N-Dimensional Raytracer Development

Revision: `2.1.2007.05.17`

Patrick A. Stein

May 17, 2007

Contents

1 Scene **13**

1.1 Constructor 14
1.2 Destructor 14
1.3 Retrieving A Pointer To A Universe 15
1.4 Rendering The Scene 15
1.5 Loading A Scene 15
1.6 The Class Definition 17
1.7 Source Files 18
1.7.1 `scene.h` 18
1.7.2 `scene.cc` 18

2 View **19**

2.1 Constructor 23
2.2 Destructor 24
2.3 Rendering The View 24
2.4 Calculating The Viewing Direction From The Pixel Coordinates 28
2.5 Incrementing The Pixel Coordinates 30
2.6 Loading A View 31
2.7 The Class Definition 38
2.8 Source Files 38
2.8.1 `view.h` 38
2.8.2 `view.cc` 39

3 Universe **41**

3.1 Constructor 42
3.2 Destructor 43
3.3 Getting And Setting The Number Of Dimensions 43
3.4 Casting A Ray Into The Universe 44
3.5 Checking For Shadows 50
3.6 Loading A Universe 52
3.7 The Class Definition 54
3.8 Source Files 55
3.8.1 `universe.h` 55
3.8.2 `universe.cc` 56

4 Universe Intersection Work **57**
4.1 Constructor 57
4.2 Destructor 58
4.3 Setting Up The Work 58
4.4 Performing The Work 58
4.5 Checking The Work 59
4.6 The Class Definition 59

5 Universe Reflection Work **61**
5.1 Constructor 62
5.2 Destructor 62
5.3 Performing The Work 63
5.4 Checking The Work 63
5.5 The Class Definition 64

6 Universe Refraction Work **65**
6.1 Constructor 66
6.2 Destructor 66
6.3 Performing The Work 67
6.4 Checking The Work 69
6.5 The Class Definition 69

7 Universe Shadow Work **71**
7.1 Constructor 72
7.2 Destructor 72
7.3 Preparing The Work 72
7.4 Performing The Work 73
7.5 Checking The Work 75
7.6 The Class Definition 75

8 Light **77**
8.1 Constructor 78
8.2 Setting The Number of Dimensions 78
8.3 Retreiving The Light Position 79
8.4 Retreiving The Fall Off Factor 79
8.5 Querying The Light Color 80
8.6 Loading A Light 81
8.7 The Class Definition 82
8.8 Source Files 83
8.8.1 `light.h` 83
8.8.2 `light.cc` 83

9 Object **85**
9.1 Contructor 86
9.2 Destructor 86
9.3 Setting The Number of Dimensions 87

9.4 Retreiving The Common Attributes . . . 87
9.5 Setting The Common Attributes . . . 88
9.6 Intersection A Ray And An Object . . . 88
9.7 Checking If A Point Is Inside An Object . . . 89
9.8 Querying The Object Color . . . 89
9.9 Forging A Hit . . . 90
9.10 Reading Parts Common To All Objects . . . 91
9.11 Loading An Object . . . 92
9.12 The Class Definition . . . 93
9.13 Source Files . . . 93
9.13.1 `object.h` . . . 93
9.13.2 `object.cc` . . . 94

10 Halfspace **95**
10.1 Constructor . . . 95
10.2 Destructor . . . 96
10.3 Setting The Number of Dimensions . . . 97
10.4 Intersection A Ray And A Halfspace . . . 97
10.5 Checking If A Point Is Inside A Halfspace . . . 101
10.6 Loading A Halfspace . . . 101
10.7 The Class Definition . . . 102
10.8 Source Files . . . 103
10.8.1 `halfspace.h` . . . 103
10.8.2 `halfspace.cc` . . . 103

11 Cylinder **105**
11.1 Constructor . . . 106
11.2 Destructor . . . 107
11.3 Setting The Number of Dimensions . . . 107
11.4 Intersection A Ray And A Cylinder . . . 108
11.4.1 Intersecting With The Spherical Portions . . . 113
11.4.2 Intersecting With The Cubical Portions . . . 115
11.5 Checking If A Point Is Inside A Cylinder . . . 117
11.6 Loading A Cylinder . . . 118
11.7 The Class Definition . . . 120
11.8 Source Files . . . 120
11.8.1 `cylinder.h` . . . 120
11.8.2 `cylinder.cc` . . . 121

12 Quadratic **123**
12.1 Constructor . . . 124
12.2 Destructor . . . 124
12.3 Setting The Number of Dimensions . . . 124
12.4 Intersection A Ray And A Quadratic . . . 125
12.5 Checking If A Point Is Inside A Quadratic . . . 131
12.6 Loading A Quadratic . . . 131

12.7 The Class Definition . . . 133
12.8 Source Files . . . 133
12.8.1 quadratic.h . . . 133
12.8.2 quadratic.cc . . . 134

13 Convex 135
13.1 Constructor . . . 136
13.2 Destructor . . . 136
13.3 Adding A Point . . . 136
13.4 Calculating The Bounding Planes . . . 137
13.5 Incrementing A Choice . . . 143
13.6 Intersection A Ray And A Convex Hull . . . 144
13.7 Loading A Convex . . . 147
13.8 The Class Definition . . . 149
13.9 Source Files . . . 150
13.9.1 convex.h . . . 150
13.9.2 convex.cc . . . 150

14 Coxeter 151
14.1 Constructor . . . 152
14.2 Destructor . . . 152
14.3 Loading A Coxeter . . . 152
14.4 The Class Definition . . . 158
14.5 Source Files . . . 158
14.5.1 coxeter.h . . . 158
14.5.2 coxeter.cc . . . 159

15 Complement 161
15.1 Constructor . . . 162
15.2 Destructor . . . 162
15.3 Setting The Number of Dimensions . . . 162
15.4 Intersection A Ray And A Complement . . . 163
15.5 Checking If A Point Is Inside A Complement . . . 165
15.6 Loading A Complement . . . 166
15.7 The Class Definition . . . 168
15.8 Source Files . . . 168
15.8.1 complement.h . . . 168
15.8.2 complement.cc . . . 169

16 Union 171
16.1 Constructor . . . 172
16.2 Destructor . . . 172
16.3 Setting The Number of Dimensions . . . 172
16.4 Intersection A Ray And A Union . . . 173
16.5 Checking If A Point Is Inside A Union . . . 179
16.6 Loading A Union . . . 180

16.7 The Class Definition . . . 182
16.8 Source Files . . . 182
16.8.1 union.h . . . 182
16.8.2 union.cc . . . 183

17 Set 185
17.1 Constructor . . . 186
17.2 Destructor . . . 186
17.3 Setting The Number of Dimensions . . . 186
17.4 Intersection A Ray And A Set . . . 187
17.5 Checking If A Point Is Inside A Set . . . 190
17.6 Loading A Set . . . 190
17.7 The Class Definition . . . 192
17.8 Source Files . . . 192
17.8.1 set.h . . . 192
17.8.2 set.cc . . . 193

18 Inter 195
18.1 Constructor . . . 196
18.2 Destructor . . . 196
18.3 Setting The Number of Dimensions . . . 196
18.4 Intersection A Ray And A Inter . . . 197
18.5 Checking If A Point Is Inside A Intersection . . . 203
18.6 Loading A Inter . . . 204
18.7 The Class Definition . . . 206
18.8 Source Files . . . 206
18.8.1 inter.h . . . 206
18.8.2 inter.cc . . . 207

19 Extrusion 209
19.1 Constructor . . . 210
19.2 Destructor . . . 210
19.3 Setting The Number of Dimensions . . . 210
19.4 Intersection A Ray And A Extrusion . . . 211
19.5 Checking If A Point Is Inside An Extrusion . . . 214
19.6 Loading A Extrusion . . . 215
19.7 The Class Definition . . . 217
19.8 Source Files . . . 217
19.8.1 extrusion.h . . . 217
19.8.2 extrusion.cc . . . 218

20 Image 219
20.1 Contructor . . . 220
20.2 Destructor . . . 220
20.3 Setting The Image Information . . . 221
20.4 Counting Output Images . . . 222

20.5 Adding Pixels . . . 222
20.6 Loading An Image . . . 222
20.7 The Class Definition . . . 224
20.8 Source Files . . . 225
20.8.1 image.h . . . 225
20.8.2 image.cc . . . 225

21 PPM Images 227
21.1 Constructor . . . 228
21.2 Destructor . . . 228
21.3 Setting The Image Information . . . 229
21.4 Adding Pixels . . . 230
21.5 Reading In A PPM . . . 231
21.6 The Class Definition . . . 233

22 PNG Images 235
22.1 Constructor . . . 236
22.2 Destructor . . . 236
22.3 Setting The Image Information . . . 237
22.4 Adding Pixels . . . 238
22.5 Reading In A PNG . . . 242
22.6 The Class Definition . . . 243

23 Accum Images 245
23.1 Constructor . . . 246
23.2 Destructor . . . 246
23.3 Setting The Image Information . . . 247
23.4 Adding Pixels . . . 248
23.5 Reading In A Accum . . . 251
23.6 The Class Definition . . . 253

24 Portal 255
24.1 Constructor . . . 256
24.2 Destructor . . . 256
24.3 Retrieving A Reference To Its Pixel Value . . . 257
24.4 Setting Up The Number Of Dimensions . . . 257
24.5 Casting A Ray Into The Portal . . . 258
24.6 Loading A Portal . . . 259
24.7 The Class Definition . . . 261
24.8 Source Files . . . 261
24.8.1 portal.h . . . 261
24.8.2 portal.cc . . . 262

25 Color 263
25.1 Constructor . . . 265
25.2 Setting Up The Number Of Dimensions . . . 266
25.3 Evaluating The Color . . . 267
25.4 Loading A Color . . . 270
25.5 The Class Definition . . . 272
25.6 Source Files . . . 272
25.6.1 `color.h` . . . 273
25.6.2 `color.cc` . . . 273

26 Prefix Notation Expressions 275
26.1 Destructor . . . 276
26.2 Checking For Constness . . . 276
26.3 Evaluating The Expression . . . 277
26.4 Loading An Expression . . . 277
26.5 Loading A Vector Of Expressions . . . 281
26.6 The Expression Table Initializers . . . 283
26.7 The Class Definition . . . 284
26.8 Source Files . . . 284
26.8.1 `expr.h` . . . 284
26.8.2 `expr.cc` . . . 285

27 Binary Expressions 287
27.1 Constructor . . . 288
27.2 Destructor . . . 288
27.3 Checking For Constness . . . 288
27.4 Evaluating The Expression . . . 289
27.5 Reading In A Binary Expression . . . 289
27.6 The Binary Expression Table . . . 290
27.7 The Class Definition . . . 293

28 Unary Expressions 295
28.1 Constructor . . . 296
28.2 Destructor . . . 296
28.3 Checking For Constness . . . 296
28.4 Evaluating The Expression . . . 297
28.5 Reading In A Unary Expression . . . 297
28.6 The Unary Expression Table . . . 298
28.7 The Class Definition . . . 300

29 Constant Expressions 301
29.1 Constructor . . . 301
29.2 Checking For Constness . . . 302
29.3 Evaluating The Expression . . . 302
29.4 Reading In A Constant Expression . . . 303
29.5 The Constant Expression Table . . . 303

29.6 The Class Definition . . . 304

30 Position Expressions **305**
30.1 Constructor . . . 305
30.2 Destructor . . . 306
30.3 Checking For Constness . . . 306
30.4 Evaluating The Expression . . . 307
30.5 Reading In A Position Expression . . . 307
30.6 The Class Definition . . . 308

31 Direction Expressions **309**
31.1 Constructor . . . 309
31.2 Destructor . . . 310
31.3 Checking For Constness . . . 310
31.4 Evaluating The Expression . . . 311
31.5 Reading In A Direction Expression . . . 311
31.6 The Class Definition . . . 312

32 Normal Expressions **313**
32.1 Constructor . . . 313
32.2 Destructor . . . 314
32.3 Checking For Constness . . . 314
32.4 Evaluating The Expression . . . 315
32.5 Reading In A Normal Expression . . . 315
32.6 The Class Definition . . . 316

33 Reference Expressions **317**
33.1 Constructor . . . 318
33.2 Destructor . . . 318
33.3 Checking For Constness . . . 318
33.4 Evaluating The Expression . . . 319
33.5 Reading In A Reference Expression . . . 319
33.6 The Class Definition . . . 320

34 Mathematical Vectors **321**
34.1 Creating Vectors . . . 323
34.2 Comparing Vectors . . . 324
34.3 Scaling A Vector . . . 326
34.4 The Sum of Vectors . . . 329
34.5 The Difference of Vectors . . . 329
34.6 The Elementwise Product of Vectors . . . 330
34.7 The Elementwise Quotient of Vectors . . . 331
34.8 The Dot Product of Two Vectors . . . 333
34.9 Normalizing A Vector . . . 333
34.10Orthogonalizing A Matrix . . . 335
34.11Orthogonalizing A Matrix . . . 337

34.12 Rotating A Vector . . . 337
34.13 Antirotating A Vector . . . 338
34.14 Solving A System Of Linear Equations . . . 339
34.15 The Class Definition . . . 342
34.16 Source Files . . . 343
34.16.1 `mathvec.h` . . . 343
34.16.2 `mathvec.cc` . . . 343

35 Basic Vectors 345
35.1 Creating Vectors . . . 345
35.2 Input and Output of Vectors . . . 348
35.3 The Class Definition . . . 350
35.4 Source Files . . . 351
35.4.1 `basevec.h` . . . 351
35.4.2 `basevec.cc` . . . 352

36 Intersection 353
36.1 Constructor . . . 354
36.2 Setting The Intersection Attributes . . . 354
36.3 Getting The Intersection Attributes . . . 355
36.4 Setting The Object Intersection Attributes . . . 356
36.5 Updating The Object Intersection Attributes . . . 356
36.6 Querying The Intersection Color . . . 358
36.7 The Class Definition . . . 358
36.8 Source Files . . . 358
36.8.1 `intersection.h` . . . 359
36.8.2 `intersection.cc` . . . 359

37 Thread 361
37.1 Constructor . . . 362
37.2 Destructor . . . 362
37.3 Setting The Number Of Workers . . . 363
37.4 Make A Worker Available Again . . . 364
37.5 Starting Work . . . 364
37.6 Locking And Unlocking The Mutex . . . 365
37.7 Waiting On A Condition . . . 366
37.8 The Class Definition . . . 367
37.9 Source Files . . . 368
37.9.1 `nkthread.h` . . . 368
37.9.2 `nkthread.cc` . . . 368

38 Work 369
38.1 Constructor . . . 369
38.2 Destructor . . . 370
38.3 Performing The Work . . . 370
38.4 The Class Definition . . . 372

39 Worker **373**
39.1 Constructor . . . 373
39.2 Destructor . . . 374
39.3 Assigning Work . . . 374
39.4 Waking Up . . . 375
39.5 Loop For Work . . . 376
39.6 The Class Definition . . . 377

40 Input Class **379**
40.1 Constructor . . . 379
40.2 Checking An Input Stream . . . 379
40.3 Ignoring Part Of An Input Stream . . . 381
40.4 Getting A Token . . . 381
40.5 Source Files . . . 383
40.5.1 `input.h` . . . 383
40.5.2 `input.cc` . . . 383

41 Main Routine **385**
41.1 Source Files . . . 388
41.1.1 `main.cc` . . . 388

Sample Input Files **389**
41.2 Simple Rendering Without Raytracing . . . 390
41.3 Rendering Of Directional Spot Lights . . . 392
41.4 Rendering Of Two Halfspaces . . . 394
41.5 Rendering Of 3-Dimensional Cylinders . . . 398
41.6 Rendering Of A Quadratic Solid . . . 402
41.7 Rendering Of A Complement Of A Sphere . . . 405
41.8 Rendering Of The Union Of Two Spheres . . . 408
41.9 Rendering Of The Union Of Two Spheres . . . 412
41.10Rendering Of An Extrusion Of A Quadratic . . . 416
41.11Rendering Of 3-Dimensional Coxeters . . . 419
41.12Rendering From Inside The Union Of A Cube And A Cylinder . . 424

Chapter 1

Scene

The scene is the top-level input for this raytracer. The scene contains zero or more views[1] and zero or more named universes.

⟨*scene bnf*⟩≡

```
scene ::= scene_parameters ;

scene_parameters ::=
      scene_parameters scene_param
    | /*empty*/
    ;

scene_param ::=
        'view' '{' view_parameters '}' ';'
    |   'universe' universe_name '{' universe_parameters '}' ';'
    ;

universe_name ::= string ;
```

Views are defined in §2. Universes are defined in §3.

The scene class provides two major functions in this raytracer. First, it is the top-level encapsulation of all that is known for a given trace. Any entity wishing to use a universe as a texture will get a handle to that universe through the scene in which the entity is contained. Secondly, the scene's `render()` method loops through all of the views and invokes the `render()` method on each one of them.

The scene class contains an array of views and a map of universe pointers. The map of universes is indexed by name.

⟨*Scene Member Variables*⟩≡

```
typedef std::vector< const view* > ViewList;
ViewList views;
typedef std::map< std::string, const universe* > UniverseMap;
```

[1] Technically, if it has no views, it isn't going to render anything, so there's not much point in it.

```
UniverseMap universes;
```

1.1 Constructor

The default constructor does nothing.

⟨*Scene Method Declarations*⟩≡
```
scene( void );
```

⟨*Scene Methods*⟩≡
```
scene::scene( void )
{
}
```

1.2 Destructor

The destructor cleans out the views and universes.

⟨*Scene Method Declarations*⟩+≡
```
~scene( void );
```

All of this would be unneeded if the universe map and the view list had actual instances rather than pointers in them. But, there would be a great deal of copying around instances of sizeable instances if that were the case.

⟨*Scene Methods*⟩+≡
```
scene::~scene( void )
{
    UniverseMap::iterator uu;
    for ( uu=this->universes.begin(); uu != this->universes.end(); ++uu ) {
        delete uu->second;
    }
    ViewList::iterator vv;
    for ( vv=this->views.begin(); vv != this->views.end(); ++vv ) {
        delete *vv;
    }
}
```

1.3 Retrieving A Pointer To A Universe

Colors which wish to reference a given universe in the scene will query the scene with the name of the universe.

⟨*Scene Method Declarations*⟩+≡
```
const universe* getUniverse( const std::string& name ) const {
    UniverseMap::const_iterator uu = this->universes.find( name );
    if ( uu != this->universes.end() ) {
        return uu->second;
    } else {
        return 0;
    }
};
```

1.4 Rendering The Scene

The following method is used to render the scene.

⟨*Scene Method Declarations*⟩+≡
```
void render( void ) const;
```

This method simply loops through each view and invokes the render method of the view.

⟨*Scene Methods*⟩+≡
```
void
scene::render( void ) const
{
    ViewList::const_iterator vv;
    for ( vv=this->views.begin(); vv != this->views.end(); ++vv ) {
        (*vv)->render();
    }
}
```

This would be a convenient place to break off into separate threads. However, for most cases with multiple views, one would rather complete an output image as quickly as possible in order to monitor the progress of the scene. As such, it is doubtful that one would prefer to be rendering multiple files at the same time since it would take longer for the first output image to complete. As such, this method simply goes serially through the views.

1.5 Loading A Scene

The following method allows one to read in a scene from an input stream.

⟨*Scene Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, scene& ss );
```

There are only two valid keywords at the top-level of the scene. These are `view` and `universe`.

⟨*Scene Methods*⟩+≡
```
std::istream&
operator >> ( std::istream& stream, scene& ss )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "view" ) {
            ⟨Scene read view⟩
        } else if ( tok == "universe" ) {
            ⟨Scene read universe⟩
        } else {
            in.ignore( "SCENE", tok.c_str() );
        }
    }

    return stream;
}
```

When a view is encountered, a new instance is allocated passing in a pointer to the scene. After that, the view is read in from the input stream. Then, the view is appended to the list of views.

⟨*Scene read view*⟩≡
```
view* vv = new view( (const scene*)&ss );
stream >> *vv >> semicolon;
ss.views.insert( ss.views.end(), vv );
```

When a universe is encountered, its name is read in. Then, a new universe instance is allocated passing in a pointer to the scene. After that, the universe is read in from the input stream. Then, the universe is added by name to the map of universes.

⟨*Scene read universe*⟩≡
```
if ( in.get( token, sizeof(token) ) == 0 ) {
    return stream;
}
std::string name( token );
universe* uu = new universe( (const scene*)&ss );
stream >> *uu >> semicolon;
ss.universes[ name ] = uu;
```

1.6 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Scene Class Declaration*⟩≡
```
class scene {
protected:
    ⟨Scene Member Variables⟩
public:
    ⟨Scene Method Declarations⟩
public:
    static const char* id;
};
```

1.7 Source Files

The following sections assemble the source files for the scene class from the chunks above.

1.7.1 scene.h

The header file for the scene class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header files for the `view` and `universe` classes and then incorporates the class defined in §1.6 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*scene.h*⟩≡
```
#ifndef _NKLEIN_SCENE_H_
#define _NKLEIN_SCENE_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "view.h"
#include "universe.h"

    ⟨Scene Class Declaration⟩

#endif /*_NKLEIN_SCENE_H_*/
```

1.7.2 scene.cc

Herein, the implementations of the methods of the scene class are included. The implementation requires the header for the input subsystem and the declaration of the scene class itself.

⟨*scene.cc*⟩≡
```
#include "input.h"
#include "scene.h"

const char* scene::id = "NKVERSION: scene( 2.1.2007.05.17 )";

⟨Scene Methods⟩
```

Chapter 2

View

Most views will contain a viewport. That view is an instance of the color class defined in §25. For that color to actually represent a view into a universe, the color must contain a portal.

The view contains a great deal of information about the shape and resolution of the view. It contains an instance of the color class at which it will cast rays. It also contains zero or more output image specifications.

⟨*view bnf*⟩≡

```
view_parameters ::=
        view_parameters view_param
    |   /* empty */
    ;

view_param ::=
        view_units
    |   view_dpu
    |   view_depth
    |   view_fisheye
    |   view_wrap
    |   'gamma' real ';'
    |   'border' integer ';'
    |   'composite' integer ';'
    |   'vertical' ';'
    |   view_trace_depth
    |   view_viewport
    |   'image' image_type_and_parameters ';'
    ;
```

The `gamma` specifies a gamma value to be applied to each output color sample. If the `gamma` is not specified, it defaults to one. The `border` is the number of pixels of border the raytracer will put around each frame of the output images. The `border` defaults to one. The `composite` parameter specifies the number of dimensions the raytracer should composite into the output image. The default

for `composite` is to cover all dimensions of the output image. If there are more output dimensions to the image than are specified as the `composite`, then the raytracer will generate a sequence of images as it scans through the excess dimensions. The `vertical` flag is used to indicate that composite frames should stack vertically before they stack horizontally.

For a four-dimensional scene, the output will be a three-dimensional image cube. But, as all of the supported image formats are two-dimensional, one has the choice of either outputing a sequence of two-dimensional frames as separate images by specifying a `composite` value of two or as a sequence of two-dimensional frames concatenated together into one larger image by specifying a `composite` value of three. If the `composite` value exceeds the number of dimensions or is zero, it will be set to that number of dimensions of the output image.

The following parameters are used to specify the shape and resolution of the view.

⟨*view bnf*⟩+≡
```
view_units ::=
        'units' real_vector ';'
    |   'unitsPerDimension' real_vector ';'
    ;

view_dpu ::=
        'dotsPerUnit' real_vector ';'
    |   'dpu' real_vector ';'
    |   'dpi' real_vector ';'
    ;

view_depth ::=
        'viewDepth' real ';'
    |   'depth' real ';'
    ;

view_fisheye ::= 'fishEye' real_vector ';' ;

view_wrap ::=
        'wrapAround' ';'
    |   'wrap' ';'
    ;
```

The view is a certain number of units in each dimensions. Those units are specified with either the `units` or `unitsPerDimension` keywords. There is no default for this parameter. The viewer of the resulting image is to be sitting a number of those units away from the image. That distance between the viewer and the resulting image is given as `viewDepth` or `depth`. The default `viewDepth` is twelve units. The resulting output image will have a number of dots per unit of measure specified with `dotsPerUnit`, `dpu`, or `dpi`. The `dotsPerUnit` parameter has no default and must be a vector of the same length as the `units` vector. The view may be fish-eyed in any of its dimensions. The amount of fish-eye effect is given per dimension in the vector `fishEye`. If one wants the fish-eye effect to wrap around beyond 180°, then one must specify wrapping with either the `wrapAround` or `wrap` keywords.

For example, if one were viewing a three-dimensional scene with no fish-eye effect and the desired output image will be five inches wide and three inches high at 300 dots per inch with the viewer expected to be eighteen inches from the page, one would specify:

⟨*view example*⟩≡
```
units [2]< 5, 3 >;
dpu [2]< 300, 300 >;
viewDepth 18;
```

If one wanted specify that this image would be very fish-eyed in the horizontal direction, but not at all fish-eyed in the vertical direction, and that the fish-eye effect be allowed to exceed 180°, one would add:

⟨*view example*⟩+≡
```
fishEye [2]< 15, 0 >;
wrap;
```

The fish-eye factor determines how much the view-angle in that direction is scaled. For example, if the output image were to be ten inches in a given direction for a viewer at a depth of five inches, the viewing angle for that direction would be 90°. If one specified a fish-eye factor of α for that direction, this 90° viewing angle would be stretched up to a $(90 \cdot (1+\alpha))^\circ$ viewing angle. Negative fish-eye factors are acceptable, though letting the fish-eye factor α dip below negative one ends up simply being a mirror-imaged version of the case with a fish-eye factor of $(-1-\alpha)$.

If one is tracing a scene with a large number of highly reflective surfaces, one can get into large amounts of work with very little return for the effort. As such, one can limit the number of times any particular ray will be reflected.

⟨*view bnf*⟩+≡
```
view_trace_depth ::=
            'traceDepth' integer ';'
        |   'maxDepth' integer ';'
        ;
```
The default `traceDepth` is twelve reflections.

The viewport for the view is simply an instance of the color class (which likely contains at least one portal).

⟨*view bnf*⟩+≡
```
view_viewport ::=
            'viewport' '{' color_parameters '}' ';'
        |   'color' '{' color_parameters '}' ';'
        ;
```
The color class is defined in §25 and the portal class in §24.

There are currently three image types supported: `png`, `ppm`, and `accum`.

⟨*view bnf*⟩+≡
```
image_type_and_parameters ::=
            'png' '{' png_image_parameters '}'
        |   'ppm' '{' ppm_image_parameters '}'
        |   'accum' '{' accum_image_parameters '}'
        ;
```
The `png` image type is described in §22. The `ppm` image type is described in §21. The `accum` image type is described in §23.

The view class contains a pointer to the scene to which it belongs.

⟨*View Member Variables*⟩≡
```
const scene* myScene;
```

The view class contains a viewport. The viewport is an instance of the `color` class defined in 25. To be useful in actually raytracing, the viewport's color should make use of a portal.

⟨*View Member Variables*⟩+≡
```
color viewport;
```

The view class represents the way the scene will be output. It defines the extents of the view area in pixels, the angular extents of the view, whether that angle can wrap around beyond 180°, a desired gamma correction, and the number of pixels to put on the border of each subimage in the composite image.

⟨*View Member Variables*⟩+≡
```
mathvec< unsigned int > size;
mathvec< double > angularExtents;
bool wrap;
double gamma;
unsigned int border;
```

The view also tracks how many of the dimensions are supposed to be composited into each output image.

⟨*View Member Variables*⟩+≡
```
unsigned int composite;
```

The view also tracks the maximum raytracing depth allowed.

⟨*View Member Variables*⟩+≡
```
unsigned int traceDepth;
```

Internally, the view class needs to permute the current pixel's coordinates so that it can output the image in the proper scan order.

⟨*View Member Variables*⟩+≡
```
mathvec<unsigned int> permutation;
std::vector<bool> contributesToWidth;
std::vector<bool> needsBorders;
bool vertical;
```

The view class also maintains a list of output images to write to.

⟨*View Member Variables*⟩+≡
```
std::vector< image* > images;
```

2.1 Constructor

The view constructors sets up some simple defaults.

⟨*View constructor defaults*⟩≡
```
wrap( false ), gamma( 1.0 ), border( 1 ), composite( 0 ), traceDepth( 12 )
```

The default constructor does nothing more than prepare the defaults.

⟨*View Method Declarations*⟩≡
```
view( const scene* ss );
```

⟨*View Methods*⟩≡
```
view::view( const scene* ss )
    : myScene( ss ), viewport( ss ), ⟨View constructor defaults⟩
{
}
```

2.2 Destructor

The view destructor loops through and releases each of the output images.

⟨*View Method Declarations*⟩+≡
```
~view( void );
```

⟨*View Methods*⟩+≡
```
view::~view( void )
{
    for ( unsigned int ii=0; ii < this->images.size(); ++ii ) {
        delete this->images[ ii ];
    }
}
```

2.3 Rendering The View

The following method is used to render the view.

⟨*View Method Declarations*⟩+≡
```
void render( void ) const;
```

This method prepares a vector to be a counter of which pixel is currently being rendered. Then, this method determines the width and height of each image in the sequence and the number of images in the sequence. After that, this method loops through every pixel of every output image, calculates the direction to cast the ray for that pixel evaluates the pixel, and writes it to each image instance held by the view.

⟨*View Methods*⟩+≡
```
void
view::render( void ) const
{
    ⟨View Render prepare pixel coordinates⟩
    ⟨View Render determine width and height⟩
    ⟨View Render determine depth⟩
    ⟨View Render set image size⟩

