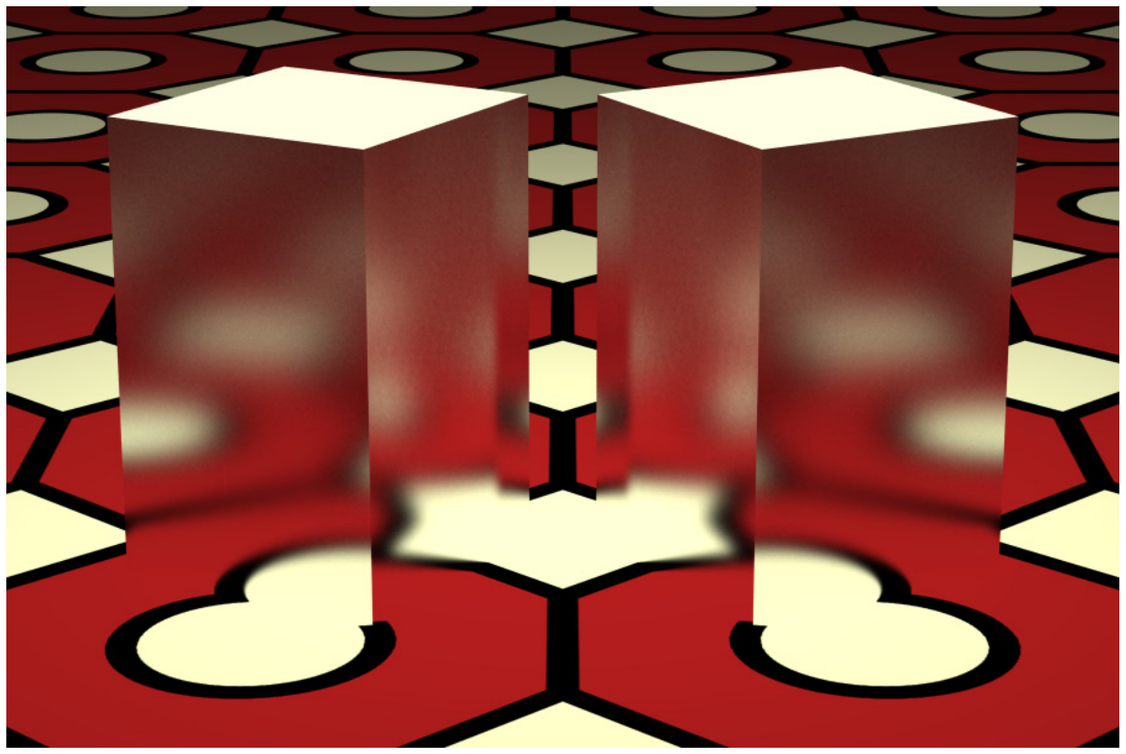
Glossy reflections and object geometry



Effect of object geometry on degree of glossy reflection

Reflection

Glossy reflections and object geometry

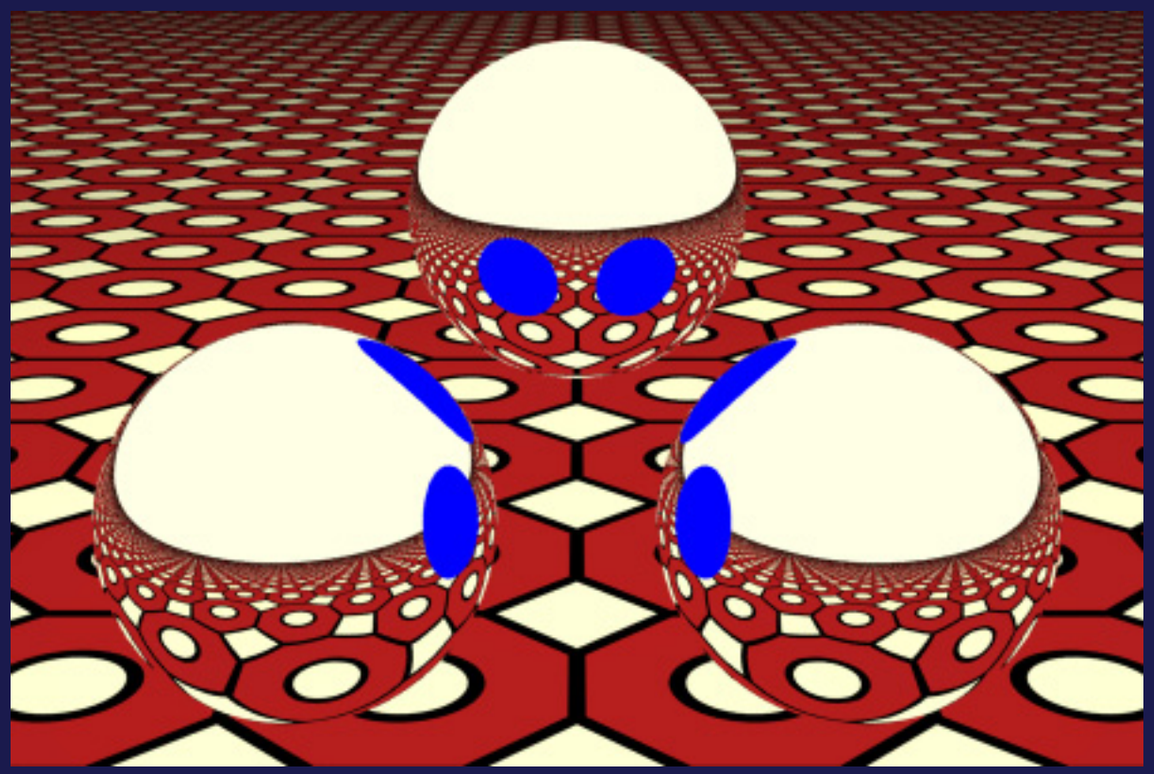


```
options "opt"
    object space
    samples 0 2
    shadow off
    contrast .1 .1 .1 1
    trace depth 8 8 8
end options

material "reflect"
    "glossy_reflection_sample_varying" (
        "samples" [100, 4, 1],
        "shiny" 20 )
end material
```

Variation of apparent glossiness due to object position

Exercise 12: Reflection shaders

1. Copy `reflection_1.mi` to `reflection.mi`
2. Change the output file to `reflection.tif`
3. Render and view.
4. Reduce trace depth to 0 and re-render, then increase trace depths.
5. Change shader `specular_reflection` so that blue is the resulting color when the trace depth is exceeded.



Exercise 12: Reflection shaders (part 2)

Change shader `specular_reflection` so that blue is the resulting color when the trace depth is exceeded.

Old:

```
if (!mi_trace_reflection(result, state, &reflection_direction))
    mi_trace_environment(result, state, &reflection_direction);
```

New:

```
if (!mi_trace_reflection(result, state, &reflection_direction)) {
    result->r = 0.0;
    result->g = 0.0;
    result->b = 1.0;
}
```

When you’re done, restore shader `specular_reflection` to the original code.

Refraction

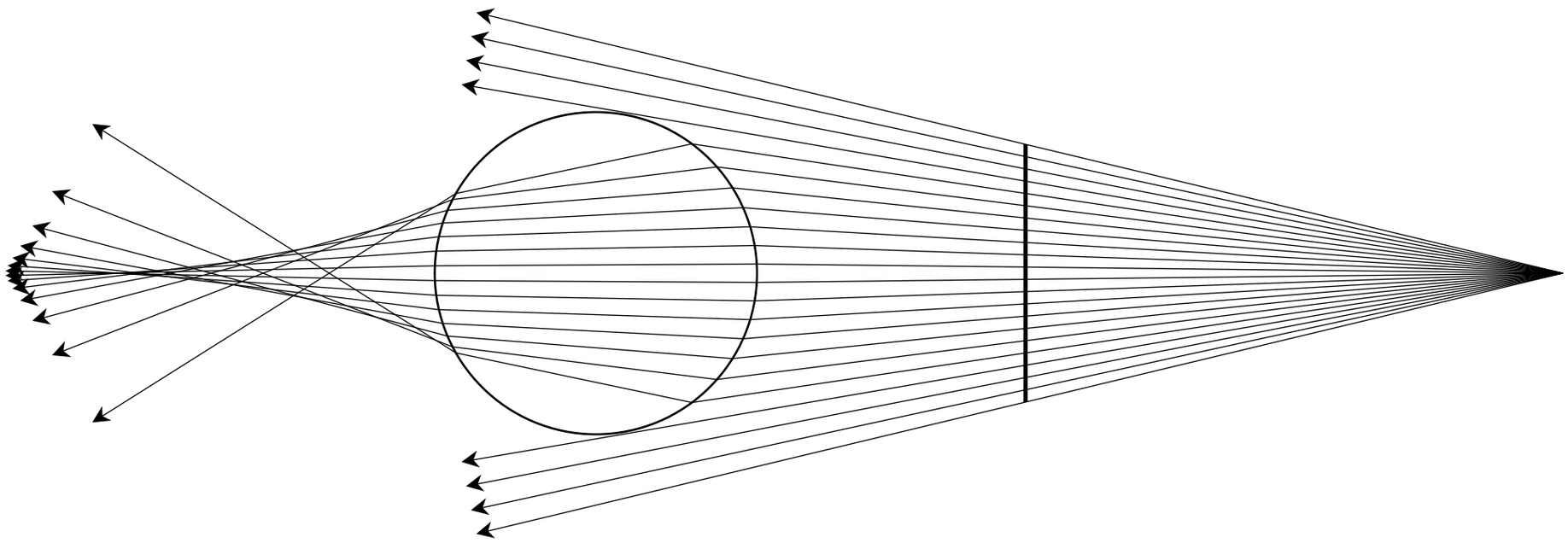
Refraction

- Specular refraction of non-intersecting objects
- Improving the determination of the indices of refraction
- Using different indices of refraction
- Glossy refraction
    - Using a single glossy refraction ray
    - Using multiple glossy refraction rays
    - Controlling the number of rays per refraction
- Combining reflection and refraction with Phenomena

The index of refraction for some physical materials

Material	*Index of refraction* tline
Vacuum	`1.0`
Air	`1.00029`
Ice	`1.31`
Water at 68F	`1.33`
Ethyl alcohol	`1.36`
Fused quartz	`1.46`
Glass	`1.517`
Emerald	`1.576`
Sapphire	`1.77`
Diamond	`2.417`

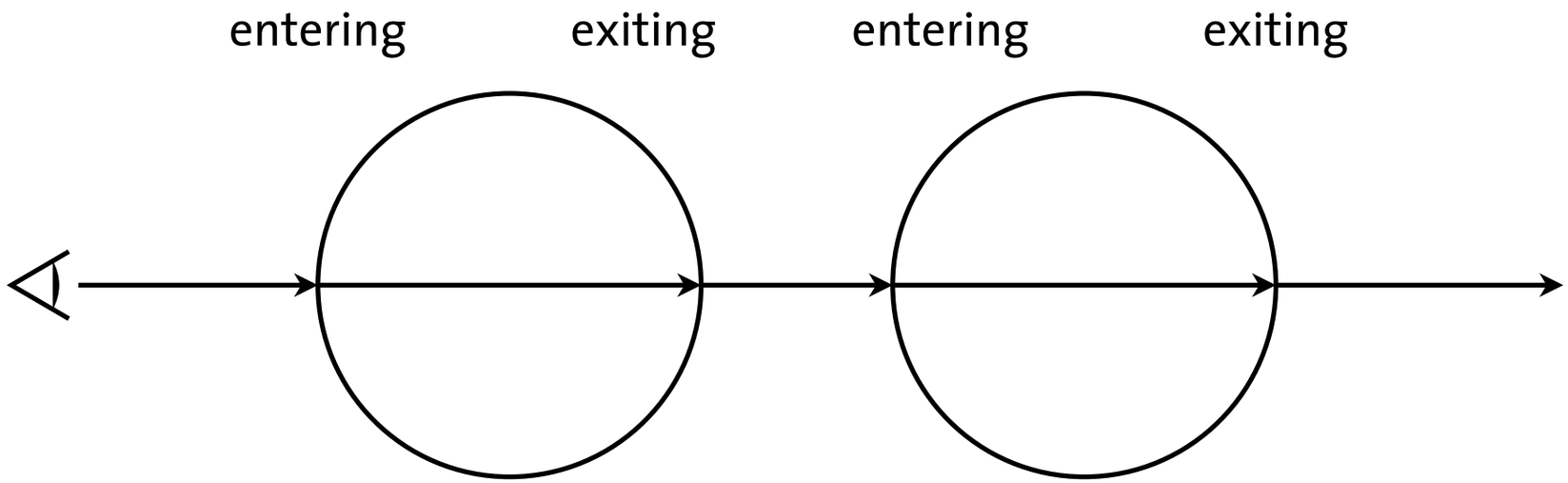
Refraction



Index of refraction of 1.33

Refraction

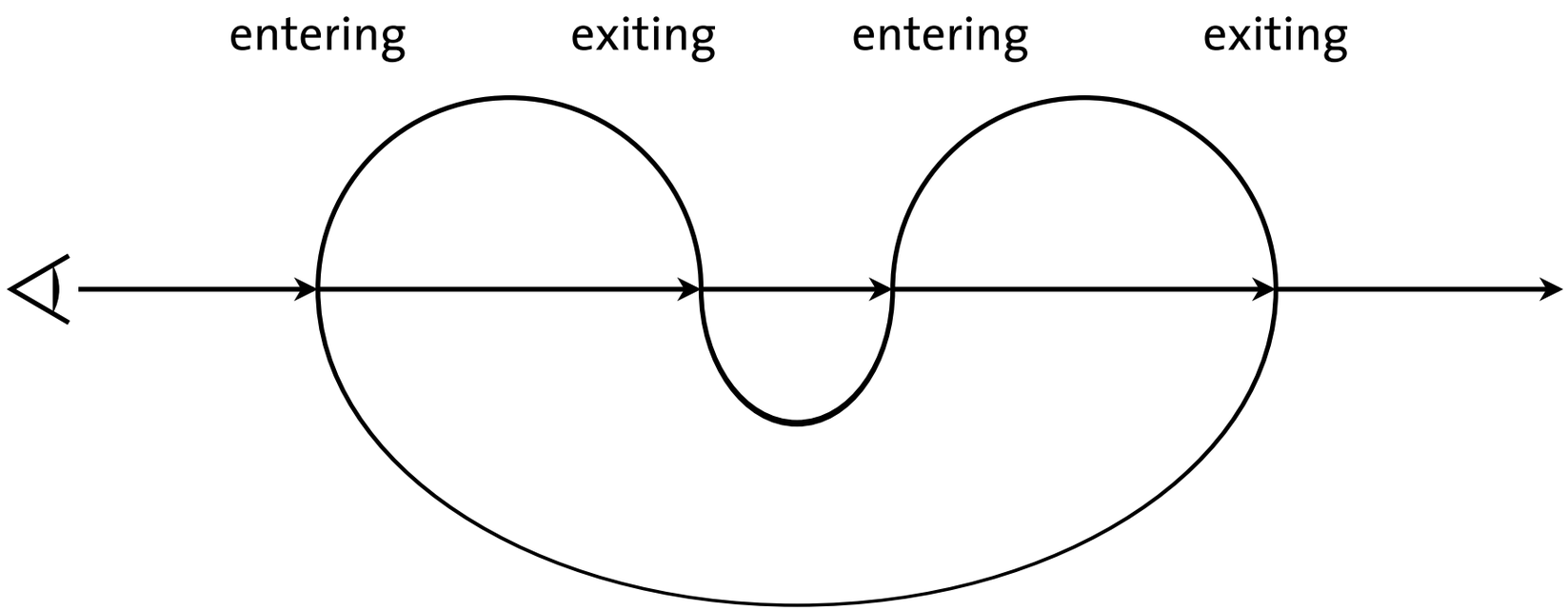
Specular refraction of non-intersecting objects



A ray entering and exiting multiple objects

Refraction

Specular refraction of non-intersecting objects



A ray entering and exiting a single object

Refraction

Specular refraction of non-intersecting objects

```
1  miBoolean miaux_ray_is_entering_material(miState *state)
2  {
3      miState *s;
4      miBoolean entering = miTRUE;
5      for (s = state; s != NULL; s = s->parent)
6          if (s->material == state->material)
7              entering = !entering;
8      return entering;
9  }
```

Auxiliary function: miaux_ray_is_entering_material

Refraction

Specular refraction of non-intersecting objects

```
declare shader
    color "specular_refraction_simple" (
         scalar "index_of_refraction" default 1.33 )
end declare
```

Scene file declaration of shader "specular_refraction_simple"

Refraction

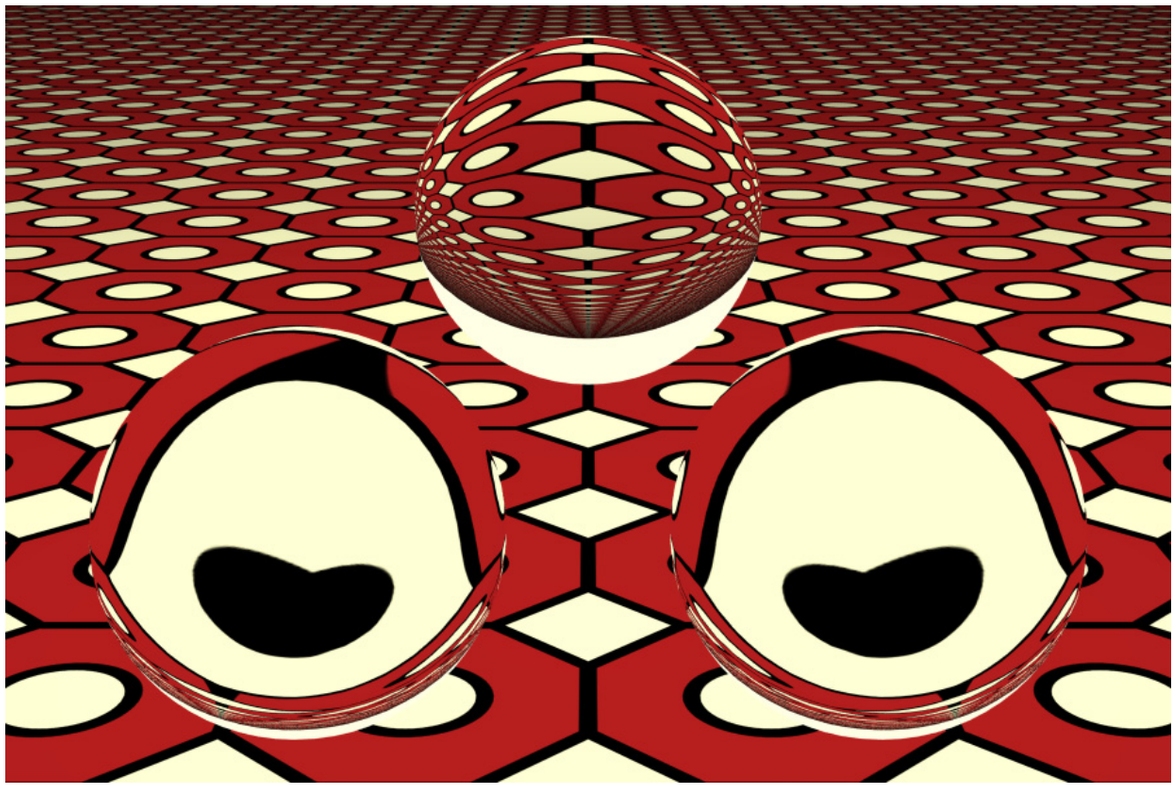
Specular refraction of non-intersecting objects

```
 1  struct specular_refraction_simple {
 2      miScalar index_of_refraction;
 3  };
 4
 5  miBoolean specular_refraction_simple (
 6      miColor *result, miState *state, struct specular_refraction_simple *params )
 7  {
 8      miScalar instance_ior = *mi_eval_scalar(&params->index_of_refraction);
 9      miScalar vacuum_ior = 1.0, incoming_ior, outgoing_ior;
10      miVector direction;
11      if (miaux_ray_is_entering_material(state)) {
12          incoming_ior = vacuum_ior;
13          outgoing_ior = instance_ior;
14      } else {
15          incoming_ior = instance_ior;
16          outgoing_ior = vacuum_ior;
17      }
18      if (mi_refraction_dir(&direction, state, incoming_ior, outgoing_ior))
19          mi_trace_refraction(result, state, &direction);
20      else {
21          mi_reflection_dir(&direction, state);
22          if (!mi_trace_reflection(result, state, &direction))
23              mi_trace_environment(result, state, &direction);
24      }
25      return miTRUE;
26  }
```

Source code of shader "specular_refraction_simple"

Refraction

Specular refraction of non-intersecting objects



```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5 5 5
    face both
end options

material "refract"
    "specular_refraction" (
        "index_of_refraction" 1.33 )
end material
```

Specular refraction with the default index of refraction of 1.33

Improving the determination of the indices of refraction

```
1  miBoolean miaux_ray_is_transmissive(miState *state)
2  {
3      return state->type == miRAY_TRANSPARENT ||
4          state->type == miRAY_REFRACT;
5  }
```

Auxiliary function: miaux_ray_is_transmissive

```
1  miBoolean miaux_parent_exists(miState *state)
2  {
3      return state->parent != NULL;
4  }
```

Auxiliary function: miaux_parent_exists

Improving the determination of the indices of refraction

```
1  miBoolean miaux_shaders_equal(miState *s1, miState *s2)
2  {
3      return s1->shader == s2->shader;
4  }
```

Auxiliary function: miaux_shaders_equal

Improving the determination of the indices of refraction

```
1  miBoolean miaux_ray_is_entering(
2      miState *state,
3      miState *state_of_this_shaders_previous_transmission)
4  {
5      miState *s;
6      miBoolean ray_is_entering = miTRUE;
7      state_of_this_shaders_previous_transmission = NULL;
8      for (s = state; s; s = s->parent)
9          if (miaux_ray_is_transmissive(s) &&
10             miaux_parent_exists(s) &&
11             miaux_shaders_equal(s->parent, state)) {
12             ray_is_entering = !ray_is_entering;
13             if (state_of_this_shaders_previous_transmission == NULL)
14                 state_of_this_shaders_previous_transmission = s->parent;
15         }
16     return ray_is_entering;
17 }
```

Auxiliary function: miaux_ray_is_entering

Improving the determination of the indices of refraction

```
1  miScalar miaux_state_incoming_ior(miState *state)
2  {
3      miScalar unassigned_ior = 0.0, default_ior = 1.0;
4      if (state != NULL && state->ior_in != unassigned_ior)
5          return state->ior_in;
6      else
7          return default_ior;
8  }
```

Auxiliary function: miaux_state_incoming_ior

Improving the determination of the indices of refraction

```
1  miScalar miaux_state_outgoing_ior(miState *state)
2  {
3      miScalar unassigned_ior = 0.0, default_ior = 1.0;
4      if (state != NULL && state->ior != unassigned_ior)
5          return state->ior;
6      else
7          return default_ior;
8  }
```

Auxiliary function: miaux_state_outgoing_ior

Improving the determination of the indices of refraction

```
1  void miaux_set_state_refraction_indices(miState *state,
2                                          miScalar material_ior)
3  {
4      miState *previous_transmission = NULL;
5      miScalar incoming_ior, outgoing_ior;
6
7      if (miaux_ray_is_entering(state, previous_transmission)) {
8          outgoing_ior = material_ior;
9          incoming_ior = miaux_state_outgoing_ior(state->parent);
10     } else {
11         incoming_ior = material_ior;
12         outgoing_ior = miaux_state_incoming_ior(previous_transmission);
13     }
14     state->ior_in = incoming_ior;
15     state->ior = outgoing_ior;
16 }
```

Auxiliary function: miaux_set_state_refraction_indices

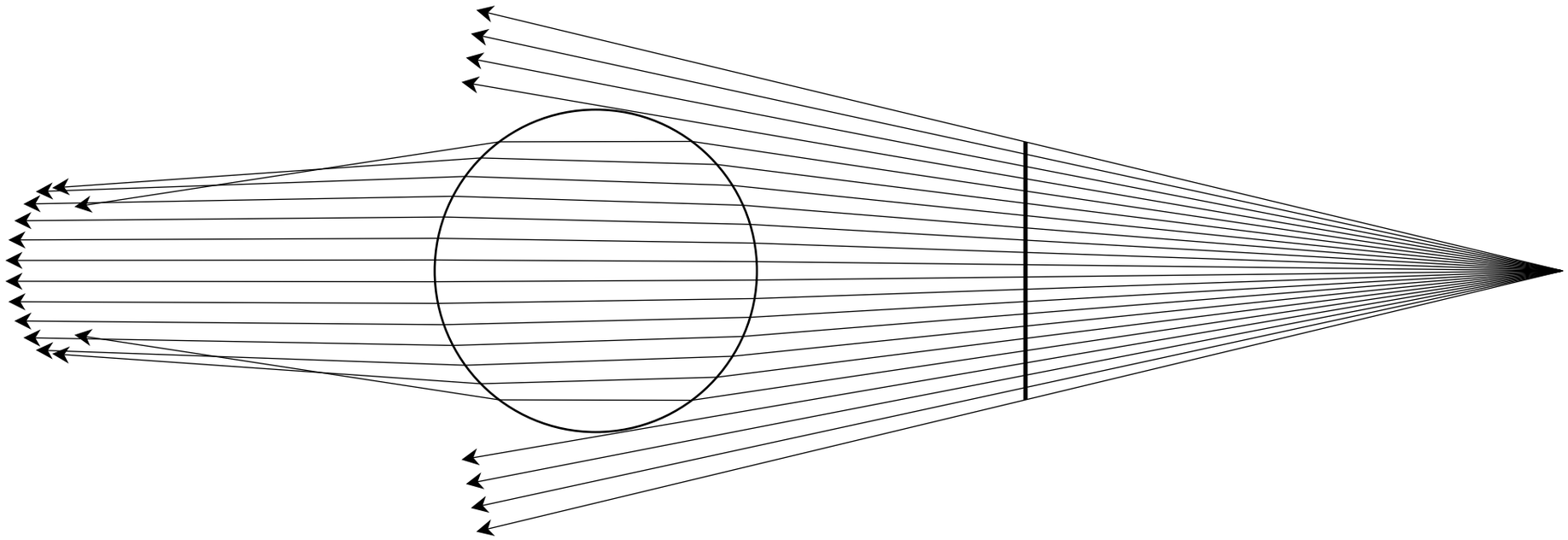
Improving the determination of the indices of refraction

```
 1  struct specular_refraction {
 2      miScalar index_of_refraction;
 3  };
 4
 5  miBoolean specular_refraction (
 6      miColor *result, miState *state, struct specular_refraction *params )
 7  {
 8      miVector direction;
 9      miScalar ior = *mi_eval_scalar(&params->index_of_refraction);
10      miaux_set_state_refraction_indices(state, ior);
11
12      if (mi_refraction_dir(&direction, state, state->ior_in, state->ior))
13          mi_trace_refraction(result, state, &direction);
14      else {
15          mi_reflection_dir(&direction, state);
16          if (!mi_trace_reflection(result, state, &direction))
17              mi_trace_environment(result, state, &direction);
18      }
19      return miTRUE;
20  }
```

