Writing mental ray shaders

by Andy Kopra

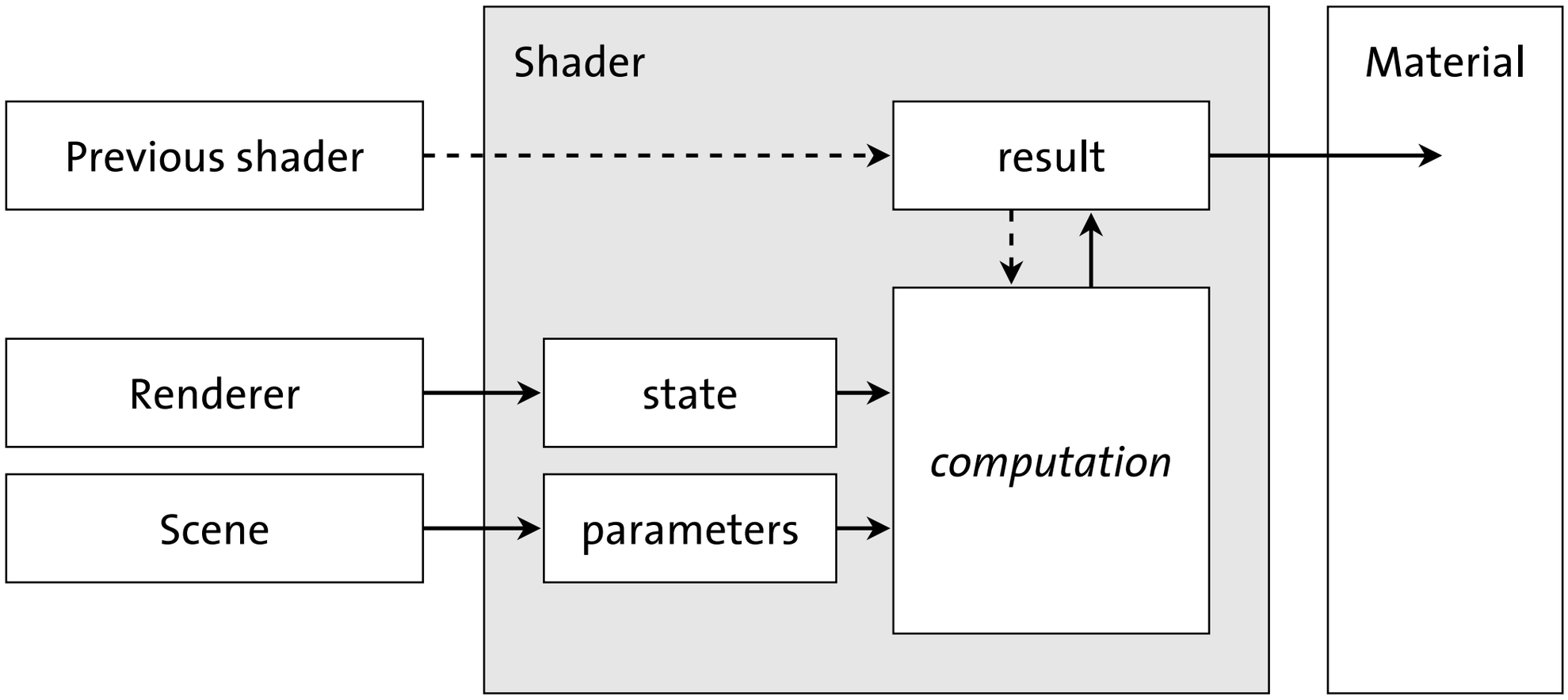
Part 2: Color

A single color

A single color

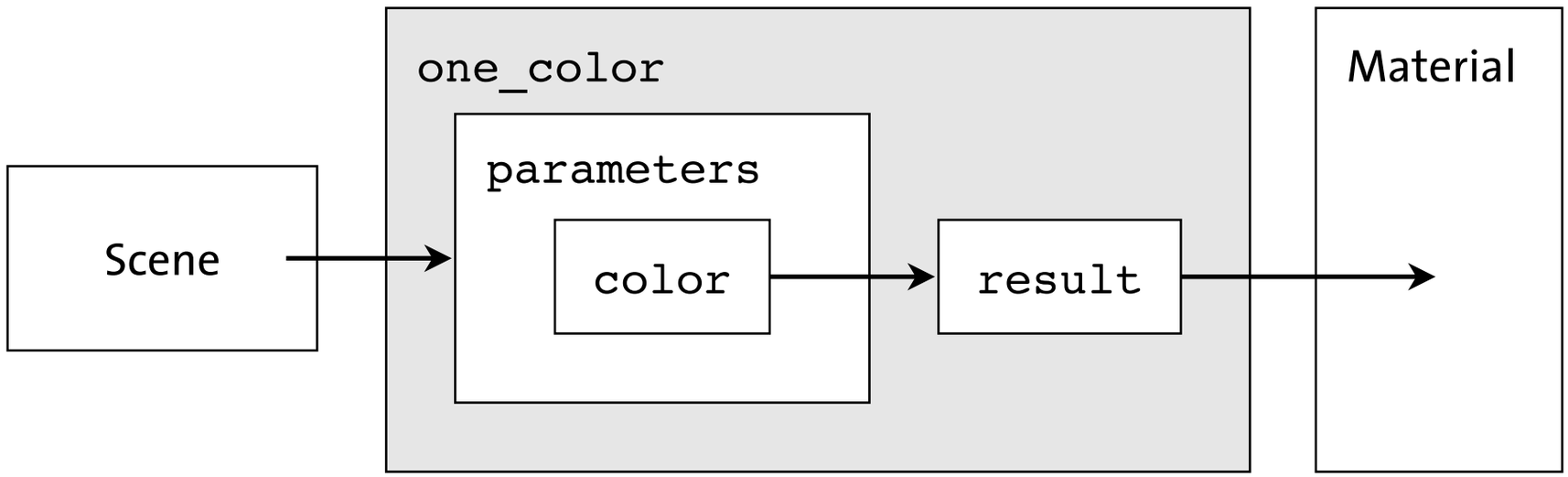
The simplest shader model
The shader function in C
The shader source file
Using a shader in a material
Shader programming style
Basic issues in writing shaders

A single color



A model for shaders as a process with inputs and outputs

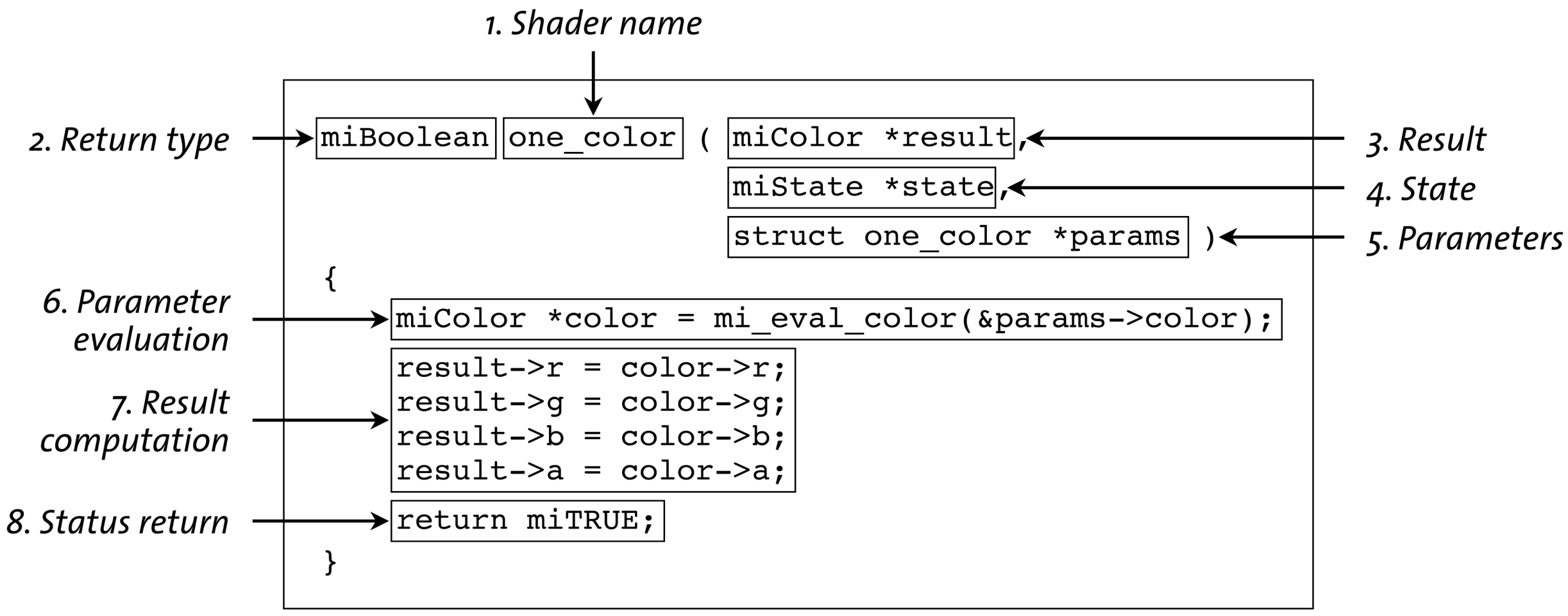
A single color



Model of shader one_color

A single color

The shader function in C



The components of a shader

A single color

The shader function in C

```
struct one_color {
    miColor color;
};
```

Parameter structure declaration

A single color

The shader source file

1. Library function declarations

```
#include "shader.h"
```

2. Version function

```
int one_color_version(void) { return 1; }
```

3. Parameter structure

```
struct one_color {
    miColor color;
};
```

4. Shader function

```
miBoolean one_color ( miColor *result,
                      miState *state,
                      struct one_color *params )
{
    miColor *color = mi_eval_color(&params->color);
    result->r = color->r;
    result->g = color->g;
    result->b = color->b;
    result->a = color->a;
    return miTRUE;
}
```

The components of a simple shader source file

A single color

The shader source file

```
int one_color_version(void) { return 1; }

struct one_color {
    miColor color;
};
```

C source file

```
declare shader
    color "one_color" (
        color "color"
    )
    version 1
end declare
```

Scene file

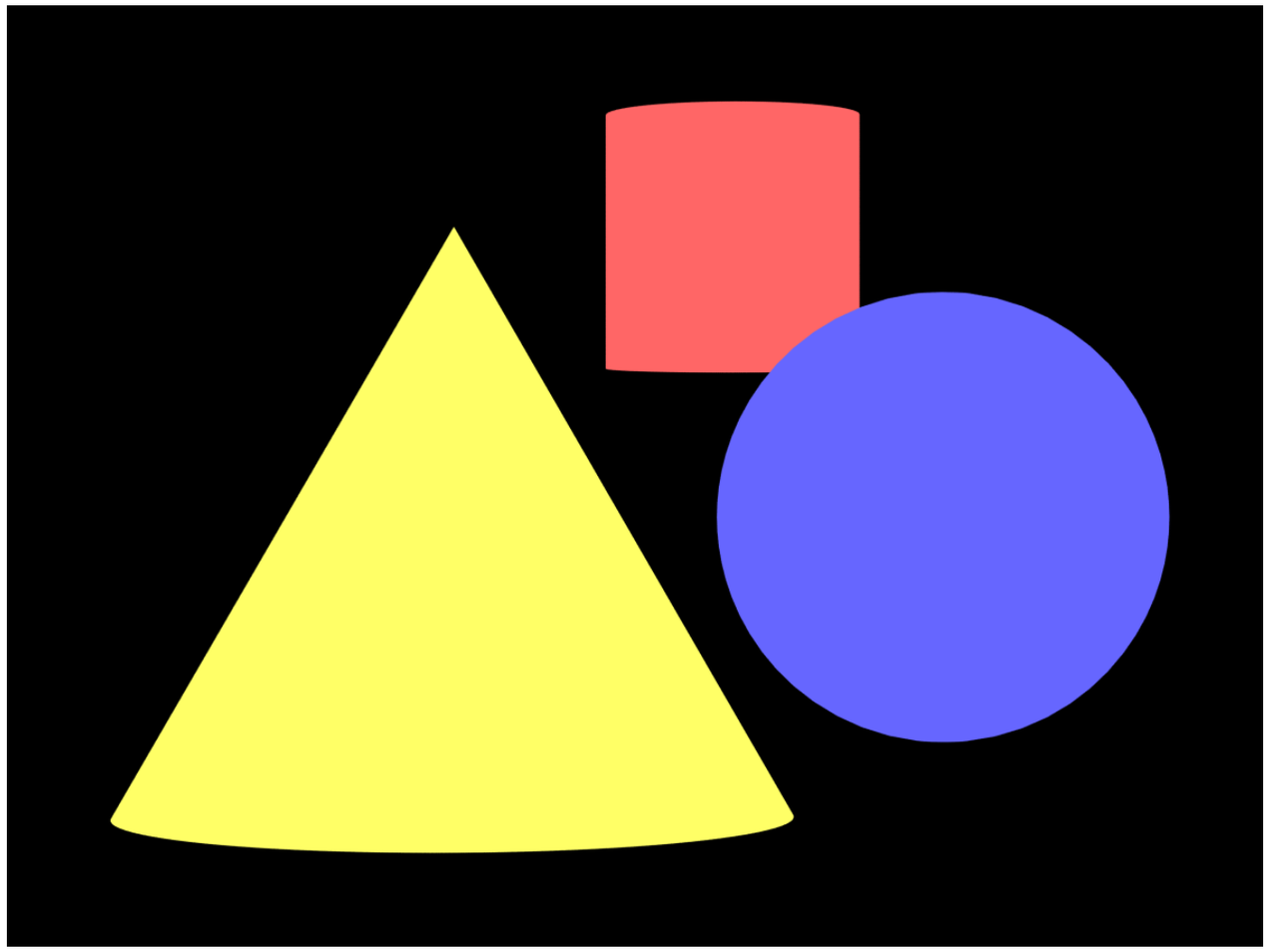
Relationship of C parameters to scene file declaration

Scene file data types and their C equivalents

.mi syntax	*C syntax* tline
`boolean`	`miBoolean`
`integer`	`miInteger`
`scalar`	`miScalar`
`vector`	`miVector`
`transform`	`miMatrix`
`color`	`miColor`
`string`	`miTag` of a `char*`
`scalar texture`	`miTag` of an `miFunction` or `miImg_image`
`vector texture`	`miTag` of an `miFunction` or `miImg_image`
`color texture`	`miTag` of an `miFunction` or `miImg_image`
`light`	`miTag` of an `miInstance` for a light or a light group
`material`	`miTag` of an `miMaterial`
`geometry`	`miTag` of an `miObject`, `miGroup`, or `miInstance`
`lightprofile`	`miTag` of an `miLight_profile`
`data`	`miTag` of an `miUserdata`
`shader`	`miTag` of an `miFunction`
`struct`	`struct`
`array` *mi-type*	Three fields: `int` – offset from base address for first element `int` – number of elements in array *C-type* [1] – base address of an array of *C-type*

A single color

Using a shader in a material



```
material "yellow"
    "one_color" (
        "color" 1 1 .4 )
end material

material "blue"
    "one_color" (
        "color" .4 .4 1 )
end material

material "red"
    "one_color" (
        "color" 1 .4 .4 )
end material
```

A single color for the entire object defined by shader one_color in the material

A single color


```
miBoolean one_color (
    miColor *result, miState *state, struct one_color *params )
{
    miColor *color = mi_eval_color(&params->color);
    result->r = color->r;
    result->g = color->g;
    result->b = color->b;
    result->a = color->a;
    return miTRUE;
}
```

Putting the standard shader arguments on a single line

A single color


```
miBoolean one_color (
    miColor *result, miState *state, struct one_color *params )
{
    miColor *color = mi_eval_color(&params->color);
    *result = *color;
    return miTRUE;
}
```

Direct assignment of the evaluated parameter to a local variable

A single color


```
miBoolean one_color (
    miColor *result, miState *state, struct one_color *params )
{
    *result = *mi_eval_color(&params->color);
    return miTRUE;
}
```

Direct assignment of the evaluated parameter to the shader's result

Color from orientation

Color from orientation

The representation of rendering state: miState
A shader based on surface orientation
Passing parameters to a shader
Other fields in miState
Calculation of surface orientation
Shaders as analysis tools

Data categories in the state structure

Data categories in the state structure

- *Frame* – Camera information and rendering options

Data categories in the state structure

- *Frame* – Camera information and rendering options
- *Image samples* – Image coordinates and current shader description

Data categories in the state structure

- *Frame* – Camera information and rendering options
- *Image samples* – Image coordinates and current shader description
- *Rays* – The current ray being cast

Data categories in the state structure

- *Frame* – Camera information and rendering options
- *Image samples* – Image coordinates and current shader description
- *Rays* – The current ray being cast
- *Intersection* – The intersection point currently being rendered

Data categories in the state structure

- *Frame* – Camera information and rendering options
- *Image samples* – Image coordinates and current shader description
- *Rays* – The current ray being cast
- *Intersection* – The intersection point currently being rendered
- *Textures*, *motion*, *derivatives* – Texture mapping information for thecurrent intersection point

Data categories in the state structure

- *Frame* – Camera information and rendering options
- *Image samples* – Image coordinates and current shader description
- *Rays* – The current ray being cast
- *Intersection* – The intersection point currently being rendered
- *Textures*, *motion*, *derivatives* – Texture mapping information for thecurrent intersection point
- *User fields* – Storage of user-defined data for use by shaders

Color from orientation

A shader based on surface orientation

```
declare shader
    color "front_bright" (
        color "tint" default 1 1 1 )
end declare
```

Scene file declaration of shader "front_bright"

Color from orientation

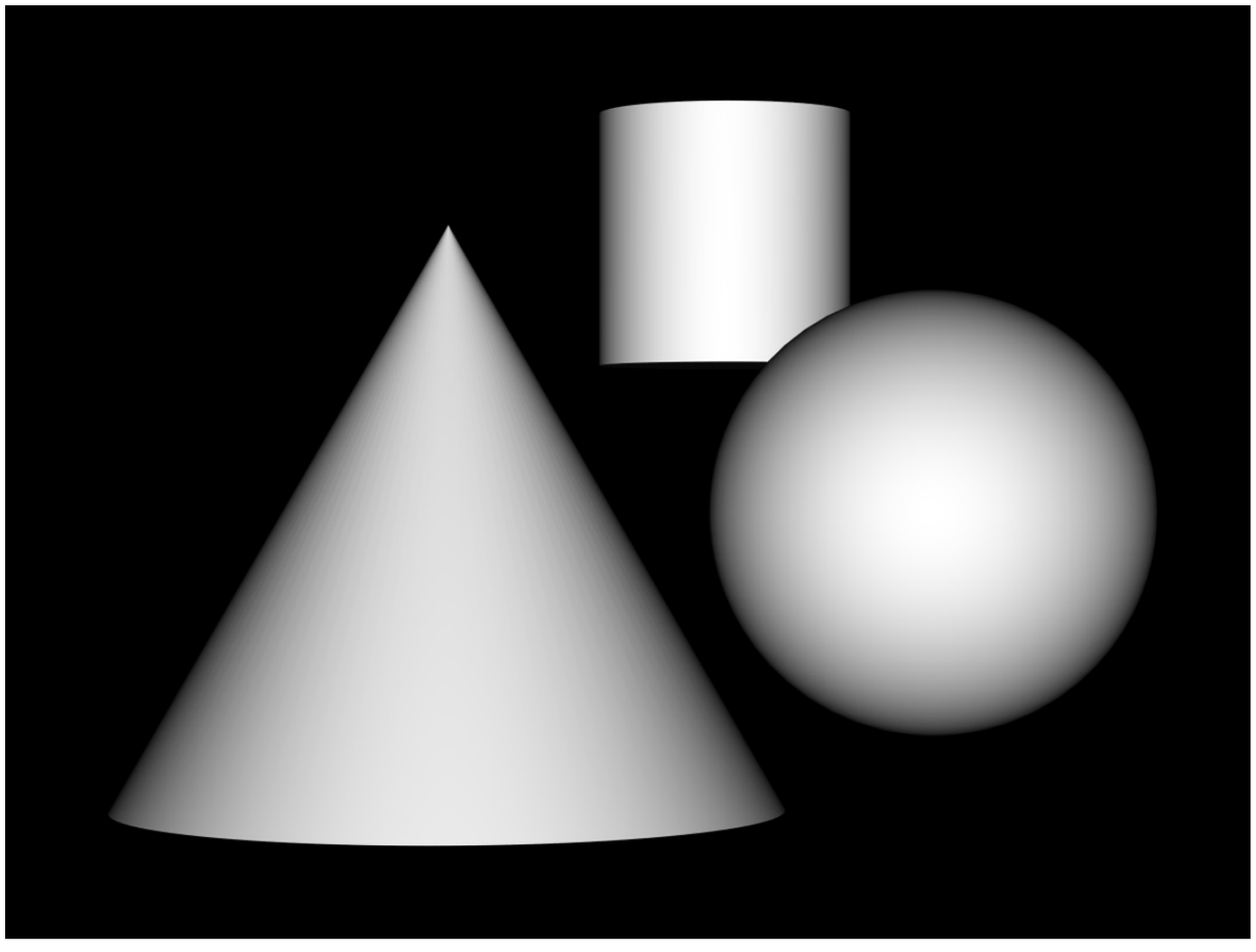
A shader based on surface orientation

```
 1  struct front_bright {
 2      miColor tint;
 3  };
 4
 5  miBoolean front_bright (
 6      miColor *result, miState *state, struct front_bright *params )
 7  {
 8      miColor *tint = mi_eval_color(&params->tint);
 9      miScalar scale = -state->dot_nd;
10      result->r = tint->r * scale;
11      result->g = tint->g * scale;
12      result->b = tint->b * scale;
13      result->a = 1.0;
14      return miTRUE;
15  }
```

Source code of shader "front_bright"

Color from orientation

A shader based on surface orientation

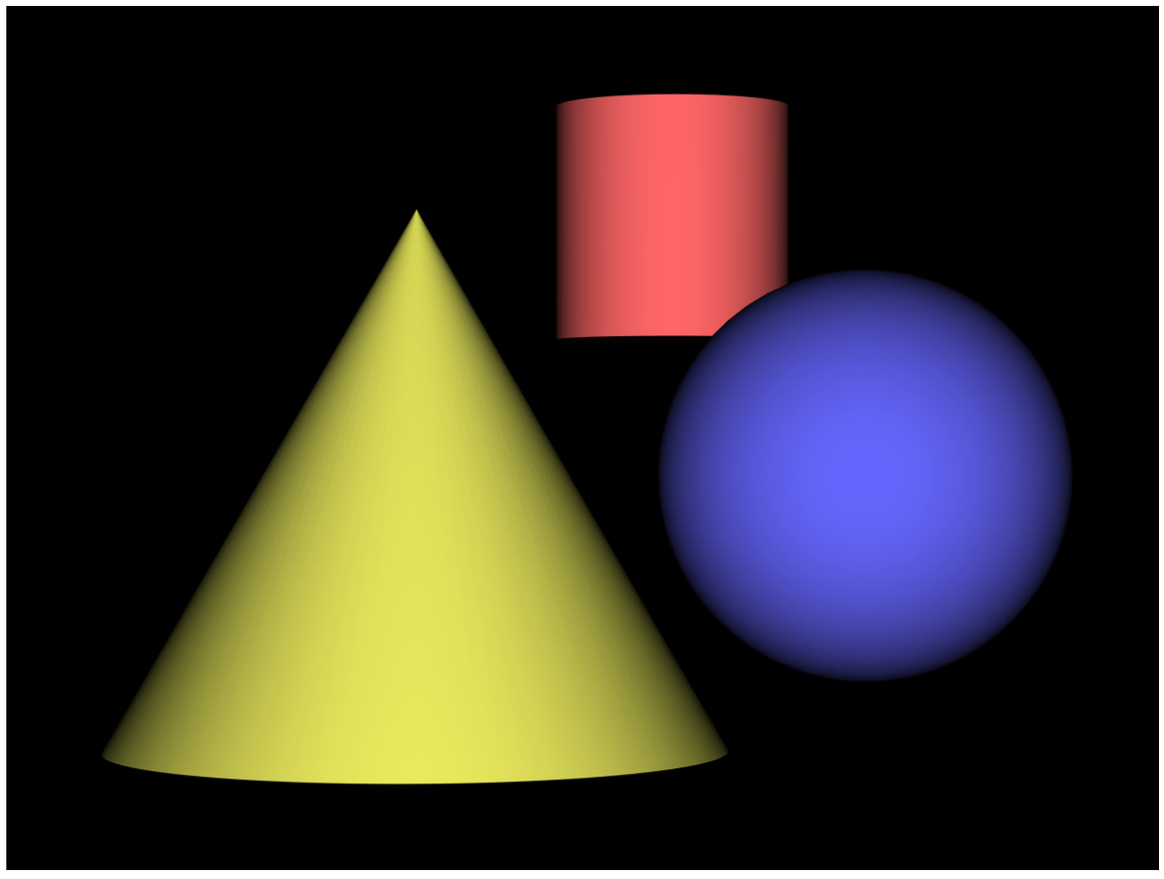


```
material "front"
    "front_bright" ()
end material
```

Using shader front_bright without explicit parameter values to define material front

Color from orientation

Passing parameters to a shader



```
material "front_red"
    "front_bright" ( "tint" 1 .4 .4 )
end material

material "front_yellow"
    "front_bright" ( "tint" 1 1 .4 )
end material

material "front_blue"
    "front_bright" ( "tint" .4 .4 1 )
end material
```

Three calls to shader front_bright with different values for the tint parameter

Documentation of `miState` in the manual

Exercise 4: Rendering state

1. Look up the documentation of rendering state in section *Using and Writing Shaders* in the on-line manual.
2. Find the description of the coordinates and the normal vector of the current sampling point.
3. Find the `miState` struct definition in header file `shader.h` (search for `typedef struct miState`).

Color from orientation

Calculation of surface orientation

```
 1  struct front_bright_dot {
 2      miColor tint;
 3  };
 4
 5  miBoolean front_bright_dot (
 6      miColor *result, miState *state, struct front_bright_dot *params )
 7  {
 8      miColor *tint = mi_eval_color(&params->tint);
 9      miScalar scale = -mi_vector_dot(&state->normal, &state->dir);
10      result->r = tint->r * scale;
11      result->g = tint->g * scale;
12      result->b = tint->b * scale;
13      result->a = 1.0;
14      return miTRUE;
15  }
```

Source code of shader "front_bright_dot"

Documentation of library functions

- Using and Writing Shaders
  - Functions for Shaders
    - DB Functions
    - RC Functions
    - Sampling with `mi_sample`
    - RC Photon Functions
    - RC Direction Functions
    - IMG Functions

Documentation of library functions

- Using and Writing Shaders
  - Functions for Shaders
    - *(continued)*
    - Math Functions
    - Noise Functions
    - KD-Tree Functions
    - Shading Models
    - Color Profile Function
    - Auxiliary Functions
    - Multipass Rendering Functions

Documentation of library functions

- Using and Writing Shaders
  - Functions for Shaders
    - *(continued)*
      - Obsolete Auxiliary Functions
      - Contour Functions
      - Light Map Functions
      - Memory Allocation
      - Thread Parallelism and Locks
      - Messages and Errors
      - Callable Functions by Shader Type

Exercise 5: The mental ray API library

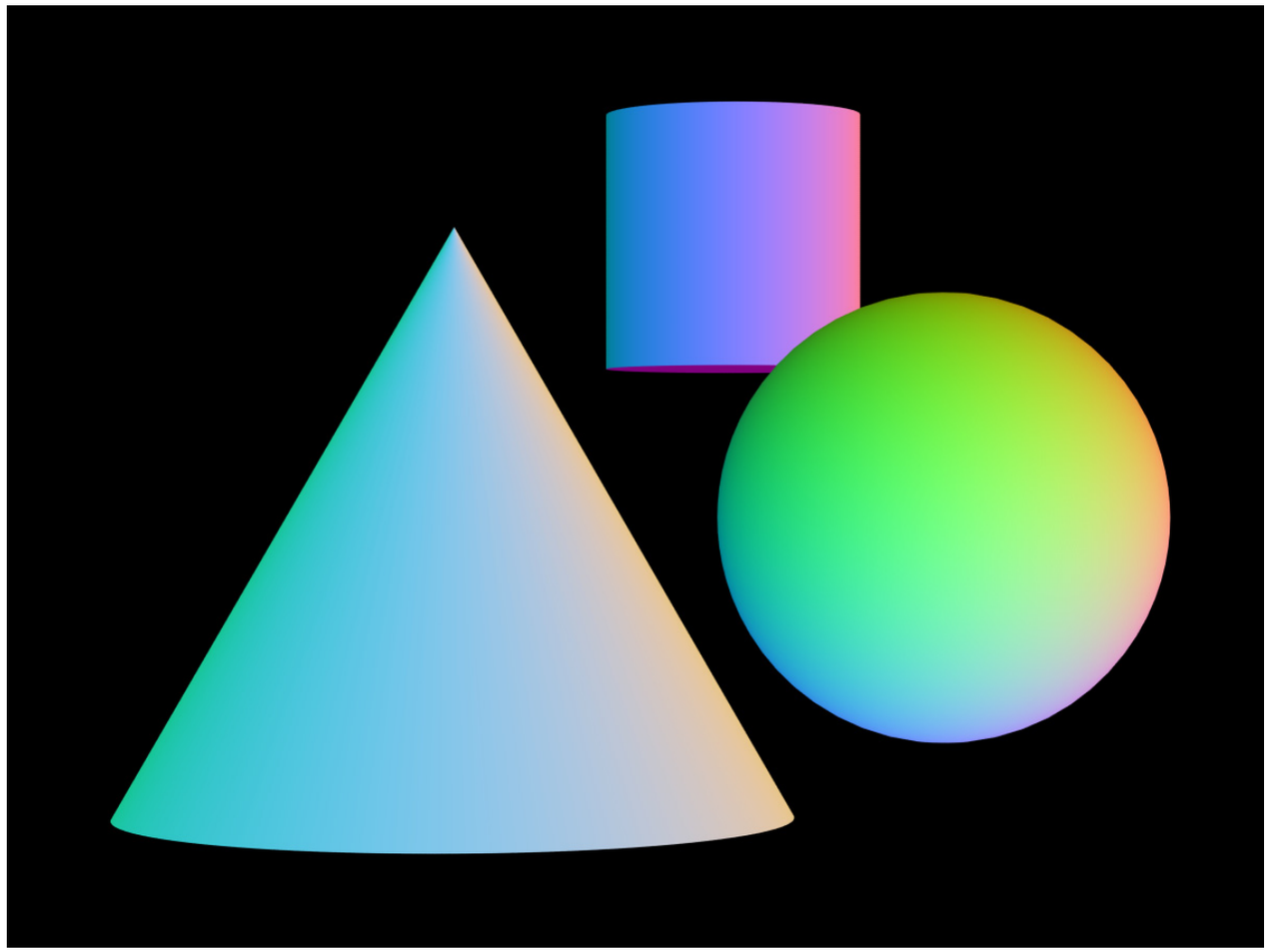
1. Look up the documentation of the mental ray API library in the *Functions for Shaders* subsection of *Using and Writing Shaders* in the on-line manual.
2. Find the documentation of `mi_vector_dot` by clicking on its entry in the *Index* section of the on-line manual.

