Writing mental ray shaders

by Andy Kopra

Part 5: Space

The environment of the scene

The environment of the scene

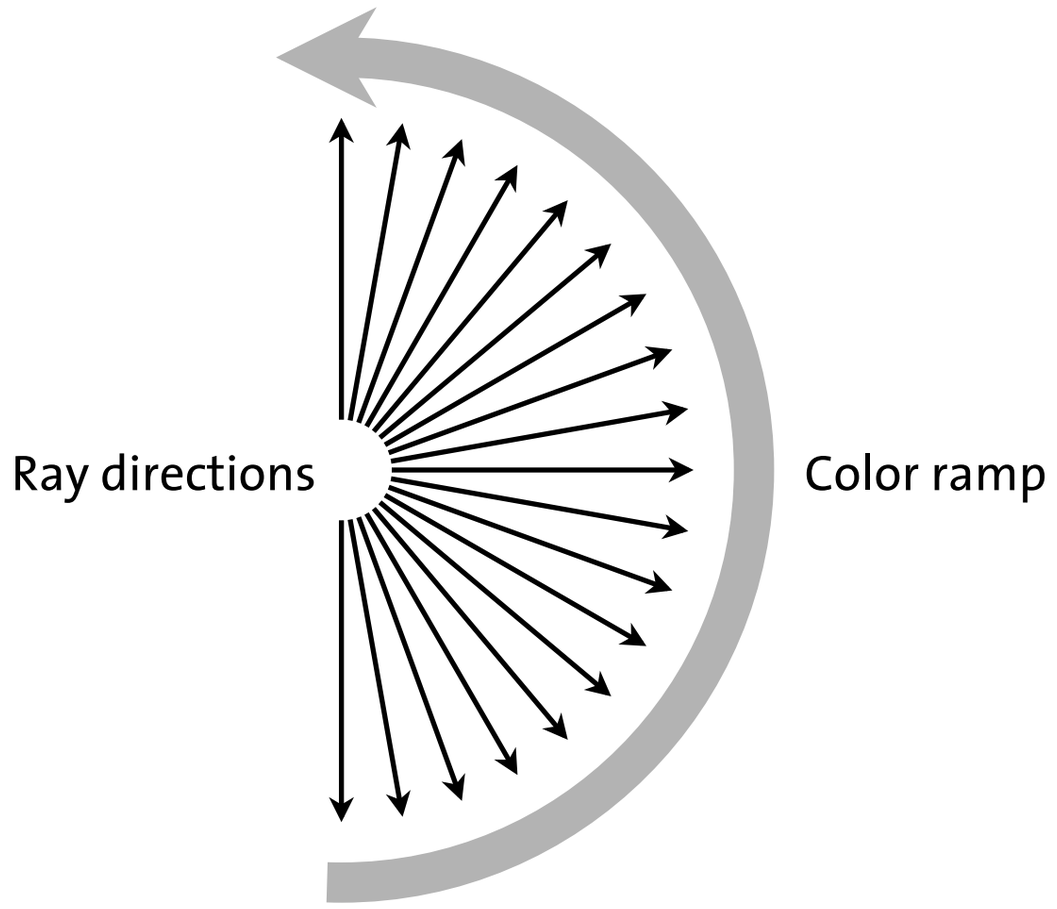
- A single color for the environment
- Defining a color ramp
- Efficient shader access to a color ramp
- Using parameter arrays for color channels in the ramp
    - Allocating memory in the init function
    - Releasing the memory allocated through the user pointer
    - Using a channel ramp in a named shader
- A color array as a shader parameter
- Environment shaders for cameras and objects
- Encapsulating constant data in a Phenomenon

The environment of the scene

A single color for the environment

```
camera "cam"
    output "rgba" "tif" "environment_1.tif"
    focal 1.5
    aperture 1.5
    aspect 1
    resolution 300 300
    environment
        "one_color" (
            "color" 1 .95 .9 )
end camera
```

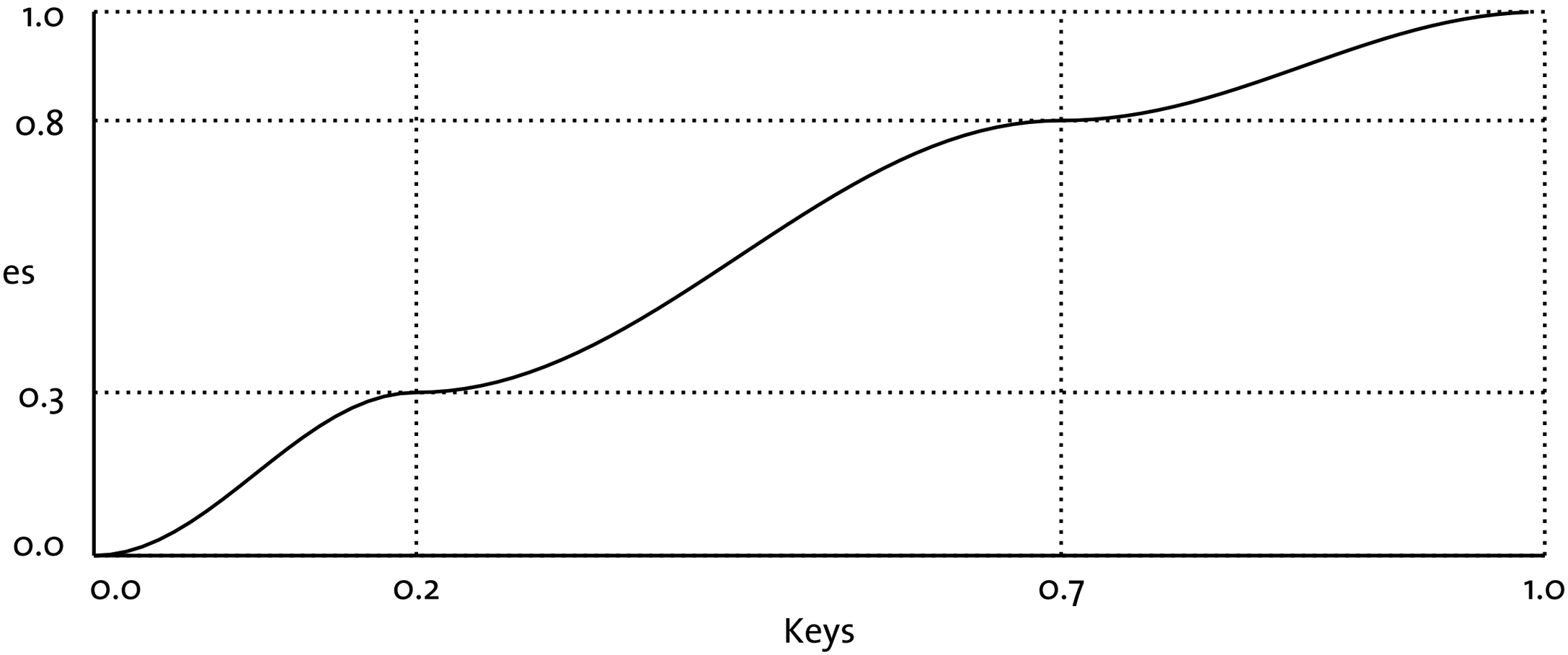
Shader one_color attached to the camera as an environment shader



Mapping altitude to positions in a ramp of colors

```
1  float miaux_altitude(miState *state)
2  {
3      miVector ray;
4      mi_vector_to_world(state, &ray, &state->dir);
5      mi_vector_normalize(&ray);
6      return miaux_fit(asin(ray.y), -M_PI_2, M_PI_2, 0.0, 1.0);
7  }
```

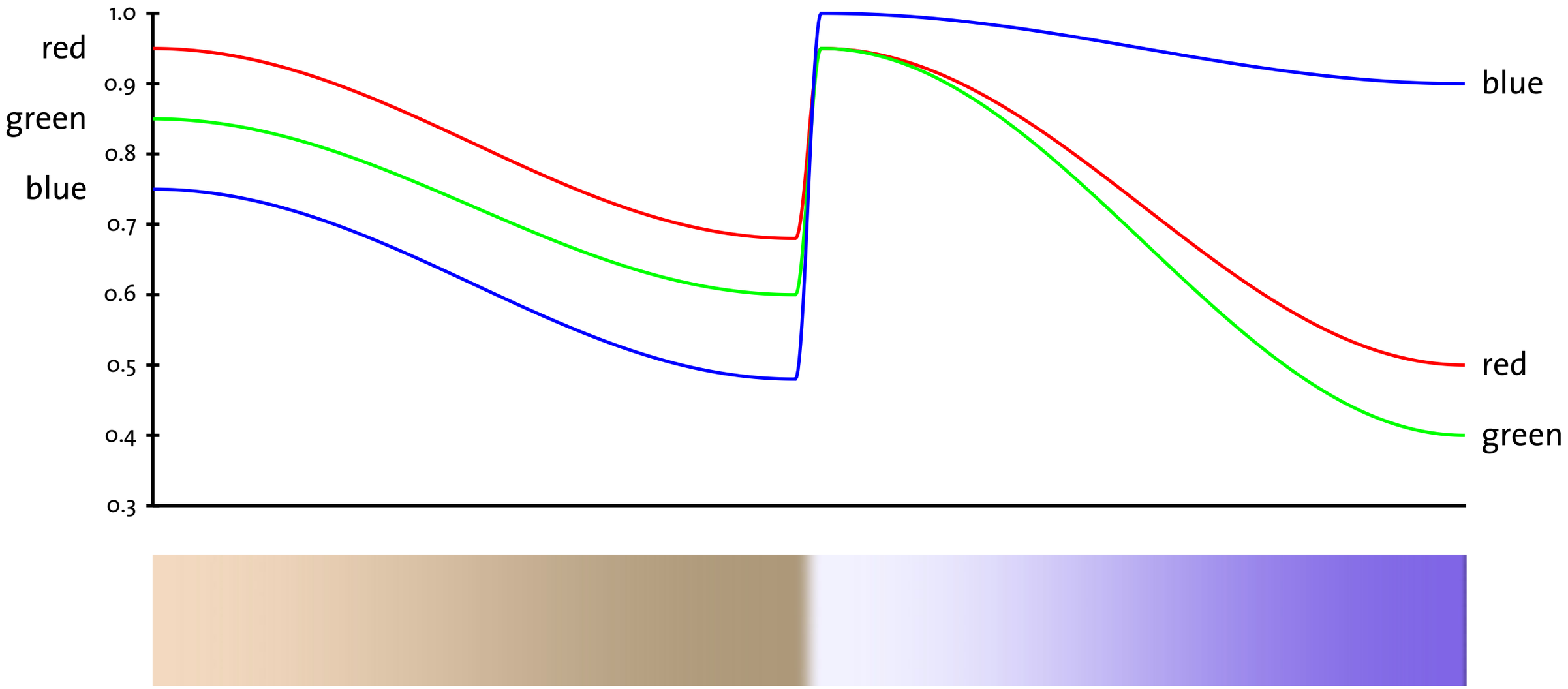
Auxiliary function: miaux_altitude



Constructing a smooth curve from sinusoidal pieces

The environment of the scene

Defining a color ramp



Channel curves from sinusoidal pieces

```
 1  void miaux_piecewise_sinusoid(
 2      miScalar result[], int result_count,
 3      int key_count, miScalar key_positions[], miScalar key_values[])
 4  {
 5      int key, i;
 6      for (key = 1; key < key_count; key++) {
 7          int start = (int)(key_positions[key-1] * result_count);
 8          int end = (int)(key_positions[key] * result_count) - 1;
 9          for (i = start; i <= end; i++) {
10              result[i] = miaux_sinusoid_fit(
11                  i, start, end, key_values[key-1], key_values[key]);
12          }
13      }
14  }
```

Auxiliary function: miaux_piecewise_sinusoid

```
 1  miScalar miaux_interpolated_lookup(miScalar lookup_table[], int table_size,
 2                                     miScalar t)
 3  {
 4      int lower_index = (int)(t * (table_size - 1));
 5      miScalar lower_value = lookup_table[lower_index];
 6      int upper_index = lower_index + 1;
 7      miScalar upper_value = lookup_table[upper_index];
 8      return miaux_fit(
 9          t * table_size, lower_index, upper_index, lower_value, upper_value);
10  }
```

Auxiliary function: miaux_interpolated_lookup

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

Scene file declaration of shader "chrome_ramp"

```
1  #define RAMPSIZE 1024
2
3  typedef struct {
4      int r_size, g_size, b_size;
5      miScalar r[RAMPSIZE];
6      miScalar g[RAMPSIZE];
7      miScalar b[RAMPSIZE];
8  } channel_ramp_table;
9
10 static channel_ramp_table *ramp;
```

Source code of struct "channel_ramp_table"

The environment of the scene

Efficient shader access to a color ramp

```
 1  miBoolean chrome_ramp_init(
 2      miState *state, void *params, miBoolean *instance_init_required)
 3  {
 4      int key_count = 4;
 5
 6      miScalar key_positions[] = {0,    0.49, 0.51, 1.0};
 7      miScalar red[]           = {0.95, 0.68, 0.95, 0.5};
 8      miScalar green[]         = {0.85, 0.6,  0.95, 0.4};
 9      miScalar blue[]          = {0.75, 0.48, 1.0,  0.9};
10
11      ramp = (channel_ramp_table*)mi_mem_allocate(sizeof(channel_ramp_table));
12
13      miaux_piecewise_sinusoid(ramp->r, RAMPSIZE, key_count, key_positions, red);
14      miaux_piecewise_sinusoid(ramp->g, RAMPSIZE, key_count, key_positions, green);
15      miaux_piecewise_sinusoid(ramp->b, RAMPSIZE, key_count, key_positions, blue);
16
17      return miTRUE;
18  }
```

Source code of init shader of "chrome_ramp"

The environment of the scene

Efficient shader access to a color ramp

```
1  miBoolean chrome_ramp (
2      miColor *result, miState *state, void *params )
3  {
4      miScalar altitude = miaux_altitude(state);
5      result->r = miaux_interpolated_lookup(ramp->r, RAMPSIZE, altitude);
6      result->g = miaux_interpolated_lookup(ramp->g, RAMPSIZE, altitude);
7      result->b = miaux_interpolated_lookup(ramp->b, RAMPSIZE, altitude);
8      return miTRUE;
9  }
```

Source code of main shader of "chrome_ramp"

```
1  miBoolean chrome_ramp_exit(miState *state, void *params)
2  {
3       mi_mem_release(ramp);
4       return miTRUE;
5  }
```

Source code of exit shader of "chrome_ramp"

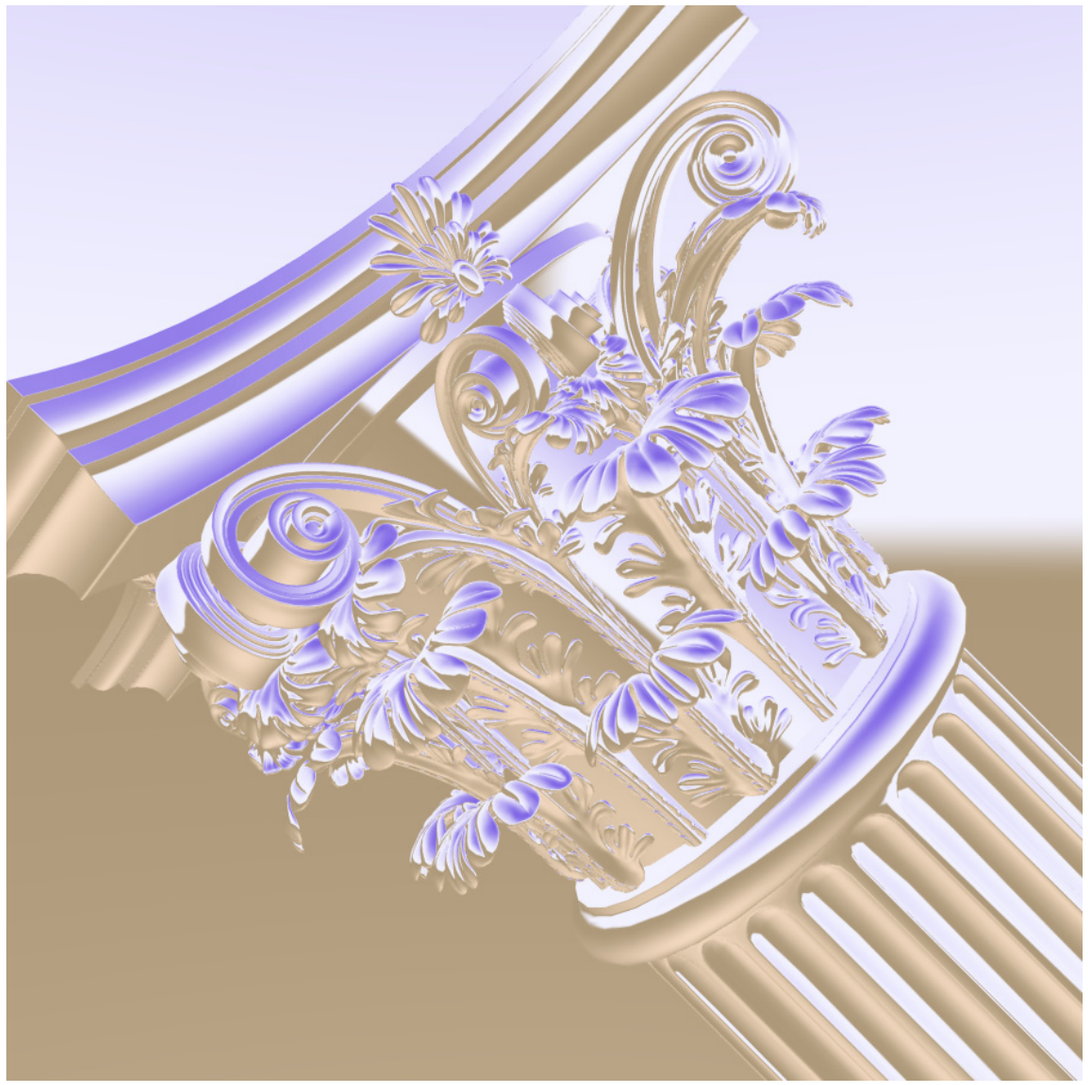
The environment of the scene

Efficient shader access to a color ramp

```
camera "cam"
    output "rgba" "tif" "environment_2.tif"
    focal 1.5
    aperture 1.5
    aspect 1
    resolution 300 300
    environment
        "chrome_ramp" ()
end camera

material "corinthian_material"
    "lambert" (
        "diffuse" .95 .85 .75,
        "lights" ["light-inst"] )
end material
```

Shader chrome_ramp attached to the camera as an environment shader.

The environment of the scene



Efficient shader access to a color ramp

```
camera "cam"
    output "rgba" "tif" "environment_3.tif"
    focal 1.5
    aperture 1.5
    aspect 1
    resolution 300 300
    environment
        "chrome_ramp" ()
end camera

material "corinthian_material"
    "specular_reflection" ()
end material
```

Shader chrome_ramp used for rays reflected with shader specular_reflection.

```
declare shader
    color "channel_ramp" (
        array scalar "keys",
        array scalar "r",
        array scalar "g",
        array scalar "b" )
end declare
```

Scene file declaration of shader "channel_ramp"

```
miBoolean shader_init(
    miState *state, struct shader *params,
    miBoolean *instance_init_required)
{
    if (params == NULL) {
        /* Do main shader initialization */
        *instance_init_required = miTRUE;
    } else {
        /* Do instance-based shader initialization */
    }
    return miTRUE;
}
```

Template for init shader function

```
1  void* miaux_user_memory_pointer(miState *state, int allocation_size)
2  {
3      void **user_pointer;
4      mi_query(miQ_FUNC_USERPTR, state, 0, &user_pointer);
5      if (allocation_size > 0) {
6          *user_pointer = mi_mem_allocate(allocation_size);
7      }
8      return *user_pointer;
9  }
```

Auxiliary function: miaux_user_memory_pointer

The environment of the scene

Using parameter arrays for color channels in the ramp

```
 1  miBoolean channel_ramp_init(
 2      miState *state, struct channel_ramp *params, miBoolean *instance_init_required)
 3  {
 4      if (params == NULL) {  /* Main shader init (not an instance) */
 5          *instance_init_required = miTRUE;
 6      } else {               /* Instance initialization */
 7          int n_keys, n_r, n_g, n_b;
 8          miScalar *keys, *red, *green, *blue;
 9          channel_ramp_table *ramp;
10
11          n_keys = *mi_eval_integer(&params->n_keys);
12          keys = mi_eval_scalar(params->keys) +
13              *mi_eval_integer(&params->i_keys);
14
15          if ((n_r = *mi_eval_integer(&params->n_r)) != n_keys)
16              mi_fatal("Incorrect number of red values: %d", n_r);
17          red = mi_eval_scalar(params->r) + *mi_eval_integer(&params->i_r);
18
19          if ((n_g = *mi_eval_integer(&params->n_g)) != n_keys)
20              mi_fatal("Incorrect number of green values: %d", n_g);
21          green = mi_eval_scalar(params->g) + *mi_eval_integer(&params->i_g);
22
23          if ((n_b = *mi_eval_integer(&params->n_b)) != n_keys)
24              mi_fatal("Incorrect number of blue values: %d", n_b);
25          blue = mi_eval_scalar(params->b) + *mi_eval_integer(&params->i_b);
26
27          ramp = miaux_user_memory_pointer(state, sizeof(channel_ramp_table));
28
29          miaux_piecewise_sinusoid(ramp->r, RAMPSIZE, n_keys, keys, red);
30          miaux_piecewise_sinusoid(ramp->g, RAMPSIZE, n_keys, keys, green);
31          miaux_piecewise_sinusoid(ramp->b, RAMPSIZE, n_keys, keys, blue);
32      }
33      return miTRUE;
34  }
```

Source code of init shader of "channel_ramp"

The environment of the scene

Using parameter arrays for color channels in the ramp

```
1  miBoolean channel_ramp (
2      miColor *result, miState *state, struct channel_ramp *params )
3  {
4      miScalar altitude = miaux_altitude(state);
5      channel_ramp_table *ramp = miaux_user_memory_pointer(state, 0);
6
7      result->r = miaux_interpolated_lookup(ramp->r, RAMPSIZE, altitude);
8      result->g = miaux_interpolated_lookup(ramp->g, RAMPSIZE, altitude);
9      result->b = miaux_interpolated_lookup(ramp->b, RAMPSIZE, altitude);
10
11     return miTRUE;
12 }
```

Source code of main shader of "channel_ramp"

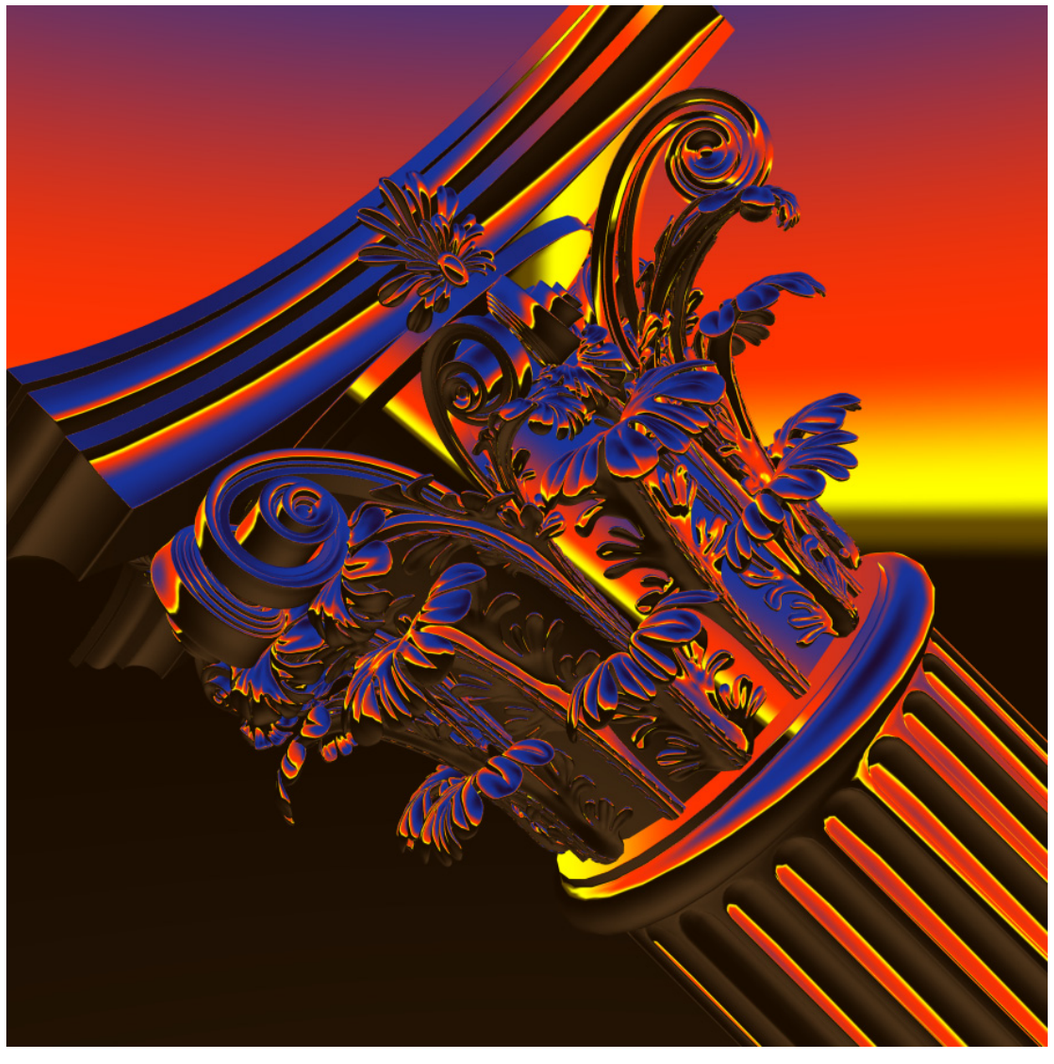
```
1  miBoolean miaux_release_user_memory(char* shader_name, miState *state, void *params)
2  {
3      if (params != NULL) {  /* Shader instance exit */
4          void **user_pointer;
5          if (!mi_query(miQ_FUNC_USERPTR, state, 0, &user_pointer))
6              mi_fatal("Could not get user pointer in shader exit function %s_exit",
7                       shader_name);
8          mi_mem_release(*user_pointer);
9      }
10     return miTRUE;
11 }
```