Source code of shader "specular_refraction"

Refraction

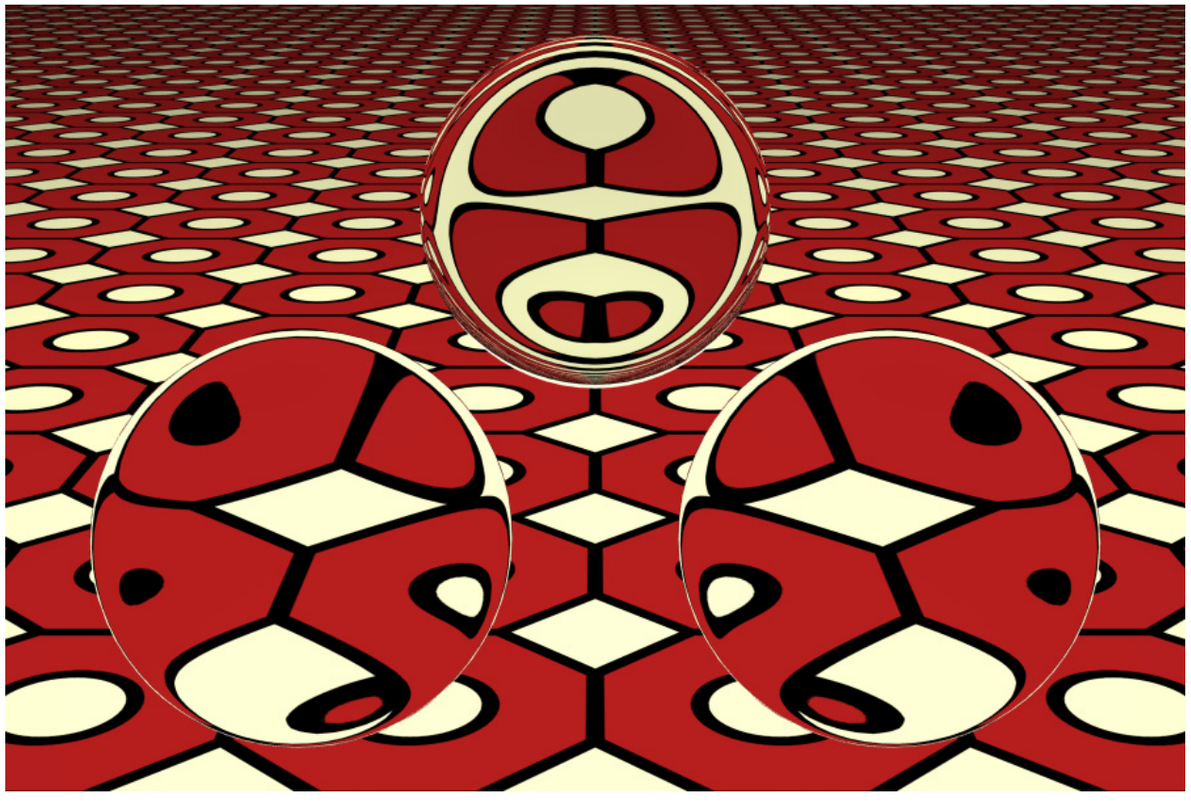
Using different indices of refraction



Index of refraction of 1.1

Refraction

Using different indices of refraction

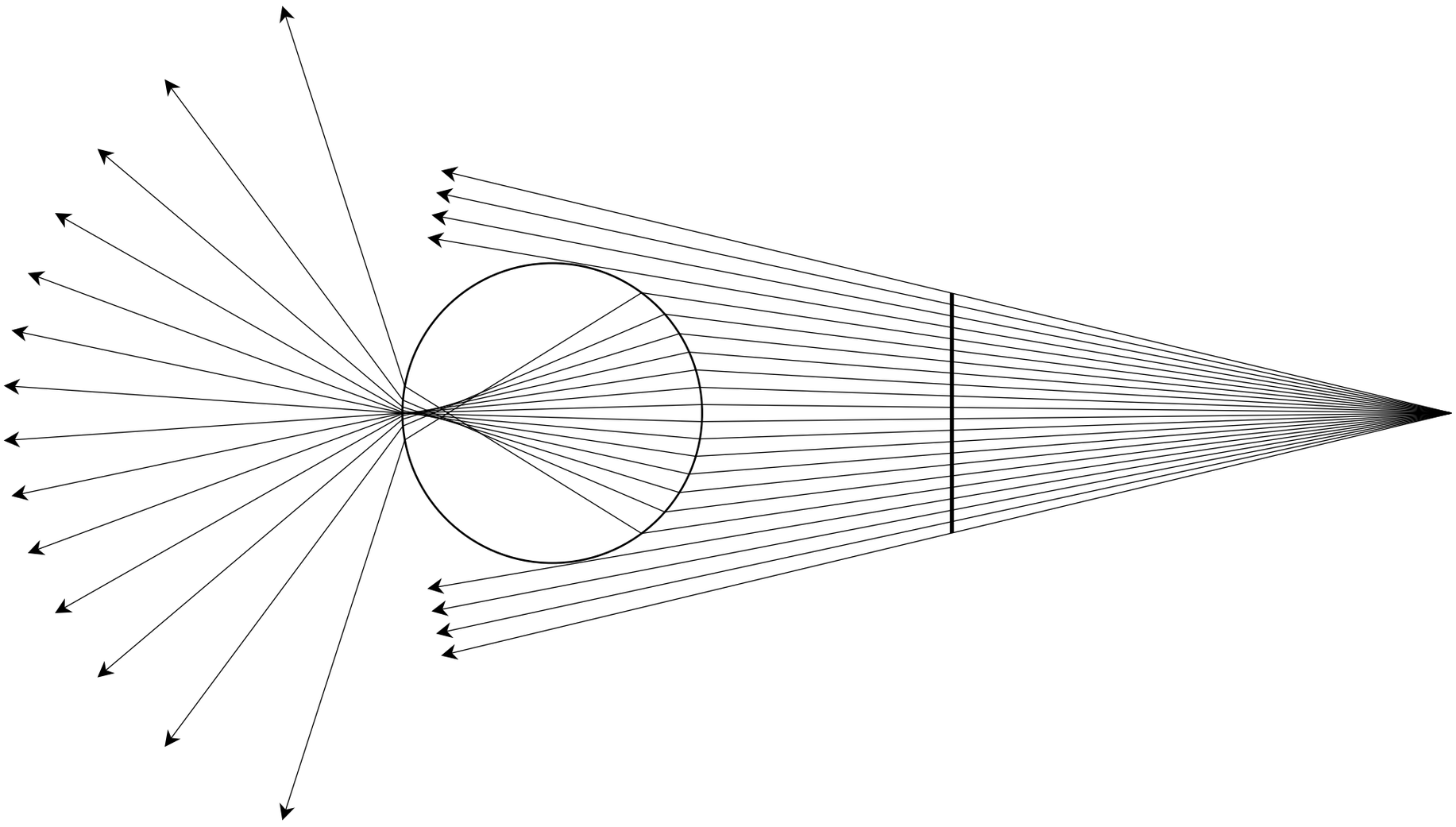


```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5 5 5
    face both
end options

material "refract"
    "specular_refraction" (
        "index_of_refraction" 1.1 )
end material
```

Specular refraction with an index of refraction of 1.1

Refraction

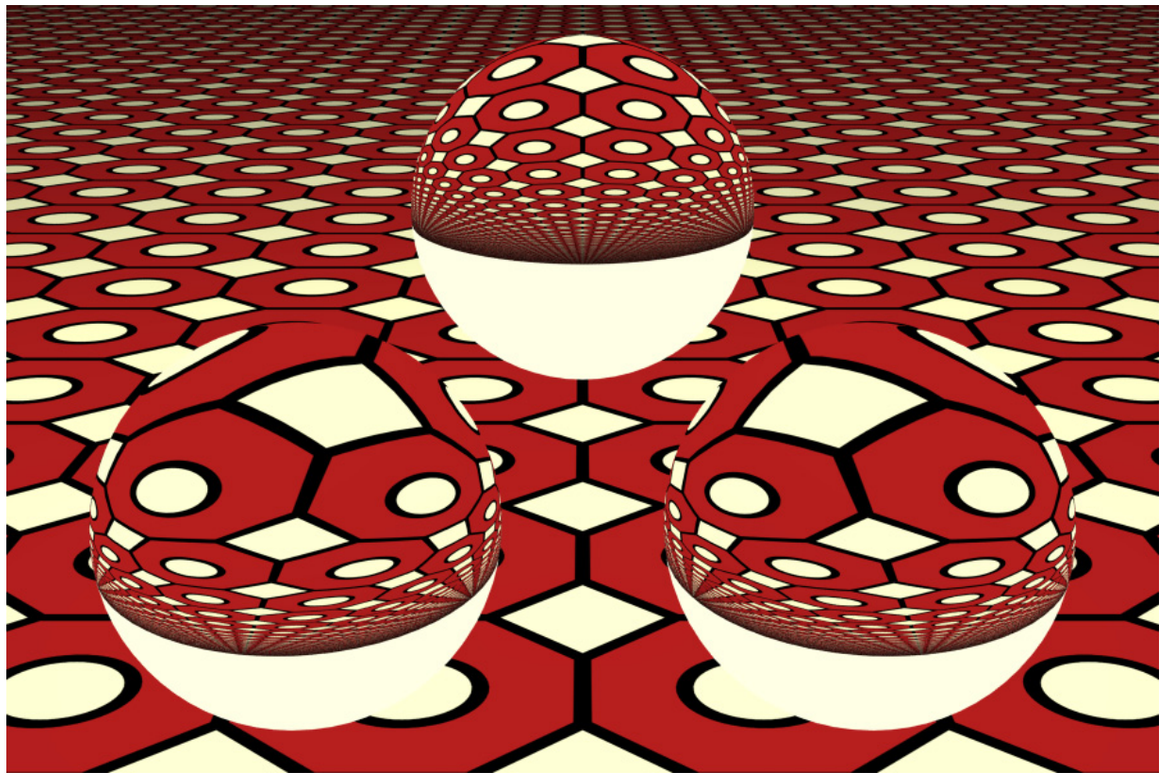
Using different indices of refraction



Index of refraction of 2.4

Refraction

Using different indices of refraction



```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5 5 5
    face both
end options

material "refract"
    "specular_refraction" (
        "index_of_refraction" 2.417 )
end material
```

Specular refraction with an index of refraction of 2.417

Refraction

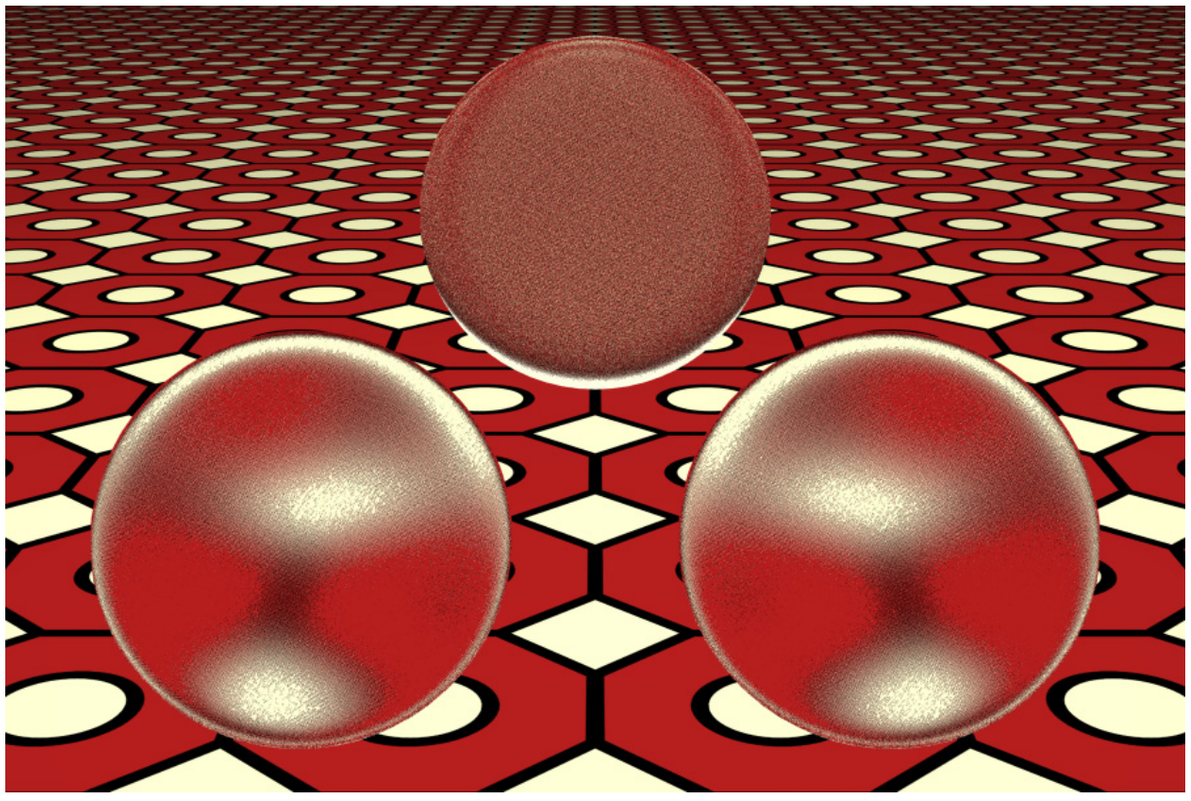
Glossy refraction

```
declare shader
    color "glossy_refraction" (
        scalar "index_of_refraction" default 1.33,
        scalar "shiny" default 5 )
end declare
```

Scene file declaration of shader "glossy_refraction"

Refraction

Glossy refraction



```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5 5 5
    face both
end options

material "refract"
    "glossy_refraction" (
        "index_of_refraction" 1.15,
        "shiny" 35 )
end material
```

Glossy refraction with an index of refraction of 1.15

Refraction

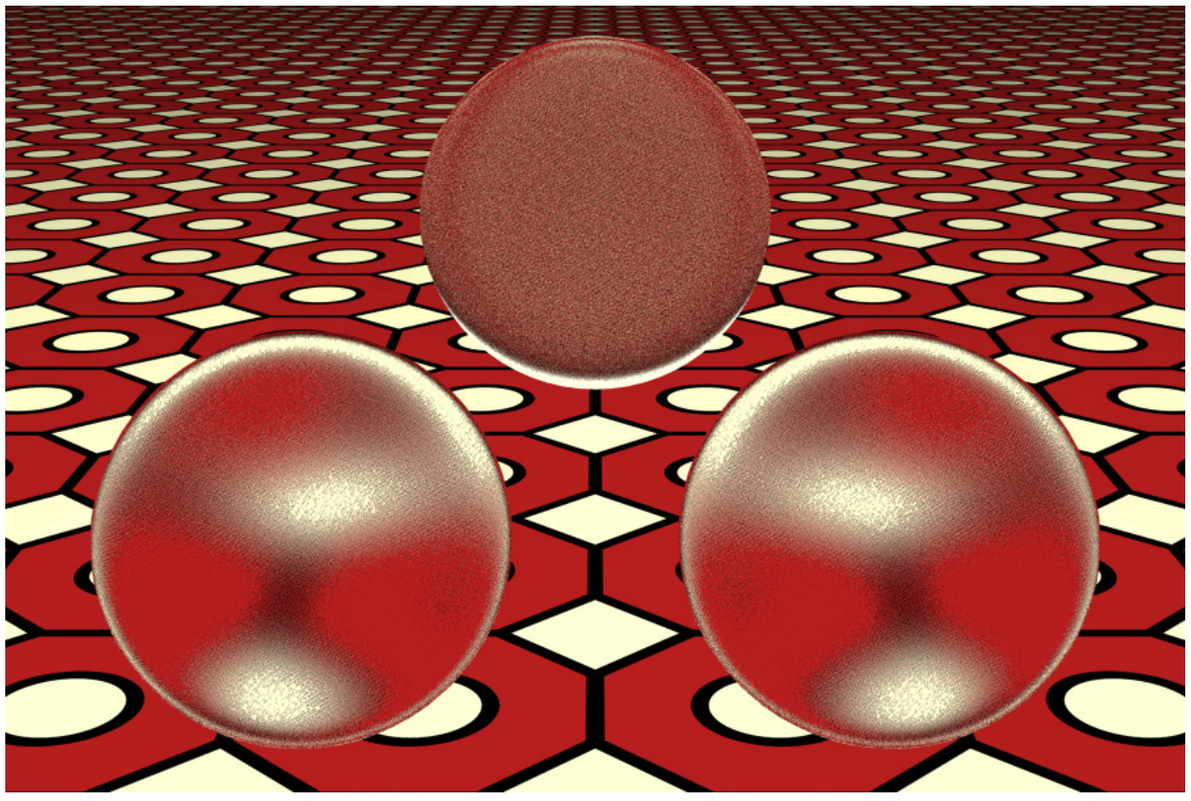
Glossy refraction

```
 1  struct glossy_refraction {
 2      miScalar index_of_refraction;
 3      miScalar shiny;
 4  };
 5
 6  miBoolean glossy_refraction(
 7      miColor *result, miState *state, struct glossy_refraction *params)
 8  {
 9      miVector direction;
10      miScalar ior = *mi_eval_scalar(&params->index_of_refraction);
11      miScalar shiny = *mi_eval_scalar(&params->shiny);
12
13      miaux_set_state_refraction_indices(state, ior);
14
15      if (mi_transmission_dir_glossy(&direction, state,
16                                     state->ior_in, state->ior, shiny))
17          mi_trace_refraction(result, state, &direction);
18      else {
19          mi_reflection_dir(&direction, state);
20          if (!mi_trace_reflection(result, state, &direction))
21              mi_trace_environment(result, state, &direction);
22      }
23      return miTRUE;
24  }
```

Source code of shader "glossy_refraction"

Refraction

Glossy refraction



```
options "opt"
    object space
    samples 0 2
    contrast .01 .01 .01
    trace depth 5 5 5
    face both
end options

material "refract"
    "glossy_refraction" (
        "index_of_refraction" 1.15,
        "shiny" 35 )
end material
```

Glossy refraction with an index of refraction of 1.15, sampling of .01 .01 .01

Refraction

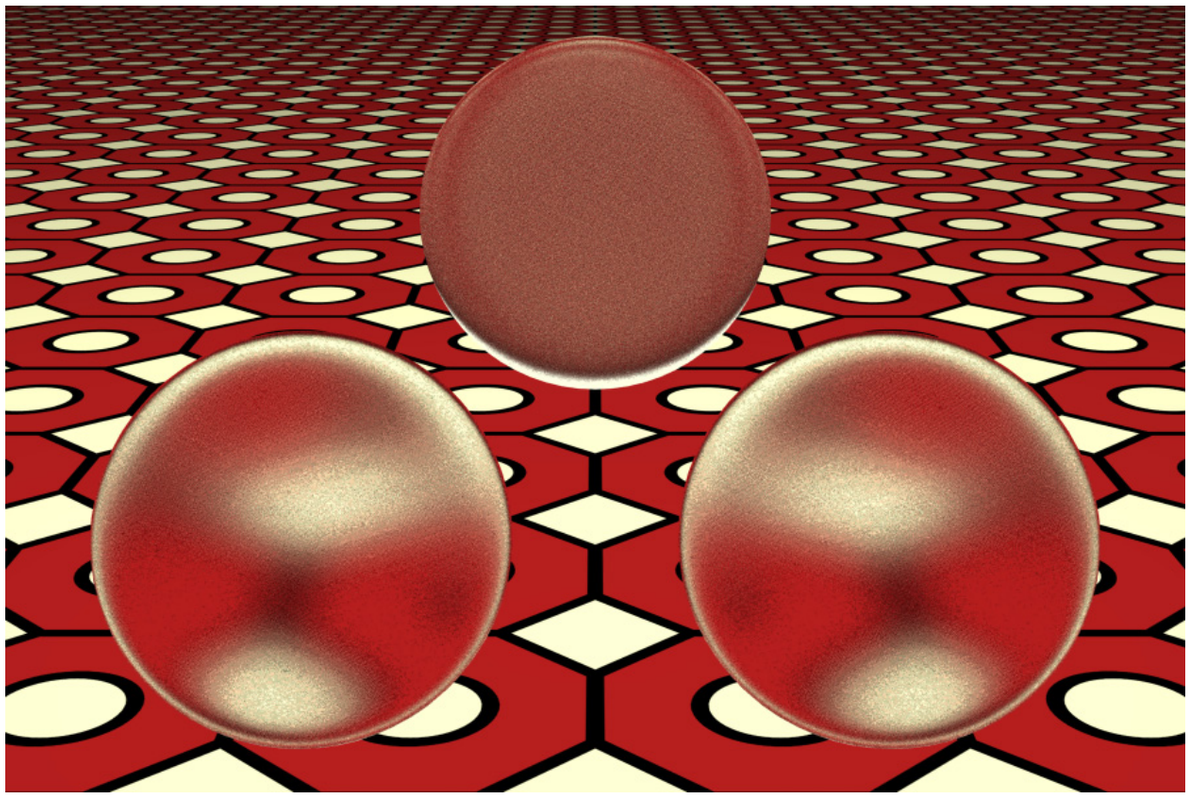
Glossy refraction

```
declare shader
    color "glossy_refraction_sample" (
        scalar "index_of_refraction" default 1.33,
        scalar "shiny" default 50,
        integer "samples" default 8 )
end declare
```

Scene file declaration of shader "glossy_refraction_sample"

Refraction

Glossy refraction



```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5 5 5
    face both
end options

material "refract"
    "glossy_refraction_sample" (
        "index_of_refraction" 1.15,
        "shiny" 35,
        "samples" 4 )
end material
```

Glossy refraction with an index of refraction of 1.15 with 4 samples per intersection

Refraction

Glossy refraction

```
 1  struct glossy_refraction_sample {
 2      miScalar index_of_refraction;
 3      miScalar shiny;
 4      miInteger samples;
 5  };
 6
 7  miBoolean glossy_refraction_sample(
 8      miColor *result, miState *state, struct glossy_refraction_sample *params)
 9  {
10      miScalar ior = *mi_eval_scalar(&params->index_of_refraction);
11      miScalar shiny = *mi_eval_scalar(&params->shiny);
12      miUint samples = *mi_eval_integer(&params->samples);
13      miVector refract_dir, reflect_dir;
14      miColor refract_color;
15      int sample_number = 0;
16      double sample[2];
17
18      miaux_set_state_refraction_indices(state, ior);
19
20      result->r = result->g = result->b = 0.0;
21      while (mi_sample(sample, &sample_number, state, 2, &samples)) {
22          if (mi_transmission_dir_glossy_x(&refract_dir, state,
23                                           state->ior_in, state->ior,
24                                           shiny, sample)) {
25              if (mi_trace_refraction(&refract_color, state, &refract_dir))
26                  miaux_add_color(result, &refract_color);
27          }
28          else {
29              mi_reflection_dir(&reflect_dir, state);
30              if (!mi_trace_reflection(&refract_color, state, &reflect_dir))
31                  mi_trace_environment(&refract_color, state, &reflect_dir);
32              miaux_add_color(result, &refract_color);
33          }
34      }
35      miaux_scale_color(result, 1.0 / samples);
36      return miTRUE;
37  }
```

Source code of shader "glossy_refraction_sample"