    mathvec< double > position( 0.0, this->size.size() + 1 );
    mathvec< double > direction( this->size.size() + 1 );
    mathvec< double > normal;
    mathvec< double > pixel;

    for ( unsigned int kk=0; kk < depth; ++kk ) {
        for ( unsigned int jj=0; jj < height; ++jj ) {
            for ( unsigned int ii=0; ii < width; ++ii ) {
                if ( this->calculateDirection( coordinates, direction ) ) {
                    ⟨View Render fake normal⟩
                    ⟨View Render evaluate pixel⟩
                    ⟨View Render write pixel⟩
                } else {
                    ⟨View Render write border pixel⟩
                }
                this->incrementDirection( coordinates );
            }
        }
    }
}
```

The render method needs to keep track of the current pixel coordinates so that it will know which direction to cast a ray and when to put in border pixels.

⟨*View Render prepare pixel coordinates*⟩≡
```
mathvec< unsigned int > coordinates( 0U, this->size.size() );
```

To determine the width and height, this method starts by assuming the width and height are both one. Then, it loops through the `size` vector. If the dimension contributes to the width, then the width is multiplied by that dimension. If the dimension contributes to the height, then the height is multiplied by that dimension.

⟨*View Render determine width and height*⟩≡
```
unsigned int width = 1;
unsigned int height = 1;
unsigned int wdirs = 0;
unsigned int hdirs = 0;
for ( unsigned int ii=0; ii < this->composite; ++ii ) {
    bool bb = this->contributesToWidth[ ii ];
    if ( bb ) {
        width  =  width * this->size[ ii ];
        ++wdirs;
    } else {
        height = height * this->size[ ii ];
        ++hdirs;
    }
}
if ( wdirs < 1 ) {
    width += 2 * this->border;
}
if ( hdirs < 1 ) {
    height += 2 * this->border;
}
```

The number of output images in the sequence is the product of all of the sizes not used within each composite image.

⟨*View Render determine depth*⟩≡
```
unsigned int depth=1;
for ( unsigned int ii=this->composite; ii < this->size.size(); ++ii ) {
    depth *= this->size[ ii ];
}
```

Once the width, height, and number of the output images is determined, the `setSize()` method is called on each image instance contained within this view.

⟨*View Render set image size*⟩≡
```
for ( unsigned int ii=0; ii < this->images.size(); ++ii ) {
    this->images[ ii ]->setSize(
        width, height, depth,
        this->viewport.getChannelCount()
    );
}
```

In case the viewport depends on the normal, then the method manufactures one from the direction.

⟨*View Render fake normal*⟩≡
```
normal = direction * -1.0;
```

To evaluate the pixel, this method first evaluates the color of the viewport and then performs gamma correction on it.

⟨*View Render evaluate pixel*⟩≡
```
this->viewport.evaluate(
    position, normal, direction,
    1.0, this->traceDepth,
    pixel
);

unsigned int len = pixel.size();
for ( unsigned int ii=0; ii < len; ++ii ) {
    double val = pixel[ ii ];
    if ( val > 0.0 ) {
        pixel[ ii ] = ::pow( val, this->gamma );
    }
}
```

To write out the pixel, this method loops through all of the contained image instances and passes the pixel along.

⟨*View Render write pixel*⟩≡
```
for ( unsigned int ll=0; ll < this->images.size(); ++ll ) {
    this->images[ ll ]->add( pixel );
}
```

To write out a border pixel, this method loops through all of the contained image instances and passes the pixel along.

⟨*View Render write border pixel*⟩≡
```
for ( unsigned int ll=0; ll < this->images.size(); ++ll ) {
    this->images[ ll ]->addBorderPixel();
}
```

2.4 Calculating The Viewing Direction From The Pixel Coordinates

At each step of its rendering loop, the view has to turn the pixel coordinates into a direction to cast a ray. This method handles that process. It returns false if the pixel should be a border pixel or if wrapping is not turned on and the cast ray would be more than 180° off of directly forward.

⟨*View Method Declarations*⟩+≡

```
bool calculateDirection(
    const mathvec< unsigned int >& coordinates,
    mathvec< double >& direction
) const;
```

This method creates a direction vector based upon the current pixel's coordinates. If the current pixel's coordinate on a given axis is the first pixel of eleven along that axis, the pixel will be at the beginning of the angular extents for that dimension. If the current pixel's coordinate on a given axis is the sixth of eleven along that axis, the pixel will be at the center of the angular extents for that dimension. All of these angles are the measured away from the x-axis and toward the given axis.

⟨*View Methods*⟩+≡

```
bool
view::calculateDirection(
    const mathvec< unsigned int >& coordinates,
    mathvec< double >& direction
) const {
    unsigned int dims = coordinates.size();

    direction[ 0 ] = 0.0;

    ⟨View calculate direction – scale counter⟩
    ⟨View calculate direction – normalize counter⟩
}
```

The following code starts building a direction vector. At each coordinate, the proportion that this pixel number is of the total extent of pixels along this axis is calculated and recorded in the direction vector. If this pixel falls within the border, however, this method returns false immediately so that the calling code can emit a border pixel.

⟨*View calculate direction – scale counter*⟩≡

```
for ( unsigned int ii=0; ii < dims; ++ii ) {
    unsigned int index = this->permutation[ ii ] + 1;

    unsigned int bb = 0;
    if ( this->needsBorders[ ii ] ) {
        bb = this->border;
    }

    double is = (double)( this->size[ ii ] - 2*bb );
    double ss = is / 2.0 - 0.5;

    if ( ss < 0.000001 ) {
        direction[ index ] = 0.0;
    } else {
        unsigned int cc = coordinates[ ii ];
        double tt = (double)( cc - bb ) - ss;

        if ( cc < bb || cc + bb >= this->size[ ii ] ) {
            return false;
        }

        direction[ index ] = this->angularExtents[ ii ] * tt / ss;
    }
}
```

The whole direction vector at this point represents the angular displacements along each axis. In order to “normalize” this, the following code takes this whole vector to represent a single angular displacement in the direction given. Then, the vector is scaled so that the direction is multiplied by the sine of the angle while the forward coordinate becomes the cosine. If wrapping is not turned on and this represents an angle greater than π radians, this method returns false just as it would if this were a border pixel.

⟨*View calculate direction – normalize counter*⟩≡
```
double mag = direction.magnitude();

if ( this->wrap == false && mag > M_PI ) {
    return false;
} else {
    if ( mag < 0.000001 ) {
        direction *= 0.0;
        direction[ 0 ] = 1.0;
    } else {
        direction *= -::sin( mag ) / mag;
        direction[ 0 ] = ::cos( mag );
    }
    return true;
}
```

2.5 Incrementing The Pixel Coordinates

This method increments the coordinates of the pixel. It does this, making sure that the coordinates “roll over” at appropriate points. This method considers the coordinate vector in order. It is shuffled by the permutation when it is referred to, rather than here.

⟨*View Method Declarations*⟩+≡
```
void incrementDirection( mathvec< unsigned int >& coordinates ) const;
```

The loop increments the first element of the coordinates. If that element overflows, it is reset to zero and the loop continues on. If the element does not overflow, then the loop exits immediately. In this way, the early entries in the vector always scan faster than the later entries.

⟨*View Methods*⟩+≡
```
void
view::incrementDirection( mathvec< unsigned int >& coordinates ) const
{
    unsigned int dims = coordinates.size();

    for ( unsigned int ii=0; ii < dims; ++ii ) {
        if ( ++coordinates[ ii ] < this->size[ ii ] ) {
            break;
        } else {
            coordinates[ ii ] = 0;
        }
    }
}
```

2.6 Loading A View

The following method allows one to read in a view from a stream.

⟨*View Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, view& cc );
```

There are a variety of keywords recognized in the view. One can specify the size in universe units of each scan direction with the `unitsPerDimension` or `units` vector. One can specify the number of pixels to devote to each unit with the `dotsPerUnit`, `dpu`, or `dpi` vector. One can specify the observer's depth from the image in universe units with the `viewDepth` or `depth` value. One can specify the amount of fish-eye effect to put in each axis using the `fishEye` vector. One can specify that wrapping should take place with the `wrapAround` or `wrap` keyword. One can specify an amount of gamma correction to be performed on the image pixels with the `gamma` value. One can specify the number of units of border to put around frames of a composite image using the `border` value. One can specify the number of dimensions to composite together with the `composite` value. One can specify the maximum number of reflections and refractions one should do in tracing a ray using the `traceDepth` or `maxDepth` value. One specifies the view into the world by specifying an instance of the color class (defined in §25) using the `viewport` or `color` tag. And, one can specify any number of output images by specifying an instantiation of the image class (defined in §20) using the `image` tag.

⟨*View Methods*⟩+≡

```
std::istream&
operator >> ( std::istream& stream, view& vv )
{
    mathvec< double > units;
    mathvec< double > dpu;
    mathvec< double > fishEye;
    double viewDepth = 12.0;

    vv.vertical = false;

    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "unitsPerDimension" || tok == "units" ) {
            stream >> units >> semicolon;
        } else if ( tok == "dotsPerUnit" || tok == "dpu" || tok == "dpi" ) {
            stream >> dpu >> semicolon;
        } else if ( tok == "viewDepth" || tok == "depth" ) {
            stream >> viewDepth >> semicolon;
```

```
        } else if ( tok == "fishEye" ) {
            stream >> fishEye >> semicolon;
        } else if ( tok == "wrapAround" || tok == "wrap" ) {
            vv.wrap = true;
            stream >> semicolon;
        } else if ( tok == "gamma" ) {
            stream >> vv.gamma >> semicolon;
        } else if ( tok == "border" ) {
            stream >> vv.border >> semicolon;
        } else if ( tok == "composite" ) {
            stream >> vv.composite >> semicolon;
        } else if ( tok == "vertical" ) {
            vv.vertical = true;
            stream >> semicolon;
        } else if ( tok == "traceDepth" || tok == "maxDepth" ) {
            stream >> vv.traceDepth >> semicolon;
        } else if ( tok == "viewport" || tok == "color" ) {
            stream >> vv.viewport >> semicolon;
        } else if ( tok == "image" ) {
            image* nn = image::read( stream );
            vv.images.insert( vv.images.end(), nn );
            stream >> semicolon;
        } else {
            in.ignore( "VIEW", tok.c_str() );
        }
    }

    ⟨View check dimensions⟩
    ⟨View prepare permutation index⟩
    ⟨View calculate sizes⟩

    vv.viewport.setDimensions( dims+1 );

    return stream;
}
```

The following code verifies that the units per dimension array and the dots per unit array both have the same length. Then, it goes on to clip the number of composite dimensions to the appropriate range and expand the `fishEye` vector to the appropriate size.

⟨*View check dimensions*⟩≡
```
unsigned int dims = units.size();
if ( dims == 0 ) {
    std::cerr << "View: length dimensions are zero" << std::endl;
    return stream;
}

dpu.resize( dims, 1.0 );

if ( vv.composite == 0 || vv.composite > dims ) {
    vv.composite = dims;
}

unsigned int ii;
fishEye.resize( dims, 0.0 );
```

For the composite dimensions, the odd dimensions will contribute to the width and the even dimensions will contribute to the height. Except that things get a little wonkier when the `vertical` flag is set. In particular, we're going to swap the last odd and even dimensions. The code involved will need to ensure that the width dimensions are the fastest scanning. To that end, the `permutation` array is prepared to re-order the pixel coordinates.

⟨*View prepare permutation index*⟩≡
```
vv.permutation.resize( dims );
vv.contributesToWidth.resize( vv.composite );
vv.needsBorders.resize( dims, false );

unsigned int evenDims = vv.composite / 2;
unsigned int oddDims = vv.composite - evenDims;

for (ii = 0; ii < oddDims; ++ii) {
    vv.permutation[ii] = ii * 2;
    vv.contributesToWidth[ ii ] = true;
}

for (ii = oddDims; ii < vv.composite; ++ii) {
    vv.permutation[ii] = ( ii - oddDims ) * 2 + 1;
    vv.contributesToWidth[ ii ] = false;
}

for (ii = vv.composite; ii < dims; ++ii) {
    vv.permutation[ii] = ii;
}

if ( vv.vertical && oddDims > 0 && evenDims > 0 ) {
    ⟨view adjust for vertical mode⟩
}

unsigned int wdims = 0;
unsigned int hdims = 0;
for ( ii = 0; ii < vv.composite; ++ii ) {
    if ( vv.contributesToWidth[ ii ] ) {
        ++wdims;
    } else {
        ++hdims;
    }
}

for ( ii = 0; ii < vv.composite; ++ii ) {
    if ( vv.contributesToWidth[ ii ] ) {
        vv.needsBorders[ ii ] = ( --wdims > 0 );
    } else {
```

```
        vv.needsBorders[ ii ] = ( --hdims > 0 );
    }
}
for ( ii = vv.composite; ii < dims; ++ii ) {
    vv.needsBorders[ ii ] = false;
}
```

First, if there are an odd number of composite dimensions, slide the last odd dimensions to the end of the evens. Then, swap the matching odd and evens.

⟨*view adjust for vertical mode*⟩≡
```
if ( ( vv.composite & 1 ) == 1 ) {
    unsigned int tt = vv.permutation[ oddDims - 1 ];
    for ( ii=oddDims; ii < vv.composite; ++ii ) {
        vv.permutation[ ii - 1 ] = vv.permutation[ ii ];
    }
    vv.permutation[ vv.composite-1 ] = tt;
    vv.contributesToWidth[ oddDims - 1 ] = false;
    --oddDims;
    ++evenDims;
}
for ( ii=1; ii < oddDims; ++ii ) {
    unsigned int tt = vv.permutation[ ii ];
    vv.permutation[ ii ] = vv.permutation[ ii + oddDims ];
    vv.permutation[ ii + oddDims ] = tt;
}
```
Actually, we could just save `oddDims` rather than `contributesToWidth`, but the code feels much more readable this way. The first `oddDims` members of `contributesToWidth` should be `true` whilst the remaining are `false`.

From the `units` and `dpu`, the following code determines the size of the output image. At the same time, it uses the viewing depth and the `fishEye` array to compute the angular viewing extents in each dimension.

⟨*View calculate sizes*⟩≡
```
vv.size.resize( dims );
vv.angularExtents.resize( dims );

for (ii = 0; ii < dims; ++ii) {
    unsigned int permuted = vv.permutation[ii];
    double un = units[permuted];
    double len = dpu[permuted] * un;
    double baseAngle;

    vv.size[ ii ] = (unsigned int)(::fabs( len ) + 0.5);
    if ( permuted+1 != vv.composite && permuted+2 != vv.composite ) {
        vv.size[ ii ] += 2 * vv.border;
    }

    baseAngle = atan2( un, 2.0 * viewDepth );
    vv.angularExtents[ ii ] = baseAngle * ( 1.0 +  fishEye[permuted]);
}
```

2.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*View Class Declaration*⟩≡
```
class view {
protected:
    ⟨View Member Variables⟩
public:
    ⟨View Method Declarations⟩
public:
    static const char* id;
};
```

2.8 Source Files

The following sections assemble the source files for the view class from the chunks above.

2.8.1 `view.h`

The header file for the view class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header files for the `color` and `image` classes and then incorporates the class defined in §2.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*view.h*⟩≡
```
#ifndef _NKLEIN_VIEW_H_
#define _NKLEIN_VIEW_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "color.h"
#include "image.h"

    ⟨View Class Declaration⟩

#endif /*_NKLEIN_VIEW_H_*/
```

2.8.2 view.cc

Herein, the implementations of the methods of the view class are included. The implementation requires the math-library header for the definition of sines and cosines, the header for the input subsystem, and the declaration of the view class itself.

⟨*view.cc*⟩≡

```
#include <math.h>
#include "input.h"
#include "view.h"

const char* view::id = "NKVERSION: view( 2.1.2007.05.17 )";

⟨View Methods⟩
```

Chapter 3

Universe

A universe instance has a certain number of spatial dimensions. It contains an optional sky color, an optional color of the ambient light, zero or more point light sources, and zero or more objects.

⟨*universe bnf*⟩≡

```
universe_parameters ::=
        universe_parameters universe_param
    |   /* empty */
    ;

universe_param ::=
        universe_dimensions
    |   universe_sky
    |   universe_ambient
    |   quickRender ';'
    |   'light' '{' light_params '}' ';'
    |   'object' object_type_and_params ';'
    ;

universe_dimensions ::=
        'dimensions' integer ';'
    |   'dims' integer ';'
    ;

universe_sky ::=
        'skyColor' '{' color_params '}' ';'
    |   'sky' '{' color_params '}' ';'
    ;

universe_ambient ::=
        'ambientLight' '{' color_params '}' ';'
    |   'ambient' '{' color_params '}' ';'
```

```
        ;
```

Lights are described in §8. Objects are described in §9. Color parameters are described in §25.

The universe instance holds a pointer to the scene to which it belongs. Colors within the scene can contain portals into other universes in the scene. The scene instance is used to match up those portals to the universes they reference.

⟨*Universe Member Variables*⟩≡

```
const scene* myScene;
```

The universe class specifies its number of spatial dimensions. It also contains point light sources and objects. Additionally, it defines ambient light, a sky color, and a quick rendering flag.

⟨*Universe Member Variables*⟩+≡

```
unsigned int dimensions;
std::vector< light* > lights;
std::vector< object* > objects;
color ambientLight;
bool gotAmbient;
color skyColor;
bool quickRender;
```

3.1 Constructor

The default constructor does very little. Mostly, it passes the scene pointer on to its constituent colors. The only other thing it does is record that it has not received a color for the ambient light in the universe.

⟨*Universe Method Declarations*⟩≡

```
universe( const scene* ss );
```

⟨*Universe Methods*⟩≡

```
universe::universe( const scene* ss )
    : myScene( ss ), ambientLight( ss ),
        gotAmbient( false ), skyColor( ss ), quickRender( false )
{
}
```

3.2 Destructor

The destructor cleans out the lights and objects arrays.

⟨*Universe Method Declarations*⟩+≡
```
~universe( void );
```

⟨*Universe Methods*⟩+≡
```
universe::~universe( void )
{
    for ( unsigned int ii=0; ii < this->objects.size(); ++ii ) {
        delete objects[ ii ];
    }
    for ( unsigned int ii=0; ii < this->lights.size(); ++ii ) {
        delete lights[ ii ];
    }
}
```

3.3 Getting And Setting The Number Of Dimensions

The following method allows the portals to query the number of spatial dimensions for this universe so that the portal can upgrade (or downgrade) its vectors to the proper number of dimensions for this universe.

⟨*Universe Method Declarations*⟩+≡
```
inline unsigned int getDimensions( void ) const {
    return this->dimensions;
};
```

When the number of spatial dimensions for the universe are set, this information is passed along to all of the constituent lights and objects so that they can resize their position vectors, scaling vectors, and orientation matrices accordingly.

⟨*Universe Method Declarations*⟩+≡
```
void setDimensions( unsigned int dims );
```

⟨*Universe Methods*⟩+≡
```
void
universe::setDimensions( unsigned int dims )
{
    this->dimensions = dims;

    for ( unsigned int ii=0; ii < this->lights.size(); ++ii ) {
        this->lights[ ii ]->setDimensions( this->dimensions );
    }

    for ( unsigned int ii=0; ii < this->objects.size(); ++ii ) {
        this->objects[ ii ]->setDimensions( this->dimensions );
    }

    this->ambientLight.setDimensions( dims );
    this->skyColor.setDimensions( dims );
}
```

3.4 Casting A Ray Into The Universe

The following method allows one to cast a ray into a universe.

⟨*Universe Method Declarations*⟩+≡
```
bool evaluate(
    const mathvec< double >& position,
    const mathvec< double >& direction,
    double contribution, unsigned int height,
    mathvec< double >& pixel
) const;
```

This method assumes that the index of refraction of the air in the universe is 1.0. Then, it turns around and invokes the method which follows to actually trace the ray.

⟨*Universe Methods*⟩+≡
```
bool
universe::evaluate(
    const mathvec< double >& position,
    const mathvec< double >& direction,
    double contribution, unsigned int height,
    mathvec< double >& pixel
) const {
    return this->evaluate(
        position, direction, 1.0, contribution, height, pixel
    );
}
```

If one prefers to specify the ambient index of refraction in the universe, one can do so using this method instead. In particular, refracted rays that are interior to an object will specify the index of refraction of the object as opposed to that of air.

⟨*Universe Method Declarations*⟩+≡
```
bool evaluate(
    const mathvec< double >& position,
    const mathvec< double >& direction,
    double indexOfRefraction,
    double contribution, unsigned int height,
    mathvec< double >& pixel
) const;
```

If the `height` parameter is zero, then this ray has already undergone as many reflections, refractions, or portal pass-throughs as allowed by the view. If the contribution left for this pixel to provide is tiny, this pixel is abandoned. Otherwise, the universe attempts to find the closest point of intersection of this ray with an object in the universe. After that, it determines the color of that spot on that object based upon the lighting in the universe and the color of the object.

⟨*Universe Methods*⟩+≡

```
bool
universe::evaluate(
    const mathvec< double >& position,
    const mathvec< double >& direction,
    double indexOfRefraction,
    double contribution, unsigned int height,
    mathvec< double >& pixel
) const {
    if ( height == 0 || contribution < 0.000001 ) {
        return false;
    } else {
        ⟨Universe find closest intersection⟩
        if ( this->quickRender ) {
            ⟨Universe fudge color⟩
        } else {
            ⟨Universe determine color⟩
        }
        return true;
    }
}
```

To find the closest intersection of the ray with an object in the universe, this method employs a helper class. It instantiates one instance of this helper class for every object. Each instance of the helper class is responsible for intersecting the ray with the given object. Then, this code loops through and checks to see which intersection was the closest to the origin of the ray. The helper class for intersections is defined in §4.

⟨*Universe find closest intersection*⟩≡
```
pixel *= 0.0;

⟨Universe prepare intersection work⟩

intersectionWork* closest = 0;
double closestDistance = MAXFLOAT;

for ( unsigned int ii=0; ii < this->objects.size(); ++ii ) {
    if ( intersections[ ii ].wasHit()
    && intersections[ ii ].getDistance() < closestDistance ) {
        closest = &intersections[ ii ];
        closestDistance = closest->getDistance();
    }
}
```

If there were no intersections, then the pixel is filled in from the sky color and returned.

⟨*Universe find closest intersection*⟩+≡
```
if ( closest == 0 ) {
    mathvec< double > normal = direction * -1.0;
    this->skyColor.evaluate(
        position, normal, direction, contribution, height, pixel
    );
    pixel *= contribution;
    return true;
}
```

Otherwise, the point of intersection is copied and the helper class instances are released.

⟨*Universe find closest intersection*⟩+≡
```
intersection hit = closest->getIntersection();
intersections.resize(0);
```

To prepare the helper classes which intersect the ray with the objects, this method simply creates an array of the intersections big enough for one for each object. Then, each helper instance is intialized with the vector and the object for which that helper instance is responsible.

⟨*Universe prepare intersection work*⟩≡

```
std::vector< intersectionWork > intersections( this->objects.size() );
for ( unsigned int ii=0; ii < this->objects.size(); ++ii ) {
    intersections[ ii ].set( position, direction, this->objects[ ii ] );
}
```

To fudge the color for quick rendering, we simply evaluate the color of the object as if there were 100ambient white light and then scale this color by the dot-product of the normal with the direction.

⟨*Universe fudge color*⟩≡

```
mathvec< double > ndir = hit.getNormal();
double dot = -( ndir ^ direction );
dot = 1.0 - 0.75 * ( 1.0 - dot );
if ( dot < 0.25 ) {
    dot = 0.25;
}
hit.evaluate( contribution, height, pixel );
pixel *= dot;
```

Once a hit is recorded, then the reflected ray is calculated. This reflected ray is passed to a helper class which evaluates the color contribution from the reflected ray. At the same time, another helper class is instantiated to evaluate the direciton and color of the refracted ray. The helper class for reflected rays is defined in §5. The helper class for refracted rays is defined in §6.

⟨*Universe determine color*⟩≡

```
mathvec< double > ndir = hit.getNormal();
mathvec< double > rdir = hit.getDirection(); // reflected ray

ndir *= -2.0 * ( rdir ^ ndir );
rdir += ndir;

reflectionWork reflected( rdir, hit, contribution, height, this );
refractionWork refracted( hit, contribution, height, indexOfRefraction, this );
```

In addition, there is a helper class instance for each light in the universe which is used to determine the color contribution of a given light source on the point of intersection. Each of these instances is given a reference to the reflected ray, the intersection, its light, and other parameters needed to check the contribution of the light source. This helper class will call back to this universe to see if the path from the light to the current intersection spot is blocked by another object. The helper class for lighting contribution is defined in §7.

⟨*Universe determine color*⟩+≡
```
std::vector< shadowWork > shadows( this->lights.size() );
for ( unsigned int ii=0; ii < shadows.size(); ++ii ) {
    shadows[ ii ].set(
        &rdir, &hit, this->lights[ ii ], contribution, height, this
    );
}
```

The incoming pixel is resized to the appropriate number of color channels. Then, the ambient light's contribution is calculated.

⟨*Universe determine color*⟩+≡
```
if ( pixel.size() < hit.getColor()->getChannelCount() ) {
    pixel.resize( hit.getColor()->getChannelCount() );
}

if ( this->gotAmbient ) {
    mathvec< double > normal = direction * -1.0;
    hit.evaluate( contribution, height, pixel );
    mathvec< double > ambientPixel;
    this->ambientLight.evaluate(
        position, normal, direction, contribution, height, ambientPixel
    );
    pixel *= ambientPixel;
} else {
    pixel *= 0.0;
}
```

After this, contribution of the reflected ray and the refracted ray are added in. Then, the contribution from each point light source in the universe is figured in. This whole amount then is scaled down based upon the contribution this ray was supposed to have.