Auxiliary function: miaux_release_user_memory

```
1  miBoolean channel_ramp_exit(miState *state, void *params)
2  {
3      return miaux_release_user_memory("channel_ramp", state, params);
4  }
```

Source code of exit shader of "channel_ramp"

The environment of the scene



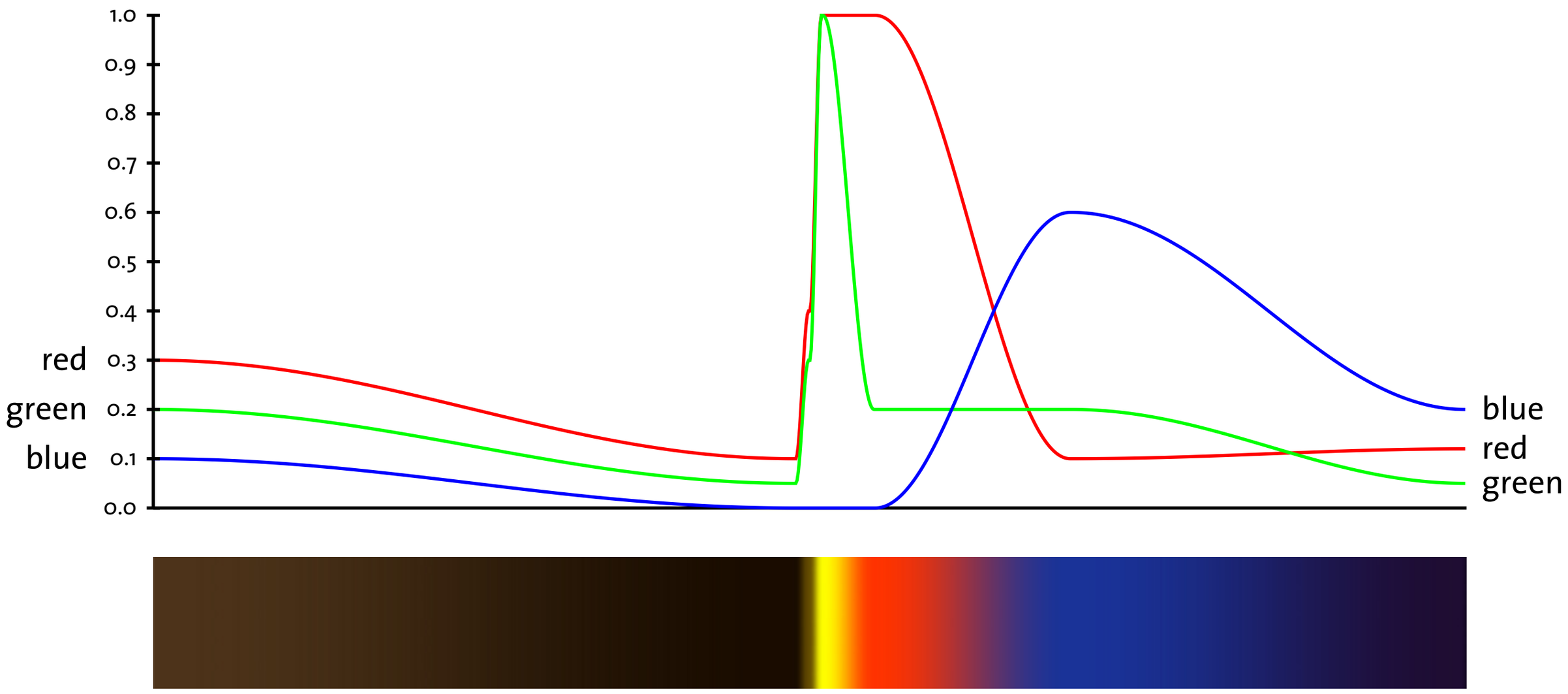
Using parameter arrays for color channels in the ramp

```
shader "sunset"
  "channel_ramp" (
    "keys" [0,  .49, .50, .51, .55, .70, 1.0],
    "r"    [.3, .1,  .4,   1,   1,  .1,  .12],
    "g"    [.2, .05, .3,   1,  .2,  .2,  .05],
    "b"    [.1,  0,   0,   0,   0,  .6,  .2] )

camera "cam"
    output "rgba" "tif" "environment_4.tif"
    focal 1.5
    aperture 1.5
    aspect 1
    resolution 300 300
    environment
        = "sunset"
end camera

material "corinthian_material"
    "specular_reflection" ()
end material
```

Color ramp values defined with arrays for key positions and channel values

The environment of the scene

Using parameter arrays for color channels in the ramp



Color curves specified in the scene file for shader channel_ramp

The environment of the scene

A color array as a shader parameter

```
declare shader
    color "color_ramp" (
        array color "colors" )
end declare
```

Scene file declaration of shader "color_ramp"

```
 1  void miaux_piecewise_color_sinusoid(
 2      miColor result[], int result_count, int key_count, miColor key_values[])
 3  {
 4      int key, i;
 5      for (key = 1; key < key_count; key++) {
 6          int start = (int)(key_values[key-1].a * result_count);
 7          int end = (int)(key_values[key].a * result_count) - 1;
 8          for (i = start; i <= end; i++) {
 9              result[i].r =
10                  miaux_sinusoid_fit(
11                      i, start, end, key_values[key-1].r, key_values[key].r);
12              result[i].g =
13                  miaux_sinusoid_fit(
14                      i, start, end, key_values[key-1].g, key_values[key].g);
15              result[i].b =
16                  miaux_sinusoid_fit(
17                      i, start, end, key_values[key-1].b, key_values[key].b);
18          }
19      }
20  }
```

Auxiliary function: miaux_piecewise_color_sinusoid

```
 1  void miaux_interpolated_color_lookup(miColor* result,
 2                                       miColor lookup_table[], int table_size,
 3                                       miScalar t)
 4  {
 5      int lower_index = (int)(t * table_size);
 6      miColor lower_value = lookup_table[lower_index];
 7      int upper_index = lower_index + 1;
 8      miColor upper_value = lookup_table[upper_index];
 9      result->r = miaux_fit(t * table_size, lower_index, upper_index,
10                            lower_value.r, upper_value.r);
11      result->g = miaux_fit(t * table_size, lower_index, upper_index,
12                            lower_value.g, upper_value.g);
13      result->b = miaux_fit(t * table_size, lower_index, upper_index,
14                            lower_value.b, upper_value.b);
15  }
```

Auxiliary function: miaux_interpolated_color_lookup

The environment of the scene

A color array as a shader parameter

```
 1  miBoolean color_ramp_init(
 2      miState *state, struct color_ramp *params,
 3      miBoolean *instance_init_required)
 4  {
 5      if (params == NULL) {  /* Main shader init (not an instance) */
 6          *instance_init_required = miTRUE;
 7      } else {               /* Instance initialization */
 8          int n_colors = *mi_eval_integer(&params->n_colors);
 9          miColor *colors =
10              mi_eval_color(params->colors) + *mi_eval_integer(&params->i_colors);
11          miColor *ramp =
12              miaux_user_memory_pointer(state, sizeof(miColor) * RAMPSIZE);
13
14          miaux_piecewise_color_sinusoid(ramp, RAMPSIZE, n_colors, colors);
15      }
16      return miTRUE;
17  }
```

Source code of init shader of "color_ramp"

The environment of the scene

A color array as a shader parameter

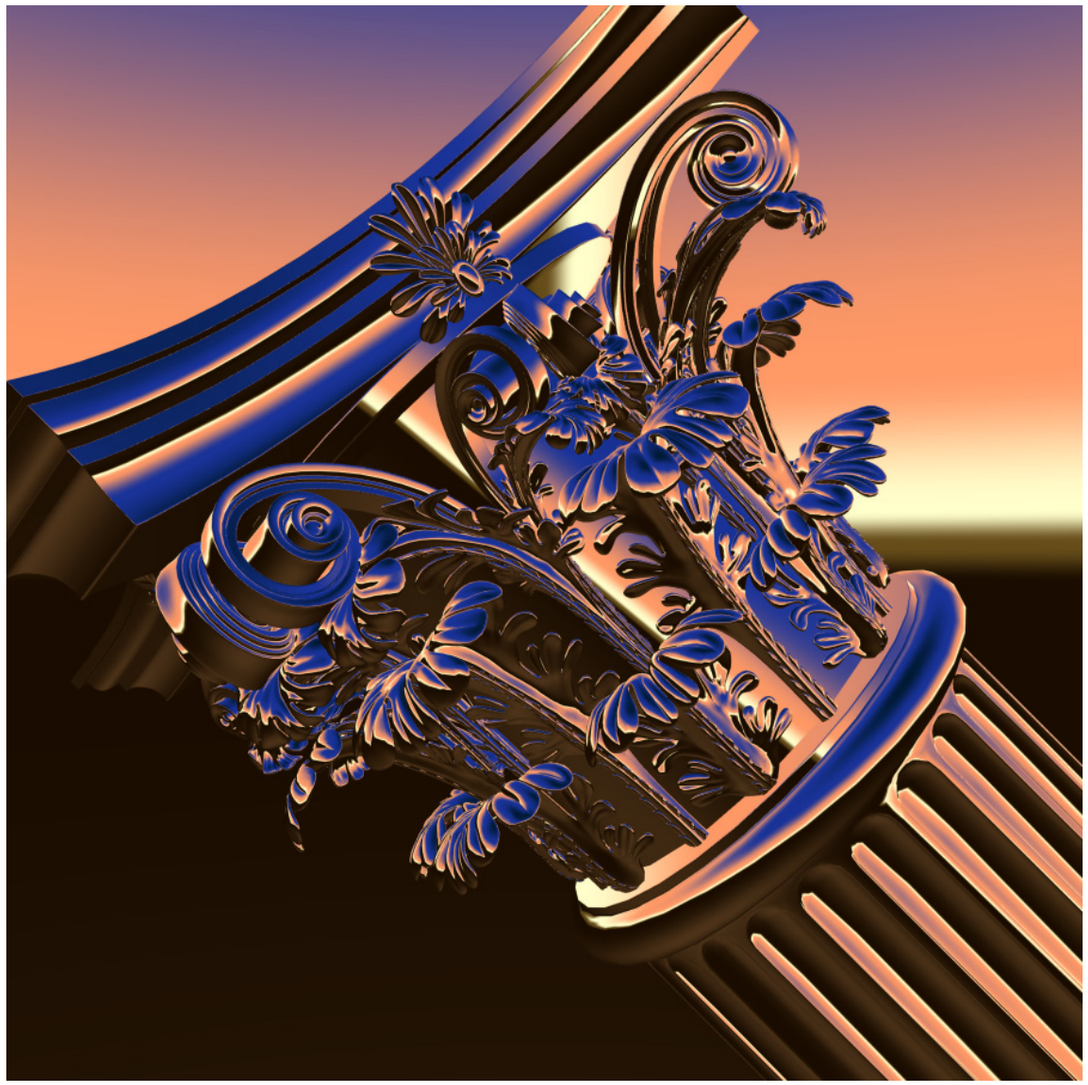
```
1  miBoolean color_ramp (
2      miColor *result, miState *state, struct color_ramp *params )
3  {
4      miColor *ramp = miaux_user_memory_pointer(state, 0);
5      miaux_interpolated_color_lookup(
6          result, ramp, RAMPSIZE, miaux_altitude(state));
7      return miTRUE;
8  }
```

Source code of main shader of "color_ramp"

```
1  miBoolean color_ramp_exit(miState *state, void *params)
2  {
3      return miaux_release_user_memory("color_ramp", state, params);
4  }
```

Source code of exit shader of "color_ramp"

The environment of the scene



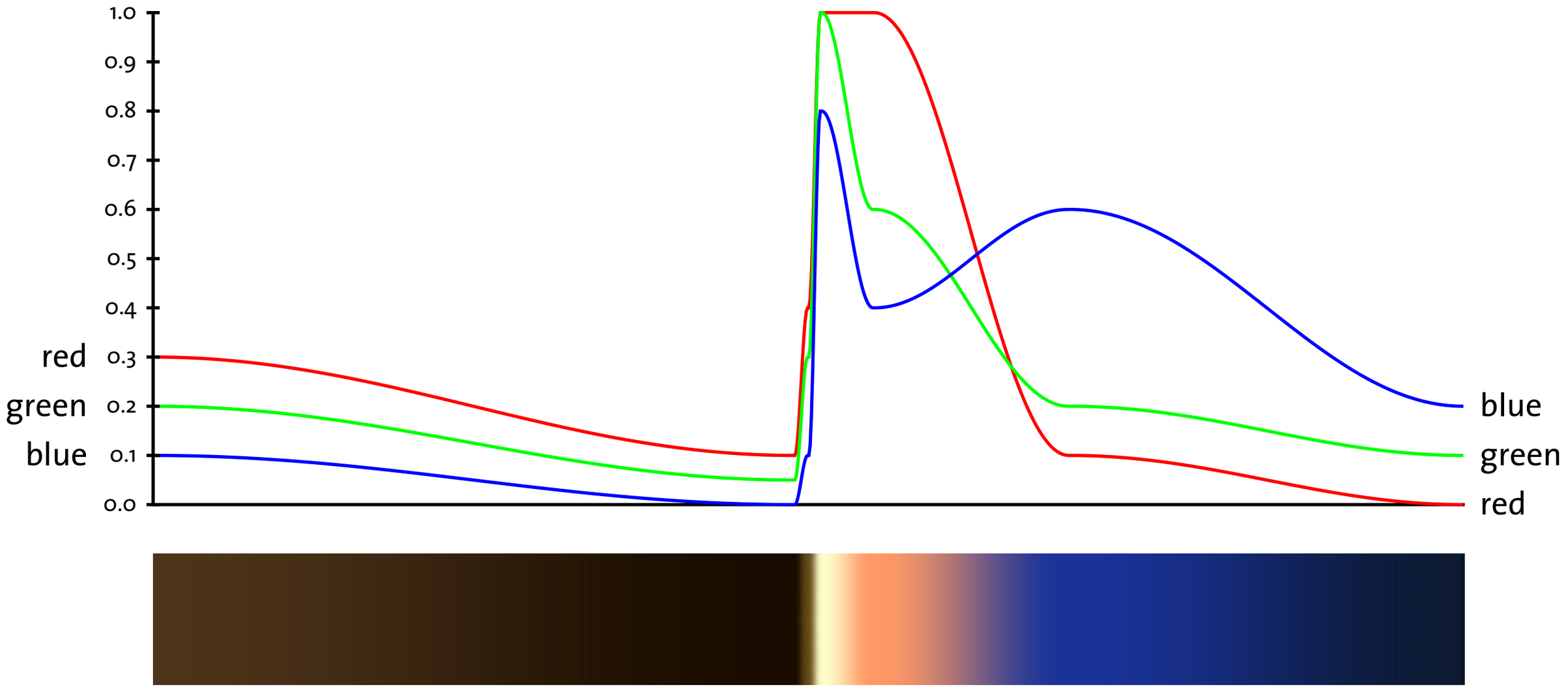
A color array as a shader parameter

```
shader "sunset"
    "color_ramp" (
        "colors"
            [ 0.3  0.2  0.1  0.0,
              0.1  0.05 0.0  0.49,
              0.4  0.3  0.1  0.50,
              1.0  1.0  0.8  0.51,
              1.0  0.6  0.4  0.55,
              0.1  0.2  0.6  0.7,
              0.0  0.1  0.2  1.0 ] )

camera "cam"
    output "rgba" "tif" "environment_5.tif"
    focal 1.5
    aperture 1.5
    aspect 1
    resolution 300 300
    environment
        = "sunset"
end camera

material "corinthian_material"
    "specular_reflection" ()
end material
```

Color ramp values defined by an array of colors with alpha as the key value

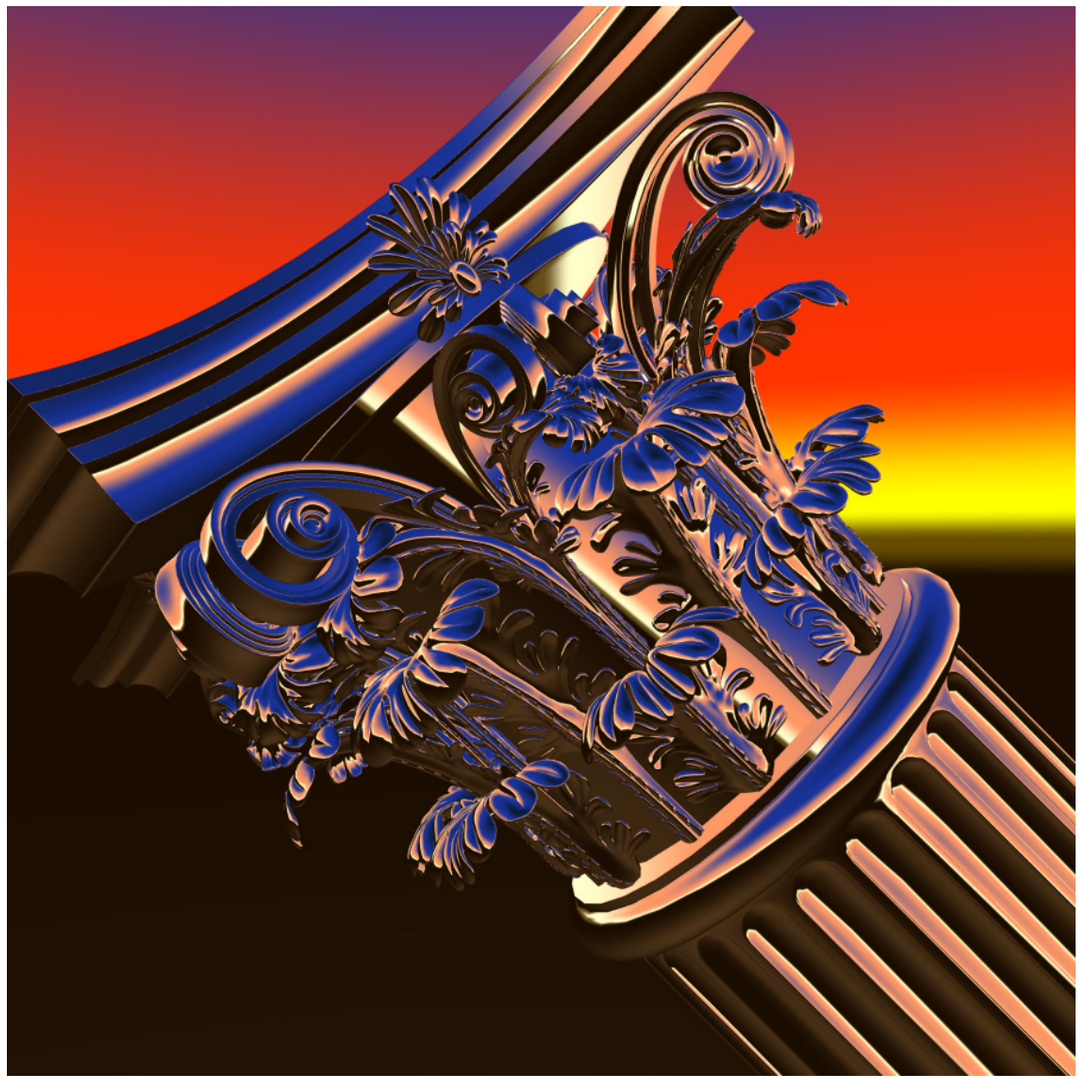
The environment of the scene

A color array as a shader parameter



Color curves specified in the scene file for shader color_ramp

The environment of the scene



Environment shaders for cameras and objects

```
shader "red_sunset"
   "color_ramp" (
      "colors" [ 0.3  0.2  0.1  0.0,
                 0.1  0.05 0.0  0.49,
                 0.4  0.3  0.0  0.5,
                 1.0  1.0  0.0  0.51,
                 1.0  0.2  0.0  0.55,
                 0.1  0.2  0.6  0.7,
                 0.12 0.05 0.2  1.0 ] )

shader "pink_sunset"
   "color_ramp" (
      "colors" [ 0.3  0.2  0.1  0.0,
                 0.1  0.05 0.0  0.49,
                 0.4  0.3  0.1  0.50,
                 1.0  1.0  0.8  0.51,
                 1.0  0.6  0.4  0.55,
                 0.1  0.2  0.6  0.7,
                 0.05 0.1  0.2  1.0 ] )

camera "cam"
    output "rgba" "tif" "environment_6.tif"
    focal 1.5
    aperture 1.5
    aspect 1
    resolution 300 300
    environment
        = "red_sunset"
end camera

material "corinthian_material"
    "specular_reflection" ()
    environment = "pink_sunset"
end material
```

Different environment shaders used for the camera and object

The environment of the scene

```
shader "sunset"
   "color_ramp" (
      "colors"
         [ 0.3  0.2  0.1  0.0,
           0.1  0.05 0.0  0.49,
           0.4  0.3  0.0  0.5,
           1.0  1.0  0.0  0.51,
           1.0  0.2  0.0  0.55,
           0.1  0.2  0.6  0.7,
           0.12 0.05 0.2  1.0 ] )
```

Encapsulating constant data in a Phenomenon

```
declare phenomenon
   color "red_sunset" ()
      shader "sunset"
         "color_ramp" (
            "colors"
               [ 0.3  0.2  0.1  0.0,
                 0.1  0.05 0.0  0.49,
                 0.4  0.3  0.0  0.5,
                 1.0  1.0  0.0  0.51,
                 1.0  0.2  0.0  0.55,
                 0.1  0.2  0.6  0.7,
                 0.12 0.05 0.2  1.0 ] )
      root = "sunset"
end declare
```