Refraction

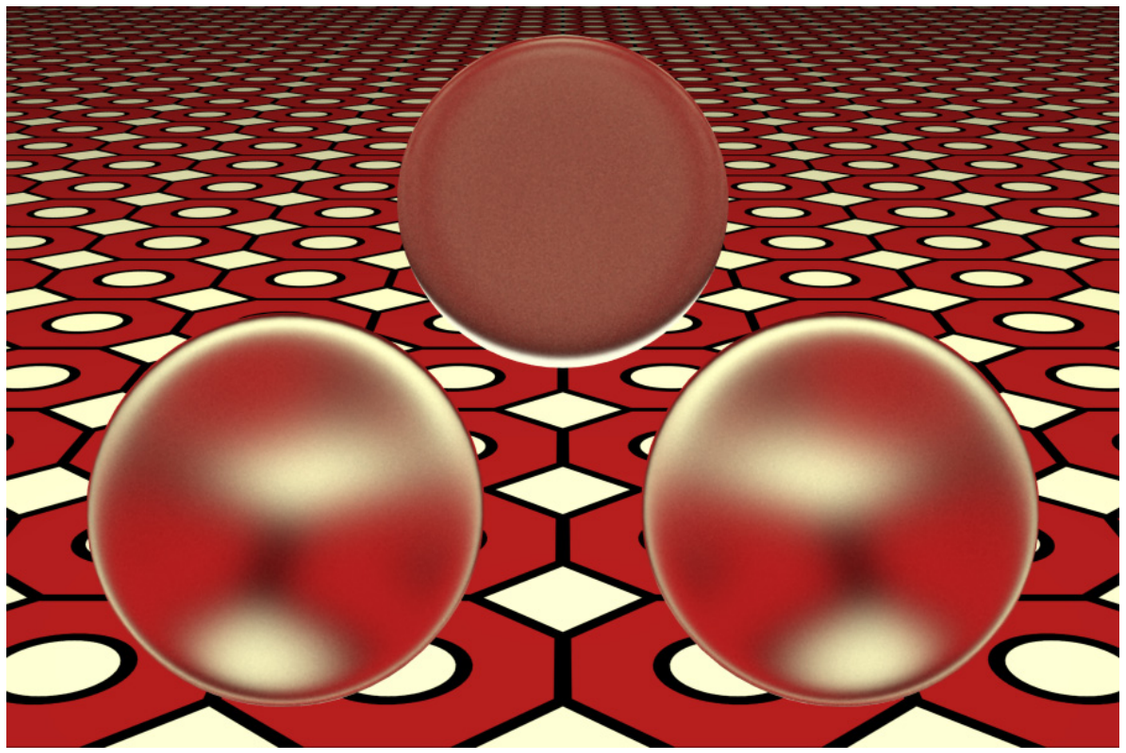
Glossy refraction

```
declare shader
    color "glossy_refraction_sample_varying" (
        scalar "index_of_refraction" default 1.33,
        scalar "shiny" default 50,
        array integer "samples" )
end declare
```

Scene file declaration of shader "glossy_refraction_sample_varying"

Refraction

Glossy refraction



```
options "opt"
    object space
    samples 0 2
    contrast .1 .1 .1
    trace depth 5 5 5
    face both
end options

material "refract"
    "glossy_refraction_sample_varying" (
        "index_of_refraction" 1.15,
        "shiny" 35,
        "samples" [100,4,1] )
end material
```

Glossy refraction with an index of refraction of 1.15 with varying sample count

Refraction

Glossy refraction

```
 1  struct glossy_refraction_sample_varying {
 2      miScalar index_of_refraction;
 3      miScalar shiny;
 4      int i_samples;
 5      int n_samples;
 6      int samples[1];
 7  };
 8
 9  miBoolean glossy_refraction_sample_varying(
10      miColor *result, miState *state,
11      struct glossy_refraction_sample_varying *params)
12  {
13      miScalar ior   = *mi_eval_scalar(&params->index_of_refraction);
14      miScalar shiny = *mi_eval_scalar(&params->shiny);
15      int i_samples  = *mi_eval_integer(&params->i_samples);
16      int n_samples  = *mi_eval_integer(&params->n_samples);
17      int *samples   = mi_eval_integer(params->samples) + i_samples;
18      miVector refraction_dir, reflect_dir;
19      miColor refract_color, reflect_color;
20      int sample_number = 0;
21      double sample[2];
22      miUint sample_count;
23
24      sample_count = samples[state->refraction_level >= n_samples ?
25                             n_samples - 1 :
26                             state->refraction_level];
27
28      miaux_set_state_refraction_indices(state, ior);
29
30      if (sample_count == 1) {
31          if (mi_transmission_dir_glossy(&refraction_dir, state,
32                                         state->ior_in, state->ior, shiny))
33              mi_trace_refraction(result, state, &refraction_dir);
34          else {
35              mi_reflection_dir(&reflect_dir, state);
36              if (!mi_trace_reflection(result, state, &reflect_dir))
37                  mi_trace_environment(result, state, &reflect_dir);
38          }
39      } else {
40          result->r = result->g = result->b = 0.0;
41          while (mi_sample(sample, &sample_number, state, 2, &sample_count)) {
42              if (mi_transmission_dir_glossy_x(&refraction_dir, state,
43                                               state->ior_in, state->ior,
44                                               shiny, sample)) {
45                  if (mi_trace_refraction(&refract_color, state, &refraction_dir))
46                      miaux_add_color(result, &refract_color);
47              }
48              else {
49                  mi_reflection_dir(&reflect_dir, state);
50                  if (!mi_trace_reflection(&reflect_color, state, &reflect_dir))
51                      mi_trace_environment(&reflect_color, state, &reflect_dir);
52                  miaux_add_color(result, &reflect_color);
53              }
54          }
55          miaux_scale_color(result, 1.0 / sample_count);
56      }
57      return miTRUE;
58  }
```

Source code of shader "glossy_refraction_sample_varying"

Mixing reflection and refraction in a Phenomenon

Mixing reflection and refraction in a Phenomenon

- Shaders that contain many components are called *monolithic*

Mixing reflection and refraction in a Phenomenon

- Shaders that contain many components are called *monolithic*
- Phenomena can combine smaller shaders, but result looks like a shader, too

Mixing reflection and refraction in a Phenomenon

- Shaders that contain many components are called *monolithic*
- Phenomena can combine smaller shaders, but result looks like a shader, too
- Shader combination requires *arithmetic operator* shaders

Arithmetic operator shaders

Arithmetic operator shaders

- Arithmetic operations by combining constant values or the outputs of other shaders

Arithmetic operator shaders

- Arithmetic operations by combining constant values or the outputs of other shaders
- Addition, subtraction, multiplication, divisions, etc.

Arithmetic operator shaders

- Arithmetic operations by combining constant values or the outputs of other shaders
- Addition, subtraction, multiplication, divisions, etc.
- Arguments are scalars, colors, or vectors

Arithmetic operator shaders

Arithmetic operator shaders

Consistent naming scheme:

Arithmetic operator shaders

Consistent naming scheme:

op + *operation* + *argument types*

Arithmetic operator shaders

Consistent naming scheme:

op + *operation* + *argument types*

For example:

Arithmetic operator shaders

Consistent naming scheme:

op + *operation* + *argument types*

For example:

`op_mul_cc` - Multiply two colors

Arithmetic operator shaders

Consistent naming scheme:

op + *operation* + *argument types*

For example:

`op_mul_cc` - Multiply two colors

A scalar argument with a color or a vector is used for all components

Arithmetic operator shaders

Consistent naming scheme:

op + *operation* + *argument types*

For example:

`op_mul_cc` - Multiply two colors

A scalar argument with a color or a vector is used for all components

`op_mul_cs` - Multiply all the components of a color by a scalar

Arithmetic operator shaders

- Consistent naming scheme:
  - op + *operation* + *argument types*
- For example:
  - `op_mul_cc` - Multiply two colors
- A scalar argument with a color or a vector is used for all components
  - `op_mul_cs` - Multiply all the components of a color by a scalar
- Documented as part of the website for the new shader book in the "Additional shaders" section

Arithmetic operator shaders

- Consistent naming scheme:
  - op + *operation* + *argument types*
- For example:
  - `op_mul_cc` - Multiply two colors
- A scalar argument with a color or a vector is used for all components
  - `op_mul_cs` - Multiply all the components of a color by a scalar
- Documented as part of the website for the new shader book in the "Additional shaders" section
  - `http://www.writingshaders.com/`

Refraction

Combining reflection and refraction with Phenomena

```
declare shader
    color "op_mix_ccc" (
        color "A",
        color "B",
        color "F" )
end declare
```

Scene file declaration of shader "op_mix_ccc"

```
1  struct op_mix_ccc {
2      miColor A;
3      miColor B;
4      miColor F;
5  };
6
7  miBoolean op_mix_ccc(miColor *result, miState *state, struct op_mix_ccc *params)
8  {
9      miColor *A = mi_eval_color(&params->A);
10     miColor *B = mi_eval_color(&params->B);
11     miColor *F = mi_eval_color(&params->F);
12     result->r = miaux_blend(A->r, B->r, F->r);
13     result->g = miaux_blend(A->g, B->g, F->g);
14     result->b = miaux_blend(A->b, B->b, F->b);
15     return miTRUE;
16 }
```

Source code of shader "op_mix_ccc"

Refraction

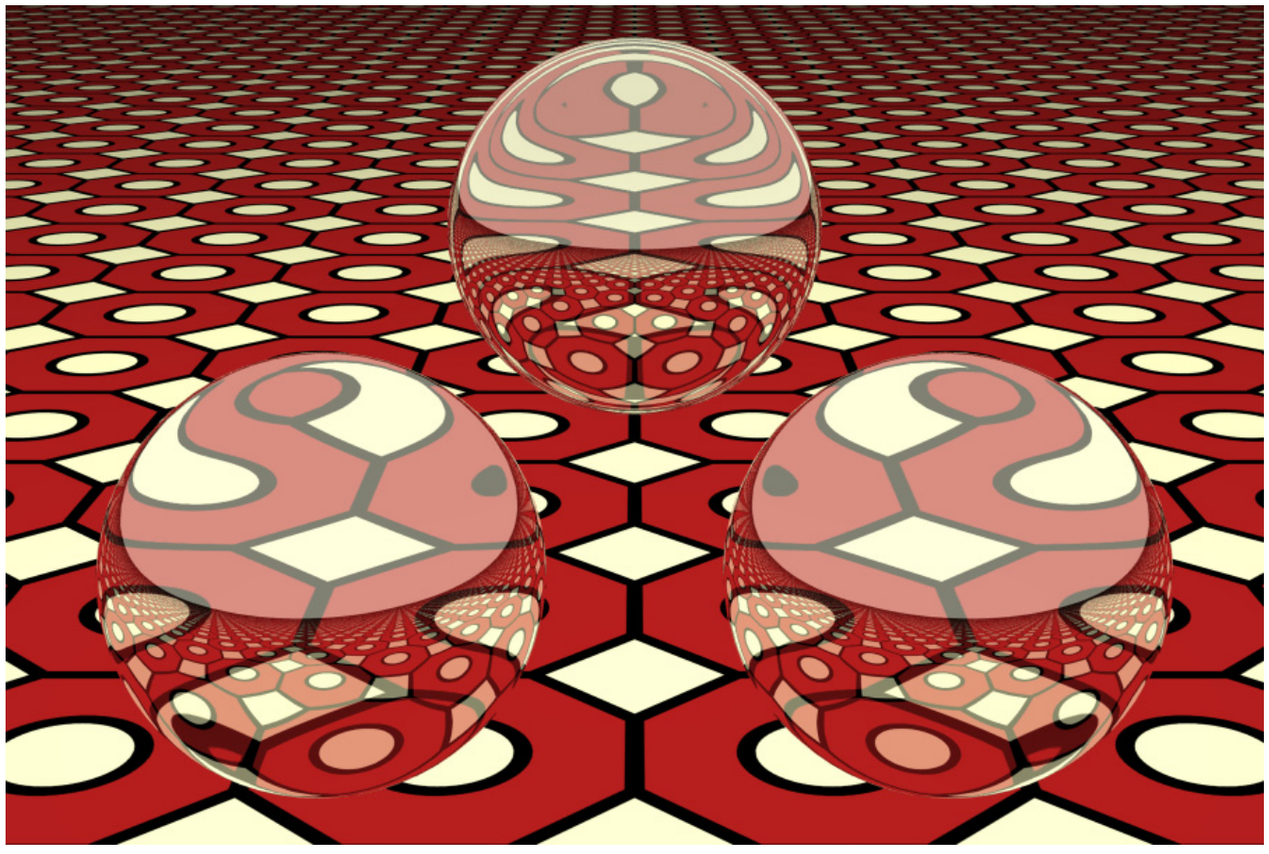
Combining reflection and refraction with Phenomena

```
declare phenomenon
    color "reflect_and_refract" (
        scalar "ior" default 1.05 )
    shader "reflect" "specular_reflection" ()
    shader "refract" "specular_refraction" (
        "index_of_refraction" = interface "ior" )
    shader "blend" "op_mix_ccc" (
        "A" = "refract",
        "B" = "reflect",
        "F" 0.5 0.5 0.5 )
    root = "blend"
end declare
```

Phenomenon reflect_and_refract

Refraction

Combining reflection and refraction with Phenomena

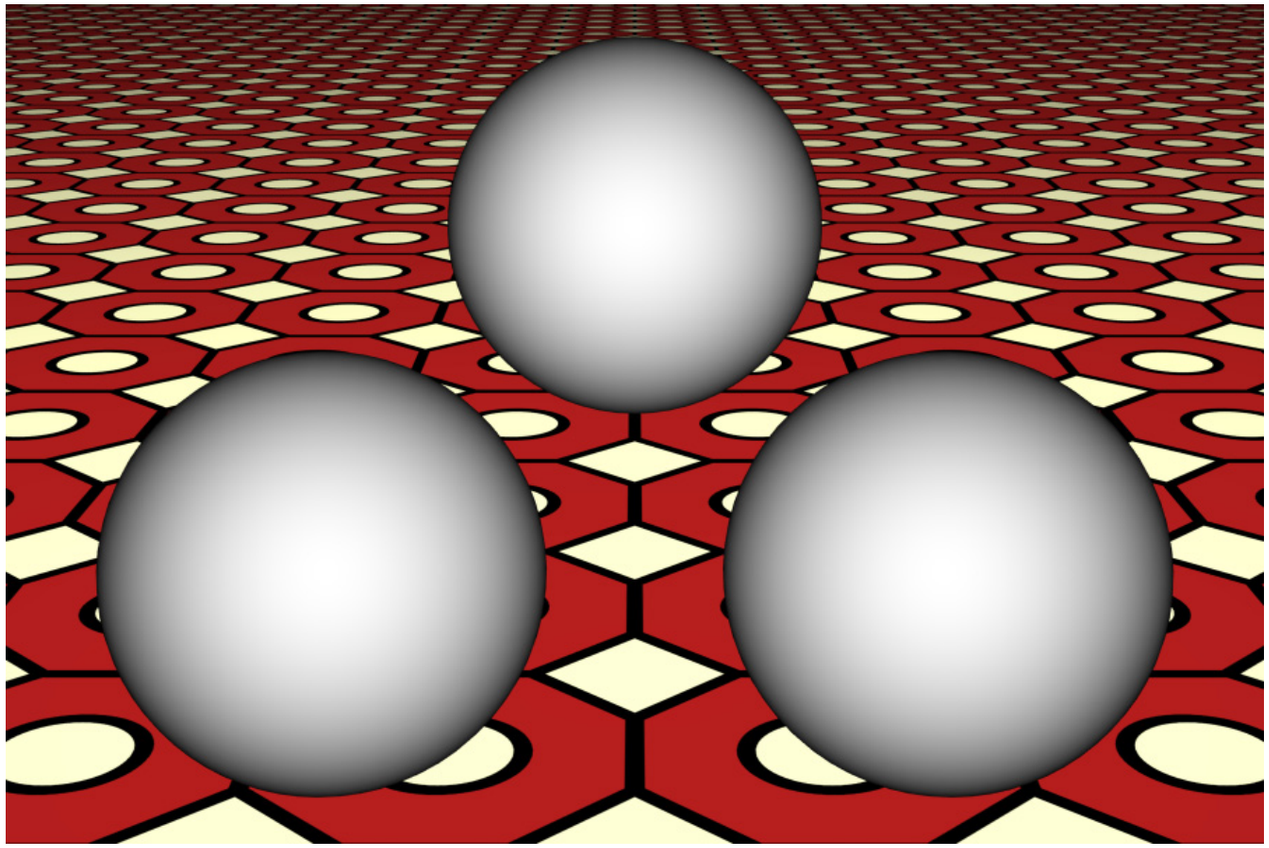


```
material "glass"
    "reflect_and_refract" ()
end material
```

Blending reflection and refraction with a Phenomenon

Refraction

Combining reflection and refraction with Phenomena



```
material "blending_weight"
    "front_bright" ()
end material
```

Shader front_bright to be used as the blending weight between reflection and refraction

Refraction

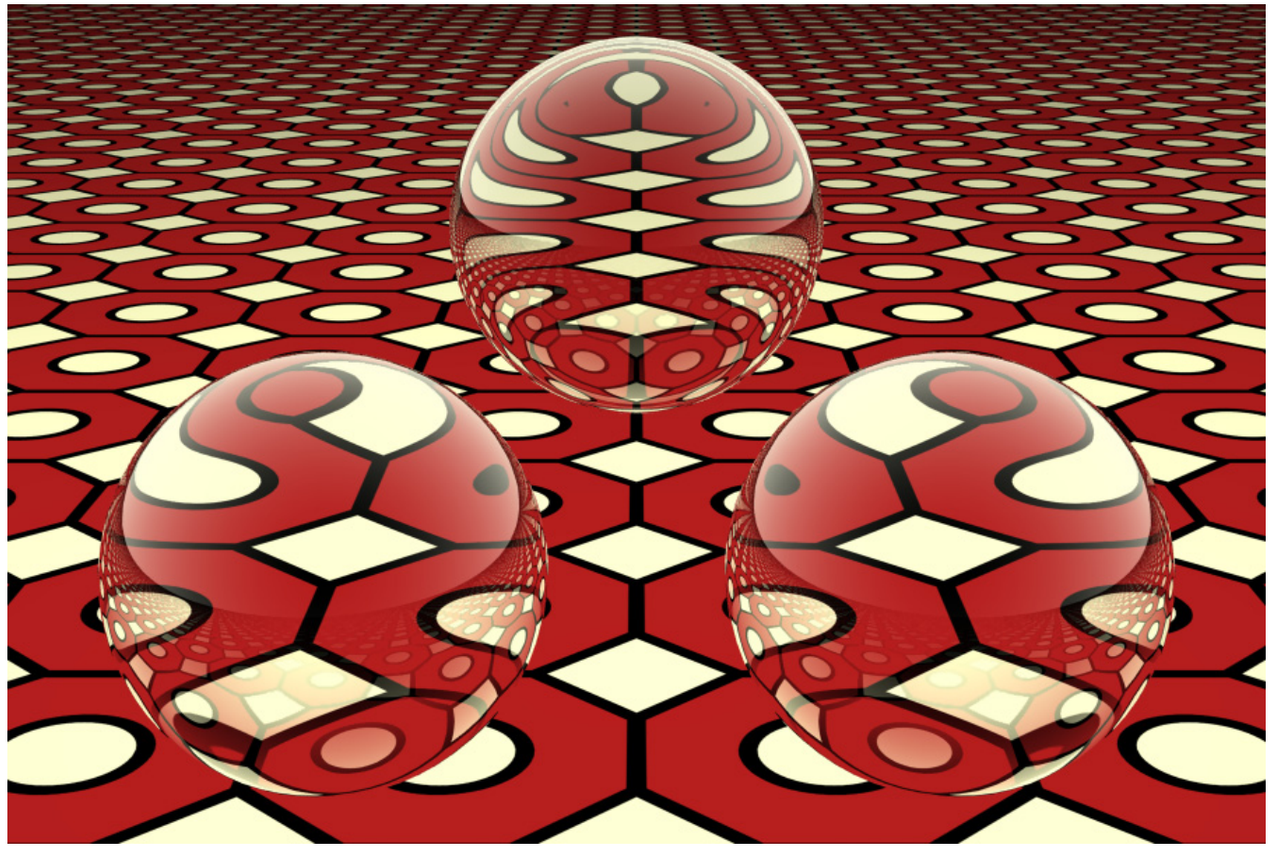
Combining reflection and refraction with Phenomena

```
declare phenomenon
    color "shiny_edge" (
        scalar "ior" default 1.05 )
    shader "reflect" "specular_reflection" ()
    shader "refract" "specular_refraction" (
        "index_of_refraction" = interface "ior" )
    shader "front" "front_bright" ()
    shader "blend" "op_mix_ccc" (
        "A" = "refract",
        "B" = "reflect",
        "F" = "front" )
    root = "blend"
end declare
```

Phenomenon shiny_edge

Refraction

Combining reflection and refraction with Phenomena



```
material "glass"
    "shiny_edge" ()
end material
```

Controlling the amount of blending with shader front_bright

```
declare shader
    color "op_pow_cs" (
        color "A",
        scalar "B" )
end declare
```

Scene file declaration of shader "op_pow_cs"

```
1  struct op_pow_cs {
2      miColor A;
3      miScalar B;
4  };
5
6  miBoolean op_pow_cs(miColor *result, miState *state, struct op_pow_cs *params)
7  {
8      miColor *A = mi_eval_color(&params->A);
9      miScalar B = *mi_eval_scalar(&params->B);
10     result->r = powf(A->r, B);
11     result->g = powf(A->g, B);
12     result->b = powf(A->b, B);
13     return miTRUE;
14 }
```

Source code of shader "op_pow_cs"

Refraction

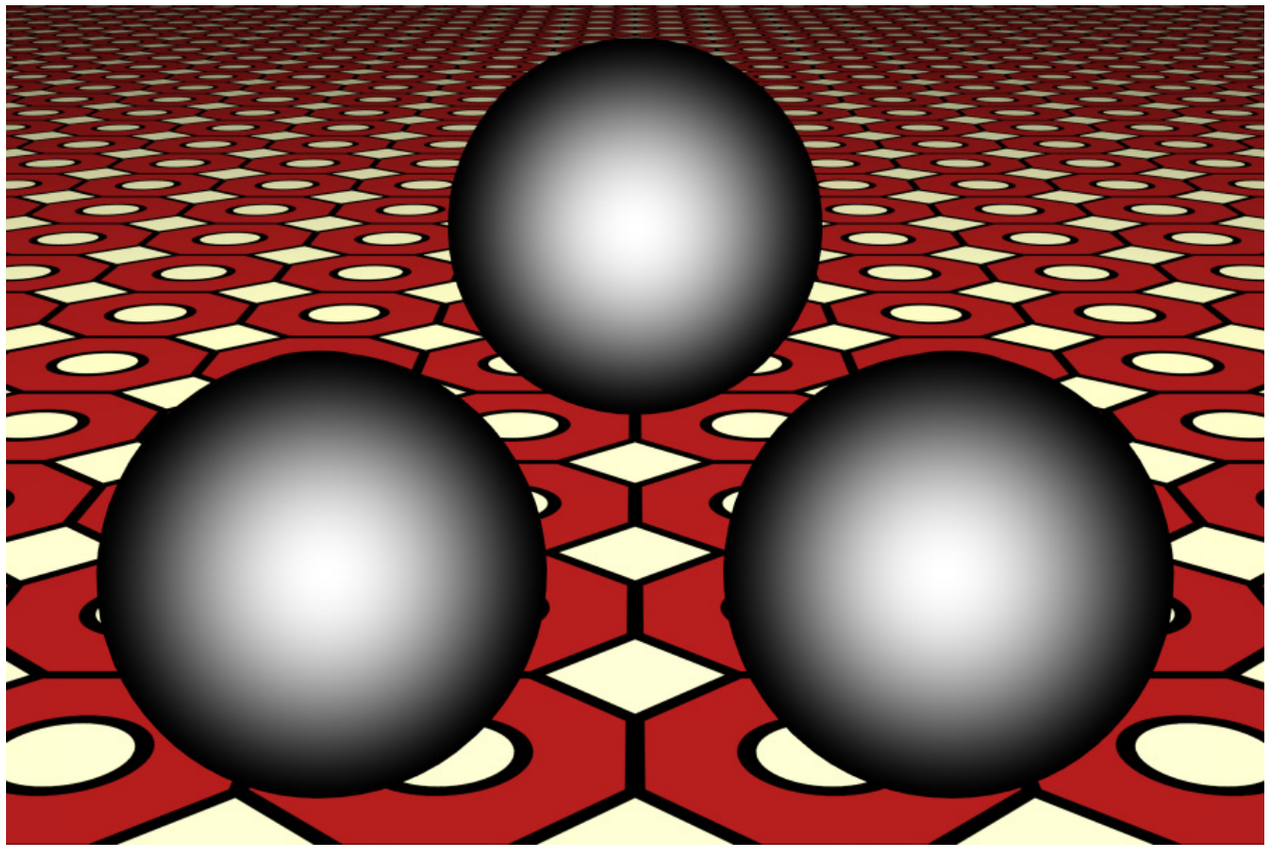
Combining reflection and refraction with Phenomena

```
declare phenomenon
    color "thick_edge" (
        scalar "thickness" default 1.0 )
    shader "front" "front_bright" ()
    shader "edge" "op_pow_cs" (
        "A" = "front",
        "B" = interface "thickness" )
    root = "edge"
end declare
```

Phenomenon thick_edge

Refraction

Combining reflection and refraction with Phenomena



```
material "edge"
    "thick_edge" (
        "thickness" 3 )
end material
```