⟨*Universe determine color*⟩+≡

```
double reflectedAmount = reflected.getAmount();
if ( reflectedAmount > 0.000001 ) {
    pixel += reflectedAmount * reflected.getPixel();
}

double refractedAmount = refracted.getAmount();
if ( refractedAmount > 0.000001 ) {
    pixel += refractedAmount * refracted.getPixel();
}

for ( unsigned int ii=0; ii < shadows.size(); ++ii ) {
    double amount = shadows[ ii ].getAmount();
    if ( amount > 0.000001 ) {
        double diffuseness = 1.0 - reflectedAmount - refractedAmount;
        pixel += amount * shadows[ ii ].getPixel( diffuseness );
    }
}

pixel *= contribution;
```

3.5 Checking For Shadows

There were helper class instances used above whose purpose was to evaluate the contribution of a particular light toward the color of an intersection point. That helper class calculates the vector from the point of intersection toward the light. This method is used to be sure that that vector does not intersect any opaque objects in the universe before it gets to the light.

⟨*Universe Method Declarations*⟩+≡

```
void checkShadow(
    const mathvec< double >& position,
    const mathvec< double >& direction,
    double lampDistance,
    double contribution, unsigned int height,
    double& amount,
    mathvec< double >& pixel
) const;
```

This method takes advantage of the same code as the `evaluate()` method of the preceding section to instantiate helper instances to check for intersections with objects. Then, it evaluates the color contribution of this light. At the time this method is invoked, the pixel has already been initialized with the color of the light.

⟨*Universe Methods*⟩+≡

```
void
universe::checkShadow(
    const mathvec< double >& position,
    const mathvec< double >& direction,
    double lampDistance,
    double contribution,
    unsigned int height,
    double& amount,
    mathvec< double >& pixel
) const
{
    ⟨Universe prepare intersection work⟩
    ⟨Universe accumulate colors⟩
}
```

For every transparent object that was hit on the way to the lamp, the pixel color is filtered by the color and transparency of the object. If an opaque object is encountered, then this method clears the contribution amount for this lamp and bails out.

⟨*Universe accumulate colors*⟩≡
```
for ( unsigned int ii=0; ii < this->objects.size(); ++ii ) {
    if ( intersections[ ii ].wasHit()
    && intersections[ ii ].getDistance() < lampDistance ) {
        const color* cc = intersections[ ii ].getIntersection().getColor();
        double tt = cc->getTransparency();
        if ( tt < 0.000001 ) {
            amount = 0.0;
            break;
        } else {
            mathvec< double > objPixel;
            if ( tt < 0.999999 ) {
                intersections[ ii ].getIntersection().evaluate(
                    contribution * tt, height, objPixel
                );
                objPixel *= ( 1.0 - tt );
                objPixel *= pixel;
                pixel *= tt;
                pixel += objPixel;
            } else {
                pixel *= tt;
            }
            amount *= tt;
        }
    }
}
```

3.6 Loading A Universe

The following method allows one to read in a universe from a stream.

⟨*Universe Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, universe& cc );
```

There are a variety of keywords recognized in the universe. One can specify the number of spatial dimensions in the universe with the `dimensions` or `dims` value. One specifies the ambient light by specifying an instance of the color class (defined in §25) using the `ambientLight` or `ambient` tag. One specifies the sky color by specifying an instance of the color class using the `skyColor` or `sky` tag. One specifies a light by specifying an instance of the light class (defined in §8) using the `light` tag. One specifies an object by specifying an instance of the object class (defined in §9). If the `quickRender` flag is present, then there will be no reflected ray, refracted ray, or shadow checking at all.

⟨*Universe Methods*⟩+≡
```
std::istream&
operator >> ( std::istream& stream, universe& uu )
{
    input in( stream );
    unsigned int dims = 2;

    uu.gotAmbient = false;

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "dimensions" || tok == "dims" ) {
            stream >> dims >> semicolon;
        } else if ( tok == "ambientLight" || tok == "ambient" ) {
            stream >> uu.ambientLight >> semicolon;
            uu.gotAmbient = true;
        } else if ( tok == "skyColor" || tok == "sky" ) {
            stream >> uu.skyColor >> semicolon;
        } else if ( tok == "quickRender" ) {
            stream >> semicolon;
            uu.quickRender = true;
        } else if ( tok == "light" ) {
            light* nn = new light( uu.myScene );
            stream >> *nn >> semicolon;
            uu.lights.insert( uu.lights.end(), nn );
        } else if ( tok == "object" ) {
            object* nn = object::read( stream, uu.myScene );
            if ( nn != 0 ) {
                uu.objects.insert( uu.objects.end(), nn );
```

```
            }
        } else {
            in.ignore( "UNIVERSE", tok.c_str() );
        }
    }

    uu.setDimensions( dims );

    return stream;
}
```

3.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Universe Class Declaration*⟩≡

```
class universe {
protected:
    ⟨Universe Member Variables⟩
public:
    ⟨Universe Method Declarations⟩
public:
    static const char* id;
};
```

3.8 Source Files

The following sections assemble the source files for the `universe` class from the chunks above.

3.8.1 universe.h

The header file for the universe class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header files for the `light` and `object` classes and then incorporates the class defined in §3.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*universe.h*⟩≡

```
#ifndef _NKLEIN_UNIVERSE_H_
#define _NKLEIN_UNIVERSE_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "light.h"
#include "object.h"

    ⟨Universe Class Declaration⟩

#endif /*_NKLEIN_UNIVERSE_H_*/
```

3.8.2 universe.cc

Herein, the implementations of the methods of the `universe` class are included as well as the the declarations and implementations of the helper classes. The implementation requires the definition of `MAXFLOAT` which it may get from the math header or from the `values.h` header, the header for the input subsystem, the header for the threading subsystem, and the declaration of the universe class itself. The implementation file also includes the definitions and implementations of all of the helper classes it requires.

⟨*universe.cc*⟩≡
```
#include <math.h>
#ifndef MAXFLOAT
#include <values.h>
#endif
#include "input.h"
#include "nkthread.h"
#include "universe.h"

const char* universe::id = "NKVERSION: universe( 2.1.2007.05.17 )";
```

⟨*Universe Intersection Work Class Declaration*⟩
⟨*Universe Reflection Work Class Declaration*⟩
⟨*Universe Refraction Work Class Declaration*⟩
⟨*Universe Shadow Work Class Declaration*⟩
⟨*Universe Methods*⟩

Chapter 4

Universe Intersection Work

This is a helper class used by the universe class from §3 to check whether a given ray intersects a given object. This helper class descends directly from the threading system's `work` class (define in §38) so that this work could potentially be performed in a subthread.

This class will store references to the ray parameters and the object. It will record the intersection of the ray and the object if they intersect and track whether or not they did intersect.

⟨*Universe Intersection Work Member Variables*⟩≡

```
const mathvec< double >* position;
const mathvec< double >* direction;
const object* obj;
intersection hit;
bool gotHit;
```

4.1 Constructor

The constructor does nothing in particular.

⟨*Universe Intersection Work Method Declarations*⟩≡

```
intersectionWork( void ) {
};
```

4.2 Destructor

The destructor waits for the work to finish if it has not already done.

⟨*Universe Intersection Work Method Declarations*⟩+≡

```
virtual ~intersectionWork( void ) {
    this->wait();
};
```

4.3 Setting Up The Work

This method is used to give the ray and object to this class and to have the class begin work checking whether the ray intersects the object or not.

⟨*Universe Intersection Work Method Declarations*⟩+≡

```
void set(
    const mathvec< double >& pp, const mathvec< double >& dd,
    const object* oo
) {
    this->position = &pp;
    this->direction = &dd;
    this->obj = oo;
    this->gotHit = false;
    this->start();
};
```

4.4 Performing The Work

To actually perform the work, this method invokes the intersection method on the object. It ignores any hits that are beyond half way to virtual infinity (though I'm not entirely sure it should).

⟨*Universe Intersection Work Method Declarations*⟩+≡

```
virtual void perform( void ) {
    this->gotHit = this->obj->intersect(
        *this->direction, *this->position, this->hit
    );
    if ( this->gotHit && this->hit.getDistance() > MAXFLOAT / 2.0 ) {
        this->gotHit = false;
    }
};
```

4.5 Checking The Work

When someone wants to know whether the ray intersected the object, they invoke the following method. The method makes sure the work has been completed and then returns the answer.

⟨*Universe Intersection Work Method Declarations*⟩+≡
```
bool wasHit( void ) {
    this->wait();
    return this->gotHit;
};
```

If someone is curious about the distance the hit was along the ray, they invoke the following method. The method makes sure the work has been completed and then returns the answer.

⟨*Universe Intersection Work Method Declarations*⟩+≡
```
double getDistance( void ) {
    this->wait();
    return this->hit.getDistance();
};
```

If someone is curious about the actual intersection instance they invoke the following method. The method makes sure the work has been completed and then returns the answer.

⟨*Universe Intersection Work Method Declarations*⟩+≡
```
const intersection& getIntersection( void ) {
    this->wait();
    return this->hit;
};
```

4.6 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Universe Intersection Work Class Declaration*⟩≡
```
class intersectionWork : public thread::work {
protected:
    ⟨Universe Intersection Work Member Variables⟩
public:
    ⟨Universe Intersection Work Method Declarations⟩
public:
    static const char* id;
};
const char* intersectionWork::id = "NKVERSION: intersectionWork( 2.1.2007.05.17 )";
```

Chapter 5

Universe Reflection Work

This is a helper class used by the universe class from §3 to check the color contribution of the ray reflected from a given intersection point. This helper class descends directly from the threading system's `work` class (define in §38) so that this work could potentially be performed in a subthread.

This class will store a reference to the ray direction, a reference to the intersection information, the contribution, the number of levels of raytracing still available, and a pointer to the universe. It will record the resulting pixel color and amount of the contribution.

⟨*Universe Reflection Work Member Variables*⟩≡

```
const mathvec< double >& reflectedDirection;
const intersection& hit;
unsigned int height;
double contribution;
const universe* univ;
double amount;
mathvec< double > pixel;
```

5.1 Constructor

The constructor initializes the reflected direction reference, the hit reference, the raytracing height remaining, the contribution of this ray, and the pointer to the universe. Then, it checks the reflectiveness of the object. It then decides whether or not the contribution is worth checking. If it is, then processing is started. Otherwise, the amount is zeroed and the lock on this work is released.

⟨*Universe Reflection Work Method Declarations*⟩≡

```
reflectionWork(
    const mathvec< double >& rr,
    const intersection& hh, double cn, unsigned int hg, const universe* uu
) : reflectedDirection( rr ), hit( hh ),
    height( hg ), contribution( cn ), univ( uu )
{
    const color* cc = hit.getColor();
    this->amount = cc->getReflectiveness();
    if ( this->contribution * this->amount > 0.000001 ) {
        this->start();
    } else {
        this->amount = 0.0;
        this->finish();
    }
};
```

5.2 Destructor

The destructor waits for the work to be completed if it has not already finished.

⟨*Universe Reflection Work Method Declarations*⟩+≡

```
virtual ~reflectionWork( void ) {
    this->wait();
};
```

5.3 Performing The Work

To do the work, a starting point for the reflected ray is determined. Then, the new ray is traced in the universe.

⟨*Universe Reflection Work Method Declarations*⟩+≡
```
virtual void perform( void ) {
    mathvec< double > pos = this->hit.getPosition();

    pos += this->reflectedDirection * 0.000001;

    if ( this->univ->evaluate(
        pos, this->reflectedDirection,
        this->contribution, this->height-1, this->pixel
    ) == false ) {
        this->amount = 0.0;
    }
};
```

5.4 Checking The Work

When someone wants to know how much the reflected ray should contribute, they invoke the following method. The method makes sure the work has been completed and then returns the answer.

⟨*Universe Reflection Work Method Declarations*⟩+≡
```
inline double getAmount( void ) {
    this->wait();
    return this->amount;
};
```

When someone wants to know what color the reflected ray should contribute, they invoke the following method. The method makes sure the work has been completed and then returns the answer.

⟨*Universe Reflection Work Method Declarations*⟩+≡
```
const mathvec< double >& getPixel( void ) {
    this->wait();
    return this->pixel;
};
```

5.5 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Universe Reflection Work Class Declaration*⟩≡

```
class reflectionWork : public thread::work {
protected:
    ⟨Universe Reflection Work Member Variables⟩
public:
    ⟨Universe Reflection Work Method Declarations⟩
public:
    static const char* id;
};

const char* reflectionWork::id = "NKVERSION: reflectionWork( 2.1.2007.05.17 )";
```

Chapter 6

Universe Refraction Work

This is a helper class used by the universe class from §3 to check the color contribution of the ray refracted from a given intersection point. This helper class descends directly from the threading system's `work` class (define in §38) so that this work could potentially be performed in a subthread.

This class will store a reference to the intersection information, the contribution, the number of levels of raytracing still available, the contribution of this ray, the index of refraction outside the object hit, and a pointer to the universe. It will record the resulting pixel color and amount of the contribution.

⟨*Universe Refraction Work Member Variables*⟩≡

```
const intersection& hit;
unsigned int height;
double contribution;
double indexOfRefraction;
const universe* univ;
double amount;
mathvec< double > pixel;
```

6.1 Constructor

The constructor initializes the hit reference. The constructor initializes the hit reference, the raytracing height remaining, the contribution of this ray, the ambient index of refraction, and the pointer to the universe. Then, it checks the transparency of the object. It then decides whether or not the contribution is worth checking. If it is, then processing is started. Otherwise, the amount is zeroed and the lock on this work is released.

⟨*Universe Refraction Work Method Declarations*⟩≡

```
refractionWork(
    const intersection& hh, double cn, unsigned int hg, double ior,
    const universe* uu
) : hit( hh ), height( hg ), contribution( cn ),
    indexOfRefraction( ior ), univ( uu )
{
    const color* cc = hit.getColor();
    this->amount = cc->getTransparency();
    if ( this->contribution * this->amount > 0.000001 ) {
        this->start();
    } else {
        this->amount = 0.0;
        this->finish();
    }
};
```

6.2 Destructor

The destructor waits for the work to complete for proceeding.

⟨*Universe Refraction Work Method Declarations*⟩+≡

```
virtual ~refractionWork( void ) {
    this->wait();
};
```

6.3 Performing The Work

Assuming that the index of refraction of the current medium is i_1 and that of the next medium is i_2, the refracted vector $\vec{r}_2$ is related to the incoming directional vector $\vec{r}_1$ via the normal at the point of contact $\vec{n}$ by the Snell's law:

$$\frac{i_1}{i_2} = \frac{|\vec{r}_2 - (\vec{r}_2 \cdot \vec{n})\vec{n}|}{|\vec{r}_1 - (\vec{r}_1 \cdot \vec{n})\vec{n}|} \tag{6.1}$$

The resulting vector $\vec{r}_2$ is in the same plane as $\vec{r}_1$ and $\vec{n}$. Thus, it can be expressed as a linear combination of the two:

$$\vec{r}_2 = \alpha\vec{r}_1 + \beta\vec{n}$$

Substituting that into equation 6.1 and using the fact that both $\vec{r}_1$ and $\vec{n}$ are unit vectors, one obtains:

$$\begin{aligned} \frac{i_1}{i_2} &= \frac{|\alpha\vec{r}_1 + \beta\vec{n} - ((\alpha\vec{r}_1 + \beta\vec{n}) \cdot \vec{n})\,\vec{n}|}{|\vec{r}_1 - (\vec{r}_1 \cdot \vec{n})\vec{n}|} \\ &= \frac{|\alpha\,(\vec{r}_1 - (\vec{r}_1 \cdot \vec{n})\vec{n})|}{|\vec{r}_1 - (\vec{r}_1 \cdot \vec{n})\vec{n}|} \\ &= \alpha \end{aligned}$$

The further restriction that $\vec{r}_2$ be a unit vector requires that

$$(\alpha\vec{r}_1 + \beta\vec{n})^2 = 1$$

$$\alpha^2 + 2\alpha\beta(\vec{r}_1 \cdot \vec{n}) + \beta^2 - 1 = 0$$

$$\beta = -\alpha(\vec{r}_1 \cdot \vec{n}) + \sqrt{\alpha^2\left((\vec{r}_1 \cdot \vec{n})^2 - 1\right)}$$

The normal vector in this method is the negative of the one expected by the equations above. As such, it is inverted before calculations get underway.

⟨*Universe Refraction Work Method Declarations*⟩+≡

```
virtual void perform( void ) {
    mathvec< double > pos = this->hit.getPosition();
    mathvec< double > dir = this->hit.getDirection();

    const color* cc = this->hit.getColor();
    double hitIOR = cc->getIndexOfRefraction();

    double ior1;
    double ior2;

    if ( this->hit.isOutside() ) {
        ior1 = this->indexOfRefraction;
        ior2 = hitIOR;
    } else {
        ior1 = hitIOR;
        ior2 = 1.0;
    }

    if ( ::fabs( ior1 - ior2 ) > 0.000001 ) {
        mathvec< double > norm = this->hit.getNormal();
        norm *= -1.0;

        double rn = ( dir ^ norm );

        if ( rn >= -0.000001 ) {
            double alpha = ior1 / ior2;
            double beta = - alpha * rn + ::sqrt(
                1.0 + alpha * alpha * ( rn * rn - 1.0 )
            );

            dir *= alpha;
            norm *= beta;
            dir += norm;
        } else {
            this->amount = 0.0;
        }
    }

    pos += dir * 0.000001;
    if ( this->univ->evaluate(
        pos, dir, ior2, this->contribution, this->height-1, this->pixel
    ) == false ) {
        this->amount = 0.0;
```

```
    }
};
```

6.4 Checking The Work

When someone wants to know how much the refracted ray should contribute, they invoke the following method. The method makes sure the work has been completed and then returns the answer.

⟨*Universe Refraction Work Method Declarations*⟩+≡
```
inline double getAmount( void ) {
    this->wait();
    return this->amount;
};
```

When someone wants to know what color the refracted ray should contribute, they invoke the following method. The method makes sure the work has been completed and then returns the answer.

⟨*Universe Refraction Work Method Declarations*⟩+≡
```
const mathvec< double >& getPixel( void ) {
    this->wait();
    return this->pixel;
};
```

6.5 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Universe Refraction Work Class Declaration*⟩≡
```
class refractionWork : public thread::work {
protected:
    ⟨Universe Refraction Work Member Variables⟩
public:
    ⟨Universe Refraction Work Method Declarations⟩
public:
    static const char* id;
};

const char* refractionWork::id = "NKVERSION: refractionWork( 2.1.2007.05.17 )";
```

Chapter 7

Universe Shadow Work

This is a helper class used by the universe class from §3 to determine the color contribution of a light on a particular spot. This helper class descends directly from the threading system's `work` class (define in §38) so that this work could potentially be performed in a subthread.

This class will store a pointer to the reflected ray's direction, a pointer to the intersection information, a pointer to the light in question, the contribution expected of this pixel, the number of levels of raytracing still available, and a pointer to the universe. It will record the amount of contribution expected from this light, the color of the diffuse contribution from this light, the color of the specular contribution from this light, and the combined diffuse and specular results.

⟨*Universe Shadow Work Member Variables*⟩≡

```
const mathvec< double >* reflectedDirection;
const intersection* hit;
const light* lamp;
double contribution;
unsigned int height;
const universe* univ;
mathvec< double > diffusePixel;
mathvec< double > specularPixel;
double amount;
mathvec< double > pixel;
```

The combined results are actually not recorded until the value is requested. This is so that if, for example, the reflected ray were not traced because the maximum number of raytracing levels had been exceeded, the proportion that would be contributed by the reflected ray is simply added into the diffuse portion of the pixel. The code in this helper class has no way of telling when such things would occur. As such, it simply waits to be told.

7.1 Constructor

The constructor does nothing
⟨*Universe Shadow Work Method Declarations*⟩≡
```
shadowWork( void )
{
}
```

7.2 Destructor

The destructor waits for the work to be completed if it has not already finished.
⟨*Universe Shadow Work Method Declarations*⟩+≡
```
virtual ~shadowWork( void ) {
    this->wait();
};
```

7.3 Preparing The Work

The initialization method here records the parameters needed and kicks off the processing.
⟨*Universe Shadow Work Method Declarations*⟩+≡
```
void set(
    const mathvec< double >* rr,
    const intersection* hh, const light* ll,
    double cn, unsigned int hg, const universe* uu
)
{
    this->reflectedDirection = rr;
    this->hit = hh;
    this->lamp = ll;
    this->contribution = cn;
    this->height = hg;
    this->univ = uu;
    this->start();
};
```

7.4 Performing The Work

The actual work of this helper class is in determining whether there is a direct line of sight from the point in question and the light of interest. If the light is on the wrong side of the surface, it cannot contribute. And, if the light is blocked, it cannot contribute. Otherwise, this method goes on to calculate the diffuse contribution from the light and the specular contribution from it. Then, it masks the diffuse reflected light by the object color.

⟨*Universe Shadow Work Method Declarations*⟩+≡
```
virtual void perform( void ) {
    ⟨Universe Shadow Work determine ray from hit to light⟩

    mathvec< double > norm = this->hit->getNormal();
    this->amount = norm ^ dir;

    double distFact = ( lampDist * this->lamp->getFallOff() );
    if ( ::fabs( distFact ) > 0.0000001 ) {
        this->amount /= distFact * distFact;
    }

    const color* cc = this->hit->getColor();

    if ( this->amount > 0.0 || cc->getTransparency() > 0.000001 ) {
        this->diffusePixel.resize( cc->getChannelCount(), 0.0 );
        ⟨Universe Shadow Work check for blockage⟩
    }

    if ( this->amount > 0.0 ) {
        ⟨Universe Shadow Work calculate diffuse from light⟩
        ⟨Universe Shadow Work calculate specular⟩
        ⟨Universe Shadow Work mask diffuse by object color⟩
    }
};
```

This portion determines the starting point and direction from the hit to the light in question.

⟨*Universe Shadow Work determine ray from hit to light*⟩≡
```
mathvec< double > pos = this->hit->getPosition();
mathvec< double > dir = this->lamp->getPosition();
dir -= pos;

double lampDist = dir.magnitude();
dir /= lampDist;

pos += dir * 0.000001;
```

To check for blockage of light, this method invokes the `checkShadow` method on the universe instance. That method is described in detail in §3.5.

⟨*Universe Shadow Work check for blockage*⟩≡
```
this->univ->checkShadow(
    pos, dir, lampDist,
    this->contribution, this->height,
    this->amount, this->diffusePixel
);
```

The diffuse contribution from the light is determined by invoking the `evaluate` method on the light.

⟨*Universe Shadow Work calculate diffuse from light*⟩≡
```
this->lamp->evaluate(
    pos, norm, dir, this->contribution, this->height, this->diffusePixel
);
```

The specular contribution initially starts with the color from the light. It is then rescaled based upon the Phong parameter of the object and the angle that light is from the reflected ray.

⟨*Universe Shadow Work calculate specular*⟩≡
```
this->specularPixel = this->diffusePixel;

double specularness = cc->getSpecularness();
if ( specularness > 0.000001 ) {
    double phong = cc->getPhong();
    double cosAngle = ( *this->reflectedDirection ^ dir );
    double factor = ::pow(cosAngle,phong) * specularness;
    this->specularPixel *= factor;
} else {
    this->specularPixel *= 0.0;
}
```

The diffuse contribution is masked by the color of the object since purely red objects do not reflect green or blue light. The specular contribution is unaffected by the object's color.

⟨*Universe Shadow Work mask diffuse by object color*⟩≡
```
mathvec< double > objPixel;
this->hit->evaluate( this->contribution, this->height, objPixel );
this->diffusePixel *= objPixel;
```

7.5 Checking The Work

When someone wants to know how much this light should contribute, they invoke the following method. The method makes sure the work has been completed and then returns the answer.

⟨*Universe Shadow Work Method Declarations*⟩+≡
```
inline double getAmount( void ) {
    this->wait();
    return this->amount;
};
```

When someone wants to know the color of the contribution from this light, they invoke the following method. It waits for the work to be completed. This method then rescales the diffuse contribution, adds in the specular contribution, and returns the result.

⟨*Universe Shadow Work Method Declarations*⟩+≡
```
const mathvec< double >& getPixel( double diffuseAmount ) {
    this->wait();
    this->pixel = this->diffusePixel;
    this->pixel *= diffuseAmount;
    this->pixel += this->specularPixel;
    return this->pixel;
};
```

7.6 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Universe Shadow Work Class Declaration*⟩≡
```
class shadowWork : public thread::work {
protected:
    ⟨Universe Shadow Work Member Variables⟩
public:
    ⟨Universe Shadow Work Method Declarations⟩
public:
    static const char* id;
};

const char* shadowWork::id = "NKVERSION: shadowWork( 2.1.2007.05.17 )";
```

Chapter 8

Light

The light class represents a point light source. The light has a position and a color. It can also specify a fall-off rate. This is the amount the distance to the lamp is scaled by before the inverse-squared effect is applied. It can also have a direction and a pair of angles. The smaller angle specifies the angular range off of the light direction that still gets full contribution of this light. The larger angle specifies the angular range off of the light direction beyond which no contribution is made from this light.

⟨*light bnf*⟩≡

```
light_parameters ::=
        light_parameters light_param
    |   /* empty */
    ;

light_param ::=
        light_pos
    |   'color' '{' color_parameters '}' ';'
    |   'falloff' real ';'
    |   'direction' real_vector ';'
    |   'angles' real ',' real ';'
    ;

light_pos ::=
        'position' real_vector ';'
    |   'center' real_vector ';'
    ;
```

Colors are described in §25. The special value `0` for the fall-off indicates that the light is the same intensity at any distance. This fall-off defaults to zero.

⟨*Light Member Variables*⟩≡

```
const scene* myScene;
```

The `light` class represents a point source of illumination within the scene. It defines the position and color of the light as well as the other parameters of the source.

⟨*Light Member Variables*⟩+≡
```
mathvec< double > position;
color chroma;
double falloff;
mathvec< double > direction;
double outerCosine;
double cosineRange;
```

8.1 Constructor

The default constructor does little beyond initializing the light and its color with the current scene pointer. Additionally, it prepares the inner and outer angle parameters to have the light default to omnidirectional.

⟨*Light Method Declarations*⟩≡
```
light( const scene* ss );
```

⟨*Light Methods*⟩≡
```
light::light( const scene* ss )
    : myScene( ss ), chroma( ss ),
        falloff( 0.0 ), outerCosine(-2.0), cosineRange(-1.0)
{
}
```

8.2 Setting The Number of Dimensions

When the number of spatial dimensions for the universe are set, this information is passed along to all of the constituent lights. The light resizes the position and direction vectors as well as the color accordingly. The direction of the light is normalized.

⟨*Light Method Declarations*⟩+≡
```
void setDimensions( unsigned int dd );
```

⟨*Light Methods*⟩+≡
```
void
light::setDimensions( unsigned int dd )
{
    this->position.resize( dd, 0.0 );
    this->chroma.setDimensions( dd );
    this->direction.resize( dd, 0.0 );

    double dmag = this->direction.magnitude();
    if ( dmag >= 0.000001 ) {
        this->direction /= dmag;
    }
}
```

8.3 Retreiving The Light Position

To query the light's position, one can invoke the following method. This is used by one of the universe's helper classes to determine the direction from the point of intersection to the light.

⟨*Light Method Declarations*⟩+≡
```
inline const mathvec< double >& getPosition( void ) const {
    return this->position;
};
```

8.4 Retreiving The Fall Off Factor

To query the light's fall-off factor, one can invoke the following method.

⟨*Light Method Declarations*⟩+≡
```
inline double getFallOff( void ) const {
    return this->falloff;
};
```

8.5 Querying The Light Color

The following method allows one to query the color of the light to a particular position, with a given normal, and from a given direction.

⟨*Light Method Declarations*⟩+≡
```
void evaluate(
    const mathvec< double >& position,
    const mathvec< double >& normal,
    const mathvec< double >& direction,
    double contribution, unsigned int height,
    mathvec< double >& pixel
) const;
```

The method first invokes the `evaluate` method on the color of the light. Then, if angles were specified for this light, it rescales the contribution based on the angular difference.

⟨*Light Methods*⟩+≡
```
void
light::evaluate(
    const mathvec< double >& pos,
    const mathvec< double >& norm,
    const mathvec< double >& dir,
    double contribution, unsigned int height,
    mathvec< double >& pixel
) const {
    this->chroma.evaluate(
        pos, norm, dir, contribution, height, pixel
    );

    if ( this->outerCosine > -1.5 ) {
        double cc = - ( dir ^ this->direction );
        double diff = cc - this->outerCosine;
        if ( diff < 0.000001 ) {
            pixel *= 0.0;
        } else if ( diff < this->cosineRange ) {
            pixel *= diff / this->cosineRange;
        }
    }
}
```

8.6 Loading A Light

The following method allows one to read in a light from a stream.

⟨*Light Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, light& cc );
```

There are several keywords recognized in the light. One can specify the location of the light with the `position` or `center` vector. One specifies the light's direction with the `direction` vector. One specifies the inner and outer angles (in degrees) with the two values of the `angles` tag. One specifies the color of the light by specifying an instance of the color class (defined in §25) using the `color` tag.