Color from orientation


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

Scene file declaration of shader "normals_as_colors"

Color from orientation



```
material "normals"
    "normals_as_colors" ()
end material
```

Interpreting the surface normal vector as a color

Color from orientation


```
 1  miBoolean normals_as_colors (
 2      miColor *result, miState *state, void *params )
 3  {
 4      miVector normal;
 5      mi_vector_to_object(state, &normal, &state->normal);
 6      mi_vector_normalize(&normal);
 7      result->r = normal.x / 2.0 + .5;
 8      result->g = normal.y / 2.0 + .5;
 9      result->b = normal.z / 2.0 + .5;
10      result->a = 1.0;
11      return miTRUE;
12  }
```

Source code of shader "normals_as_colors"

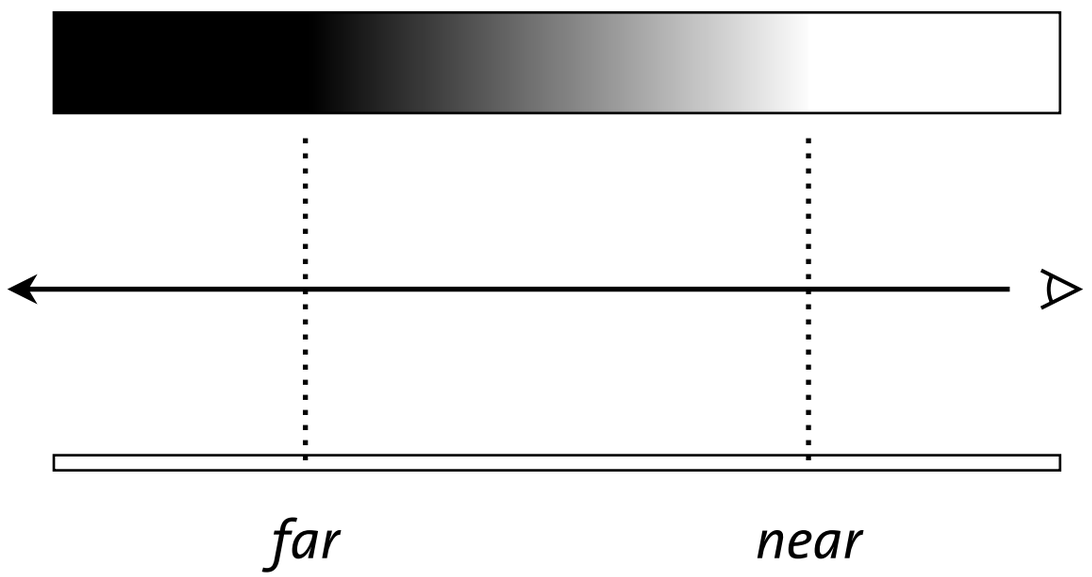
Color from position

Color from position

- Intensity based on distance
- Interpolating between colors based on distance
- Clarifying the shader with auxiliary functions

Color from position

Intensity based on distance



Intensity values defined by the distance from the camera

Color from position

Intensity based on distance

```
declare shader
    color "depth_fade" (
        scalar "near",
        scalar "far" )
end declare
```

Scene file declaration of shader "depth_fade"

Color from position

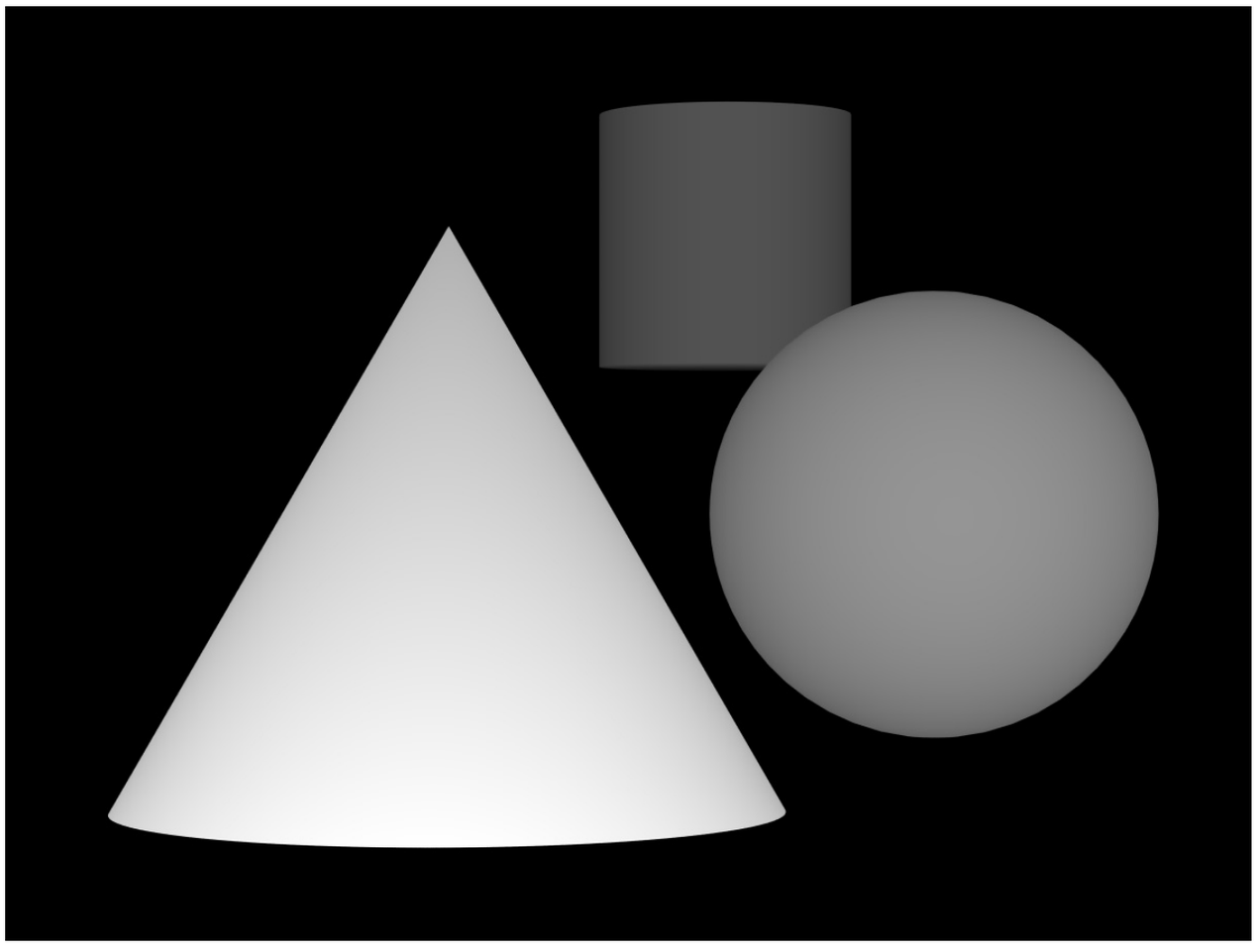
Intensity based on distance

```
 1  struct depth_fade {
 2      miScalar near;
 3      miScalar far;
 4  };
 5
 6  miBoolean depth_fade (
 7      miColor *result, miState *state, struct depth_fade *params )
 8  {
 9      miScalar near = *mi_eval_scalar(&params->near);
10      miScalar far = *mi_eval_scalar(&params->far);
11      miScalar zpos = state->point.z, factor;
12      if (zpos > near)
13          factor = 1.0;
14      else if (zpos < far)
15          factor = 0.0;
16      else
17          factor =  (zpos - far) / (near - far);
18      result->r = result->g = result->b = factor;
19      return miTRUE;
20  }
```

Source code of shader "depth_fade"

Color from position

Intensity based on distance



```
material "depth"
    "depth_fade" (
        "near" 1.25,
        "far" -1.15 )
end material
```

Mapping the z coordinate of a point on a surface to a grayscale value

Color from position

Interpolating between colors based on distance

```
declare shader
    color "depth_fade_tint" (
        scalar "near",
        color  "near_color" default 1 1 1,
        scalar "far",
        color  "far_color" default 0 0 0 )
end declare
```

Scene file declaration of shader "depth_fade_tint"

Color from position

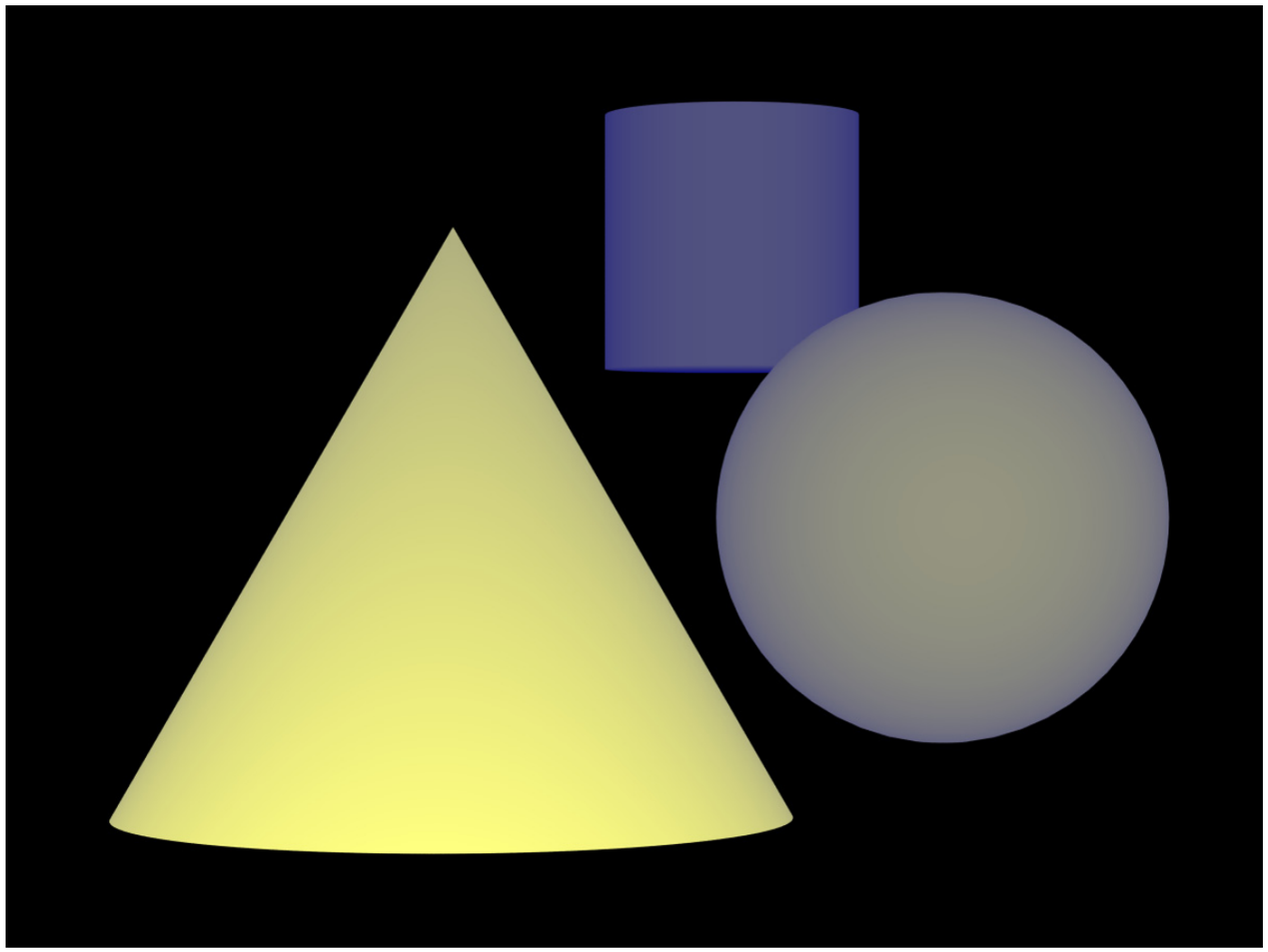
Interpolating between colors based on distance

```
 1  struct depth_fade_tint {
 2      miScalar near;
 3      miColor near_color;
 4      miScalar far;
 5      miColor far_color;
 6  };
 7
 8  miBoolean depth_fade_tint (
 9      miColor *result, miState *state, struct depth_fade_tint *params )
10  {
11      miScalar near = *mi_eval_scalar(&params->near);
12      miColor *near_color = mi_eval_color(&params->near_color);
13      miScalar far = *mi_eval_scalar(&params->far);
14      miColor *far_color = mi_eval_color(&params->far_color);
15      miScalar zpos = state->point.z, factor;
16
17      if (zpos > near)
18          factor = 1.0;
19      else if (zpos < far)
20          factor = 0.0;
21      else
22          factor =  (zpos - far) / (near - far);
23
24      result->r = near_color->r * factor + far_color->r * (1.0 - factor);
25      result->g = near_color->g * factor + far_color->g * (1.0 - factor);
26      result->b = near_color->b * factor + far_color->b * (1.0 - factor);
27
28      return miTRUE;
29  }
```

Source code of shader "depth_fade_tint"

Color from position

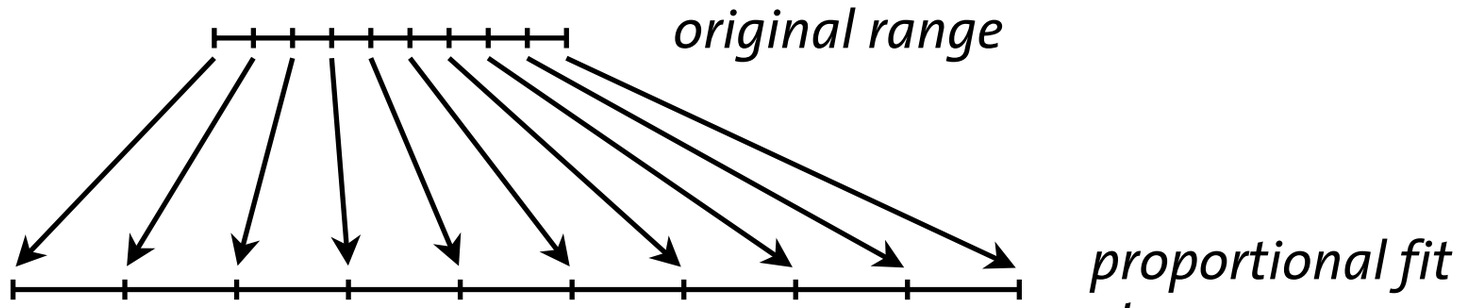
Interpolating between colors based on distance



```
material "depth"
    "depth_fade_tint" (
        "near" 1.25,
        "near_color" 1 1 .5,
        "far" -1.15,
        "far_color" 0 0 .5 )
end material
```

Blending between two colors based on the z coordinate of a point on a surface

Color from position



Converting from one scale to another

Color from position

Clarifying the shader with auxiliary functions

```
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

Color from position

Clarifying the shader with auxiliary functions

```
1  double miaux_blend(miScalar a, miScalar b, miScalar factor)
2  {
3      return a * factor + b * (1.0 - factor);
4  }
```

Auxiliary function: miaux_blend

Color from position

Clarifying the shader with auxiliary functions

```
1  void miaux_blend_colors(miColor *result,
2                          miColor *color1, miColor *color2, miScalar factor)
3  {
4      result->r = miaux_blend(color1->r, color2->r, factor);
5      result->g = miaux_blend(color1->g, color2->g, factor);
6      result->b = miaux_blend(color1->b, color2->b, factor);
7  }
```

Auxiliary function: miaux_blend_colors

Color from position

Clarifying the shader with auxiliary functions

```
 1  struct depth_fade_tint_2 {
 2      miScalar near;
 3      miColor near_color;
 4      miScalar far;
 5      miColor far_color;
 6  };
 7
 8  miBoolean depth_fade_tint_2 (
 9      miColor *result, miState *state, struct depth_fade_tint_2 *params )
10  {
11      miScalar near = *mi_eval_scalar(&params->near);
12      miColor *near_color = mi_eval_color(&params->near_color);
13      miScalar far = *mi_eval_scalar(&params->far);
14      miColor *far_color = mi_eval_color(&params->far_color);
15
16      miaux_blend_colors(result, near_color, far_color,
17                         miaux_fit(state->point.z, far, near, 0.0, 1.0));
18
19      return miTRUE;
20  }
```

Source code of shader "depth_fade_tint_2"

Color from position

Clarifying the shader with auxiliary functions

```
 1  struct depth_fade_tint_3 {
 2      miScalar near;
 3      miColor near_color;
 4      miScalar far;
 5      miColor far_color;
 6  };
 7
 8  miBoolean depth_fade_tint_3 (
 9      miColor *result, miState *state, struct depth_fade_tint_3 *params )
10  {
11      miaux_blend_colors(result,
12                         mi_eval_color(&params->near_color),
13                         mi_eval_color(&params->far_color),
14                         miaux_fit(state->point.z,
15                                   *mi_eval_scalar(&params->far),
16                                   *mi_eval_scalar(&params->near),
17                                   0.0, 1.0));
18      return miTRUE;
19  }
```

Source code of shader "depth_fade_tint_3"

Exercise 6: Compile and use color shaders

1. Check that `one_color.c` has been compiled. (How?)
2. Render `single_color_1.mi` and view the result. (How?)
3. In the `MRT/shaders` directory, compile `front_bright.c` using `front_bright.bat` (Windows) or `make front_bright.so` (Linux and OS X).
4. Copy `single_color_1.mi` to `color.mi` and change output to `color.tif`
5. Change `color.mi` to use shader `front_bright` without parameters, and re-render.
6. Change the use of `front_bright` in the material to use the `tint` parameter and re-render.
7. Compile `depth_fade_tint.c` using `depth_fade_tint.bat`
8. Change `color.mi` to use `depth_fade_tint` and re-render.

Exercise 7: Combining orientation and depth in a Phenomenon

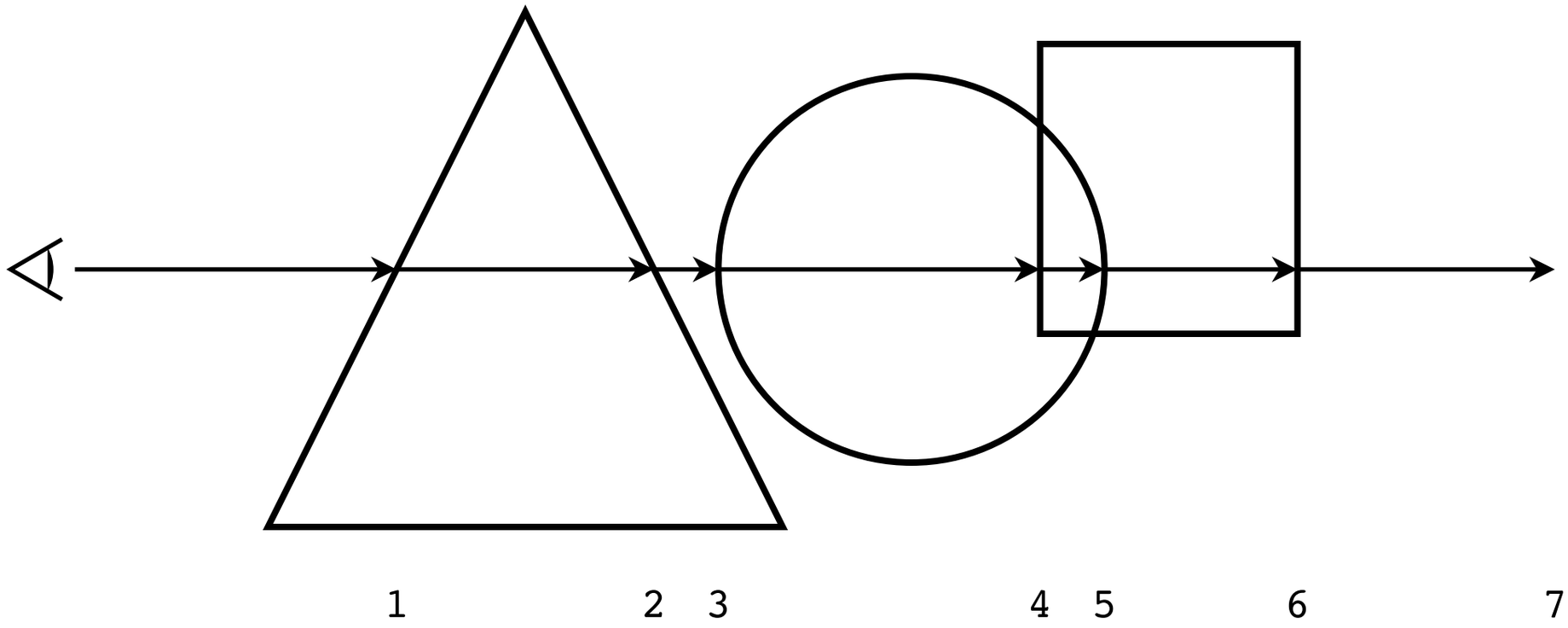
1. Render `phenomena_A.mi`.
2. Compare `phenomena_A.mi` to scene `depth_2.mi`. Notice how the shader in the material in `depth_2.mi`, `depth_fade_tint`, has been enclosed in a Phenomena and multiplied by `front_bright` using the shader `op_mul_cc`.
3. Examine Phenomenon `red_front_depth`. Create a material that uses it like material `fade` uses `front` depth. Assign this new material to an object.
4. Make a Phenomenon like `red_front_depth` using another color. Create a material using it and assign it to an object.
5. Find camera `side-cam` and its instance. Use it instead in the root group and the `render` command.

The transparency of a surface

The transparency of a surface

Using the API library to trace a ray

Functions as a description of a process



Ray intersections in the calculation of the transparent value

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

Scene file declaration of shader "transparent"

The transparency of a surface

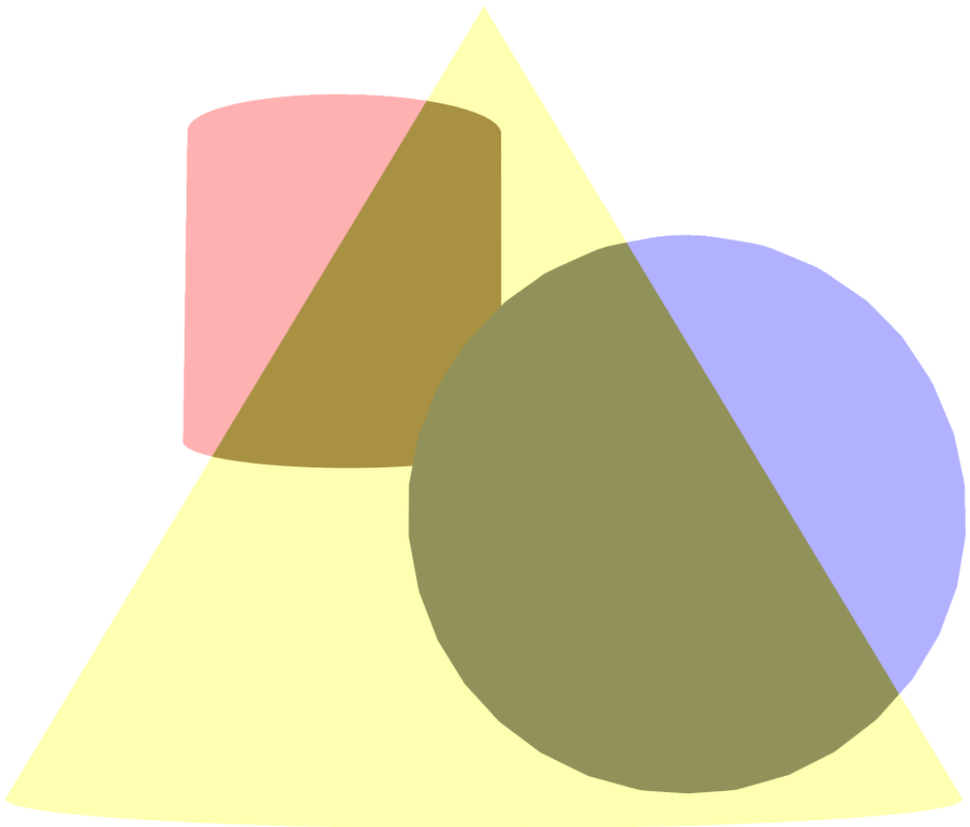
Using the API library to trace a ray

```
 1  struct transparent {
 2      miColor color;
 3      miColor transparency;
 4  };
 5
 6  miBoolean transparent (
 7      miColor *result, miState *state, struct transparent *params )
 8  {
 9      miColor *transparency = mi_eval_color(&params->transparency);
10      if (transparency->r == 0.0 && transparency->g == 0.0 &&
11          transparency->b == 0.0 && transparency->a == 0.0)
12          *result = *mi_eval_color(&params->color);
13      else {
14          mi_trace_transparent(result, state);
15          if (!(transparency->r == 1.0 && transparency->g == 1.0 &&
16                transparency->b == 1.0 && transparency->a == 1.0)) {
17              miColor *color = mi_eval_color(&params->color);
18              miColor opacity;
19              opacity.r = 1.0 - transparency->r;
20              opacity.g = 1.0 - transparency->g;
21              opacity.b = 1.0 - transparency->b;
22              opacity.a = 1.0 - transparency->a;
23              mi_opacity_set(state, &opacity);
24              result->r = result->r * transparency->r + color->r * opacity.r;
25              result->g = result->g * transparency->g + color->g * opacity.g;
26              result->b = result->b * transparency->b + color->b * opacity.b;
27          }
28      }
29      return miTRUE;
30  }
```

Source code of shader "transparent"

The transparency of a surface

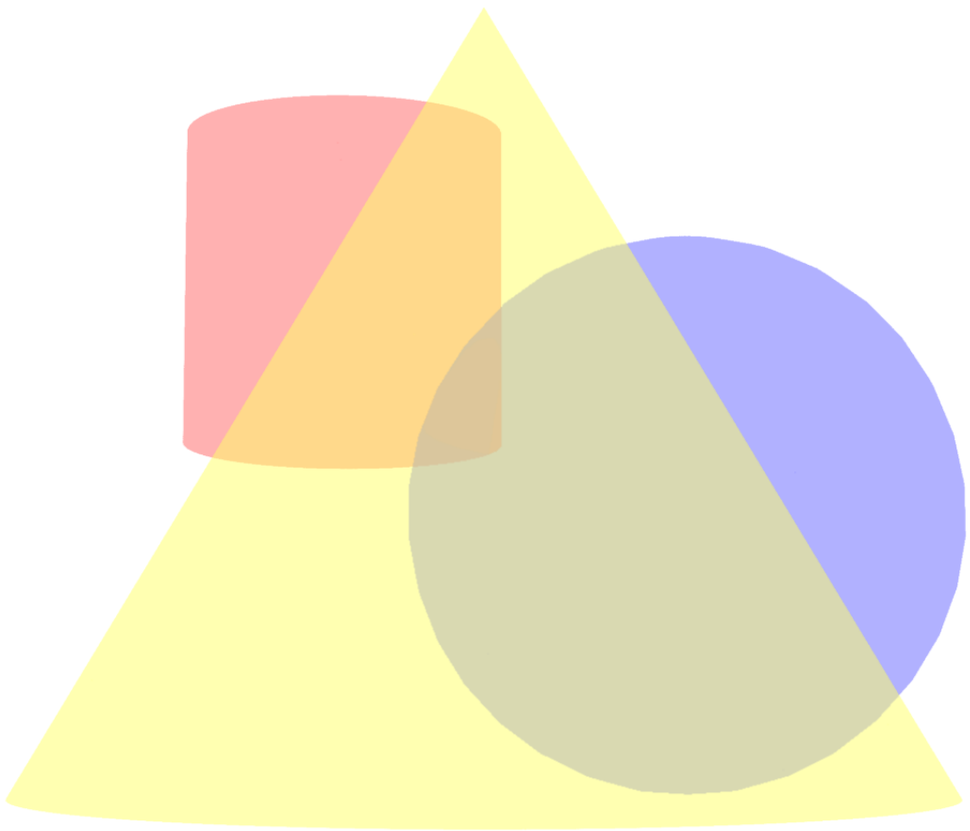
Using the API library to trace a ray



```
material "red_transparent"
    "transparent" (
        "color" 1 .4 .4,
        "transparency" .7 .7 .7 )
end material

material "yellow_transparent"
    "transparent" (
        "color" 1 1 .4,
        "transparency" .7 .7 .7 )
end material

material "blue_transparent"
    "transparent" (
        "color" .4 .4 1,
        "transparency" .7 .7 .7 )
end material
```

Three objects with transparency

The transparency of a surface

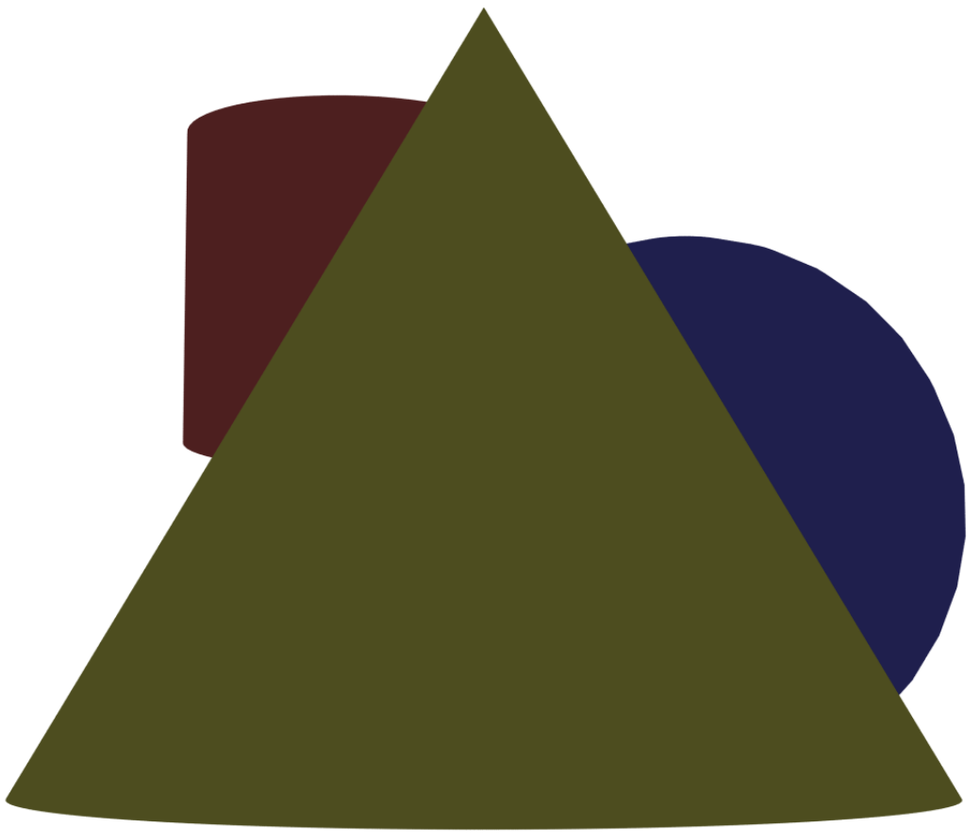


Using the API library to trace a ray

```
options "opt"
    object space
    contrast .1 .1 .1 1
    samples 0 2
    trace depth 0 6 6
end options

material "red_transparent"
    "transparent" (
        "color" 1 .4 .4,
        "transparency" .7 .7 .7 )
end material

material "yellow_transparent"
    "transparent" (
        "color" 1 1 .4,
        "transparency" .7 .7 .7 )
end material

material "blue_transparent"
    "transparent" (
        "color" .4 .4 1,
        "transparency" .7 .7 .7 )
end material
```

Adjusting the trace depth for transparency control

The transparency of a surface



Using the API library to trace a ray

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

material "red_transparent"
    "transparent" (
        "color" 1 .4 .4,
        "transparency" .7 .7 .7 )
end material

material "yellow_transparent"
    "transparent" (
        "color" 1 1 .4,
        "transparency" .7 .7 .7 )
end material

material "blue_transparent"
    "transparent" (
        "color" .4 .4 1,
        "transparency" .7 .7 .7 )
end material
```

A transmission trace depth of zero---opaque objects

```
1  miBoolean miaux_all_channels_equal(miColor *c, miScalar v)
2  {
3      if (c->r == v && c->g == v && c->b == v && c->a == v)
4          return miTRUE;
5      else
6          return miFALSE;
7  }
```

Auxiliary function: miaux_all_channels_equal

The transparency of a surface

Functions as a description of a process

```
1  void miaux_blend_channels(miColor *result,
2                            miColor *blend_color, miColor *blend_fraction)
3  {
4      result->r = miaux_blend(result->r, blend_color->r, blend_fraction->r);
5      result->g = miaux_blend(result->g, blend_color->g, blend_fraction->g);
6      result->b = miaux_blend(result->b, blend_color->b, blend_fraction->b);
7  }
```