Encapsulating a named shader in a Phenomenon

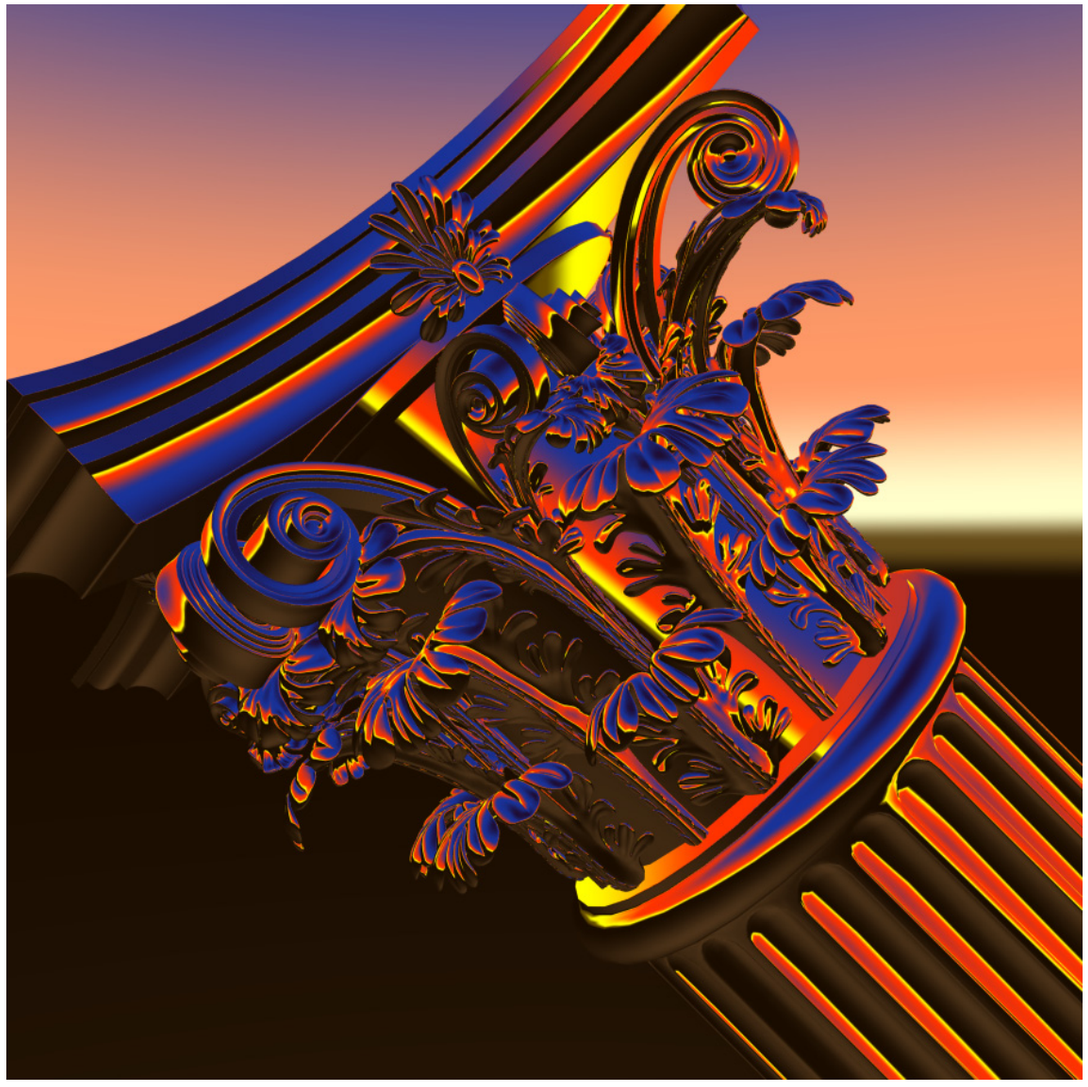
The environment of the scene

Encapsulating constant data in a Phenomenon

```
declare phenomenon
    color "red_sunset" ()
    shader "sunset"
        "color_ramp" (
            "colors"
                [ 0.3  0.2  0.1  0.0,
                  0.1  0.05 0.0  0.49,
                  0.4  0.3  0.0  0.5,
                  1.0  1.0  0.0  0.51,
                  1.0  0.2  0.0  0.55,
                  0.1  0.2  0.6  0.7,
                  0.12 0.05 0.2  1.0 ]
        )
    root = "sunset"
end declare

declare phenomenon
    color "pink_sunset" ()
    shader "sunset"
        "color_ramp" (
            "colors"
                [ 0.3  0.2  0.1  0.0,
                  0.1  0.05 0.0  0.49,
                  0.4  0.3  0.1  0.50,
                  1.0  1.0  0.8  0.51,
                  1.0  0.6  0.4  0.55,
                  0.1  0.2  0.6  0.7,
                  0.05 0.1  0.2  1.0 ]
        )
    root = "sunset"
end declare
```

Encapsulating a particular set of color_ramp parameters as Phenomena

The environment of the scene



Encapsulating constant data in a Phenomenon

```
camera "cam"
    output "rgba" "tif" "environment_7.tif"
    focal 1.5
    aperture 1.5
    aspect 1
    resolution 300 300
    environment
        "pink_sunset" ()
end camera

material "corinthian_material"
    "specular_reflection" ()
    environment
        "red_sunset" ()
end material
```

Defining Phenomena to encapsulate complex shader parameter settings

Exercise 18: Environment shaders

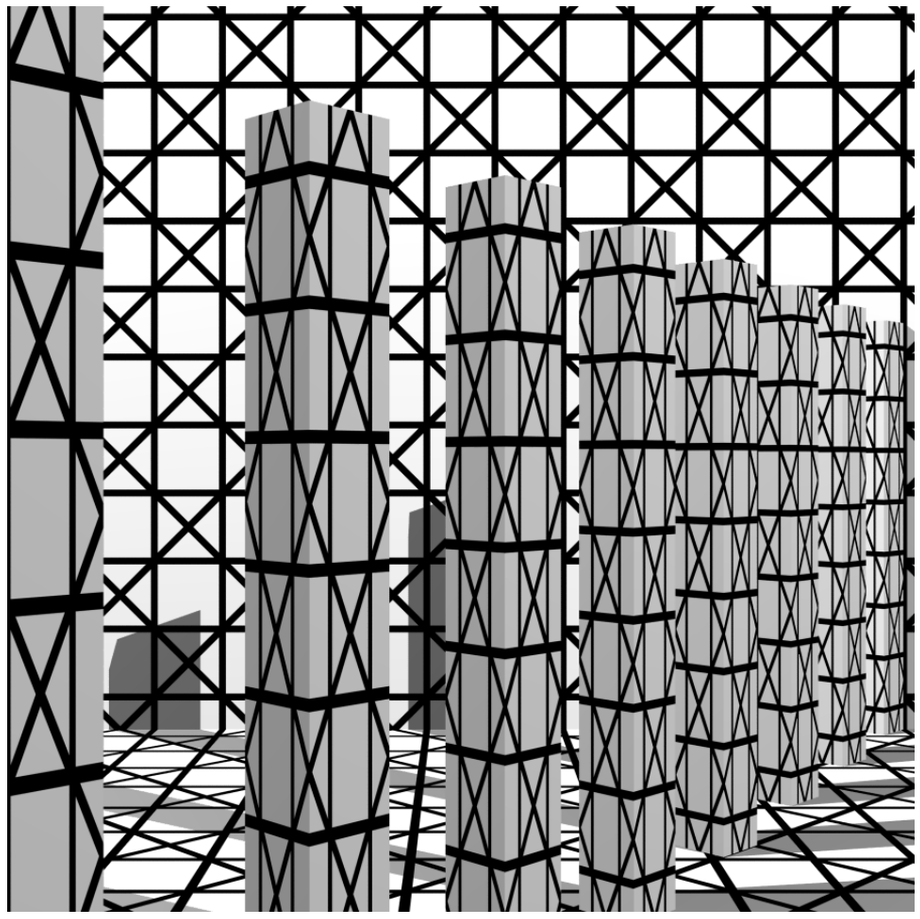
1. Copy `environment_1.mi` to `environment.mi`, change output file to `environment.tif`, render and view.
2. Add `environment` statement to camera block — refer to `environment_2.mi`
3. Change object material to use specular reflection — refer to `reflection_1.mi`
4. Change camera environment to use a Phenomenon — refer to `environment_7.mi`
5. Find the statements that allocate and deallocate memory for the color ramp in shader `color_ramp`

A visible atmosphere

A visible atmosphere

- Volume shaders and the camera
- Changing the fog density
- Adding a debugging mode to a shader
- Varying the ground fog layer positions

A visible atmosphere



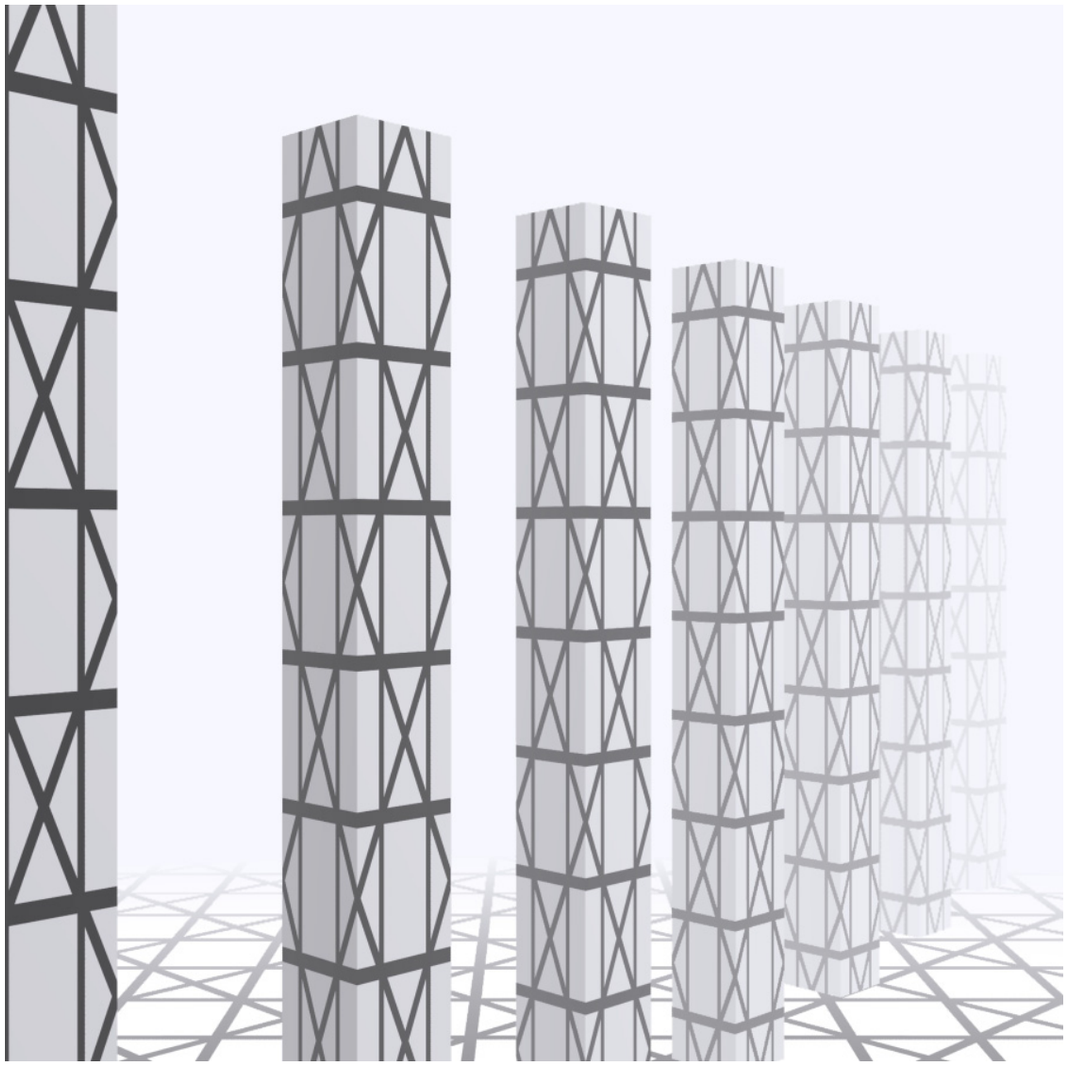
```
camera "cam"
    output "rgba" "tif"
        "atmosphere_1.tif"
    focal 1.5
    aperture 1
    aspect 1
    resolution 300 300
end camera
```

A scene with a typical camera

```
declare shader
    color "fog" (
        scalar "full_fade_distance" default 10,
        color "fog_color" default 1 1 1 )
end declare
```

Scene file declaration of shader "fog"

A visible atmosphere

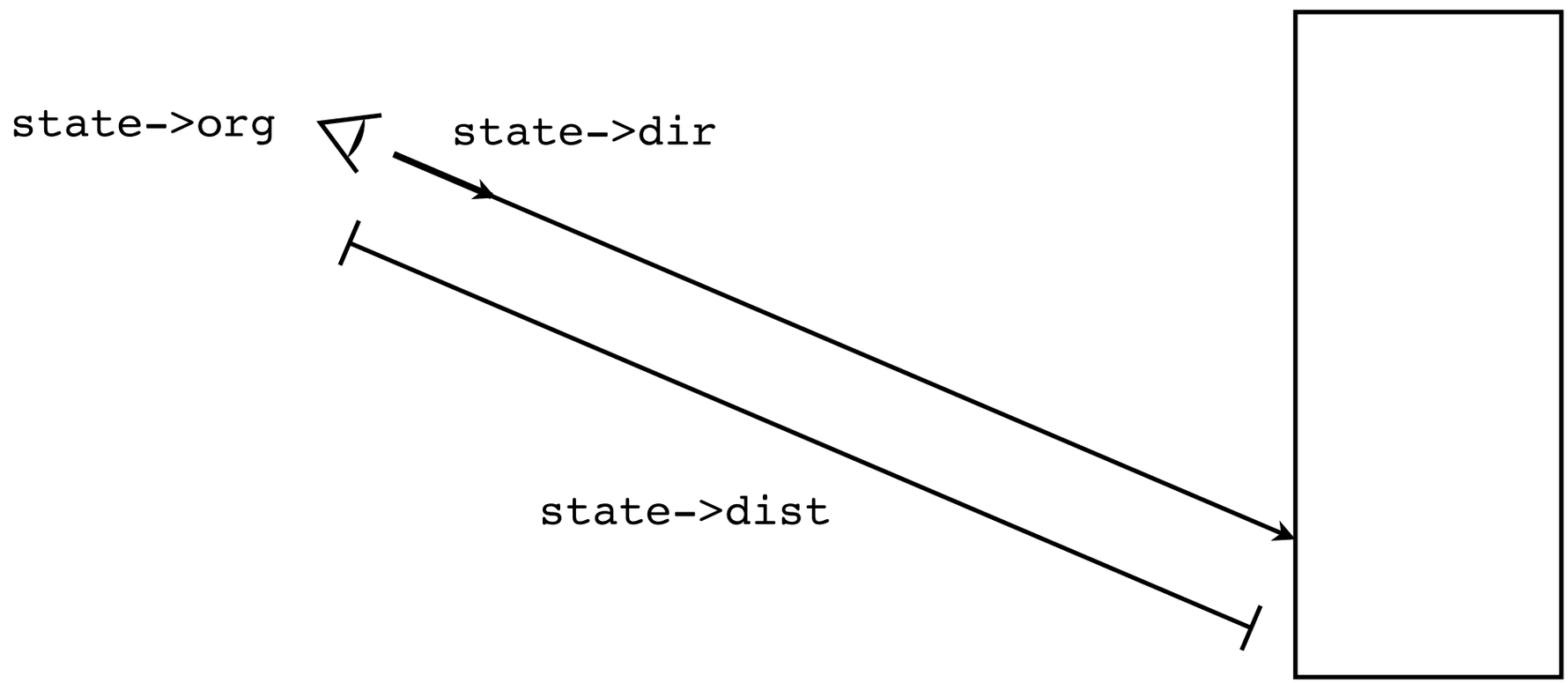


```
camera "cam"
    output "rgba" "tif"
        "atmosphere_2.tif"
    focal 1.5
    aperture 1
    aspect 1
    resolution 300 300
    volume
        "fog" (
            "full_fade_distance" 5,
            "fog_color" .97 .97 1 )
end camera
```

Camera with fog volume shader, full fade distance set to 5

A visible atmosphere

Volume shaders and the camera

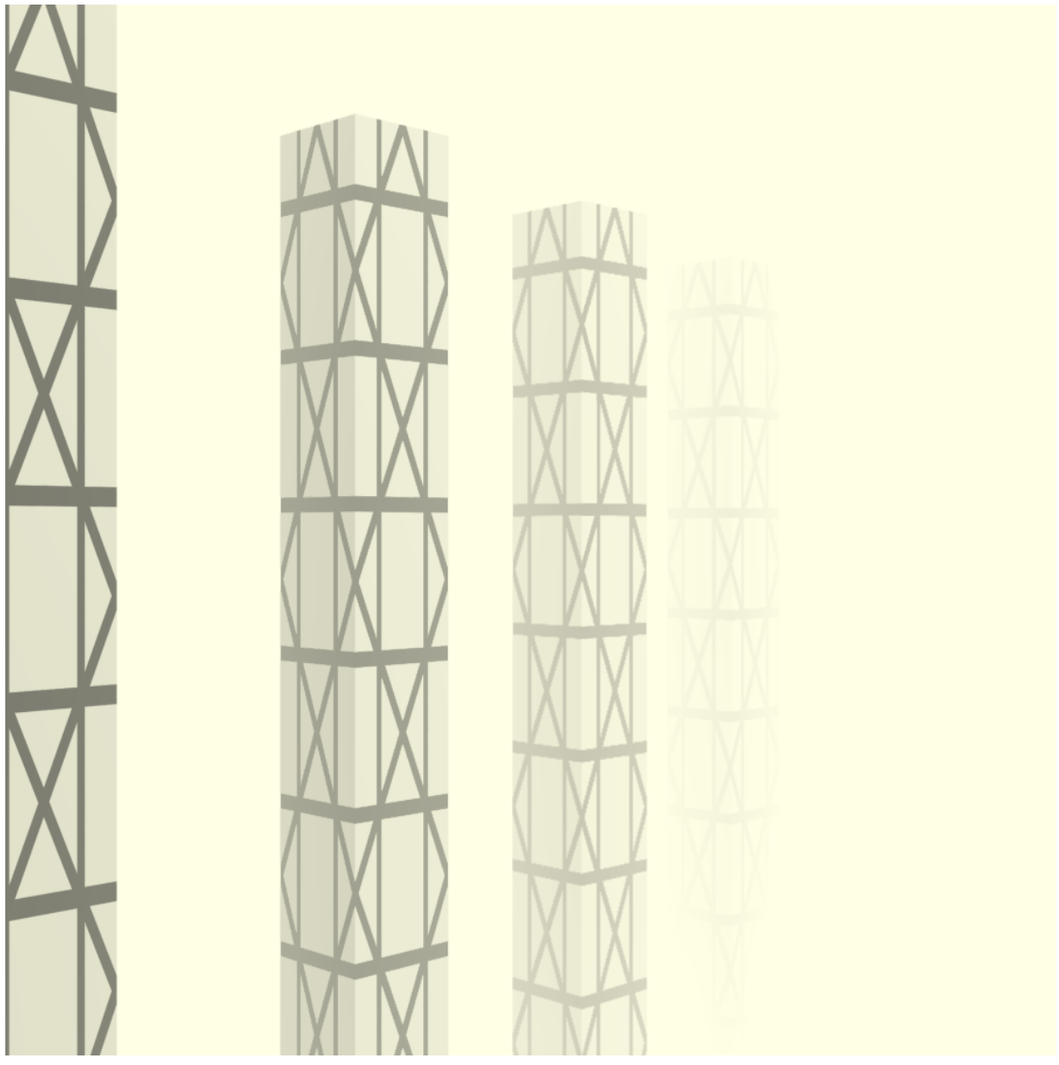


Fields in miState useful in a volume shader

```
 1  struct fog {
 2      miScalar full_fade_distance;
 3      miColor fog_color;
 4  };
 5
 6  miBoolean fog (
 7       miColor *result, miState *state, struct fog *params )
 8  {
 9      miScalar full_fade_distance = *mi_eval_scalar(&params->full_fade_distance);
10      miColor *fog_color = mi_eval_color(&params->fog_color);
11
12      if (state->dist > full_fade_distance || state->dist == 0.0)
13          *result = *fog_color;
14      else
15          miaux_blend_colors(
16              result, fog_color, result, state->dist/full_fade_distance);
17
18      return miTRUE;
19  }
```

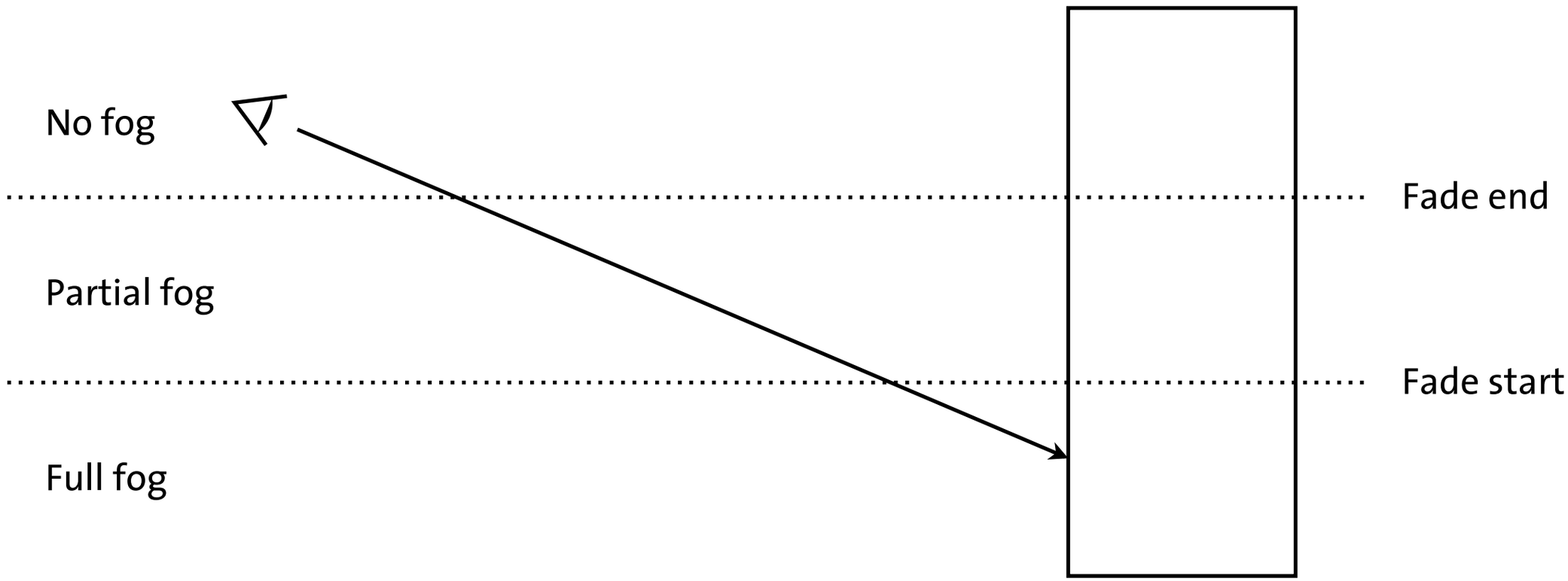
Source code of shader "fog"

A visible atmosphere



```
camera "cam"
    output "rgba" "tif"
        "atmosphere_3.tif"
    focal 1.5
    aperture 1
    aspect 1
    resolution 300 300
    volume
        "fog" (
            "full_fade_distance" 3.1,
            "fog_color" 1 1 .9 )
end camera
```

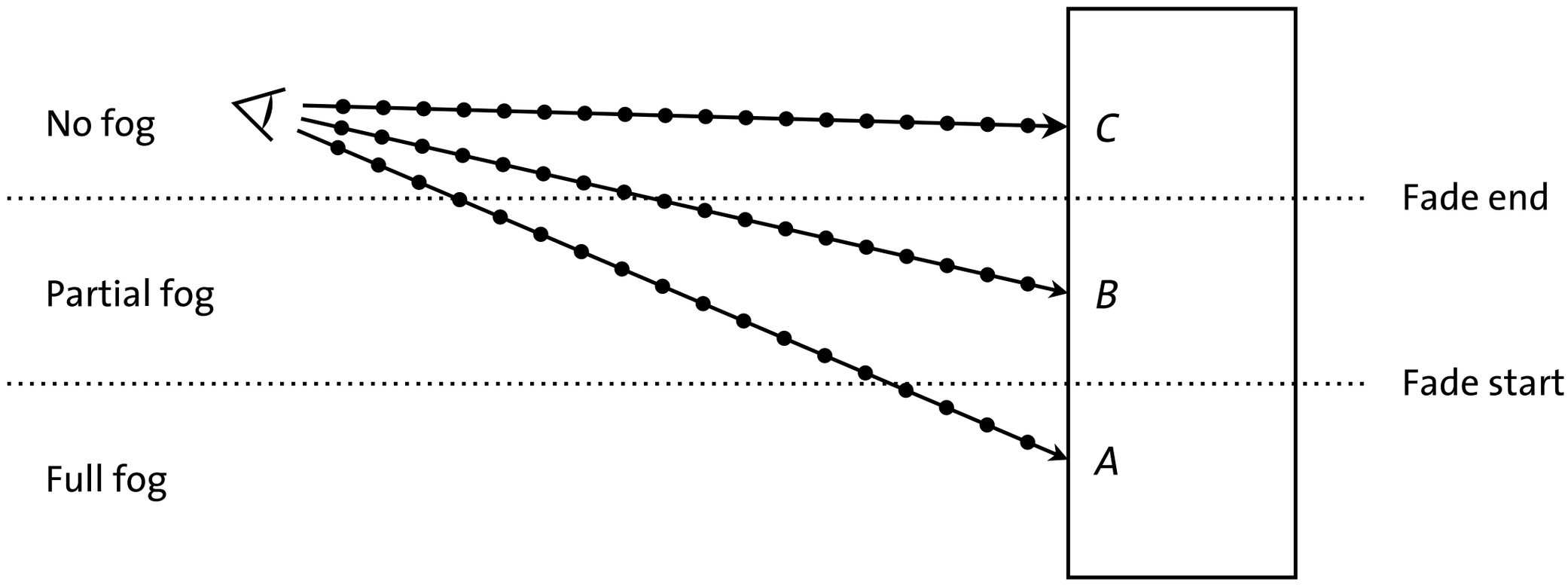
Camera with fog volume shader, full fade distance set to 3.1