⟨*Light Methods*⟩+≡
```
std::istream&
operator >> ( std::istream& stream, light& ll )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "position" || tok == "center" ) {
            stream >> ll.position >> semicolon;
        } else if ( tok == "direction" ) {
            stream >> ll.direction >> semicolon;
        } else if ( tok == "falloff" ) {
            stream >> ll.falloff >> semicolon;
        } else if ( tok == "angles" ) {
            ⟨Light read angles⟩
        } else if ( tok == "color" ) {
            stream >> ll.chroma >> semicolon;
        } else {
            in.ignore( "LIGHT", tok.c_str() );
        }
    }

    return stream;
}
```

To calculate the angle parameters, the angles are read in. Then, their cosines are determined and sorted. Then, the cosine of the outer angle and the difference in cosines are stored in the light.

⟨*Light read angles*⟩≡
```
double inner;
char comma;
double outer;
stream >> inner >> comma >> outer >> semicolon;

inner = ::cos( inner * M_PI / 180.0 );
outer = ::cos( outer * M_PI / 180.0 );

if ( inner < outer ) {
    double tmp = inner;
    inner = outer;
    outer = tmp;
}

ll.outerCosine = outer;
ll.cosineRange = inner - outer;
```

8.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Light Class Declaration*⟩≡
```
class light {
protected:
    ⟨Light Member Variables⟩
public:
    ⟨Light Method Declarations⟩
public:
    static const char* id;
};
```

8.8 Source Files

The following sections assemble the source files for the `light` class from the chunks above.

8.8.1 `light.h`

The header file for the light class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header files for the `color` class and then incorporates the class defined in §8.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*light.h*⟩≡
```
#ifndef _NKLEIN_LIGHT_H_
#define _NKLEIN_LIGHT_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "color.h"

    ⟨Light Class Declaration⟩

#endif /*_NKLEIN_LIGHT_H_*/
```

8.8.2 `light.cc`

Herein, the implementations of the methods of the `light` class are included. The implementation requires the header for the `string` class and the math library.

⟨*light.cc*⟩≡
```
#include <string>
#include <math.h>
#include "input.h"
#include "light.h"

const char* light::id = "NKVERSION: light( 2.1.2007.05.17 )";

⟨Light Methods⟩
```

Chapter 9

Object

This class serves as the base class for all geometric object. It keeps track of the attributes common to all such objects.

⟨*object bnf*⟩≡

```
object_parameters ::=
        'scale' real_vector ';'
    |   'orientation' real_matrix ';'
    |   'center' real_vector ';'
    |   'color' '{' color_parameters '}' ';'
    ;
```

There are a variety of objects which inherit from this base class.

⟨*object bnf*⟩+≡

```
object_type_and_params ::=
        'halfspace'    '{' halfspace_params '}' ';'
    |   'cylinder'     '{' cylinder_params '}' ';'
    |   'quadratic'    '{' quadratic_params '}' ';'
    |   'convex'       '{' convex_params '}' ';'
    |   'coxeter'      '{' coxeter_params '}' ';'
    |   'complement'   '{' complement_params '}' ';'
    |   'union'        '{' csg_union_params '}' ';'
    |   'intersection' '{' csg_intersection_params '}' ';'
    |   'extrusion'    '{' extrusion_params '}' ';'
    ;
```

The object keeps track of the scene to which it belongs.

⟨*Object Member Variables*⟩≡

```
const scene* myScene;
```

The object class is a base class common to all geometric objects. It defines some properties shared by all objects: scale, orientation, center, and color.

⟨*Object Member Variables*⟩+≡
```
double defaultScale;
mathvec< double > scale;
basevec< mathvec< double > > orientation;
mathvec< double > center;
color chroma;
bool gotColor;
```

Many objects will also need to forge transparent hits a great distance away. The following static member of the object class allows that to happen easily.

⟨*Object Member Variables*⟩+≡
```
static color fakeColor;
```

⟨*Object Static Member Initializations*⟩≡
```
color object::fakeColor(
    "channels [3]< 0 , 0 , 0 >;\n"
    "reflectiveness 0.0;\n"
    "specularness 0.0;\n"
    "transparency 1.0;\n"
);
```

9.1 Contructor

This contructor initializes the scene information.

⟨*Object Method Declarations*⟩≡
```
object( const scene* ss );
```

⟨*Object Methods*⟩≡
```
object::object( const scene* ss )
    : myScene( ss ), defaultScale( 1.0 ), chroma( ss ), gotColor(false)
{
};
```

9.2 Destructor

This destructor is simply a place-holder to ensure that the base classes are destructed properly.

⟨*Object Method Declarations*⟩+≡
```
virtual ~object( void );
```

⟨*Object Methods*⟩+≡
```
object::~object( void ) {
};
```

9.3 Setting The Number of Dimensions

Even though this method is virtual, there is an implementation that derived classes should invoke. It resizes the scaling vector, the orientation matrix, the center vector, and the color.

⟨*Object Method Declarations*⟩+≡
```
virtual void setDimensions( unsigned int dd );
```

⟨*Object Methods*⟩+≡
```
void
object::setDimensions( unsigned int dd )
{
    this->scale.resize( dd, this->defaultScale );
    ::orthogonalize( this->orientation, dd );
    this->center.resize( dd, 0.0 );
    this->chroma.setDimensions( dd );
}
```

9.4 Retreiving The Common Attributes

These methods allow one to query the attributes common to all objects.

⟨*Object Method Declarations*⟩+≡
```
inline const mathvec< double >& getScale( void ) const {
    return this->scale;
};
inline const basevec< mathvec< double > >& getOrientation( void ) const {
    return this->orientation;
};
inline const mathvec< double >& getCenter( void ) const {
    return this->center;
};
inline const color& getColor( void ) const {
    return this->chroma;
};
inline color* getColorPtr( void ) {
    return &this->chroma;
};
```

9.5 Setting The Common Attributes

These methods allow one to set some of the attributes common to all objects.

⟨*Object Method Declarations*⟩+≡

```
inline void setScale( const mathvec< double >& ss ) {
    this->scale = ss;
};
inline void setOrientation( const basevec< mathvec< double > >& oo ) {
    this->orientation = oo;
};
inline void setCenter( const mathvec< double >& cc ) {
    this->center = cc;
};
```

9.6 Intersection A Ray And An Object

This first intersection method retrieves only the closest intersection to the ray's origin. The ray is specified by its direction and start. This is a pure-virtual method which must be overridden by the derived class.

⟨*Object Method Declarations*⟩+≡

```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const = 0;
```

This second intersection method retrieves all intersections between this object and the ray. The ray is specified by its direction and start. The array of hits is sorted in order of increasing distance from the start of the ray. This is a pure-virtual method which be overridden by the derived class.

⟨*Object Method Declarations*⟩+≡

```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const = 0;
```

9.7 Checking If A Point Is Inside An Object

The following methods are used to determine if a point is inside the object. The first method `inside()` is used when the point is not yet in object coordinates. The other method `_inside()` is used when the point has already be oriented into the object's coordinate system.

⟨*Object Method Declarations*⟩+≡
```
virtual bool inside( const mathvec< double >& start ) const = 0;
virtual bool _inside( const mathvec< double >& start ) const = 0;
```

9.8 Querying The Object Color

The following method allows one to query the color of the object at a particular position, with a given normal, and from a given direction.

⟨*Object Method Declarations*⟩+≡
```
void evaluate(
    const mathvec< double >& position,
    const mathvec< double >& normal,
    const mathvec< double >& direction,
    double contribution, unsigned int height,
    mathvec< double >& pixel
) const;
```

This method simply calls the appropriate method on the color.

⟨*Object Methods*⟩+≡
```
void
object::evaluate(
    const mathvec< double >& position,
    const mathvec< double >& normal,
    const mathvec< double >& direction,
    double contribution, unsigned int height,
    mathvec< double >& pixel
) const {
    this->chroma.evaluate(
        position, normal, direction, contribution, height, pixel
    );
}
```

9.9 Forging A Hit

There are times when an object needs to forge a hit an infinite distance away. For example, when the entire ray is below the plane of a halfspace, the halfspace forges a hit an infinite distance away so that this object is not ignored when taken as part of an intersection of two objects. This method provides a common way for objects to forge this hit information.

⟨*Object Protected Method Declarations*⟩≡

```
void forgeHit(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const;
```

⟨*Object Methods*⟩+≡

```
void
object::forgeHit(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const
{
    mathvec< double > pos = direction;
    pos *= MAXFLOAT;
    pos += start;

    mathvec< double > norm = direction;
    norm *= -1.0;

    hit.setPosition( pos );
    hit.setNormal( norm );
    hit.setDirection( direction );

    hit.setObjectPosition( pos );
    hit.setObjectNormal( norm );
    hit.setObjectDirection( direction );

    hit.setInside();
    hit.setDistance( MAXFLOAT );
    hit.setObjectColor( &object::fakeColor );
}
```

9.10 Reading Parts Common To All Objects

The following method allows one to read in an object from a stream.

⟨*Object Method Declarations*⟩+≡

```
friend std::istream& operator >> ( std::istream& in, object& cc );
```

⟨*Object Methods*⟩+≡

```
std::istream&
operator >> ( std::istream& stream, object& ll )
{
    input in( stream );

    ll.gotColor = false;

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "scale" ) {
            stream >> ll.scale >> semicolon;
        } else if ( tok == "orientation" ) {
            stream >> ll.orientation >> semicolon;
        } else if ( tok == "antiorientation" ) {
            stream >> ll.orientation >> semicolon;
            ::orthogonalize( ll.orientation, ll.orientation[0].size() );
            ::transpose( ll.orientation );
        } else if ( tok == "center" ) {
            stream >> ll.center >> semicolon;
        } else if ( tok == "color" ) {
            stream >> ll.chroma >> semicolon;
            ll.gotColor = true;
        } else {
            in.ignore( "OBJECT", tok.c_str() );
        }
    }

    return stream;
}
```

9.11 Loading An Object

The following method allows one to read in an object from a stream.

⟨*Object Method Declarations*⟩+≡

```
static object* read( std::istream& in, const scene* ss );
```

This method pulls code in from all of the derived objects through the ⟨*Derived Object Read Cases*⟩ chunk which filled in more in each particular object's source.

⟨*Object Methods*⟩+≡

```
object*
object::read( std::istream& stream, const scene* ss )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return 0;
        }

        std::string objectType( token );
        char semicolon;

        ⟨Derived Object Read Cases⟩ {
            in.ignore( "OBJECT READER", objectType.c_str() );
        }
    }

    return 0;
}
```

9.12 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Object Class Declaration*⟩≡
```
class object {
protected:
    ⟨Object Member Variables⟩
protected:
    ⟨Object Protected Method Declarations⟩
public:
    ⟨Object Method Declarations⟩
public:
    static const char* id;
};
```

9.13 Source Files

The following sections assemble the source files for the `object` class from the chunks above.

9.13.1 `object.h`

The header file the `object` class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header file for the `intersection` class and then incorporates the class defined in 9.12 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*object.h*⟩≡
```
#ifndef _NKLEIN_OBJECT_H_
#define _NKLEIN_OBJECT_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "intersection.h"

    ⟨Object Class Declaration⟩

#endif /*_NKLEIN_OBJECT_H_*/
```

9.13.2 object.cc

Herein, the implementations of the methods of the `object` class are included. Also, the header files for the derived objects are included via the ⟨*Derived Object Includes*⟩ chunk that is appended to in each derived class's source file. Those includes allow the ⟨*Derived Object Read Cases*⟩ chunks used above to function properly here.

⟨*object.cc*⟩≡

```
#include <string>
#include <sstream>
#include <math.h>
#ifndef MAXFLOAT
#include <values.h>
#endif
#include "input.h"
#include "object.h"

⟨Derived Object Includes⟩

const char* object::id = "NKVERSION: object( 2.1.2007.05.17 )";

⟨Object Static Member Initializations⟩
⟨Object Methods⟩
```

Chapter 10

Halfspace

The halfspace (before it is scaled, oriented, and positioned) is all points with an x-coordinate less than or equal to zero. The halfspace itself requires no parameters beyond those defined in the object base class.

⟨*halfspace bnf*⟩≡

```
halfspace_params ::=
        'base' '{' object_parameters '}' ';'
    ;
```

The following snippets are used so that the object's factory method can create instance of this class.

⟨*Derived Object Includes*⟩≡

```
#include "halfspace.h"
```

⟨*Derived Object Read Cases*⟩≡

```
if ( objectType == "halfspace" ) {
    halfspace* obj = new halfspace( ss );
    stream >> *obj >> semicolon;
    return obj;
} else
```

The halfspace requires no member variables. Its normal will be the x-axis.

10.1 Constructor

The default constructor does nothing except initialize the scene pointer.

⟨*Halfspace Method Declarations*⟩≡

```
halfspace( const scene* ss );
```

⟨*Halfspace Methods*⟩≡
```
halfspace::halfspace( const scene* ss ) : object( ss )
{
}
```

This constructor initializes from a given vector setting both the center and normal.

⟨*Halfspace Method Declarations*⟩+≡
```
halfspace( const mathvec< double >& cc );
```

⟨*Halfspace Methods*⟩+≡
```
halfspace::halfspace( const mathvec<double>& cc ) : object( 0 )
{
    this->center = cc;

    this->orientation.resize( 1 );
    this->orientation[ 0 ] = cc;
    this->orientation[ 0 ].normalize();
}
```

This constructor initializes from a given center and normal.

⟨*Halfspace Method Declarations*⟩+≡
```
halfspace( const mathvec< double >& cc, const mathvec< double >& nn );
```

⟨*Halfspace Methods*⟩+≡
```
halfspace::halfspace(
    const mathvec<double>& cc, const mathvec<double>& nn
) : object( 0 )
{
    this->center = cc;

    this->orientation.resize( 1 );
    this->orientation[ 0 ] = nn;
    this->orientation[ 0 ].normalize();
}
```

10.2 Destructor

This destructor has nothing to do.

⟨*Halfspace Method Declarations*⟩+≡
```
virtual ~halfspace( void );
```

⟨*Halfspace Methods*⟩+≡
```
halfspace::~halfspace( void ) {
};
```

10.3 Setting The Number of Dimensions

This method simply invokes the corresponding method on the base class.

⟨*Halfspace Method Declarations*⟩+≡
```
virtual void setDimensions( unsigned int dd );
```

⟨*Halfspace Methods*⟩+≡
```
void
halfspace::setDimensions( unsigned int dd )
{
    this->object::setDimensions( dd );
}
```

10.4 Intersection A Ray And A Halfspace

This first intersection method retrieves only the closest intersection to the ray's origin. The ray is specified by its direction and start.

⟨*Halfspace Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const;
```

⟨*Halfspace Methods*⟩+≡
```
bool
halfspace::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const {
    ⟨Halfspace intersect common parts⟩

    if ( tt > 0.000001 ) {
        ⟨Halfspace intersect fill in hit⟩
    } else {
        return false;
    }

    return true;
}
```

⟨*Halfspace intersect common parts*⟩≡
```
unsigned int len = this->center.size();

mathvec< double > rd = direction;
rd.antirotate( this->orientation );
rd /= this->scale;
rd.normalize();

mathvec< double > rp = start - this->center;
rp.antirotate( this->orientation );
rp /= this->scale;

mathvec< double > nn( 0.0, len );
if ( rp[ 0 ] > 0.0 ) {
    nn[ 0 ] = 1.0;
} else {
    nn[ 0 ] = -1.0;
}

double tt;
double dd = rd[ 0 ];

if ( ::fabs( dd ) > 0.000001 ) {
    tt = - rp[ 0 ] / dd;
} else {
    tt = - 1.0;
}
```

⟨*Halfspace intersect fill in hit*⟩≡
```
mathvec< double > spot = rp + ( rd * tt );

if ( rp[ 0 ] > 0.0 ) {
    hit.setOutside();
} else {
    hit.setInside();
}

hit.setObjectPosition( spot );
hit.setObjectNormal( nn );
hit.setObjectDirection( rd.normalize() );

hit.setPosition( spot );
hit.setNormal( nn );
hit.setDirection( rd );

hit.reorient( this->scale, this->orientation, this->center );
hit.setDistance( ( hit.getPosition() - start ).magnitude() );

hit.setObjectColor( &this->chroma );
```

This second intersection method retrieves all intersections between this half-space and the ray. The ray is specified by its direction and start. The array of hits is sorted in order of increasing distance from the start of the ray.

⟨*Halfspace Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const;
```

⟨*Halfspace Methods*⟩+≡
```
bool
halfspace::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const {
    ⟨Halfspace intersect common parts⟩

    hits.clear();

    if ( tt > 0.000001 ) {
        intersection* hh = new intersection();
        intersection& hit( *hh );
        ⟨Halfspace intersect fill in hit⟩
        hits.insert( hits.end(), hh );
    } else if ( rp[ 0 ] <= 0.0 ) {
        intersection* hh = new intersection();
        this->forgeHit( direction, start, *hh );
        hits.insert( hits.end(), hh );
    }

    return hits.size() > 0;
}
```

We're also going to provide a way just to get the distance that a point is from the plane.

⟨*Halfspace Method Declarations*⟩+≡
```
double distance(
    const mathvec< double >& start
) const;
```

⟨*Halfspace Methods*⟩+≡
```
double
halfspace::distance(
    const mathvec< double >& start
) const {
    mathvec< double > rp = start - this->center;
    rp.antirotate( this->orientation );
    rp /= this->scale;
    return rp[ 0 ];
}
```

10.5 Checking If A Point Is Inside A Halfspace

⟨*Halfspace Method Declarations*⟩+≡
```
virtual bool inside( const mathvec< double >& start ) const;
virtual bool _inside( const mathvec< double >& rp ) const;
```

⟨*Halfspace Methods*⟩+≡
```
bool
halfspace::inside( const mathvec< double >& start ) const
{
    mathvec< double > rp = start - this->center;
    rp.antirotate( this->orientation );
    rp /= this->scale;

    return this->_inside( rp );
}

bool
halfspace::_inside( const mathvec< double >& rp ) const
{
    return rp[ 0 ] <= 0.0;
}
```

10.6 Loading A Halfspace

The following method allows one to read in an halfspace from a stream.

⟨*Halfspace Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, halfspace& cc );
```

⟨*Halfspace Methods*⟩+≡
```
std::istream&
operator >> ( std::istream& stream, halfspace& hh )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "base" ) {
            stream >> (object&)hh >> semicolon;
        } else {
            in.ignore( "HALFSPACE", tok.c_str() );
        }
    }

    return stream;
}
```

10.7 The Class Definition

This class inherits directly from the object base class. This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Halfspace Class Declaration*⟩≡
```
class halfspace : public object {
public:
    ⟨Halfspace Method Declarations⟩
public:
    static const char* id;
};
```

10.8 Source Files

The following sections assemble the source files for the `halfspace` class from the chunks above.

10.8.1 `halfspace.h`

The header file the `halfspace` class includes the `iostream` and `string` headers from the standard C++ libraries and the header file for the `object` class and then incorporates the class defined in 10.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*halfspace.h*⟩≡
```
#ifndef _NKLEIN_HALFSPACE_H_
#define _NKLEIN_HALFSPACE_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "object.h"

    ⟨Halfspace Class Declaration⟩

#endif /*_NKLEIN_HALFSPACE_H_*/
```

10.8.2 `halfspace.cc`

Herein, the implementations of the methods of the `halfspace` class are included as well as the the declarations and implementations of the derived classes.

⟨*halfspace.cc*⟩≡
```
#include <string>
#include <sstream>
#include <math.h>
#include "input.h"
#include "halfspace.h"

const char* halfspace::id = "NKVERSION: halfspace( 2.1.2007.05.17 )";

⟨Halfspace Methods⟩
```

Chapter 11

Cylinder

The cylinder has a certain number of round dimensions (r) in an n-dimensional scene (where $0 \leq r \leq n$). The cylinder (before it is scale, oriented, and positioned) is all points $\vec{x}$ such that $\sum_{i=1} rx_i^2 \leq 1$ and $\max_{i=r+1}^{n} x_i^2 \leq 1$. The sphere is the special case where $r = n$ and the cube is the special case where $r = 0$ (or $r = 1$).

The cylinder uses a parameter specifying r the number of round dimensions and can specify the parameters for the object base class.

⟨*cylinder bnf*⟩≡

```
cylinder_params ::=
        'roundDimensions' integer ';'
    |   'base' '{' object_parameters '}' ';'
    ;
```

The following snippets are used so that the object's factory method can create instance of this class.

⟨*Derived Object Includes*⟩+≡

```
#include "cylinder.h"
```

⟨*Derived Object Read Cases*⟩+≡

```
if ( objectType == "cylinder" ) {
    cylinder* obj = new cylinder( ss );
    stream >> *obj >> semicolon;
    return obj;
} else
```

The `cylinder` class is a subclass of the `object` class from 9. It implements spheres, cubes, and all of the cylinders in between. In addition to the properties of all `objects`, the cylinder tracks the number of round dimensions. If the number of round dimensions is r and the number of dimensions overall is n, then the cylinder obeys the following:

$$x_1^2 + x_2^2 + \cdots + x_r^2 \leq 1$$

$$|x_{r+1}| \leq 1$$

$$|x_{r+2}| \leq 1$$

$$\cdots$$

$$|x_n| \leq 1$$

⟨*Cylinder Member Variables*⟩≡
```
unsigned int roundDimensions;
```

In addition, the class keeps a mask where the first round dimensions are ones and the remaining entries are zero.

⟨*Cylinder Member Variables*⟩+≡
```
mathvec< double > sphereMask;
```

11.1 Constructor

The default constructor does nothing.

⟨*Cylinder Method Declarations*⟩≡
```
cylinder( const scene* ss );
```

⟨*Cylinder Methods*⟩≡
```
cylinder::cylinder( const scene* ss ) : object( ss )
{
}
```

There is a constructor which takes a radius and makes this a sphere.

⟨*Cylinder Method Declarations*⟩+≡
```
cylinder( double radius, const scene* ss );
```

⟨*Cylinder Methods*⟩+≡
```
cylinder::cylinder( double radius, const scene* ss ) : object( ss )
{
    this->defaultScale = radius;
    this->roundDimensions = ~0U;
}
```

11.2 Destructor

This destructor has nothing to do.

⟨*Cylinder Method Declarations*⟩+≡
```
virtual ~cylinder( void );
```

⟨*Cylinder Methods*⟩+≡
```
cylinder::~cylinder( void ) {
};
```

11.3 Setting The Number of Dimensions

Even though this method is virtual, there is an implementation that derived classes should invoke.

⟨*Cylinder Method Declarations*⟩+≡
```
virtual void setDimensions( unsigned int dd );
```

⟨*Cylinder Methods*⟩+≡
```
void
cylinder::setDimensions( unsigned int dd )
{
    this->sphereMask.resize( dd );

    if ( this->roundDimensions == 1 ) {
        this->roundDimensions = 0;
    }

    if ( this->roundDimensions > dd ) {
        this->roundDimensions = dd;
    }

    for ( unsigned int ii=0; ii < this->roundDimensions; ++ii ) {
        this->sphereMask[ ii ] = 1.0;
    }

    for ( unsigned int ii=this->roundDimensions; ii < dd; ++ii ) {
        this->sphereMask[ ii ] = 0.0;
    }

    this->object::setDimensions( dd );
}
```

11.4 Intersection A Ray And A Cylinder

This first intersection method retrieves only the closest intersection to the ray's origin. The ray is specified by its direction and start.

⟨*Cylinder Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const;
```

⟨*Cylinder Methods*⟩+≡
```
bool
cylinder::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const {
    ⟨Cylinder intersect common parts⟩

    if ( !hitSphere || !hitCube ) {
        if ( this->_inside( rp ) ) {
            this->forgeHit( direction, start, hit );
            return true;
        } else {
            return false;
        }
    }

    if ( t1 >= 0.000001 ) {
        mathvec< double >& nn( n1 );
        double tt = t1;
        double normalFactor = 1.0;
        ⟨Cylinder intersect fill in hit⟩
        hit.setOutside();
    } else if ( t2 >= 0.000001 ) {
        mathvec< double >& nn( n2 );
        double tt = t2;
        double normalFactor = -1.0;
        ⟨Cylinder intersect fill in hit⟩
        hit.setInside();
    } else {
        return false;
    }

    return true;
}
```

⟨*Cylinder intersect common parts*⟩≡
```
mathvec< double > rd = direction;
rd.antirotate( this->orientation );
rd /= this->scale;
rd.normalize();

mathvec< double > rp = start - this->center;
rp.antirotate( this->orientation );
rp /= this->scale;

double t1 = -MAXFLOAT;
double t2 = MAXFLOAT;

mathvec< double > n1;
mathvec< double > n2;

bool hitSphere = this->sphereIntersect( rd, rp, &t1, &t2 );
bool hitCube = hitSphere && this->cubeIntersect( rd, rp, &t1, &t2, n1, n2 );
```

⟨*Cylinder intersect fill in hit*⟩≡
```
mathvec< double > spot = rp + ( rd * tt );

if ( nn.size() == 0 ) {
    nn = spot * this->sphereMask * normalFactor;
    nn.normalize();
}

hit.setObjectPosition( spot );
hit.setObjectNormal( nn );
hit.setObjectDirection( rd );

hit.setPosition( spot );
hit.setNormal( nn );
hit.setDirection( rd );

hit.reorient( this->scale, this->orientation, this->center );
hit.setDistance( ( hit.getPosition() - start ).magnitude() );

hit.setObjectColor( &this->chroma );
```

This second intersection method retrieves all intersections between this cylinder and the ray. The ray is specified by its direction and start. The array of hits is sorted in order of increasing distance from the start of the ray.