Modifying the front_bright value with op_pow_cs in Phenomenon thick_edge

Refraction

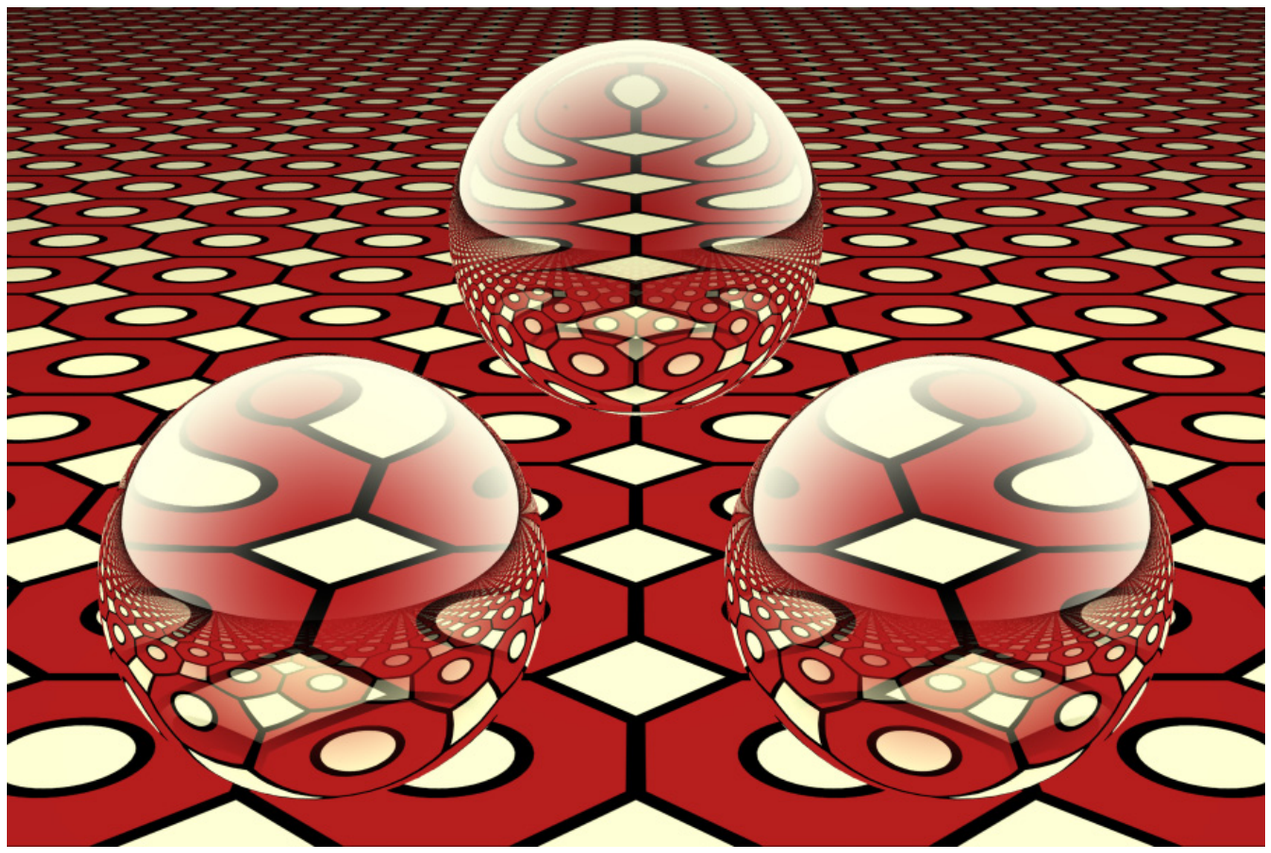
Combining reflection and refraction with Phenomena

```
declare phenomenon
    color "shiny_thick_edge" (
        scalar "ior" default 1.05,
        scalar "thickness" default 1.0 )
    shader "reflect" "specular_reflection" ()
    shader "refract" "specular_refraction" (
        "index_of_refraction" = interface "ior" )
    shader "front" "front_bright" ()
    shader "edge" "op_pow_cs" (
        "A" = "front",
        "B" = interface "thickness" )
    shader "blend" "op_mix_ccc" (
        "A" = "refract",
        "B" = "reflect",
        "F" = "edge" )
    root = "blend"
end declare
```

Phenomenon shiny_thick_edge

Refraction

Combining reflection and refraction with Phenomena



```
material "glass"
    "shiny_thick_edge" (
        "thickness" 3 )
end material
```

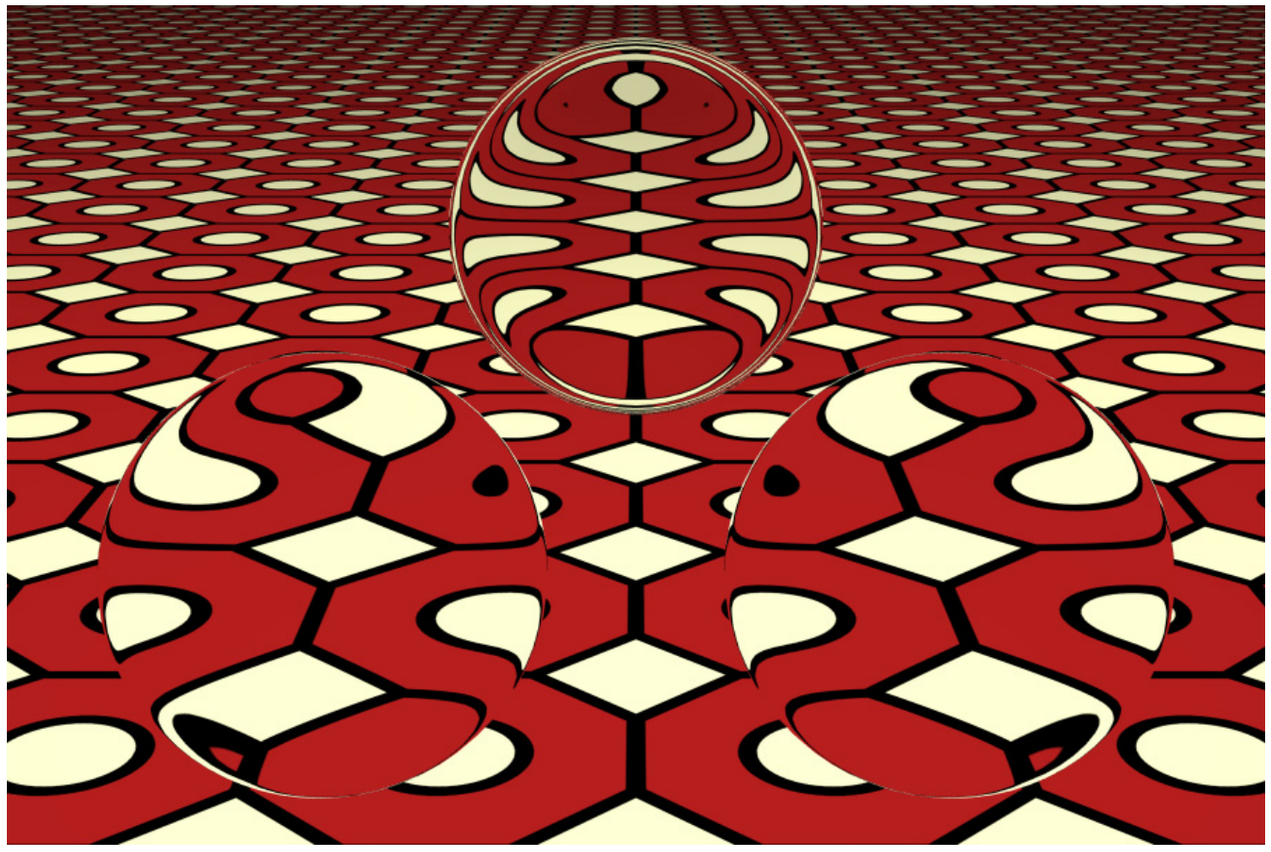
Modifying the front_bright value to affect the blending weight

```
declare phenomenon
    color "shiny_thick_edge" (
        scalar "ior" default 1.05,
        scalar "thickness" default 1.0 )
    shader "reflect" "specular_reflection" ()
    shader "refract" "specular_refraction" (
        "index_of_refraction" = interface "ior" )
    shader "front" "front_bright" ()
    shader "edge" "op_pow_cs" (
        "A" = "front",
        "B" = interface "thickness" )
    shader "blend" "op_mix_ccc" (
        "A" = "refract",
        "B" = "reflect",
        "F" = "edge" )
    # root = "blend"
    root = "refract"
end declare
```

A Phenomenon with the root shader replaced by an intermediate shader in the graph

Refraction

Combining reflection and refraction with Phenomena



```
material "glass"
    "shiny_thick_edge" (
        "thickness" 3 )
end material
```

Rendering intermediate values in the shader graph by redefining the root shader

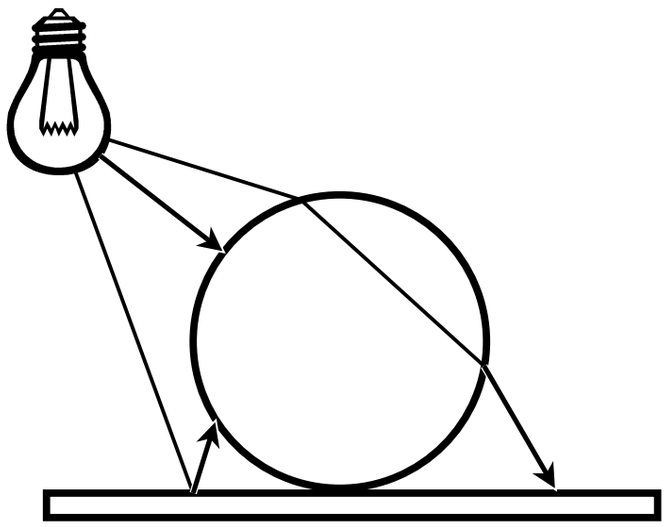
Exercise 13: Combining reflection and refraction

1. Render `phenomena_B.mi`.
2. Examine Phenomenon `shiny_clear` in `phenomena_B.mi`, identifying what each shader does.
3. Look at the Phenomenon declared at the end of `phenomena_B.mi` called `shiny_edge`. Find where the edge has been made more opaque by shader `op_pow_cs`.
4. Replace `shiny_clear` with `shiny_edge` and re-render.

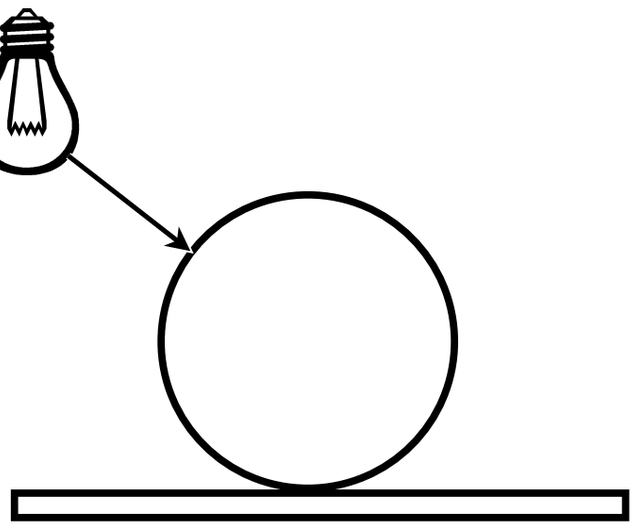
Light from other surfaces

Light from other surfaces

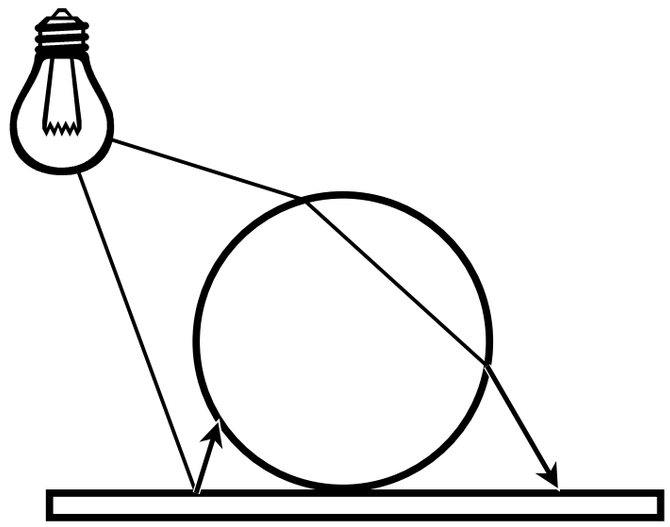
- Using the photon map for global illumination
  - Adding global illumination to the Lambert shader
  - The photon shader
  - Global illumination statements in the options block
- Global illumination as the ambient parameter
  - Overriding global illumination options in a shader
- Final gathering
- Caustics
- Visualizing the photon map
- Ambient occlusion
  - Sampling the environment by tracing rays to detect occlusion
  - Restricting ray length in occlusion detection