Defining different fog density based on height

A visible atmosphere

Changing the fog density



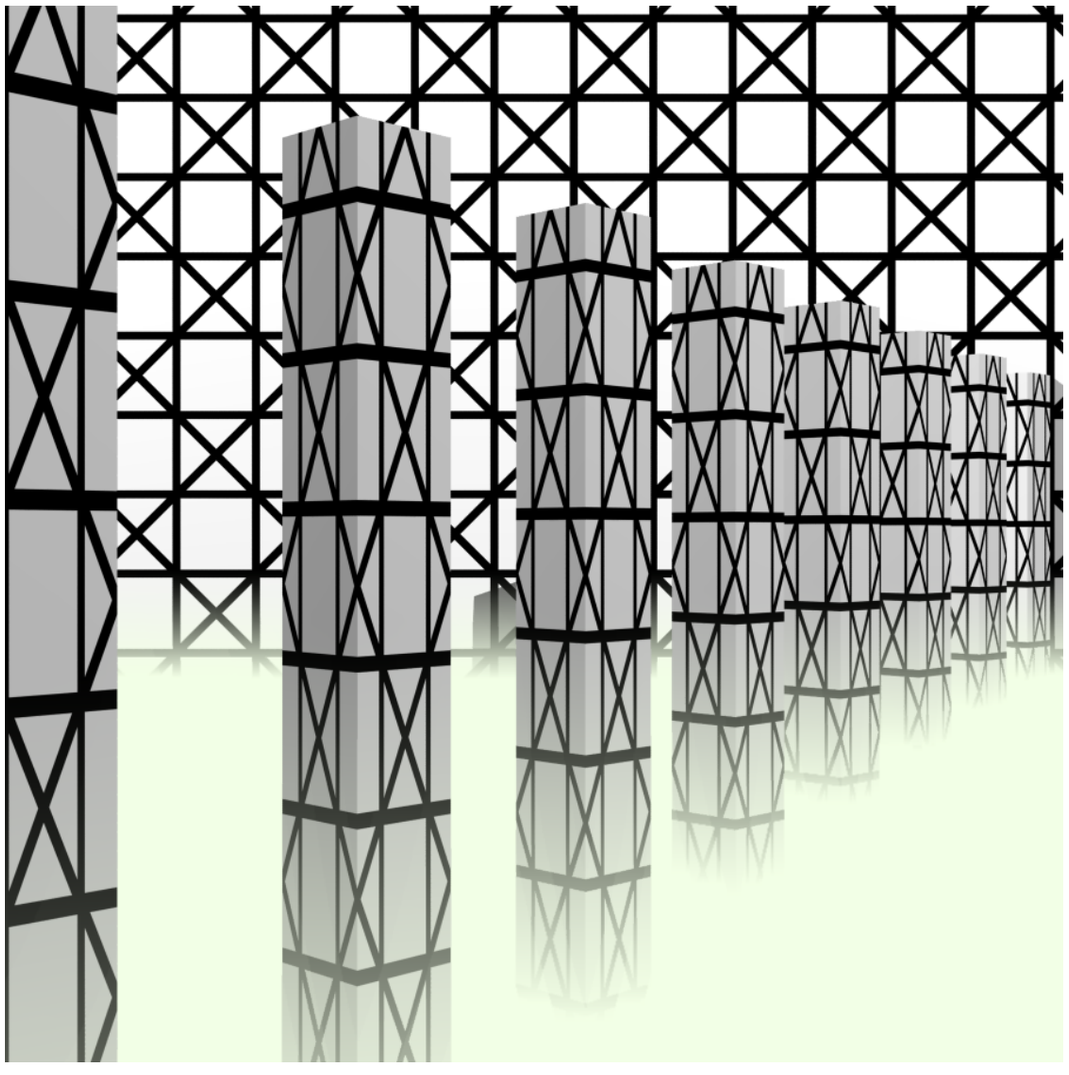
Points along the ray for fog density accumulation

```
declare shader
    color "ground_fog" (
        color   "fog_color"          default 1 1 1,
        color   "fog_density"        default 1 1 1,
        scalar  "fade_start"         default 0,
        scalar  "fade_end"           default 1,
        scalar  "unit_density"       default 1,
        scalar  "march_increment"    default 0.1, )
end declare
```

Scene file declaration of shader "ground_fog"

A visible atmosphere

Changing the fog density



```
camera "cam"
    output "rgba" "tif"
        "atmosphere_4.tif"
    focal 1.5
    aperture 1
    aspect 1
    resolution 300 300
    volume
        "ground_fog" (
            "fog_color" .95 1 .9,
            "fade_start" -.8,
            "fade_end" -.1,
            "march_increment" .05 )
end camera
```

The effect of ray marching in a volume shader to simulate fog of non-uniform density

```
1  void miaux_point_along_vector(
2      miVector *result, miVector *point, miVector *direction, miScalar distance)
3  {
4      result->x = point->x + distance * direction->x;
5      result->y = point->y + distance * direction->y;
6      result->z = point->z + distance * direction->z;
7  }
```

Auxiliary function: miaux_point_along_vector

```
1  void miaux_march_point(
2      miVector *result, miState *state, miScalar distance)
3  {
4      miaux_point_along_vector(result, &state->org, &state->dir, distance);
5  }
```

Auxiliary function: miaux_march_point

```
1  void miaux_world_space_march_point(
2      miVector *result, miState *state, miScalar distance)
3  {
4      miaux_march_point(result, state, distance);
5      mi_vector_to_world(state, result, result);
6  }
```

Auxiliary function: miaux_world_space_march_point

A visible atmosphere

Changing the fog density

```
 1  struct ground_fog {
 2      miColor fog_color;
 3      miColor fog_density;
 4      miScalar fade_start;
 5      miScalar fade_end;
 6      miScalar unit_density;
 7      miScalar march_increment;
 8  };
 9
10  miBoolean ground_fog (
11      miColor *result, miState *state, struct ground_fog *params )
12  {
13      miScalar fade_start, fade_end, fog_density, distance, density_factor,
14          march_increment, unit_density, accumulated_density;
15      miVector march_point;
16
17      if (state->dist == 0)
18          return miTRUE;
19
20      fade_start = *mi_eval_scalar(&params->fade_start);
21      fade_end = *mi_eval_scalar(&params->fade_end);
22      fog_density = *mi_eval_scalar(&params->fog_density);
23      march_increment = *mi_eval_scalar(&params->march_increment);
24      unit_density = *mi_eval_scalar(&params->unit_density);
25      accumulated_density = 0.0;
26
27      for (distance = 0; distance < state->dist; distance += march_increment) {
28          miaux_world_space_march_point(&march_point, state, distance);
29          density_factor = miaux_fit_clamp(
30              march_point.y, fade_start, fade_end, fog_density, 0.0);
31          accumulated_density += density_factor * march_increment * unit_density;
32          if (accumulated_density > 1.0) {
33              *result = *mi_eval_color(&params->fog_color);
34              return miTRUE;
35          }
36      }
37      if (accumulated_density > 0.0)
38          miaux_blend_colors(result, result, mi_eval_color(&params->fog_color),
39                             1.0 - accumulated_density);
40      return miTRUE;
41  }
```

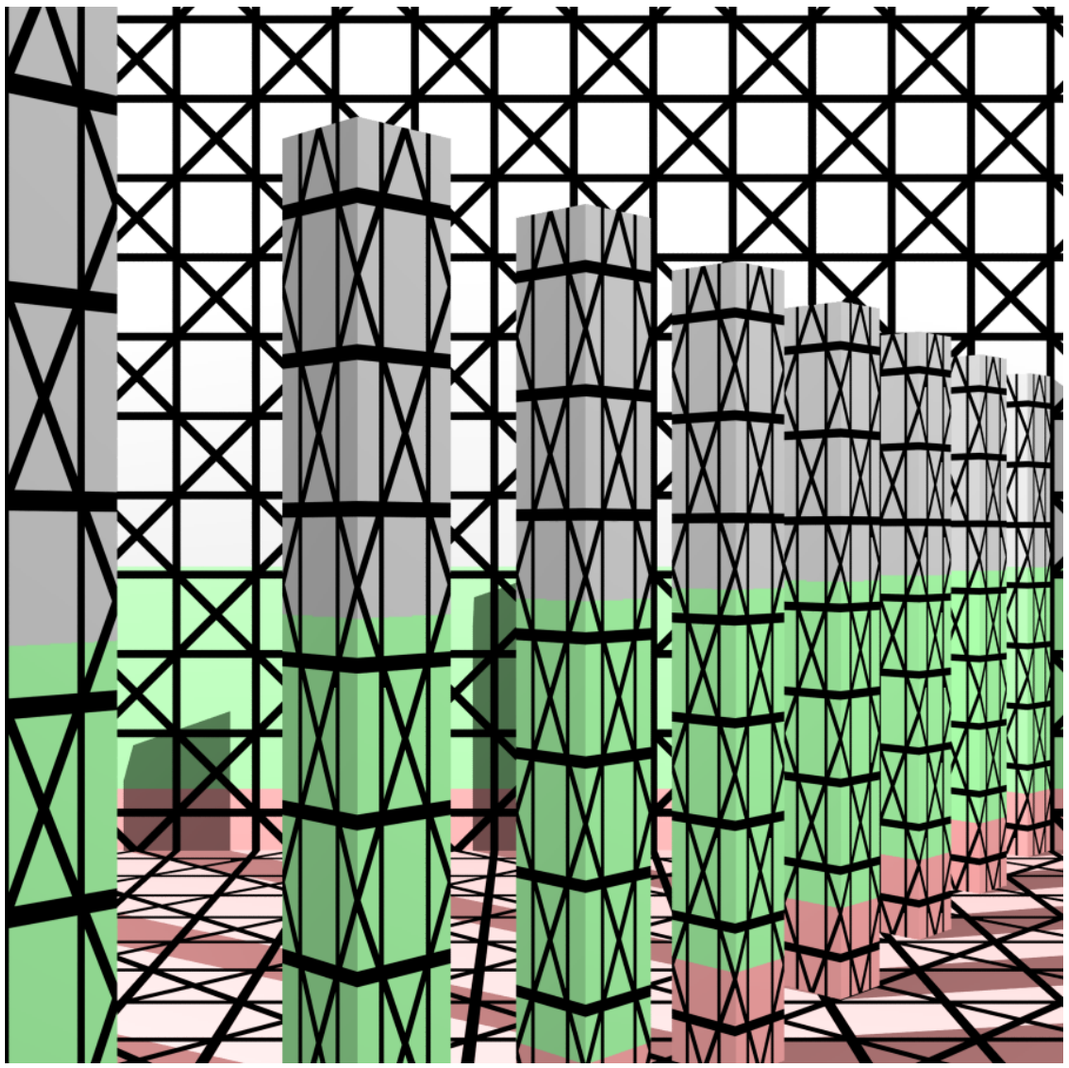
Source code of shader "ground_fog"

```
declare shader
    color "ground_fog_layers" (
        color   "fog_color"          default 1 1 1,
        color   "fog_density"        default 1 1 1,
        scalar  "fade_start"         default 0,
        scalar  "fade_end"           default 1,
        scalar  "unit_density"       default 1,
        scalar  "march_increment"    default 0.1,
        boolean "show_layers"        default off,
        color   "full_fog_marker"    default 1.1 .9 .9,
        color   "partial_fog_marker" default .9 1.1 .9 )
end declare
```

Scene file declaration of shader "ground_fog_layers"

A visible atmosphere

Adding a debugging mode to a shader



```
camera "cam"
    output "rgba" "tif"
        "atmosphere_5.tif"
    focal 1.5
    aperture 1
    aspect 1
    resolution 300 300
    volume
        "ground_fog_layers" (
            "show_layers" on,
            "fog_color" .95 1 .9,
            "fade_start" -.8,
            "fade_end" -.1,
            "march_increment" .01 )
end camera
```

Using the show_regions parameter of ground_fog_layers to display the fog layers

A visible atmosphere

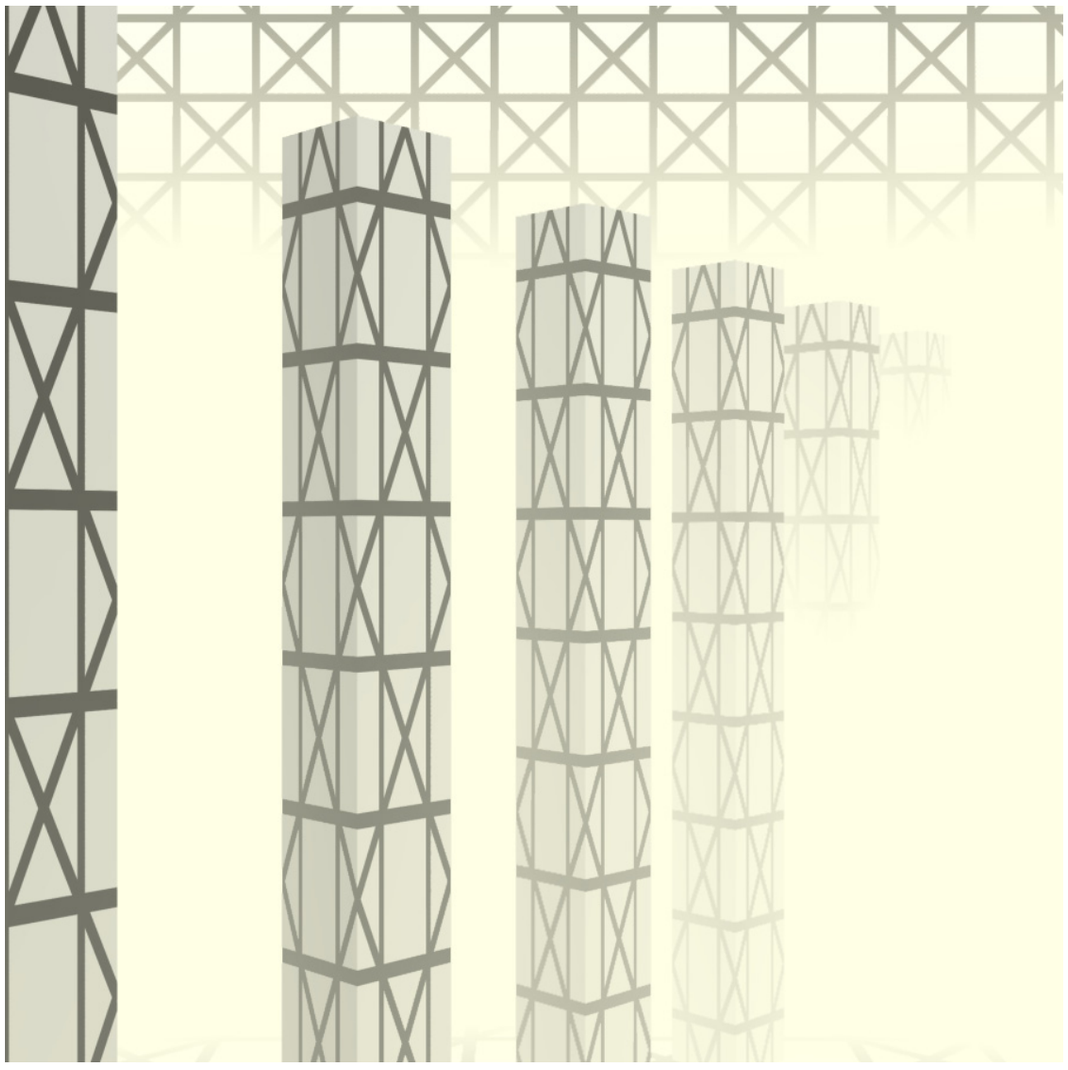
Adding a debugging mode to a shader

```
 1  struct ground_fog_layers {
 2      miColor fog_color;
 3      miColor fog_density;
 4      miScalar fade_start;
 5      miScalar fade_end;
 6      miScalar unit_density;
 7      miScalar march_increment;
 8      miBoolean show_layers;
 9      miColor full_fog_marker;
10      miColor partial_fog_marker;
11  };
12
13  miBoolean ground_fog_layers (
14       miColor *result, miState *state, struct ground_fog_layers *params )
15  {
16      miScalar fade_start = *mi_eval_scalar(&params->fade_start);
17      miScalar fade_end = *mi_eval_scalar(&params->fade_end);
18      miVector world_point, march_point;
19
20      if (state->dist == 0.0)
21          return miTRUE;
22      else if (*mi_eval_boolean(&params->show_layers)) {
23          miColor* full_fog_marker =
24              mi_eval_color(&params->full_fog_marker);
25          miColor* partial_fog_marker =
26              mi_eval_color(&params->partial_fog_marker);
27
28          mi_point_to_world(state, &world_point, &state->point);
29          if (world_point.y < fade_start)
30              miaux_multiply_colors(result, result, full_fog_marker);
31          else if (world_point.y < fade_end)
32              miaux_multiply_colors(result, result, partial_fog_marker);
33      } else {
34          miScalar fog_density = *mi_eval_scalar(&params->fog_density);
35          miScalar march_increment = *mi_eval_scalar(&params->march_increment);
36          miScalar unit_density = *mi_eval_scalar(&params->unit_density);
37          miScalar accumulated_density = 0.0, distance, density_factor;
38
39          for (distance = 0; distance < state->dist; distance += march_increment) {
40              miaux_world_space_march_point(&march_point, state, distance);
41              density_factor = miaux_fit_clamp(
42                  march_point.y, fade_start, fade_end, fog_density, 0.0);
43              accumulated_density +=
44                  density_factor * march_increment * unit_density;
45              if (accumulated_density > 1.0) {
46                  *result = *mi_eval_color(&params->fog_color);
47                  return miTRUE;
48              }
49          }
50          if (accumulated_density > 0.0) {
51              miaux_blend_colors(result, result, mi_eval_color(&params->fog_color),
52                                 1.0 - accumulated_density);
53          }
54      }
55      return miTRUE;
56  }
```

Source code of shader "ground_fog_layers"

A visible atmosphere

Varying the ground fog layer positions

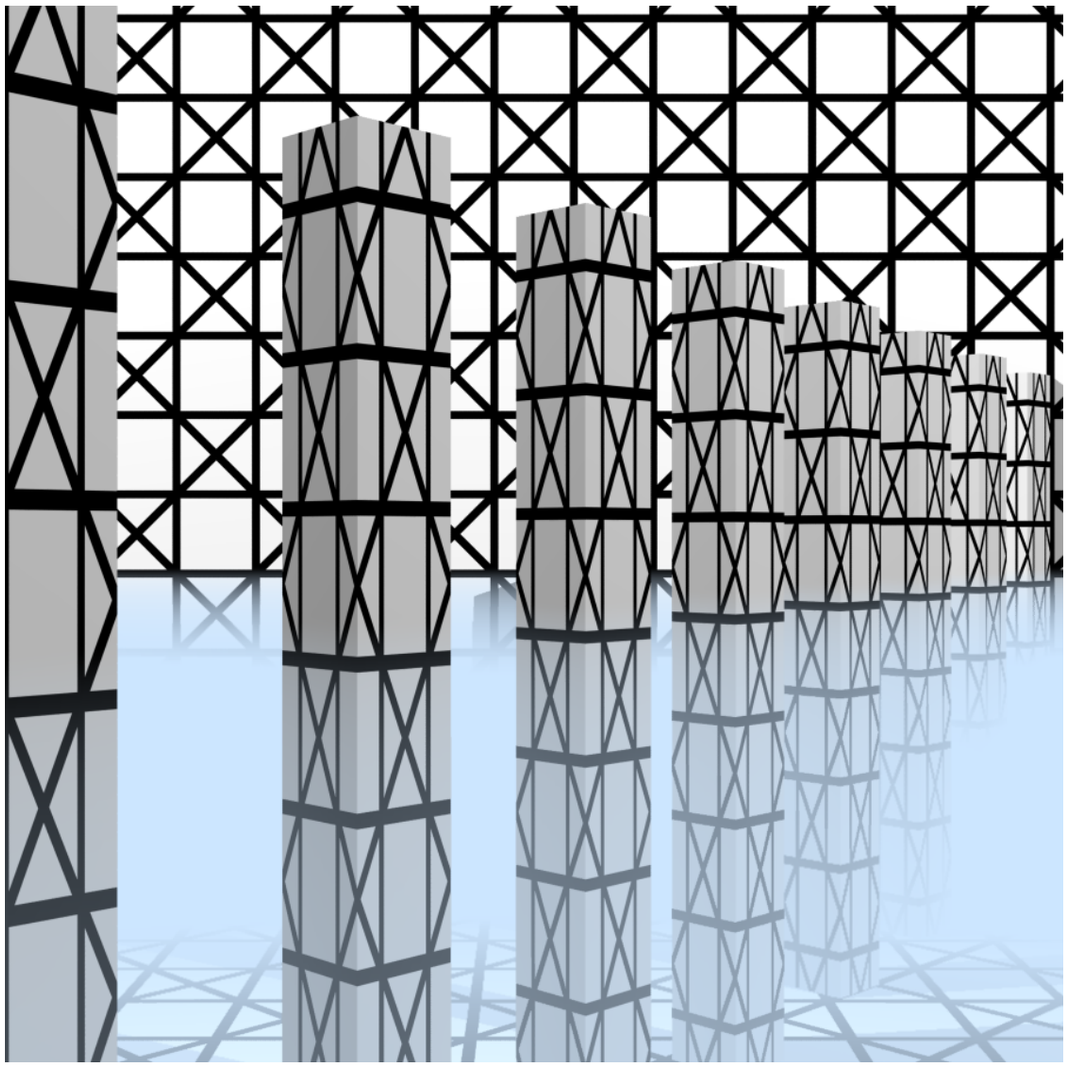


```
camera "cam"
    output "rgba" "tif"
        "atmosphere_6.tif"
    focal 1.5
    aperture 1
    aspect 1
    resolution 300 300
    volume
        "ground_fog" (
            "fog_color" 1 1 .9,
            "fade_start" -.2,
            "fade_end" 1.6,
            "fog_density" .5,
            "unit_density" .6,
            "march_increment" .01 )
end camera
```

Expanding the vertical range within which the fog dissipates

A visible atmosphere

Varying the ground fog layer positions



```
camera "cam"
    output "rgba" "tif"
        "atmosphere_7.tif"
    focal 1.5
    aperture 1
    aspect 1
    resolution 300 300
    volume
        "ground_fog" (
            "fog_color" .8 .9 1,
            "fade_start" -.12,
            "fade_end" -.1,
            "fog_density" .38,
            "unit_density" .7,
            "march_increment" .01 )
end camera
```

Compressing the vertical fade range and lessening the maximum density of the fog

Exercise 19: Atmosphere

1. Copy `atmosphere_1.mi` to `atmosphere.mi`, change output to `atmosphere.tif`
2. Add `fog` as a volume shader to the scene by attaching a named shader to the camera:

```
shader "white_fog"
    "fog" ( "full_fade_distance" 16 )
```

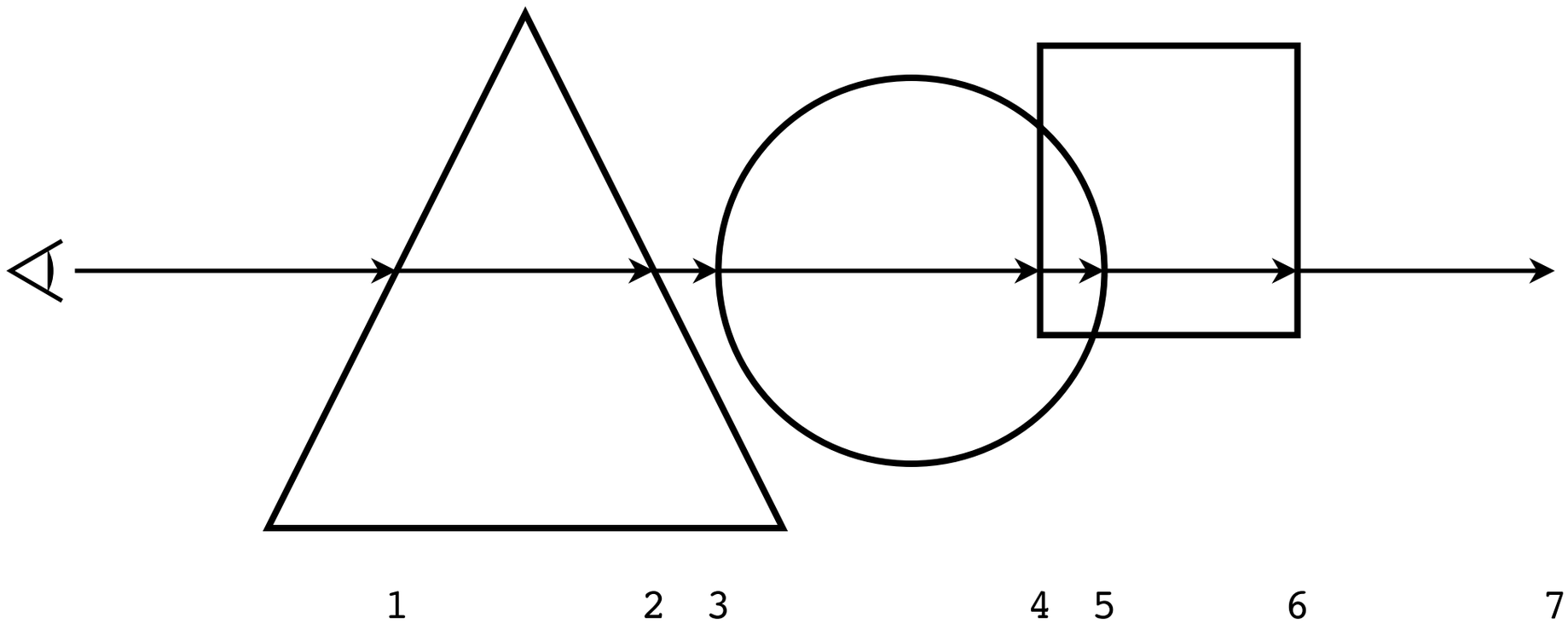
3. Add `white_fog` to the camera as a volume shader, render and view.
4. Add another shader definition of using `fog`, reducing distance to under 0.5.
5. Remove cube material shader, and add the new `fog` shader to the cube material.
6. In the application, try using an object volume shader without a material shader.