⟨*Cylinder Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const;
```

⟨*Cylinder Methods*⟩+≡

```
bool
cylinder::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const {
    ⟨Cylinder intersect common parts⟩

    hits.clear();

    if ( !hitSphere || !hitCube ) {
        if ( this->_inside( rp ) ) {
            intersection* hh = new intersection();
            this->forgeHit( direction, start, *hh );
            hits.insert( hits.end(), hh );
            return true;
        } else {
            return false;
        }
    }

    if ( t1 >= 0.000001 ) {
        intersection* h1 = new intersection();
        intersection& hit( *h1 );
        mathvec< double >& nn( n1 );
        double tt = t1;
        double normalFactor = 1.0;
        ⟨Cylinder intersect fill in hit⟩
        hit.setOutside();
        hits.insert( hits.end(), h1 );
    }

    if ( t2 >= 0.000001 ) {
        intersection* h2 = new intersection();
        intersection& hit( *h2 );
        mathvec< double >& nn( n2 );
        double tt = t2;
        double normalFactor = -1.0;
        ⟨Cylinder intersect fill in hit⟩
        hit.setInside();
        hits.insert( hits.end(), h2 );
    }

    return hits.size() > 0;
}
```

11.4.1 Intersecting With The Spherical Portions

⟨*Cylinder Protected Method Declarations*⟩≡

```
bool sphereIntersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    double* t1, double* t2
) const;
```

⟨*Cylinder Methods*⟩+≡

```
bool
cylinder::sphereIntersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    double* t1, double* t2
) const {
    if ( this->roundDimensions == 0 ) {
        return true;
    }

    mathvec< double > rds = direction * this->sphereMask;
    mathvec< double > rps = start * this->sphereMask;

    double aa = rds ^ rds;
    double cc = ( rps ^ rps ) - 1.0;

    if ( aa < 0.000001 ) {
        return ( cc < 0.000001 );
    }

    double bb = 2.0 * ( rds ^ rps );
    double discriminant = bb * bb - 4.0 * aa * cc;


    if ( discriminant < 0.000001 ) {
        return false;
    }

    double root = ::sqrt( discriminant );

    if ( aa > 0 ) {
        *t1 = ( -bb - root ) / ( 2.0 * aa );
        *t2 = ( -bb + root ) / ( 2.0 * aa );
    } else {
        *t1 = ( -bb + root ) / ( 2.0 * aa );
        *t2 = ( -bb - root ) / ( 2.0 * aa );
    }

    if ( *t2 < 0.000001 ) {
        return false;
    }

    return true;
}
```

11.4.2 Intersecting With The Cubical Portions

⟨*Cylinder Protected Method Declarations*⟩+≡

```
bool cubeIntersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    double* t1, double* t2,
    mathvec< double >& n1, mathvec< double >& n2
) const;
```

⟨*Cylinder Methods*⟩+≡
```
bool
cylinder::cubeIntersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    double* t1, double* t2,
    mathvec< double >& n1, mathvec< double >& n2
) const {
    unsigned int dims = this->center.size();

    if ( this->roundDimensions == dims ) {
        return true;
    }

    unsigned int n1dim = dims;
    double n1mag = 1.0;
    unsigned int n2dim = dims;
    double n2mag = 1.0;

    for ( unsigned int ii=this->roundDimensions; ii < dims; ++ii ) {
        double t1c;
        double t2c;
        double nn = direction[ ii ];
        double ss = start[ ii ];

        if ( ::fabs( nn ) < 0.000001 ) {
            if ( ss < -1.0 || ss > 1.0 ) {
                return false;
            } else {
                continue;
            }
        }

        t1c = ( -ss + 1.0 ) / nn;
        t2c = ( -ss - 1.0 ) / nn;

        if ( t1c < 0.000001 && t2c < 0.000001 ) {
            return false;
        }

        double mag = 1.0;

        if ( t1c > t2c ) {
            double tmp = t1c;
            t1c = t2c;
            t2c = tmp;
```

```
            mag = -1.0;
        }

        if ( t2c < *t1 || t1c > *t2  ) {
            return false;
        }

        if ( t1c > *t1 ) {
            *t1 = t1c;
            n1dim = ii;
            n1mag = mag;
        }
        if ( t2c < *t2 ) {
            *t2 = t2c;
            n2dim = ii;
            n2mag = mag;
        }
    }

    if ( n1dim < dims ) {
        n1.resize( dims, 0.0 );
        n1[ n1dim ] = n1mag;
    }
    if ( n2dim < dims ) {
        n2.resize( dims, 0.0 );
        n2[ n2dim ] = n2mag;
    }

    return true;
}
```

11.5 Checking If A Point Is Inside A Cylinder

⟨*Cylinder Method Declarations*⟩+≡

```
virtual bool inside( const mathvec< double >& start ) const;
virtual bool _inside( const mathvec< double >& rp ) const;
```

⟨*Cylinder Methods*⟩+≡
```
bool
cylinder::inside( const mathvec< double >& start ) const
{
    mathvec< double > rp = start - this->center;
    rp.antirotate( this->orientation );
    rp /= this->scale;

    return this->_inside( rp );
}

bool
cylinder::_inside( const mathvec< double >& rp ) const
{
    unsigned int dims = this->center.size();
    for ( unsigned int ii=this->roundDimensions; ii < dims; ++ii ) {
        if ( ::fabs( rp[ ii ] ) > 1.0 ) {
            return false;
        }
    }

    double sum = 0.0;
    for ( unsigned int ii=0; ii < this->roundDimensions; ++ii ) {
        sum += rp[ ii ] * rp[ ii ];
    }

    return sum <= 1.0;
}
```

11.6 Loading A Cylinder

The following method allows one to read in an cylinder from a stream.

⟨*Cylinder Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, cylinder& cc );
```

⟨*Cylinder Methods*⟩+≡

```
std::istream&
operator >> ( std::istream& stream, cylinder& cc )
{
    input in( stream );

    cc.roundDimensions = 0;

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "roundDimensions" ) {
            stream >> cc.roundDimensions >> semicolon;
        } else if ( tok == "base" ) {
            stream >> (object&)cc >> semicolon;
        } else {
            in.ignore( "CYLINDER", tok.c_str() );
        }
    }

    return stream;
}
```

11.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Cylinder Class Declaration*⟩≡

```
class cylinder : public object {
protected:
    ⟨Cylinder Member Variables⟩
    ⟨Cylinder Protected Method Declarations⟩
public:
    ⟨Cylinder Method Declarations⟩
public:
    static const char* id;
};
```

11.8 Source Files

The following sections assemble the source files for the `cylinder` class from the chunks above.

11.8.1 `cylinder.h`

The header file the `cylinder` class includes the `iostream` and `string` headers from the standard C++ libraries and the header file for the `mathvec<>` class and then incorporates the class defined in 11.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*cylinder.h*⟩≡

```
#ifndef _NKLEIN_CYLINDER_H_
#define _NKLEIN_CYLINDER_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "object.h"

    ⟨Cylinder Class Declaration⟩

#endif /*_NKLEIN_CYLINDER_H_*/
```

11.8.2 cylinder.cc

Herein, the implementations of the methods of the cylinder class are included as well as the the declarations and implementations of the derived classes.

⟨*cylinder.cc*⟩≡

```
#include <string>
#include <sstream>
#include <math.h>
#ifndef MAXFLOAT
#include <values.h>
#endif
#include "input.h"
#include "cylinder.h"

const char* cylinder::id = "NKVERSION: cylinder( 2.1.2007.05.17 )";

⟨Cylinder Methods⟩
```

Chapter 12

Quadratic

The quadratic surface is specified using a matrix M, a vector $\vec{V}$, and a scalar S. The quadratic surface (before it is scaled, oriented, and positined) is all points $\vec{x}$ satisfying:

$$\vec{x}^T M \vec{x} + \vec{V}^T \vec{x} + S \leq 0$$

The quadratic surface must specify the matrix M of squared terms, the vector V of linear terms, and the scalar S. It can also specify the parameters for the object base class.

⟨*quadratic bnf*⟩≡

```
quadratic_params ::=
        'squares' real_matrix ';'
    |   'linears' real_vector ';'
    |   'scalar' real ';'
    |   'base' '{' object_parameters '}' ';'
    ;
```

The following snippets are used so that the object's factory method can create instance of this class.

⟨*Derived Object Includes*⟩+≡

```
#include "quadratic.h"
```

⟨*Derived Object Read Cases*⟩+≡

```
if ( objectType == "quadratic" ) {
    quadratic* obj = new quadratic( ss );
    stream >> *obj >> semicolon;
    return obj;
} else
```

⟨*Quadratic Member Variables*⟩≡
```
basevec< mathvec< double > > squares;
mathvec< double > linears;
double scalar;
```

12.1 Constructor

The default constructor does nothing.

⟨*Quadratic Method Declarations*⟩≡
```
quadratic( const scene* ss );
```

⟨*Quadratic Methods*⟩≡
```
quadratic::quadratic( const scene* ss ) : object( ss )
{
}
```

12.2 Destructor

This destructor has nothing to do.

⟨*Quadratic Method Declarations*⟩+≡
```
virtual ~quadratic( void );
```

⟨*Quadratic Methods*⟩+≡
```
quadratic::~quadratic( void ) {
}
```

12.3 Setting The Number of Dimensions

Even though this method is virtual, there is an implementation that derived classes should invoke.

⟨*Quadratic Method Declarations*⟩+≡
```
virtual void setDimensions( unsigned int dd );
```

⟨*Quadratic Methods*⟩+≡
```
void
quadratic::setDimensions( unsigned int dd )
{
    this->squares.resize( dd );
    for ( unsigned int ii=0; ii < dd; ++ii ) {
        this->squares[ ii ].resize( dd, 0.0 );
    }
    this->linears.resize( dd );

    this->object::setDimensions( dd );
}
```

12.4 Intersection A Ray And A Quadratic

This first intersection method retrieves only the closest intersection to the ray's origin. The ray is specified by its direction and start.

⟨*Quadratic Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const;
```

⟨*Quadratic Methods*⟩+≡
```
bool
quadratic::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const {
    ⟨Quadratic intersect common parts⟩

    if ( ! hitQuadratic ) {
        if ( this->_inside( rp ) ) {
            this->forgeHit( direction, start, hit );
            return true;
        }
        return false;
    }

    if ( t1 >= 0.000001 ) {
        mathvec< double > nn;
        double tt = t1;
        ⟨Quadratic intersect fill in hit⟩
    } else if ( t2 >= 0.000001 ) {
        mathvec< double > nn;
        double tt = t2;
        ⟨Quadratic intersect fill in hit⟩
    } else {
        return false;
    }

    return true;
}
```

⟨*Quadratic intersect common parts*⟩≡

```
mathvec< double > rd = direction;
rd.antirotate( this->orientation );
rd /= this->scale;
rd.normalize();

mathvec< double > rp = start - this->center;
rp.antirotate( this->orientation );
rp /= this->scale;

double t1 = -1.0;
double t2 = -1.0;

double AA = 0.0;
double BB = 0.0;
double CC = this->scalar;

unsigned int len = this->squares.size();

for ( unsigned int ii=0; ii < len; ++ii ) {
    for ( unsigned int jj=0; jj < len; ++jj ) {
        double ss = this->squares[ ii ][ jj ];
        AA += ss * rd[ ii ] * rd[ jj ];
        BB += ss * rd[ ii ] * rp[ jj ];
        BB += ss * rp[ ii ] * rd[ jj ];
        CC += ss * rp[ ii ] * rp[ jj ];
    }

    double ll = this->linears[ ii ];
    BB += ll * rd[ ii ];
    CC += ll * rp[ ii ];
}

bool hitQuadratic = false;

if ( ::fabs( AA ) > 0.000001 ) {
    double DD = BB * BB - 4.0 * AA * CC;
    if ( DD > 0.000001 ) {
        double dd = ::sqrt( DD );
        t1 = ( - BB - dd ) / ( 2.0 * AA );
        t2 = ( - BB + dd ) / ( 2.0 * AA );
        if ( t2 < t1 ) {
            double tmp = t1;
            t1 = t2;
            t2 = tmp;
        }
```

```
        hitQuadratic = t2 >= 0.000001;
    }
} else if ( ::fabs( BB ) > 0.000001 ) {
    t1 = -CC / BB;
    hitQuadratic = t1 >= 0.000001;
}
```

⟨*Quadratic intersect fill in hit*⟩≡

```
mathvec< double > spot = rp + ( rd * tt );

nn.resize( len );
for ( unsigned int ii=0; ii < len; ++ii ) {
    double vv = this->linears[ ii ];
    for ( unsigned int jj=0; jj < len; ++jj ) {
        double ff = ( this->squares[ ii ][ jj ] + this->squares[ jj ][ ii ] );
        vv += ff * spot[ jj ];
    }
    nn[ ii ] = vv;
}

if ( this->_inside( rp ) ) {
    nn *= -1.0;
    hit.setInside();
} else {
    hit.setOutside();
}

nn.normalize();

hit.setObjectPosition( spot );
hit.setObjectNormal( nn );
hit.setObjectDirection( rd );

hit.setPosition( spot );
hit.setNormal( nn );
hit.setDirection( rd );

hit.reorient( this->scale, this->orientation, this->center );
hit.setDistance( ( hit.getPosition() - start ).magnitude() );

hit.setObjectColor( &this->chroma );
```

This second intersection method retrieves all intersections between this quadratic and the ray. The ray is specified by its direction and start. The array of hits is sorted in order of increasing distance from the start of the ray.

⟨*Quadratic Method Declarations*⟩+≡

```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const;
```

⟨*Quadratic Methods*⟩+≡

```
bool
quadratic::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const {
    ⟨Quadratic intersect common parts⟩

    hits.clear();

    if ( !hitQuadratic ) {
        if ( this->_inside( rp ) ) {
            intersection* hh = new intersection();
            hits.insert( hits.end(), hh );
            this->forgeHit( direction, start, *hh );
        }
    } else {
        if ( t1 >= 0.000001 ) {
            intersection* h1 = new intersection();
            hits.insert( hits.end(), h1 );

            intersection& hit( *h1 );
            mathvec< double > nn;
            double tt = t1;
            ⟨Quadratic intersect fill in hit⟩
            hit.setOutside();
        }

        if ( t2 >= 0.000001 ) {
            intersection* h2 = new intersection();
            hits.insert( hits.end(), h2 );

            intersection& hit( *h2 );
            mathvec< double > nn;
            double tt = t2;
            ⟨Quadratic intersect fill in hit⟩
            hit.setInside();
        }
    }

    return hits.size() > 0;
}
```

12.5 Checking If A Point Is Inside A Quadratic

⟨*Quadratic Method Declarations*⟩+≡
```
virtual bool inside( const mathvec< double >& start ) const;
virtual bool _inside( const mathvec< double >& rp ) const;
```

⟨*Quadratic Methods*⟩+≡
```
bool
quadratic::inside( const mathvec< double >& start ) const
{
    mathvec< double > rp = start - this->center;
    rp.antirotate( this->orientation );
    rp /= this->scale;

    return this->_inside( rp );
}

bool
quadratic::_inside( const mathvec< double >& rp ) const
{
    unsigned int len = this->center.size();
    double sum = this->scalar;

    for ( unsigned int ii=0; ii < len; ++ii ) {
        double vv = this->linears[ ii ];
        for ( unsigned int jj=0; jj < len; ++jj ) {
            vv += this->squares[ ii ][ jj ] * rp[ jj ];
        }
        sum += vv * rp[ ii ];
    }

    return sum <= 0.0;
}
```

12.6 Loading A Quadratic

The following method allows one to read in an quadratic from a stream.

⟨*Quadratic Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, quadratic& cc );
```

⟨*Quadratic Methods*⟩+≡

```
std::istream&
operator >> ( std::istream& stream, quadratic& qq )
{
    input in( stream );

    qq.scalar = 0.0;

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "squares" ) {
            stream >> qq.squares >> semicolon;
        } else if ( tok == "linears" ) {
            stream >> qq.linears >> semicolon;
        } else if ( tok == "scalar" ) {
            stream >> qq.scalar >> semicolon;
        } else if ( tok == "base" ) {
            stream >> (object&)qq >> semicolon;
        } else {
            in.ignore( "QUADRATIC", tok.c_str() );
        }
    }

    qq.linears.resize( qq.squares.size(), 0.0 );
    for ( unsigned int ii=0; ii < qq.squares.size(); ++ii ) {
        qq.squares[ ii ].resize( qq.squares.size(), 0.0 );
    }

    return stream;
}
```

12.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Quadratic Class Declaration*⟩≡
```
class quadratic : public object {
protected:
    ⟨Quadratic Member Variables⟩
public:
    ⟨Quadratic Method Declarations⟩
public:
    static const char* id;
};
```

12.8 Source Files

The following sections assemble the source files for the `quadratic` class from the chunks above.

12.8.1 `quadratic.h`

The header file the `quadratic` class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header file for the `object` base class and then incorporates the class defined in 12.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*quadratic.h*⟩≡
```
#ifndef _NKLEIN_QUADRATIC_H_
#define _NKLEIN_QUADRATIC_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "object.h"

    ⟨Quadratic Class Declaration⟩

#endif /*_NKLEIN_QUADRATIC_H_*/
```

12.8.2 quadratic.cc

Herein, the implementations of the methods of the `quadratic` class are included as well as the the declarations and implementations of the derived classes.

⟨*quadratic.cc*⟩≡

```
#include <string>
#include <sstream>
#include <math.h>
#include "input.h"
#include "quadratic.h"

const char* quadratic::id = "NKVERSION: quadratic( 2.1.2007.05.17 )";

⟨Quadratic Methods⟩
```

Chapter 13

Convex

Before scaling, orienting, and position, the convex object represents the convex hull (along with the interior) of a given set of points. *Note:* if the points are n-dimensional, but there are not n linearly independent vectors from one point to another in the set of points, the result will be a hyperplane which contains all of the points.

One can specify a matrix of points or a point at a time. The list is never cleared, so one could accumulate points with multiple point vectors and/or multiple point matrixes. One can also specify parameters for the object base class.

⟨*convex bnf*⟩≡

```
    convex_params ::=
            'points' real_matrix ';'
        |   'point' real_vector ';'
        |   'colorDistance' real ';'
        |   'specialColor' integer '{' color_parameters '}' ';'
        |   'base' '{' object_parameters '}' ';'
        ;
```

The `colorDistance` parameter is used in conjunction with the `specialColor` parameters to give different coloring to edges, vertexes, and the like. The `specialColor` has an integer parameter and color information. See §9 for information about the `color_parameters`. The integer parameter tells the dimension of the subfacet to color. For example, zero colors the vertexes, 1 colors the edges, 2 colors the planar faces, etc. The intersection is colored with this color if the point is within `colorDistance` of the right number of planes for this subfacet.

The following snippets are used so that the object's factory method can create instance of this class.

⟨*Derived Object Includes*⟩+≡

```
#include "inter.h"
#include "convex.h"
```

⟨*Derived Object Read Cases*⟩+≡
```
if ( objectType == "convex" ) {
    convex* obj = new convex( ss );
    stream >> *obj >> semicolon;
    return obj;
} else
```

⟨*Convex Data Members*⟩≡
```
std::vector< mathvec< double > > pts;
```

⟨*Convex Protected Data Members*⟩≡
```
double colorDistance;
std::vector< color* > specialColors;
```

13.1 Constructor

The default constructor does nothing.

⟨*Convex Method Declarations*⟩≡
```
convex( const scene* ss );
```

⟨*Convex Methods*⟩≡
```
convex::convex( const scene* ss ) : csginter( ss ), colorDistance( 0.0 )
{
}
```

13.2 Destructor

The destructor has nothing to do.

⟨*Convex Method Declarations*⟩+≡
```
virtual ~convex( void );
```

⟨*Convex Methods*⟩+≡
```
convex::~convex( void ) {
};
```

13.3 Adding A Point

⟨*Convex Method Declarations*⟩+≡
```
void addPoint( const mathvec< double >& point );
```

⟨*Convex Methods*⟩+≡
```
void
convex::addPoint( const mathvec< double >& point )
{
    this->pts.insert( this->pts.end(), point );
};
```

13.4 Calculating The Bounding Planes

This class uses the inherited CSG intersection methods from §18 and then does some simple post-processing on the results. As such, here we adds the bounding hyperplanes to the CSG intersection's list of objects. This method calculates the bounding hyperplanes.

⟨*Convex Method Declarations*⟩+≡
```
void calculateBoundingPlanes( bool dbg = false );
```

This method could stand a major algorithmic upgrade if there are better methods available. For an object with 240 vertexes with $E8$ symmetry, this code has spent more than a month trying to calculate the bounding planes and shows no sign of nearing completion yet.

⟨*Convex Methods*⟩+≡

```
class PlaneInfo {
public:
    PlaneInfo(
        const mathvec< double >& nn,
        const mathvec< double >& pp,
        double oo,
        std::vector< mathvec< double > >& pts
    ) : normal( nn ), point( pp ), offset( oo )
    {
        this->mask.resize( pts.size() );
        for ( unsigned int ii=0; ii < pts.size(); ++ii ) {
            double vv = ( this->normal ^ pts[ ii ] ) - this->offset;
            this->mask[ ii ] = ( ::fabs( vv ) < 0.000001 );
        }
    };

public:
    mathvec< double > normal;
    mathvec< double > point;
    double offset;
    std::vector< bool > mask;
    bool exceeded;
};

void
convex::calculateBoundingPlanes( bool dbg )
{
    std::list< PlaneInfo > planes;

    ⟨Convex calculate dimensions⟩

    ⟨Convex find bounding sphere⟩
    ⟨Convex commit bounding sphere⟩

    ⟨Convex construct choice⟩

    ⟨Convex find initial plane⟩

    if ( planes.size() > 0 ) {
        for ( unsigned int kk=rr; kk < nn; ++kk ) {
            const mathvec< double >& cur( this->pts[ kk ] );
```

```
            ⟨Convex find planes the point exceeds⟩
            if ( pool.size() >= rr-1 ) {
                ⟨Convex make any new planes⟩
            }
        }
        ⟨Convex commit planes⟩
    }

    if ( dbg ) {
        std::cerr << "(" << planes.size() << ") faces" << std::endl;
        std::list< PlaneInfo >::const_iterator it = planes.begin();
        while ( it != planes.end() ) {
            std::cerr << "    " << it->normal << std::endl;
            ++it;
        }
    }
}
```

⟨*Convex calculate dimensions*⟩≡

```
unsigned int nn = this->pts.size();
if ( nn == 0 ) {
    return;
}

unsigned int rr = this->pts[ 0 ].size();
for ( unsigned int ii=1; ii < nn; ++ii ) {
    unsigned int len = this->pts[ ii ].size();
    if ( len > rr ) {
        rr = len;
    }
}

if ( rr > nn ) {
    std::cerr << "CONVEX: Not enough points to form a plane" << std::endl;
    return;
}

for ( unsigned int ii=0; ii < nn; ++ii ) {
    this->pts[ ii ].resize( rr, 0.0 );
}
```

⟨*Convex construct choice*⟩≡
```
mathvec< unsigned int > choice;
choice.resize( rr );
for ( unsigned int ii=0; ii < rr; ++ii ) {
    unsigned int jj = rr - ii - 1;
    choice[ ii ] = jj;
}
```

⟨*Convex find bounding sphere*⟩≡
```
double radius2 = 0.0;
for ( unsigned int kk=0; kk < nn; ++kk ) {
    const mathvec< double >& cur( this->pts[ kk ] );
    double r2 = cur ^ cur;
    if ( r2 > radius2 ) {
        radius2 = r2;
    }
}
```

⟨*Convex commit bounding sphere*⟩≡
```
cylinder* sphere = new cylinder( ::sqrt( radius2 )+0.01, this->myScene );
this->objs.insert( this->objs.begin(), sphere );
```

⟨*Convex find initial plane*⟩≡
```
do {
    ⟨Convex create chosen plane⟩
    if ( plane ) {
        ⟨Convex calculate normal and offset⟩
        ⟨Convex add plane (two-sided)⟩
    }
} while ( planes.size() == 0 && this->incrementChoice( choice, nn ) );
```

⟨*Convex create chosen plane*⟩≡
```
basevec< mathvec< double > > matrix;
matrix.resize( rr-1 );
mathvec< double >& first( this->pts[ choice[ 0 ] ] );
for ( unsigned int ii=1; ii < rr; ++ii ) {
    mathvec< double >& cur( this->pts[ choice[ ii ] ] );
    matrix[ ii-1 ] = cur - first;
}

bool plane = ::orthogonalize( matrix, rr );
```

⟨*Convex calculate normal and offset*⟩≡
```
mathvec< double > norm( matrix[ rr-1 ] );
double dd = ( norm ^ first );
```

⟨*Convex add plane (two-sided)*⟩≡

```
⟨Convex add plane⟩
norm *= -1.0;
dd = ( norm ^ first );
⟨Convex add plane⟩
```

⟨*Convex add plane*⟩≡

```
{
    bool unique = true;

    std::list< PlaneInfo >::iterator pit = planes.begin();
    while ( unique && pit != planes.end() ) {
        if ( ::fabs( pit->offset - dd ) < 0.000001 ) {
            if ( ( pit->normal - norm ).magnitude() < 0.000001 ) {
                unique = false;
            }
        }
        ++pit;
    }

    if ( unique ) {
        planes.insert( planes.end(), PlaneInfo( norm, first, dd, this->pts ) );
    }
}
```

⟨*Convex find planes the point exceeds*⟩≡

```
std::vector< mathvec< double > > pool;
std::vector< bool > pointMask( kk, false );
std::list< PlaneInfo >::iterator pit = planes.begin();
while ( pit != planes.end() ) {
    double diff = ( pit->normal ^ cur ) - pit->offset;
    pit->exceeded = ( diff > 0.000001 );
    if ( pit->exceeded ) {
        for ( unsigned int ii=0; ii < kk; ++ii ) {
            if ( pit->mask[ ii ] == true && pointMask[ ii ] == false ) {
                pool.insert( pool.end(), this->pts[ ii ] );
                pointMask[ ii ] = true;
            }
        }
    }
    std::list< PlaneInfo >::iterator last = pit++;
    if ( last->exceeded ) {
        planes.erase( last );
    }
}
```

```
⟨Convex make any new planes⟩≡
  ⟨Convex construct partial choice⟩
  do {
      ⟨Convex create partially chosen plane⟩
      if ( plane ) {
          ⟨Convex calculate normal and offset⟩
          ⟨Convex verify normal sign⟩
          if ( bounding ) {
              ⟨Convex add plane⟩
          }
      }
  } while ( this->incrementChoice( choice, pool.size() ) );
```

```
⟨Convex construct partial choice⟩≡
  mathvec< unsigned int > choice;
  choice.resize( rr-1 );
  for ( unsigned int ii=0; ii < rr-1; ++ii ) {
      unsigned int jj = rr - ii - 2;
      choice[ ii ] = jj;
  }
```

```
⟨Convex create partially chosen plane⟩≡
  basevec< mathvec< double > > matrix;
  matrix.resize( rr-1 );
  mathvec< double >& first( this->pts[ kk ] );
  for ( unsigned int ii=0; ii < rr-1; ++ii ) {
      mathvec< double >& cur( pool[ choice[ ii ] ] );
      matrix[ ii ] = cur - first;
  }

  bool plane = ::orthogonalize( matrix, rr );
```

⟨*Convex verify normal sign*⟩≡
```
bool bounding = true;
unsigned int above = 0;
unsigned int below = 0;
unsigned int ps = pool.size();
for ( unsigned int ii=0; bounding && ii < ps; ++ii ) {
    double vv = ( norm ^ pool[ ii ] ) - dd;
    if ( vv > 0.000001 ) {
        ++above;
    } else if ( vv < -0.000001 ) {
        ++below;
    }
    bounding = ( above == 0 || below == 0 );
}
if ( bounding && above > 0 ) {
    norm *= -1.0;
    dd = ( norm ^ first );
}
```

⟨*Convex commit planes*⟩≡
```
std::list< PlaneInfo >::iterator pit = planes.begin();
while ( pit != planes.end() ) {
    halfspace* hh = new halfspace( pit->point, pit->normal );
    this->objs.insert( this->objs.end(), hh );
    ++pit;
}
```

13.5 Incrementing A Choice

⟨*Convex Method Declarations*⟩+≡
```
bool incrementChoice( mathvec< unsigned int >& choice, unsigned int nn );
```

⟨*Convex Methods*⟩+≡
```
bool
convex::incrementChoice( mathvec< unsigned int >& choice, unsigned int nn )
{
    unsigned int dims = choice.size();
    bool ret = false;

    for ( unsigned int ii=0; !ret && ii < dims; ++ii ) {
        if ( ++choice[ ii ] < nn - ii  ) {
            for ( unsigned int jj=0; jj < ii; ++jj ) {
                unsigned int kk = ii - jj - 1;
                choice[ kk ] = choice[ ii ] + jj + 1;
            }
            ret = true;
        }
    }

    return ret;
}
```

13.6 Intersection A Ray And A Convex Hull

This first intersection method retrieves only the closest intersection to the ray's origin. The ray is specified by its direction and start.