Auxiliary function: miaux_blend_channels

```
1  void miaux_invert_channels(miColor *result, miColor *color)
2  {
3      result->r = 1.0 - color->r;
4      result->g = 1.0 - color->g;
5      result->b = 1.0 - color->b;
6      result->a = 1.0 - color->a;
7  }
```

Auxiliary function: miaux_invert_channels

The transparency of a surface

Functions as a description of a process

```
 1  struct transparent_modularized {
 2      miColor color;
 3      miColor transparency;
 4  };
 5
 6  miBoolean transparent_modularized (
 7      miColor *result, miState *state, struct transparent_modularized *params )
 8  {
 9      miColor *transparency = mi_eval_color(&params->transparency);
10
11      if (miaux_all_channels_equal(transparency, 0.0))
12        *result = *mi_eval_color(&params->color);
13
14      else if (miaux_all_channels_equal(transparency, 1.0))
15        mi_trace_transparent(result, state);
16
17      else {
18        miColor *color = mi_eval_color(&params->color), opacity;
19        mi_trace_transparent(result, state);
20        miaux_invert_channels(&opacity, transparency);
21        mi_opacity_set(state, &opacity);
22        miaux_blend_channels(result, color, transparency);
23      }
24
25      return miTRUE;
26  }
```

Source code of shader "transparent_modularized"

Exercise 8: Transparency shaders

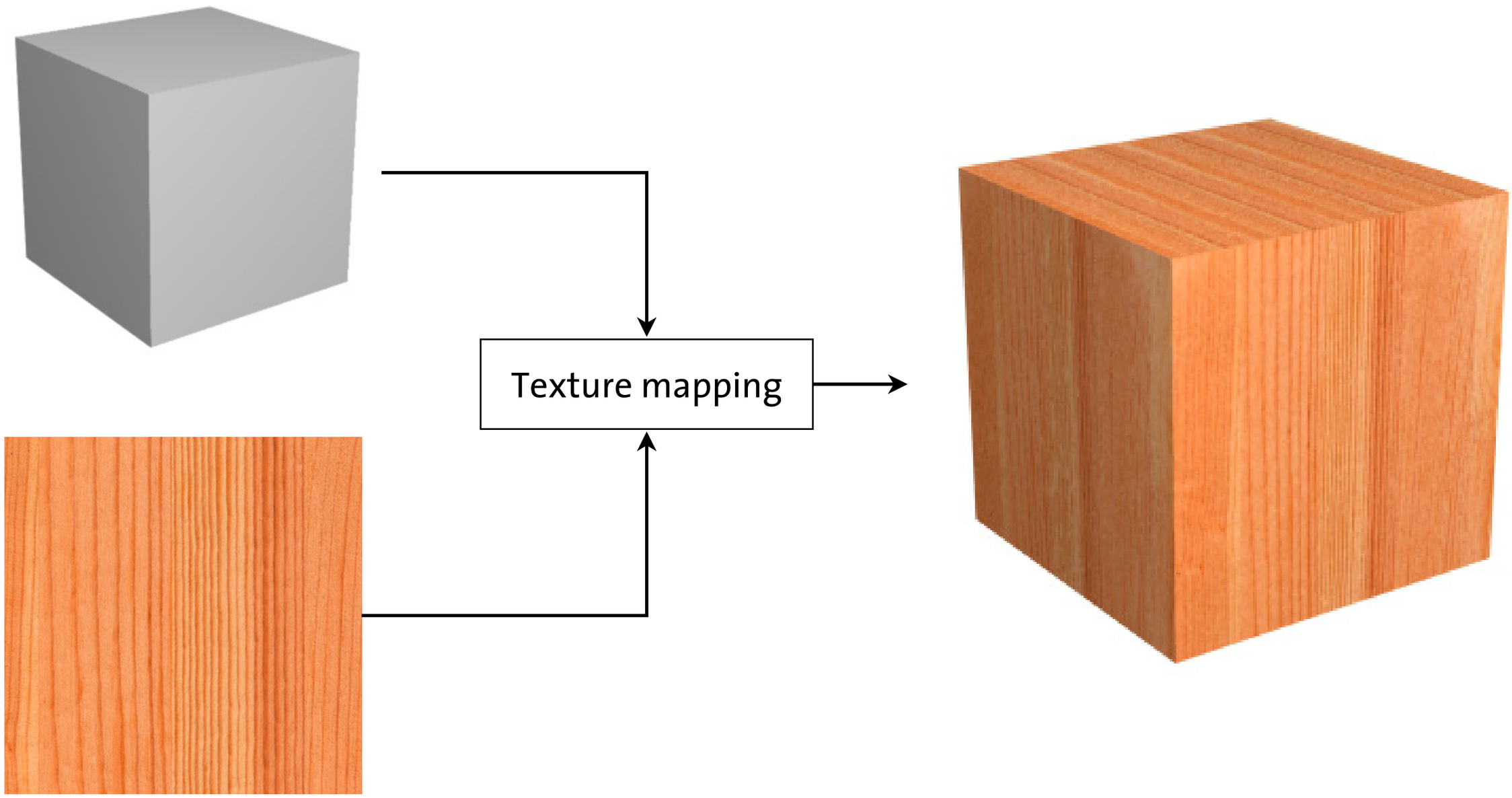
1. Compile all the shaders with `all_shaders.bat` (Windows) or `make all` (Linux and OS X).
2. Copy `transparency_1.mi` to `transparency.mi`, change output to `transparency.tif`
3. Render `transparency.mi`, view `transparency.tif` with `imf_disp`
4. Change `transparency` parameter and re-render.
5. Change `trace depth` in the options block to `0 0 0` and re-render.
6. Change `trace depth` to `0 2 2` and re-render.
7. Change `trace depth` to `0 6 6` and re-render.

Color from functions

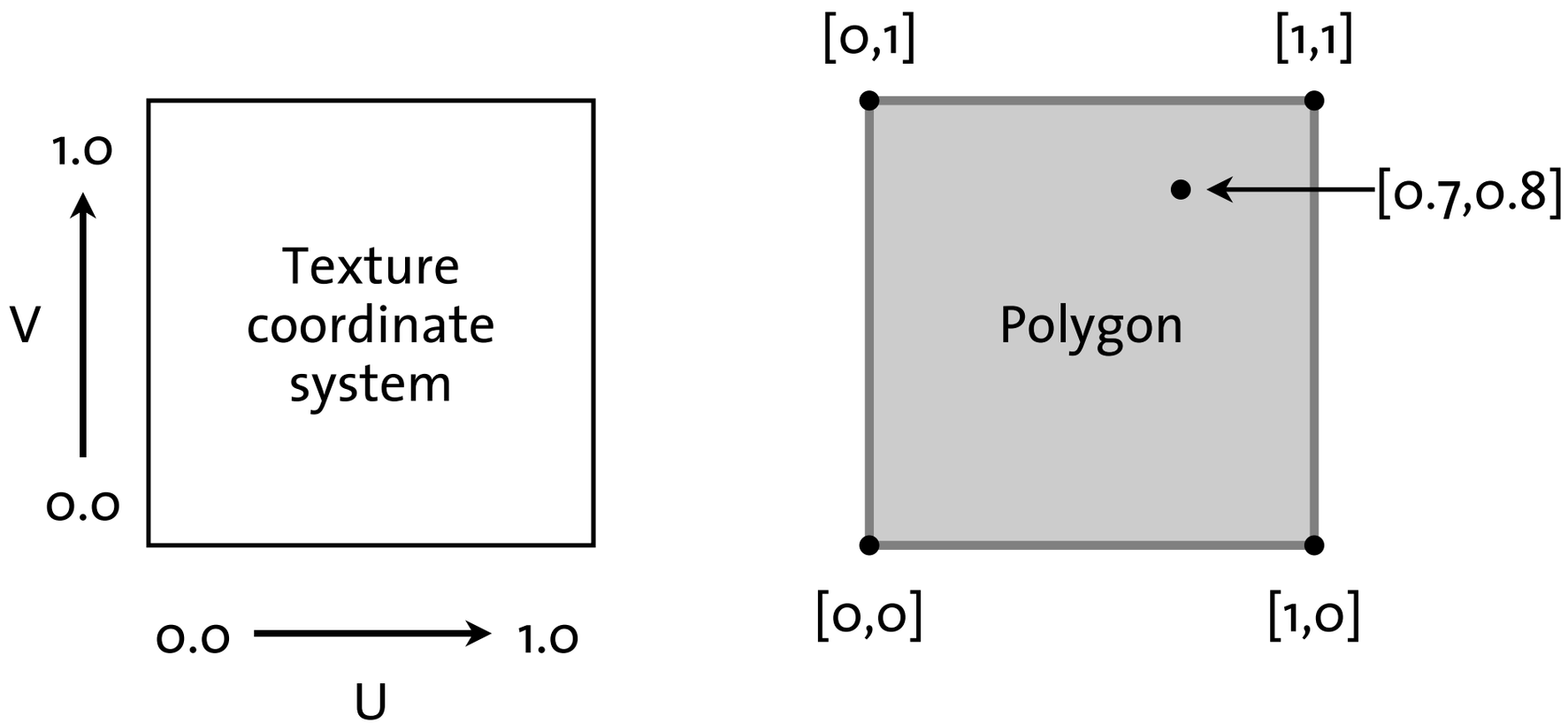
Color from functions

- The texture coordinate system
- Quantizing the texture coordinates
- Using the texture coordinates for texture mapping
- Manipulating the texture coordinates
- Defining texture maps that tile well
- Multiple texture spaces
- Noise functions

Color from functions



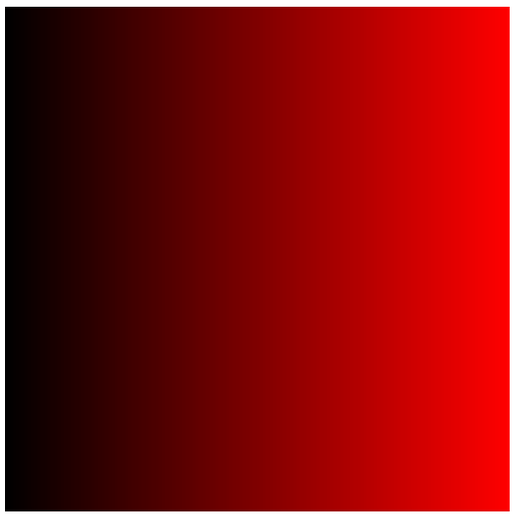
Using an image to define the surface color of an object



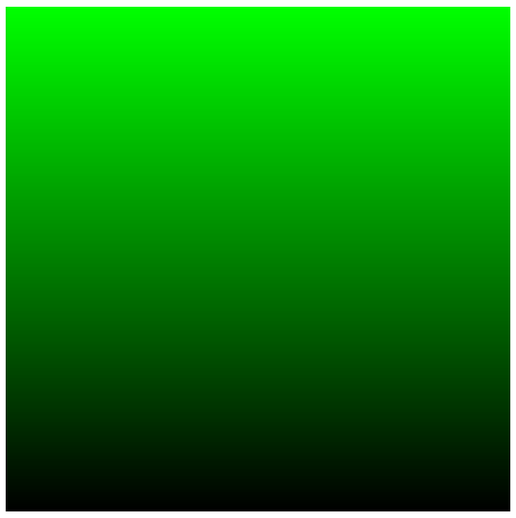
A texture coordinate system and the resulting texture coordinates

Color from functions

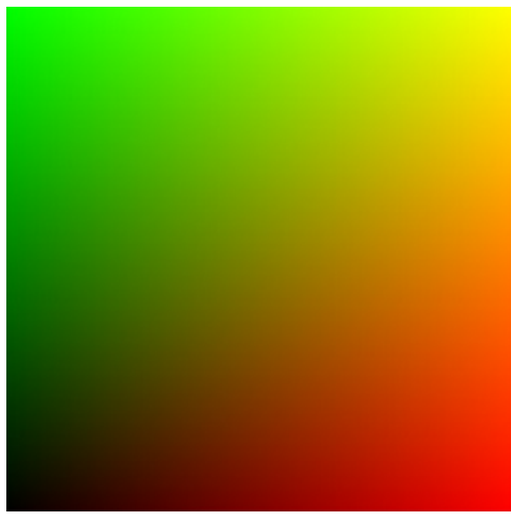
The texture coordinate system



U value as red



V value as green



U and V together

Position in the uv coordinate space interpreted as color

Color from functions

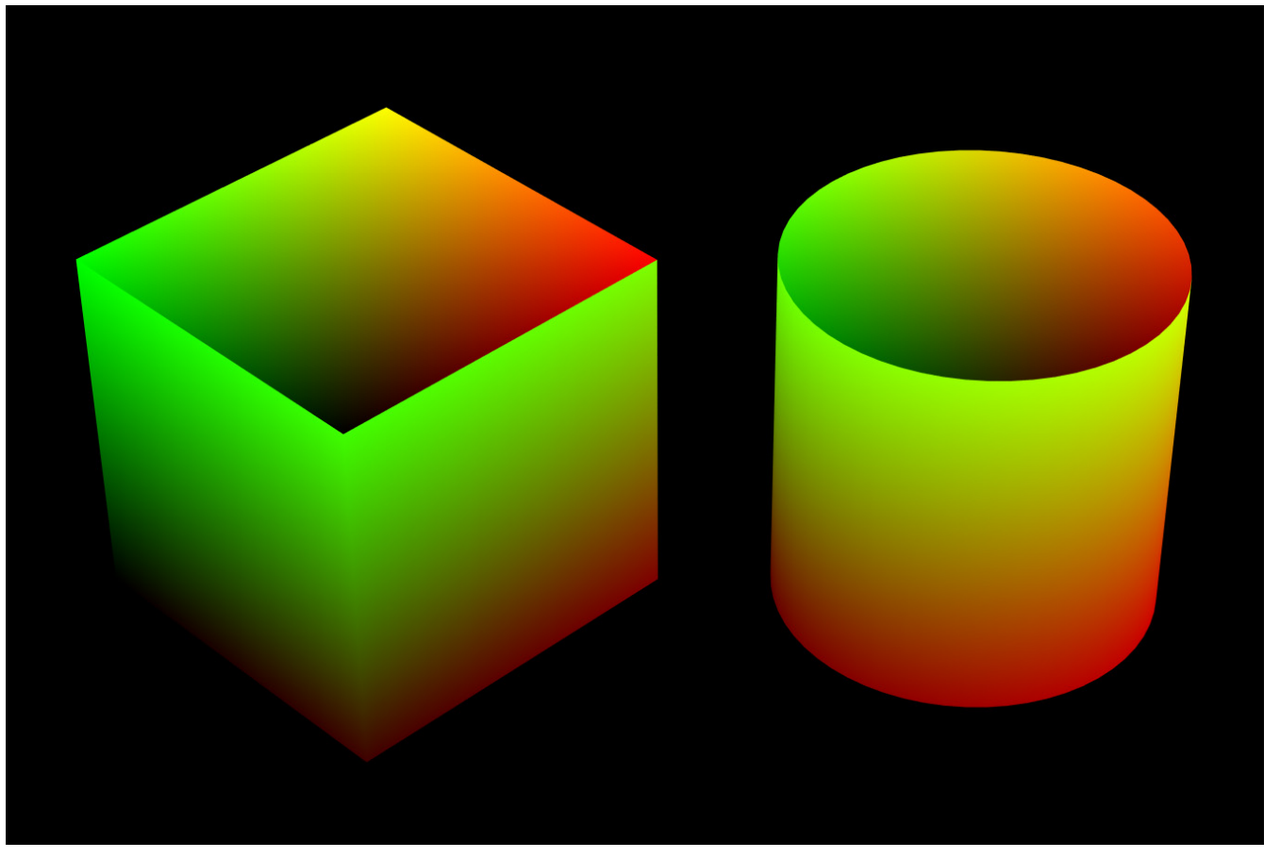
The texture coordinate system

```
declare shader
    color "show_uv" (
        boolean "u" default on,
        boolean "v" default on )
end declare
```

Scene file declaration of shader "show_uv"

Color from functions

The texture coordinate system



```
material "uv"
    "show_uv" ()
end material
```

The two-dimesional texture coordinates of a surface mapped to red and green values

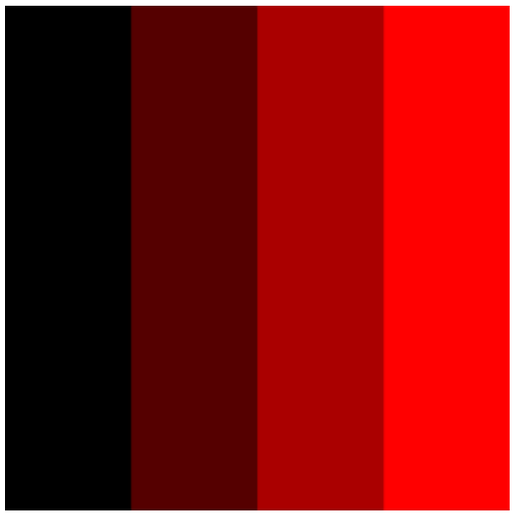
Color from functions


```
1  struct show_uv {
2      miBoolean u;
3      miBoolean v;
4  };
5
6  miBoolean show_uv(
7      miColor *result, miState *state, struct show_uv *params)
8  {
9      result->r = result->g = result->b = 0;
10     if (*mi_eval_boolean(&params->u))
11         result->r = state->tex_list[0].x;
12     if (*mi_eval_boolean(&params->v))
13         result->g = state->tex_list[0].y;
14     return miTRUE;
15 }
```

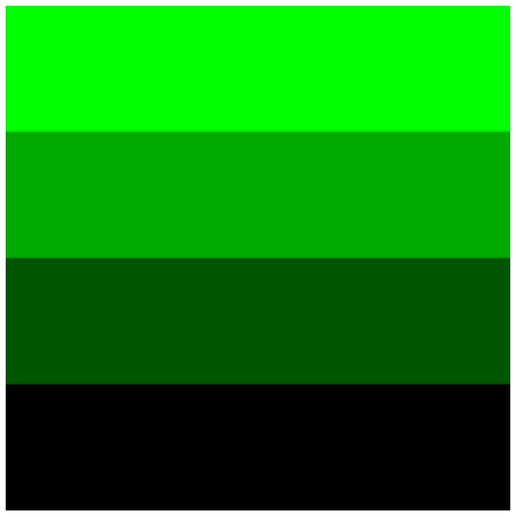
Source code of shader "show_uv"

Color from functions

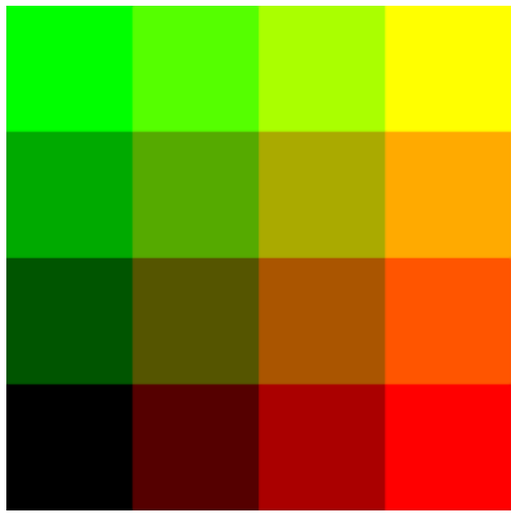
Quantizing the texture coordinates



U value as red



V value as green



U and V together

Quantized texture coordinate values as colors

Color from functions

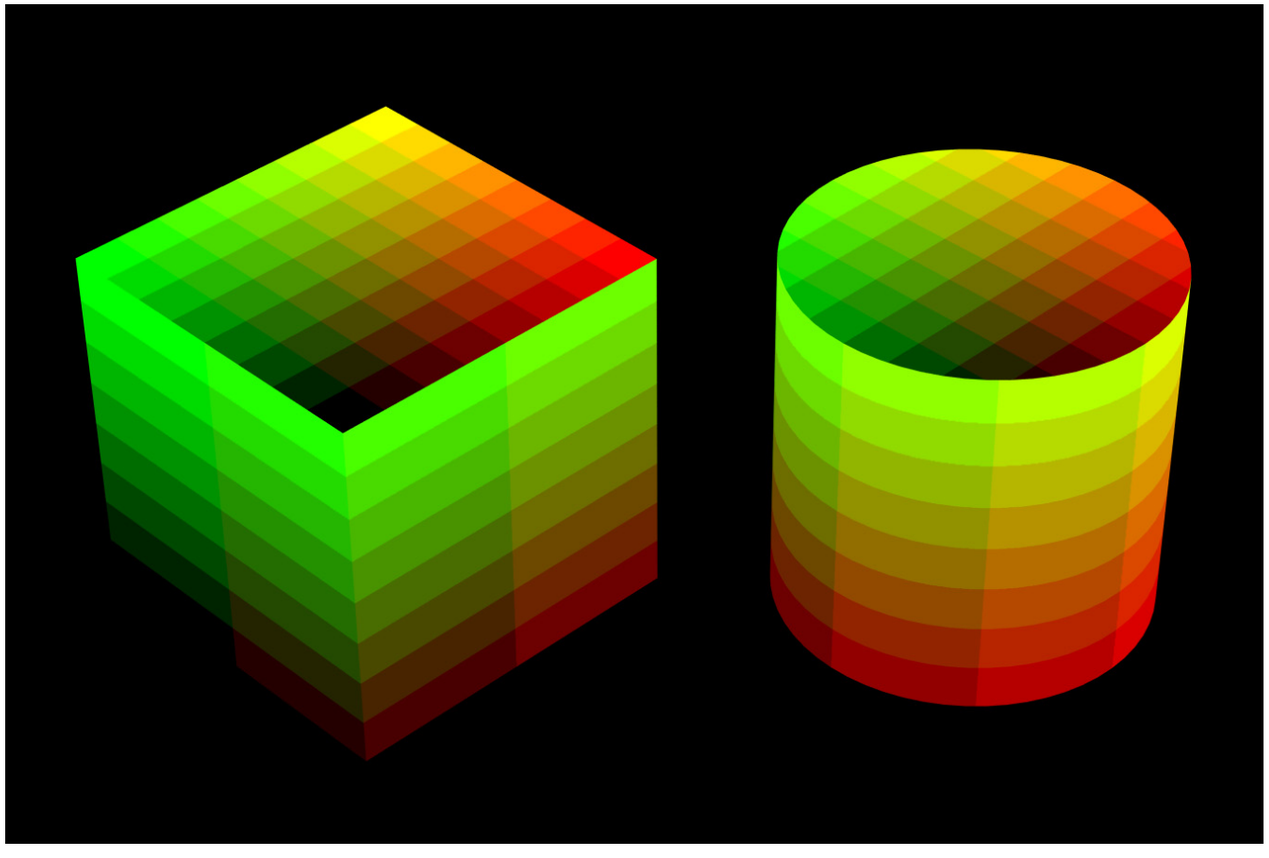
Quantizing the texture coordinates

```
declare shader
    color "show_uv_steps" (
        integer "u_count" default 4,
        integer "v_count" default 4 )
end declare
```

Scene file declaration of shader "show_uv_steps"

Color from functions

Quantizing the texture coordinates



```
material "uv"
    "show_uv_steps" (
        "u_count" 8,
        "v_count" 8 )
end material
```

Banding effect that demonstrates the scale of the texture space

Color from functions

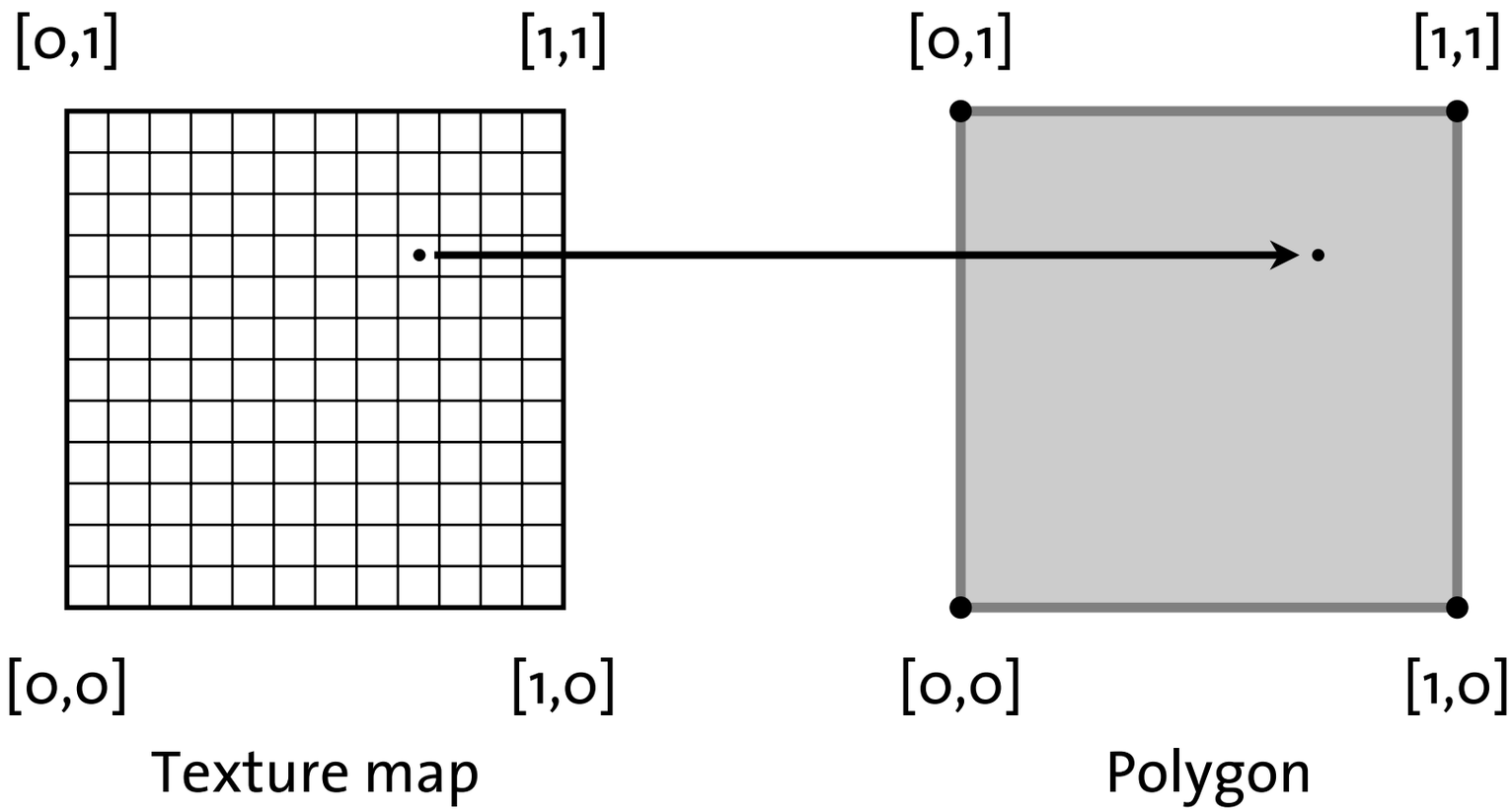
Quantizing the texture coordinates

```
 1  struct show_uv_steps {
 2      miInteger u_count;
 3      miInteger v_count;
 4  };
 5
 6  miScalar quantize(miScalar value, miInteger count)
 7  {
 8      miScalar q = (miScalar)count;
 9      if (count < 2)
10          return q;
11      else
12          return (miScalar)((int)(value * q) / (q - 1));
13  }
14
15  miBoolean show_uv_steps(
16      miColor *result, miState *state, struct show_uv_steps *params)
17  {
18      result->r = quantize(state->tex_list[0].x,
19                           *mi_eval_integer(&params->u_count));
20      result->g = quantize(state->tex_list[0].y,
21                           *mi_eval_integer(&params->v_count));
22      result->b = 0;
23      return miTRUE;
24  }
```

Source code of shader "show_uv_steps"

Color from functions

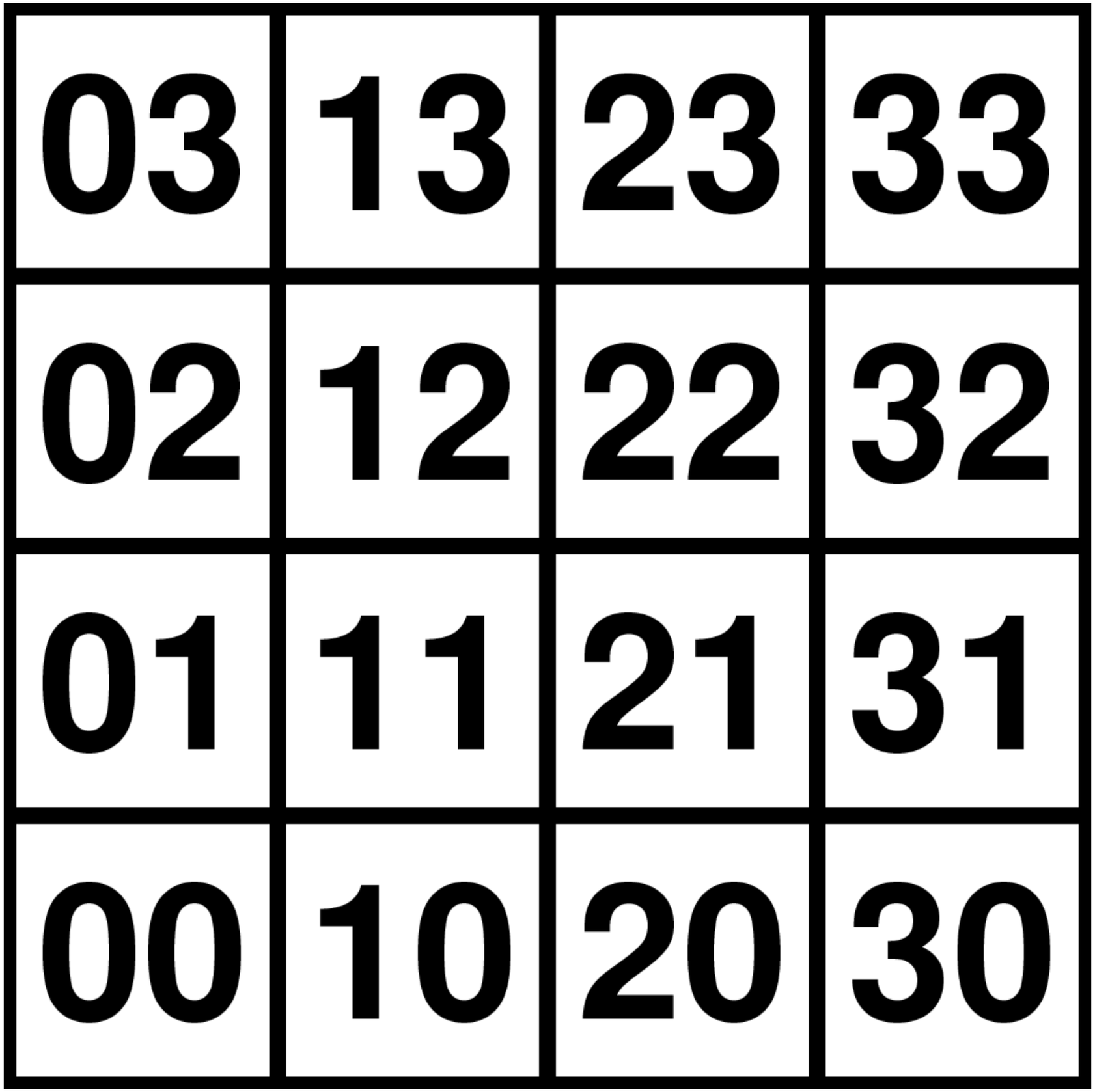
Using the texture coordinates for texture mapping