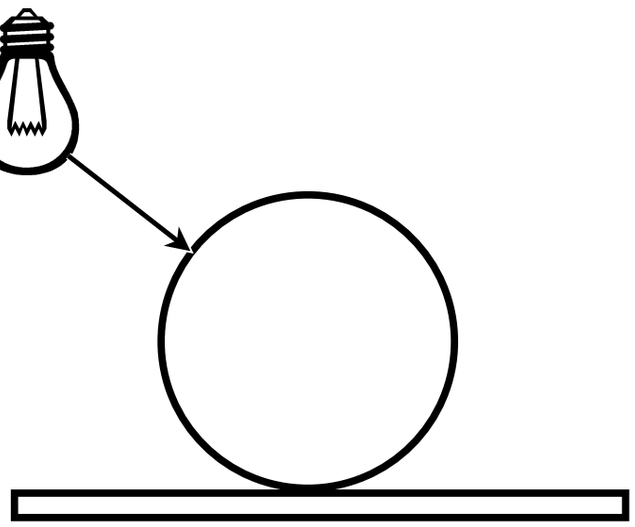
Global illumination



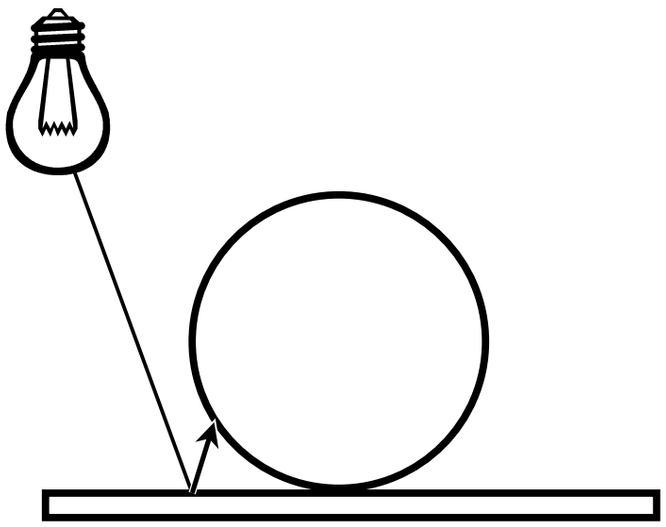
Direct illumination



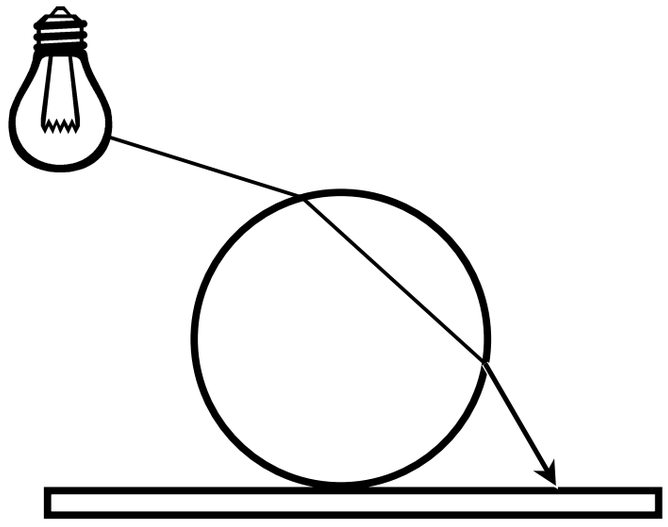
Global illumination



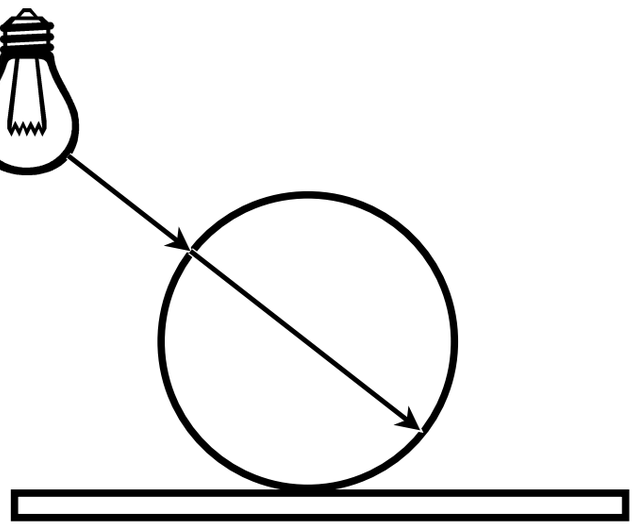
Direct illumination



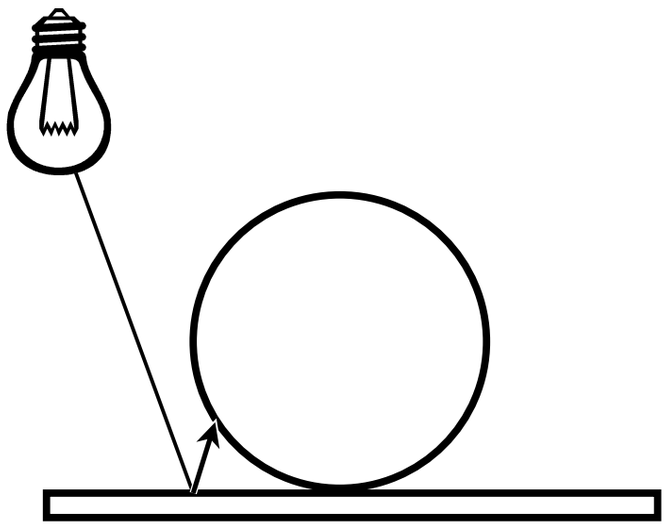
Global illumination



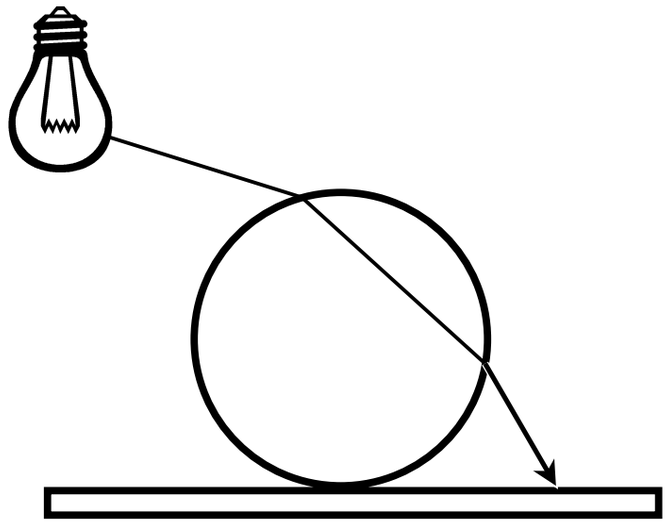
Caustics



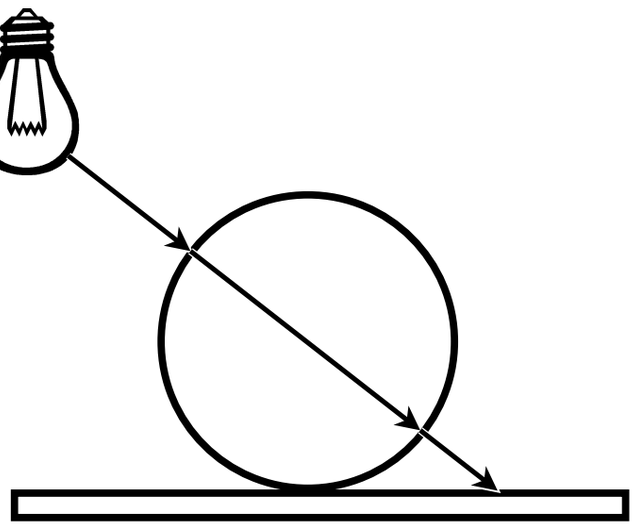
Direct illumination



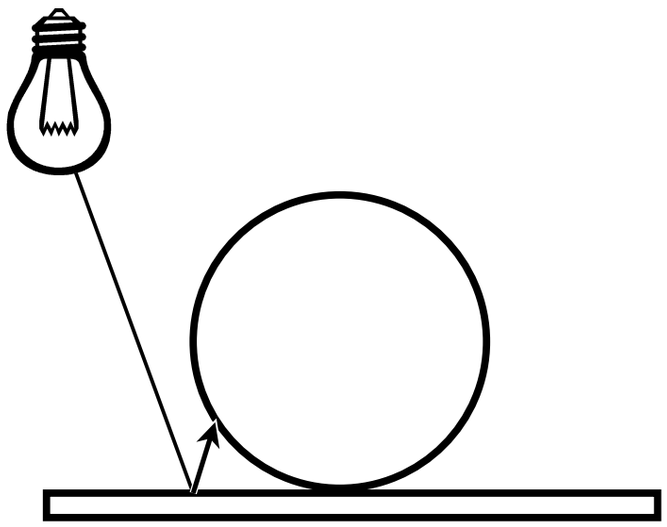
Global illumination



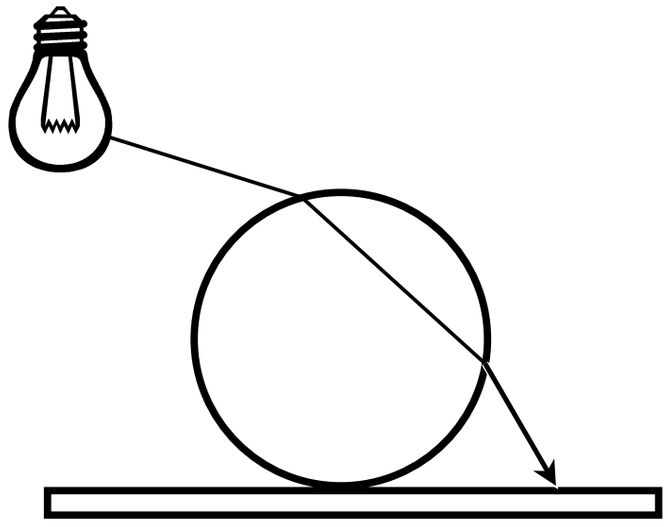
Caustics



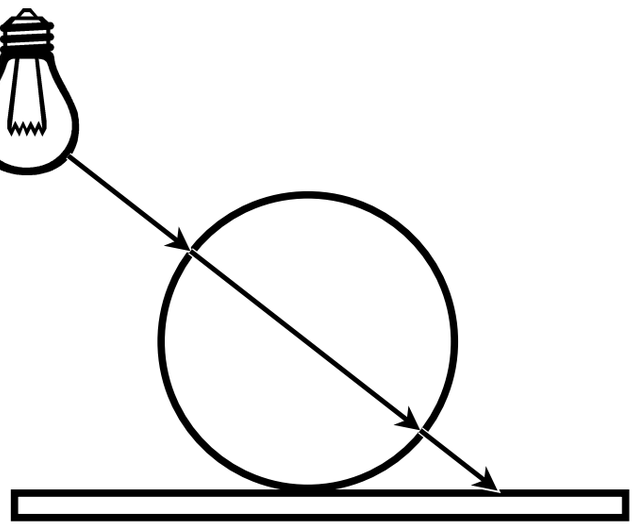
Direct illumination



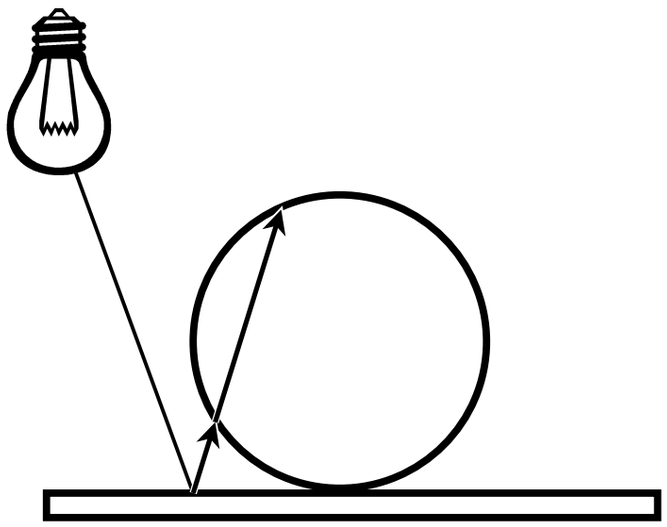
Global illumination



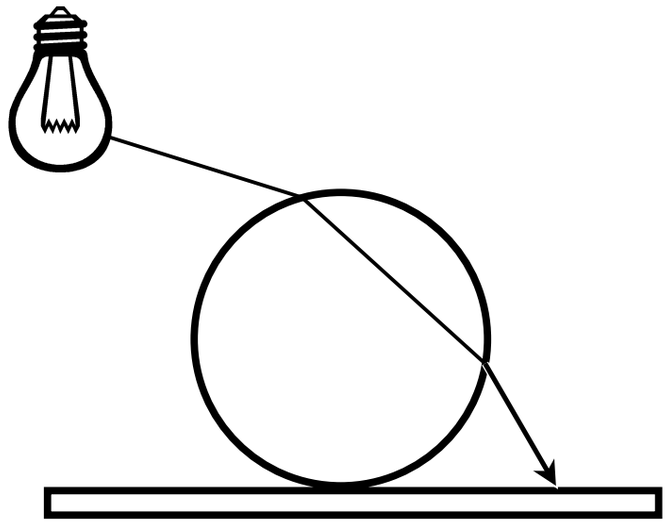
Caustics



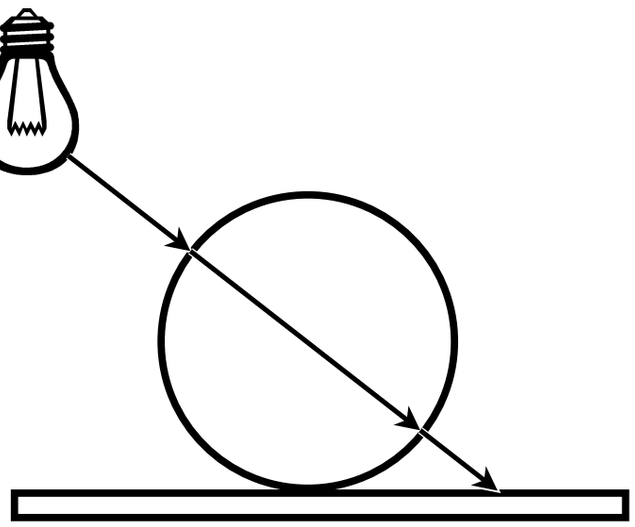
Direct illumination



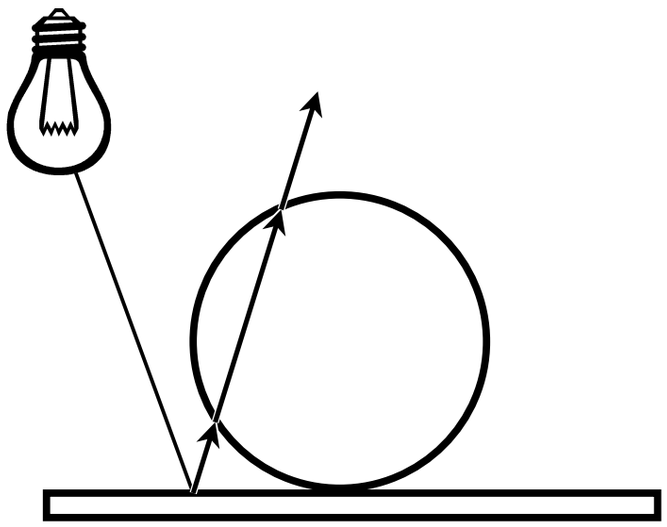
Global illumination



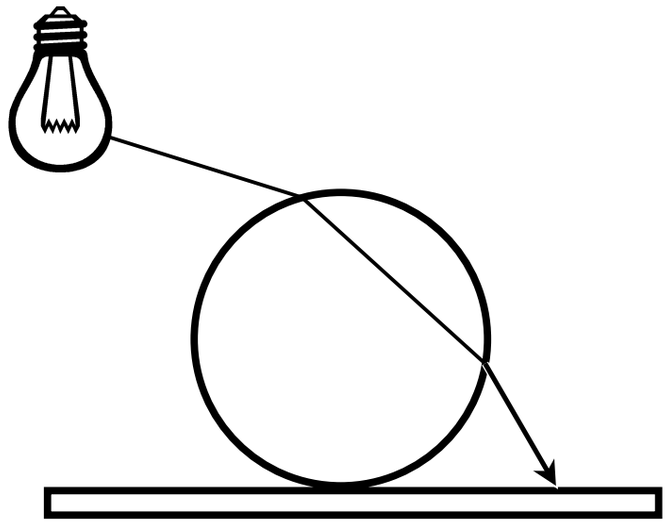
Caustics



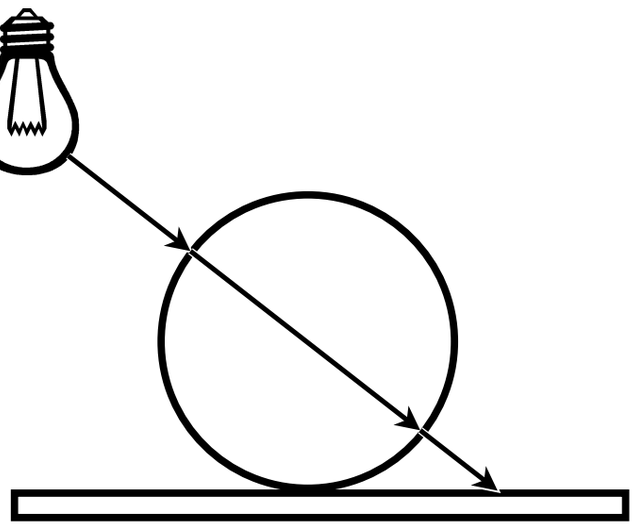
Direct illumination



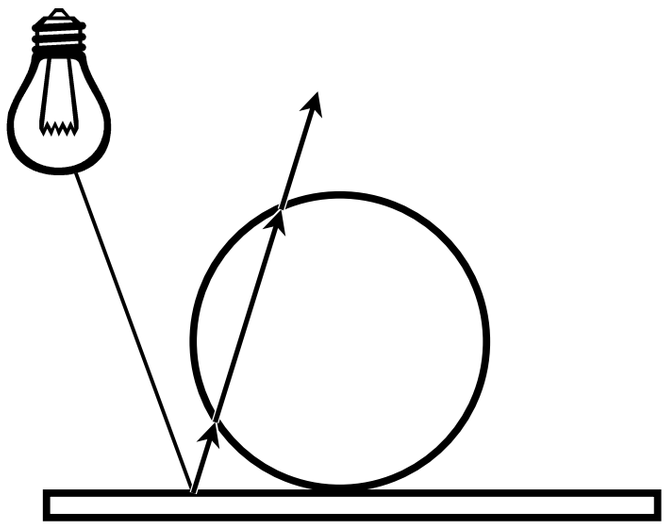
Global illumination



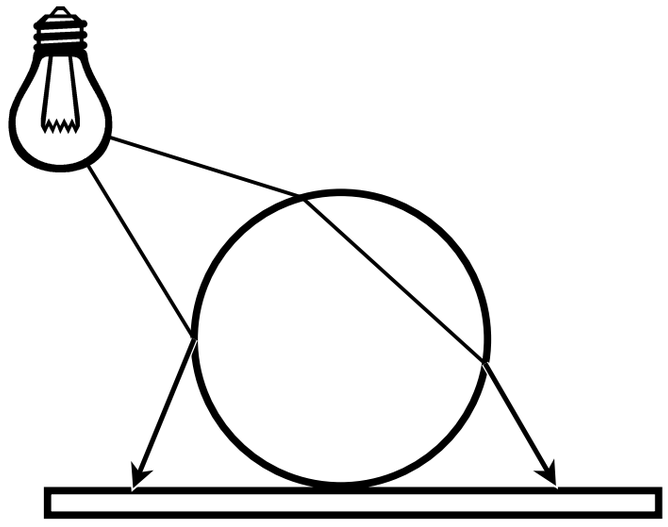
Caustics



Direct illumination



Global illumination



Caustics

Light from other surfaces

Direct	diffuse	reflection
		transmission
	glossy	reflection
		transmission
	specular	reflection
		transmission
Indirect	diffuse	reflection
		transmission
	glossy	reflection
		transmission
	specular	reflection
		transmission

Combinations of light categories

Light from other surfaces

Diffuse	direct	reflection
		transmission
	indirect	reflection
		transmission
Glossy	direct	reflection
		transmission
	indirect	reflection
		transmission
Specular	direct	reflection
		transmission
	indirect	reflection
		transmission

Combinations of light categories

Light from other surfaces

Direct	reflection	diffuse
		glossy
		specular
	transmission	diffuse
		glossy
		specular
Indirect	reflection	diffuse
		glossy
		specular
	transmission	diffuse
		glossy
		specular

Combinations of light categories

Light from other surfaces

Diffuse	reflection	direct
		indirect
	transmission	direct
		indirect
Glossy	reflection	direct
		indirect
	transmission	direct
		indirect
Specular	reflection	direct
		indirect
	transmission	direct
		indirect

Combinations of light categories

Light from other surfaces

Reflection	direct	diffuse
		glossy
		specular
	indirect	diffuse
		glossy
		specular
Transmission	direct	diffuse
		glossy
		specular
	indirect	diffuse
		glossy
		specular

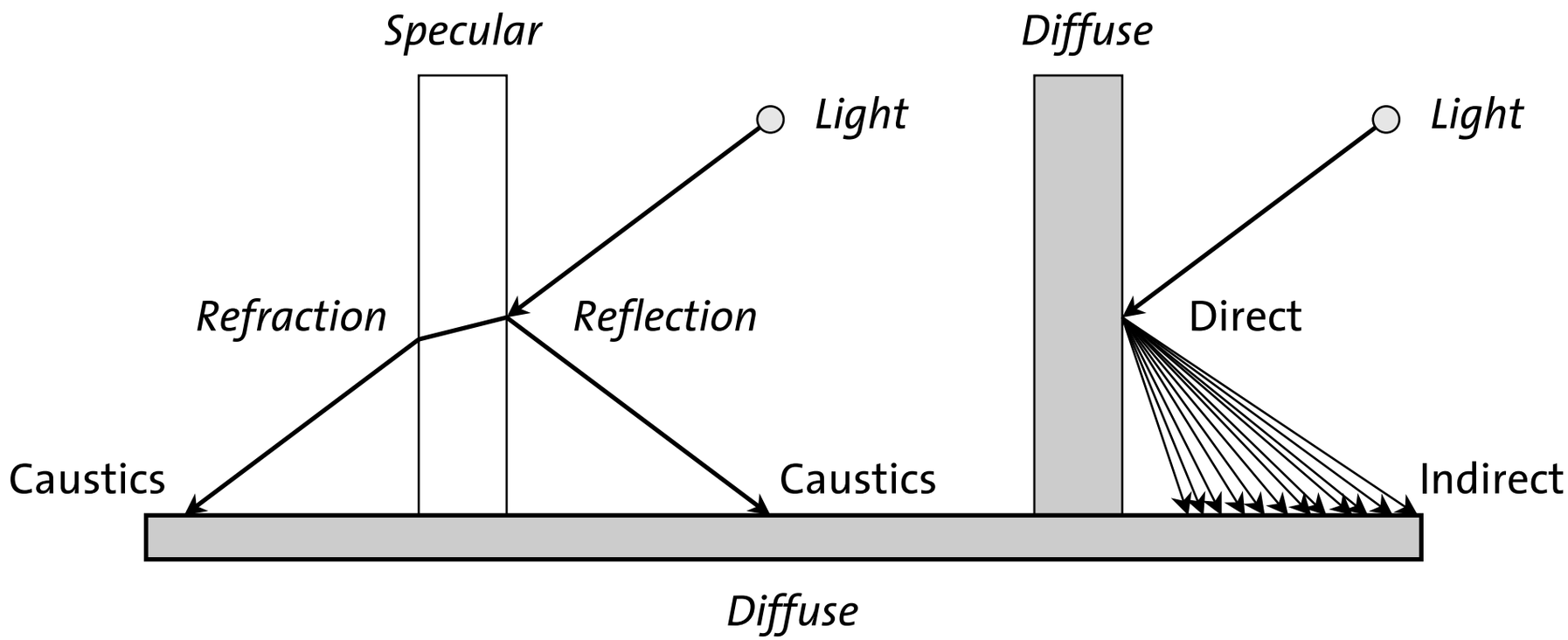
Combinations of light categories

Light from other surfaces

Reflection	diffuse	direct
		indirect
	glossy	direct
		indirect
	specular	direct
		indirect
Transmission	diffuse	direct
		indirect
	glossy	direct
		indirect
	specular	direct
		indirect

Combinations of light categories

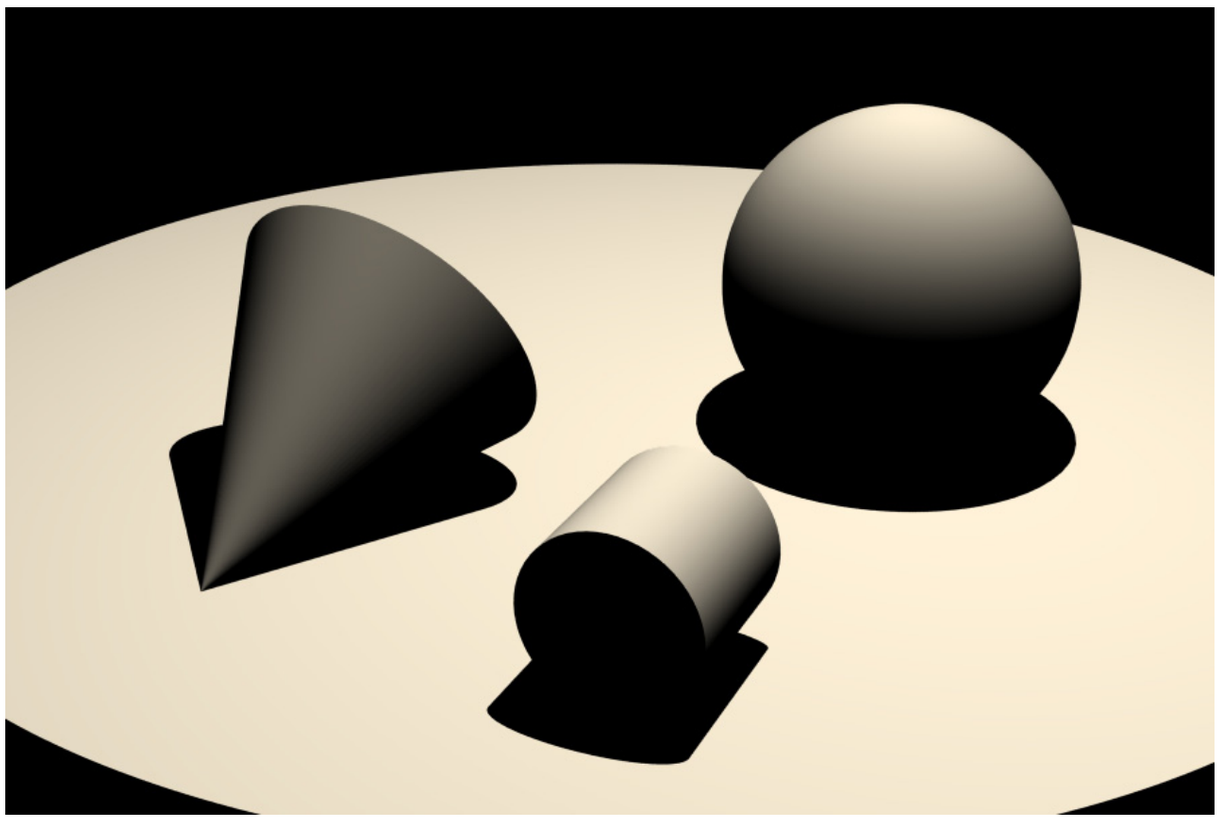
Light from other surfaces



Light categories in mental ray and their relationship to material types

Light from other surfaces

Using the photon map for global illumination



```
light "light"
    "spotlight" ()
    origin 1 5 0
    direction -.14 -1 -.1
    spread .912
end light

instance "light_inst" "light"
end instance

material "diffuse"
    "lambert" (
        "diffuse" 1 .95 .85,
        "lights" ["light_inst"] )
end material
```

Lambert shading only

```
declare shader
    color "global_lambert" (
        color "ambient" default 0 0 0,
        color "diffuse" default 1 1 1 1,
        array light "lights" )
end declare
```

Scene file declaration of shader "global_lambert"

Light from other surfaces

Using the photon map for global illumination

```
 1  struct global_lambert {
 2      miColor ambient;
 3      miColor diffuse;
 4      int     i_light;
 5      int     n_light;
 6      miTag   light[1];
 7  };
 8
 9  miBoolean global_lambert (
10      miColor *result, miState *state, struct global_lambert *params )
11  {
12      int i, light_count, light_sample_count;
13      miColor sum, light_color, global_illumination_color;
14      miScalar dot_nl;
15      miTag *light;
16
17      miColor *diffuse  = mi_eval_color(&params->diffuse);
18      miaux_light_array(&light, &light_count, state,
19                        &params->i_light, &params->n_light, params->light);
20      *result = *mi_eval_color(&params->ambient);
21
22      for (i = 0; i < light_count; i++, light++) {
23          miaux_set_channels(&sum, 0);
24          light_sample_count = 0;
25          while (mi_sample_light(&light_color, NULL, &dot_nl,
26                                 state, *light, &light_sample_count))
27              miaux_add_diffuse_component(&sum, dot_nl, diffuse, &light_color);
28          if (light_sample_count)
29              miaux_add_scaled_color(result, &sum, 1.0/light_sample_count);
30      }
31      mi_compute_avg_radiance(&global_illumination_color, state, 'f', NULL);
32      miaux_add_multiplied_colors(result, &global_illumination_color, diffuse);
33      result->a = 1.0;
34
35      return miTRUE;
36  }
```

Source code of shader "global_lambert"

```
declare shader
    color "store_diffuse_photon" (
        color "diffuse_color" default 1 1 1 )
end declare
```

Scene file declaration of shader "store_diffuse_photon"

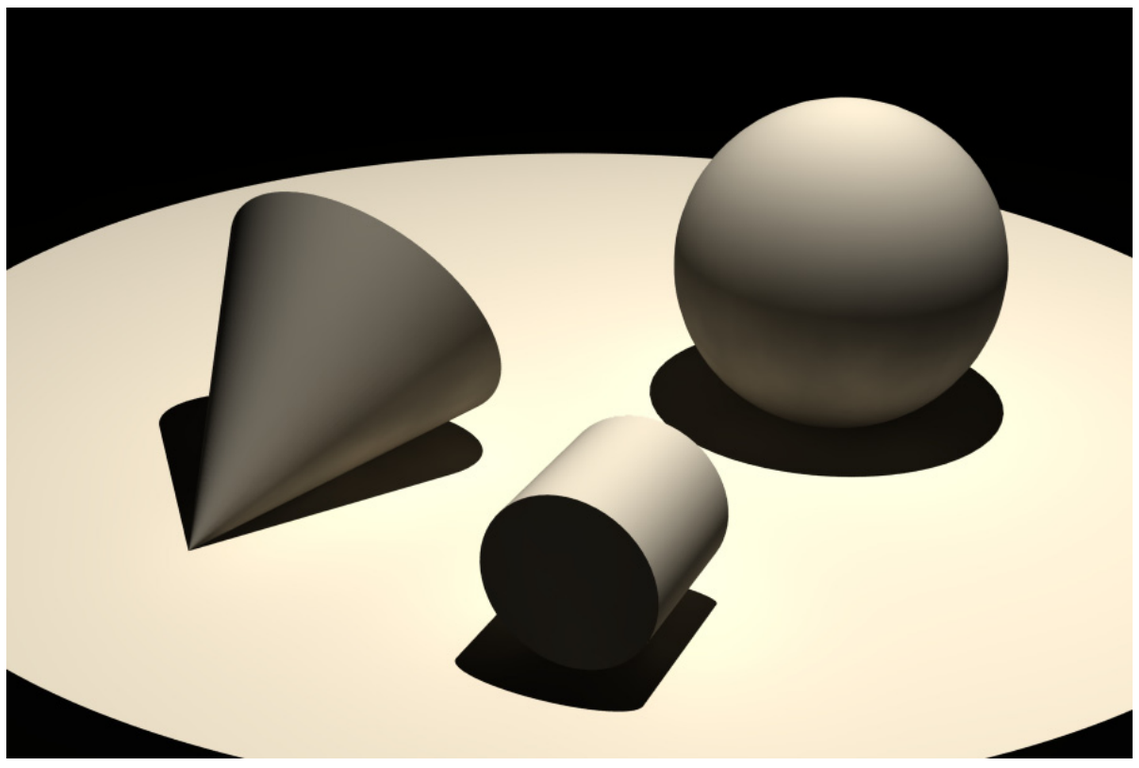
Light from other surfaces

Using the photon map for global illumination

```
1  struct store_diffuse_photon {
2      miColor diffuse_color;
3  };
4
5  miBoolean store_diffuse_photon (
6      miColor *result, miState *state, struct store_diffuse_photon *params )
7  {
8      miVector diffuse_direction;
9      miColor *diffuse_color = mi_eval_color(&params->diffuse_color);
10
11     mi_store_photon(result, state);
12
13     miaux_multiply_color(result, diffuse_color);
14     mi_reflection_dir_diffuse(&diffuse_direction, state);
15     return mi_photon_reflection_diffuse(result, state, &diffuse_direction);
16 }
```

Source code of shader "store_diffuse_photon"

Light from other surfaces



Using the photon map for global illumination

```
options "opt"
    object space
    contrast .1 .1 .1 1
    samples 0 2
    globillum on
    globillum accuracy 500 2
    shadow on
end options

light "light"
    "spotlight" ()
    origin 1 5 0
    direction -.14 -1 -.1
    spread .912
    energy 600 600 600
    globillum photons 50000
end light

instance "light_inst" "light"
end instance

material "global_diffuse"
    "global_lambert" (
        "diffuse" 1 .95 .85,
        "lights" ["light_inst"] )
    photon
        "store_diffuse_photon" (
            "diffuse_color" 1 .95 .85 )
end material
```

Lambert shading with global illumination

Light from other surfaces

Global illumination as the ambient parameter

```
declare shader
    color "average_radiance" (
        color "color" default 1 1 1 )
end declare
```

Scene file declaration of shader "average_radiance"

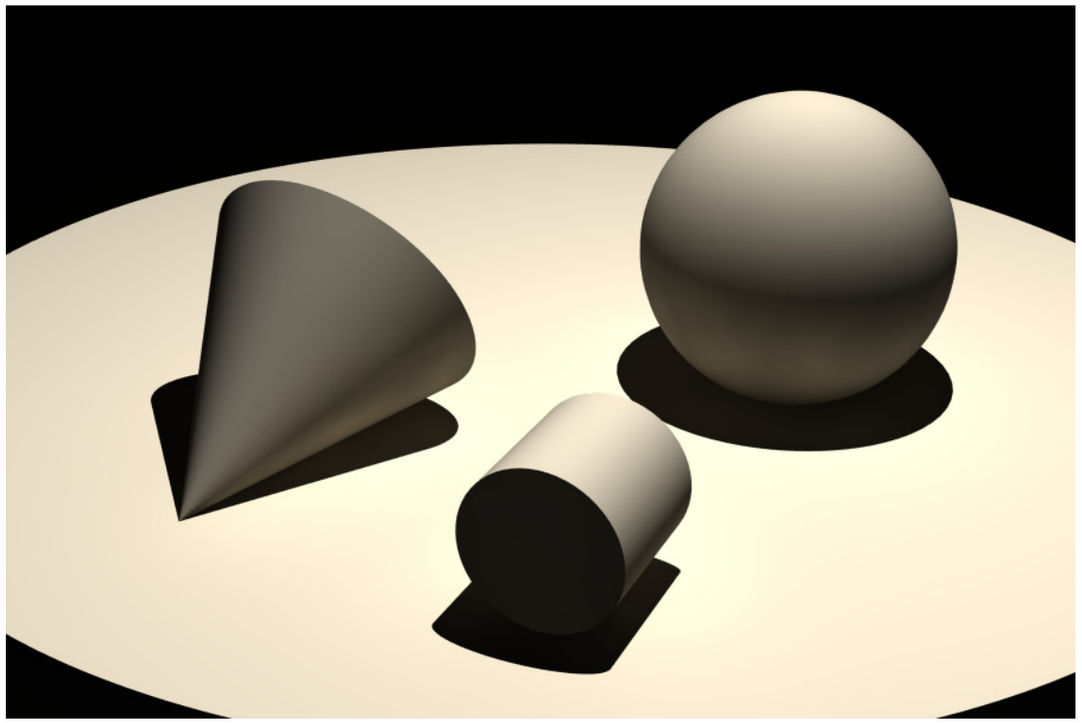
Light from other surfaces

Global illumination as the ambient parameter

```
1  struct average_radiance {
2      miColor color;
3  };
4
5  miBoolean average_radiance (
6      miColor *result, miState *state, struct average_radiance *params )
7  {
8      miColor *color = mi_eval_color(&params->color);
9      mi_compute_avg_radiance(result, state, 'f', NULL);
10     miaux_multiply_color(result, color);
11     return miTRUE;
12 }
```

Source code of shader "average_radiance"

Light from other surfaces



Global illumination as the ambient parameter

```
options "opt"
    object space
    contrast .1 .1 .1 1
    samples 0 2
    globillum on
    globillum accuracy 500 2
    shadow on
end options

shader "radiance"
    "average_radiance" (
        "color" 1 .95 .85 )

light "light"
    "spotlight" ()
    origin 1 5 0
    direction -.14 -1 -.1
    spread .912
    energy 600 600 600
    globillum photons 50000
end light

instance "light_inst" "light"
end instance

material "global_diffuse"
    "lambert" (
        "ambient" = "radiance",
        "diffuse" 1 .95 .85,
        "lights" ["light_inst"] )
    photon
        "store_diffuse_photon" (
            "diffuse_color" 1 .95 .85 )
end material
```