Volumetric effects

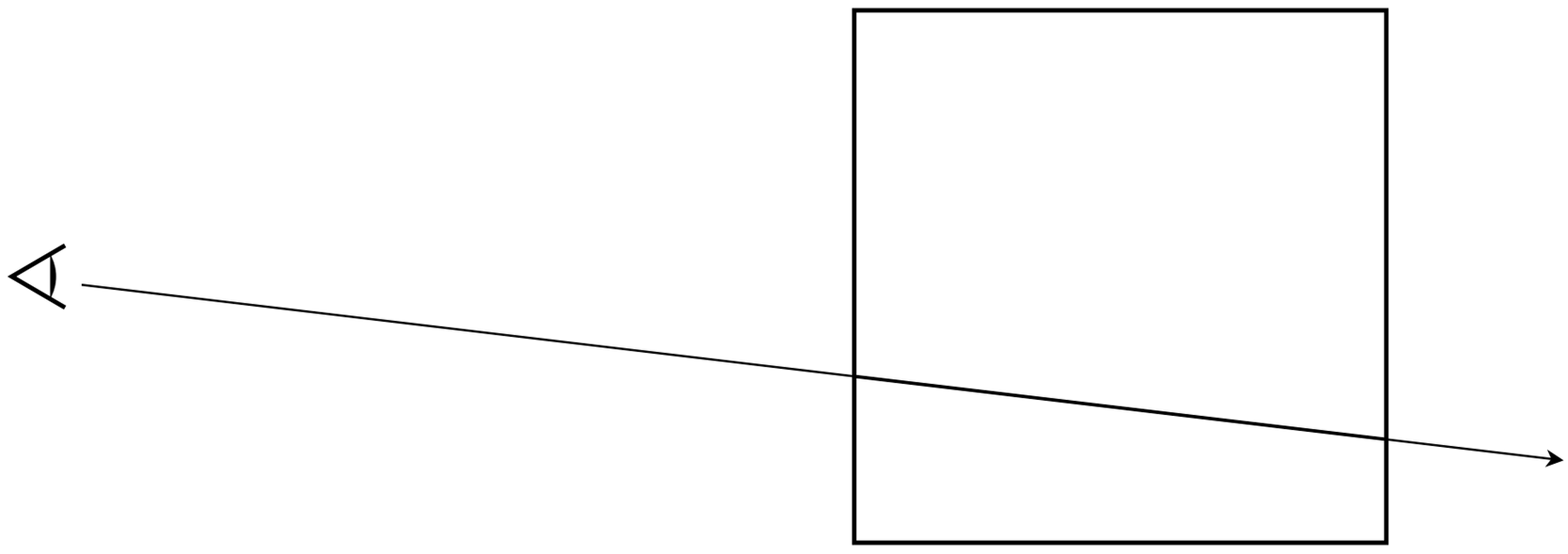
Volumetric effects

- Rays in volume shaders
- Using a threshold value in the volume
- Cumulative density for a ray in the volume
- Illumination of samples in the volume
    - Fractional occlusion
    - Total light at a point in the volume
    - Accumulating and blending volume colors
    - The illuminated volume
- Defining the density function with a shader
- Using voxel data sets for the density function
- Summary of the volume shaders

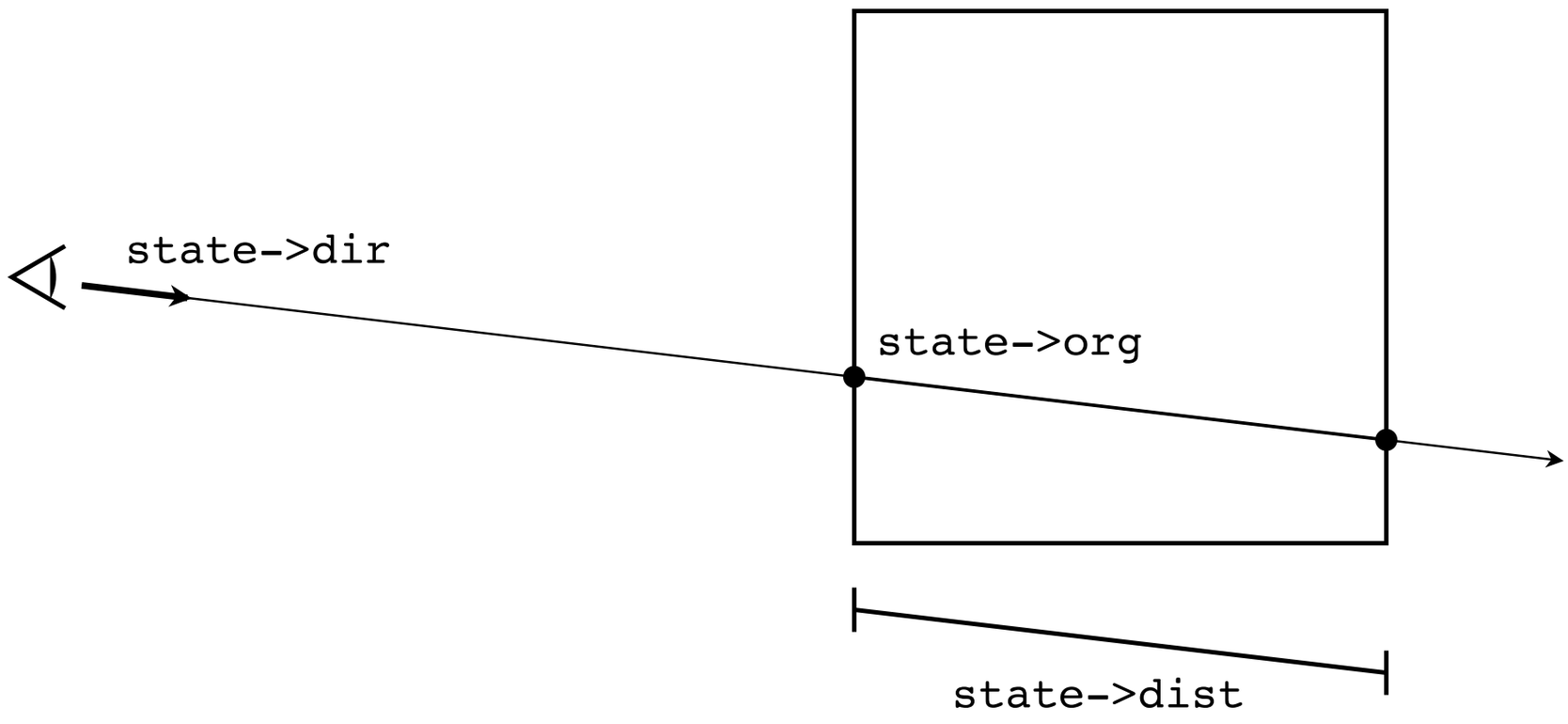
Volumetric effects



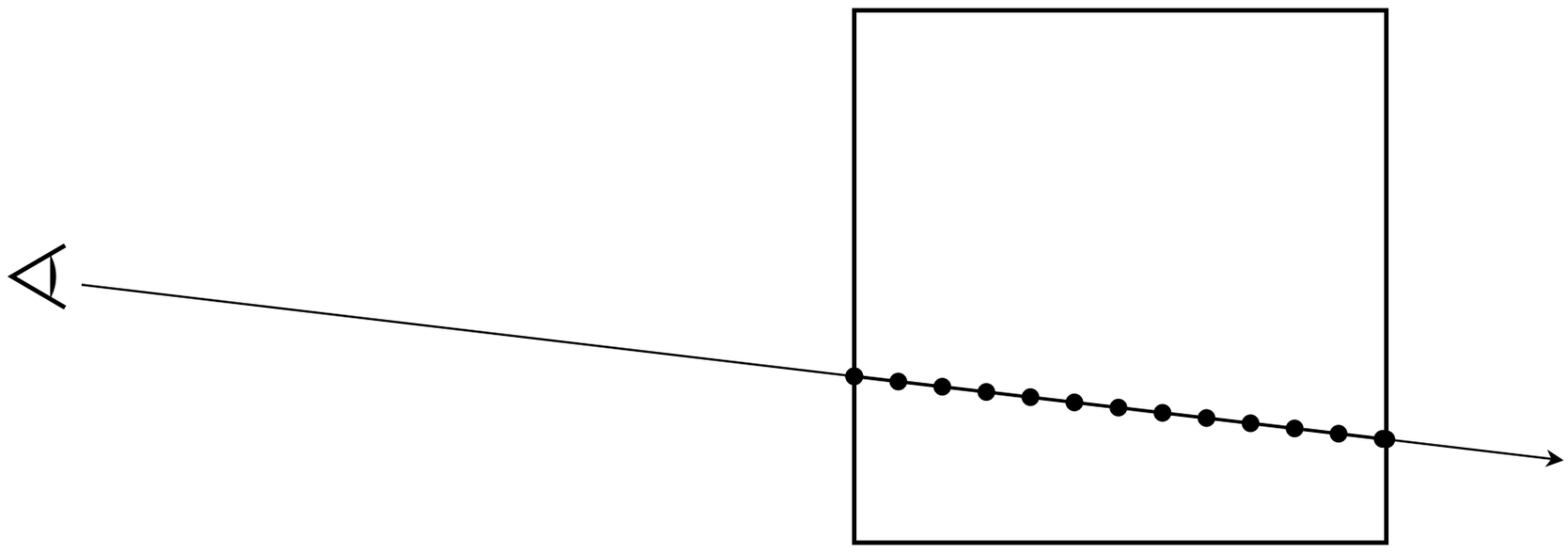
Accumulating the transparency effect of multiple objects along a ray



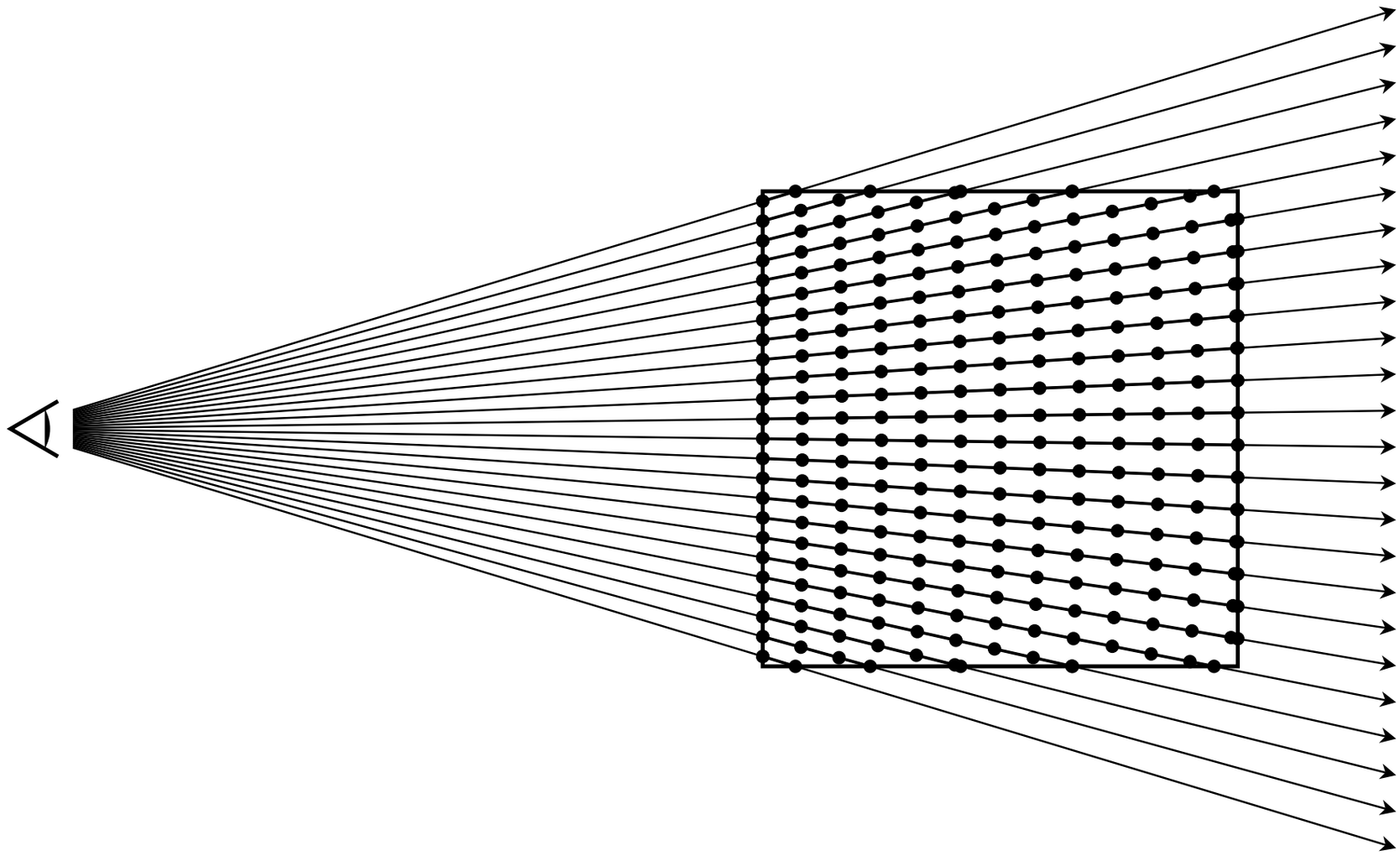
An eye ray passing through an object volume



State variables for the ray direction, intersection and length in the volume



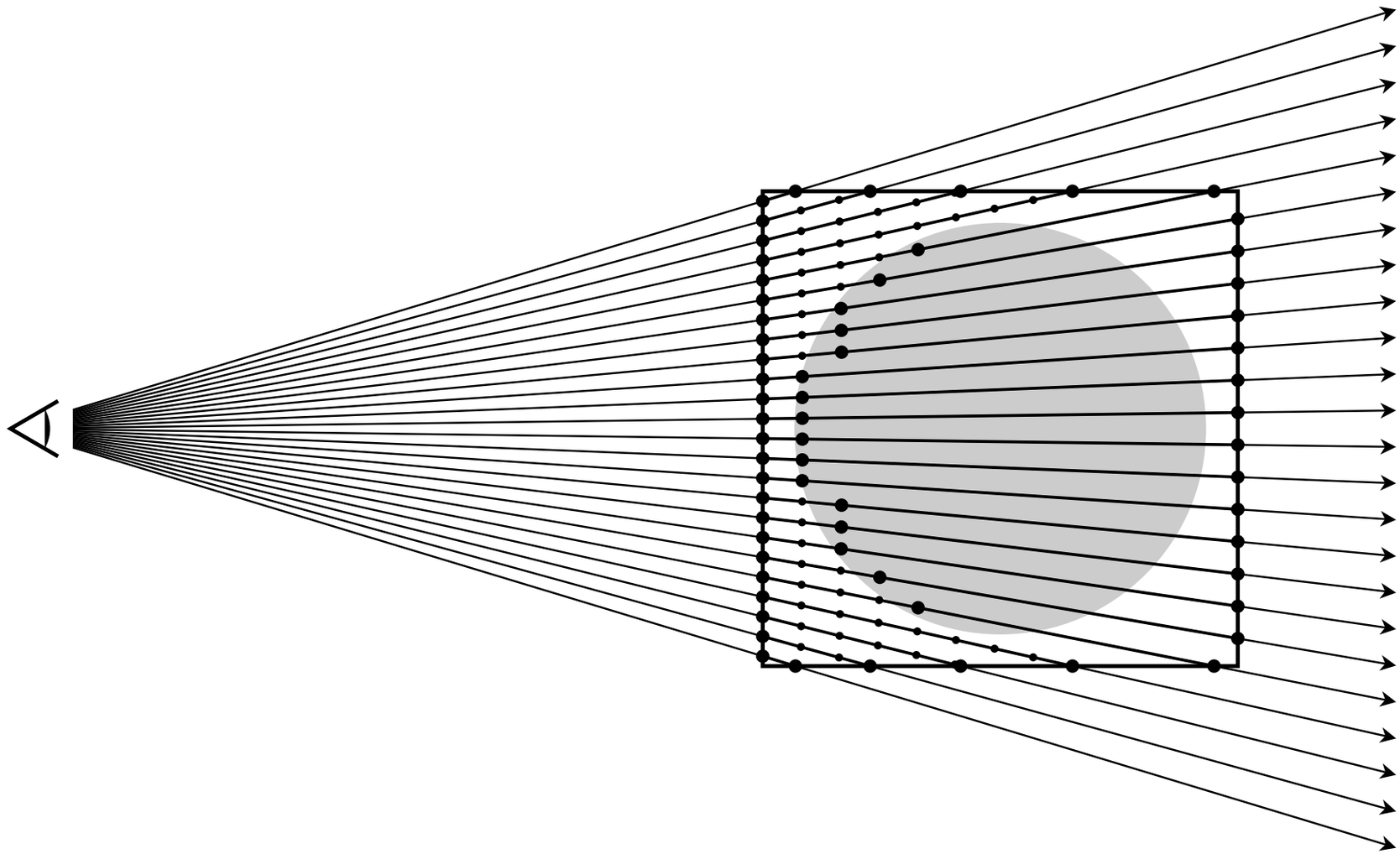
Sample points along the ray inside the volume



Sample points along all rays inside the volume

Volumetric effects

Using a threshold value in the volume



A spherical volume function and the first point encountered by the ray

Using a threshold value in the volume

```
1  miScalar miaux_threshold_density(
2      miVector *point, miVector *center, miScalar radius,
3      miScalar unit_density, miScalar march_increment)
4  {
5      miScalar distance = mi_vector_dist(center, point);
6      if (distance <= radius)
7          return unit_density * march_increment;
8      else
9          return 0.0;
10 }
```

Auxiliary function: miaux_threshold_density

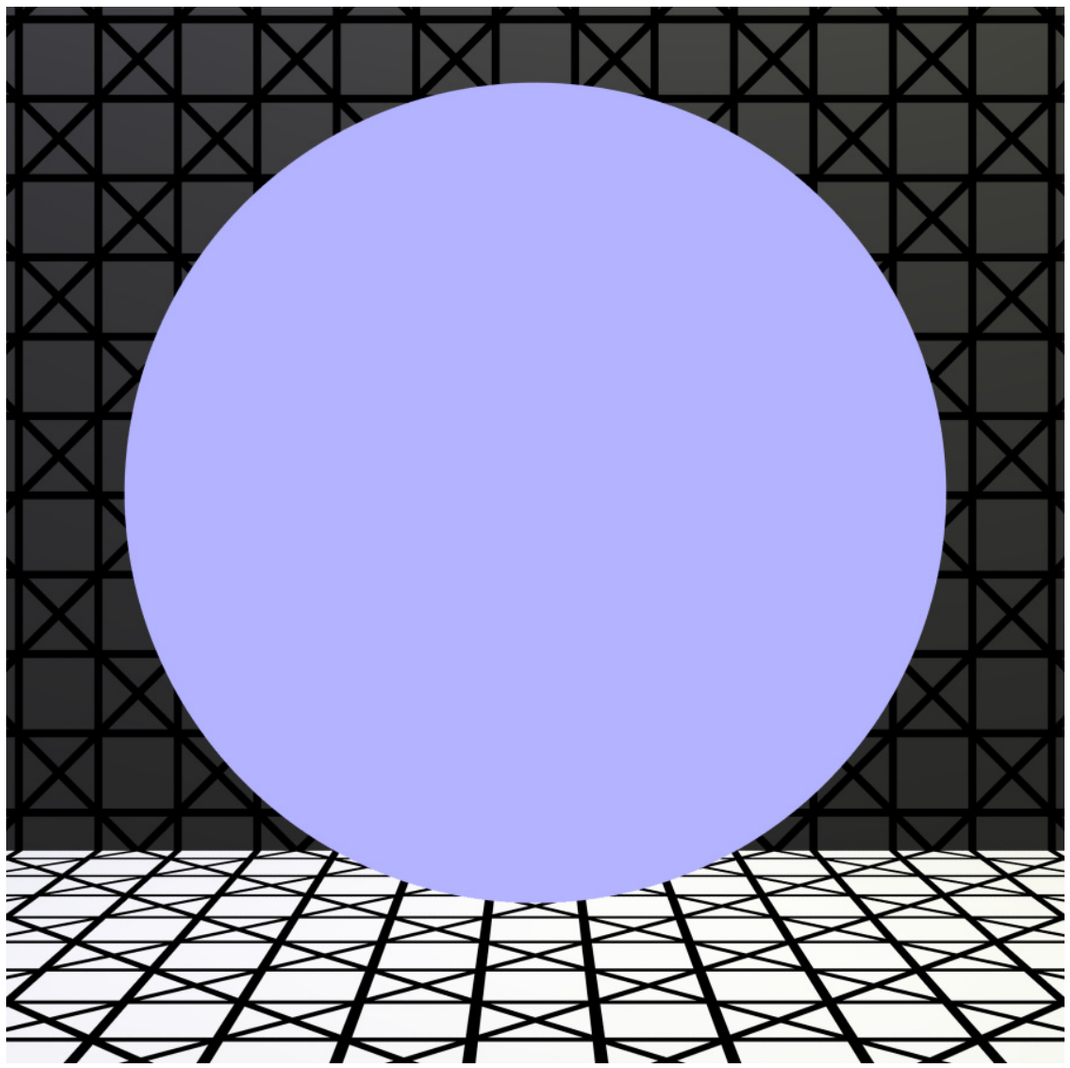
Using a threshold value in the volume

```
declare shader
    color "threshold_volume" (
        color  "color" default 1 1 1 1,
        vector "center" default 0 0 0,
        scalar "radius" default 1,
        scalar "density_threshold" default 0,
        scalar "unit_density" default 1,
        scalar "march_increment" default 0.1 )
end declare
```

Scene file declaration of shader "threshold_volume"