A mapping from a texture map to a polygon

Color from functions

Using the texture coordinates for texture mapping



A source image for texture mapping

Color from functions

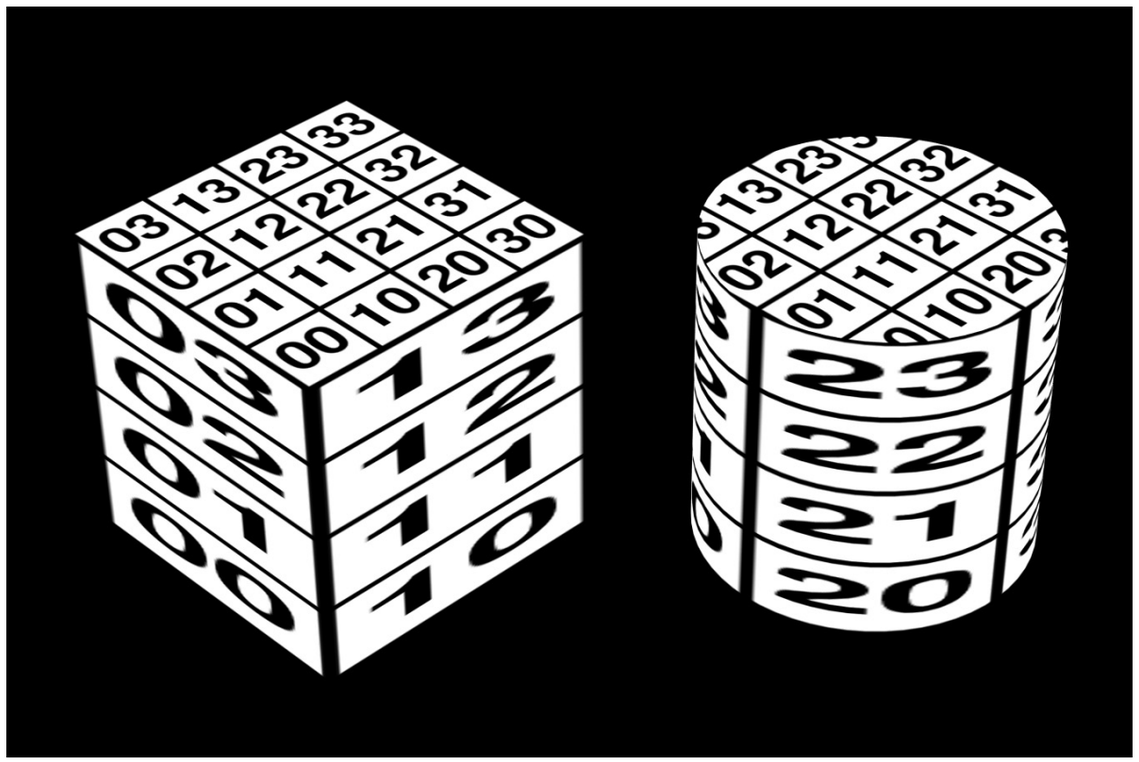
Using the texture coordinates for texture mapping

```
declare shader
    color "texture_uv_simple" (
        color texture "tex" )
end declare
```

Scene file declaration of shader "texture_uv_simple"

Color from functions

Using the texture coordinates for texture mapping




```
color texture "fourgrid" "fourgrid.tif"

material "grid"
    "texture_uv_simple" (
        "tex" "fourgrid" )
end material
```

Using a texture map to define surface color

Color from functions

Using the texture coordinates for texture mapping

```
1  struct texture_uv_simple {
2      miTag tex;
3  };
4
5  miBoolean texture_uv_simple(
6      miColor *result, miState *state, struct texture_uv_simple *paras)
7  {
8      mi_lookup_color_texture(
9          result, state, *mi_eval_tag(&paras->tex), &state->tex_list[0]);
10
11     return miTRUE;
12 }
```

Source code of shader "texture_uv_simple"

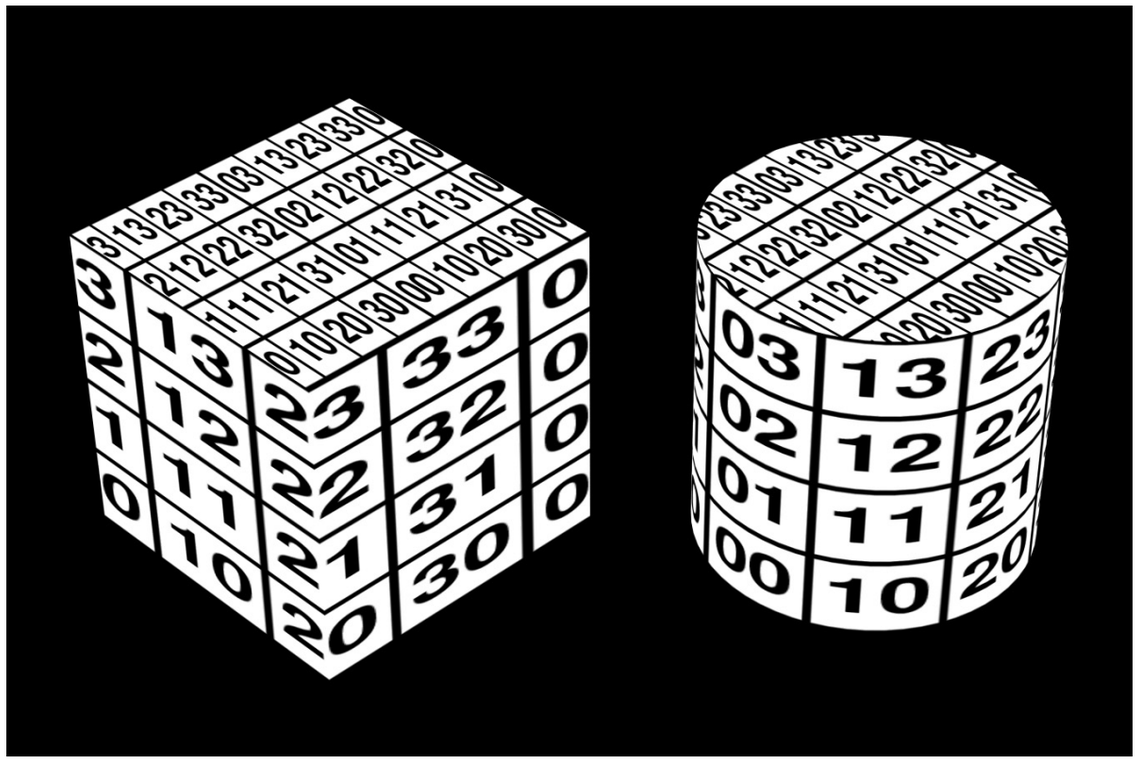
Color from functions

Manipulating the texture coordinates

```
declare shader
    color "texture_uv" (
        color texture "tex",
        scalar "u_scale"  default 1,
        scalar "v_scale"  default 1,
        scalar "u_offset" default 0,
        scalar "v_offset" default 0 )
end declare
```

Scene file declaration of shader "texture_uv"

Color from functions



Manipulating the texture coordinates

```
color texture "fourgrid" "fourgrid.tif"

material "grid"
    "texture_uv" (
        "tex" "fourgrid",
        "u_scale" 2,
        "u_offset" .125 )
end material
```

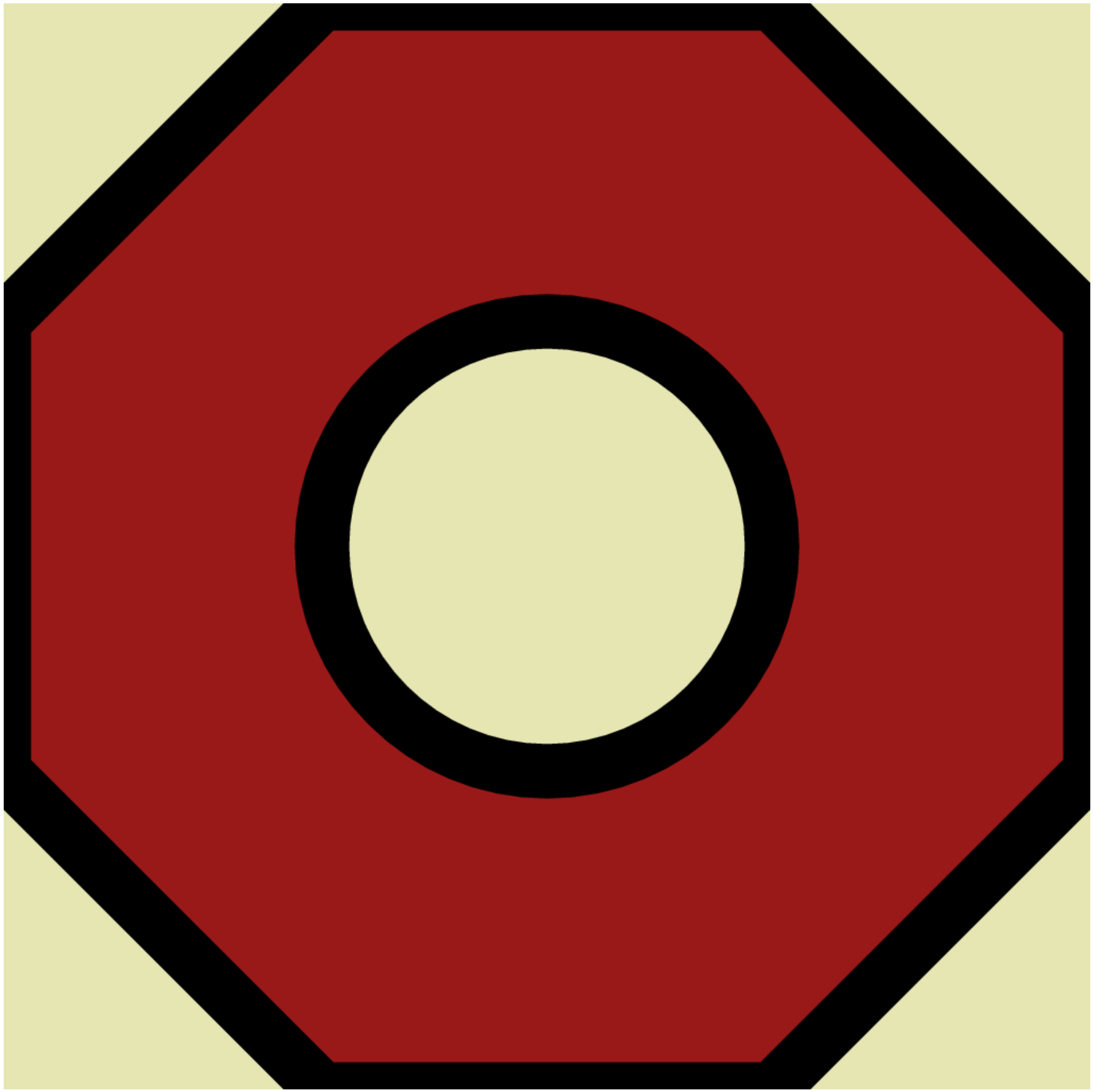
Using a texture map with an offset and scale to change its position on the object

Color from functions

Manipulating the texture coordinates

```
 1  struct texture_uv {
 2      miTag tex;
 3      miScalar u_scale;
 4      miScalar v_scale;
 5      miScalar u_offset;
 6      miScalar v_offset;
 7  };
 8
 9  miBoolean texture_uv(
10      miColor *result, miState *state, struct texture_uv *params)
11  {
12      miVector uv_coord = {0.0, 0.0, 0.0};
13      miScalar u_scale  = *mi_eval_scalar(&params->u_scale);
14      miScalar v_scale  = *mi_eval_scalar(&params->v_scale);
15      miScalar u_offset = *mi_eval_scalar(&params->u_offset);
16      miScalar v_offset = *mi_eval_scalar(&params->v_offset);
17
18      uv_coord.x = fmod((state->tex_list[0].x * u_scale) + u_offset, 1.0);
19      uv_coord.y = fmod((state->tex_list[0].y * v_scale) + v_offset, 1.0);
20
21      mi_lookup_color_texture(
22          result, state, *mi_eval_tag(&params->tex), &uv_coord);
23
24      return miTRUE;
25  }
```

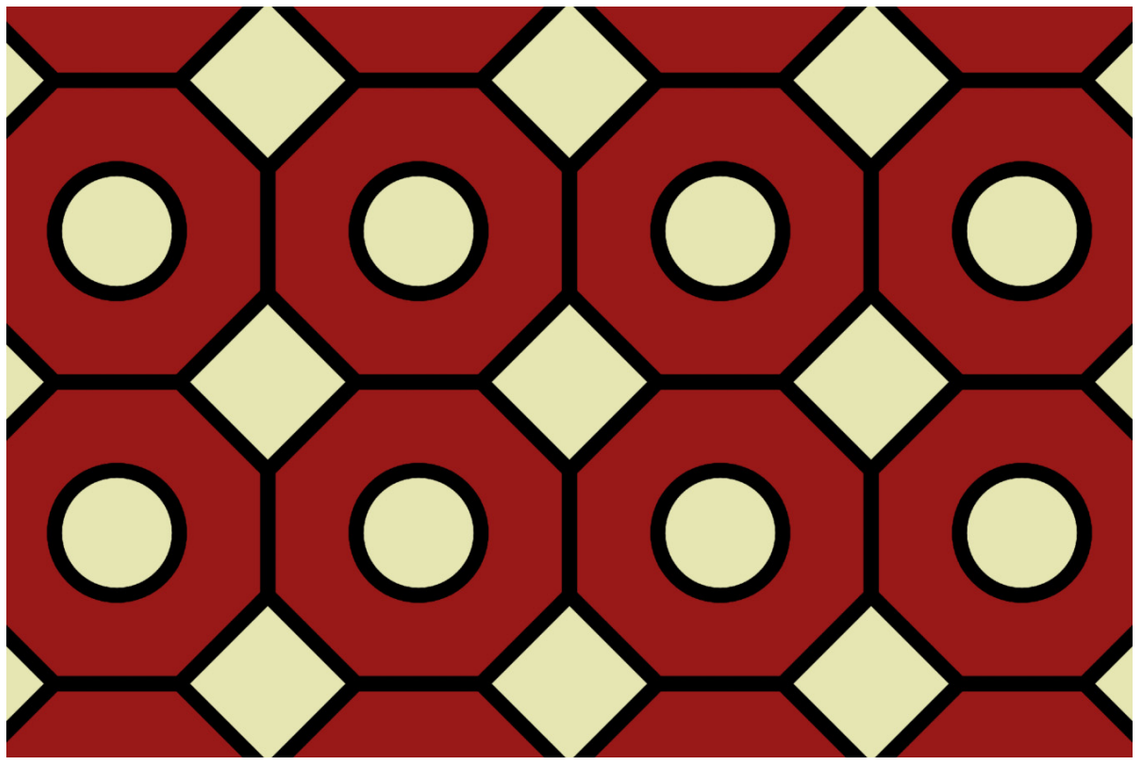
Source code of shader "texture_uv"



Texture map source image, with edges designed for tiling

Color from functions

Defining texture maps that tile well



```
color texture "octatile" "octatile.tif"

material "tile"
    "texture_uv" (
        "tex" "octatile",
        "u_scale" 4,
        "v_scale" 4 )
end material
```

An even tiling based on the design of the texture map image

Color from functions

Multiple texture spaces

```
declare shader
    color "vertex_color" (
        integer "uv_index" default 0 )
end declare
```

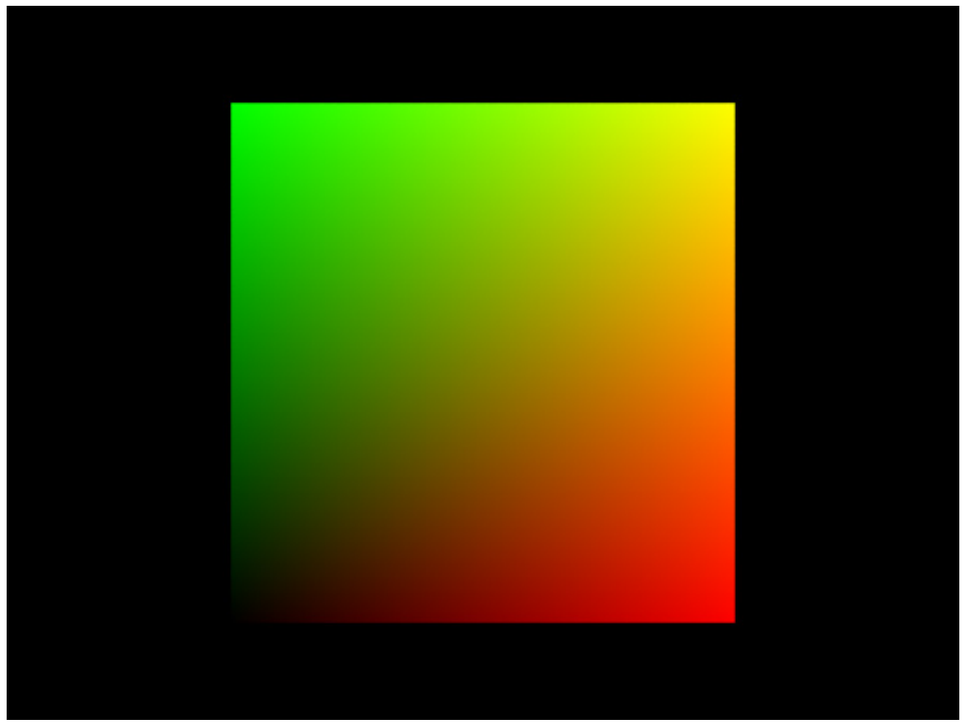
Scene file declaration of shader "vertex_color"

Color from functions

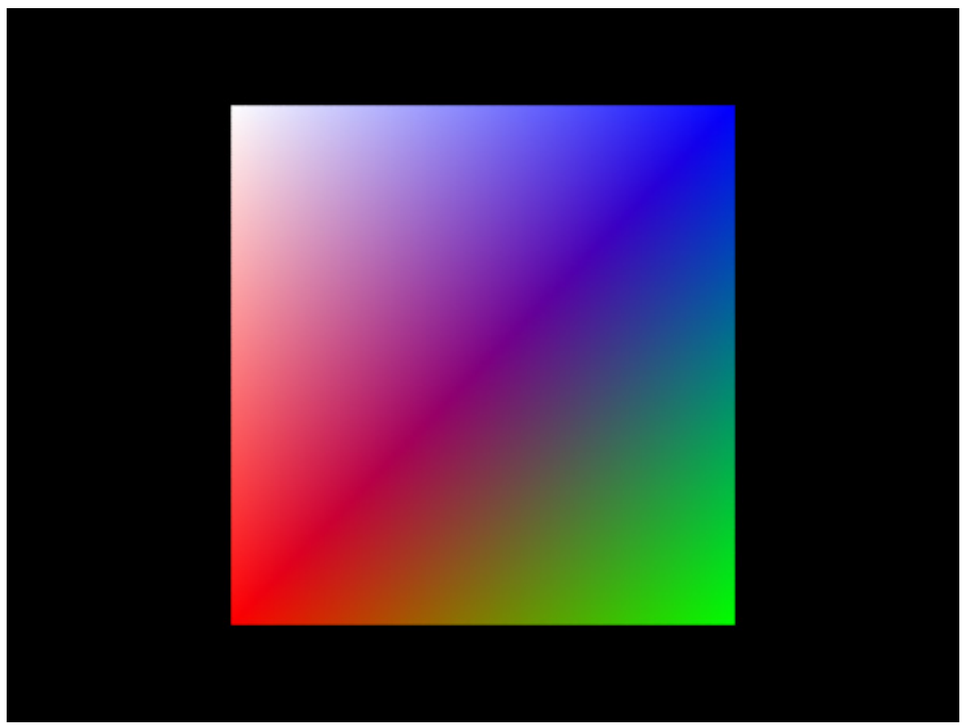

```
 1  struct vertex_color {
 2      miInteger uv_index;
 3  };
 4
 5  miBoolean vertex_color(
 6      miColor *result, miState *state, struct vertex_color *params)
 7  {
 8      miInteger uv_index = *mi_eval_integer(&params->uv_index);
 9
10      result->r = state->tex_list[uv_index].x;
11      result->g = state->tex_list[uv_index].y;
12      result->b = state->tex_list[uv_index].z;
13
14      return miTRUE;
15  }
```

Source code of shader "vertex_color"

Color from functions



texture_7.tif



texture_7_uv_1.tif

Multiple texture spaces

```
material "texture_space_0"
    "vertex_color" (
        "uv_index" 0 )
end material

material "texture_space_1"
    "vertex_color" (
        "uv_index" 1 )
end material
```

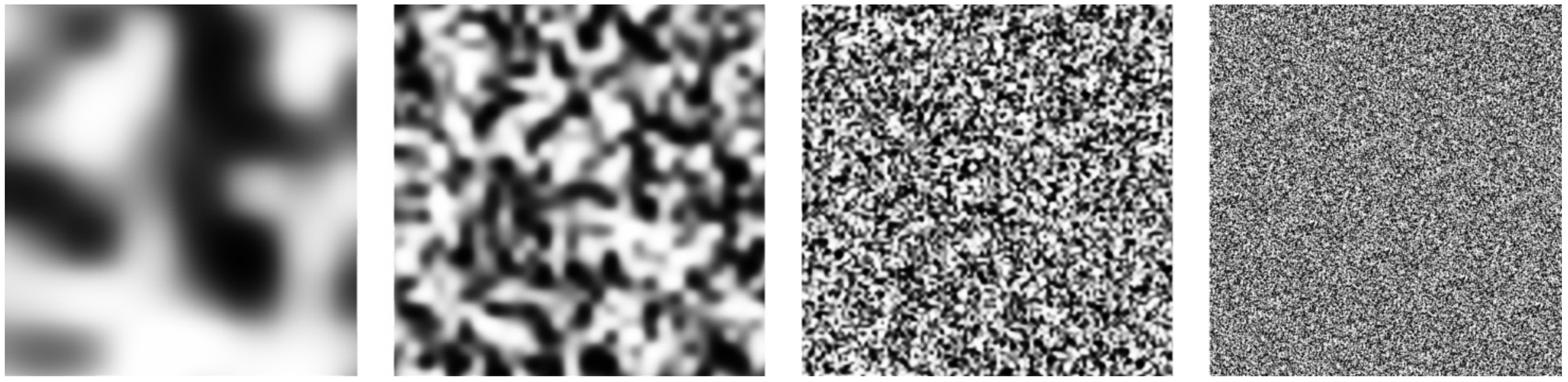
Selecting different texture spaces defined in the object

```
object "square"
    visible
    group
        # Coordinate data for XYZ position:
        -.5 -.5  0
         .5 -.5  0
         .5  .5  0
        -.5  .5  0
        # First set of texture coordinates
        # (see "Textures, Motion, Derivatives" in documentation of miState):
        0 0 0
        1 0 0
        1 1 0
        0 1 0
        # Second set of texture coordinates:
        1 0 0
        0 1 0
        0 0 1
        1 1 1
        # Following integers are indices into previous list of triplets:
        # "v" is vertex coordinate
        # "t" is tex_list array element, starting at 0
        v 0 t 4 t 8
        v 1 t 5 t 9
        v 2 t 6 t 10
        v 3 t 7 t 11
        # Define polygon; now integers refer to "v" definitions:
        p 0 1 2 3
    end group
end object
```

Multiple texture spaces defined in vertices with multiple ``t" attributes

Color from functions

Noise functions



Different scales of noise using the API function mi_unoise_3d

```
 1  double miaux_summed_noise (
 2      miVector *point,
 3      double summing_weight, double octave_scaling, int octave_count)
 4  {
 5      int i;
 6      double noise_value,
 7          noise_sum = 0.0, noise_scale = 1.0, maximum_noise_sum = 0.0;
 8      miVector scaled_point;
 9      miaux_set_vector(&scaled_point, point->x, point->y, point->z);
10
11      for (i = 0; i < octave_count; i++) {
12          noise_value = mi_unoise_3d(&scaled_point);
13          noise_sum += noise_value / noise_scale;
14          maximum_noise_sum += 1.0 / noise_scale;
15          noise_scale *= summing_weight;
16          miaux_scale_vector(&scaled_point, octave_scaling);
17      }
18      return noise_sum/maximum_noise_sum;
19  }
```

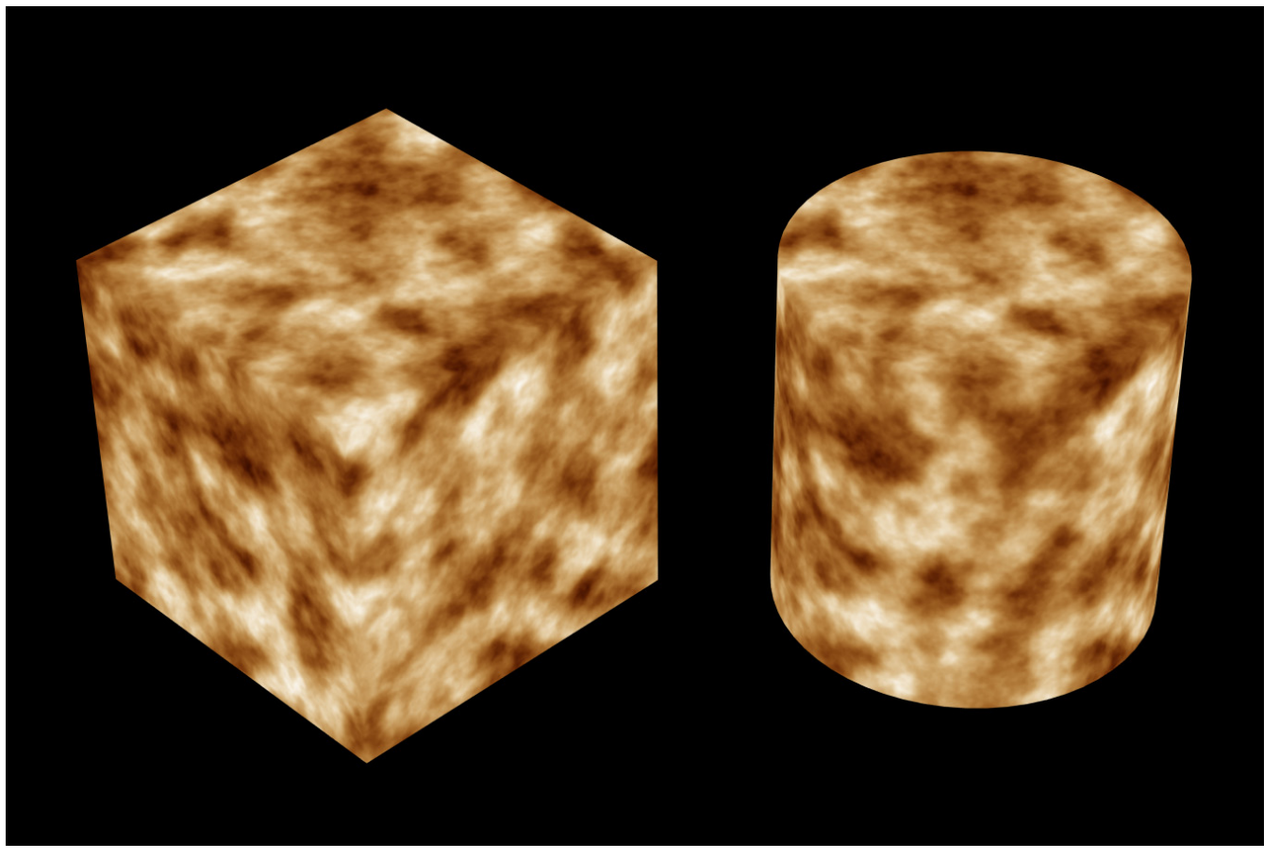
Auxiliary function: miaux_summed_noise

```
declare shader
    color "summed_noise_color" (
        scalar "point_scale" default 1,
        scalar "octave_scaling" default 2,
        scalar "summing_weight" default 2,
        integer "number_of_octaves" default 5,
        scalar "red_exponent" default 1,
        scalar "green_exponent" default 1,
        scalar "blue_exponent" default 1 )
end declare
```

Scene file declaration of shader "summed_noise_color"

Color from functions

Noise functions



```
material "summed_noise"
    "summed_noise_color" (
        "point_scale" 10,
        "red_exponent" .5,
        "green_exponent" 1,
        "blue_exponent" 2 )
end material
```

Using noise to create textures based on object coordinates

Color from functions


```
 1  struct summed_noise_color {
 2      miScalar point_scale;
 3      miScalar octave_scaling;
 4      miScalar summing_weight;
 5      miInteger number_of_octaves;
 6      miScalar red_exponent;
 7      miScalar green_exponent;
 8      miScalar blue_exponent;
 9  };
10
11  miBoolean summed_noise_color(
12      miColor *result, miState *state, struct summed_noise_color *params)
13  {
14      miScalar noise_sum;
15      miScalar red_exponent = *mi_eval_scalar(&params->red_exponent);
16      miScalar green_exponent = *mi_eval_scalar(&params->green_exponent);
17      miScalar blue_exponent = *mi_eval_scalar(&params->blue_exponent);
18      miVector object_point;
19      mi_point_to_object(state, &object_point, &state->point);
20      mi_vector_mul(&object_point, *mi_eval_scalar(&params->point_scale));
21
22      noise_sum =
23          miaux_summed_noise(&object_point,
24                             *mi_eval_scalar(&params->summing_weight),
25                             *mi_eval_scalar(&params->octave_scaling),
26                             *mi_eval_integer(&params->number_of_octaves));
27      result->r =
28          red_exponent == 1 ? noise_sum : pow(noise_sum, red_exponent);
29      result->g =
30          green_exponent == 1 ? noise_sum : pow(noise_sum, green_exponent);
31      result->b =
32          blue_exponent == 1 ? noise_sum : pow(noise_sum, blue_exponent);
33
34      return miTRUE;
35  }
```

Source code of shader "summed_noise_color"

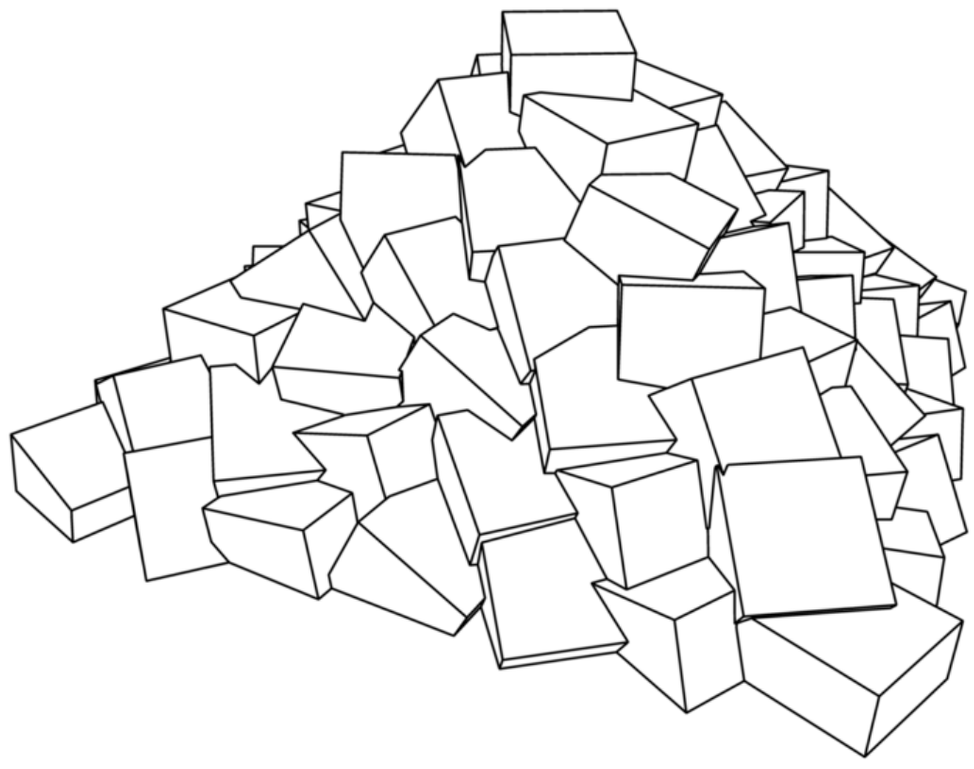
Exercise 9: Texture shaders

1. Copy `texture_1.mi` to `texture.mi`, change output to `texture.tif`
2. Render `texture.mi` and view `texture.tif`
3. Change to use `show_uv_steps` shader and re-render.
4. Modify the parameters to `show_uv_steps` and re-render.
5. Add `fourgrid` color texture to texture.mi — refer to `color texture` statement in `texture_3.mi`
6. Change to use `texture_uv` shader and re-render.
7. Try `octatile.tif` texture.
8. Look up the definition of `imf_copy` in the appendices in the HTML documentation.
9. Use `imf_copy` to make a `.map` file.