Lambert shading with global illumination shader for ambient value

Light from other surfaces

Global illumination as the ambient parameter

```
#define miIRRAD_DEFAULT(irrad, state) do {                                  \
  (irrad)->size                   = sizeof(miIrrad_options);                \
  (irrad)->finalgather_rays       = (state)->options->finalgather_rays;     \
  (irrad)->finalgather_maxradius = (state)->options->finalgather_maxradius; \
  (irrad)->finalgather_minradius = (state)->options->finalgather_minradius; \
  (irrad)->finalgather_view       = (state)->options->finalgather_view;     \
  (irrad)->finalgather_filter     = (state)->options->finalgather_filter;   \
  (irrad)->padding1[0]            = 0;                                      \
  (irrad)->padding1[1]            = 0;                                      \
  (irrad)->globillum_accuracy     = (state)->options->globillum_accuracy;   \
  (irrad)->globillum_radius       = (state)->options->globillum_radius;     \
  (irrad)->caustic_accuracy       = (state)->options->caustic_accuracy;     \
  (irrad)->caustic_radius         = (state)->options->caustic_radius;       \
  (irrad)->finalgather_points     = 0;     /* use the default one */        \
  (irrad)->importance             = -1.0f; /* use the default one */        \
} while(0)
```

The miIRRAD_DEFAULT macro in shader.h for initializing global illumination options

Light from other surfaces

Global illumination as the ambient parameter

```
declare shader
    color "average_radiance_options" (
        color    "color"                  default 1 1 1,
        integer  "globillum_accuracy"     default -1,
        scalar   "globillum_radius"       default -1.0,
        integer  "finalgather_rays"       default -1,
        scalar   "finalgather_maxradius" default -1.0,
        scalar   "finalgather_minradius" default -1.0,
        integer  "finalgather_view"       default -1,
        integer  "finalgather_filter"     default -1,
        integer  "finalgather_points"     default 0,
        scalar   "importance"             default -1.0,
        integer  "caustic_accuracy"       default -1,
        scalar   "caustic_radius"         default -1.0 )
end declare
```

Scene file declaration of shader "average_radiance_options"

```
1  void miaux_set_integer_if_not_default(
2      miInteger *result, miState *state, miInteger *param)
3  {
4      miInteger use_default_flag = -1;
5      miInteger param_value = *mi_eval_integer(param);
6      if (param_value != use_default_flag)
7          *result = param_value;
8  }
```

Auxiliary function: miaux_set_integer_if_not_default

Light from other surfaces

Global illumination as the ambient parameter

```
1  void miaux_set_scalar_if_not_default(
2      miScalar *result, miState *state, miScalar *param)
3  {
4      miScalar use_default_flag = -1.0;
5      miScalar param_value = *mi_eval_scalar(param);
6      if (param_value != use_default_flag)
7          *result = param_value;
8  }
```

Auxiliary function: miaux_set_scalar_if_not_default

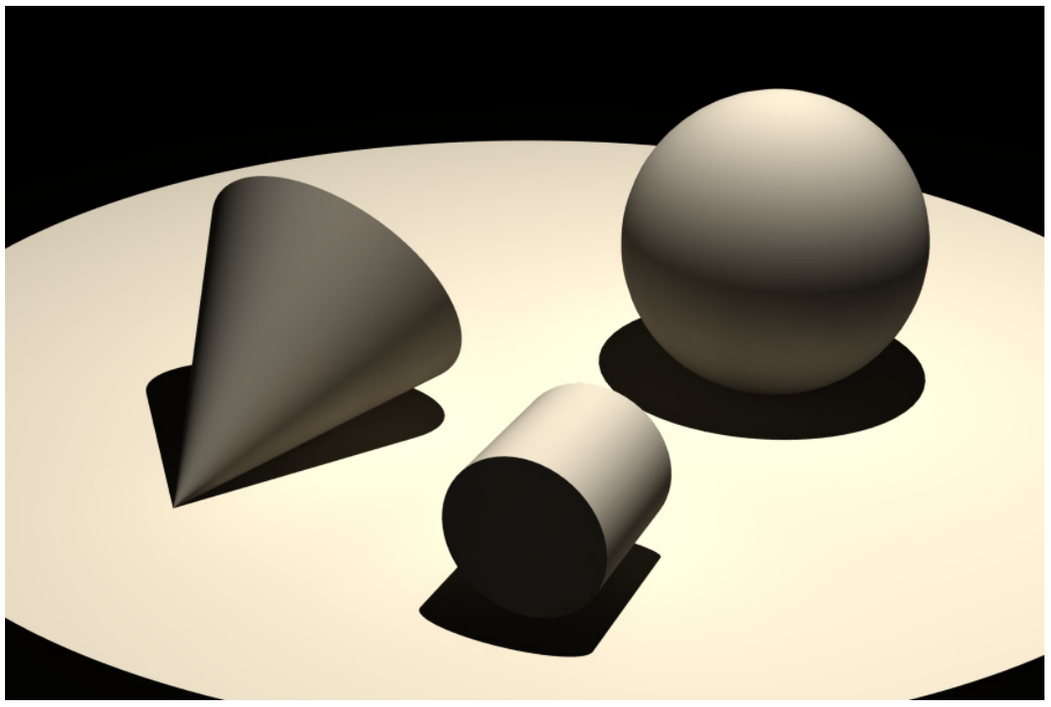
Light from other surfaces

Global illumination as the ambient parameter

```
 1  struct average_radiance_options {
 2      miColor    color;
 3      miInteger  gi_accuracy;
 4      miScalar   gi_radius;
 5      miInteger  fg_rays;
 6      miScalar   fg_maxradius;
 7      miScalar   fg_minradius;
 8      miInteger  fg_view;
 9      miInteger  fg_filter;
10      miInteger  fg_points;
11      miScalar   importance;
12      miInteger  caustic_accuracy;
13      miScalar   caustic_radius;
14  };
15
16  miBoolean average_radiance_options (
17      miColor *result, miState *state, struct average_radiance_options *params )
18  {
19      miColor *color = mi_eval_color(&params->color);
20      miIrrad_options options;
21      miIRRAD_DEFAULT(&options, state);
22
23      miaux_set_integer_if_not_default(
24          &options.globillum_accuracy, state, &params->gi_accuracy);
25      miaux_set_scalar_if_not_default(
26          &options.globillum_radius, state, &params->gi_radius);
27      miaux_set_integer_if_not_default(
28          &options.finalgather_rays, state, &params->fg_rays);
29      miaux_set_scalar_if_not_default(
30          &options.finalgather_maxradius, state, &params->fg_maxradius);
31      miaux_set_scalar_if_not_default(
32          &options.finalgather_minradius, state, &params->fg_minradius);
33      miaux_set_integer_if_not_default(
34          (miInteger*)&options.finalgather_view, state, &params->fg_view);
35      miaux_set_integer_if_not_default(
36          (miInteger*)&options.finalgather_filter, state, &params->fg_filter);
37      miaux_set_integer_if_not_default(
38          (miInteger*)&options.finalgather_points, state, &params->fg_points);
39      miaux_set_scalar_if_not_default(
40          &options.importance, state, &params->importance);
41      miaux_set_integer_if_not_default(
42          &options.caustic_accuracy, state, &params->caustic_accuracy);
43      miaux_set_scalar_if_not_default(
44          &options.caustic_radius, state, &params->caustic_radius);
45
46      mi_compute_avg_radiance(result, state, 'f', &options);
47      miaux_multiply_color(result, color);
48
49      return miTRUE;
50  }
```

Source code of shader "average_radiance_options"

Light from other surfaces



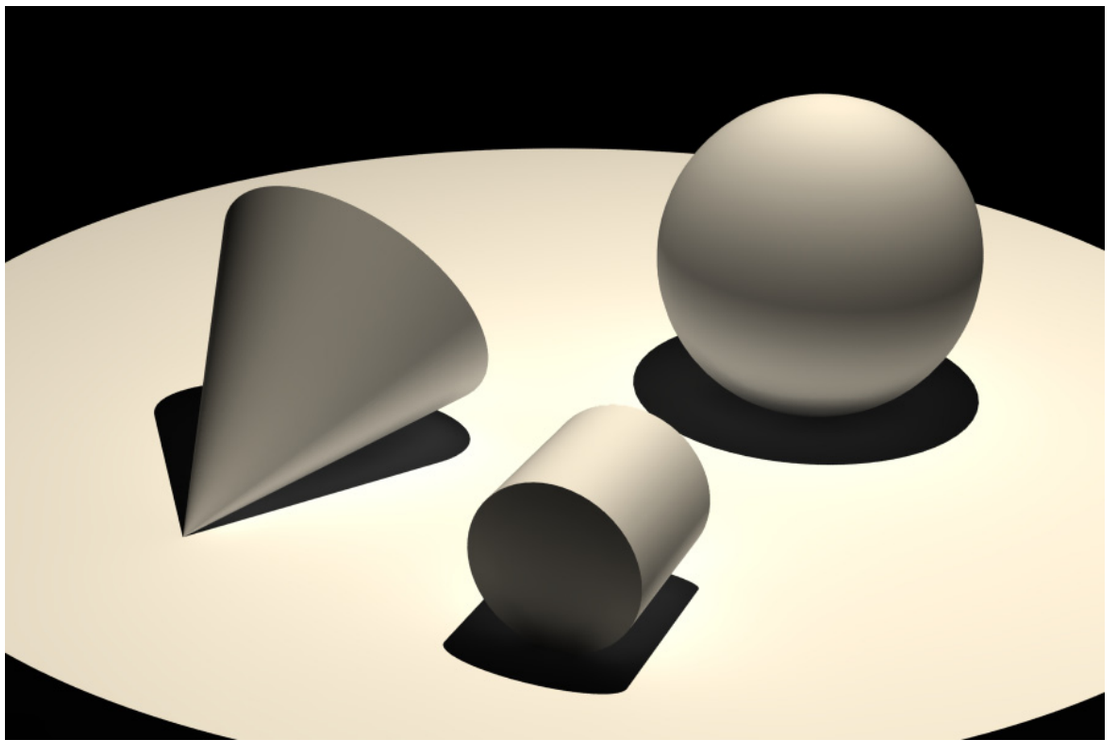
Global illumination as the ambient parameter

```
options "opt"
    object space
    contrast .1 .1 .1 1
    samples 0 2
    globillum on
    globillum accuracy 500 2
    shadow on
end options

shader "radiance"
    "average_radiance_options" (
        "color" 1 .95 .85,
        "globillum_accuracy" 1000 )

light "light"
    "spotlight" ()
    origin 1 5 0
    direction -.14 -1 -.1
    spread .912
    energy 600 600 600
    globillum photons 50000
end light

instance "light_inst" "light"
end instance

material "global_diffuse"
    "lambert" (
        "ambient" = "radiance",
        "diffuse" 1 .95 .85,
        "lights" ["light_inst"] )
    photon
        "store_diffuse_photon" (
            "diffuse_color" 1 .95 .85 )
end material
```

Lambert shading with global illumination shader for ambient value with option override

Light from other surfaces

Final gathering

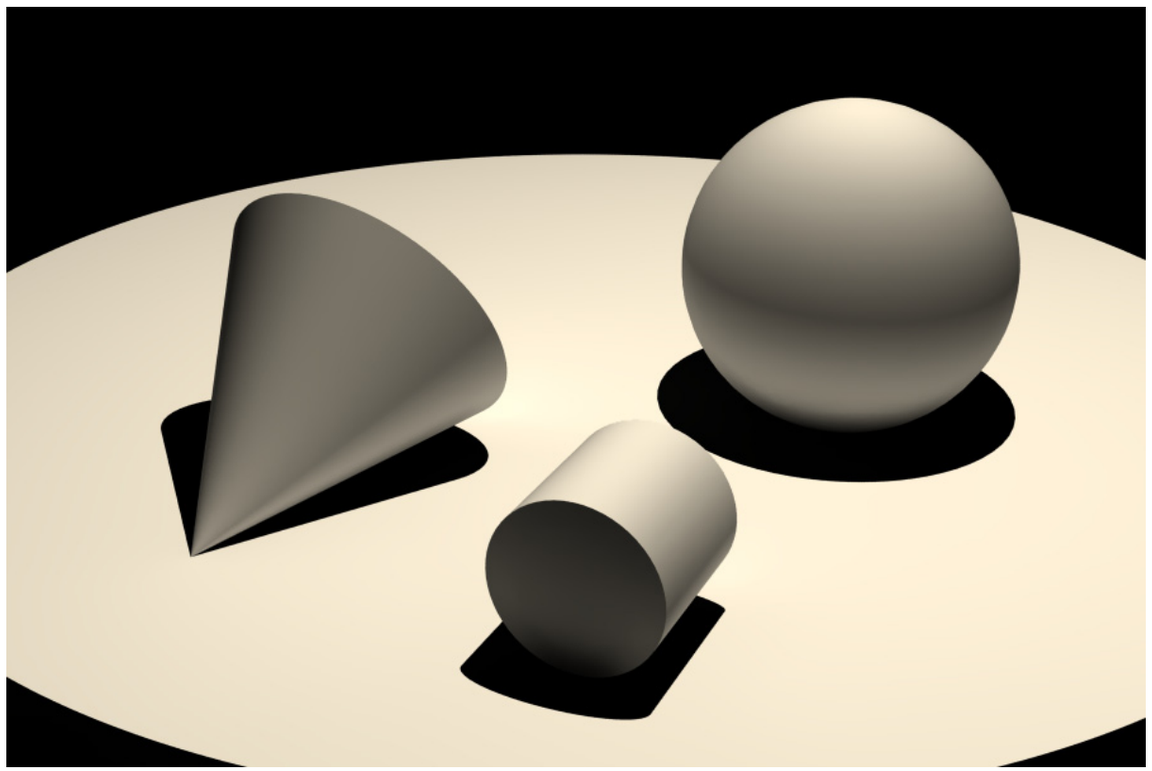


```
options "opt"
    object space
    contrast .1 .1 .1 1
    samples 0 2
    globillum on
    globillum accuracy 0 2
    shadow on
    finalgather on
    finalgather accuracy view 500 30 4
end options

shader "radiance"
    "average_radiance" ()

light "light"
    "spotlight" ()
    origin 1 5 0
    direction -.14 -1 -.1
    spread .912
    energy 600 600 600
    globillum photons 50000
end light

instance "light_inst" "light"
end instance

material "global_diffuse"
    "lambert" (
        "diffuse" 1 .95 .85,
        "ambient" = "radiance",
        "lights" ["light_inst"] )
    photon "store_diffuse_photon" ()
end material
```

Final gathering added to the result of the photon map

Light from other surfaces

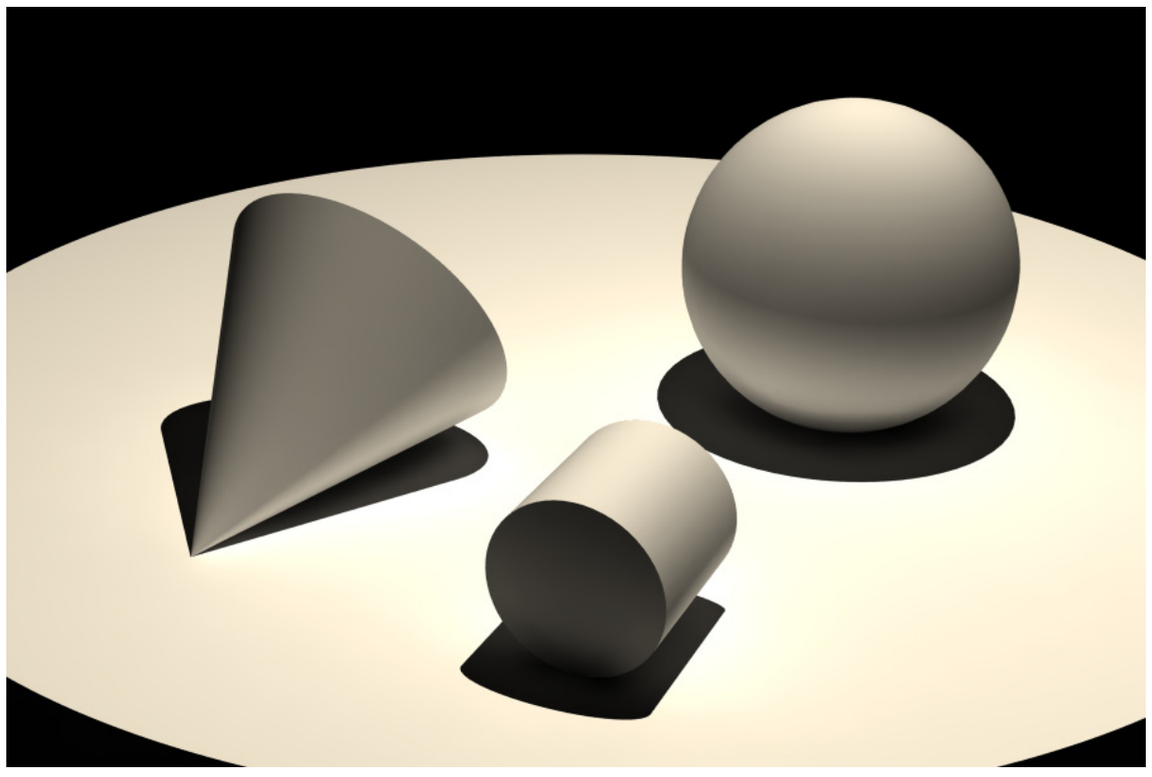
Final gathering



```
options "opt"
    object space
    contrast .1 .1 .1 1
    samples 0 2
    shadow on
    finalgather on
    finalgather accuracy view 500 30 4
end options

material "global_diffuse"
    "lambert" (
        "diffuse" 1 .95 .85,
        "ambient" = "radiance",
        "lights" ["light_inst"] )
    photon "store_diffuse_photon" ()
end material
```

Final gathering only

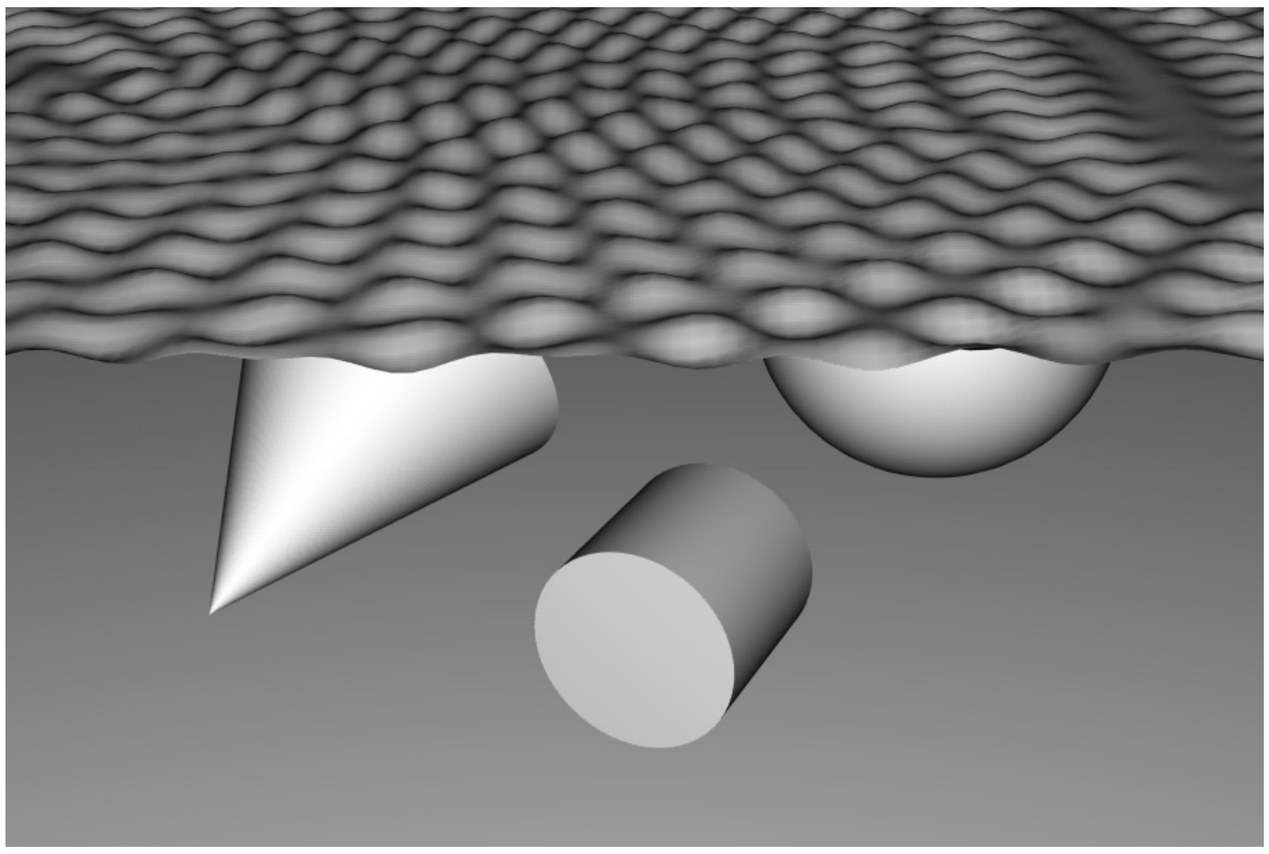
Light from other surfaces



```
options "opt"
    object space
    contrast .1 .1 .1 1
    samples 0 2
    shadow on
    finalgather on
    finalgather accuracy view 500 30 4
    finalgather trace depth 2 0 2 2
end options

material "global_diffuse"
    "lambert" (
        "diffuse" 1 .95 .85,
        "ambient" = "radiance",
        "lights" ["light_inst"] )
    photon "store_diffuse_photon" ()
end material
```

Final gathering only; two diffuse bounces to improve accuracy

Light from other surfaces



```
material "displace"
    "front_bright"  ()
    displace
        "displace_ripple" (
            "center" .2 .5 0,
            "frequency" 20,
            "amplitude" .01 )
        "displace_ripple" (
            "center" .8 .8 0,
            "frequency" 20,
            "amplitude" .01 )
        "displace_ripple" (
            "center" .8 .2 0,
            "frequency" 20,
            "amplitude" .01 )
end material
```

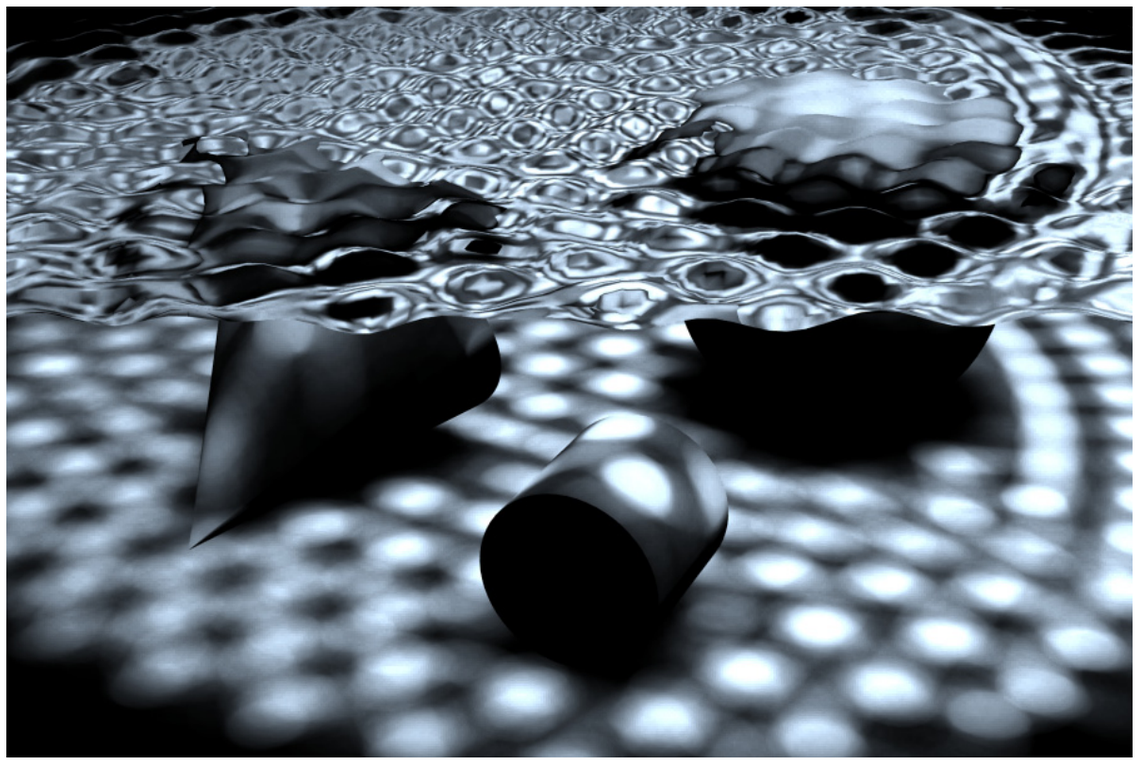
A square with displacements added to the scene

Light from other surfaces

```
declare shader
    color "transmit_specular_photon" (
        color "transparency" default 1 1 1,
        scalar "index_of_refraction" default 1.33 )
end declare
```

Scene file declaration of shader "transmit_specular_photon"

Light from other surfaces



```
options "opt"
    object space
    contrast .1 .1 .1 1
    samples -1 2
    shadow on
    displace on
    max displace .2
    caustic on
    caustic accuracy 500 .1
end options

material "refract"
    "specular_refraction" (
        "index_of_refraction" 1.5 )
    displace
        "displace_ripple" (
            "center" .2 .5 0,
            "frequency" 20,
            "amplitude" .01 )
        "displace_ripple" (
            "center" .8 .8 0,
            "frequency" 20,
            "amplitude" .01 )
        "displace_ripple" (
            "center" .8 .2 0,
            "frequency" 20,
            "amplitude" .01 )
    photon
        "transmit_specular_photon" (
            "index_of_refraction" 1.5 )
end material
```

Transmission of photons through a displaced surface to create caustics on a diffuse surface