Volumetric effects

Using a threshold value in the volume

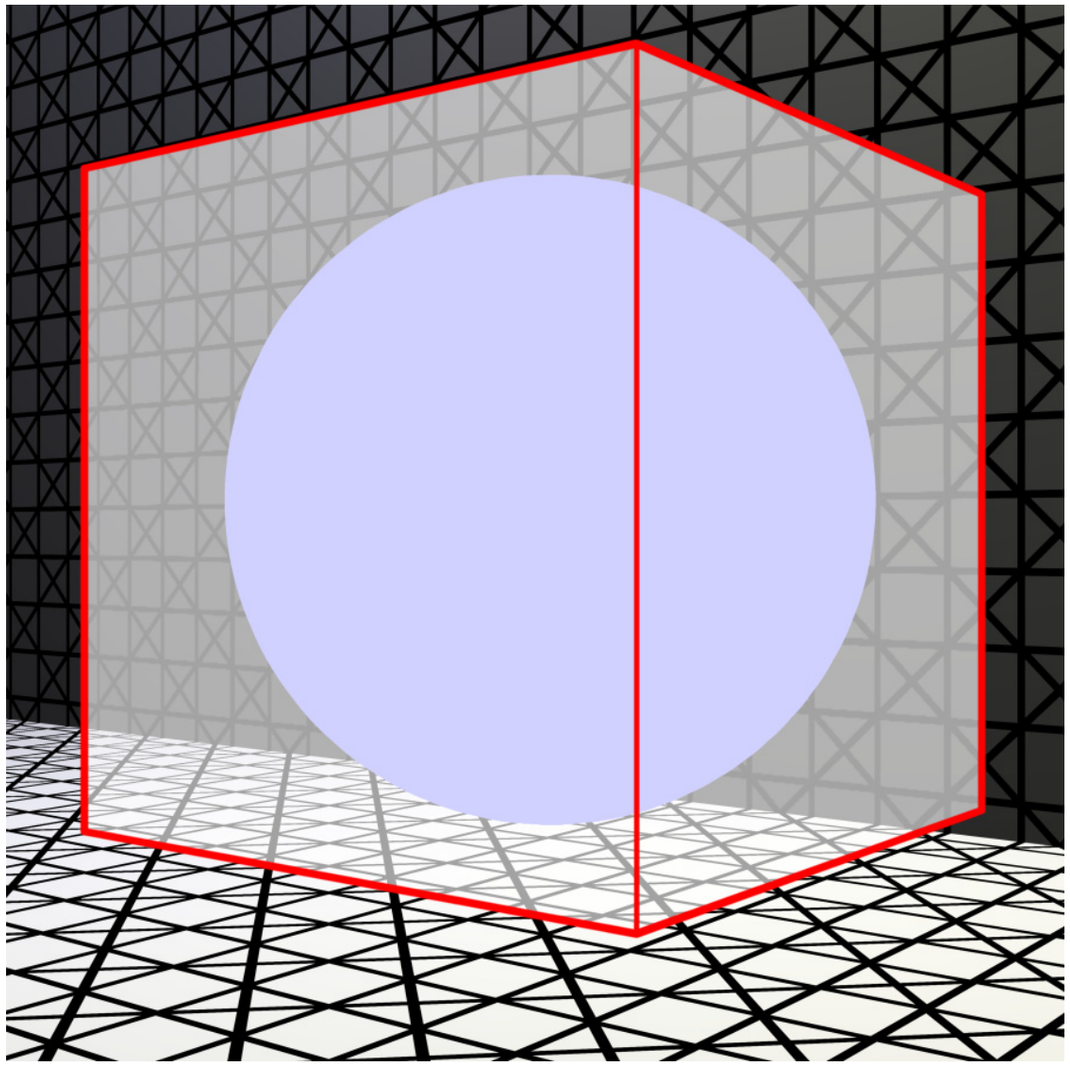


```
material "sphere_volume"
    volume "threshold_volume" (
        "color" .7 .7 1,
        "radius" 1,
        "unit_density" 1,
        "density_threshold" 0,
        "march_increment" .1 )
end material
```

A sphere defined by volume shader threshold_volume

Volumetric effects

Using a threshold value in the volume

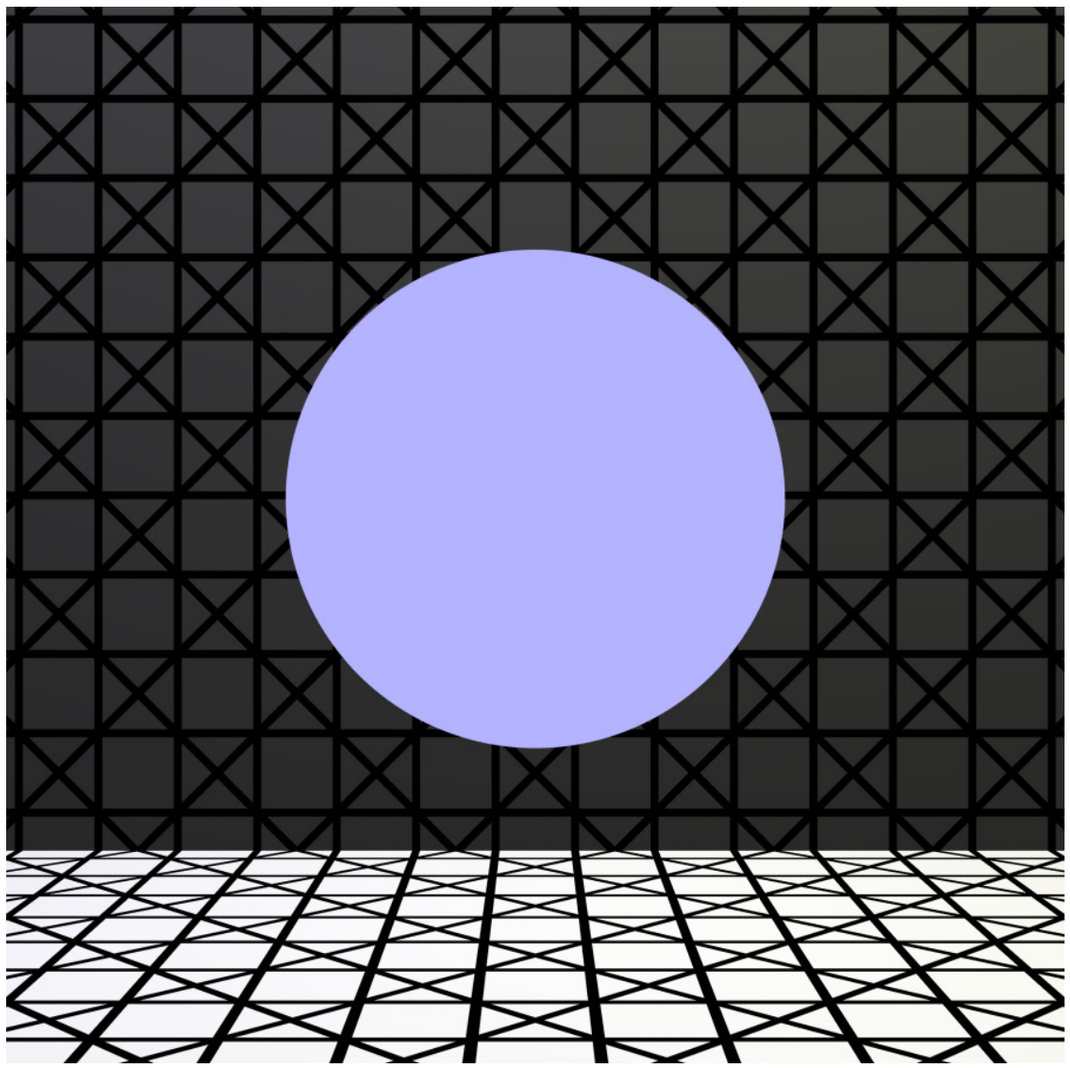


```
material "outline"
    "transparent" (
        "transparency" .6 .6 .6 )
    contour "c_contour" (
        "width" .5,
        "color" 1 0 0 1 )
end material

material "sphere_volume"
    volume "threshold_volume" (
        "color" .7 .7 1,
        "radius" 1,
        "unit_density" 1,
        "density_threshold" 0,
        "march_increment" .1 )
end material
```

The hull object that contains the ray march

Volumetric effects

Using a threshold value in the volume



```
material "sphere_volume"
    volume "threshold_volume" (
        "color" .7 .7 1,
        "radius" 1,
        "unit_density" 1,
        "density_threshold" 1.6,
        "march_increment" .1 )
end material
```

A sphere defined by shader threshold_volume with a smaller density threshold

Volumetric effects

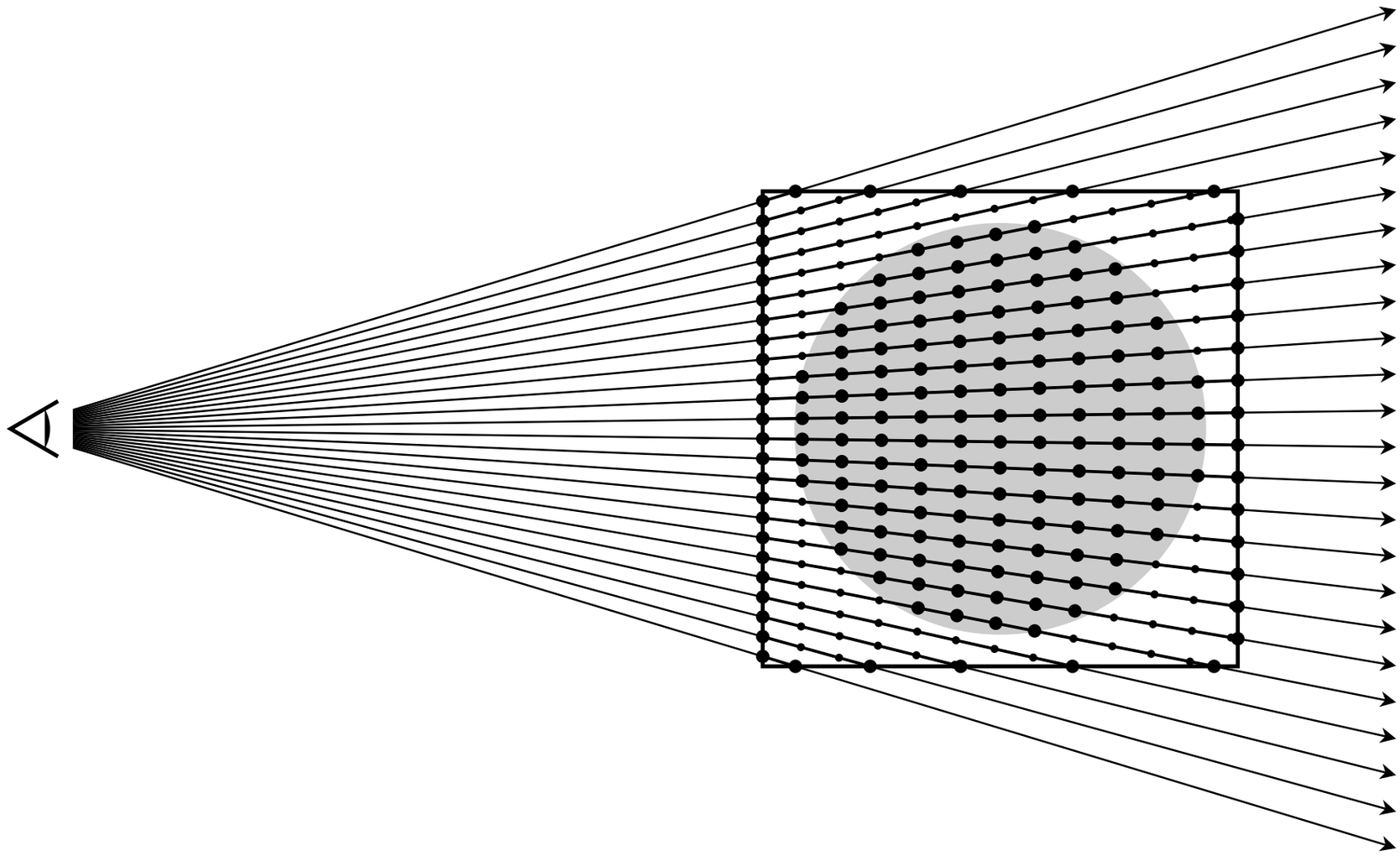
Using a threshold value in the volume

```
 1  struct threshold_volume {
 2      miColor color;
 3      miVector center;
 4      miScalar radius;
 5      miScalar density_threshold;
 6      miScalar unit_density;
 7      miScalar march_increment;
 8  };
 9
10  miBoolean threshold_volume (
11      miColor *result, miState *state, struct threshold_volume *params )
12  {
13      miScalar march_increment = *mi_eval_scalar(&params->march_increment);
14      miColor *color = mi_eval_color(&params->color);
15      miVector *center = mi_eval_vector(&params->center);
16      miScalar radius = *mi_eval_scalar(&params->radius);
17      miScalar unit_density = *mi_eval_scalar(&params->unit_density);
18      miScalar density_threshold = *mi_eval_scalar(&params->density_threshold);
19      miScalar distance, accumulated_density = 0.0;
20      miVector march_point, internal_center;
21      mi_point_from_object(state, &internal_center, center);
22
23      for (distance = 0; distance < state->dist; distance += march_increment) {
24          miaux_march_point(&march_point, state, distance);
25          accumulated_density +=
26              miaux_threshold_density(&march_point, &internal_center, radius,
27                                      unit_density, march_increment);
28          if (accumulated_density > density_threshold) {
29              miaux_copy_color(result, color);
30              break;
31          }
32      }
33      return miTRUE;
34  }
```

Source code of shader "threshold_volume"

Volumetric effects

Cumulative density for a ray in the volume



A spherical volume function with its enclosed sample points

Volumetric effects

Cumulative density for a ray in the volume

```
1  miScalar miaux_density_falloff(
2      miVector *point, miVector *center, miScalar radius,
3      miScalar unit_density, miScalar march_increment)
4  {
5      return march_increment * unit_density *
6          miaux_fit_clamp(mi_vector_dist(center, point), 0.0, radius, 1.0, 0.0);
7  }
```

Auxiliary function: miaux_density_falloff

Volumetric effects

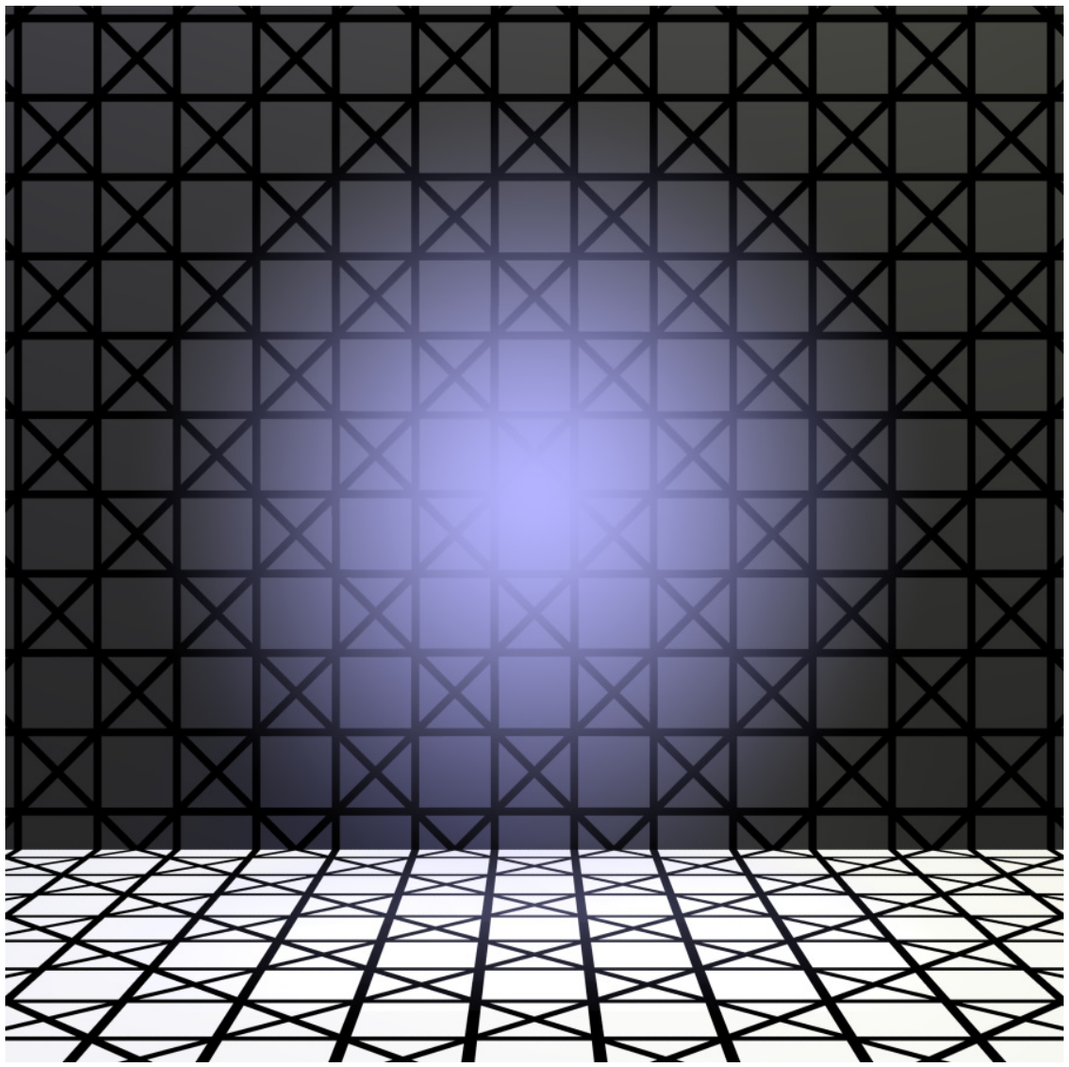
Cumulative density for a ray in the volume

```
declare shader
    color "density_volume" (
        color  "color" default 1 1 1 1,
        vector "center" default 0 0 0,
        scalar "radius" default 1,
        scalar "unit_density" default 1,
        scalar "march_increment" default 0.1 )
end declare
```

Scene file declaration of shader "density_volume"

Volumetric effects

Cumulative density for a ray in the volume



```
material "sphere_volume"
    volume "density_volume" (
        "color" .7 .7 1,
        "radius" 1,
        "unit_density" 1,
        "march_increment" .05 )
end material
```

A sphere defined by volume shader density_volume

Volumetric effects

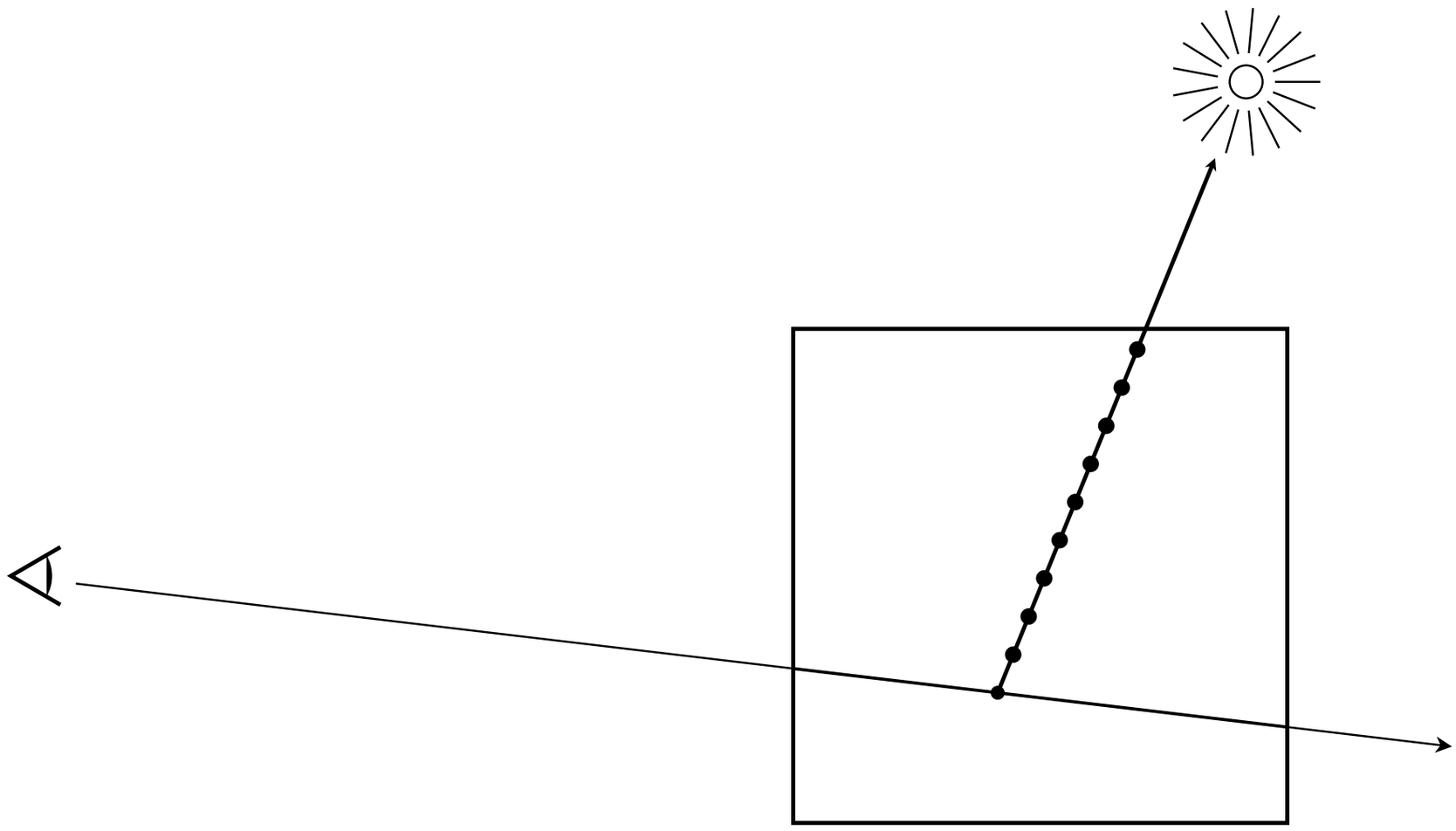
Cumulative density for a ray in the volume

```
 1  struct density_volume {
 2      miColor color;
 3      miVector center;
 4      miScalar radius;
 5      miScalar unit_density;
 6      miScalar march_increment;
 7  };
 8
 9  miBoolean density_volume (
10      miColor *result, miState *state, struct density_volume *params )
11  {
12      miScalar march_increment = *mi_eval_scalar(&params->march_increment);
13      miColor *color = mi_eval_color(&params->color);
14      miVector *center = mi_eval_vector(&params->center);
15      miScalar radius = *mi_eval_scalar(&params->radius);
16      miScalar unit_density = *mi_eval_scalar(&params->unit_density);
17      miScalar distance, accumulated_density = 0.0;
18      miVector march_point, internal_center;
19      mi_point_from_object(state, &internal_center, center);
20
21      for (distance = 0; distance <= state->dist; distance += march_increment) {
22          miaux_march_point(&march_point, state, distance);
23          accumulated_density +=
24              miaux_density_falloff(&march_point, &internal_center, radius,
25                                    unit_density, march_increment);
26          if (accumulated_density > 1.0) {
27              accumulated_density = 1.0;
28              break;
29          }
30      }
31      if (accumulated_density > 0.0)
32          miaux_blend_colors(result, result, color, 1.0 - accumulated_density);
33
34      return miTRUE;
35  }
```

Source code of shader "density_volume"