The color of edges

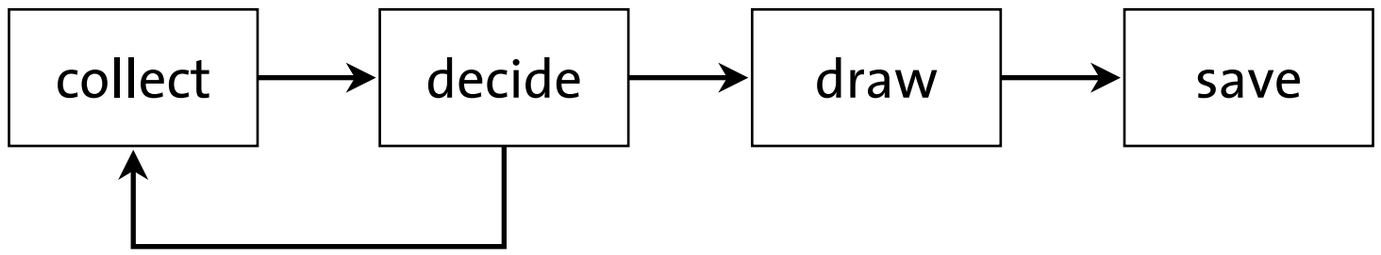
The color of edges

The color of edges



Scene rendered with shader front_bright and with contours

The color of edges



A schematic overview of the four contour shading phases

Contour questions answered by shaders

Contour questions answered by shaders

1. *Store* – What data will be necessary when deciding if a contour should be drawn?

Contour questions answered by shaders

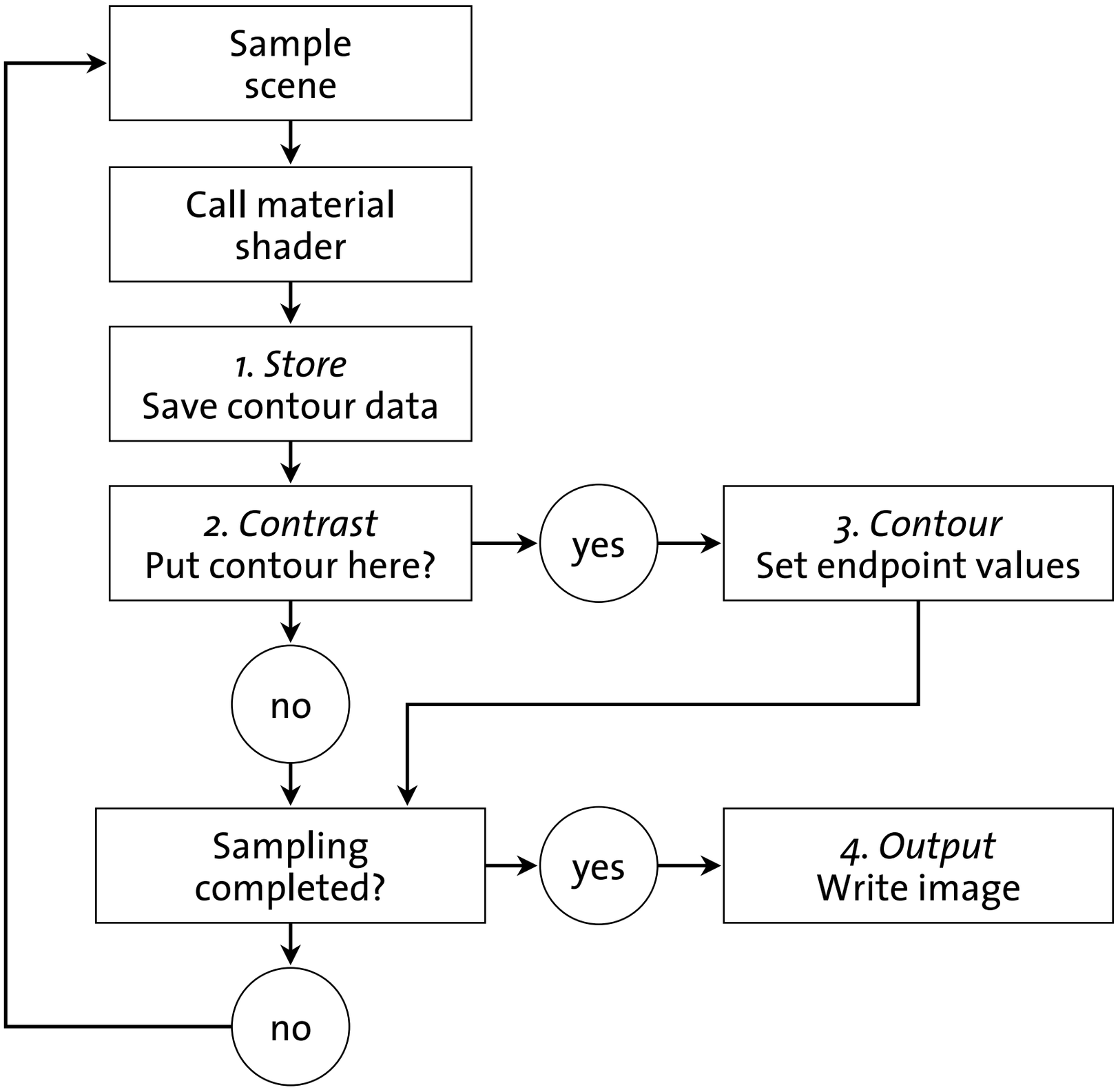
1. *Store* – What data will be necessary when deciding if a contour should be drawn?
2. *Contrast* – Has enough data been collected to draw a contour?

Contour questions answered by shaders

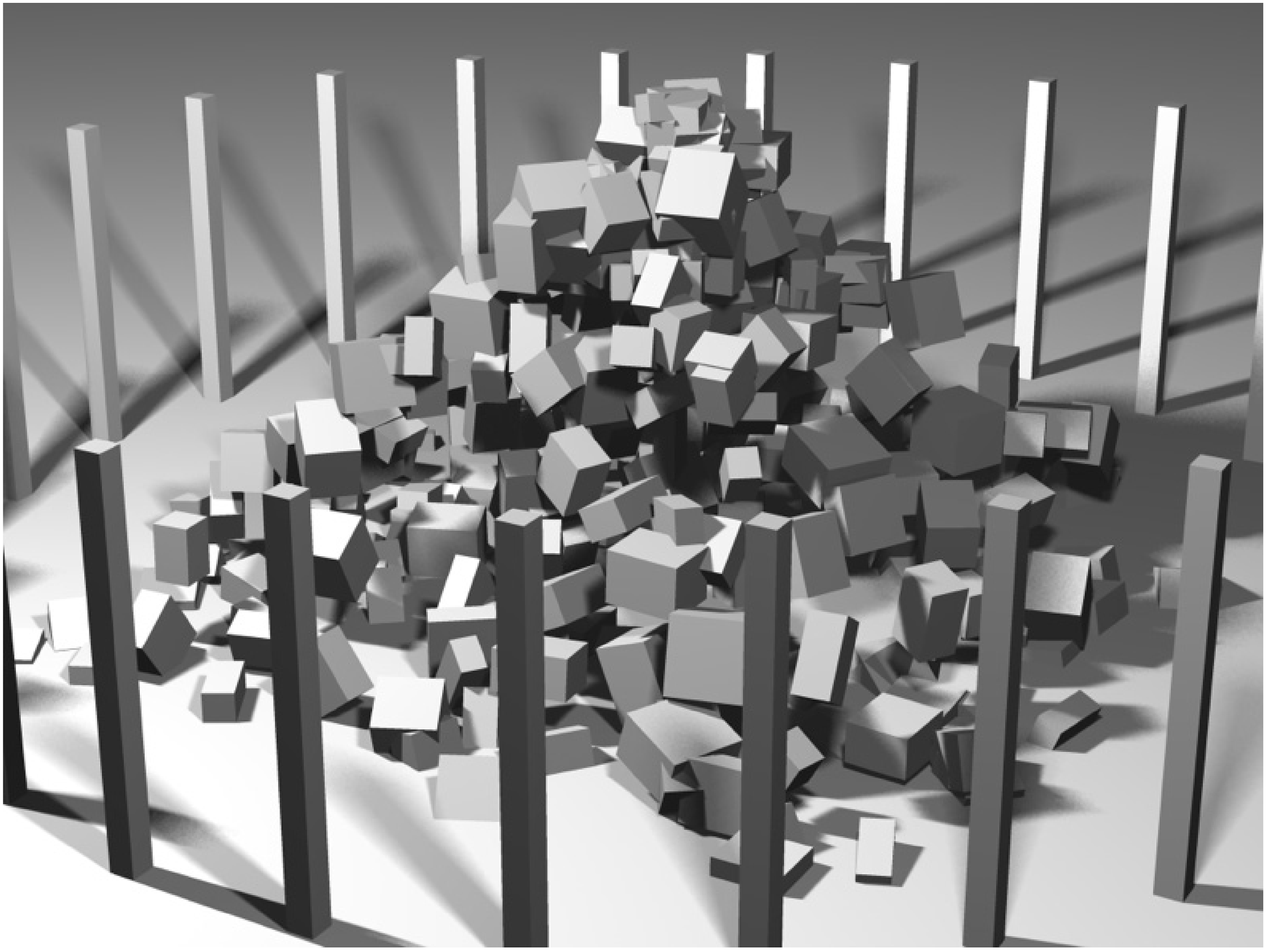
1. *Store* – What data will be necessary when deciding if a contour should be drawn?
2. *Contrast* – Has enough data been collected to draw a contour?
3. *Contour* – How should the contour be drawn?

Contour questions answered by shaders

1. *Store* – What data will be necessary when deciding if a contour should be drawn?
2. *Contrast* – Has enough data been collected to draw a contour?
3. *Contour* – How should the contour be drawn?
4. *Output* – How should the contours be used in creating the output image?



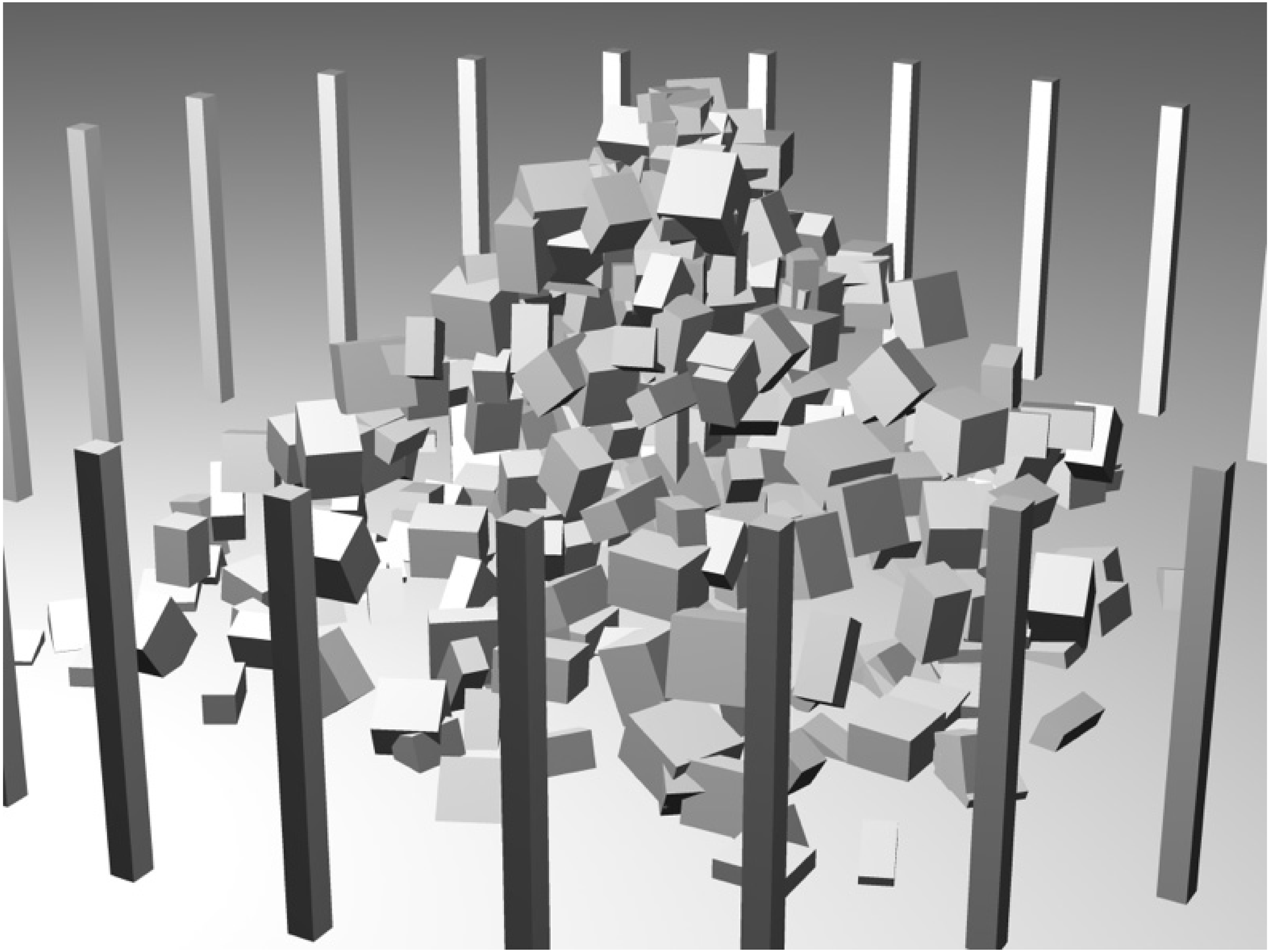
Order of execution and interaction of the four contour shaders