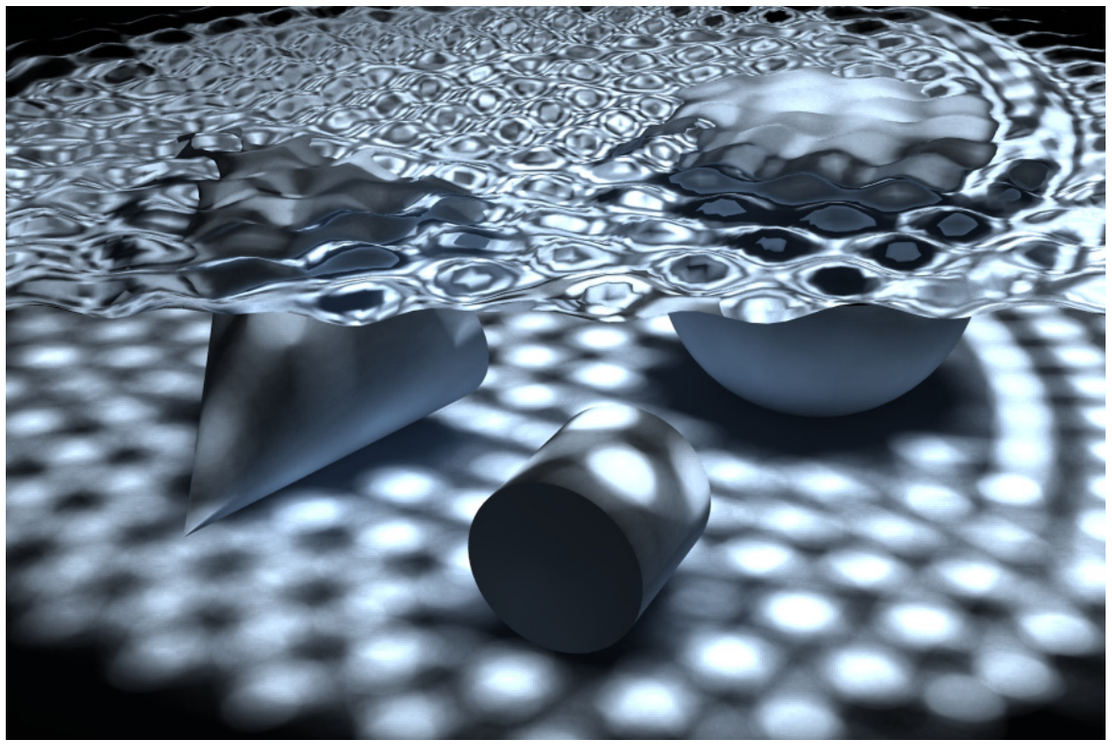
Light from other surfaces


```
 1  struct transmit_specular_photon {
 2      miColor transparency;
 3      miScalar index_of_refraction;
 4  };
 5
 6  miBoolean transmit_specular_photon (
 7      miColor *result, miState *state, struct transmit_specular_photon *params )
 8  {
 9      miVector photon_direction;
10      miColor new_energy;
11
12      miaux_multiply_colors(&new_energy, result,
13                            mi_eval_color(&params->transparency));
14      mi_refraction_dir(&photon_direction, state, 1.0,
15                        *mi_eval_scalar(&params->index_of_refraction));
16      mi_photon_transmission_specular(&new_energy, state, &photon_direction);
17
18      return miTRUE;
19  }
```

Source code of shader "transmit_specular_photon"

Light from other surfaces

Caustics

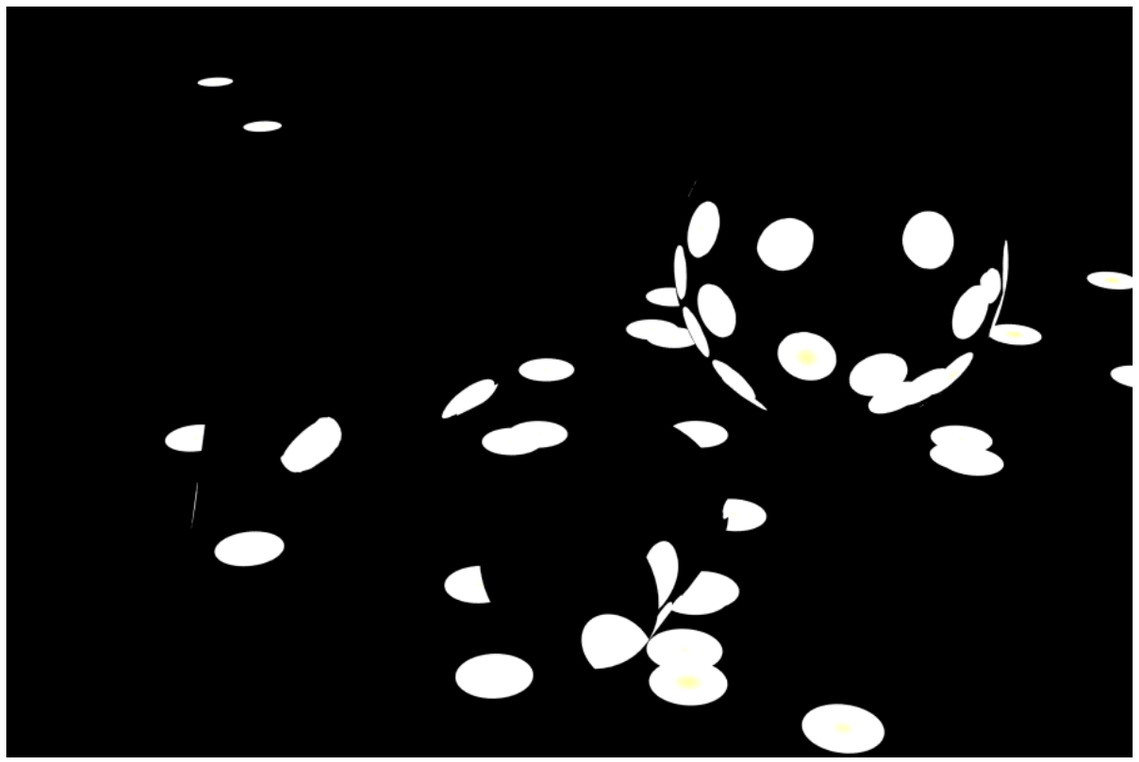


```
options "opt"
    object space
    contrast .1 .1 .1 1
    samples -1 2
    shadow on
    displace on
    max displace .2
    globillum on
    globillum accuracy 500 2
    caustic on
    caustic accuracy 500 .1
end options

material "refract"
    "specular_refraction" (
        "index_of_refraction" 1.5 )
    displace
        "displace_ripple" (
            "center" .2 .5 0,
            "frequency" 20,
            "amplitude" .01 )
        "displace_ripple" (
            "center" .8 .8 0,
            "frequency" 20,
            "amplitude" .01 )
        "displace_ripple" (
            "center" .8 .2 0,
            "frequency" 20,
            "amplitude" .01 )
    photon
        "transmit_specular_photon" (
            "index_of_refraction" 1.5 )
end material
```

Caustics and global illumination used together

Light from other surfaces

Visualizing the photon map

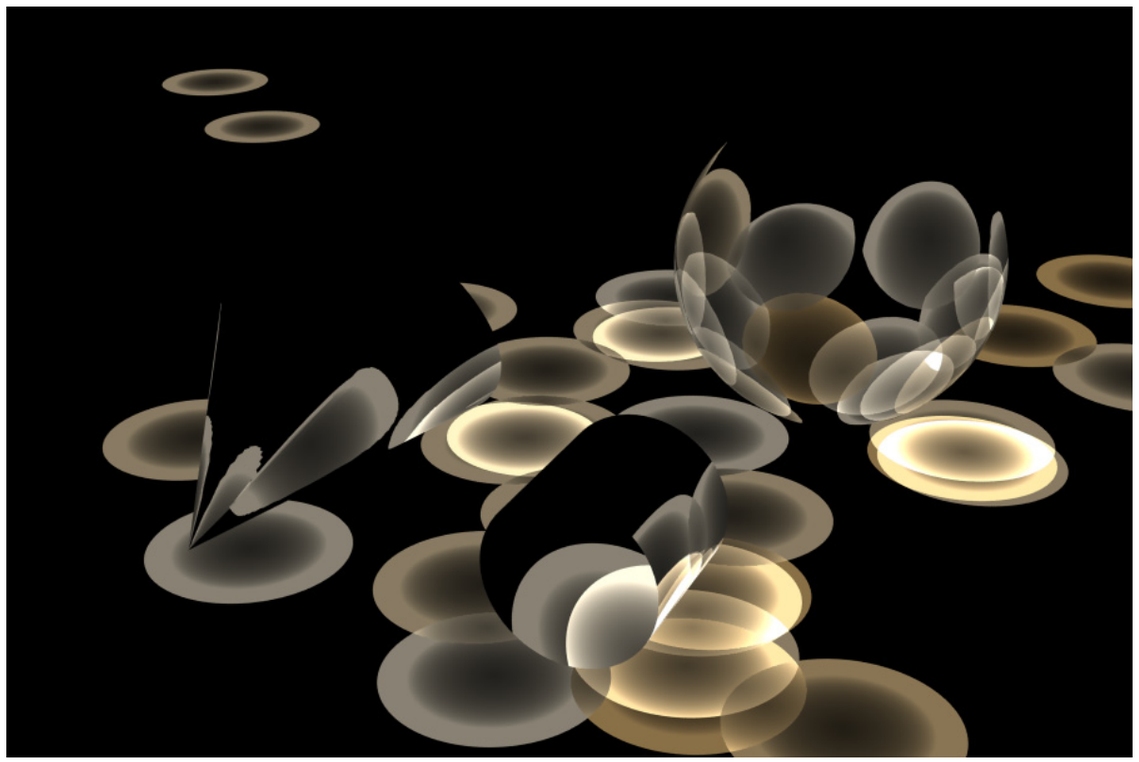


```
light "light"
    "spotlight" ()
    origin 1 5 0
    direction -.14 -1 -.1
    spread .912
    energy 1500 1500 1500
    globillum photons 100
end light

material "indirect"
    "average_radiance_options" (
        "globillum_accuracy" 100,
        "globillum_radius" .1 )
    photon
        "store_diffuse_photon" (
            "diffuse_color" 1 .95 .85 )
end material
```

Global illumination only: 100 photons with a radius of .1

Light from other surfaces

Visualizing the photon map

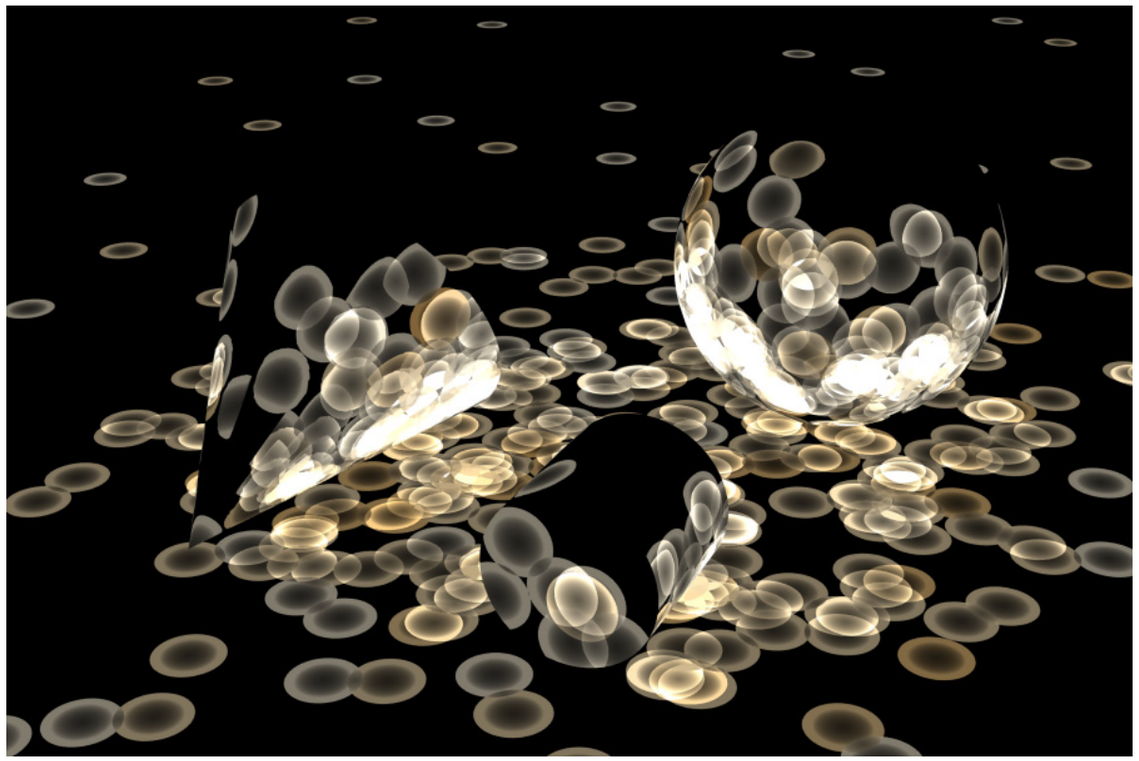


```
light "light"
    "spotlight" ()
    origin 1 5 0
    direction -.14 -1 -.1
    spread .912
    energy 1500 1500 1500
    globillum photons 100
end light

material "indirect"
    "average_radiance_options" (
        "globillum_accuracy" 100,
        "globillum_radius" .3 )
    photon
        "store_diffuse_photon" (
            "diffuse_color" 1 .95 .85 )
end material
```

Global illumination only: 100 photons with a radius of .3

Light from other surfaces

Visualizing the photon map

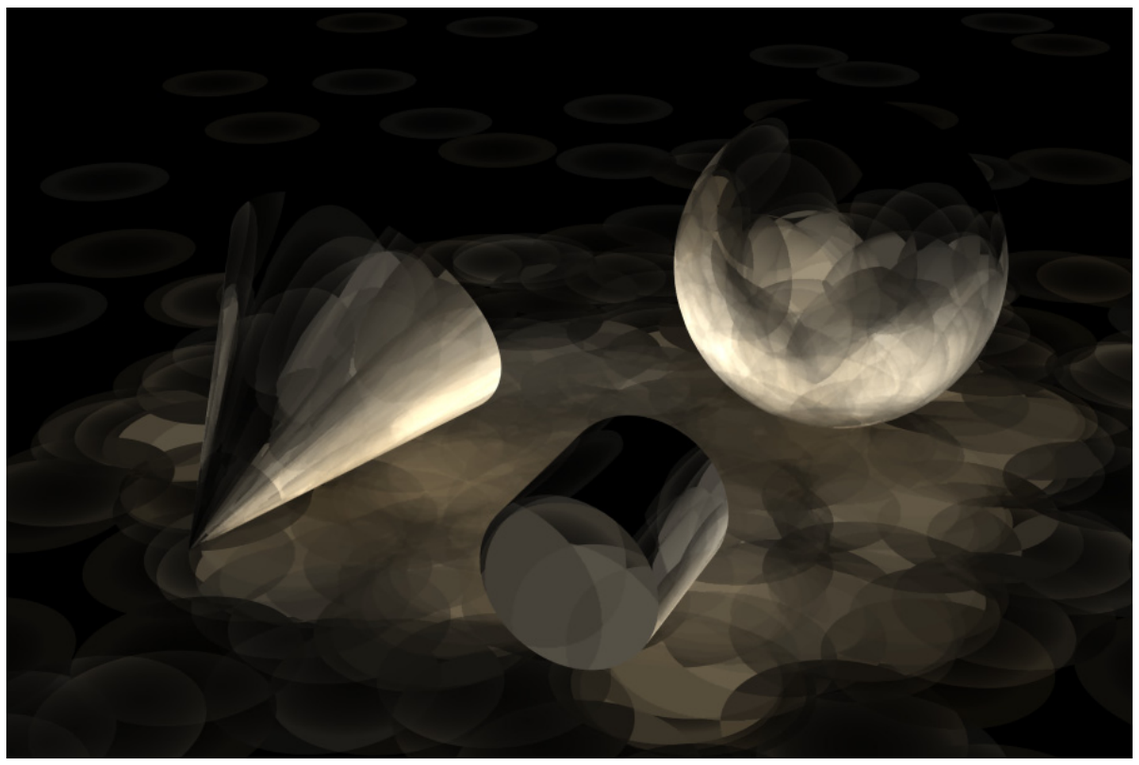


```
light "light"
    "spotlight" ()
    origin 1 5 0
    direction -.14 -1 -.1
    spread .912
    energy 1500 1500 1500
    globillum photons 1000
end light

material "indirect"
    "average_radiance_options" (
        "globillum_accuracy" 1000,
        "globillum_radius" .1 )
    photon
        "store_diffuse_photon" (
            "diffuse_color" 1 .95 .85 )
end material
```

Global illumination only: 1000 photons with a radius of .1

Light from other surfaces

Visualizing the photon map

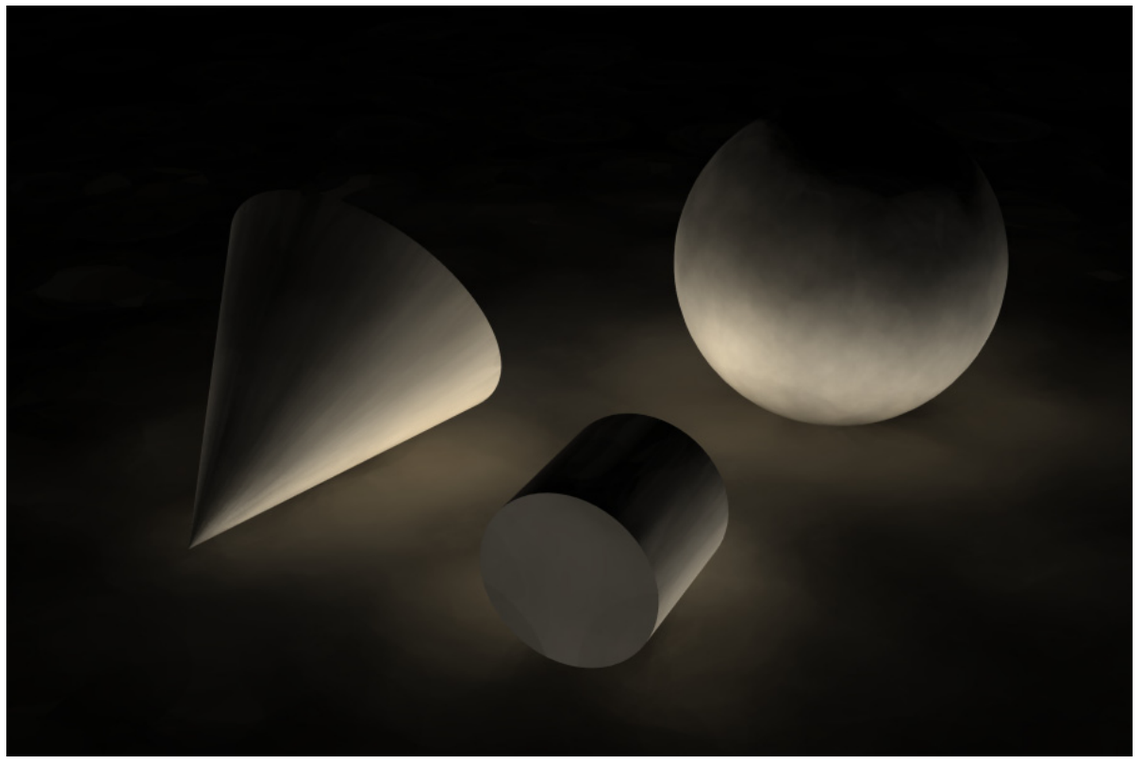


```
light "light"
    "spotlight" ()
    origin 1 5 0
    direction -.14 -1 -.1
    spread .912
    energy 1500 1500 1500
    globillum photons 1000
end light

material "indirect"
    "average_radiance_options" (
        "globillum_accuracy" 1000,
        "globillum_radius" .3 )
    photon
        "store_diffuse_photon" (
            "diffuse_color" 1 .95 .85 )
end material
```

Global illumination only: 1000 photons with a radius of .3

Light from other surfaces

Visualizing the photon map



```
light "light"
    "spotlight" ()
    origin 1 5 0
    direction -.14 -1 -.1
    spread .912
    energy 1500 1500 1500
    globillum photons 10000
end light

material "indirect"
    "average_radiance_options" (
        "globillum_accuracy" 10000,
        "globillum_radius" .3 )
    photon
        "store_diffuse_photon" (
            "diffuse_color" 1 .95 .85 )
end material
```

Global illumination only: 10000 photons with a radius of .3

The four conditions required for global illumination

The four conditions required for global illumination

1. Set global illumination options.

The four conditions required for global illumination

1. Set global illumination options.
2. Emit photons from the light.

The four conditions required for global illumination

1. Set global illumination options.
2. Emit photons from the light.
3. Include a photon shader in materials.

The four conditions required for global illumination

1. Set global illumination options.
2. Emit photons from the light.
3. Include a photon shader in materials.
4. Use a material shader that includes global illumination.

Exercise 14: Use photon mapping

1. Copy `globillum_2.mi` to `globillum.mi`, change output to `globillum.tif`
2. Render and view.
3. Change to use 100 photons and a smaller photon radius (from 2.0 to 0.1).
4. Increase to 1,000 photons and render.
5. Increase to 0.3 radius and render.
6. Increase to 10,000, then 50,000 photons.
7. In Maya, 3ds Max or XSI, open global illumination exercises scene file.
8. Note light emission switch in light shape node.
9. Repeat the above changes to the photon count and radius.

Exercise 15: Using indirect illumination as the ambient component

1. Render `phenomena_C.mi`. This scene defines a *material Phenomenon* that contains a photon shader.
2. Find where shader `average_radiance` defines the ambient component of `lambert`.
3. Turn off the direct light contribution by setting the `diffuse` parameter in the `lambert` shader to black. Re-render.
4. Visualize the location of photons by changing their size and number:

   Change the radius for photon averaging:

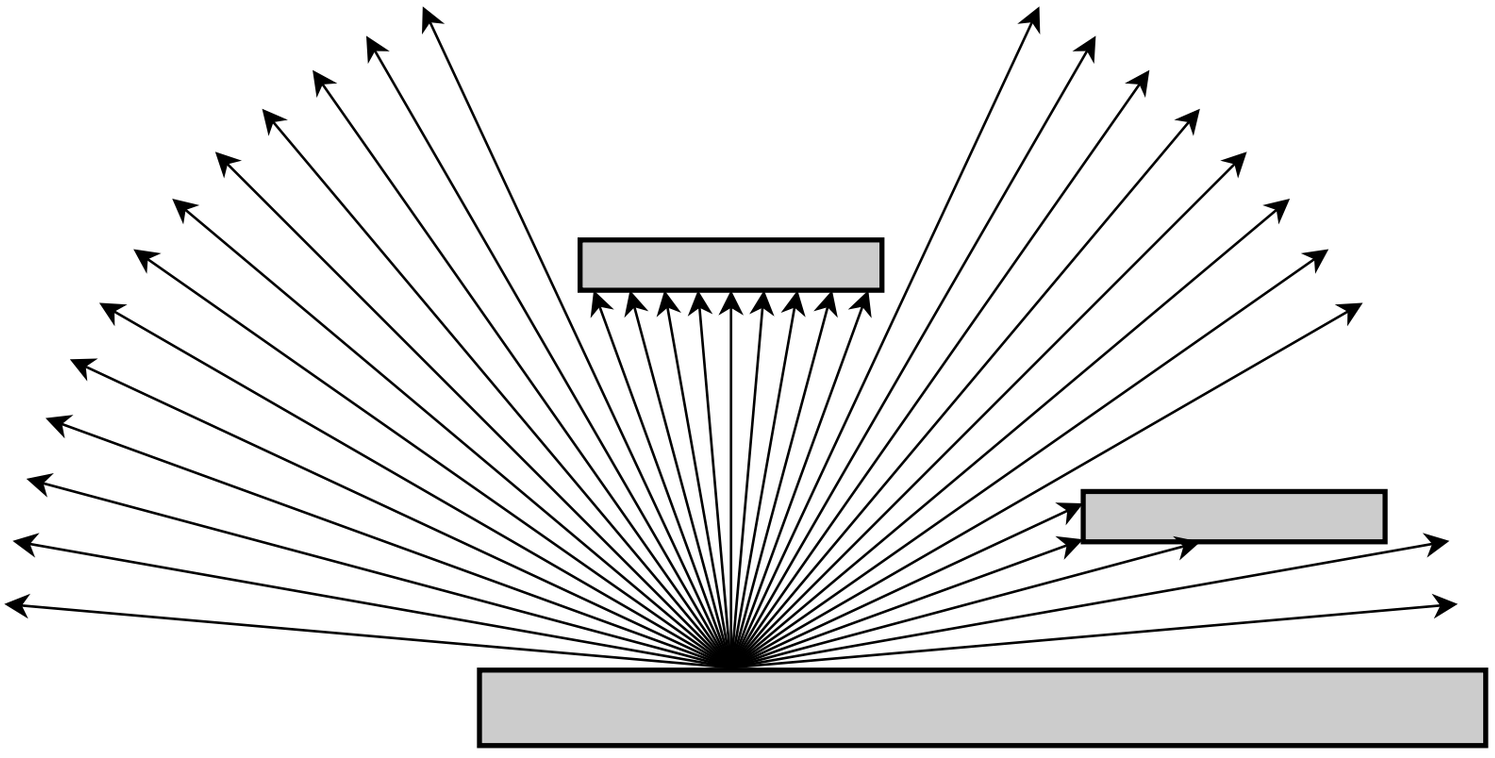
   ```
   globillum accuracy 500 .1
   ```