Volumetric effects

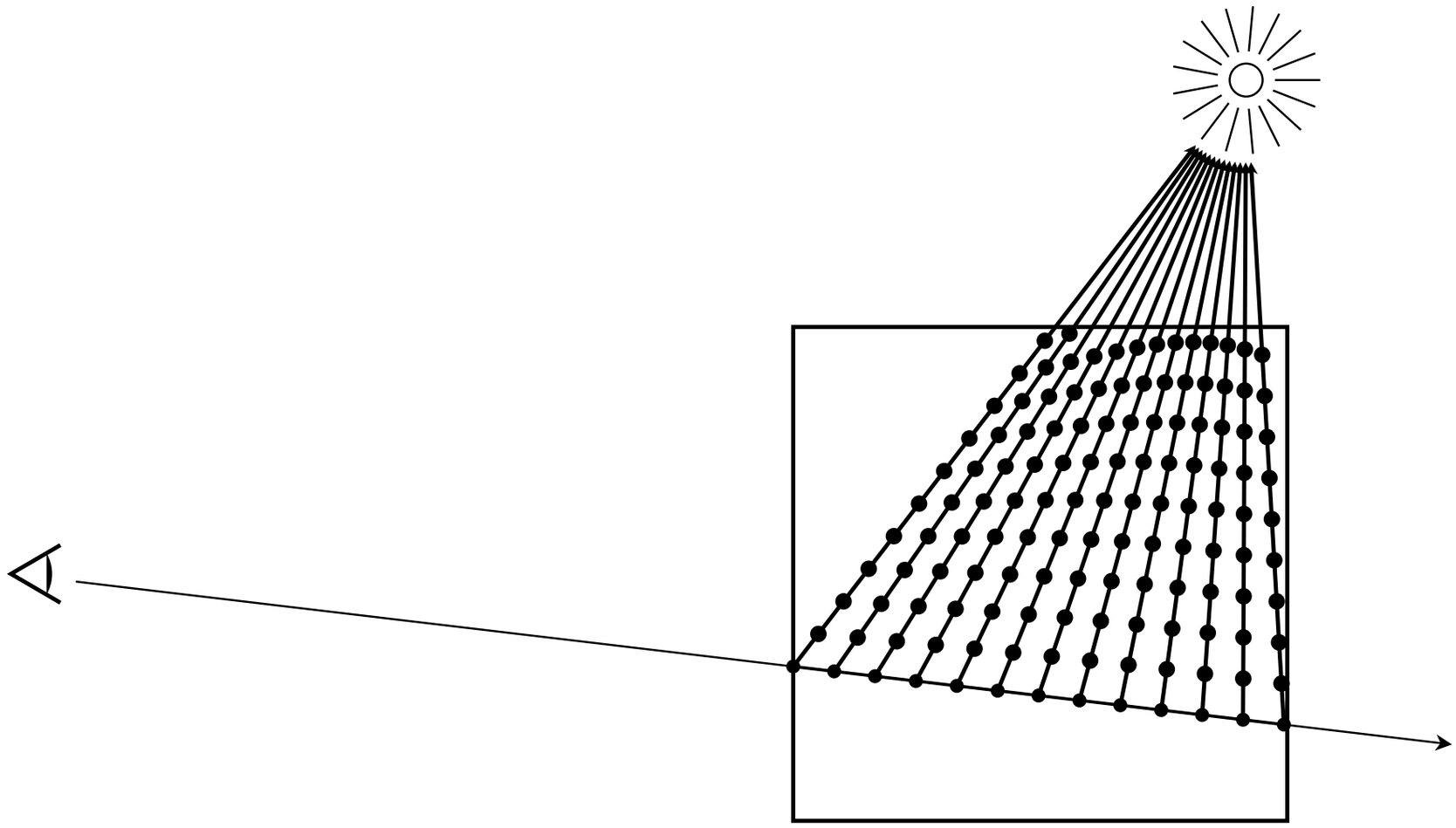
Illumination of samples in the volume



Adding up the density between a sample point and a light

Volumetric effects

Illumination of samples in the volume



Calculating the illumination values for all the samples along a ray

Volumetric effects

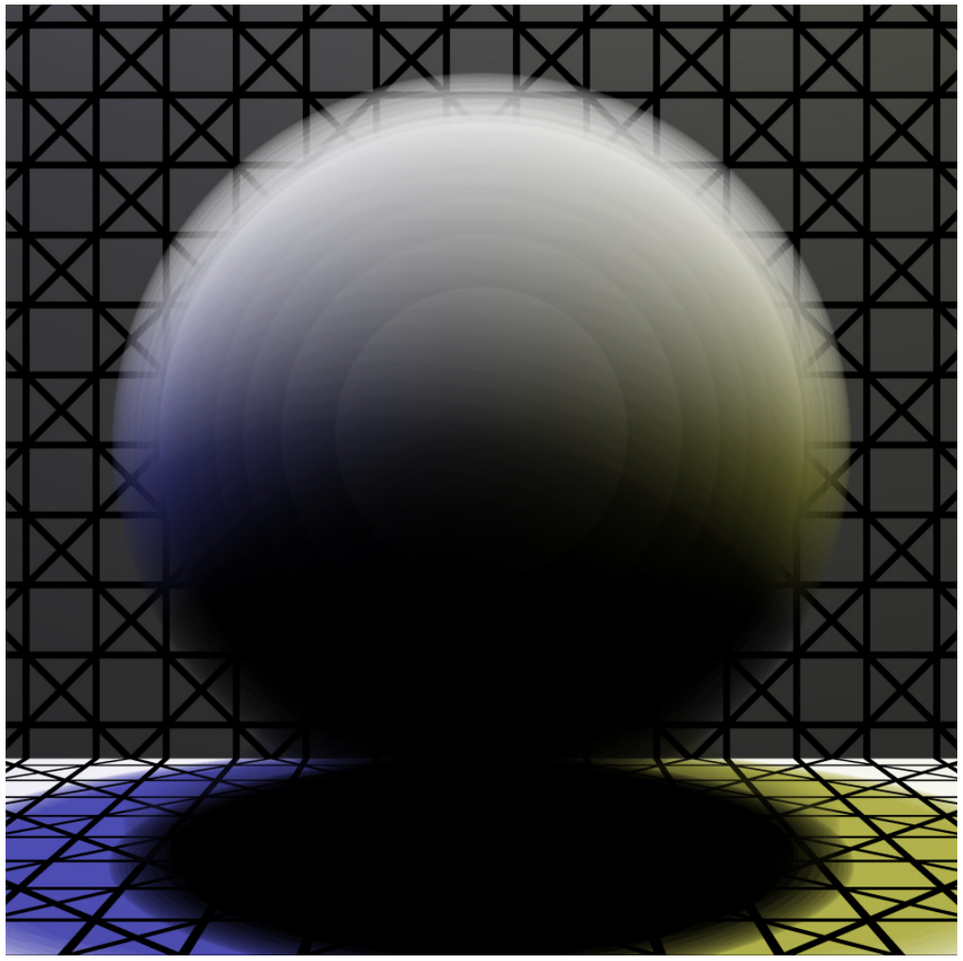
Illumination of samples in the volume

```
declare shader
    color "illuminated_volume" (
        color  "color" default 1 1 1 1,
        vector "center" default 0 0 0,
        scalar "radius" default 1,
        scalar "unit_density" default 1,
        scalar "march_increment" default 0.1,
        array light "lights" )
end declare
```

Scene file declaration of shader "illuminated_volume"

Volumetric effects

Illumination of samples in the volume



```
options "opt"
    object space
    contrast .1 .1 .1 1
    samples 0 2
    volume on
    shadow segments
end options

light "light1"
    "point_light_shadow" (
        "light_color" .7 .7 .3 )
    origin 1 5 0
end light

instance "light1-i" "light1" end instance

 light "light2"
    "point_light_shadow" (
        "light_color" .3 .3 .7 )
    origin -1 5 0
end light

instance "light2-i" "light2" end instance

material "sphere_volume"
    volume "illuminated_volume" (
        "color" 1 1 1,
        "radius" 1,
        "unit_density" 1,
        "march_increment" .05,
        "lights" ["light1-i", "light2-i"] )
end material
```

Lighting with shadows on a sphere volume

```
 1  miScalar miaux_fractional_occlusion_at_point(
 2      miVector *start_point, miVector *direction,
 3      miScalar total_distance, miVector *center, miScalar radius,
 4      miScalar unit_density, miScalar march_increment)
 5  {
 6      miScalar distance, occlusion = 0.0;
 7      miVector march_point;
 8      mi_vector_normalize(direction);
 9      for (distance = 0; distance <= total_distance; distance += march_increment) {
10          miaux_point_along_vector(&march_point, start_point, direction, distance);
11          occlusion += miaux_threshold_density(&march_point, center, radius,
12                                               unit_density, march_increment);
13          if (occlusion >= 1.0) {
14              occlusion = 1.0;
15              break;
16          }
17      }
18      return occlusion;
19  }
```

Auxiliary function: miaux_fractional_occlusion_at_point

Volumetric effects

Illumination of samples in the volume

```
 1  void miaux_total_light_at_point(
 2      miColor *result, miVector *point, miState *state,
 3      miTag* light, int light_count)
 4  {
 5      miColor sum, light_color;
 6      int i, light_sample_count;
 7      miVector original_point = state->point;
 8      state->point = *point;
 9
10      miaux_set_channels(result, 0.0);
11      for (i = 0; i < light_count; i++, light++) {
12          miVector direction_to_light;
13          light_sample_count = 0;
14          miaux_set_channels(&sum, 0.0);
15          while (mi_sample_light(&light_color, &direction_to_light, NULL,
16                                 state, *light, &light_sample_count))
17              miaux_add_scaled_color(&sum, &light_color, 1.0);
18
19          if (light_sample_count)
20              miaux_add_scaled_color(result, &sum, 1/light_sample_count);
21      }
22      state->point = original_point;
23  }
```

Auxiliary function: miaux_total_light_at_point

```
1  void miaux_add_transparent_color(
2      miColor *result, miColor *color, miScalar transparency)
3  {
4      miScalar new_alpha = result->a + transparency;
5      if (new_alpha > 1.0)
6          transparency = 1.0 - result->a;
7      result->r += color->r * transparency;
8      result->g += color->g * transparency;
9      result->b += color->b * transparency;
10     result->a += transparency;
11 }
```

Auxiliary function: miaux_add_transparent_color

```
1  void miaux_alpha_blend_colors(
2      miColor *result, miColor *foreground, miColor *background)
3  {
4      double bg_fraction = 1.0 - foreground->a;
5      result->r = foreground->r + background->r * bg_fraction;
6      result->g = foreground->g + background->g * bg_fraction;
7      result->b = foreground->b + background->b * bg_fraction;
8  }
```

Auxiliary function: miaux_alpha_blend_colors

Volumetric effects

Illumination of samples in the volume

```
 1  struct illuminated_volume {
 2      miColor color;
 3      miVector center;
 4      miScalar radius;
 5      miScalar unit_density;
 6      miScalar march_increment;
 7      int     i_light;
 8      int     n_light;
 9      miTag   light[1];
10  };
11
12  miBoolean illuminated_volume (
13      miColor *result, miState *state, struct illuminated_volume *params )
14  {
15      miScalar radius, unit_density, march_increment, density, distance;
16      miVector *center, internal_center, march_point;
17      int light_count;
18      miTag *light;
19
20      if (state->type == miRAY_LIGHT)
21          return miTRUE;
22
23      center = mi_eval_vector(&params->center);
24      radius = *mi_eval_scalar(&params->radius);
25      unit_density = *mi_eval_scalar(&params->unit_density);
26      march_increment = *mi_eval_scalar(&params->march_increment);
27      mi_point_from_object(state, &internal_center, center);
28
29      if (state->type == miRAY_SHADOW) {
30          miScalar occlusion = miaux_fractional_occlusion_at_point (
31              &state->org, &state->dir, state->dist,
32              &internal_center, radius, unit_density, march_increment);
33          miaux_scale_color(result, 1.0 - occlusion);
34      } else {
35          miColor *color = mi_eval_color(&params->color);
36          miColor volume_color = {0,0,0,0}, light_color, point_color;
37          void* original_state_pri = state->pri;
38          state->pri = NULL;
39          miaux_light_array(&light, &light_count, state,
40                            &params->i_light, &params->n_light, params->light);
41
42          for (distance = 0; distance <= state->dist; distance += march_increment) {
43              miaux_march_point(&march_point, state, distance);
44              density = miaux_threshold_density(
45                  &march_point, &internal_center, radius,
46                  unit_density, march_increment);
47              if (density > 0.0) {
48                  miaux_total_light_at_point(
49                      &light_color, &march_point, state, light, light_count);
50                  miaux_multiply_colors(&point_color, color, &light_color);
51                  miaux_add_transparent_color(&volume_color, &point_color, density);
52              }
53              if (volume_color.a == 1.0)
54                  break;
55          }
56          miaux_alpha_blend_colors(result, &volume_color, result);
57          state->pri = original_state_pri;
58      }
59      return miTRUE;
60  }
```

Source code of shader "illuminated_volume"

Defining the density function with a shader

```
declare shader
    scalar "spherical_density" (
        vector "center" default 0 0 0,
        scalar "radius" default 1 )
end declare
```

Scene file declaration of shader "spherical_density"

Defining the density function with a shader

```
 1  struct spherical_density {
 2      miVector center;
 3      miScalar radius;
 4  };
 5
 6  miBoolean spherical_density (
 7      miScalar *result, miState *state, struct spherical_density *params )
 8  {
 9      miVector *center = mi_eval_vector(&params->center);
10      miScalar radius = *mi_eval_scalar(&params->radius);
11      miVector point;
12      mi_vector_to_world(state, &point, &state->point);
13
14      if (mi_vector_dist(center, &point) <= radius)
15          *result = 1.0;
16      else
17          *result = 0.0;
18
19      return miTRUE;
20  }
```

Source code of shader "spherical_density"

Volumetric effects

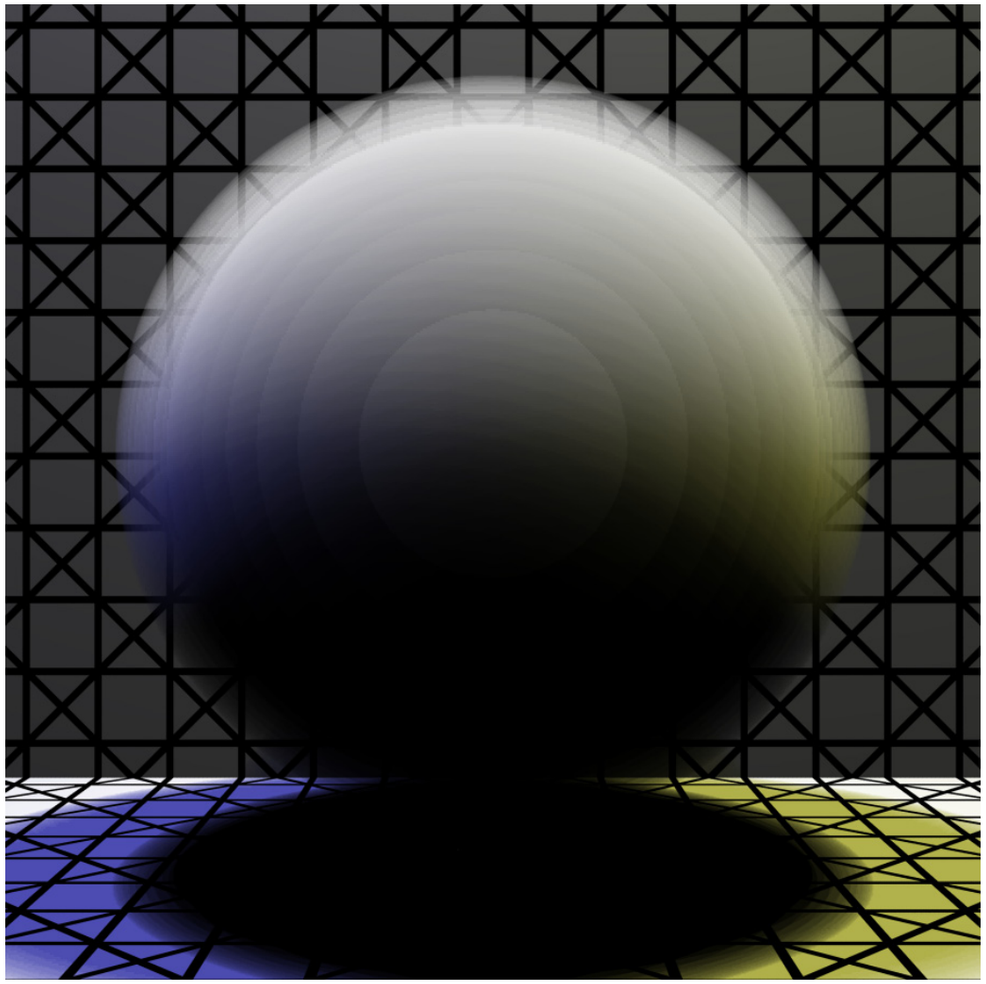
Defining the density function with a shader

```
declare shader
    color "parameter_volume" (
        color "color" default 1 1 1,
        shader "density_shader",
        scalar "unit_density" default 1,
        scalar "march_increment" default 0.1,
        array light "lights" )
end declare
```

Scene file declaration of shader "parameter_volume"

Volumetric effects

Defining the density function with a shader



```
shader "sphere"
    "spherical_density" (
        "radius" 1,
        "center" 0 .1 0 )

material "sphere_volume"
    volume "parameter_volume" (
        "density_shader" "sphere",
        "unit_density" 1,
        "march_increment" .05,
        "lights" ["light1-i", "light2-i"] )
end material
```

Defining density with shader spherical_volume

Defining the density function with a shader

```
 1  miScalar miaux_fractional_shader_occlusion_at_point(
 2      miState *state, miVector *start_point, miVector *direction,
 3      miScalar total_distance, miTag density_shader,
 4      miScalar unit_density, miScalar march_increment)
 5  {
 6      miScalar density, distance, occlusion = 0.0;
 7      miVector march_point;
 8      miVector original_point = state->point;
 9      mi_vector_normalize(direction);
10      for (distance = 0; distance <= total_distance; distance += march_increment) {
11          miaux_point_along_vector(&march_point, start_point, direction, distance);
12          state->point = march_point;
13          mi_call_shader_x((miColor*)&density, miSHADER_MATERIAL, state,
14                           density_shader, NULL);
15          occlusion += density * unit_density * march_increment;
16          if (occlusion >= 1.0) {
17              occlusion = 1.0;
18              break;
19          }
20      }
21      state->point = original_point;
22      return occlusion;
23  }
```

Auxiliary function: miaux_fractional_shader_occlusion_at_point

Volumetric effects

Defining the density function with a shader

```
declare shader
    color "parameter_volume" (
        color "color" default 1 1 1,
        shader "density_shader",
        scalar "unit_density" default 1,
        scalar "march_increment" default 0.1,
        array light "lights" )
end declare
```

Scene file declaration of shader "parameter_volume"

Defining the density function with a shader

```
1  struct parameter_volume {
2      miColor color;
3      miTag density_shader;
4      miScalar unit_density;
5      miScalar march_increment;
6      int      i_light;
7      int      n_light;
8      miTag    light[1];
9  };
```

Source code of struct "parameter_volume"

Volumetric effects

Defining the density function with a shader

```
 1  miBoolean parameter_volume (
 2      miColor *result, miState *state, struct parameter_volume *params )
 3  {
 4      miScalar unit_density, march_increment, density;
 5      miTag density_shader, *light;
 6      int light_count;
 7
 8      if (state->type == miRAY_LIGHT)
 9          return miTRUE;
10
11      density_shader = *mi_eval_tag(&params->density_shader);
12      unit_density = *mi_eval_scalar(&params->unit_density);
13      march_increment = *mi_eval_scalar(&params->march_increment);
14      miaux_light_array(&light, &light_count, state,
15                        &params->i_light, &params->n_light, params->light);
16
17      if (state->type == miRAY_SHADOW) {
18          miScalar occlusion = miaux_fractional_shader_occlusion_at_point (
19              state, &state->org, &state->dir, state->dist,
20              density_shader, unit_density, march_increment);
21          miaux_scale_color(result, 1.0 - occlusion);
22      } else {
23          miColor *color = mi_eval_color(&params->color);
24          miScalar distance;
25          miColor volume_color = {0,0,0,0}, light_color, point_color;
26          miVector original_point = state->point;
27          void* original_state_pri = state->pri;
28          state->pri = NULL;
29
30          for (distance = 0; distance <= state->dist; distance += march_increment) {
31              miVector march_point;
32              miaux_march_point(&march_point, state, distance);
33              state->point = march_point;
34              mi_call_shader_x((miColor*)&density,
35                               miSHADER_MATERIAL, state, density_shader, NULL);
36              if (density > 0) {
37                  density *= unit_density * march_increment;
38                  miaux_total_light_at_point(
39                      &light_color, &march_point, state, light, light_count);
40                  miaux_multiply_colors(&point_color, color, &light_color);
41                  miaux_add_transparent_color(&volume_color, &point_color, density);
42              }
43              if (volume_color.a == 1.0)
44                  break;
45          }
46          miaux_alpha_blend_colors(result, &volume_color, result);
47          state->point = original_point;
48          state->pri = original_state_pri;
49      }
50      return miTRUE;
51  }
```

Shader source of parameter_volume

Defining the density function with a shader

```
declare shader
    scalar "radial_falloff" (
        vector "center" default 0 0 0,
        scalar "radius" default 1,
        scalar "center_value" default 1,
        scalar "radius_value" default 0 )
end declare
```

Scene file declaration of shader "radial_falloff"

Volumetric effects

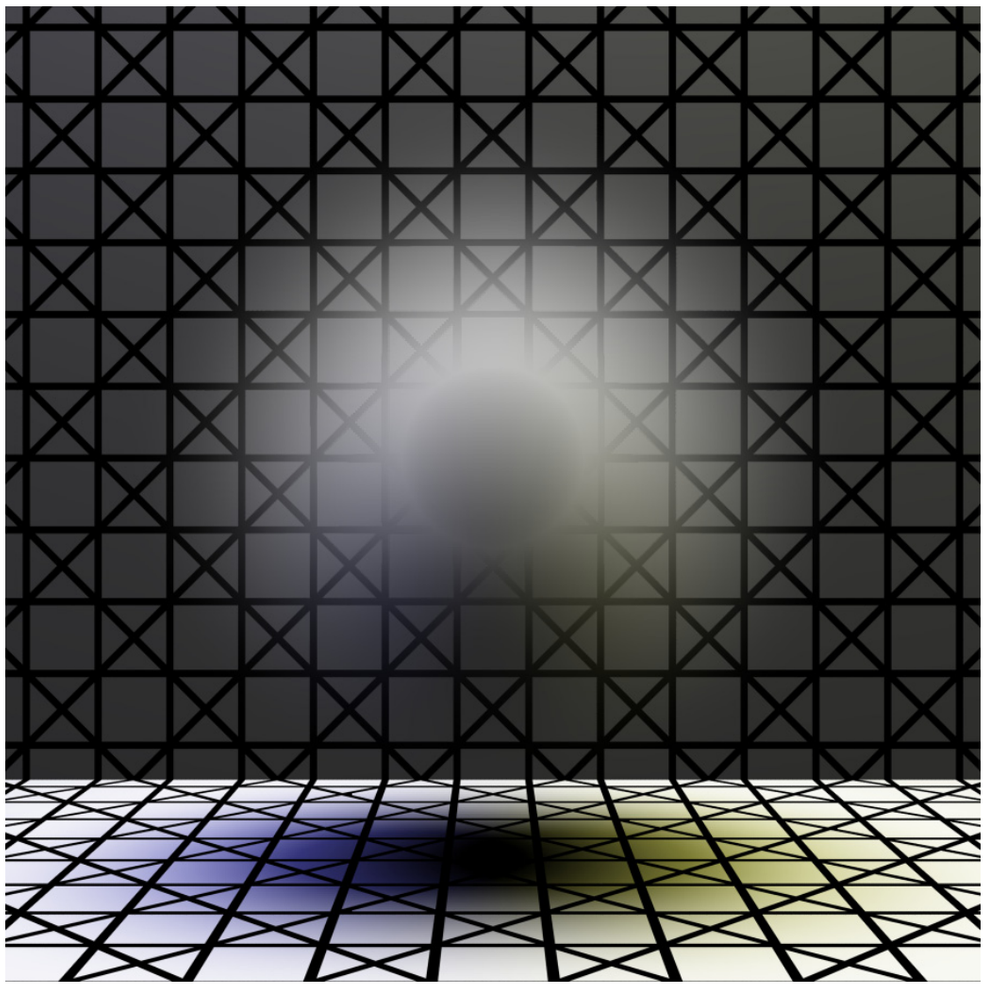
Defining the density function with a shader

```
 1  struct radial_falloff {
 2      miVector center;
 3      miScalar radius;
 4      miScalar center_value;
 5      miScalar radius_value;
 6  };
 7
 8  miBoolean radial_falloff (
 9      miScalar *result, miState *state, struct radial_falloff *params )
10  {
11      miVector *center = mi_eval_vector(&params->center);
12      miScalar radius = *mi_eval_scalar(&params->radius);
13      miScalar center_value = *mi_eval_scalar(&params->center_value);
14      miScalar radius_value = *mi_eval_scalar(&params->radius_value);
15      miScalar distance;
16      miVector point;
17
18      mi_vector_to_world(state, &point, &state->point);
19      distance = mi_vector_dist(center, &point);
20      *result = miaux_sinusoid_fit_clamp(
21          distance, 0.0, radius, center_value, radius_value);
22
23      return miTRUE;
24  }
```

Source code of shader "radial_falloff"

Volumetric effects

Defining the density function with a shader



```
shader "sphere"
    "radial_falloff" (
        "radius" 1,
        "center" 0 .1 0,
        "center_value" 1.2 )

material "sphere_volume"
    volume "parameter_volume" (
        "density_shader" "sphere",
        "unit_density" 1,
        "march_increment" .05,
        "lights" ["light1-i", "light2-i"] )
end material
```