Scene with shadows

The color of edges

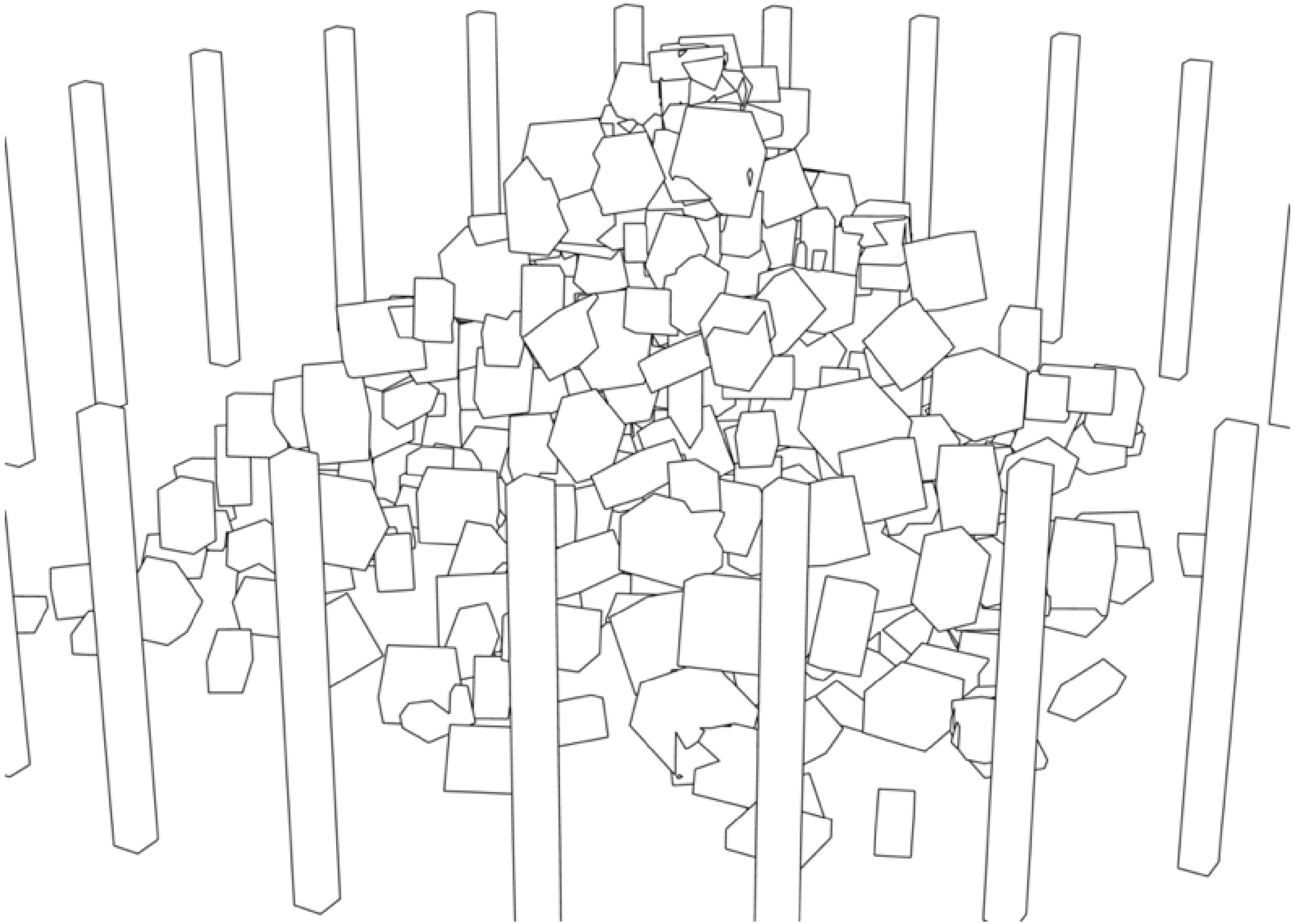
The four contour shader types



Scene without shadows

The color of edges

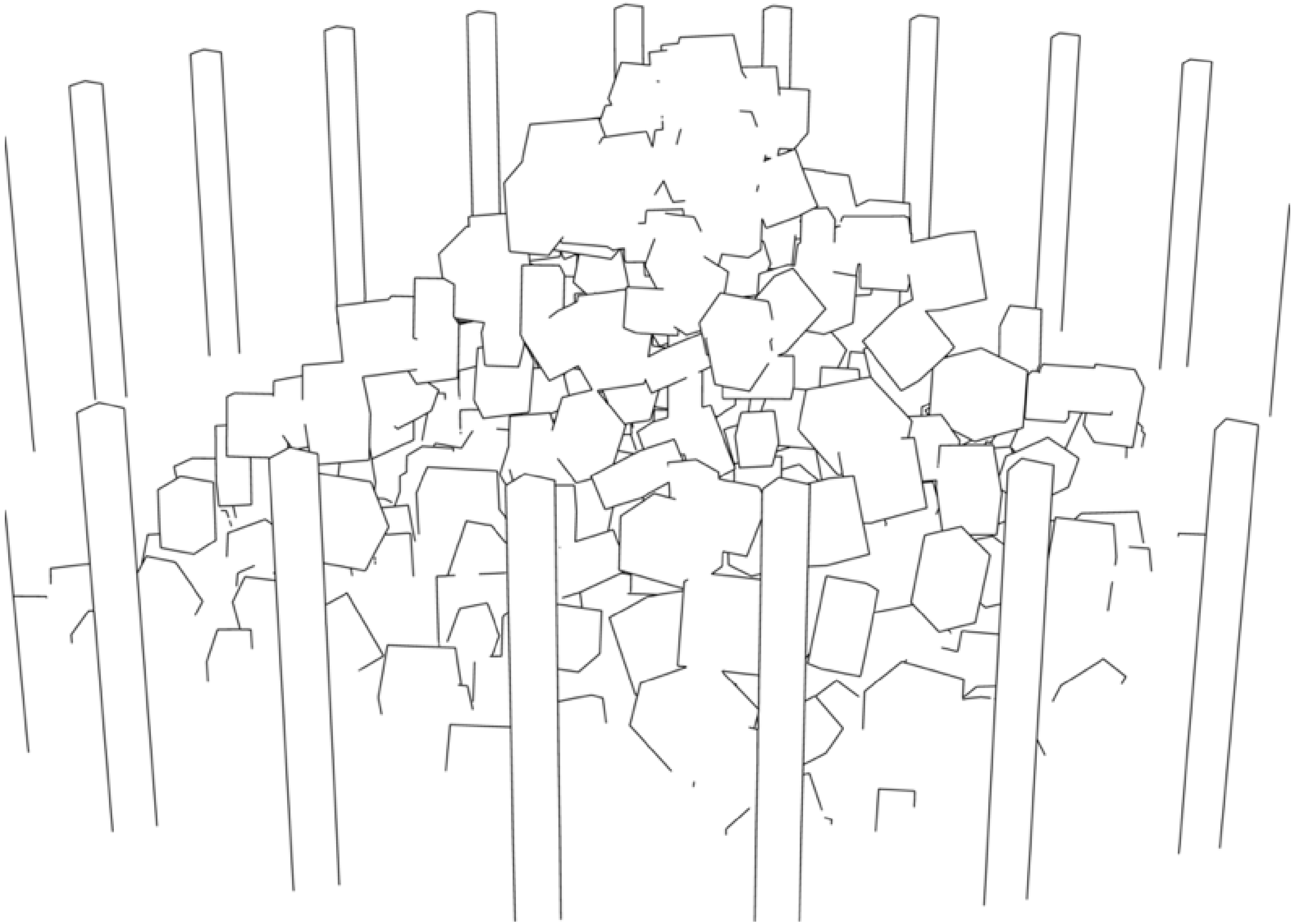
The four contour shader types



Contour shader: object boundaries

The color of edges

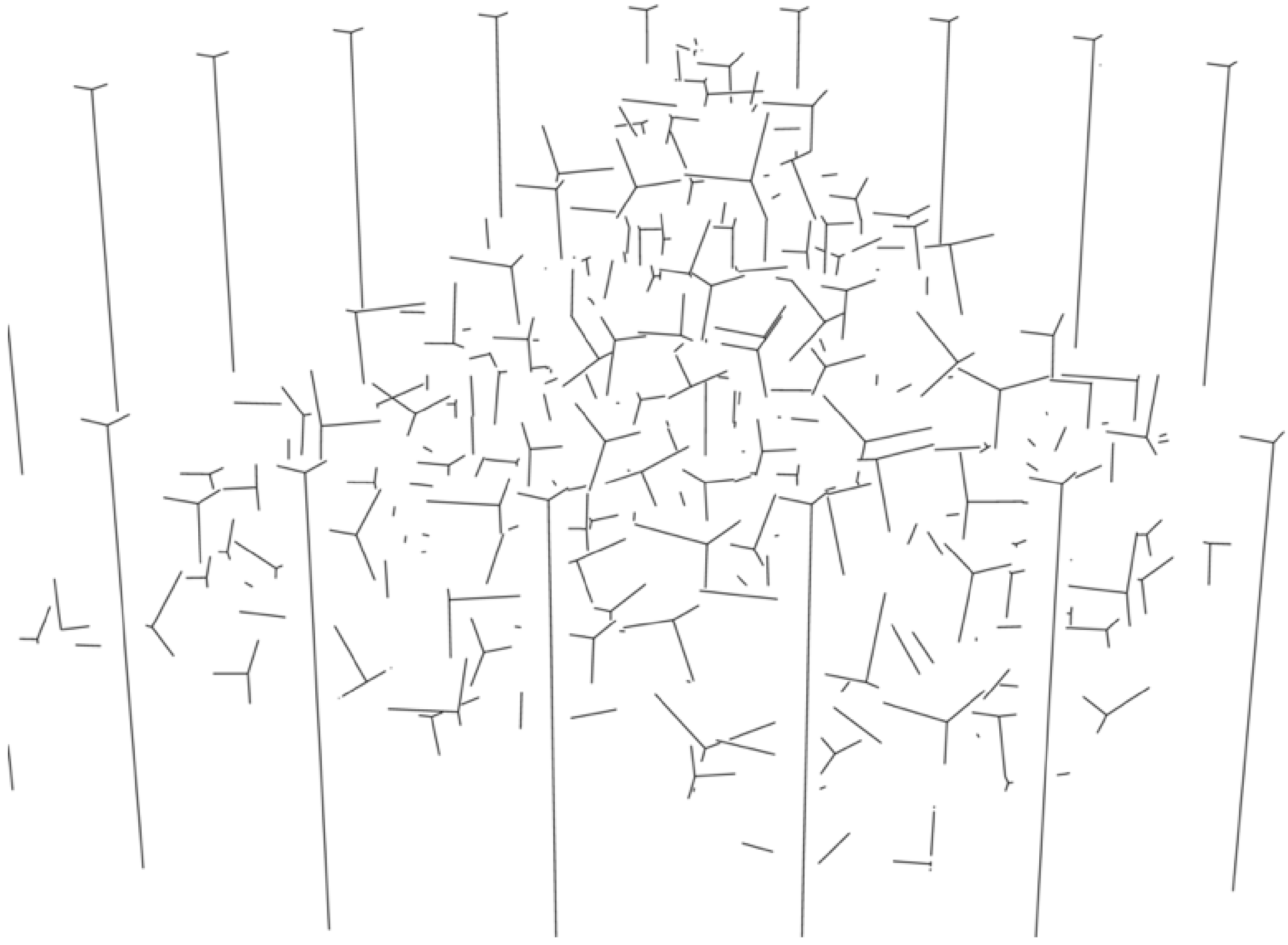
The four contour shader types



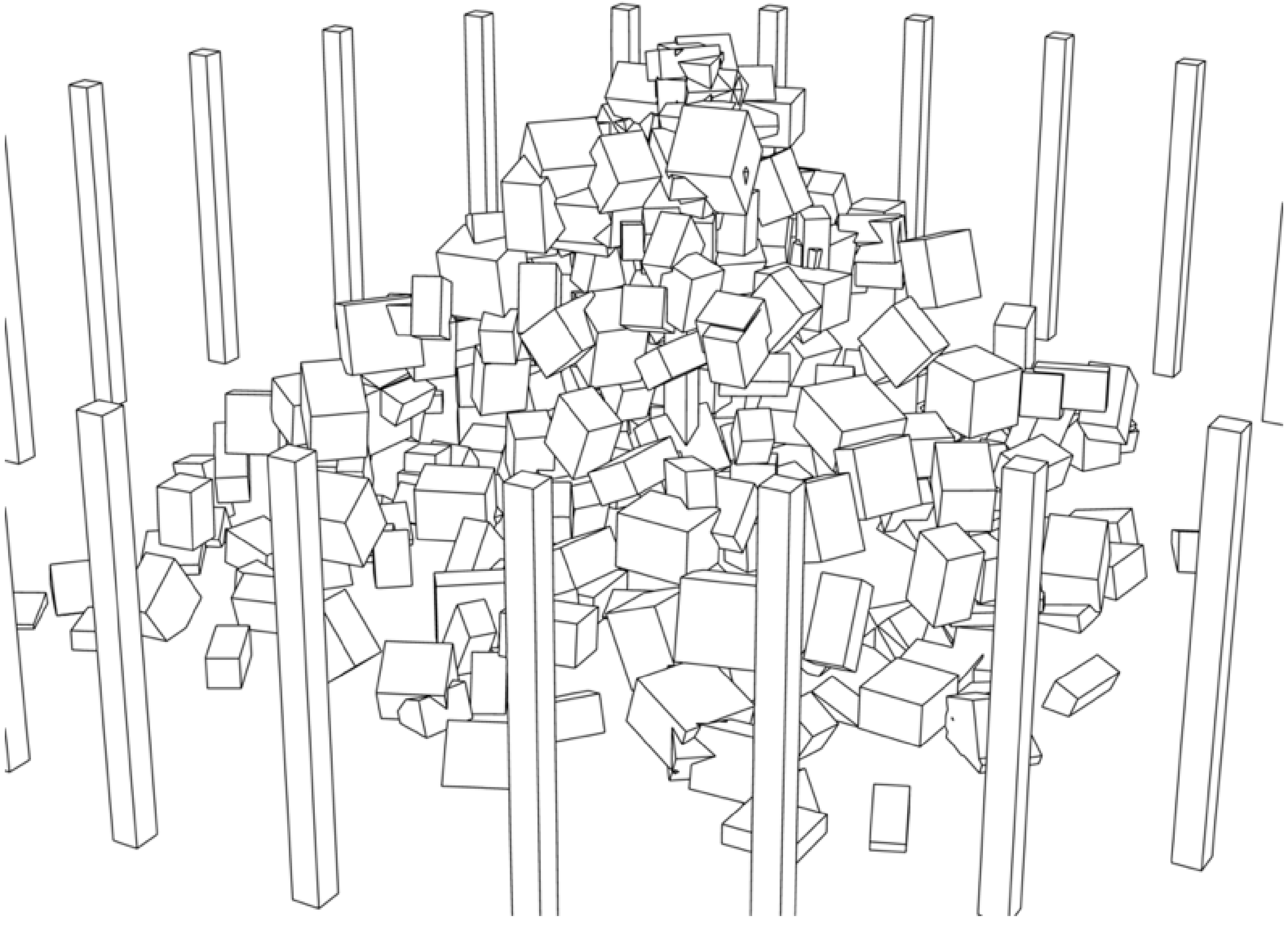
Contour shader: silhouette edges

The color of edges

The four contour shader types



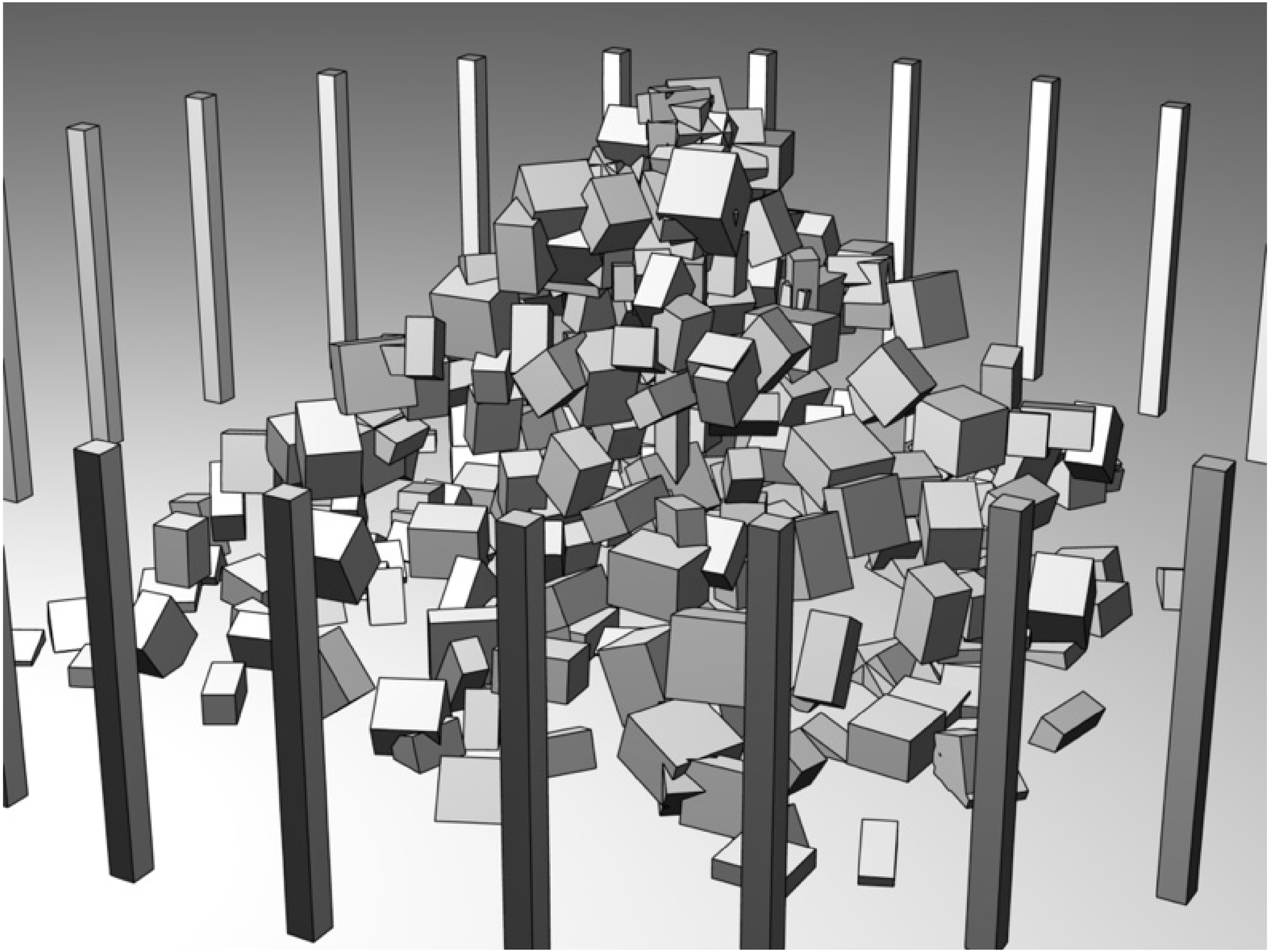
Contour shader: internal object edges



Contour shader: all edges

The color of edges

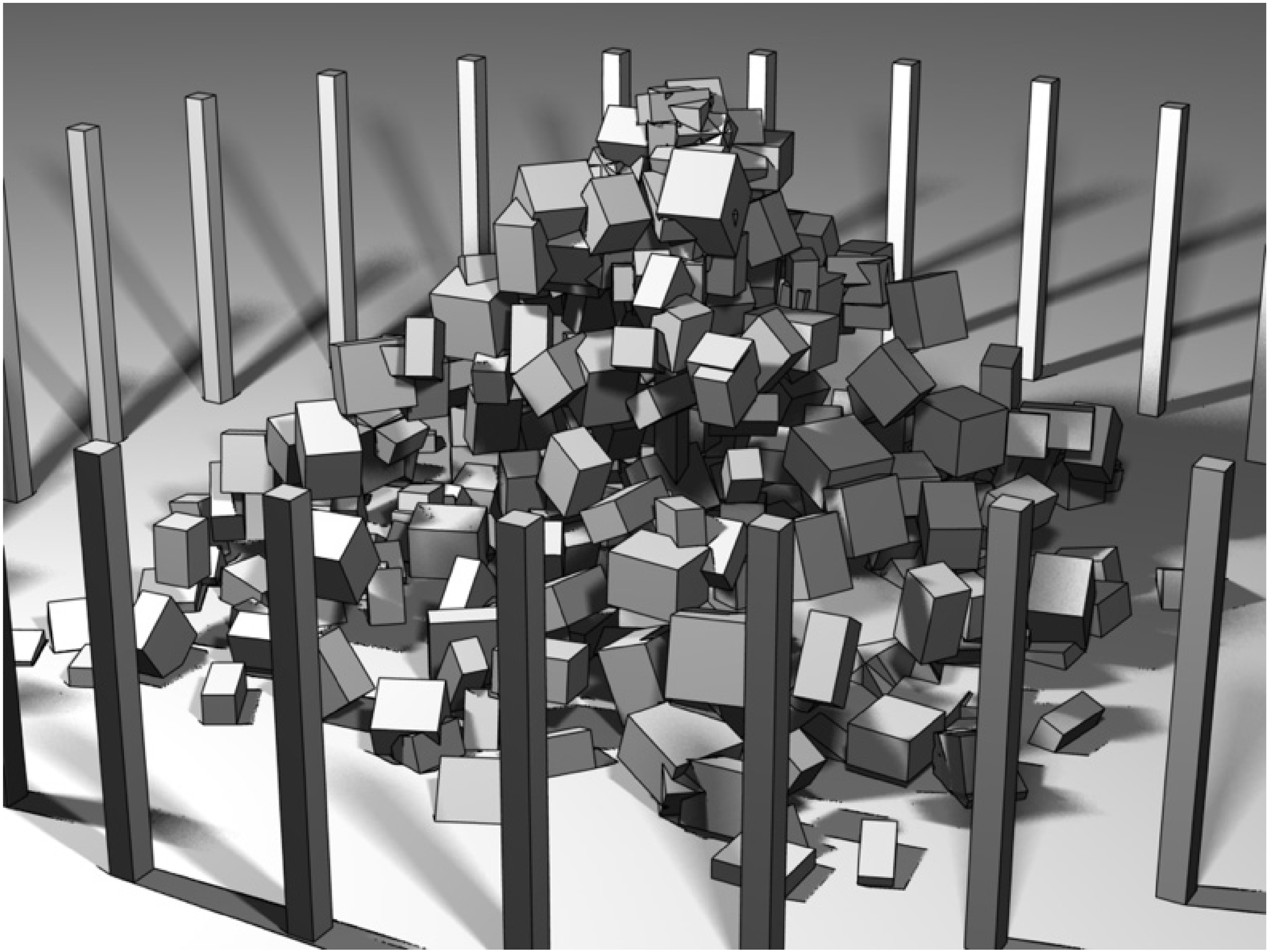
The four contour shader types



Contours composited with scene

The color of edges

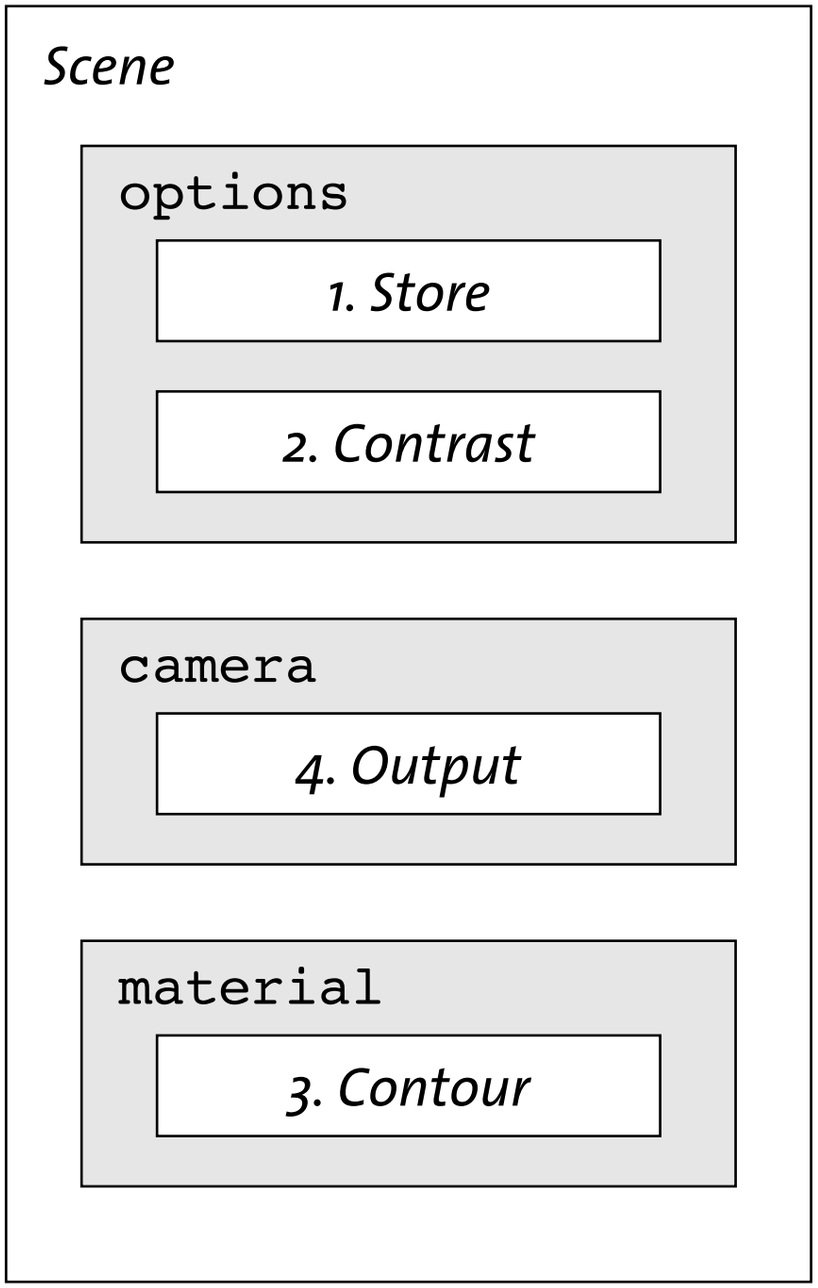
The four contour shader types



Contours composited with scene

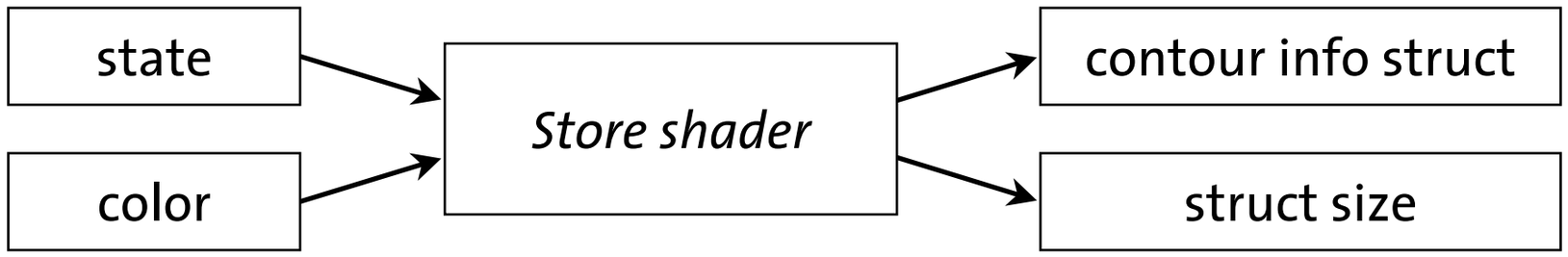
The color of edges

Location of the contour shaders in the scene file




```
options "opt"
    object space
    samples 0 3
    contour store "c_store" ()
    contour contrast "c_contrast" ()
end options

camera "cam"
    output "contour" "c_output" ()
    focal       32
    aperture    33.3
    aspect      1.33333
    resolution 400 300
end camera

material "outline"
    contour "c_contour" ( "width" .2 )
end material
```

Location of the contour shaders in the scene file



Store shader

```
struct contour_info {
    miTag    instance;
    miVector normal;
};
```

An example C struct for a Store shader

The color of edges

The Store shader: defining and saving contour data

```
struct contour_info {
    miTag    instance;
    miVector normal;
};
```

```
declare shader
    struct { geometry "instance",
             vector   "normal" }
        "c_store" ()
end declare
```

Specifying a struct result type in C as the return value for a shader declaration in .mi syntax

The color of edges

The Store shader: defining and saving contour data

```
declare shader
    struct { geometry "instance",
             vector   "normal" }
        "c_store" ()
end declare
```

Scene file declaration of shader "c_store"

The color of edges

The Store shader: defining and saving contour data

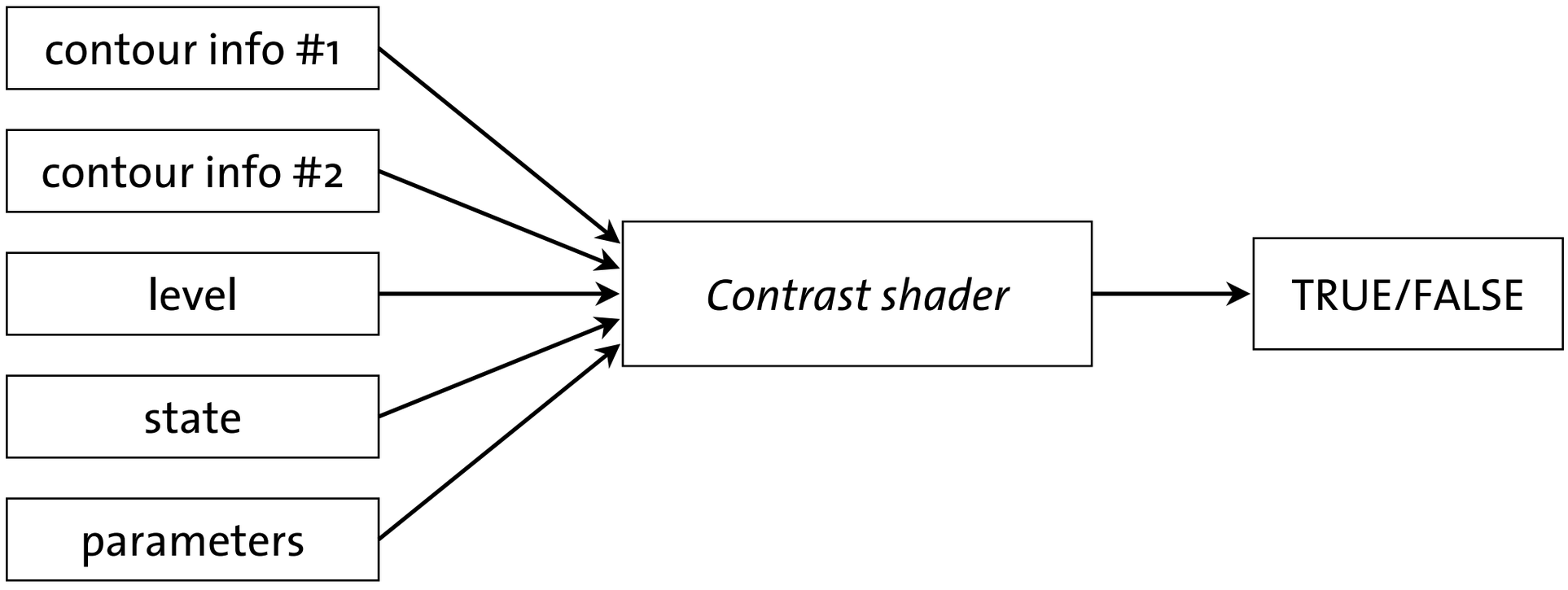
```
#ifndef _CONTOUR_INFO_H_
#define _CONTOUR_INFO_H_

#include "shader.h"

typedef struct {
    miTag    instance;
    miVector normal;
} contour_info;

#endif
```

File contour_info.h

```
 1  miBoolean c_store (
 2      void     *info_pointer,
 3      int      *info_size,
 4      miState  *state,
 5      miColor  *color)
 6  {
 7      contour_info *info = (contour_info*)info_pointer;
 8
 9      info->instance = state->instance;
10      info->normal = state->normal;
11      *info_size = sizeof(contour_info);
12
13      return miTRUE;
14  }
```

Source code of shader "c_store"

The Contrast shader: determining the location of contours



Contrast shader

The color of edges

The Contrast shader: determining the location of contours

```
declare shader
    "c_contrast" (
        scalar "dot_threshold" default 0.71, )
end declare
```

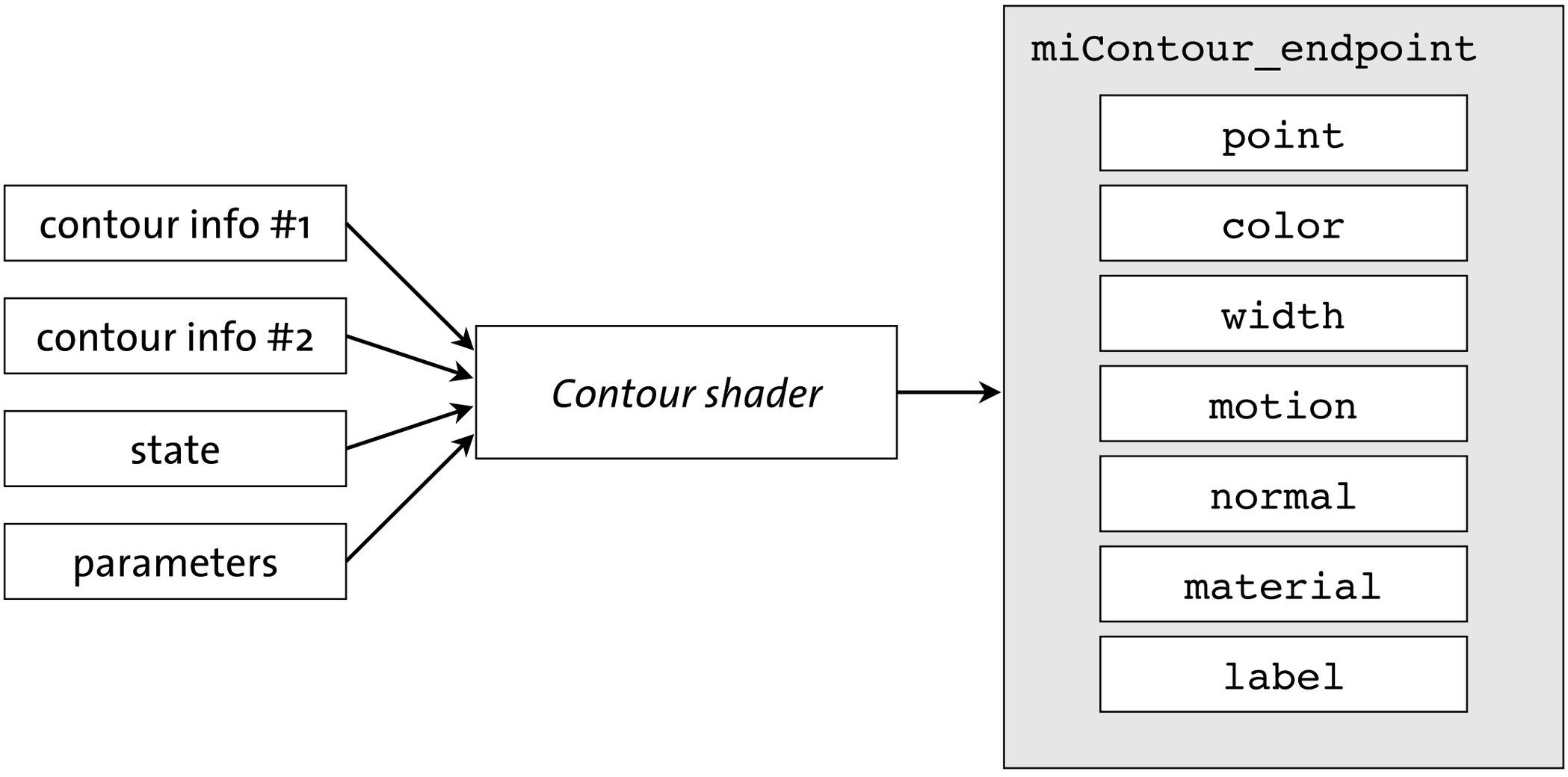
Scene file declaration of shader "c_contrast"

```
 1  struct c_contrast {
 2      miScalar dot_threshold;
 3  };
 4
 5  miBoolean c_contrast(
 6      contour_info      *info1,
 7      contour_info      *info2,
 8      int               level,
 9      miState           *state,
10      struct c_contrast *params )
11  {
12      if (info1 == NULL ||
13          info2 == NULL ||
14          info1->instance != info2->instance ||
15          (mi_vector_dot(&info1->normal, &info2->normal) <
16           *mi_eval_scalar(&params->dot_threshold)))
17          return miTRUE;
18      else
19          return miFALSE;
20  }
```

Source code of shader "c_contrast"

The color of edges

The Contour shader: defining the properties of contours



Contour shader

The color of edges

The Contour shader: defining the properties of contours

```
typedef struct {
    miVector  point;
    miColor   color;
    float     width;
    miVector  motion;
    miVector  normal;
    miTag     material;
    int       label;
} miContour_endpoint;
```

C struct defined by mental ray in shader.h for specifying contour data

The color of edges

The Contour shader: defining the properties of contours

```
declare shader
    "c_contour" (
        color "color" default 0 0 0 1,
        scalar "width" default 1 )
end declare
```

Scene file declaration of shader "c_contour"

The color of edges

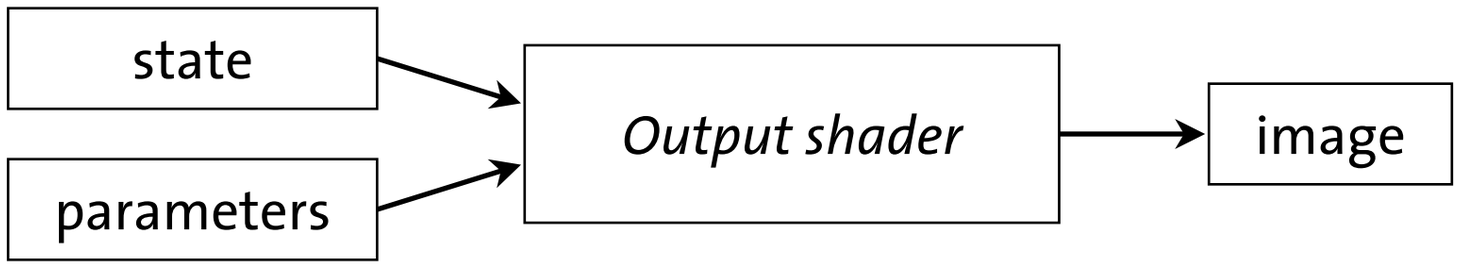
The Contour shader: defining the properties of contours

```
 1  struct c_contour {
 2      miColor color;
 3      miScalar width;
 4  };
 5
 6  miBoolean c_contour(
 7      miContour_endpoint *result,
 8      contour_info       *info_near,
 9      contour_info       *info_far,
10      miState            *state,
11      struct c_contour   *params)
12  {
13      result->color = *mi_eval_color(&params->color);
14      result->width = *mi_eval_scalar(&params->width);
15      return miTRUE;
16  }
```

Source code of shader "c_contour"

The color of edges

The Output shader: writing the contour image



Output shader

```
declare shader
    "c_output" (
        string "postscript_filename" )
end declare
```

Scene file declaration of shader "c_output"

```
1  char* miaux_tag_to_string(miTag tag, char *default_value)
2  {
3      char *result = default_value;
4      if (tag != 0) {
5          result = (char*)mi_db_access(tag);
6          mi_db_unpin(tag);
7      }
8      return result;
9  }
```

Auxiliary function: miaux_tag_to_string

The color of edges

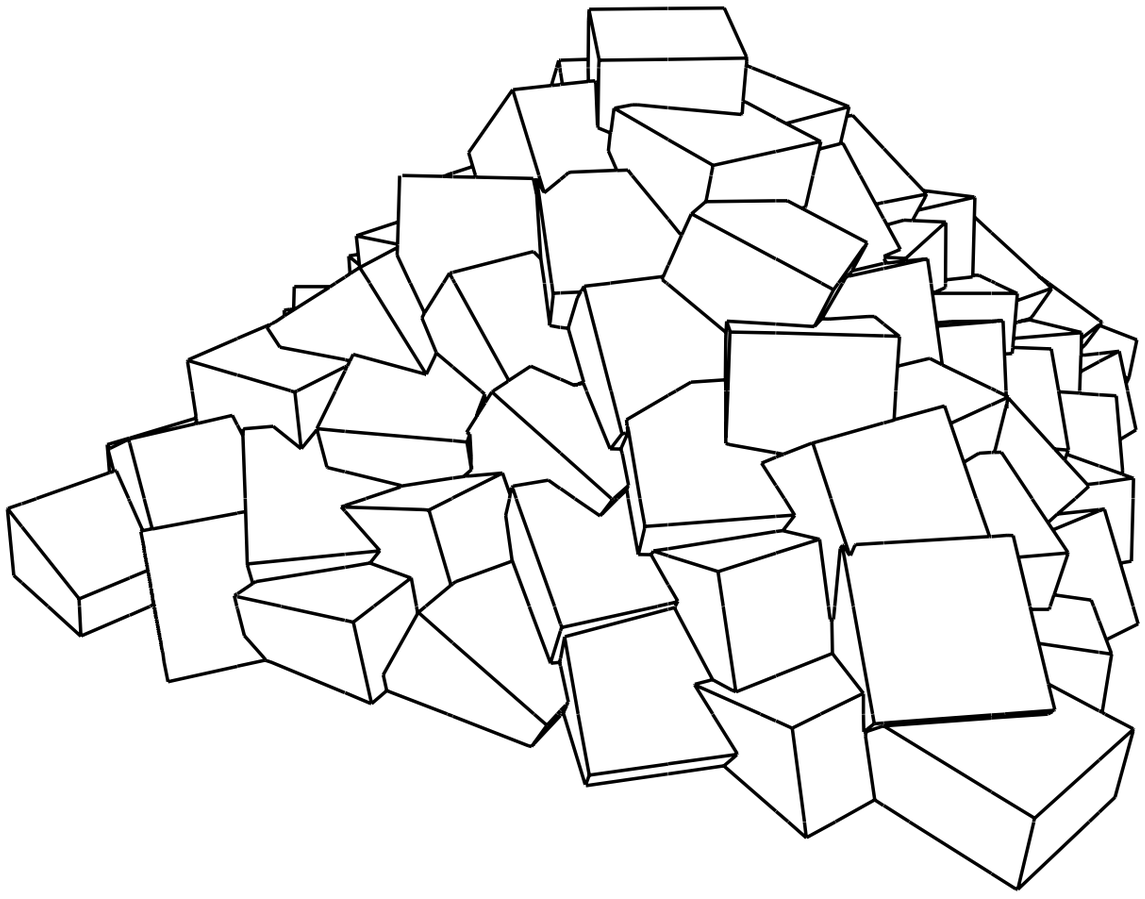
The Output shader: writing the contour image

```
 1  struct c_output {
 2      miTag postscript_filename;
 3  };
 4
 5  miBoolean c_output(
 6      miColor *result, miState *state, struct c_output *params)
 7  {
 8      miContour_endpoint p1;
 9      miContour_endpoint p2;
10
11      FILE* fp;
12      char* postscript_filename =
13          miaux_tag_to_string(*mi_eval_tag(&params->postscript_filename), NULL);
14      if (postscript_filename == NULL)
15          postscript_filename = "contour_output.ps";
16
17      mi_progress("Writing contour PostScript data to file %s",
18                  postscript_filename);
19
20      fp = fopen(postscript_filename, "w");
21      fprintf(fp, "%%!\n");
22      fprintf(fp, "%%%%BoundingBox: 0 0 %d %d\n",
23             state->camera->x_resolution, state->camera->y_resolution);
24
25      while (mi_get_contour_line(&p1, &p2))
26          fprintf(fp, "%g %g moveto %g %g lineto stroke\n",
27                 p1.point.x, p1.point.y,
28                 p2.point.x, p2.point.y);
29
30      fprintf(fp, "showpage\n");
31      fclose(fp);
32
33      return miTRUE;
34  }
```

Source code of shader "c_output"

The color of edges

Using the four contour shaders

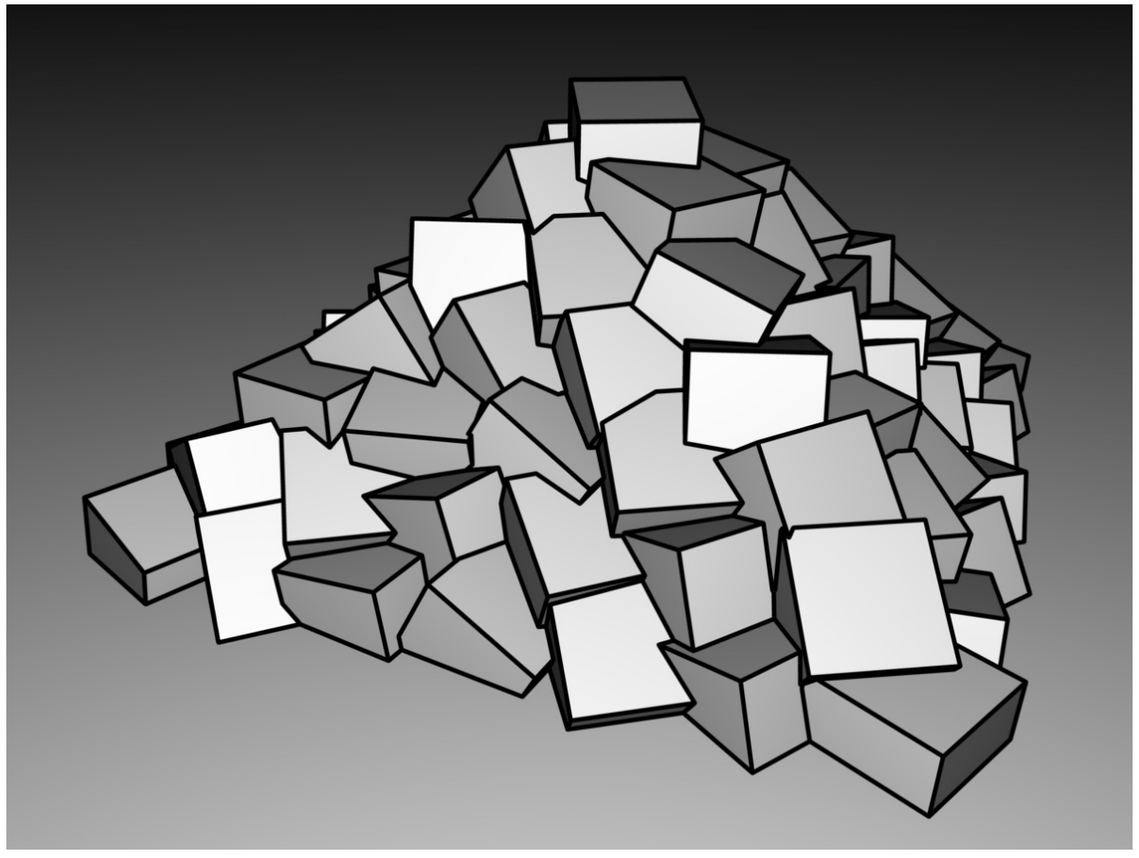


```
options "opt"
    object space
    samples 0 3
    contour store "c_store" ()
    contour contrast "c_contrast" ()
end options

camera "cam"
    output "contour" "c_output" ()
    focal 32
    aperture 33.3
    aspect 1.333
    resolution 400 300
end camera

material "outline"
    contour "c_contour" ( "width" .2 )
end material
```