⟨*Convex Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const;
```

⟨*Convex Methods*⟩+≡
```
bool
convex::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const {
    bool ret = this->csginter::intersect( direction, start, hit );
    if ( ret && this->colorDistance > 0.000001 ) {
        ⟨Convex recolor hit⟩
    }
    return ret;
}
```

To recolor the hit, we run through and count the number of hyperplanes that this hit is near. Then, we use the appropriate color to try

⟨*Convex recolor hit*⟩≡

```
const mathvec< double >& loc = hit.getPosition();
mathvec< double > rp = loc - this->center;
rp.antirotate( this->orientation );
rp /= this->scale;

unsigned int clen = this->center.size();
unsigned int cmax = clen;
for ( unsigned int kk=0; kk < clen; ++kk ) {
    if ( this->specialColors[ kk ] != 0 ) {
        cmax = clen - kk;
        break;
    }
}
unsigned int len = this->specialColors.size();
unsigned int hitCount = 0;
unsigned int cindex = 0;
std::vector< object* >::const_iterator jj = this->objs.begin();
for ( ++jj; hitCount < cmax && jj != this->objs.end(); ++jj ) {
    halfspace* hh = (halfspace*)*jj;
    double dd = ::fabs( hh->distance( rp ) );
    if ( dd <= this->colorDistance ) {
        cindex = clen - ++hitCount;
    }
}
if ( cindex < len ) {
    color* cc = this->specialColors[ cindex ];
    if ( cc != 0 ) {
        hit.recolor( *cc );
    }
}
```

This second intersection method retrieves all intersections between this convex hull and the ray. The ray is specified by its direction and start. The array of hits is sorted in order of increasing distance from the start of the ray.

⟨*Convex Method Declarations*⟩+≡

```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const;
```

The intersection of a set of objects is the complement of the union of the complements. Thus, this logic follows that of the union with inside-ness and outside-ness reversed.

⟨*Convex Methods*⟩+≡
```
bool
convex::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const {
    bool ret = this->csginter::intersect( direction, start, hits );
    if ( ret && this->colorDistance > 0.000001 ) {
        std::vector< intersection* >::iterator it;
        for ( it = hits.begin(); it != hits.end(); ++it ) {
            intersection& hit( *(*it) );
            ⟨Convex recolor hit⟩
        }
    }
    return ret;
}
```

13.7 Loading A Convex

The following method allows one to read in an convex from a stream.

⟨*Convex Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, convex& cc );
```

⟨*Convex Methods*⟩+≡
```
std::istream&
operator >> ( std::istream& stream, convex& hh )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        ⟨Convex special color parsing⟩
        else if ( tok == "points" ) {
            basevec< mathvec< double > > pts;
            stream >> pts >> semicolon;
            for ( unsigned int ii=0; ii < pts.size(); ++ii ) {
                hh.addPoint( pts[ii] );
            }
        } else if ( tok == "point" ) {
            mathvec< double > point;
            stream >> point >> semicolon;
            hh.addPoint( point );
        } else if ( tok == "base" ) {
            stream >> (object&)hh >> semicolon;
        } else {
            in.ignore( "CONVEX", tok.c_str() );
        }
    }

    hh.calculateBoundingPlanes();

    return stream;
}
```

⟨*Convex special color parsing*⟩≡
```
if ( tok == "colorDistance" ) {
    stream >> hh.colorDistance >> semicolon;
} else if ( tok == "specialColor" ) {
    unsigned int ii;
    color* nn = new color( hh.myScene );
    stream >> ii >> *nn;
    if ( ii >= hh.specialColors.size() ) {
        hh.specialColors.resize( ii+1, 0 );
    }
    color* oo = hh.specialColors[ ii ];
    if ( oo != 0 ) {
        delete oo;
    }
    hh.specialColors[ ii ] = nn;
}
```

13.8 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Convex Class Declaration*⟩≡
```
class convex : public csginter {
private:
    ⟨Convex Data Members⟩
protected:
    ⟨Convex Protected Data Members⟩
public:
    ⟨Convex Method Declarations⟩
public:
    static const char* id;
};
```

13.9 Source Files

The following sections assemble the source files for the `convex` class from the chunks above.

13.9.1 convex.h

The header file the `convex` class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header file for the `object` base class and then incorporates the class defined in 13.8 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*convex.h*⟩≡
```
#ifndef _NKLEIN_CONVEX_H_
#define _NKLEIN_CONVEX_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "object.h"

    ⟨Convex Class Declaration⟩

#endif /*_NKLEIN_CONVEX_H_*/
```

13.9.2 convex.cc

Herein, the implementations of the methods of the `convex` class are included as well as the the declarations and implementations of the derived classes.

⟨*convex.cc*⟩≡
```
#include <string>
#include <sstream>
#include <list>
#include <math.h>
#include "input.h"
#include "cylinder.h"
#include "halfspace.h"
#include "inter.h"
#include "convex.h"

const char* convex::id = "NKVERSION: convex( 2.1.2007.05.17 )";

⟨Convex Methods⟩
```

Chapter 14

Coxeter

Before scaling, orienting, and position, the coxter object represents a Wythoff construction based upon the Coxeter-Dynkin diagram of the symmetry group of the vertexes.

One can specify the incidence matrix for the Coxeter-Dynkin diagram. The diagonal elements are used to determine the starting vertex. One can specify a debug flag which has the raytracer emit (to standard error) the number of vertexes and the number of cells.

⟨*coxeter bnf*⟩≡

```
coxeter_params ::=
        coxeter_incidence_matrix
    |   'debug' ';'
    |   'colorDistance' real ';'
    |   'specialColor' integer '{' color_parameters '}' ';'
    |   'base' '{' object_parameters '}' ';'
    ;

coxeter_incidence_matrix ::=
        'matrix' real_matrix ';'
    |   'incidence' real_matrix ';'
    ;
```

See §13 for more information about the `colorDistance` and `specialColor` parameters. See the examples in §41.11 for a good starting point on using the incidence matrix.

The following snippets are used so that the object's factory method can create instance of this class.

⟨*Derived Object Includes*⟩+≡

```
#include "inter.h"
#include "convex.h"
#include "coxeter.h"
```

⟨*Derived Object Read Cases*⟩+≡
```
if ( objectType == "coxeter" ) {
    coxeter* obj = new coxeter( ss );
    stream >> *obj >> semicolon;
    return obj;
} else
```

14.1 Constructor

The default constructor does nothing.

⟨*Coxeter Method Declarations*⟩≡
```
coxeter( const scene* ss );
```

⟨*Coxeter Methods*⟩≡
```
coxeter::coxeter( const scene* ss ) : convex( ss )
{
}
```

14.2 Destructor

This destructor has nothing to do.

⟨*Coxeter Method Declarations*⟩+≡
```
virtual ~coxeter( void );
```

⟨*Coxeter Methods*⟩+≡
```
coxeter::~coxeter( void ) {
}
```

14.3 Loading A Coxeter

This class inherits directly from the convex class defined in §13. The loading routine calculates the symmetry group of the vertexes, calculates an initial vertex, mirrors that vertex through the symmetry group, and feeds those points to the convex class to form their convex hull.

The following method allows one to read in an coxeter from a stream.

⟨*Coxeter Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, coxeter& cc );
```

⟨*Coxeter Methods*⟩+≡
```
std::istream&
operator >> ( std::istream& stream, coxeter& hh )
{
    input in( stream );

    basevec< mathvec< unsigned int > > imat;
    unsigned int limit = 1024;
    bool debug = false;

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        ⟨Convex special color parsing⟩
        else if ( tok == "matrix" || tok == "incidence" ) {
            stream >> imat >> semicolon;
        } else if ( tok == "limit" ) {
            stream >> limit >> semicolon;
        } else if ( tok == "debug" ) {
            debug = true;
            stream >> semicolon;
        } else if ( tok == "base" ) {
            stream >> (object&)hh >> semicolon;
        } else {
            std::cerr << "COXETER: " << tok << std::endl;
        }
    }

    if ( imat.size() > 0 ) {
        unsigned int nn = imat.size();
        ⟨Coxeter make imat square⟩
        ⟨Coxeter calculate dot products⟩
        ⟨Coxeter create vectors⟩
        ⟨Coxeter create vertexes⟩
    }

    return stream;
}
```

⟨*Coxeter make imat square*⟩≡
```
for ( unsigned int jj=0; jj < nn; ++jj ) {
    imat[jj].resize( nn, 0 );
}
```

⟨*Coxeter calculate dot products*⟩≡
```
basevec< mathvec< double > > dd;
dd.resize( nn );

for ( unsigned int jj=0; jj < nn; ++jj ) {
    mathvec< double >& row( dd[ jj ] );

    row.resize( nn, 0.0 );
    row *= 0.0;
    row[ jj ] = 1.0;
}

for ( unsigned int jj=0; jj < (nn-1); ++jj ) {
    for ( unsigned int ii=jj+1; ii < nn; ++ii ) {
        unsigned int mm = imat[ ii ][ jj ];
        unsigned int m2 = imat[ jj ][ ii ];
        if ( m2 > mm ) {
            mm = m2;
        }
        if ( mm < 2 ) {
            mm = 2;
        }
        double cc = ::cos( M_PI / (double)( mm ) );
        dd[ ii ][ jj ] = cc;
        dd[ jj ][ ii ] = cc;
    }
}
```

⟨*Coxeter create vectors*⟩≡

```
basevec< mathvec< double > > aa;
aa.resize( nn );

aa[ 0 ].resize( nn, 0.0 );
aa[ 0 ] *= 0.0;
aa[ 0 ][ 0 ] = 1.0;

for ( unsigned int kk=1; kk < nn; ++kk ) {
    mathvec< double >& cur( aa[ kk ] );
    double ll = 0.0;

    cur.resize( nn, 0.0 );
    cur *= 0.0;

    for ( unsigned int jj=0; jj < kk; ++jj ) {
        const mathvec< double >& versus( aa[ jj ] );
        double ss = 0.0;
        for ( unsigned int ii=0; ii < jj; ++ii ) {
            ss += cur[ ii ] * versus[ ii ];
        }
        cur[ jj ] = ( dd[ jj ][ kk ] - ss ) / versus[ jj ];
        ll += cur[ jj ] * cur[ jj ];
    }

    cur[ kk ] = ::sqrt( 1.0 - ll );
}
```

⟨*Coxeter create vertexes*⟩≡

```
unsigned int oldSize = 0;

basevec< mathvec< double > > xx;

xx.resize( xx.size() + 1 );
xx[ 0 ].resize( nn );

⟨Coxeter create initial vertex⟩

unsigned int newSize = xx.size();

while ( newSize > oldSize && newSize < limit ) {
    unsigned int cntr = 0;

    for ( unsigned int jj=0; jj < nn; ++jj ) {
        for ( unsigned int ii=oldSize; ii < newSize; ++ii ) {
            mathvec< double > vv;
            ⟨Coxeter reflect vector⟩
            ⟨Coxeter check uniqueness⟩
        }
    }

    oldSize = newSize;
    newSize += cntr;
}

if ( debug ) {
    std::cerr << "(" << xx.size() << ") vertexes" << std::endl;
    std::cerr << xx << std::endl;
}

for ( unsigned int jj=0; jj < xx.size(); ++jj ) {
    hh.addPoint( xx[ jj ] );
}

hh.calculateBoundingPlanes( debug );
```

⟨*Coxeter create initial vertex*⟩≡
```
mathvec< double > bb( nn );
for ( unsigned int ii=0; ii < nn; ++ii ) {
    bb[ ii ] = (double)imat[ ii ][ ii ];
}

if ( ( bb ^ bb ) < 0.5 ) {
    bb[ 0 ] = 1.0;
}

xx[ 0 ] = bb.solve( aa );
if ( ( xx[ 0 ] ^ xx[ 0 ] ) < 0.5 ) {
    if ( debug ) {
        std::cerr << "bad solution: " << xx[0] << std::endl;
    }
    xx[ 0 ] = aa[ 0 ];
} else {
    xx[ 0 ].normalize();
    if ( debug ) {
        std::cerr << aa << " * " << xx[0] << " = " << bb << std::endl;
    }
}
```

⟨*Coxeter reflect vector*⟩≡
```
vv = aa[ jj ];
vv *= -2.0 * ( aa[ jj ] ^ xx[ ii ] );
vv += xx[ ii ];
```

⟨*Coxeter check uniqueness*⟩≡
```
bool unique = true;
for ( unsigned int kk=0; unique && kk < newSize+cntr; ++kk ) {
    double dd = ( vv - xx[ kk ] ).magnitude();
    unique = dd > 0.000001;
}
if ( unique ) {
    xx.resize( newSize + cntr + 1 );
    xx[ newSize + cntr ] = vv;
    ++cntr;
}
```

14.4 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Coxeter Class Declaration*⟩≡
```
class coxeter : public convex {
public:
    ⟨Coxeter Method Declarations⟩
public:
    static const char* id;
};
```

14.5 Source Files

The following sections assemble the source files for the `coxeter` class from the chunks above.

14.5.1 `coxeter.h`

The header file the `coxeter` class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header file for the `object` base class and then incorporates the class defined in 14.4 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*coxeter.h*⟩≡
```
#ifndef _NKLEIN_COXETER_H_
#define _NKLEIN_COXETER_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "object.h"

    ⟨Coxeter Class Declaration⟩

#endif /*_NKLEIN_COXETER_H_*/
```

14.5.2 coxeter.cc

Herein, the implementations of the methods of the coxeter class are included as well as the the declarations and implementations of the derived classes.

⟨*coxeter.cc*⟩≡

```
#include <string>
#include <sstream>
#include <math.h>
#include "input.h"
#include "halfspace.h"
#include "inter.h"
#include "convex.h"
#include "coxeter.h"

const char* coxeter::id = "NKVERSION: coxeter( 2.1.2007.05.17 )";
```

⟨*Coxeter Methods*⟩

Chapter 15

Complement

The complement is a CSG (Constructive Solid Geometry) primitive. Intersections with it are simply intersections with the object it contains with all of the inside hits turned to outside hits (and vice-versa). The complement contains a sub-object. And, it can specify parameters from the base object class.

⟨*complement bnf*⟩≡

```
    complement_params ::=
            'object' object_type_and_params ';'
        |   'base' '{' object_parameters '}' ';'
        ;
```

The complement class can only contain one sub-object.

The following snippets are used so that the object's factory method can create instance of this class.

⟨*Derived Object Includes*⟩+≡

```
#include "complement.h"
```

⟨*Derived Object Read Cases*⟩+≡

```
if ( objectType == "complement" ) {
    complement* obj = new complement( ss );
    stream >> *obj >> semicolon;
    return obj;
} else
```

The `complement` class is a subclass of the `object` class from 9. It contains a sub-object. It simply flips the inside/outside-ness of the object it contains.

⟨*Complement Member Variables*⟩≡

```
object* obj;
```

⟨*Complement Member Initializations*⟩≡

```
, obj( 0 )
```

15.1 Constructor

The default constructor does nothing.

⟨*Complement Method Declarations*⟩≡
```
complement( const scene* ss );
```

⟨*Complement Methods*⟩≡
```
complement::complement( const scene* ss )
    : object( ss ) ⟨Complement Member Initializations⟩
{
}
```

15.2 Destructor

This destructor has to delete the sub-object.

⟨*Complement Method Declarations*⟩+≡
```
virtual ~complement( void );
```

⟨*Complement Methods*⟩+≡
```
complement::~complement( void ) {
    delete this->obj;
};
```

15.3 Setting The Number of Dimensions

Even though this method is virtual, there is an implementation that derived classes should invoke.

⟨*Complement Method Declarations*⟩+≡
```
virtual void setDimensions( unsigned int dd );
```

⟨*Complement Methods*⟩+≡
```
void
complement::setDimensions( unsigned int dd )
{
    if ( this->obj != 0 ) {
        this->obj->setDimensions( dd );
    }
    this->object::setDimensions( dd );
}
```

15.4 Intersection A Ray And A Complement

This first intersection method retrieves only the closest intersection to the ray's origin. The ray is specified by its direction and start.

⟨*Complement Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const;
```

⟨*Complement Methods*⟩+≡
```
bool
complement::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const {
    if ( this->obj == 0 ) {
        return false;
    }
    ⟨Complement intersect common parts⟩
    bool ret = this->obj->intersect( rd, rp, hit );
    if ( ret ) {
        ⟨Complement update hit⟩
    }
    return ret;
}
```

⟨*Complement intersect common parts*⟩≡
```
mathvec< double > rd = direction;
rd.antirotate( this->orientation );
rd /= this->scale;
rd.normalize();

mathvec< double > rp = start - this->center;
rp.antirotate( this->orientation );
rp /= this->scale;
```

⟨*Complement update hit*⟩≡
```
hit.reorient( this->scale, this->orientation, this->center );
hit.setDistance( ( hit.getPosition() - start ).magnitude() );
if ( this->gotColor ) {
    hit.recolor( this->chroma );
}
if ( hit.isInside() ) {
    hit.setOutside();
} else {
    hit.setInside();
}
```

This second intersection method retrieves all intersections between this complement and the ray. The ray is specified by its direction and start. The array of hits is sorted in order of increasing distance from the start of the ray.

⟨*Complement Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const;
```

⟨*Complement Methods*⟩+≡
```
bool
complement::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const {
    if ( this->obj == 0 ) {
        return false;
    }
    ⟨Complement intersect common parts⟩
    bool ret = this->obj->intersect( rd, rp, hits );
    if ( ret ) {
        for ( unsigned int ii=0; ii < hits.size(); ++ii ) {
            intersection& hit( *hits[ ii ] );
            ⟨Complement update hit⟩
        }
    } else {
        intersection* nn = new intersection();
        this->forgeHit( direction, start, *nn );
        hits.insert( hits.end(), nn );
    }

    return true;
}
```

15.5 Checking If A Point Is Inside A Complement

⟨*Complement Method Declarations*⟩+≡
```
virtual bool inside( const mathvec< double >& start ) const;
virtual bool _inside( const mathvec< double >& rp ) const;
```

⟨*Complement Methods*⟩+≡
```
bool
complement::inside( const mathvec< double >& start ) const
{
    if ( this->obj == 0 ) {
        return false;
    }
    mathvec< double > rp = start - this->center;
    rp.antirotate( this->orientation );
    rp /= this->scale;

    return this->_inside( rp );
}

bool
complement::_inside( const mathvec< double >& rp ) const
{
    if ( this->obj == 0 ) {
        return false;
    }
    return ! this->obj->inside( rp );
}
```

15.6 Loading A Complement

The following method allows one to read in an complement from a stream.
⟨*Complement Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, complement& cc );
```

⟨*Complement Methods*⟩+≡

```
std::istream&
operator >> ( std::istream& stream, complement& cc )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "object" ) {
            object* nn = object::read( stream, cc.myScene );
            if ( nn != 0 ) {
                delete cc.obj;
                cc.obj = nn;
            } else {
                std::cerr << "COMPLEMENT got no object!" << std::endl;
            }
        } else if ( tok == "base" ) {
            stream >> (object&)cc >> semicolon;
        } else {
            in.ignore( "COMPLEMENT", tok.c_str() );
        }
    }

    return stream;
}
```

15.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Complement Class Declaration*⟩≡
```
class complement : public object {
protected:
    ⟨Complement Member Variables⟩
public:
    ⟨Complement Method Declarations⟩
public:
    static const char* id;
};
```

15.8 Source Files

The following sections assemble the source files for the `complement` class from the chunks above.

15.8.1 complement.h

The header file the `complement` class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header file for the `object` base class and then incorporates the class defined in 15.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*complement.h*⟩≡
```
#ifndef _NKLEIN_COMPLEMENT_H_
#define _NKLEIN_COMPLEMENT_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "object.h"

    ⟨Complement Class Declaration⟩

#endif /*_NKLEIN_COMPLEMENT_H_*/
```

15.8.2 complement.cc

Herein, the implementations of the methods of the `complement` class are included as well as the the declarations and implementations of the derived classes.

⟨*complement.cc*⟩≡

```
#include <string>
#include <sstream>
#include <math.h>
#ifndef MAXFLOAT
#include <values.h>
#endif
#include "input.h"
#include "complement.h"

const char* complement::id = "NKVERSION: complement( 2.1.2007.05.17 )";

⟨Complement Methods⟩
```

Chapter 16

Union

The union is a CSG (Constructive Solid Geometry) primitive. If this object contains sub-objects A_i, then this object represents the object with boundary $\partial \bigcup A_i$. The union contains zero or more sub-objects. And, it can specify parameters from the base object class.

⟨*csg union bnf*⟩≡

```
    csg_union_params ::=
            'object' object_type_and_params ';'
        |   'base' '{' object_parameters '}' ';'
        ;
```

The following snippets are used so that the object's factory method can create instance of this class.

⟨*Derived Object Includes*⟩+≡

```
#include "union.h"
```

⟨*Derived Object Read Cases*⟩+≡

```
if ( objectType == "union" ) {
    csgunion* obj = new csgunion( ss );
    stream >> *obj >> semicolon;
    return obj;
} else
```

The `union` class is a subclass of the `object` class from 9. It contains a list of sub-objects.

⟨*Union Member Variables*⟩≡

```
std::vector< object* > objs;
```

16.1 Constructor

The default constructor does nothing.

⟨*Union Method Declarations*⟩≡
```
csgunion( const scene* ss );
```

⟨*Union Methods*⟩≡
```
csgunion::csgunion( const scene* ss ) : object( ss )
{
}
```

16.2 Destructor

This destructor has to release all of the sub-objects.

⟨*Union Method Declarations*⟩+≡
```
virtual ~csgunion( void );
```

⟨*Union Methods*⟩+≡
```
csgunion::~csgunion( void ) {
    std::vector< object* >::iterator it;
    for ( it=this->objs.begin(); it != this->objs.end(); ++it ) {
        delete *it;
    }
};
```

16.3 Setting The Number of Dimensions

This method has to update the dimension of all of its sub-objects and the base class.

⟨*Union Method Declarations*⟩+≡
```
virtual void setDimensions( unsigned int dd );
```

⟨*Union Methods*⟩+≡
```
void
csgunion::setDimensions( unsigned int dd )
{
    std::vector< object* >::iterator it;
    for ( it=this->objs.begin(); it != this->objs.end(); ++it ) {
        (*it)->setDimensions( dd );
    }
    this->object::setDimensions( dd );
}
```

16.4 Intersection A Ray And A Union

This first intersection method retrieves only the closest intersection to the ray's origin. The ray is specified by its direction and start.

⟨*Union Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const;
```

⟨*Union Methods*⟩+≡
```
bool
csgunion::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const {
    std::vector< intersection* > hits;
    bool ret = this->intersect( direction, start, hits );
    if ( ret ) {
        hit = *hits[ 0 ];
        std::vector< intersection* >::iterator it;
        for ( it = hits.begin(); it != hits.end(); ++it ) {
            delete *it;
        }
    }
    return ret;
}
```

This second intersection method retrieves all intersections between this union and the ray. The ray is specified by its direction and start. The array of hits is sorted in order of increasing distance from the start of the ray.

⟨*Union Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const;
```

⟨*Union Methods*⟩+≡
```
bool
csgunion::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const {
    hits.clear();
    ⟨Union intersect common parts⟩
    ⟨Union perform all intersections⟩
    if ( gotHit ) {
        ⟨Union create union of hits⟩
        ⟨Union clean up hits⟩
        if ( hits.size() == 0 ) {
            intersection* nn = new intersection();
            this->forgeHit( direction, start, *nn );
            hits.insert( hits.end(), nn );
        }
    }
    return gotHit;
}
```

⟨*Union intersect common parts*⟩≡
```
mathvec< double > rd = direction;
rd.antirotate( this->orientation );
rd /= this->scale;
rd.normalize();

mathvec< double > rp = start - this->center;
rp.antirotate( this->orientation );
rp /= this->scale;
```

⟨*Union perform all intersections*⟩≡
```
unsigned int len = this->objs.size();

std::vector< std::vector< intersection* > > hitLists( len );
std::vector< unsigned int > hitStarts( len );

bool gotHit = false;
bool amInside = false;

for ( unsigned int ii=0; ii < len; ++ii ) {
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    object* obj = this->objs[ ii ];

    if ( obj->intersect( rd, rp, curHits ) ) {
        gotHit = true;
        if ( ! amInside ) {
            amInside = curHits[ 0 ]->isInside();
        }
    }
    hitStarts[ ii ] = 0;
}
```

⟨*Union create union of hits*⟩≡
```
intersection* bestHit;
unsigned int bestII = 0;
unsigned int bestJJ = 0;
do {
    bestHit = 0;
    if ( amInside ) {
        ⟨Union find farthest inside hit⟩
    } else {
        ⟨Union find closest outside hit⟩
        ⟨Union advance all inside folks⟩
    }
    if ( bestHit != 0 ) {
        ⟨Union add hit⟩
    }
    amInside = !amInside;
} while ( bestHit != 0 );
```

⟨*Union find farthest inside hit*⟩≡
```
bool done = false;
while ( !done ) {
    ⟨Union find next inside hit⟩
    ⟨Union make sure hit outside all other objects⟩
}
```

⟨*Union find next inside hit*⟩≡
```
bool notSet = true;
double dist = MAXFLOAT;
for ( unsigned int ii=0; ii < len; ++ii ) {
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    unsigned int hStart = hitStarts[ ii ];
    unsigned int hLen = curHits.size();
    if ( hStart < hLen ) {
        intersection* hit = curHits[ hStart ];
        if ( hit->isInside() ) {
            double curDist = hit->getDistance();
            if ( notSet || curDist < dist ) {
                bestII = ii;
                bestJJ = hStart;
                bestHit = hit;
                dist = curDist - 0.000001;
                notSet = false;
            }
        }
    }
}
if ( bestHit == 0 ) {
    break;
} else {
    hitStarts[ bestII ] = bestJJ + 1;
}
```

Assume we're done unless something is inside.