   Change the number of photons emitted from the light:

   ```
   globillum photons 300
   ```

Light from other surfaces

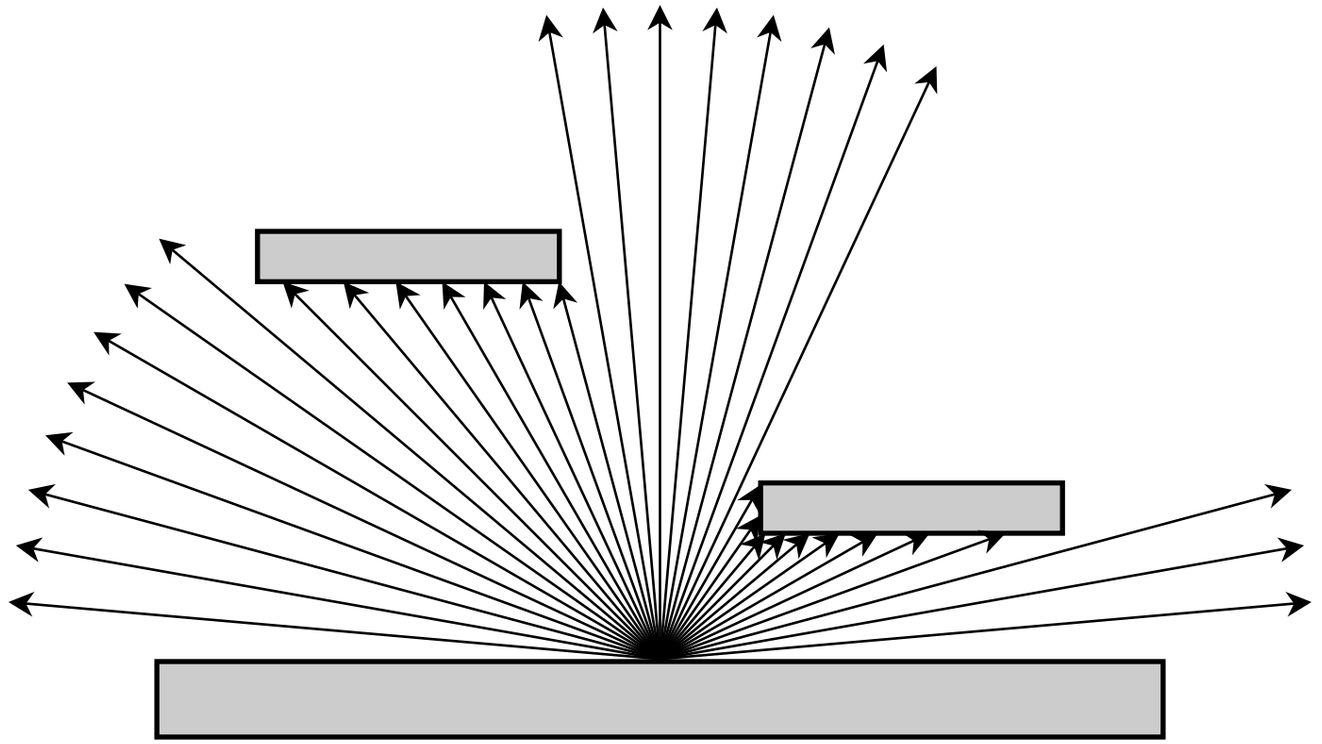
Ambient occlusion



Calculating ambient occlusion: 12 intersections out of 35, or 34% occlusion

Light from other surfaces

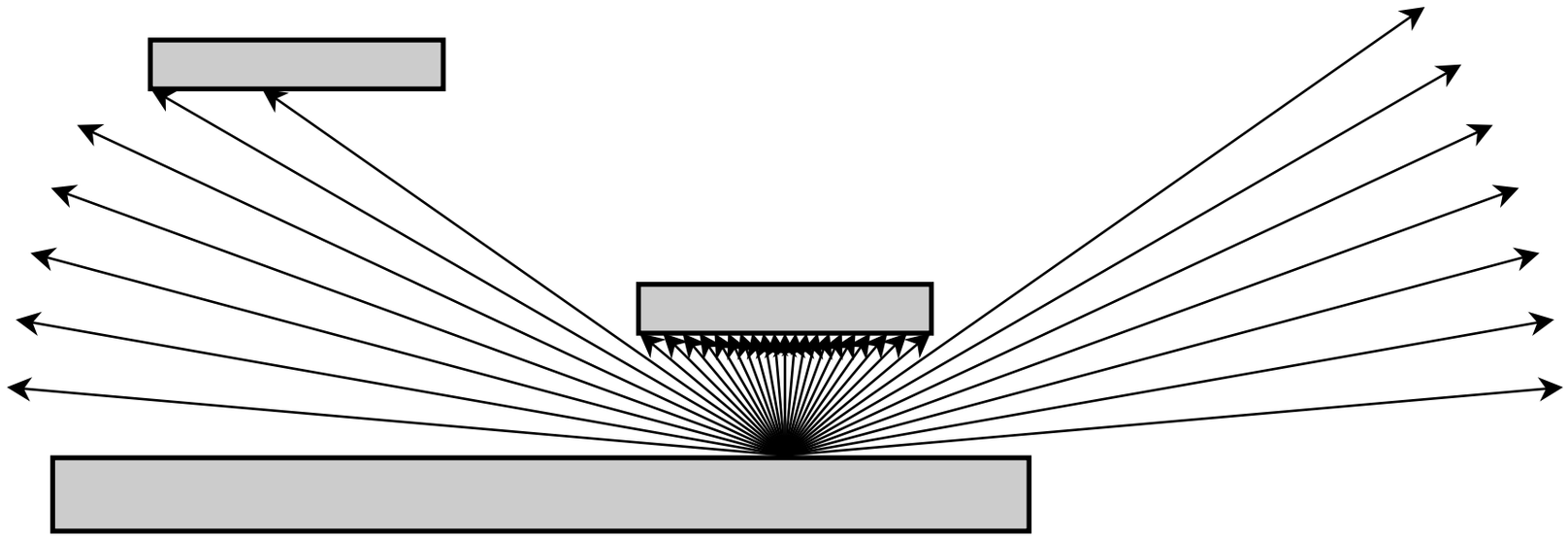
Ambient occlusion



Calculating ambient occlusion: 16 intersections out of 35, or 46% occlusion

Light from other surfaces

Ambient occlusion



Calculating ambient occlusion: 23 intersections out of 35, or 66% occlusion

Light from other surfaces

Ambient occlusion

```
declare shader
    color "ambient_occlusion" (
        integer "samples" default 8 )
end declare
```

Scene file declaration of shader "ambient_occlusion"

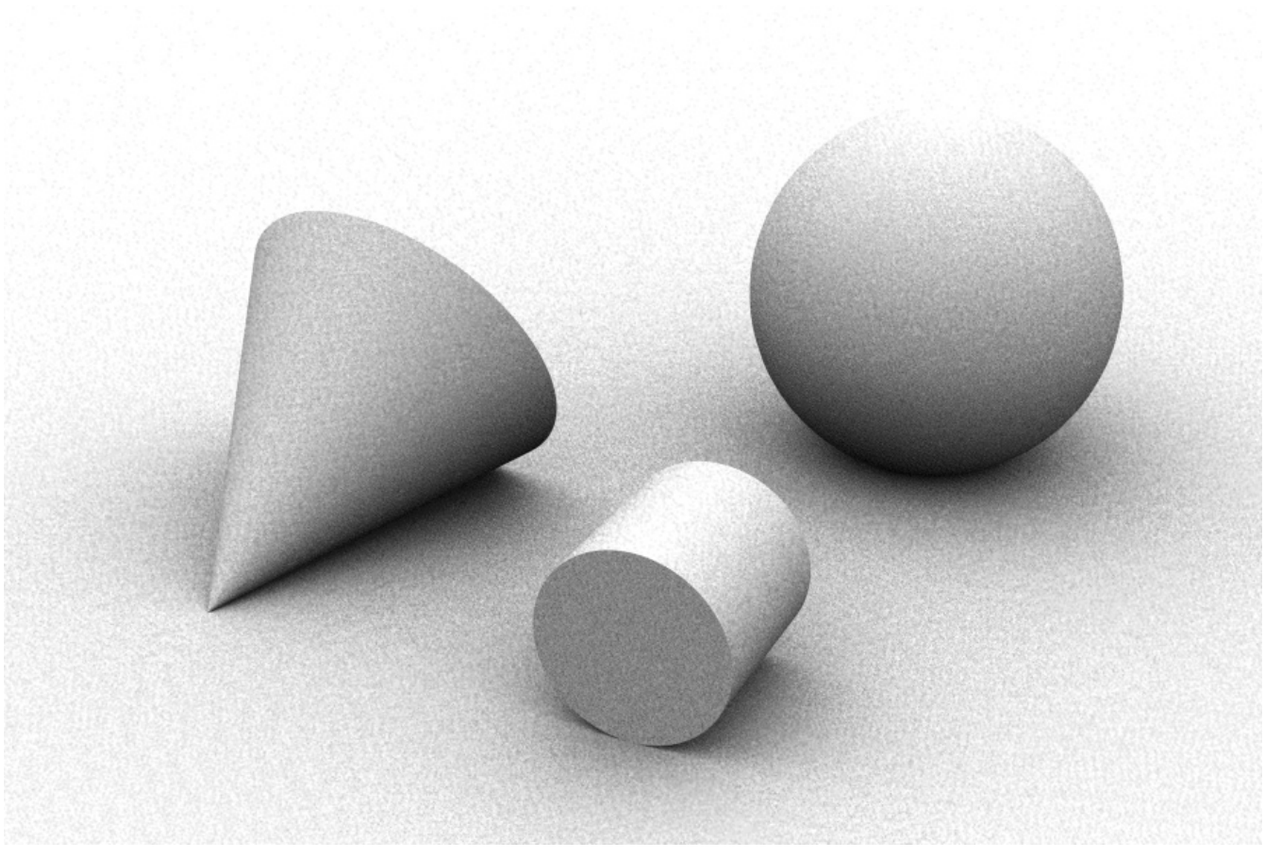
Light from other surfaces


```
 1  struct ambient_occlusion {
 2      miUint samples;
 3  };
 4
 5  miBoolean ambient_occlusion(
 6      miColor *result, miState *state, struct ambient_occlusion *params)
 7  {
 8      miUint samples  = *mi_eval_integer(&params->samples);
 9      miVector trace_direction;
10      int object_hit = 0, sample_number = 0;
11      double sample[2], hit_fraction, ambient_exposure;
12
13      while (mi_sample(sample, &sample_number, state, 2, &samples)) {
14          mi_reflection_dir_diffuse_x(&trace_direction, state, sample);
15          if (mi_trace_probe(state, &trace_direction, &state->point))
16              object_hit++;
17      }
18      hit_fraction = ((double)object_hit / (double)samples);
19      ambient_exposure = 1.0 - hit_fraction;
20      result->r = result->g = result->b = ambient_exposure;
21      result->a = 1.0;
22
23      return miTRUE;
24  }
```

Source code of shader "ambient_occlusion"

Light from other surfaces

Ambient occlusion

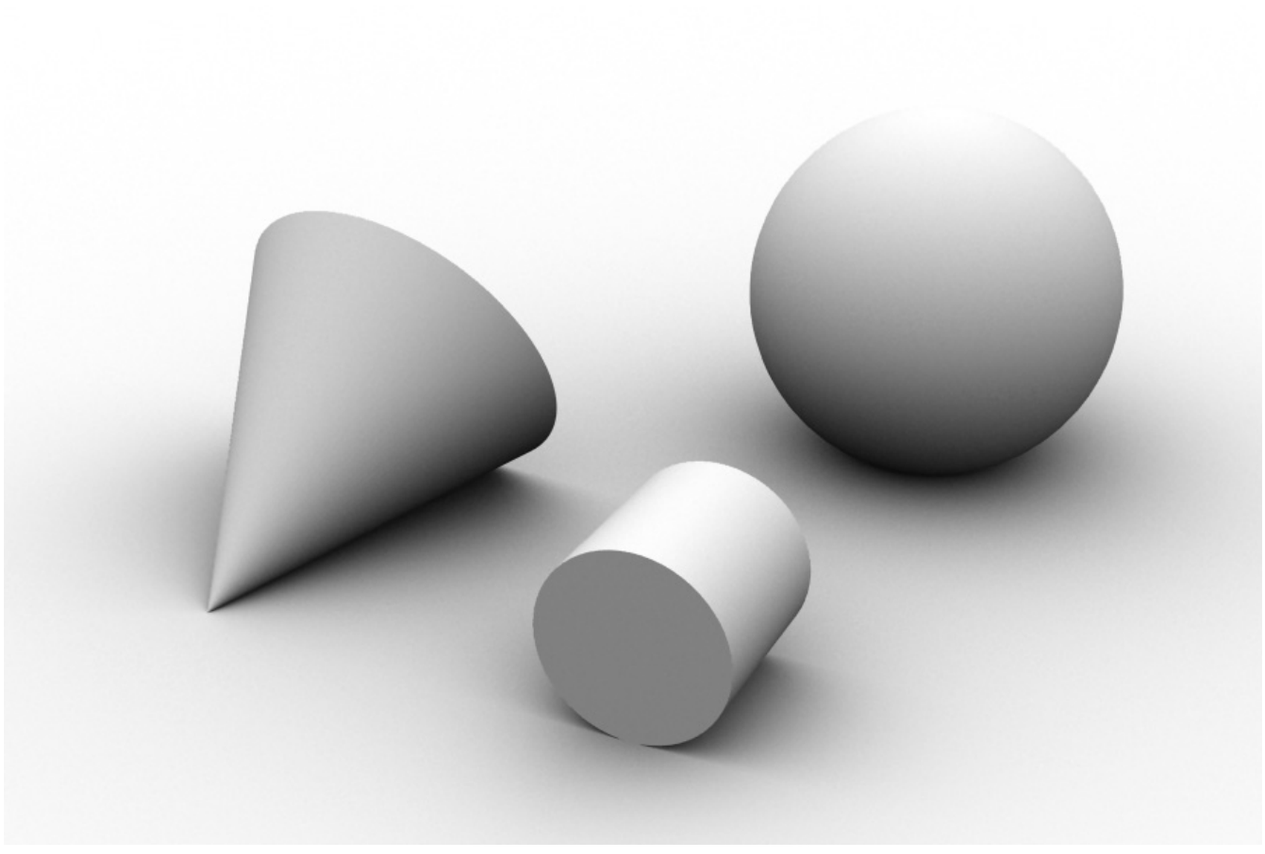


```
material "amb_occlude"
    "ambient_occlusion" (
        "samples" 10 )
end material
```

Ambient occlusion using ten samples to determine the degree of occlusion

Light from other surfaces

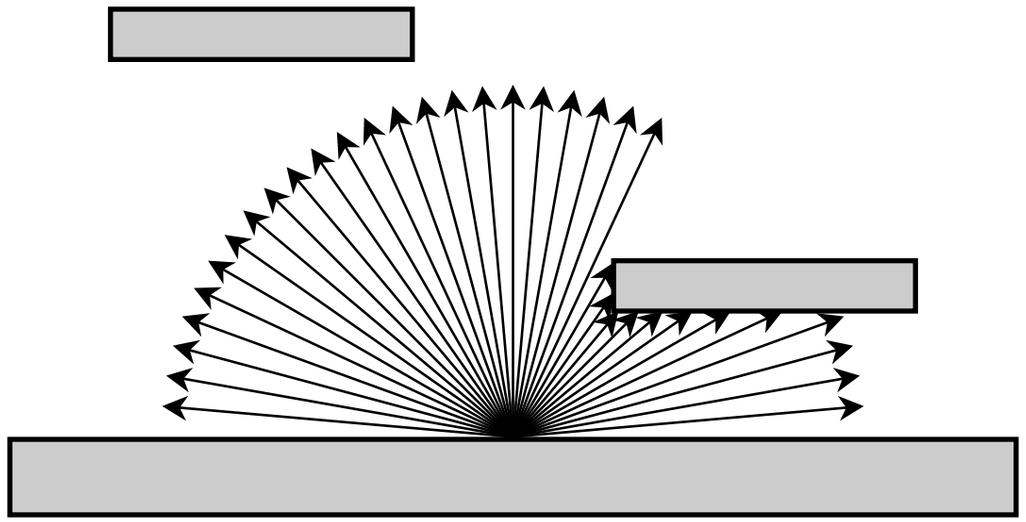
Ambient occlusion



```
material "amb_occlude"
    "ambient_occlusion" (
        "samples" 200 )
end material
```

Using more ambient occlusion samples to improve image quality

Light from other surfaces



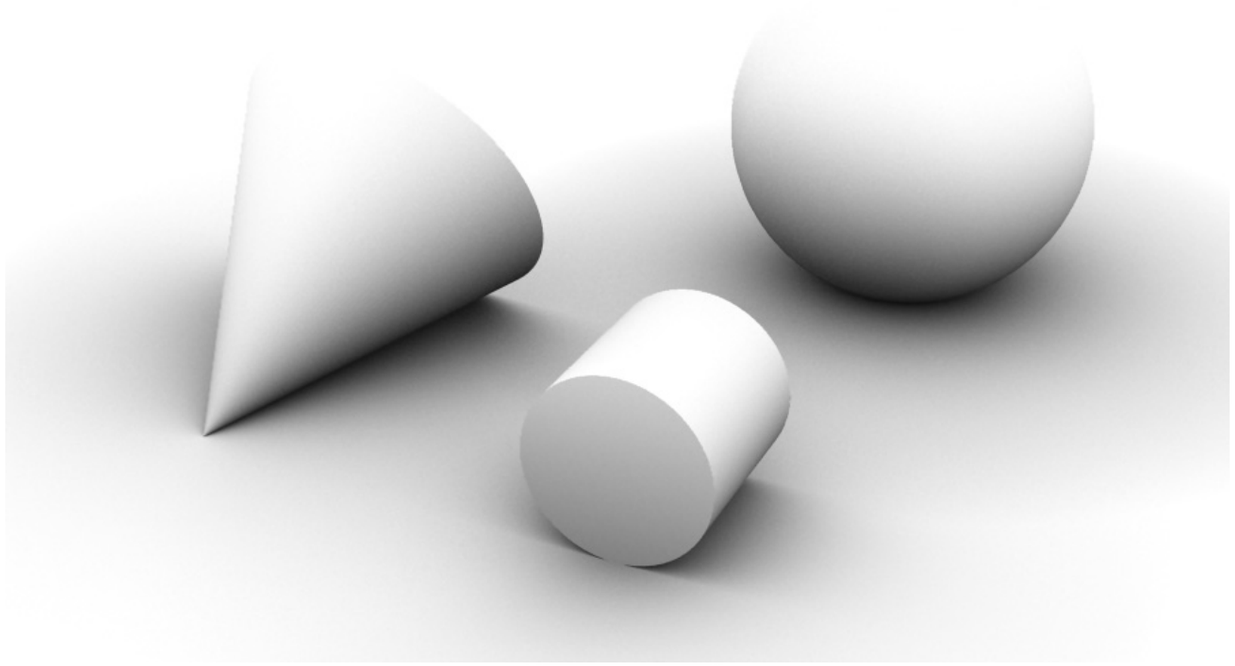
Calculating ambient occlusion using a cutoff: 8 intersections out of 35, or 23% occlusion

```
declare shader
    color "ambient_occlusion_cutoff" (
        integer "samples" default 8,
        scalar  "cutoff_distance" default 1, )
end declare
```

Scene file declaration of shader "ambient_occlusion_cutoff"

Light from other surfaces

Ambient occlusion



```
material "amb_occlude"
    "ambient_occlusion_cutoff" (
        "samples" 200,
        "cutoff_distance" 1 )
end material
```

Ambient occlusion calculated with sample rays of restricted length

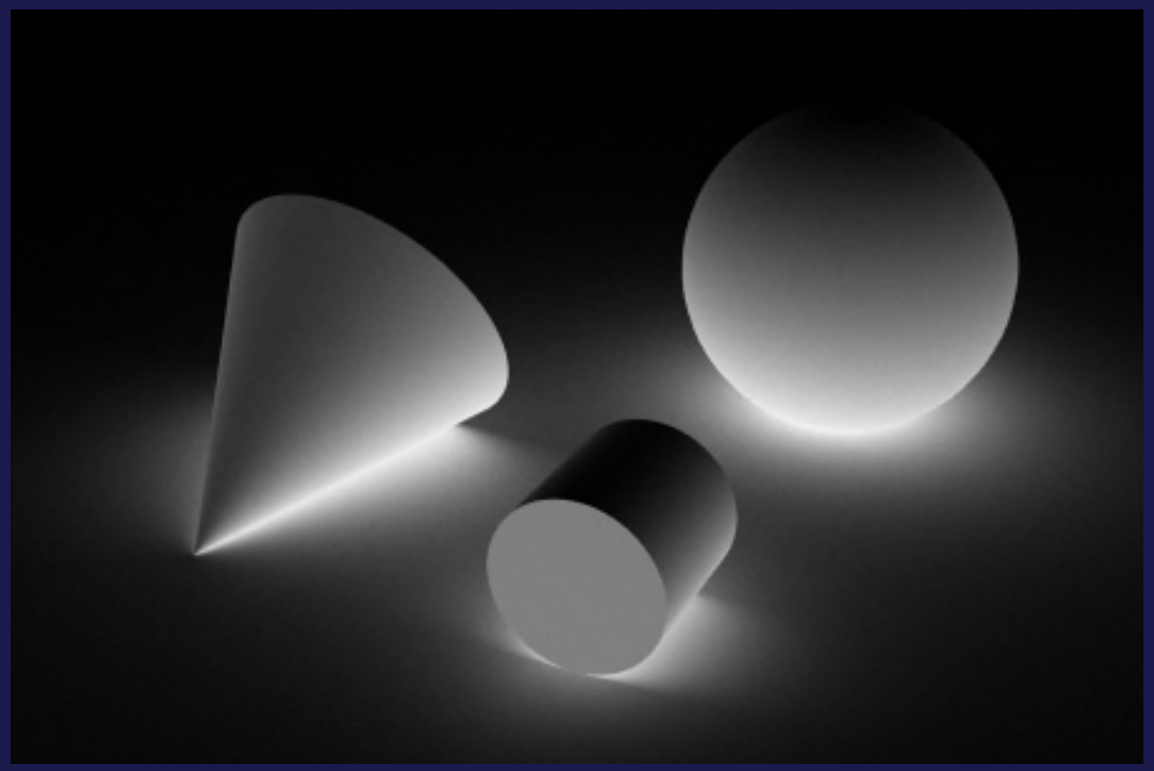
Light from other surfaces


```
 1  struct ambient_occlusion_cutoff {
 2      miUint samples;
 3      miScalar cutoff_distance;
 4  };
 5
 6  miBoolean ambient_occlusion_cutoff(
 7      miColor *result, miState *state, struct ambient_occlusion_cutoff *params)
 8  {
 9      miUint samples  = *mi_eval_integer(&params->samples);
10      miScalar cutoff_distance = *mi_eval_scalar(&params->cutoff_distance);
11      miVector trace_direction;
12      int object_hit = 0, sample_number = 0;
13      double sample[2], hit_fraction, ambient_exposure,
14          falloff_start, falloff_stop;
15
16      falloff_start = falloff_stop = cutoff_distance;
17      mi_ray_falloff(state, &falloff_start, &falloff_stop);
18
19      while (mi_sample(sample, &sample_number, state, 2, &samples)) {
20          mi_reflection_dir_diffuse_x(&trace_direction, state, sample);
21          if (mi_trace_probe(state, &trace_direction, &state->point))
22              object_hit++;
23      }
24      hit_fraction = ((double)object_hit / (double)samples);
25      ambient_exposure = 1.0 - hit_fraction;
26      result->r = result->g = result->b = ambient_exposure;
27      result->a = 1.0;
28
29      mi_ray_falloff(state, &falloff_start, &falloff_stop);
30      return miTRUE;
31  }
```

Source code of shader "ambient_occlusion_cutoff"

Exercise 16: Ambient occlusion

1. Copy `ambocclude_1.mi` to `ambocclude.mi`, change output filename to `ambocclude.tif` and render.
2. Change `samples` parameter and re-render.
3. Change shader `ambient_occlusion` so that white means that the point is occluded, and black means that it is not.



Exercise 16: Ambient occlusion (part 2)

Change shader `ambient_occlusion` so that white means that the point is occluded, and black means that it is not.

Old:

```
hit_fraction = ((double)object_hit / (double)samples);
ambient_exposure = 1.0 - hit_fraction;
result->r = result->g = result->b = ambient_exposure;
```

New:

```
hit_fraction = ((double)object_hit / (double)samples);
result->r = result->g = result->b = hit_fraction;
```

or:

```
result->r = result->g = result->b =
    ((double)object_hit / (double)samples);
```

Exercise 17: Final gathering and ambient occlusion

1. Render `phenomena_D.mi`.
2. A Phenomenon can use other Phenomena in its shader graph.
   Examine how `rescale` is used in `fg_with_ao`.
3. You can see intermediate results in the Phenomena graph by setting the root to other shaders. In `fg_with_ao`, change `root = "darken"` to `root = "ao"` and re-render.
4. Ambient occlusion is used in `fg_with_ao` to darken corners. Adjust the `cutoff_distance` and `new_min` to modify the ambient occlusion value. (Change `samples` to `1` when experimenting.)
5. The blue cast in the shadows comes from the enviroment shader attached to the camera. Change it to a reddish hue.