PostScript file generated by four custom contour shaders

The color of edges

Compositing contours with the color from the material shader

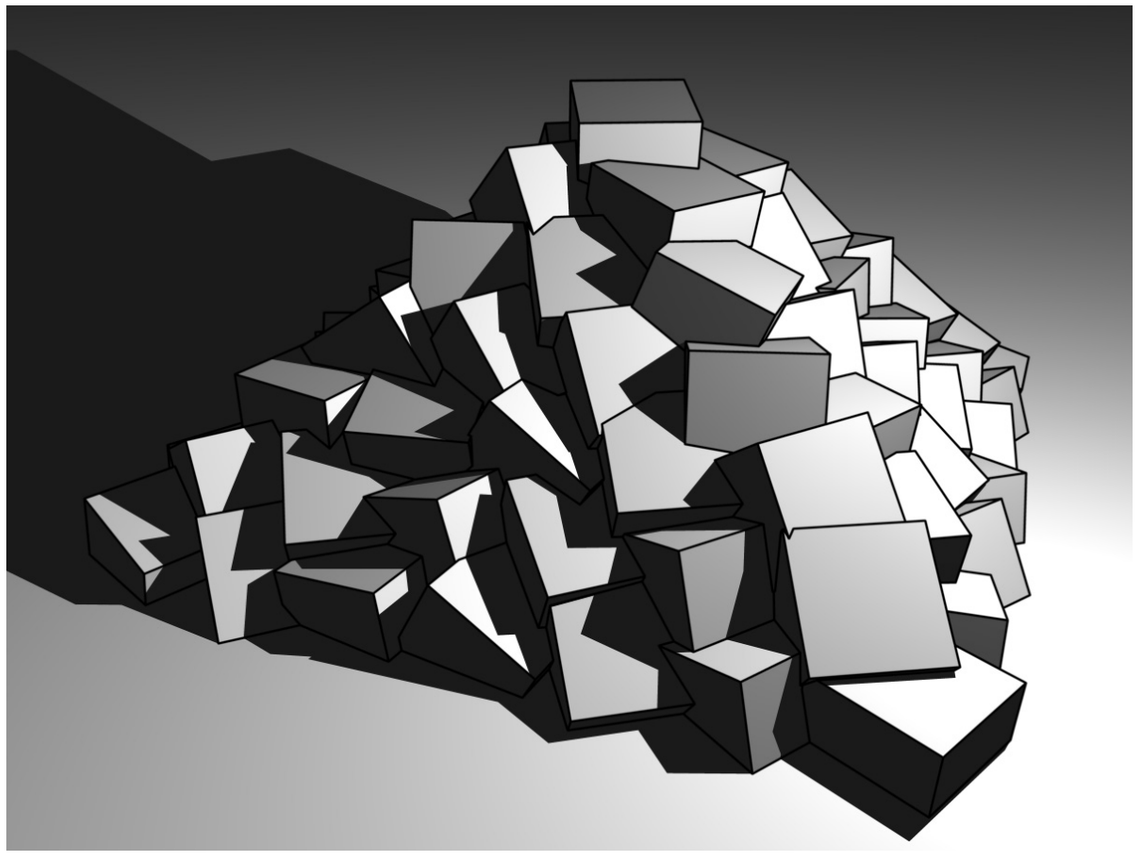


```
options "opt"
    object space
    samples 0 3
    contour store "c_store" ()
    contour contrast "c_contrast" ()
end options

camera "cam"
    output "contour,rgba"
        "contour_composite" ()
    output "rgba" "tif"
        "contours_3.tif"
    focal 32
    aperture 33.3
    aspect 1.333
    resolution 400 300
end camera

material "outline"
    "front_bright" ()
     contour "c_contour" ( "width" .4 )
end material
```

Compositing contours with the result of shader front_bright

The color of edges

Compositing contours with the color from the material shader



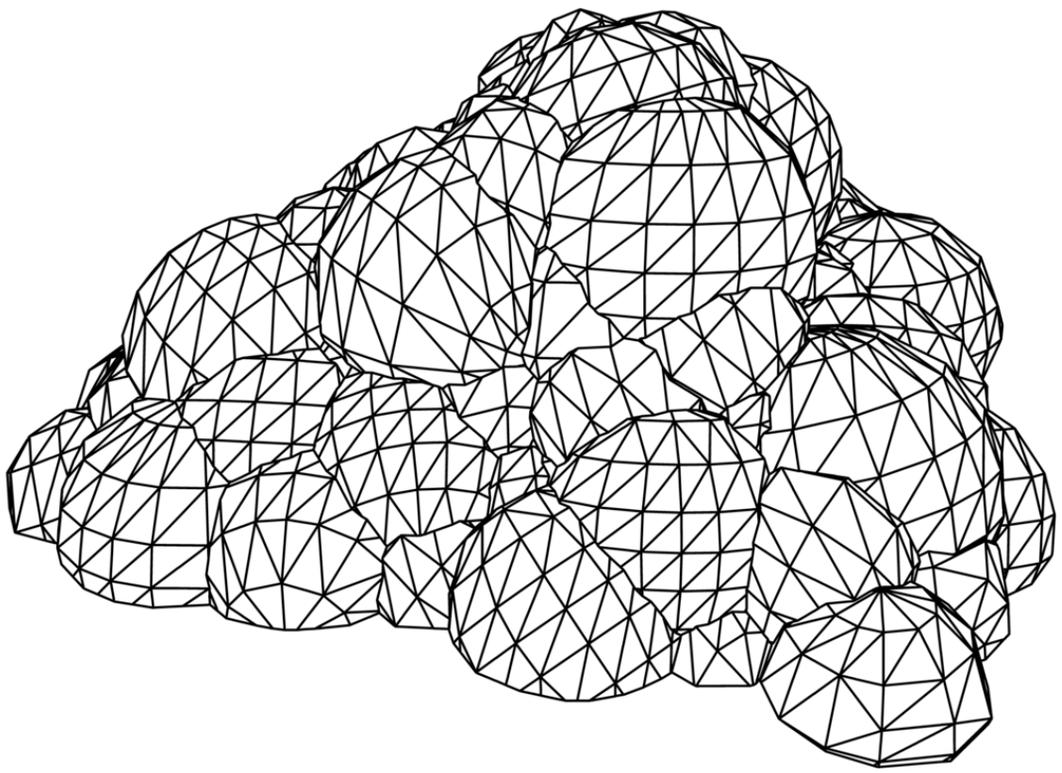
```
options "opt"
    object space
    samples 0 3
    contour store "c_store" ()
    contour contrast "c_contrast" ()
end options

camera "cam"
    output "contour,rgba"
        "contour_composite" ()
    output "rgba" "tif"
        "contours_4.tif"
    focal 32
    aperture 33.3
    aspect 1.333
    resolution 400 300
end camera

material "outline"
    "lambert" (
        "diffuse" 1 1 1,
        "ambient" .1 .1 .1,
        "lights" ["light_instance"] )
     contour "c_contour" ( "width" .2 )
end material
```

Compositing contours with the result of shader lambert with shadows

```
declare shader
    struct { integer "prim_index" }
        "c_tessellate_store" ()
end declare

declare shader
    "c_tessellate_contrast" ()
end declare

declare shader
    "c_tessellate_contour" (
        color "color" default 0 0 0 1,
        scalar "width" default 1 )
end declare
```

Scene file declaration of shader "c_tessellate"

The color of edges



Displaying tessellation with contour shaders

```
options "opt"
    object space
    samples 0 2
    contour store
        "c_tessellate_store" ()
    contour contrast
        "c_tessellate_contrast" ()
    approximate fine view length 20 all
end options

camera "cam"
    output "contour,rgba"
        "contour_composite" ()
    output "rgba" "tif"
        "contours_5.tif"
    focal 32
    aperture 33.3
    aspect 1.333
    resolution 400 300
end camera

material "outline"
    "one_color" (
        "color" 1 1 1 )
    contour
        "c_tessellate_contour" (
        "width" .2 )
end material
```

Defining contours to show the tessellation of object instances

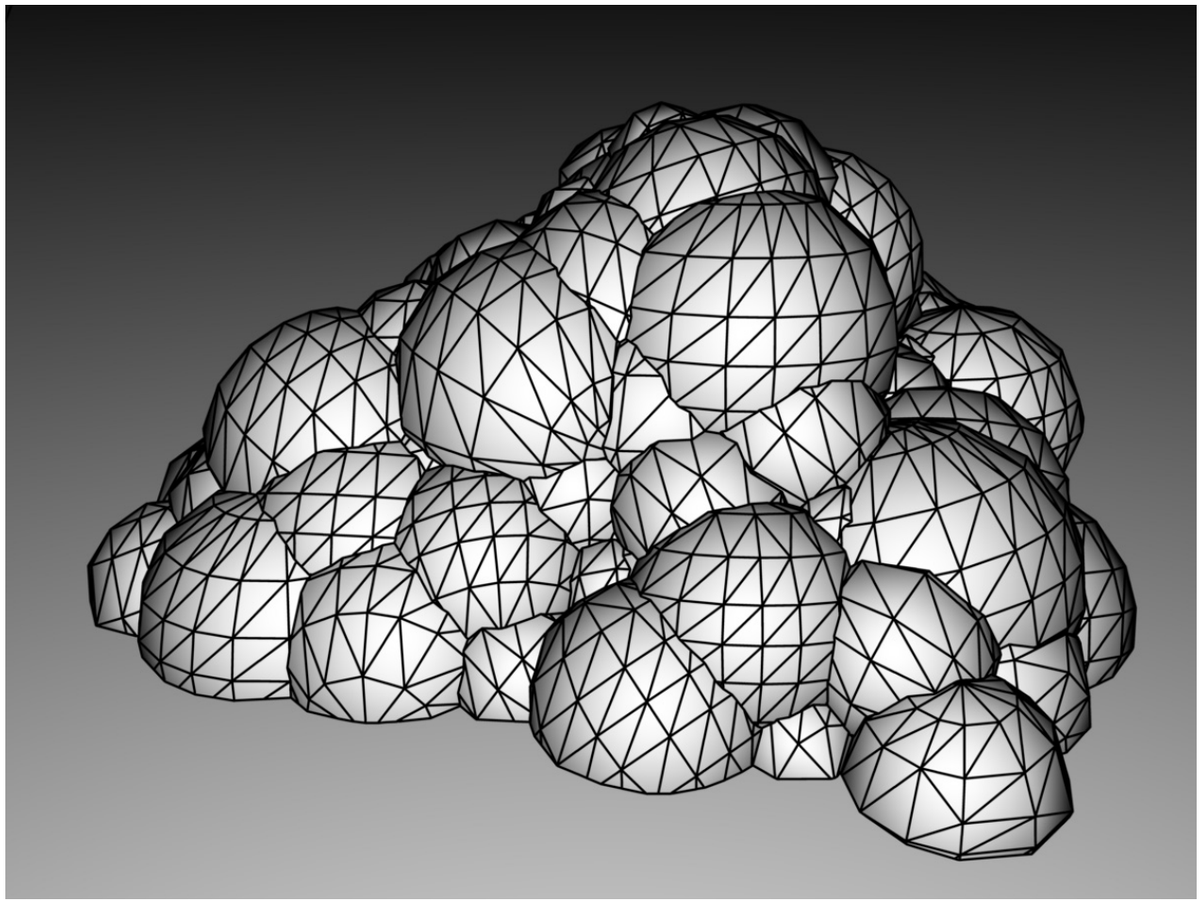
The color of edges

Displaying tessellation with contour shaders

```
 1  /* Info struct */
 2
 3  typedef struct {
 4      int primitive_index;
 5  } c_tessellate_info;
 6
 7  /* Store shader */
 8
 9  miBoolean c_tessellate_store (
10      void *info_pointer,
11      int *info_size,
12      miState *state,
13      miColor *color)
14  {
15      c_tessellate_info *info = (c_tessellate_info*)info_pointer;
16      int primitive_index;
17      mi_query(miQ_PRI_INDEX, state, 0, &primitive_index);
18      info->primitive_index = primitive_index;
19      *info_size = sizeof(c_tessellate_info);
20      return miTRUE;
21  }
22
23  /* Contrast shader */
24
25  miBoolean c_tessellate_contrast(
26      c_tessellate_info *info1,
27      c_tessellate_info *info2,
28      int level,
29      miState *state,
30      void *params)
31  {
32      if (info1 == NULL ||
33          info2 == NULL ||
34          info1->primitive_index != info2->primitive_index)
35          return miTRUE;
36      else
37          return miFALSE;
38  }
39
40  /* Contour shader */
41
42  struct c_tessellate_contour {
43      miColor color;
44      miScalar width;
45  };
46
47  miBoolean c_tessellate_contour(
48      miContour_endpoint *result,
49      c_tessellate_info *info_near,
50      c_tessellate_info *info_far,
51      miState *state,
52      struct c_tessellate_contour *params)
53  {
54      result->color = *mi_eval_color(&params->color);
55      result->width = *mi_eval_scalar(&params->width);
56      return miTRUE;
57  }
```

Three shader functions for contour

The color of edges



Displaying tessellation with contour shaders

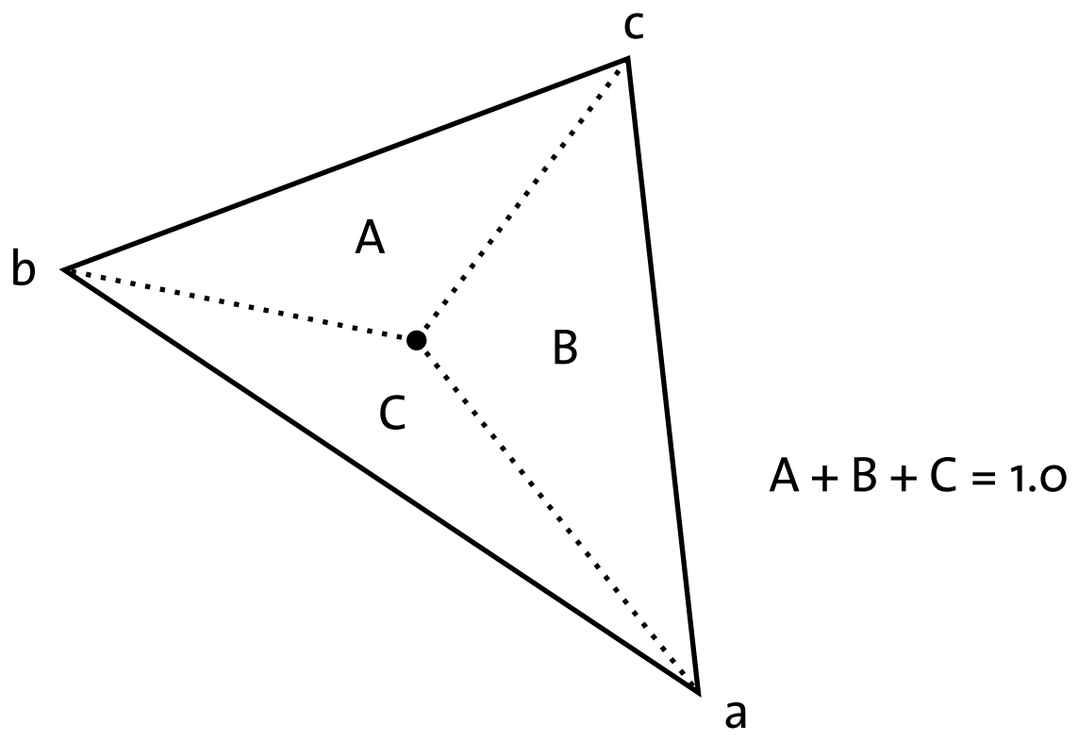
```
options "opt"
    object space
    samples 0 2
    contour store
        "c_tessellate_store" ()
    contour contrast
        "c_tessellate_contrast" ()
    approximate fine view length 20 all
end options

camera "cam"
    output "contour,rgba"
        "contour_composite" ()
    output "rgba" "tif"
        "contours_6.tif"
    focal 32
    aperture 33.3
    aspect 1.333
    resolution 400 300
end camera

material "outline"
    "front_bright" ()
     contour
         "c_tessellate_contour" (
             "width" .2 )
end material
```

Combining the tessellation display with shader front_bright

Exercise 10: Contour shaders

1. Copy `contours_6.mi` to `contours.mi`, change output file to `contours.tif`, render and view.
2. Change the material shader from `front_bright` to `one_color` with a `color` parameter of white.
3. Modify the `approximate` option to use a length of 10.
4. Look up *approximation* in the “Index” page of the on-line documentation.

The color of edges



Barycentric coordinates as the ratio of areas of subtriangles

```
declare shader
    color "show_barycentric" (
        color "a" default 1 .3 .3,
        color "b" default .3 1 .3,
        color "c" default .3 .3 1 )
end declare
```

Scene file declaration of shader "show_barycentric"

```
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  }
```

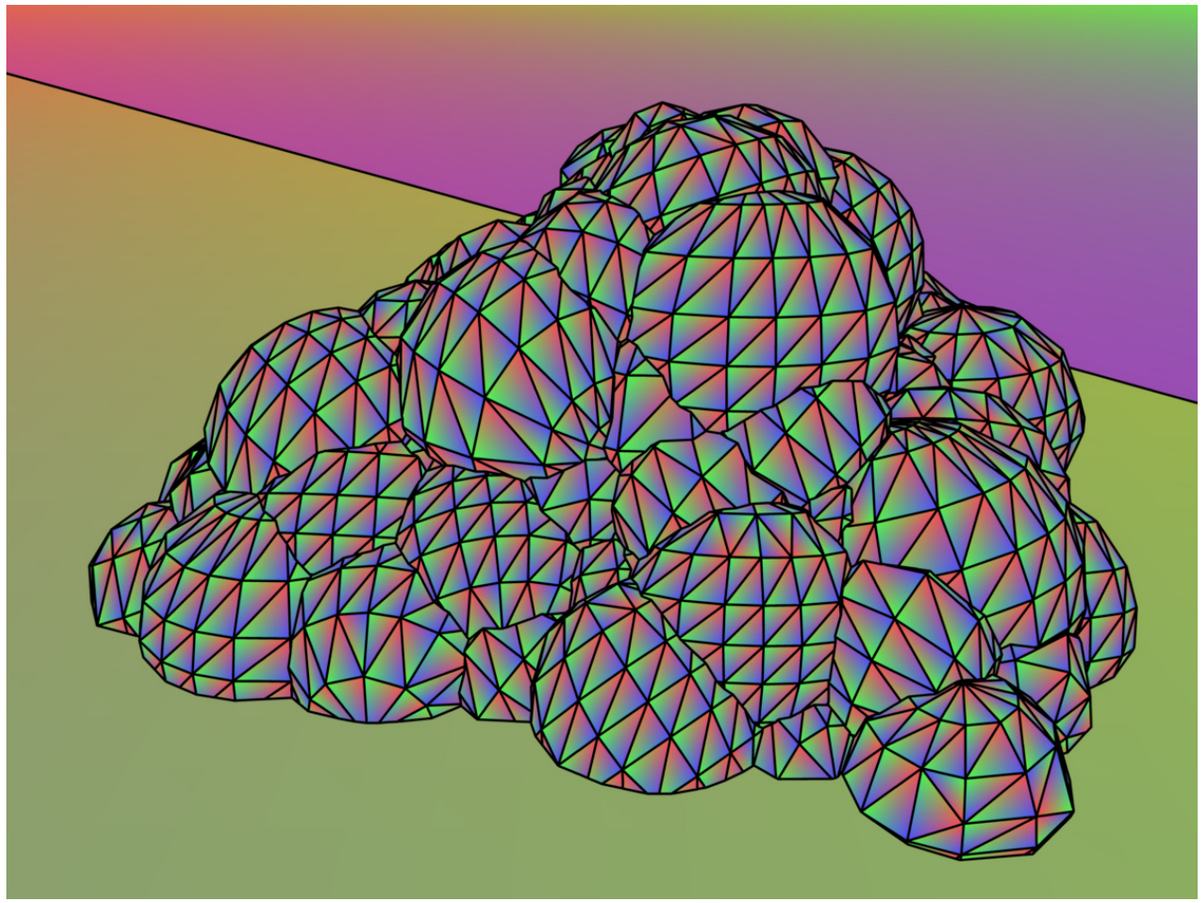
Auxiliary function: miaux_add_scaled_color

The color of edges


```
1  struct show_barycentric {
2      miColor a;
3      miColor b;
4      miColor c;
5  };
6
7  miBoolean show_barycentric (
8      miColor *result, miState *state, struct show_barycentric *params )
9  {
10     miaux_add_scaled_color(result, mi_eval_color(&params->a), state->bary[0]);
11     miaux_add_scaled_color(result, mi_eval_color(&params->b), state->bary[1]);
12     miaux_add_scaled_color(result, mi_eval_color(&params->c), state->bary[2]);
13     return miTRUE;
14 }
```

Source code of shader "show_barycentric"

The color of edges



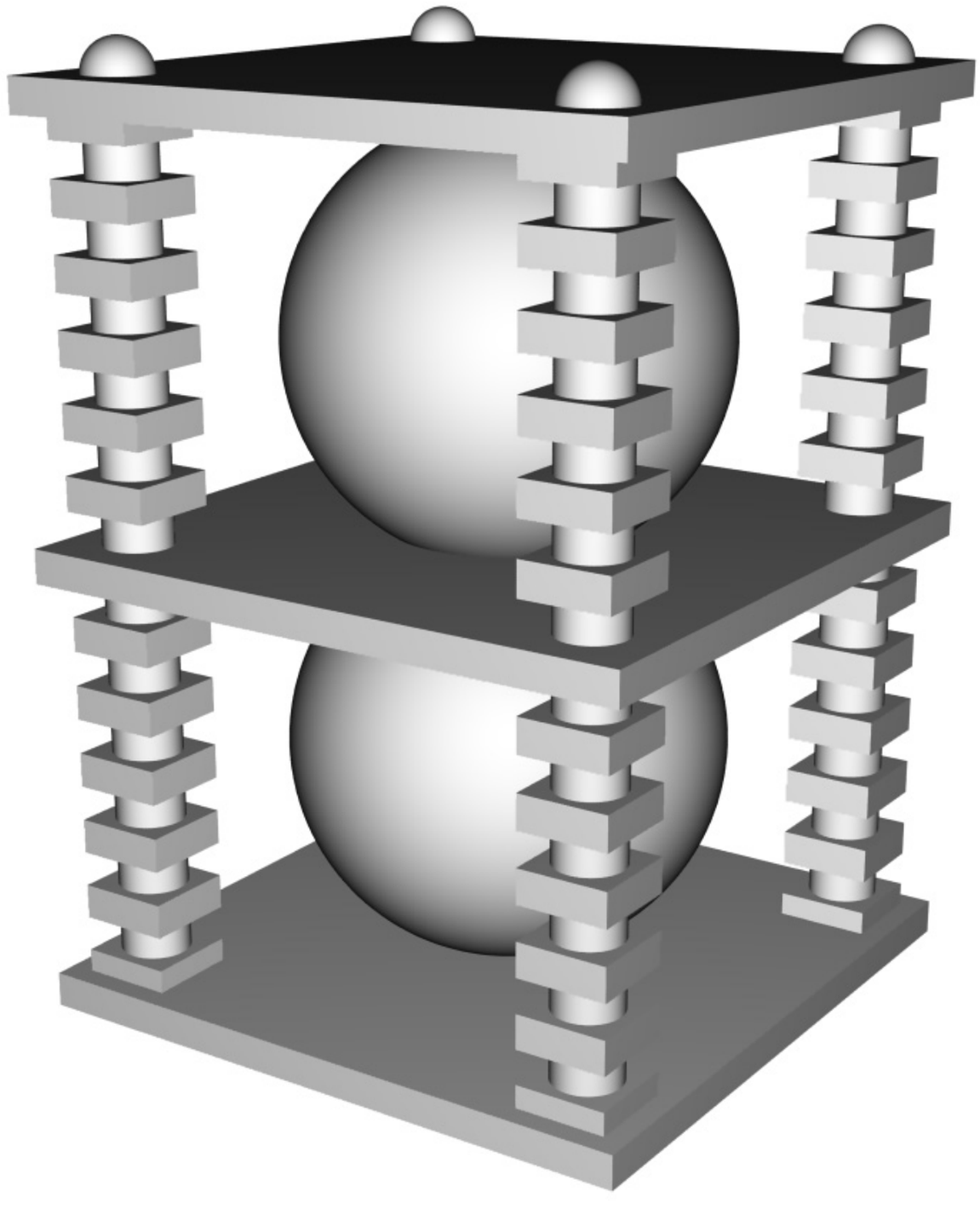
Displaying tessellation with contour shaders

```
options "opt"
    object space
    samples 0 2
    contour store
        "c_tessellate_store" ()
    contour contrast
        "c_tessellate_contrast" ()
    approximate fine view length 20 all
end options

camera "cam"
    output "contour,rgba"
        "contour_composite" ()
    output "rgba" "tif"
        "contours_7.tif"
    focal 32
    aperture 33.3
    aspect 1.333
    resolution 400 300
end camera

material "outline"
    "show_barycentric" ()
    contour
        "c_tessellate_contour" (
            "width" .2 )
end material
```

Displaying tessellation with contours composited over barycentric coordinates as color

The color of edges

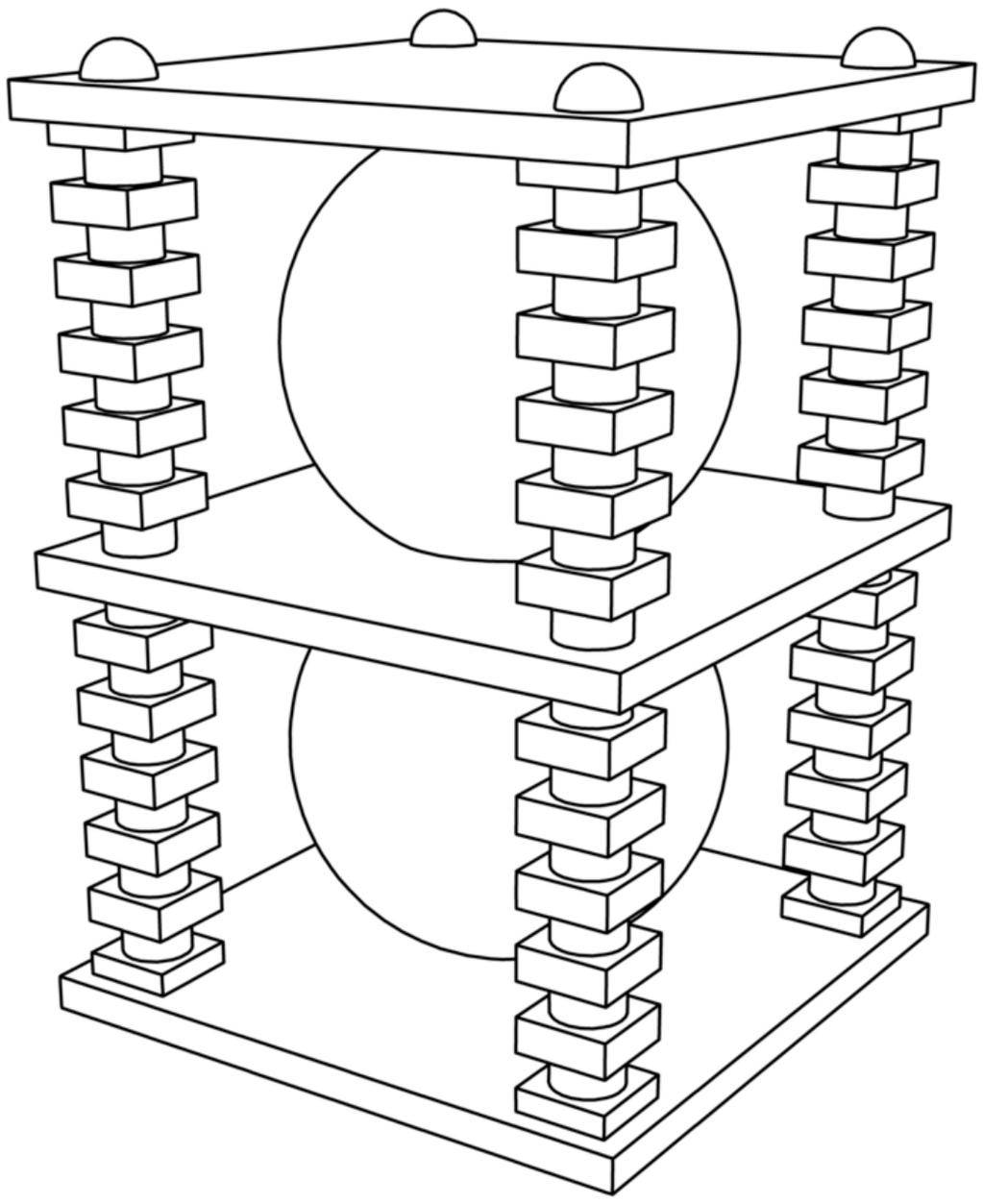
Including color in the determination of contour lines



```
material "front"
     "front_bright" ()
end material
```

Shader front_bright with ambiguous geometric details

The color of edges



Including color in the determination of contour lines

```
options "opt"
    object space
    samples 0 2
    contour store "c_store" ()
    contour contrast "c_contrast" ()
end options

camera "cam"
    output "contour,rgba"
        "contour_composite" ()
    output "rgba" "tif" "contours_9.tif"
    focal 2.3
    aperture 1
    aspect 0.833333
    resolution 250 300
    environment
        "one_color" (
            "color" 1 1 1 )
end camera

material "contours"
    "one_color" (
        "color" 1 1 1 )
    contour
        "c_contour" (
            "width" .3,
            "color" 0 0 0 1 )
end material
```

Contour shading drawing silhouettes, object intersection and internal edges

The color of edges

Including color in the determination of contour lines

```
1  miScalar miaux_quantize(miScalar value, miInteger count)
2  {
3      miScalar q = (miScalar)count;
4      if (count < 2)
5          return q;
6      else
7          return (miScalar)((int)(value * q) / (q - 1));
8  }
```

Auxiliary function: miaux_quantize

The color of edges

Including color in the determination of contour lines

```
declare shader
    color "front_bright_steps" (
        color "tint" default 1 1 1,
        integer "steps" default 3 )
end declare
```

Scene file declaration of shader "front_bright_steps"

The color of edges

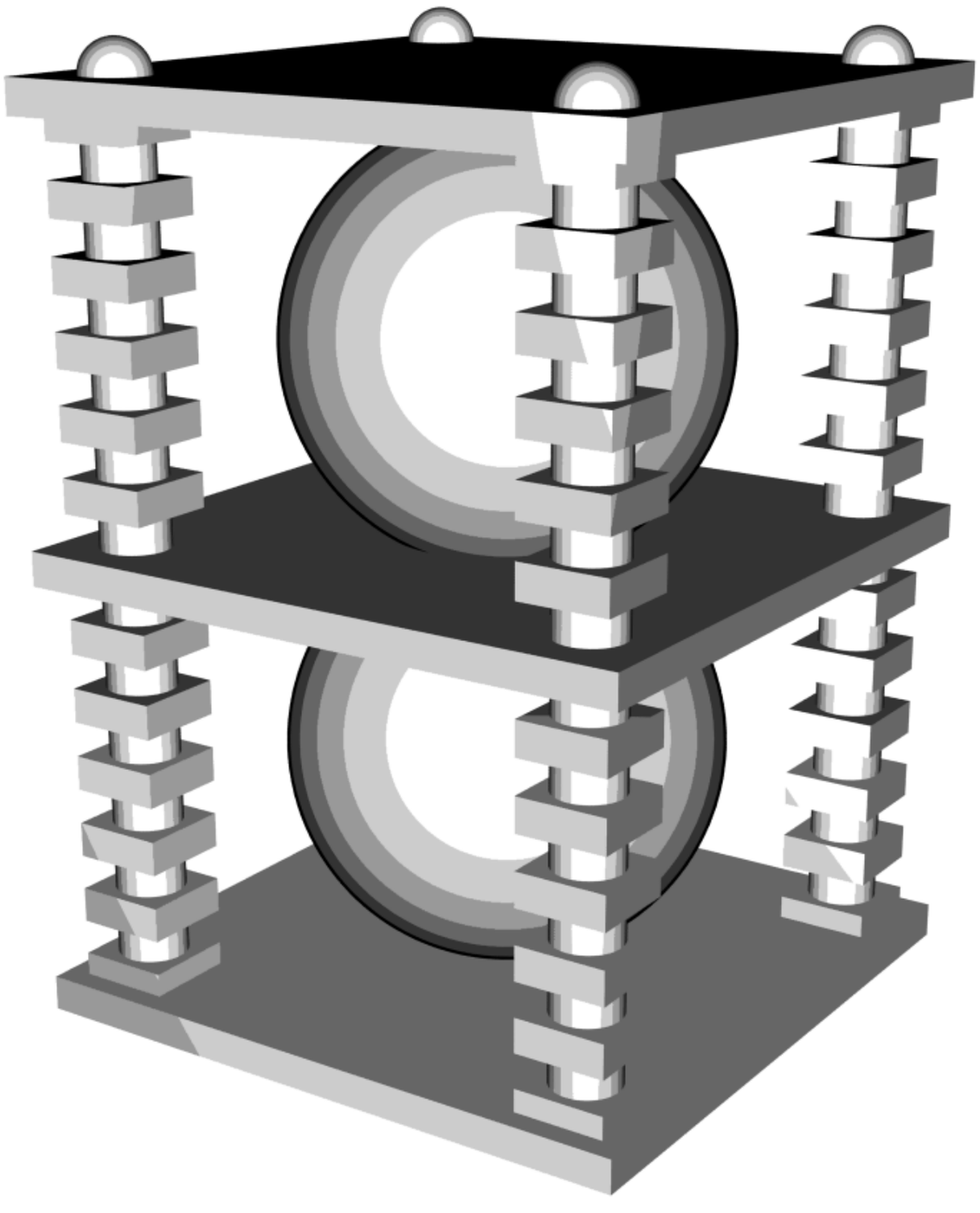
Including color in the determination of contour lines

```
 1  struct front_bright_steps {
 2      miColor tint;
 3      miInteger steps;
 4  };
 5
 6  miBoolean front_bright_steps (
 7      miColor *result, miState *state, struct front_bright_steps *params )
 8  {
 9      miColor *tint = mi_eval_color(&params->tint);
10      miScalar scale =
11          miaux_quantize(-state->dot_nd, *mi_eval_integer(&params->steps));
12      result->r = tint->r * scale;
13      result->g = tint->g * scale;
14      result->b = tint->b * scale;
15      result->a = 1.0;
16      return miTRUE;
17  }
```