⟨*Union make sure hit outside all other objects*⟩≡

```
bool outsideAll = true;
for ( unsigned int ii=0; outsideAll && ii < len; ++ii ) {
    if ( ii == bestII ) {
        continue;
    }
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    unsigned int hLen = curHits.size();

    for ( unsigned int jj=hitStarts[ ii ]; jj < hLen; ++jj ) {
        intersection* hit = curHits[ jj ];
        double curDist = hit->getDistance();
        if ( curDist < dist ) {
            ++hitStarts[ ii ];
        } else if ( hit->isInside() ) {
            outsideAll = false;
            break;
        } else {
            break;
        }
    }
    if ( hitStarts[ ii ] == hLen && hLen > 0 && curHits[ hLen-1 ] != 0
    && curHits[ hLen-1  ]->isOutside() ) {
        outsideAll = false;
    }
}
if ( !outsideAll ) {
    bestHit = 0;
} else {
    done = true;
}
```

⟨*Union find closest outside hit*⟩≡
```
bool notSet = true;
double dist = MAXFLOAT;
for ( unsigned int ii=0; ii < len; ++ii ) {
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    unsigned int hStart = hitStarts[ ii ];
    unsigned int hLen = curHits.size();
    if ( hStart < hLen ) {
        intersection* hit = curHits[ hStart ];
        if ( hit->isOutside() ) {
            double curDist = hit->getDistance();
            if ( notSet || curDist < dist ) {
                bestII = ii;
                bestJJ = hStart;
                bestHit = hit;
                dist = curDist - 0.000001;
                notSet = false;
            }
        }
    }
}
```

⟨*Union advance all inside folks*⟩≡
```
if ( bestHit != 0 ) {
    for ( unsigned int ii=0; ii < len; ++ii ) {
        if ( ii == bestII ) {
            continue;
        }
        std::vector< intersection* >& curHits( hitLists[ ii ] );
        unsigned int hLen = curHits.size();

        for ( unsigned int jj=hitStarts[ ii ]; jj < hLen; ++jj ) {
            intersection* hit = curHits[ jj ];
            double curDist = hit->getDistance();
            if ( curDist < dist ) {
                ++hitStarts[ ii ];
            }
        }
    }
}
```

⟨*Union add hit*⟩≡
```
std::vector< intersection* >& curHits( hitLists[ bestII ] );
curHits[ bestJJ ] = 0;
hitStarts[ bestII ] = bestJJ+1;
bestHit->reorient( this->scale, this->orientation, this->center );
bestHit->setDistance( ( bestHit->getPosition() - start ).magnitude() );
if ( this->gotColor ) {
    bestHit->recolor( this->chroma );
}
hits.insert( hits.end(), bestHit );
```

⟨*Union clean up hits*⟩≡
```
for ( unsigned int ii=0; ii < len; ++ii ) {
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    unsigned int hLen = curHits.size();
    for ( unsigned int jj=0; jj < hLen; ++jj ) {
        delete curHits[ jj ];
    }
    curHits.clear();
}
```

16.5 Checking If A Point Is Inside A Union

⟨*Union Method Declarations*⟩+≡
```
virtual bool inside( const mathvec< double >& start ) const;
virtual bool _inside( const mathvec< double >& rp ) const;
```

⟨*Union Methods*⟩+≡
```
bool
csgunion::inside( const mathvec< double >& start ) const
{
    mathvec< double > rp = start - this->center;
    rp.antirotate( this->orientation );
    rp /= this->scale;

    return this->_inside( rp );
}

bool
csgunion::_inside( const mathvec< double >& rp ) const
{
    unsigned int len = this->objs.size();
    bool outside = true;

    for ( unsigned int ii=0; outside && ii < len; ++ii ) {
        outside = ! this->objs[ ii ]->inside( rp );
    }

    return ! outside;
}
```

16.6 Loading A Union

The following method allows one to read in an union from a stream.

⟨*Union Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, csgunion& cc );
```

⟨*Union Methods*⟩+≡
```
std::istream&
operator >> ( std::istream& stream, csgunion& cc )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "object" ) {
            object* nn = object::read( stream, cc.myScene );
            if ( nn != 0 ) {
                cc.objs.insert( cc.objs.end(), nn );
            } else {
                std::cerr << "UNION failed loading an object!" << std::endl;
            }
        } else if ( tok == "base" ) {
            stream >> (object&)cc >> semicolon;
        } else {
            in.ignore( "UNION", tok.c_str() );
        }
    }

    return stream;
}
```

16.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Union Class Declaration*⟩≡

```
class csgunion : public object {
protected:
    ⟨Union Member Variables⟩
public:
    ⟨Union Method Declarations⟩
public:
    static const char* id;
};
```

16.8 Source Files

The following sections assemble the source files for the `union` class from the chunks above.

16.8.1 union.h

The header file the `union` class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header file for the `object` base class and then incorporates the class defined in 16.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*union.h*⟩≡

```
#ifndef _NKLEIN_UNION_H_
#define _NKLEIN_UNION_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "object.h"

    ⟨Union Class Declaration⟩

#endif /*_NKLEIN_UNION_H_*/
```

16.8.2 union.cc

Herein, the implementations of the methods of the union class are included.

⟨*union.cc*⟩≡

```
#include <string>
#include <sstream>
#include <math.h>
#ifndef MAXFLOAT
#include <values.h>
#endif
#include "input.h"
#include "union.h"

const char* csgunion::id = "NKVERSION: csgunion( 2.1.2007.05.17 )";
```

⟨*Union Methods*⟩

Chapter 17

Set

The set is a CSG (Constructive Solid Geometry) primitive. If this object contains sub-objects A_i, then this object represents the object with boundary $\bigcup \partial A_i$. The set contains zero or more sub-objects. And, it can specify parameters from the base object class.

⟨*csg set bnf*⟩≡

```
csg_set_params ::=
        'object' object_type_and_params ';'
    |   'base' '{' object_parameters '}' ';'
    ;
```

The following snippets are used so that the object's factory method can create instance of this class.

⟨*Derived Object Includes*⟩+≡

```
#include "set.h"
```

⟨*Derived Object Read Cases*⟩+≡

```
if ( objectType == "set" ) {
    csgset* obj = new csgset( ss );
    stream >> *obj >> semicolon;
    return obj;
} else
```

The `set` class is a subclass of the `object` class from 9. It contains a list of sub-objects.

⟨*Set Member Variables*⟩≡

```
std::vector< object* > objs;
```

17.1 Constructor

The default constructor does nothing.

⟨*Set Method Declarations*⟩≡
```
csgset( const scene* ss );
```

⟨*Set Methods*⟩≡
```
csgset::csgset( const scene* ss ) : object( ss )
{
}
```

17.2 Destructor

This destructor has to release all of the sub-objects.

⟨*Set Method Declarations*⟩+≡
```
virtual ~csgset( void );
```

⟨*Set Methods*⟩+≡
```
csgset::~csgset( void ) {
    std::vector< object* >::iterator it;
    for ( it=this->objs.begin(); it != this->objs.end(); ++it ) {
        delete *it;
    }
};
```

17.3 Setting The Number of Dimensions

This method has to update the dimension of all of its sub-objects and the base class.

⟨*Set Method Declarations*⟩+≡
```
virtual void setDimensions( unsigned int dd );
```

⟨*Set Methods*⟩+≡
```
void
csgset::setDimensions( unsigned int dd )
{
    std::vector< object* >::iterator it;
    for ( it=this->objs.begin(); it != this->objs.end(); ++it ) {
        (*it)->setDimensions( dd );
    }
    this->object::setDimensions( dd );
}
```

17.4 Intersection A Ray And A Set

This first intersection method retrieves only the closest intersection to the ray's origin. The ray is specified by its direction and start.

⟨*Set Method Declarations*⟩+≡

```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const;
```

⟨*Set Methods*⟩+≡

```
bool
csgset::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const {
    std::vector< intersection* > hits;
    bool ret = this->intersect( direction, start, hits );
    if ( ret ) {
        hit = *hits[ 0 ];
        std::vector< intersection* >::iterator it;
        for ( it = hits.begin(); it != hits.end(); ++it ) {
            delete *it;
        }
    }
    return ret;
}
```

This second intersection method retrieves all intersections between this set and the ray. The ray is specified by its direction and start. The array of hits is sorted in order of increasing distance from the start of the ray.

⟨*Set Method Declarations*⟩+≡

```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const;
```

⟨*Set Methods*⟩+≡
```
bool
csgset::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const {
    ⟨Set intersect common parts⟩
    ⟨Set perform all intersections⟩
    if ( gotHit ) {
        ⟨Set create set of hits⟩
    }
    return gotHit;
}
```

⟨*Set intersect common parts*⟩≡
```
mathvec< double > rd = direction;
rd.antirotate( this->orientation );
rd /= this->scale;
rd.normalize();

mathvec< double > rp = start - this->center;
rp.antirotate( this->orientation );
rp /= this->scale;
```

⟨*Set perform all intersections*⟩≡
```
unsigned int len = this->objs.size();

std::vector< std::vector< intersection* > > hitLists( len );
std::vector< unsigned int > hitStarts( len );

bool gotHit = false;

for ( unsigned int ii=0; ii < len; ++ii ) {
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    object* obj = this->objs[ ii ];

    if ( obj->intersect( rd, rp, curHits ) ) {
        gotHit = true;
    }
    hitStarts[ ii ] = 0;
}
```

⟨*Set create set of hits*⟩≡
```
unsigned int bestII = 0;
unsigned int bestJJ = 0;
intersection* bestHit;
do {
    bestHit = 0;
    ⟨Set find closest hit⟩
    if ( bestHit != 0 ) {
        ⟨Set add hit⟩
    }
} while ( bestHit != 0 );
```

⟨*Set find closest hit*⟩≡
```
bool notSet = true;
double dist = MAXFLOAT;
for ( unsigned int ii=0; ii < len; ++ii ) {
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    unsigned int hStart = hitStarts[ ii ];
    unsigned int hLen = curHits.size();
    if ( hStart < hLen ) {
        intersection* hit = curHits[ hStart ];
        double curDist = hit->getDistance();
        if ( notSet || curDist < dist ) {
            bestII = ii;
            bestJJ = hStart;
            bestHit = hit;
            dist = curDist;
            notSet = false;
        }
    }
}
```

⟨*Set add hit*⟩≡
```
std::vector< intersection* >& curHits( hitLists[ bestII ] );
curHits[ bestJJ ] = 0;
hitStarts[ bestII ] = bestJJ+1;
bestHit->reorient( this->scale, this->orientation, this->center );
bestHit->setDistance( ( bestHit->getPosition() - start ).magnitude() );
if ( this->gotColor ) {
    bestHit->recolor( this->chroma );
}
hits.insert( hits.end(), bestHit );
```

17.5 Checking If A Point Is Inside A Set

⟨*Set Method Declarations*⟩+≡
```
virtual bool inside( const mathvec< double >& start ) const;
virtual bool _inside( const mathvec< double >& rp ) const;
```

⟨*Set Methods*⟩+≡
```
bool
csgset::inside( const mathvec< double >& start ) const
{
    mathvec< double > rp = start - this->center;
    rp.antirotate( this->orientation );
    rp /= this->scale;

    return this->_inside( rp );
}

bool
csgset::_inside( const mathvec< double >& rp ) const
{
    unsigned int len = this->objs.size();
    bool outside = true;

    for ( unsigned int ii=0; outside && ii < len; ++ii ) {
        outside = ! this->objs[ ii ]->inside( rp );
    }

    return ! outside;
}
```

17.6 Loading A Set

The following method allows one to read in an set from a stream.

⟨*Set Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, csgset& cc );
```

⟨*Set Methods*⟩+≡
```
std::istream&
operator >> ( std::istream& stream, csgset& cc )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "object" ) {
            object* nn = object::read( stream, cc.myScene );
            if ( nn != 0 ) {
                cc.objs.insert( cc.objs.end(), nn );
            } else {
                std::cerr << "SET failed to get an object!" << std::endl;
            }
        } else if ( tok == "base" ) {
            stream >> (object&)cc >> semicolon;
        } else {
            in.ignore( "SET", tok.c_str() );
        }
    }

    return stream;
}
```

17.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Set Class Declaration*⟩≡
```
class csgset : public object {
protected:
    ⟨Set Member Variables⟩
public:
    ⟨Set Method Declarations⟩
public:
    static const char* id;
};
```

17.8 Source Files

The following sections assemble the source files for the `set` class from the chunks above.

17.8.1 `set.h`

The header file the `set` class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header file for the `object` base class and then incorporates the class defined in 17.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*set.h*⟩≡
```
#ifndef _NKLEIN_SET_H_
#define _NKLEIN_SET_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "object.h"

    ⟨Set Class Declaration⟩

#endif /*_NKLEIN_SET_H_*/
```

17.8.2 set.cc

Herein, the implementations of the methods of the set class are included.

⟨*set.cc*⟩≡

```
#include <string>
#include <sstream>
#include <math.h>
#ifndef MAXFLOAT
#include <values.h>
#endif
#include "input.h"
#include "set.h"

const char* csgset::id = "NKVERSION: csgset( 2.1.2007.05.17 )";

⟨Set Methods⟩
```

Chapter 18

Inter

The intersection is a CSG (Constructive Solid Geometry) primitive. If this object contains sub-objects A_i, then this object represents the object with boundary $\partial \bigcap A_i$. The intersection contains zero or more sub-objects. And, it can specify parameters from the base object class.

⟨*csg intersection bnf*⟩≡

```
    csg_intersection_params ::=
            'object' object_type_and_params ';'
        |   'base' '{' object_parameters '}' ';'
        ;
```

The following snippets are used so that the object's factory method can create instance of this class.

⟨*Derived Object Includes*⟩+≡

```
#include "inter.h"
```

⟨*Derived Object Read Cases*⟩+≡

```
if ( objectType == "intersection" || objectType == "inter" ) {
    csginter* obj = new csginter( ss );
    stream >> *obj >> semicolon;
    return obj;
} else
```

The `inter` class is a subclass of the `object` class from 9. It contains a list of sub-objects.

⟨*Inter Member Variables*⟩≡

```
std::vector< object* > objs;
```

18.1 Constructor

The default constructor does nothing.

⟨*Inter Method Declarations*⟩≡
```
csginter( const scene* ss );
```

⟨*Inter Methods*⟩≡
```
csginter::csginter( const scene* ss ) : object( ss )
{
}
```

18.2 Destructor

This destructor has to release all of the sub-objects.

⟨*Inter Method Declarations*⟩+≡
```
virtual ~csginter( void );
```

⟨*Inter Methods*⟩+≡
```
csginter::~csginter( void ) {
    std::vector< object* >::iterator it;
    for ( it=this->objs.begin(); it != this->objs.end(); ++it ) {
        delete *it;
    }
};
```

18.3 Setting The Number of Dimensions

This method has to update the dimensions of each of the sub-objects and of the base class.

⟨*Inter Method Declarations*⟩+≡
```
virtual void setDimensions( unsigned int dd );
```

⟨*Inter Methods*⟩+≡
```
void
csginter::setDimensions( unsigned int dd )
{
    std::vector< object* >::iterator it;
    for ( it=this->objs.begin(); it != this->objs.end(); ++it ) {
        (*it)->setDimensions( dd );
    }
    this->object::setDimensions( dd );
}
```

18.4 Intersection A Ray And A Inter

This first intersection method retrieves only the closest intersection to the ray's origin. The ray is specified by its direction and start.

⟨*Inter Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const;
```

⟨*Inter Methods*⟩+≡
```
bool
csginter::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const {
    std::vector< intersection* > hits;
    bool ret = this->intersect( direction, start, hits );
    if ( ret ) {
        hit = *hits[ 0 ];
        std::vector< intersection* >::iterator it;
        for ( it = hits.begin(); it != hits.end(); ++it ) {
            delete *it;
        }
    }
    return ret;
}
```

This second intersection method retrieves all intersections between this inter and the ray. The ray is specified by its direction and start. The array of hits is sorted in order of increasing distance from the start of the ray.

⟨*Inter Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const;
```

The intersection of a set of objects is the complement of the union of the complements. Thus, this logic follows that of the union with inside-ness and outside-ness reversed.

⟨*Inter Methods*⟩+≡
```
bool
csginter::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const {
    hits.clear();
    ⟨Inter intersect common parts⟩
    ⟨Inter perform all intersections⟩
    if ( allHit ) {
        ⟨Inter create inter of hits⟩
    }
    ⟨Inter clean up hits⟩
    return hits.size() > 0;
}
```

⟨*Inter intersect common parts*⟩≡
```
mathvec< double > rd = direction;
rd.antirotate( this->orientation );
rd /= this->scale;
rd.normalize();

mathvec< double > rp = start - this->center;
rp.antirotate( this->orientation );
rp /= this->scale;
```

⟨*Inter perform all intersections*⟩≡
```
unsigned int len = this->objs.size();

std::vector< std::vector< intersection* > > hitLists( len );
std::vector< unsigned int > hitStarts( len );

bool allHit = true;
bool amOutside = false;

for ( unsigned int ii=0; allHit && ii < len; ++ii ) {
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    object* obj = this->objs[ ii ];

    if ( obj->intersect( rd, rp, curHits ) ) {
        if ( ! amOutside ) {
            amOutside = curHits[ 0 ]->isOutside();
        }
    } else {
        allHit = false;
    }
    hitStarts[ ii ] = 0;
}
```

⟨*Inter create inter of hits*⟩≡
```
intersection* bestHit;
unsigned int bestII = 0;
unsigned int bestJJ = 0;
do {
    bestHit = 0;
    if ( amOutside ) {
        ⟨Inter find farthest outside hit⟩
    } else {
        ⟨Inter find closest inside hit⟩
        ⟨Inter advance all outside folks⟩
    }
    if ( bestHit != 0 ) {
        ⟨Inter add hit⟩
    }
    amOutside = !amOutside;
} while ( bestHit != 0 );
```

⟨*Inter find farthest outside hit*⟩≡
```
bool done = false;
while ( !done ) {
    ⟨Inter find next outside hit⟩
    ⟨Inter make sure hit inside all other objects⟩
}
```

⟨*Inter find next outside hit*⟩≡
```
bool notSet = true;
double dist = MAXFLOAT;
for ( unsigned int ii=0; ii < len; ++ii ) {
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    unsigned int hStart = hitStarts[ ii ];
    unsigned int hLen = curHits.size();
    if ( hStart < hLen ) {
        intersection* hit = curHits[ hStart ];
        if ( hit->isOutside() ) {
            double curDist = hit->getDistance();
            if ( notSet || curDist < dist ) {
                bestII = ii;
                bestJJ = hStart;
                bestHit = hit;
                dist = curDist - 0.000001;
                notSet = false;
            }
        }
    }
}
if ( bestHit == 0 ) {
    break;
} else {
    hitStarts[ bestII ] = bestJJ + 1;
}
```

Assume we're done unless something is outside.

⟨*Inter make sure hit inside all other objects*⟩≡

```
bool insideAll = true;
for ( unsigned int ii=0; insideAll && ii < len; ++ii ) {
    if ( ii == bestII ) {
        continue;
    }
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    unsigned int hLen = curHits.size();

    for ( unsigned int jj=hitStarts[ ii ]; jj < hLen; ++jj ) {
        intersection* hit = curHits[ jj ];
        double curDist = hit->getDistance();
        if ( curDist <= dist ) {
            ++hitStarts[ ii ];
        } else if ( hit->isOutside() ) {
            insideAll = false;
            break;
        } else {
            break;
        }
    }
    if ( hitStarts[ ii ] == hLen && hLen > 0 && curHits[ hLen-1  ] != 0
    && curHits[ hLen-1  ]->isInside() ) {
        insideAll = false;
    }
}
if ( !insideAll ) {
    bestHit = 0;
} else {
    done = true;
}
```

⟨*Inter find closest inside hit*⟩≡

```
bool notSet = true;
double dist = MAXFLOAT;
for ( unsigned int ii=0; ii < len; ++ii ) {
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    unsigned int hStart = hitStarts[ ii ];
    unsigned int hLen = curHits.size();
    if ( hStart < hLen ) {
        intersection* hit = curHits[ hStart ];
        if ( hit->isInside() ) {
            double curDist = hit->getDistance();
            if ( notSet || curDist < dist ) {
                bestII = ii;
                bestJJ = hStart;
                bestHit = hit;
                dist = curDist - 0.000001;
                notSet = false;
            }
        }
    }
}
```

⟨*Inter advance all outside folks*⟩≡

```
if ( bestHit != 0 ) {
    for ( unsigned int ii=0; ii < len; ++ii ) {
        if ( ii == bestII ) {
            continue;
        }
        std::vector< intersection* >& curHits( hitLists[ ii ] );
        unsigned int hLen = curHits.size();

        for ( unsigned int jj=hitStarts[ ii ]; jj < hLen; ++jj ) {
            intersection* hit = curHits[ jj ];
            double curDist = hit->getDistance();
            if ( curDist <= dist ) {
                ++hitStarts[ ii ];
            }
        }
    }
}
```

⟨*Inter add hit*⟩≡
```
std::vector< intersection* >& curHits( hitLists[ bestII ] );
curHits[ bestJJ ] = 0;
hitStarts[ bestII ] = bestJJ+1;
bestHit->reorient( this->scale, this->orientation, this->center );
bestHit->setDistance( ( bestHit->getPosition() - start ).magnitude() );
if ( this->gotColor ) {
    bestHit->recolor( this->chroma );
}
hits.insert( hits.end(), bestHit );
```

⟨*Inter clean up hits*⟩≡
```
for ( unsigned int ii=0; ii < len; ++ii ) {
    std::vector< intersection* >& curHits( hitLists[ ii ] );
    unsigned int hLen = curHits.size();
    for ( unsigned int jj=0; jj < hLen; ++jj ) {
        delete curHits[ jj ];
    }
    curHits.clear();
}
```

18.5 Checking If A Point Is Inside A Intersection

⟨*Inter Method Declarations*⟩+≡
```
virtual bool inside( const mathvec< double >& start ) const;
virtual bool _inside( const mathvec< double >& rp ) const;
```

⟨*Inter Methods*⟩+≡
```
bool
csginter::inside( const mathvec< double >& start ) const
{
    mathvec< double > rp = start - this->center;
    rp.antirotate( this->orientation );
    rp /= this->scale;

    return this->_inside( rp );
}

bool
csginter::_inside( const mathvec< double >& rp ) const
{
    unsigned int len = this->objs.size();
    bool in = len > 0;

    for ( unsigned int ii=0; in && ii < len; ++ii ) {
        in = this->objs[ ii ]->inside( rp );
    }

    return in;
}
```

18.6 Loading A Inter

The following method allows one to read in an inter from a stream.

⟨*Inter Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, csginter& cc );
```

⟨*Inter Methods*⟩+≡

```
std::istream&
operator >> ( std::istream& stream, csginter& cc )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "object" ) {
            object* nn = object::read( stream, cc.myScene );
            if ( nn != 0 ) {
                cc.objs.insert( cc.objs.end(), nn );
            } else {
                std::cerr << "INTER failed to get an object!" << std::endl;
            }
        } else if ( tok == "base" ) {
            stream >> (object&)cc >> semicolon;
        } else {
            in.ignore( "INTER", tok.c_str() );
        }
    }

    return stream;
}
```

18.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Inter Class Declaration*⟩≡
```
class csginter : public object {
protected:
    ⟨Inter Member Variables⟩
public:
    ⟨Inter Method Declarations⟩
public:
    static const char* id;
};
```

18.8 Source Files

The following sections assemble the source files for the `inter` class from the chunks above.

18.8.1 inter.h

The header file the `inter` class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header file for the `object` base class and then incorporates the class defined in 18.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*inter.h*⟩≡
```
#ifndef _NKLEIN_INTER_H_
#define _NKLEIN_INTER_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "object.h"

    ⟨Inter Class Declaration⟩

#endif /*_NKLEIN_INTER_H_*/
```

18.8.2 inter.cc

Herein, the implementations of the methods of the inter class are included.

⟨*inter.cc*⟩≡

```
#include <string>
#include <sstream>
#include <math.h>
#ifndef MAXFLOAT
#include <values.h>
#endif
#include "input.h"
#include "inter.h"

const char* csginter::id = "NKVERSION: csginter( 2.1.2007.05.17 )";

⟨Inter Methods⟩
```

Chapter 19

Extrusion

The extrusion is a CSG (Constructive Solid Geometry) primitive. If this object is in an n-dimensional universe and specifies an internal dimension k and contains a sub-object, then this object represents the extrusion of the k-dimensional sub-object into n dimensions. The extrusion contains a single sub-object. And, it can specify parameters from the base object class.

⟨*csg extrusion bnf*⟩≡

```
csg_extrusion_params ::=
        'object' object_type_and_params ';'
    |   'base' '{' object_parameters '}' ';'
    ;
```

The following snippets are used so that the object's factory method can create instance of this class.

⟨*Derived Object Includes*⟩+≡

```
#include "extrusion.h"
```

⟨*Derived Object Read Cases*⟩+≡

```
if ( objectType == "extrusion" ) {
    extrusion* obj = new extrusion( ss );
    stream >> *obj >> semicolon;
    return obj;
} else
```

The `extrusion` class is a subclass of the `object` class from 9. It contains a number of dimensions for its sub-object and a sub-object.

⟨*Extrusion Member Variables*⟩≡

```
unsigned int subDimensions;
object* obj;
```

⟨*Extrusion Member Initializations*⟩≡
```
, subDimensions( 0 ), obj( 0 )
```

19.1 Constructor

The default constructor does nothing.

⟨*Extrusion Method Declarations*⟩≡
```
extrusion( const scene* ss );
```

⟨*Extrusion Methods*⟩≡
```
extrusion::extrusion( const scene* ss )
    : object( ss ) ⟨Extrusion Member Initializations⟩
{
}
```

19.2 Destructor

This destructor has to release the sub-object

⟨*Extrusion Method Declarations*⟩+≡
```
virtual ~extrusion( void );
```

⟨*Extrusion Methods*⟩+≡
```
extrusion::~extrusion( void ) {
    delete this->obj;
};
```

19.3 Setting The Number of Dimensions

This method has to set the number of dimensions on the sub-object and the base object class.

⟨*Extrusion Method Declarations*⟩+≡
```
virtual void setDimensions( unsigned int dd );
```

⟨*Extrusion Methods*⟩+≡
```
void
extrusion::setDimensions( unsigned int dd )
{
    if ( this->subDimensions == 0 || this->subDimensions > dd ) {
        this->subDimensions = dd;
    }
    if ( this->obj != 0 ) {
        this->obj->setDimensions( this->subDimensions );
    }
    this->object::setDimensions( dd );
}
```

19.4 Intersection A Ray And A Extrusion

This first intersection method retrieves only the closest intersection to the ray's origin. The ray is specified by its direction and start.

⟨*Extrusion Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const;
```

⟨*Extrusion Methods*⟩+≡
```
bool
extrusion::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    intersection& hit
) const {
    if ( this->obj == 0 ) {
        return false;
    }
    ⟨Extrusion intersect common parts⟩
    bool ret = false;
    if ( ( rd ^ rd ) > 0.000001 ) {
        ret = this->obj->intersect( rd, rp, hit );
        if ( ret ) {
            ⟨Extrusion update hit⟩
        }
    }
    return ret;
}
```

⟨*Extrusion intersect common parts*⟩≡
```
mathvec< double > rd = direction;
rd.antirotate( this->orientation );
rd /= this->scale;

mathvec< double > fullrd = rd;
rd.resize( this->subDimensions, 0.0 );

fullrd.normalize();
rd.normalize();

mathvec< double > rp = start - this->center;
rp.antirotate( this->orientation );
rp /= this->scale;

mathvec< double > fullrp = rp;
rp.resize( this->subDimensions, 0.0 );
```

⟨*Extrusion update hit*⟩≡
```
mathvec< double > hitPosition;
if ( hit.getDistance() < MAXFLOAT / 2.0 ) {
    hitPosition = fullrd;
    rd.resize( fullrd.size(), 0.0 );
    hitPosition *= hit.getDistance();
    double distFactor = ( rd ^ fullrd );
    if ( distFactor > 0.000001 ) {
        hitPosition /= distFactor;
    }
    hitPosition += fullrp;
    hit.setPosition( hitPosition );
} else {
    hitPosition = hit.getPosition();
    hitPosition.resize( this->center.size(), 0.0 );
    hit.setPosition( hitPosition );
}

hit.setDirection( fullrd );

mathvec< double > hitNormal = hit.getNormal();
hitNormal.resize( this->center.size(), 0.0 );
hit.setNormal( hitNormal );

hit.reorient( this->scale, this->orientation, this->center );
hit.setDistance( ( hitPosition - start ).magnitude() );

if ( this->gotColor ) {
    hit.recolor( this->chroma );
}
```

This second intersection method retrieves all intersections between this extrusion and the ray. The ray is specified by its direction and start. The array of hits is sorted in order of increasing distance from the start of the ray.