Defining density with shader radial_falloff

Volumetric effects

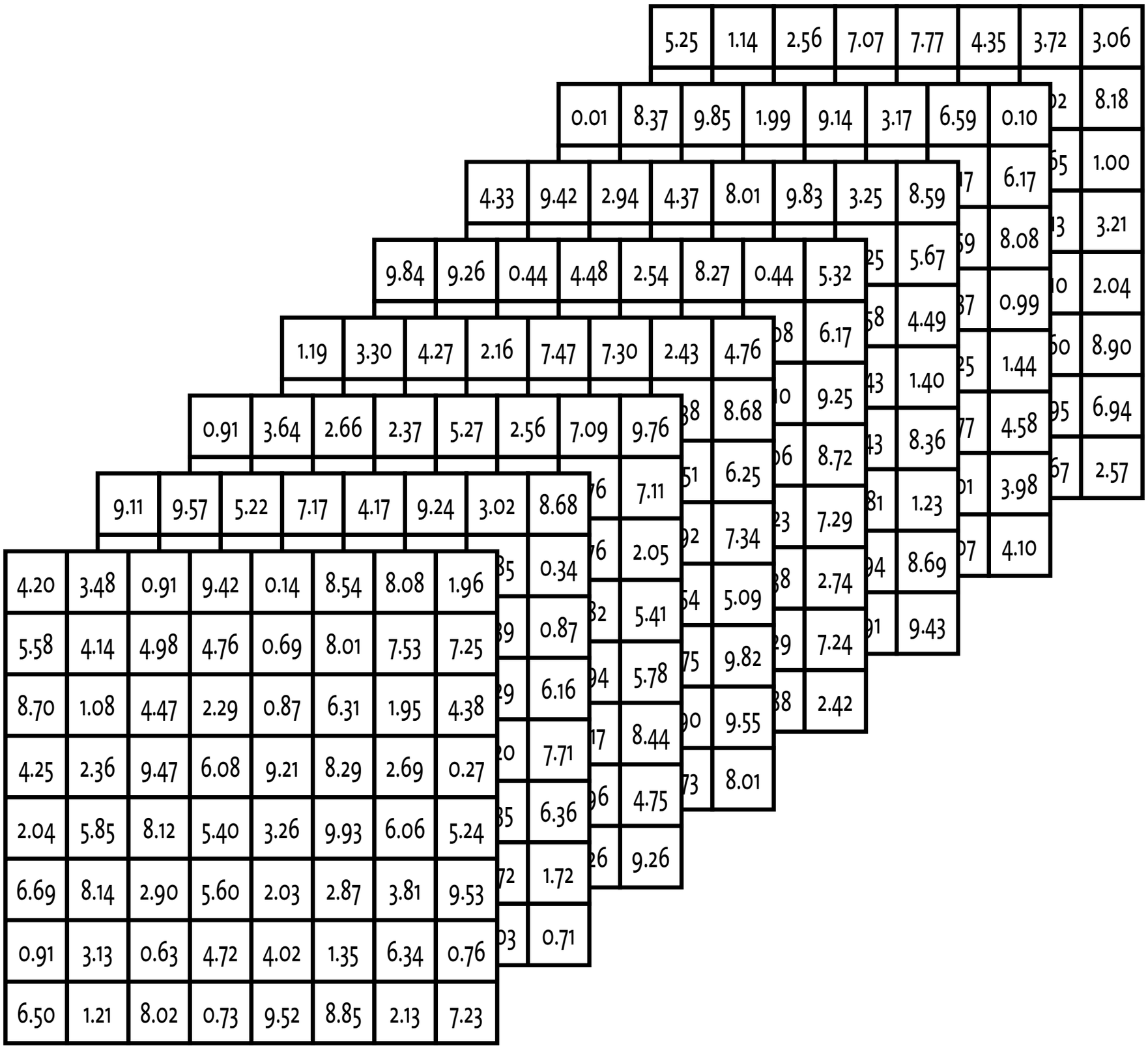
Using voxel data sets for the density function

4.20	3.48	0.91	9.42	0.14	8.54	8.08	1.96
5.58	4.14	4.98	4.76	0.69	8.01	7.53	7.25
8.70	1.08	4.47	2.29	0.87	6.31	1.95	4.38
4.25	2.36	9.47	6.08	9.21	8.29	2.69	0.27
2.04	5.85	8.12	5.40	3.26	9.93	6.06	5.24
6.69	8.14	2.90	5.60	2.03	2.87	3.81	9.53
0.91	3.13	0.63	4.72	4.02	1.35	6.34	0.76
6.50	1.21	8.02	0.73	9.52	8.85	2.13	7.23

Pixels as a plane of numbers

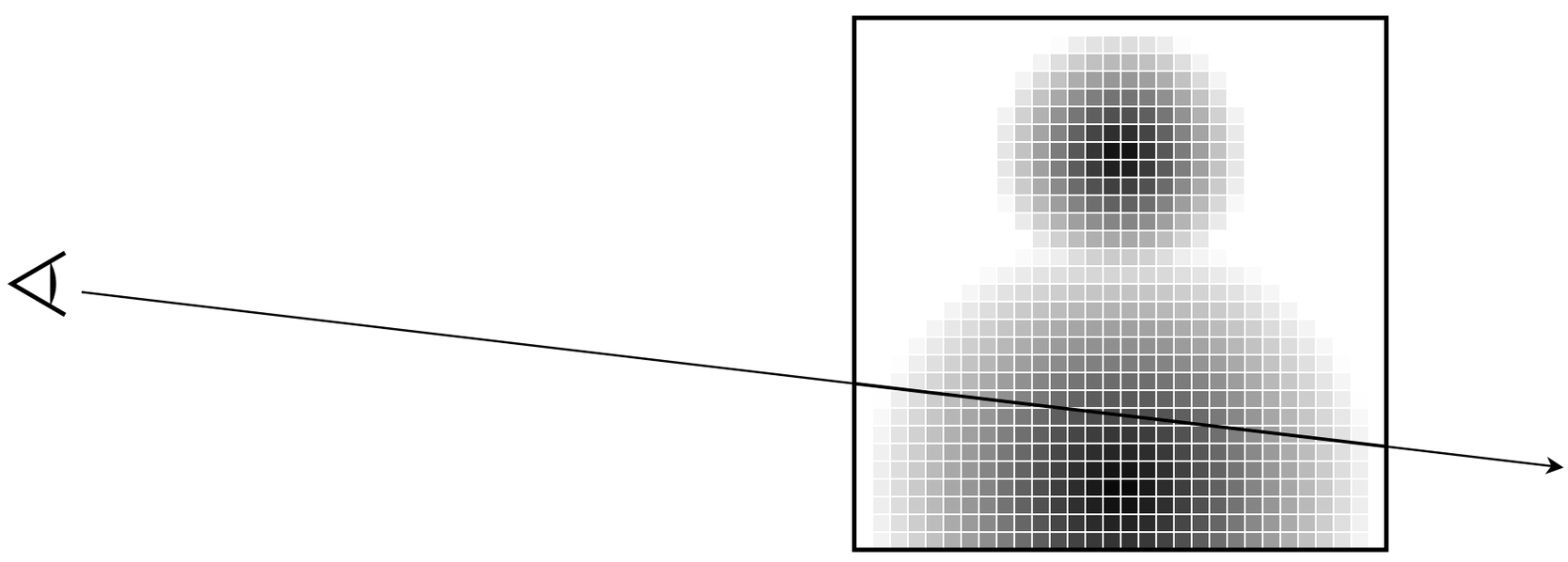
Volumetric effects

Using voxel data sets for the density function



Voxels as a set of planes of numbers

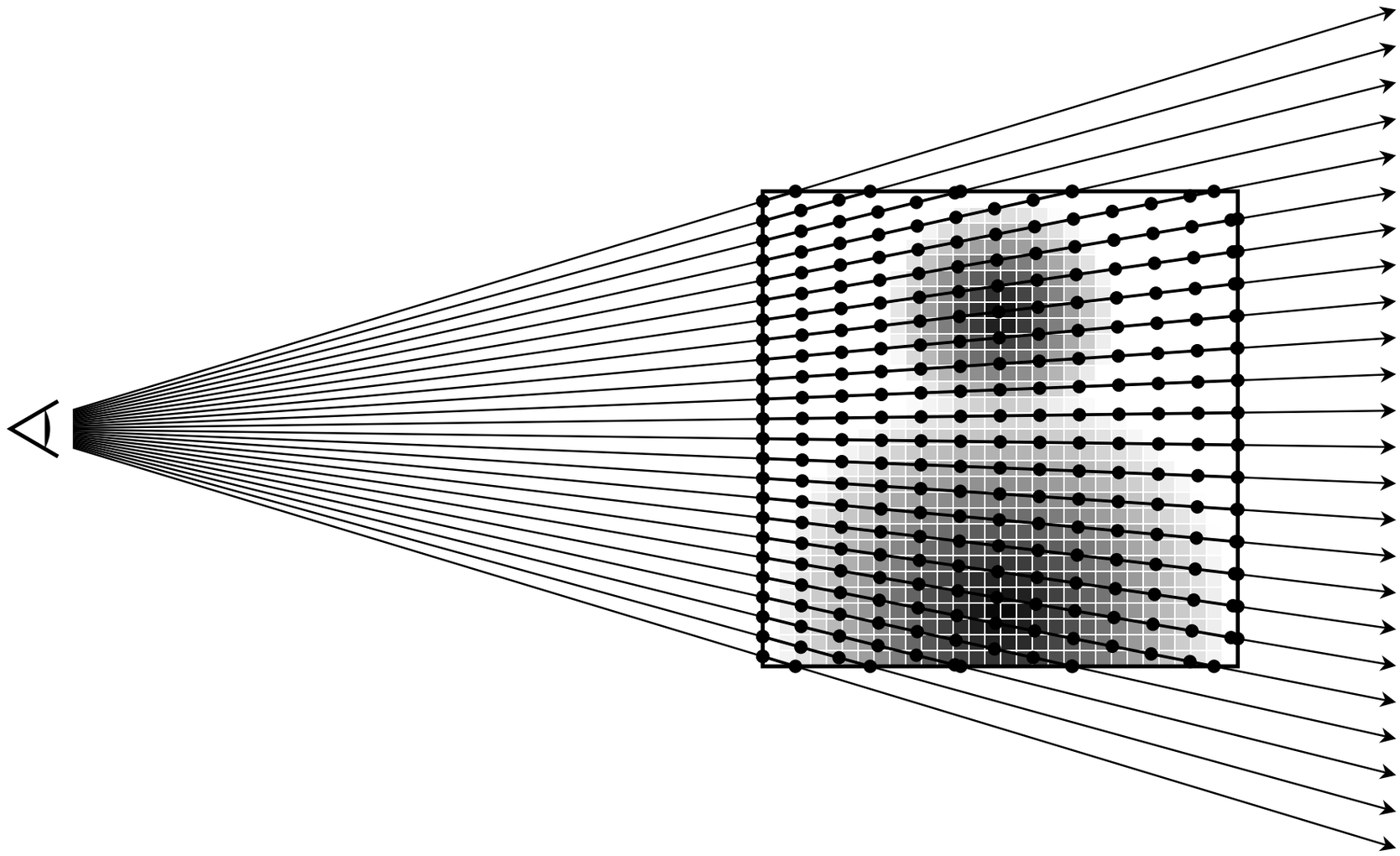
Using voxel data sets for the density function



A voxel data set as density measures in a volume

Volumetric effects

Using voxel data sets for the density function



Raymarching through a voxel data set

Using voxel data sets for the density function

```
1  void miaux_read_volume_block(
2      char* filename,
3      int *width, int *height, int *depth, float* block)
4  {
5      int count;
6      FILE* fp = fopen(filename, "r");
7      if (fp == NULL) {
8          mi_fatal("Error opening file "%s".", filename);
9      }
10     fscanf(fp, "%d %d %d ", width, height, depth);
11     count = (*width) * (*height) * (*depth);
12     mi_progress("Volume dataset: %dx%dx%d", *width, *height, *depth);
13     fread(block, sizeof(float), count, fp);
14 }
```

Auxiliary function: miaux_read_volume_block

Volumetric effects

Using voxel data sets for the density function

```
1  typedef struct {
2      int width, height, depth;
3      float block[MAX_DATASET_SIZE];
4  } voxel_data;
```

Struct for storage of voxel data in shader

Using voxel data sets for the density function

```
1  float voxel_value(voxel_data *voxels, float x, float y, float z)
2  {
3      return voxels->block[
4          ((int)(z + .5)) * voxels->depth * voxels->height +
5          ((int)(y + .5)) * voxels->height +
6          ((int)(x + .5)) ];
7  }
```

Source code of shader "voxel_density"

Using voxel data sets for the density function

```
declare shader
    scalar "voxel_density" (
        string "filename",
        vector "min_point" default -1 -1 -1,
        vector "max_point" default 1 1 1,
        color "color" default 1 1 1, )
end declare
```

Scene file declaration of shader "voxel_density"

Volumetric effects

Using voxel data sets for the density function

```
 1  miBoolean voxel_density_init(
 2      miState *state, struct voxel_density *params,
 3      miBoolean *instance_init_required)
 4  {
 5      if (!params) {  /* Main shader init (not an instance): */
 6          *instance_init_required = miTRUE;
 7      } else {        /* Instance initialization: */
 8          char* filename =
 9              miaux_tag_to_string(*mi_eval_tag(&params->filename_tag), NULL);
10          if (filename) {
11              voxel_data *voxels =
12                  miaux_user_memory_pointer(state, sizeof(voxel_data));
13              miaux_read_volume_block(
14                  filename, &voxels->width, &voxels->height, &voxels->depth,
15                  voxels->block);
16              mi_progress("Voxel dataset: %dx%dx%d",
17                          voxels->width, voxels->height, voxels->depth);
18          }
19      }
20      return miTRUE;
21  }
```

Source code of init shader of "voxel_density"

Using voxel data sets for the density function

```
1  miBoolean voxel_density_exit(miState *state, void *params)
2  {
3      return miaux_release_user_memory("voxel_density", state, params);
4  }
```

Source code of exit shader of "voxel_density"

```
1  miBoolean miaux_point_inside(miVector *p, miVector *min_p, miVector *max_p)
2  {
3      return p->x >= min_p->x && p->y >= min_p->y && p->z >= min_p->z &&
4          p->x <= max_p->x && p->y <= max_p->y && p->z <= max_p->z;
5  }
```

Auxiliary function: miaux_point_inside

Volumetric effects

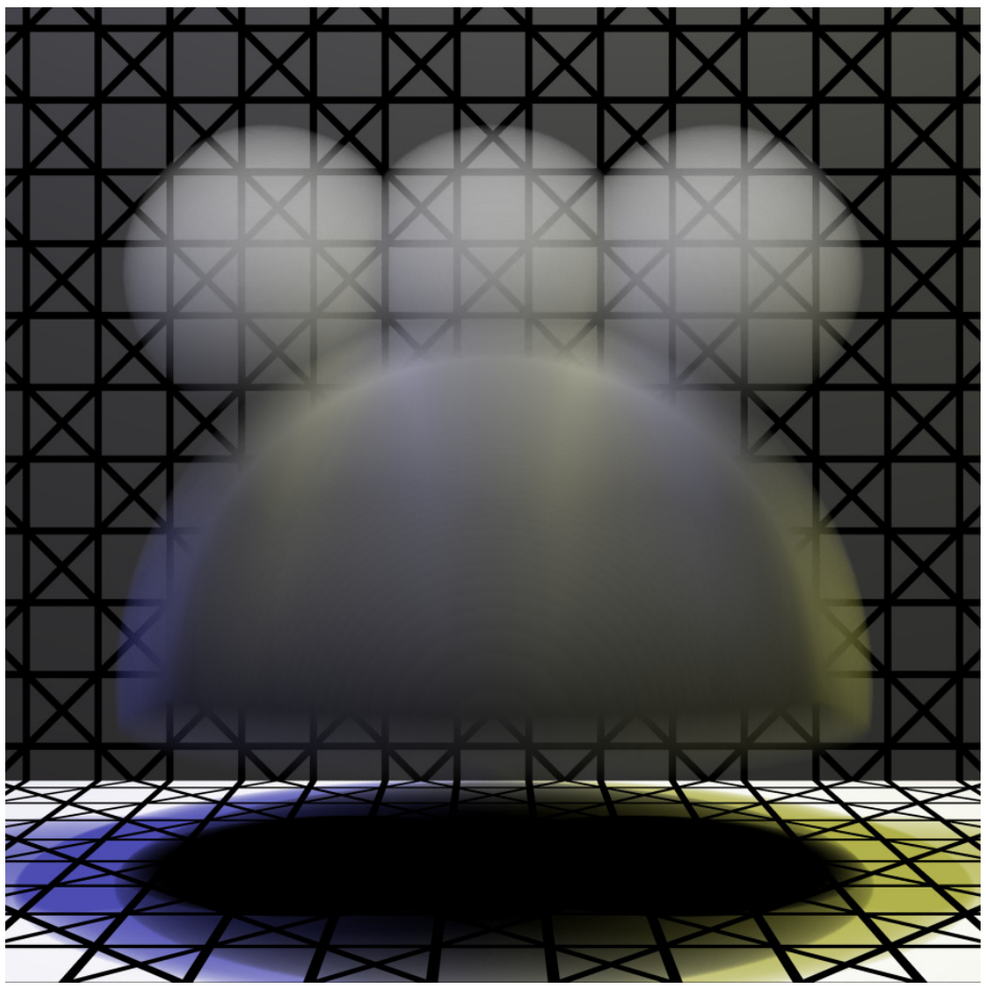
Using voxel data sets for the density function

```
 1  miBoolean voxel_density (
 2      miScalar *result, miState *state, struct voxel_density *params )
 3  {
 4      miVector *min_point = mi_eval_vector(&params->min_point);
 5      miVector *max_point = mi_eval_vector(&params->max_point);
 6      miVector *p = &state->point;
 7
 8      if (miaux_point_inside(p, min_point, max_point)) {
 9          float x, y, z;
10          voxel_data *voxels = miaux_user_memory_pointer(state, 0);
11          x = miaux_fit(p->x, min_point->x, max_point->x, 0, voxels->width-1);
12          y = miaux_fit(p->y, min_point->y, max_point->y, 0, voxels->height-1);
13          z = miaux_fit(p->z, min_point->z, max_point->z, 0, voxels->depth-1);
14          *result = voxel_value(voxels, x, y, z);
15      } else
16          *result = 0.0;
17
18      return miTRUE;
19  }
```

Source code of main shader of "voxel_density"

Volumetric effects

Using voxel data sets for the density function

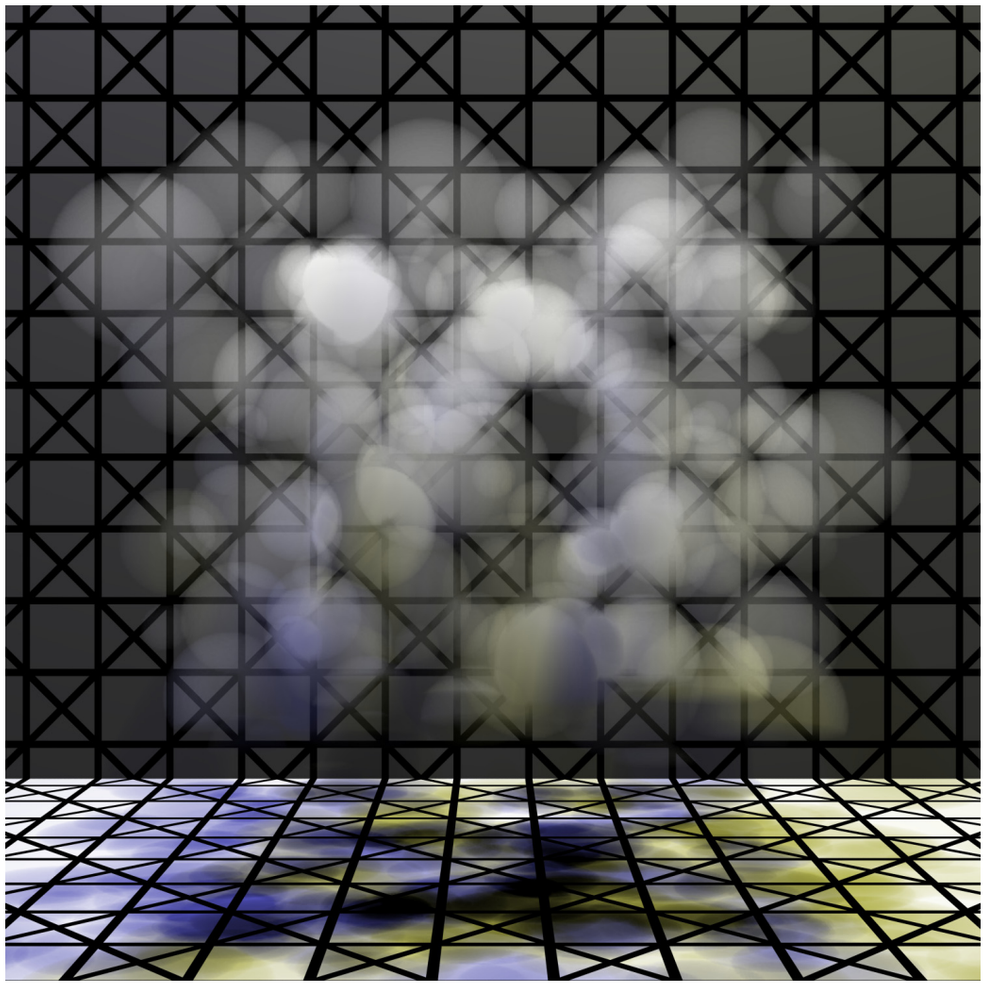


```
shader "spheres"
    "voxel_density" (
        "filename" "three_over_one.vol" )

material "sphere_volume"
    volume "parameter_volume" (
        "density_shader" "spheres",
        "unit_density" 1,
        "march_increment" .01,
        "lights" ["light1-i", "light2-i"] )
end material
```

Reading a voxel dataset with a shader

Volumetric effects

Using voxel data sets for the density function

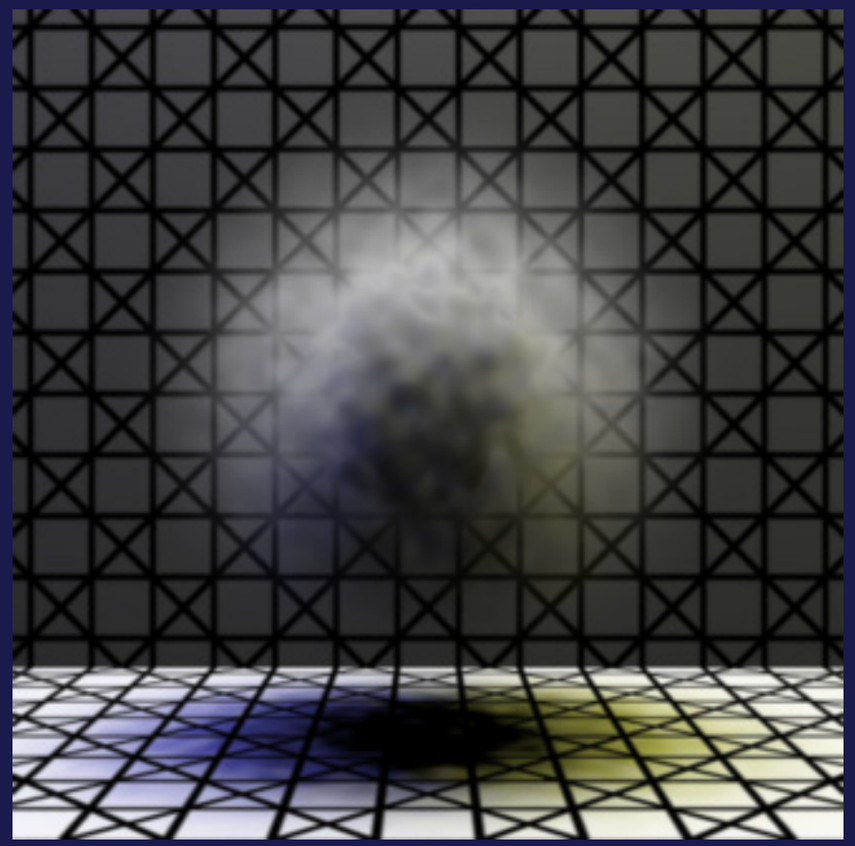


```
shader "spheres"
    "voxel_density" (
        "filename" "sphere_cloud.vol" )

material "sphere_volume"
    volume "parameter_volume" (
        "density_shader" "spheres",
        "unit_density" 1,
        "march_increment" .01,
        "lights" ["light1-i", "light2-i"] )
end material
```

Multiple spheres in a volume data set

Exercise 20: Volume shaders

1. Copy `volume_5A.mi` to `volume.mi`, change output file to `volume.tif`, render and view.
2. Change the `radius` parameter of shader `radial_falloff` to 0.5, render and view.
3. Modify shader `radial_falloff` to scale the density value by one of the noise functions in the API library.



Exercise 20: Volume shaders (part 2)

Modify shader `radial_density` to scale the density value by one of the noise functions in the API library.

Old:

```
mi_vector_to_world(state, &point, &state->point);
distance = mi_vector_dist(center, &point);
*result = miaux_sinusoid_fit_clamp(
    distance, 0.0, radius, center_value, radius_value);
```

New:

```
mi_vector_to_world(state, &point, &state->point);
distance = mi_vector_dist(center, &point);
point.x *= 20;
point.y *= 20;
point.z *= 20;
*result = mi_noise_3d(&point) *
    miaux_sinusoid_fit_clamp(
        distance, 0.0, radius, center_value, radius_value);
```