Source code of shader "front_bright_steps"

The color of edges

Including color in the determination of contour lines



```
material "front_bands"
    "front_bright_steps" (
        "steps" 6 )
end material
```

Quantization of shader front_bright to produce bands of constant gray values

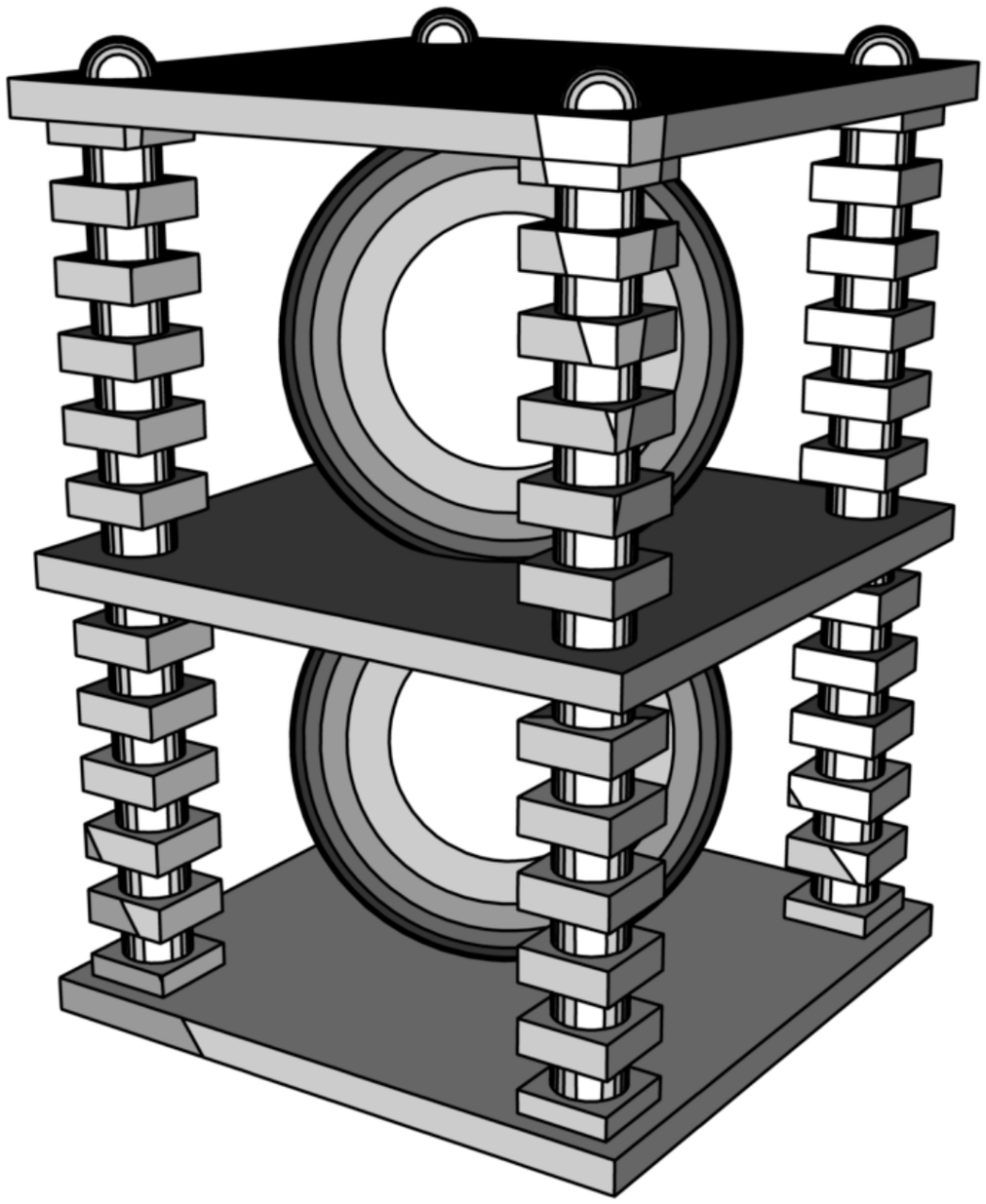
The color of edges

Including color in the determination of contour lines

```
 1  typedef struct {
 2      miTag instance;
 3      miVector normal;
 4      miColor color;
 5  } c_toon_info;
 6
 7  miBoolean c_toon_store (
 8      void *info_pointer, int *info_size, miState *state, miColor *color)
 9  {
10      c_toon_info *info = (c_toon_info*)info_pointer;
11
12      info->instance = state->instance;
13      info->normal = state->normal;
14      info->color = *color;
15      *info_size = sizeof(c_toon_info);
16
17      return miTRUE;
18  }
19
20  struct c_toon_contrast {
21      miScalar dot_threshold;
22  };
23
24  miBoolean c_toon_contrast (
25      c_toon_info *info1, c_toon_info *info2, int level, miState *state,
26      struct c_toon_contrast *params )
27  {
28      if (info1 == NULL ||
29          info2 == NULL ||
30          info1->instance != info2->instance ||
31          (mi_vector_dot(&info1->normal, &info2->normal) <
32           *mi_eval_scalar(&params->dot_threshold)) ||
33          info1->color.r != info2->color.r ||
34          info1->color.g != info2->color.g ||
35          info1->color.b != info2->color.b)
36          return miTRUE;
37      else
38          return miFALSE;
39  }
40
41  struct c_toon_contour {
42      miColor color;
43      miScalar width;
44  };
45
46  miBoolean c_toon_contour (
47      miContour_endpoint *result, c_toon_info *info_near, c_toon_info *info_far,
48      miState *state, struct c_toon_contour *params )
49  {
50      result->color = *mi_eval_color(&params->color);
51      result->width = *mi_eval_scalar(&params->width);
52      return miTRUE;
53  }
```

The Store, Contrast, and Contour shaders for

The color of edges



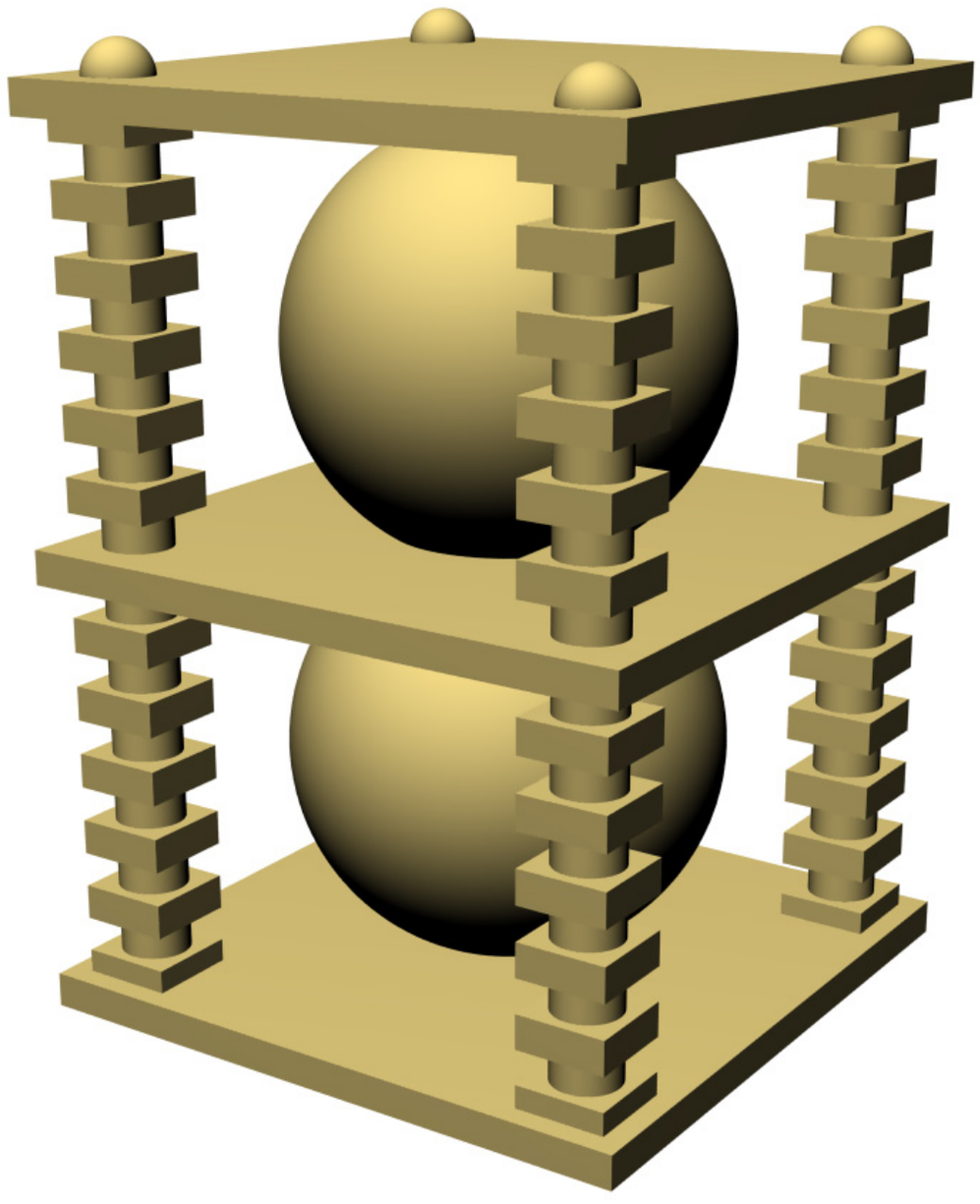
Including color in the determination of contour lines

```
options "opt"
    object space
    samples 0 2
    contour store "c_toon_store" ()
    contour contrast "c_toon_contrast" ()
end options

camera "cam"
    output "contour,rgba"
        "contour_composite" ()
    output "rgba" "tif" "contours_11.tif"
    focal 2.3
    aperture 1
    aspect 0.833333
    resolution 250 300
    environment
        "one_color" (
            "color" 1 1 1 )
end camera

material "front_bright_contours"
    "front_bright_steps" (
        "steps" 6 )
    contour
        "c_toon_contour" (
           "width" .3,
           "color" 0 0 0 1 )
end material
```

Combining shader front_bright_steps with contours at color boundaries

The color of edges

Including color in the determination of contour lines



```
light "light"
    "point_light" (
        "light_color" 1 1 1 )
    origin 20 80 60
end light

instance "light_inst" "light" end instance

material "diffuse"
    "lambert" (
        "diffuse" 1 .9 .55,
        "lights" ["light_inst"] )
end material
```

A diffuse surface with a single light

The color of edges

Including color in the determination of contour lines

```
declare shader
    color "lambert_steps" (
        color "ambient" default 0 0 0,
        color "diffuse" default 1 1 1,
        integer "steps" default 3,
        array light "lights" )
end declare
```

Scene file declaration of shader "lambert_steps"

The color of edges

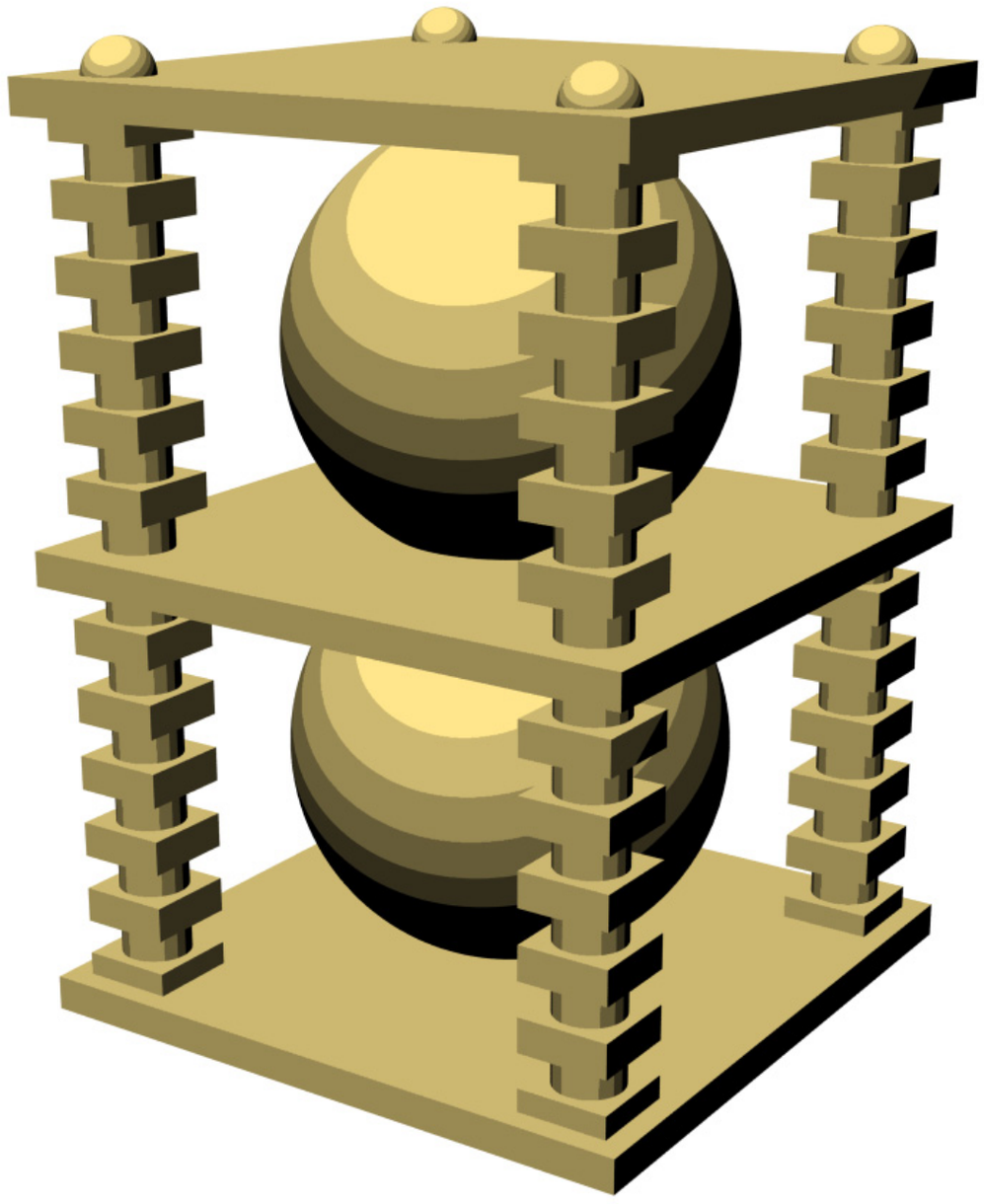
Including color in the determination of contour lines

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

Source code of shader "lambert_steps"

The color of edges

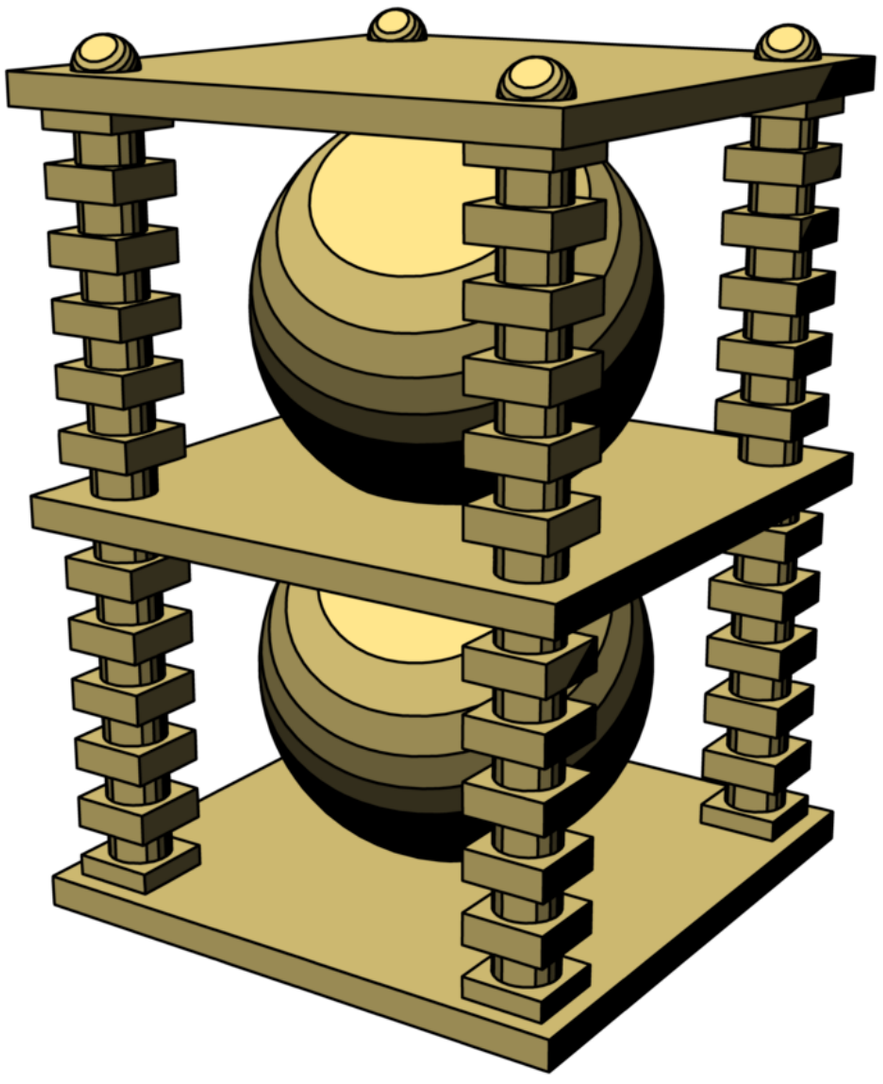
Including color in the determination of contour lines



```
light "light"
    "point_light" (
        "light_color" 1 1 1 )
    origin 20 80 60
end light

instance "light_inst" "light" end instance

material "diffuse_steps"
    "lambert_steps" (
        "steps" 6,
        "diffuse" 1 .9 .55,
        "lights" ["light_inst"] )
end material
```

A diffuse surface with a single light with color banding

The color of edges

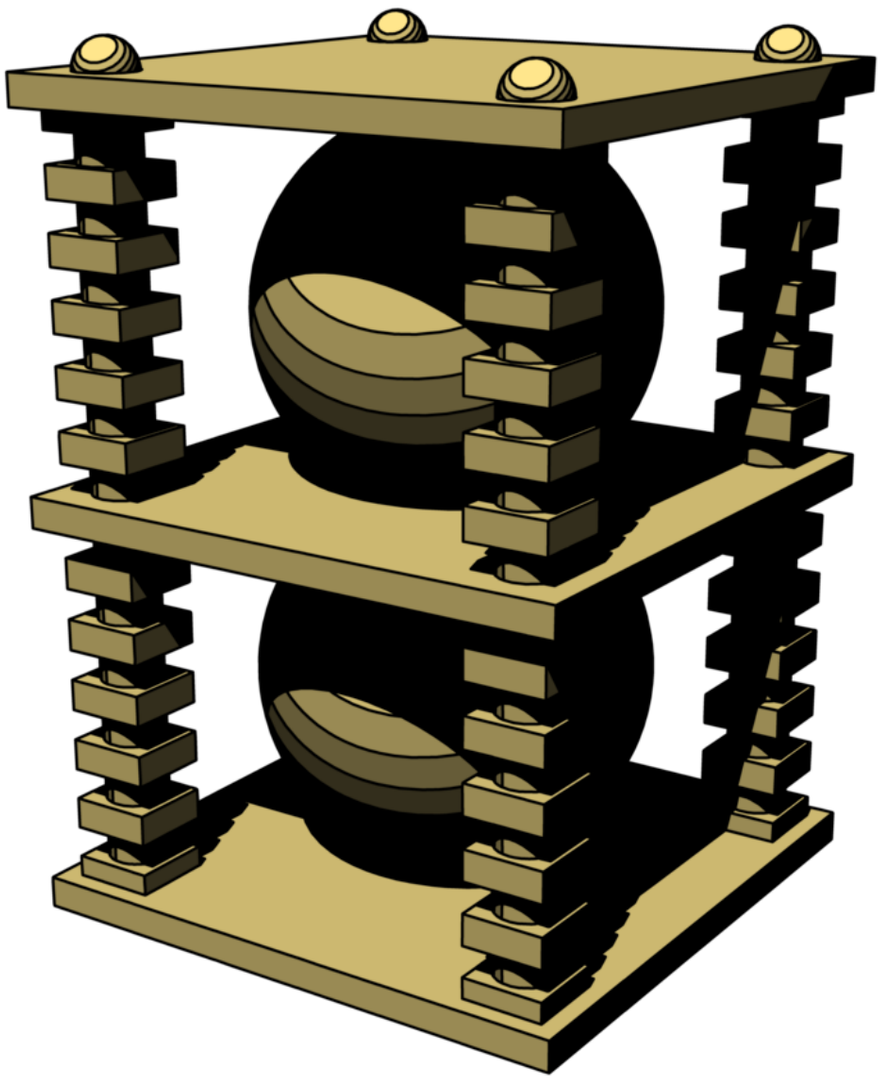


Including color in the determination of contour lines

```
options "opt"
    object space
    samples 0 2
    contour store "c_toon_store" ()
    contour contrast "c_toon_contrast" ()
end options

camera "cam"
    output "contour,rgba"
        "contour_composite" ()
    output "rgba" "tif" "contours_14.tif"
    focal 2.3
    aperture 1
    aspect 0.833333
    resolution 250 300
    environment
        "one_color" (
            "color" 1 1 1 )
end camera

light "light"
    "point_light" (
        "light_color" 1 1 1 )
    origin 20 80 60
end light

instance "light_inst" "light" end instance

material "contour_diffuse_steps"
    "lambert_steps" (
        "steps" 6,
        "diffuse" 1 .9 .55,
        "lights" ["light_inst"] )
    contour
        "c_toon_contour" (
           "width" .3,
           "color" 0 0 0 1 )
end material
```

Adding contours to the color banding of a diffuse surface

The color of edges

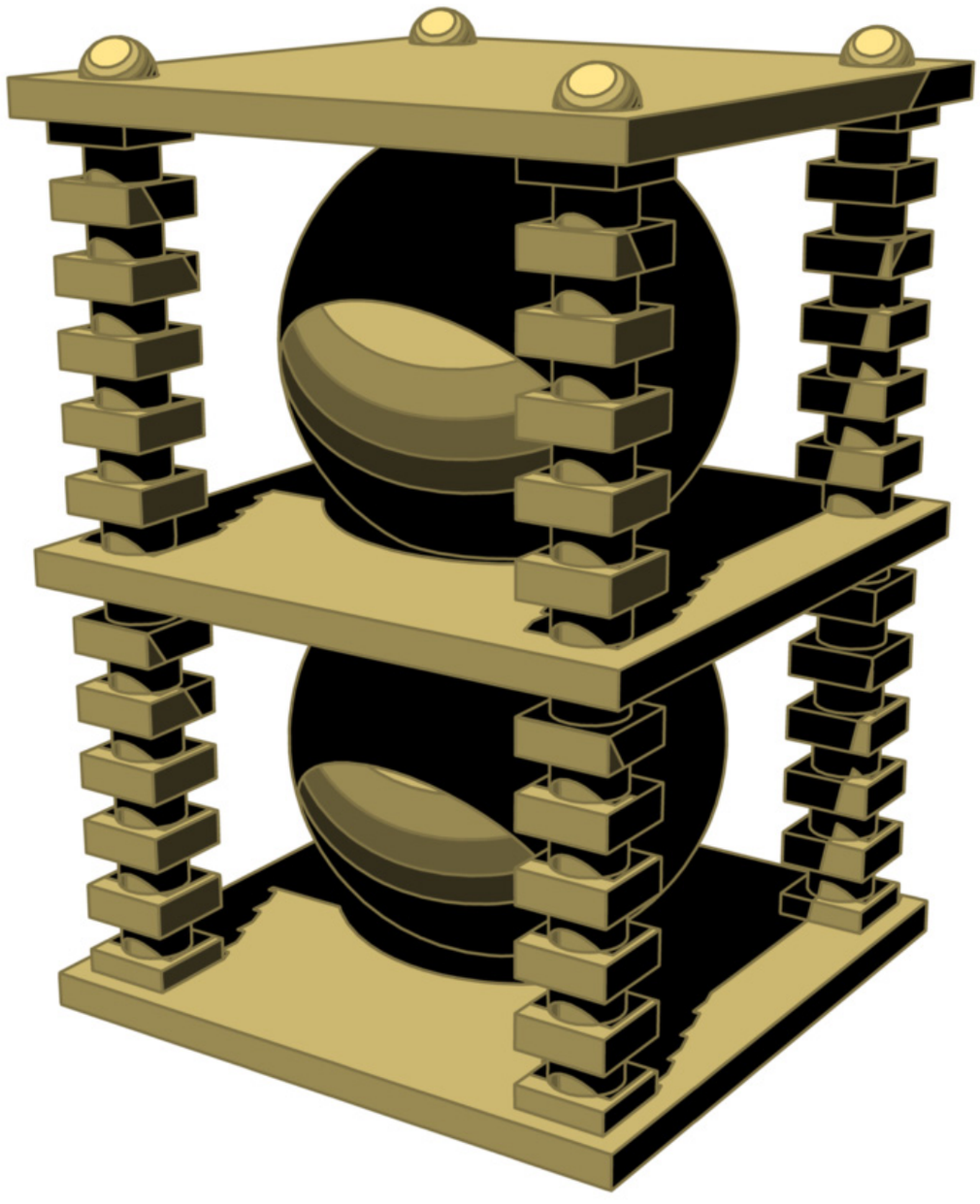


Including color in the determination of contour lines

```
options "opt"
    object space
    samples 0 2
    contour store "c_toon_store" ()
    contour contrast "c_toon_contrast" ()
end options

camera "cam"
    output "contour,rgba"
        "contour_composite" ()
    output "rgba" "tif" "contours_15.tif"
    focal 2.3
    aperture 1
    aspect 0.833333
    resolution 250 300
    environment
        "one_color" (
            "color" 1 1 1 )
end camera

light "light"
    "point_light_shadow" (
        "light_color" 1 1 1 )
    origin 20 80 60
end light

instance "light_inst" "light" end instance

material "contour_diffuse_steps"
    "lambert_steps" (
        "steps" 6,
        "diffuse" 1 .9 .55,
        "lights" ["light_inst"] )
    contour
        "c_toon_contour" (
           "width" .3,
           "color" 0 0 0 1 )
end material
```

Adding shadows to the contour rendering of a banded diffuse surface

The color of edges



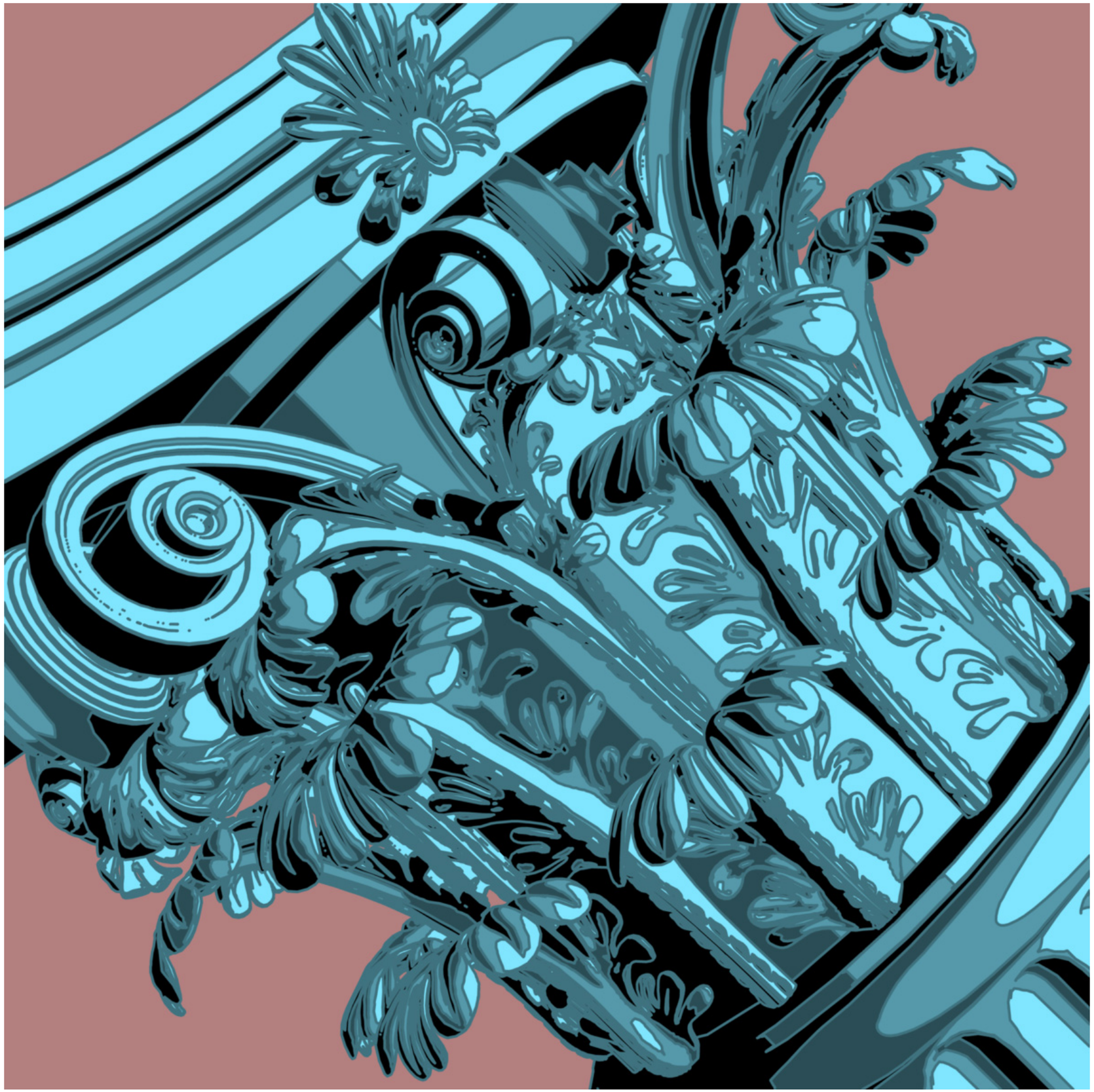
Including color in the determination of contour lines

```
options "opt"
    object space
    samples 0 2
    contour store "c_toon_store" ()
    contour contrast "c_toon_contrast" ()
end options

camera "cam"
    output "contour,rgba"
        "contour_composite" ()
    output "rgba" "tif" "contours_16.tif"
    focal 2.3
    aperture 1
    aspect 0.833333
    resolution 250 300
    environment
        "one_color" (
            "color" 1 1 1 )
end camera

material "contour_diffuse_steps"
    "lambert_steps" (
        "steps" 6,
        "diffuse" 1 .9 .55,
        "lights" ["light_inst"] )
    contour
        "c_toon_contour" (
          "width" .3,
          "color" .5 .45 .275 1 )
end material
```

Revealing contour detail in shadows with a brighter line color

The color of edges

Including color in the determination of contour lines



Three color bands with a midrange color for contour lines