⟨*Extrusion Method Declarations*⟩+≡
```
virtual bool intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const;
```

⟨*Extrusion Methods*⟩+≡
```
bool
extrusion::intersect(
    const mathvec< double >& direction,
    const mathvec< double >& start,
    std::vector< intersection* >& hits
) const {
    if ( this->obj == 0 ) {
        return false;
    }
    ⟨Extrusion intersect common parts⟩
    hits.clear();
    bool ret;

    if ( ( rd ^ rd ) > 0.000001 ) {
        ret = this->obj->intersect( rd, rp, hits );
        if ( ret ) {
            for ( unsigned int ii=0; ii < hits.size(); ++ii ) {
                intersection& hit( *hits[ ii ] );
                ⟨Extrusion update hit⟩
            }
        }
    } else {
        ret = this->obj->inside( rp );
        if ( ret ) {
            intersection* nn = new intersection();
            this->forgeHit( direction, start, *nn );
            hits.insert( hits.end(), nn );
        }
    }

    return ret;
}
```

19.5 Checking If A Point Is Inside An Extrusion

⟨*Extrusion Method Declarations*⟩+≡
```
virtual bool inside( const mathvec< double >& start ) const;
virtual bool _inside( const mathvec< double >& rp ) const;
```

⟨*Extrusion Methods*⟩+≡
```
bool
extrusion::inside( const mathvec< double >& start ) const
{
    if ( this->obj == 0 ) {
        return false;
    }
    mathvec< double > rp = start - this->center;
    rp.antirotate( this->orientation );
    rp /= this->scale;

    rp.resize( this->subDimensions, 0.0 );

    return this->_inside( rp );
}

bool
extrusion::_inside( const mathvec< double >& rp ) const
{
    if ( this->obj == 0 ) {
        return false;
    }
    return this->obj->inside( rp );
}
```

19.6 Loading A Extrusion

The following method allows one to read in an extrusion from a stream.

⟨*Extrusion Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, extrusion& cc );
```

⟨*Extrusion Methods*⟩+≡

```
std::istream&
operator >> ( std::istream& stream, extrusion& ee )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "dimensions" ) {
            stream >> ee.subDimensions >> semicolon;
        } else if ( tok == "object" ) {
            object* nn = object::read( stream, ee.myScene );
            if ( nn != 0 ) {
                delete ee.obj;
                ee.obj = nn;
            } else {
                std::cerr << "EXTRUSION failed to get an object!" << std::endl;
            }
        } else if ( tok == "base" ) {
            stream >> (object&)ee >> semicolon;
        } else {
            in.ignore( "EXTRUSION", tok.c_str() );
        }
    }

    return stream;
}
```

19.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Extrusion Class Declaration*⟩≡
```
class extrusion : public object {
protected:
    ⟨Extrusion Member Variables⟩
public:
    ⟨Extrusion Method Declarations⟩
public:
    static const char* id;
};
```

19.8 Source Files

The following sections assemble the source files for the `extrusion` class from the chunks above.

19.8.1 `extrusion.h`

The header file the `extrusion` class includes the `iostream`, `string`, and `map` headers from the standard C++ libraries and the header file for the `object` base class and then incorporates the class defined in 19.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*extrusion.h*⟩≡
```
#ifndef _NKLEIN_EXTRUSION_H_
#define _NKLEIN_EXTRUSION_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "object.h"

    ⟨Extrusion Class Declaration⟩

#endif /*_NKLEIN_EXTRUSION_H_*/
```

19.8.2 extrusion.cc

Herein, the implementations of the methods of the `extrusion` class are included.

⟨*extrusion.cc*⟩≡

```
#include <string>
#include <sstream>
#include <math.h>
#ifndef MAXFLOAT
#include <values.h>
#endif
#include "input.h"
#include "extrusion.h"

const char* extrusion::id = "NKVERSION: extrusion( 2.1.2007.05.17 )";

⟨Extrusion Methods⟩
```

Chapter 20

Image

Currently, there are three different image types supported. The `ppm` type outputs a Portable Pixel Map image. The `png` type outputs a Portable Network Graphics image. The `accum` type is a meta-image type. It contains another image instance which actually outputs the results. The `accum` type accumulates pixels across various frames and outputs an image where each pixel is a function of its values across all of the frames.

⟨*image bnf*⟩≡

```
image_type_and_parameters ::=
        'ppm'         '{' ppm_parameters '}' ';'
    |   'png'         '{' png_parameters '}' ';'
    |   'accumulator' '{' accum_parameters '}' ';'
    |   'accum'       '{' accum_parameters '}' ';'
    ;
```

The `ppm` images are defined in §21. The `png` images are defined in §22. The `accum` images are defined in §23.

Both the `ppm` and the `png` include a `base` parameter which is responible for tracking the things common to all image classes. At the moment, the only thing common across image classes is the basename attribute which is used to create filenames for frames.

⟨*image bnf*⟩+≡

```
image_base ::=
        image_name
    ;

image_name ::=
        'basename' string ';'
    |   'filename' string ';'
    ;
```

The `image` class is an abstract base-class for all of the image writers available. The base class keeps track of the width and height and number of the images to be written and the number of color channels for the image.

⟨*Image Member Variables*⟩≡
```
unsigned int width;
unsigned int height;
unsigned int depth;
unsigned int channelCount;
```

Additionally, this class tracks the basename for generated files and the number of files that are to be generated.

⟨*Image Member Variables*⟩+≡
```
std::string basename;
```

The image base class tracks how many output images have been created so that it can number them properly in the `getNumberedBasename()` method below.

⟨*Image Member Variables*⟩+≡
```
unsigned int currentImage;
```

20.1 Contructor

The `image` class has a minimal constructor which simply resets the current image count.

⟨*Image Method Declarations*⟩≡
```
image( void );
```

⟨*Image Methods*⟩≡
```
image::image( void ) : currentImage( 0 )
{
}
```

20.2 Destructor

The `image` class has a virtual destructor to ensure that the derived classes will be destructed properly. The destructor itself has nothing to do.

⟨*Image Method Declarations*⟩+≡
```
virtual ~image( void );
```

⟨*Image Methods*⟩+≡
```
image::~image( void )
{
}
```

20.3 Setting The Image Information

The meta-information about the image is the width and height of each image, the depth of the image stack, and the number of color channels. These are set by the `view` at the beginning of the `render()` method. The image basename, however, is set during the reading of the image.

⟨*Image Method Declarations*⟩+≡
```
virtual void setSize(
    unsigned int ww, unsigned int hh,
    unsigned int dd,
    unsigned int cc
);
virtual void setBasename( const std::string& bb );
```

⟨*Image Methods*⟩+≡
```
void
image::setSize(
    unsigned int ww, unsigned int hh,
    unsigned int dd,
    unsigned int cc
)
{
    this->width = ww;
    this->height = hh;
    this->depth = dd;
    this->channelCount = cc;
}
```

⟨*Image Methods*⟩+≡
```
void
image::setBasename( const std::string& bb )
{
    this->basename = bb;
}
```

20.4 Counting Output Images

The derived image classes will have to output numbered filenames. This method allows the code to appear in only one place.

⟨*Image Method Declarations*⟩+≡
```
virtual std::string getNumberedBasename( void );
```

If there is going to be more than one image, then the number is appended to the basename and returned.

⟨*Image Methods*⟩+≡
```
std::string
image::getNumberedBasename( void )
{
    std::ostringstream fname;
    fname << this->basename;
    if ( this->depth > 1 || this->currentImage > 1 ) {
        unsigned int digits
            = (unsigned int)( ::log10( (double)this->depth ) + 1.0000001 );
        fname << std::setw( digits ) << std::setfill( '0' );
        fname << this->currentImage++;
    }
    return fname.str();
}
```

20.5 Adding Pixels

The following methods are used by the `view` class in 2 to place pixels in the image.

⟨*Image Method Declarations*⟩+≡
```
virtual void add( const mathvec< double >& pixel ) = 0;
virtual void addBorderPixel( void ) = 0;
virtual void addBorderRow( void ) = 0;
```

20.6 Loading An Image

The following method is a factory method which allows the raytracer to read image instances in from a stream.

⟨*Image Method Declarations*⟩+≡
```
static image* read( std::istream& in );
```

Most of this method is actually incorporated from the derived classes.

⟨*Image Methods*⟩+≡

```
image*
image::read( std::istream& stream )
{
    input in( stream );

    if ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return 0;
        }

        std::string tok( token );

        ⟨Image input load particular type⟩
        {
            in.ignore( "IMAGE FACTORY", tok.c_str() );
        }
    }

    return 0;
}
```

The following method allows one to read in the details common to all images from an input stream.

⟨*Image Protected Method Declarations*⟩≡

```
virtual std::istream& get( std::istream& stream );
```

⟨*Image Methods*⟩+≡
```
std::istream&
image::get( std::istream& stream )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );

        if ( tok == "basename" || tok == "filename" ) {
            in.get( token, sizeof(token) );
            this->basename = token;
        } else {
            in.ignore( "IMAGE", tok.c_str() );
        }
    }

    return stream;
}
```

20.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Image Class Declaration*⟩≡
```
class image {
protected:
    ⟨Image Member Variables⟩
    ⟨Image Protected Method Declarations⟩
public:
    ⟨Image Method Declarations⟩
public:
    static const char* id;
};
```

20.8 Source Files

The following sections assemble the source files for the `image` class from the chunks above.

20.8.1 image.h

The header file the `image` class includes the `iostream` and `string` headers from the standard C++ libraries and the header file for the `mathvec<>` class and then incorporates the class defined in 20.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*image.h*⟩≡
```
#ifndef _NKLEIN_IMAGE_H_
#define _NKLEIN_IMAGE_H_ 1

#include <iostream>
#include <string>
#include "mathvec.h"

    ⟨Image Class Declaration⟩

#endif /*_NKLEIN_IMAGE_H_*/
```

20.8.2 image.cc

Herein, the implementations of the methods of the `image` class are included as well as the the declarations and implementations of the derived classes.

⟨*image.cc*⟩≡
```
#include <string>
#include <sstream>
#include <fstream>
#include <iomanip>
#include <math.h>
#include "input.h"
#include "image.h"

const char* image::id = "NKVERSION: image( 2.1.2007.05.17 )";

⟨Image Derived Classes⟩
⟨Image Methods⟩
```

Chapter 21

PPM Images

The Portable Pixmap Image (PPM) format is the simplest useful image format. The code in this section allows output to PPM format. This class inherits directly from the image class defined in §20.

There are no parameters specific to the PPM. The only parameters are those of the base class.

⟨*ppm bnf*⟩≡
```
    ppm_parameters ::=
            'common' '{' image_base '}' ';'
        ;
```

The class tracks the stream for the output file.

⟨*ImagePPM Data Members*⟩≡
```
std::ostream* out;
```

The class also keeps a whole row of border pixels for any time a whole row of them is emitted.

⟨*ImagePPM Data Members*⟩+≡
```
unsigned char* borderRow;
```

The class also tracks how many pixels have been emitted for the current frame.

⟨*ImagePPM Data Members*⟩+≡
```
unsigned int pixelCount;
```

21.1 Constructor

The constructor simply has to set the pointers defined above to null values.

⟨*ImagePPM Method Declarations*⟩≡

```
imagePPM( void ) : out( 0 ), borderRow( 0 ) {
};
```

21.2 Destructor

The destructor here has to release any memory allocated for the outgoing file stream and the border-pixel buffer.

⟨*ImagePPM Method Declarations*⟩+≡

```
virtual ~imagePPM( void ) {
    delete this->out;
    delete[] this->borderRow;
};
```

21.3 Setting The Image Information

This method overrides the one in the base class. It restricts the number of color channels to either three or one as those are the only values a PPM can use. After that, this method allocates a buffer it will use to output border pixels.

⟨*ImagePPM Method Declarations*⟩+≡

```
virtual void setSize(
    unsigned int ww, unsigned int hh,
    unsigned int dd,
    unsigned int cc
) {
    if ( cc > 3 ) {
        cc = 3;
    } else if ( cc < 3 ) {
        cc = 1;
    }

    this->image::setSize( ww, hh, dd, cc );

    unsigned int len = this->width * this->channelCount;

    delete[] this->borderRow;
    this->borderRow = new unsigned char[ len ];

    for ( unsigned int ii=0; ii < len; ++ii ) {
        this->borderRow[ ii ] = (unsigned char)0xFF;
    }

    this->pixelCount = 0;
};
```

21.4 Adding Pixels

The following methods are used by the `view` class in 2 to place pixels in the image. The first method here adds a single pixel value to the output image. If the pixel size is smaller than expected, the last value is repeated to fill any remaining output channels. On the other hand, if the pixel size is larger than expected, the average value is used for all channels.

⟨*ImagePPM Method Declarations*⟩+≡

```
virtual void add( const mathvec< double >& pixel ) {
    this->checkOutput();
    if ( pixel.size() <= this->channelCount ) {
        unsigned char lastVal = 0;
        for ( unsigned int ii=0; ii < pixel.size(); ++ii ) {
            int val = (int)( pixel[ ii ] * 255.0 + 0.5 );
            lastVal = (unsigned char)( val<0 ) ? 0 : ( val>255 ) ? 255 : val ;
            this->out->put( lastVal );
        }
        for ( unsigned int ii=pixel.size(); ii < this->channelCount; ++ii ) {
            this->out->put( lastVal );
        }
    } else {
        int total = 0;
        for ( unsigned int ii=0; ii < pixel.size(); ++ii ) {
            int val = (int)( pixel[ ii ] * 255.0 + 0.5 );
            total += ( val < 0 ) ? 0 : ( val > 255 ) ? 255 : val ;
        }
        unsigned char val = (unsigned char)( total / pixel.size() );
        for ( unsigned int ii=0; ii < this->channelCount; ++ii ) {
            this->out->put( val );
        }
    }
};
```

To emit a border pixel, this method simply copies a single pixel from the border buffer.

⟨*ImagePPM Method Declarations*⟩+≡

```
virtual void addBorderPixel( void ) {
    this->checkOutput();
    this->out->write( (const char*)borderRow, this->channelCount );
};
```

To emit a border row, this method copies a whole row from the border buffer.

⟨*ImagePPM Method Declarations*⟩+≡
```
virtual void addBorderRow( void ) {
    this->checkOutput( this->width );
    this->out->write( (const char*)borderRow, this->width*this->channelCount );
};
```

The output check decides if its time to output the next frame. The frame header is emitted if needed.

⟨*ImagePPM Protected Method Declarations*⟩≡
```
void checkOutput( unsigned int count = 1 ) {
    if ( this->pixelCount < count ) {
        delete this->out;
        this->out = 0;
        this->pixelCount = this->width * this->height - count;
    } else {
        this->pixelCount -= count;
    }
    if ( this->out == 0 ) {
        std::string fname = this->getNumberedBasename() + ".ppm";
        this->out = new std::ofstream( fname.c_str() );
        if ( this->channelCount == 1 ) {
            *this->out << "P5" << std::endl;
        } else {
            *this->out << "P6" << std::endl;
        }
        *this->out << this->width << " " << this->height << std::endl;
        *this->out << "255" << std::endl;
    }
};
```

21.5 Reading In A PPM

This segment of code gets included in the image read function to create an instance of this class if it is needed.

⟨*Image input load particular type*⟩≡
```
#ifdef USE_PPM
    if ( tok == "ppm" ) {
        image* nn = new imagePPM();
        nn->get( stream );
        return nn;
    } else
#endif
```

With a PPM image, the only information necessary is that information common to all images.

⟨*ImagePPM Method Declarations*⟩+≡

```
virtual std::istream& get( std::istream& stream )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "common" ) {
            this->image::get( stream );
            stream >> semicolon;
        } else {
            in.ignore( "PPM", tok.c_str() );
        }
    }

    return stream;
};
```

21.6 The Class Definition

This class inherits directly from the image class defined in §20. It includes all of the methods declared above. And, it includes some version identification information.

⟨*Image Derived Classes*⟩≡

```
#ifdef USE_PPM
class imagePPM : public image {
public:
    ⟨ImagePPM Method Declarations⟩
protected:
    ⟨ImagePPM Protected Method Declarations⟩
    ⟨ImagePPM Data Members⟩
public:
    static const char* id;
};

const char* imagePPM::id = "NKVERSION: imagePPM( 2.1.2007.05.17 )";
#endif
```

Chapter 22

PNG Images

This raytracer also supports output in the Portable Network Graphics (PNG) format. The code in this section allows output to PNG format. This class inherits directly from the image class defined in §20.

There are no parameters specific to the PNG. The only parameters are those of the base class.

⟨*png bnf*⟩≡
```
    png_parameters ::=
            'common' '{' image_base '}' ';'
        ;
```

The class tracks the stream for the output file.

⟨*ImagePNG Data Members*⟩≡
```
std::ostream* out;
```

The class also keeps a whole row of border pixels for any time a whole row of them is emitted. It also keeps track of several other buffers for output and the current location within the pixel buffer.

⟨*ImagePNG Data Members*⟩+≡
```
unsigned char* borderRow;
unsigned char* tmpBuffer;
png_bytep pixelBuffer;
unsigned int rowSpot;
```

The class also tracks some state from the PNG library.

⟨*ImagePNG Data Members*⟩+≡
```
png_structp pngPtr;
png_infop infoPtr;
```

The class also tracks how many pixels have been emitted for the current frame.

⟨*ImagePNG Data Members*⟩+≡
```
unsigned int pixelCount;
```

22.1 Constructor

The constructor simply has to set the pointers defined above to null values.

⟨*ImagePNG Method Declarations*⟩≡
```
imagePNG( void ) : out( 0 ), borderRow( 0 ), tmpBuffer( 0 ), pixelBuffer( 0 ),
    pngPtr( 0 ), infoPtr( 0 )
{
};
```

22.2 Destructor

The destructor here has to release any memory allocated for the outgoing file stream and the border-pixel buffer.

⟨*ImagePNG Method Declarations*⟩+≡
```
virtual ~imagePNG( void ) {
    if ( this->pngPtr != 0 ) {
        ::png_write_end( this->pngPtr, this->infoPtr );
        ::png_destroy_write_struct( &this->pngPtr, &this->infoPtr );
    }
    delete this->out;
    delete[] this->pixelBuffer;
    delete[] this->borderRow;
    delete[] this->tmpBuffer;
};
```

22.3 Setting The Image Information

This method overrides the one in the base class. It restricts the number of color channels to either three or one as those are the only values a PNG can use. After that, this method allocates a buffer it will use to output border pixels.

⟨*ImagePNG Method Declarations*⟩+≡

```
virtual void setSize(
    unsigned int ww, unsigned int hh,
    unsigned int dd,
    unsigned int cc
) {
    if ( cc > 4 ) {
        cc = 4;
    }

    this->image::setSize( ww, hh, dd, cc );

    unsigned int len = this->width * this->channelCount;

    delete[] this->borderRow;
    this->borderRow = new unsigned char[ len ];

    delete[] this->tmpBuffer;
    this->tmpBuffer = new unsigned char[ len ];

    delete[] this->pixelBuffer;
    this->pixelBuffer = new png_byte[ len ];

    for ( unsigned int ii=0; ii < len; ++ii ) {
        this->borderRow[ ii ] = (unsigned char)0xFF;
    }

    this->pixelCount = 0;
};
```

22.4 Adding Pixels

The following methods are used by the `view` class in 2 to place pixels in the image. The first method here adds a single pixel value to the output image. If the pixel size is smaller than expected, the last value is repeated to fill any remaining output channels. On the other hand, if the pixel size is larger than expected, the average value is used for all channels.

⟨*ImagePNG Method Declarations*⟩+≡
```
virtual void add( const mathvec< double >& pixel ) {
    unsigned int curSample = 0;
    if ( pixel.size() <= this->channelCount ) {
        unsigned char lastVal = 0;
        for ( unsigned int ii=0; ii < pixel.size(); ++ii ) {
            int val = (int)( pixel[ ii ] * 255.0 + 0.5 );
            lastVal = (unsigned char)( val<0 ) ? 0 : ( val>255 ) ? 255 : val ;
            this->tmpBuffer[ curSample++ ] = lastVal;
        }
        for ( unsigned int ii=pixel.size(); ii < this->channelCount; ++ii ) {
            this->tmpBuffer[ curSample++ ] = lastVal;
        }
    } else {
        int total = 0;
        for ( unsigned int ii=0; ii < pixel.size(); ++ii ) {
            int val = (int)( pixel[ ii ] * 255.0 + 0.5 );
            total += ( val < 0 ) ? 0 : ( val > 255 ) ? 255 : val ;
        }
        unsigned char val = (unsigned char)( total / pixel.size() );
        for ( unsigned int ii=0; ii < this->channelCount; ++ii ) {
            this->tmpBuffer[ curSample++ ] = val;
        }
    }
    this->write( this->tmpBuffer, 1 );
};
```

To emit a border pixel, this method simply copies a single pixel from the border buffer.

⟨*ImagePNG Method Declarations*⟩+≡
```
virtual void addBorderPixel( void ) {
    this->write( this->borderRow, 1 );
};
```

To emit a border row, this method copies a whole row from the border buffer.

⟨*ImagePNG Method Declarations*⟩+≡

```
virtual void addBorderRow( void ) {
    this->write( this->borderRow, this->width );
};
```

This method is used to emit pixels. There is a fair amount of code required to emit a PNG header.

⟨*ImagePNG Protected Method Declarations*⟩≡

```
void write( const unsigned char* buffer, unsigned int count ) {
    if ( this->pixelCount < count ) {
        if ( this->pngPtr != 0 ) {
            ::png_write_end( this->pngPtr, this->infoPtr );
            ::png_destroy_write_struct( &this->pngPtr, &this->infoPtr );
        }
        delete this->out;
        this->out = 0;
        this->pixelCount = this->width * this->height - count;
    } else {
        this->pixelCount -= count;
    }
    if ( this->out == 0 ) {
        std::string fname = this->getNumberedBasename() + ".png";
        this->out = new std::ofstream( fname.c_str() );

        this->pngPtr = ::png_create_write_struct(
            PNG_LIBPNG_VER_STRING, 0, 0, 0
        );
        if ( this->pngPtr == 0 ) {
            std::cerr << "Unable to create png write structure" << std::endl;
            return;
        }

        this->infoPtr = ::png_create_info_struct( this->pngPtr );
        if ( this->infoPtr == 0 ) {
            std::cerr << "Unable to create png info structure" << std::endl;
            ::png_destroy_write_struct( &this->pngPtr, 0 );
            return;
        }

        if ( ::setjmp( this->pngPtr->jmpbuf ) ) {
            std::cerr << "Something wrong during write png" << std::endl;
            ::png_destroy_write_struct( &this->pngPtr, &this->infoPtr );
        }

        ::png_set_write_fn(
            this->pngPtr, this->out, imagePNG::writeData, imagePNG::flushData
        );
        ::png_set_compression_level( this->pngPtr, Z_BEST_COMPRESSION );

        static int colorTypes[] = {
            0,
```

```
            PNG_COLOR_TYPE_GRAY, PNG_COLOR_TYPE_GRAY_ALPHA,
            PNG_COLOR_TYPE_RGB, PNG_COLOR_TYPE_RGB_ALPHA
        };

        ::png_set_IHDR(
            this->pngPtr, this->infoPtr, this->width, this->height, 8,
            colorTypes[ this->channelCount ], PNG_INTERLACE_NONE,
            PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT
        );

        ::png_write_info( this->pngPtr, this->infoPtr );

        this->rowSpot = 0;
    }

    count *= this->channelCount;
    for ( unsigned int ii=0; ii < count; ++ii ) {
        this->pixelBuffer[ rowSpot++ ] = buffer[ ii ];
    }

    if ( rowSpot >= this->width * this->channelCount ) {
        ::png_write_row( this->pngPtr, this->pixelBuffer );
        this->rowSpot = 0;
    }
};
```

These methods are used to allow the PNG library to write to the output stream.

⟨*ImagePNG Protected Method Declarations*⟩+≡

```
static void writeData( png_structp ctx, png_bytep buffer, png_size_t size )
{
    std::ofstream* dst = (std::ofstream*)::png_get_io_ptr( ctx );
    dst->write( (const char*)buffer, size );
}
static void flushData( png_structp ctx )
{
    std::ofstream* dst = (std::ofstream*)::png_get_io_ptr( ctx );
    dst->flush();
}
```

22.5 Reading In A PNG

This segment of code gets included in the image read function to create an instance of this class if it is needed.

⟨*Image input load particular type*⟩+≡

```
#ifdef USE_PNG
    if ( tok == "png" ) {
        image* nn = new imagePNG();
        nn->get( stream );
        return nn;
    } else
#endif
```

With a PNG image, the only information necessary is that information common to all images.

⟨*ImagePNG Method Declarations*⟩+≡

```
virtual std::istream& get( std::istream& stream )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "common" ) {
            this->image::get( stream );
            stream >> semicolon;
        } else {
            in.ignore( "PNG", tok.c_str() );
        }
    }

    return stream;
};
```

22.6 The Class Definition

This class inherits directly from the image class defined in §20. It includes all of the methods declared above. And, it includes some version identification information.

⟨*Image Derived Classes*⟩+≡

```
#ifdef USE_PNG
#include <png.h>

class imagePNG : public image {
public:
    ⟨ImagePNG Method Declarations⟩
protected:
    ⟨ImagePNG Protected Method Declarations⟩
    ⟨ImagePNG Data Members⟩
public:
    static const char* id;
};

const char* imagePNG::id = "NKVERSION: imagePNG( 2.1.2007.05.17 )";
#endif
```

Chapter 23

Accum Images

Unlike the other classes derived from the image class, this class does not actually write an output image. This class accumulates pixels from multiple frames and emits them to a different image instance. This class inherits directly from the image class defined in §20.

The accumulation method and the contained image are the parameters available for the accumulation image. The default method is averaging.

⟨*accum bnf*⟩≡
```
accum_parameters ::=
          'method'  accum_method_value ';'
      |   'image' image_type_and_parameters ';'
      ;

accum_method_value ::=
          'minimum'
      |   'min'
      |   'maximum'
      |   'max'
      |   'average'
      |   'avg'
      |   'first'
      ;
```

There are four basic modes this accumulator can use.

⟨*ImageAccum Method Types*⟩≡
```
typedef enum {
    MINIMUM, MAXIMUM, AVERAGE, FIRST
} MethodType;
```

This class tracks the contained image instance, a full frame of the image, a little information about the current offset into the image, the operating mode, the current pixel, and a full-on pixel.

⟨*ImageAccum Data Members*⟩≡
```
image* sub;
double* accumulator;
unsigned int curIndex;
unsigned int pixelsRemaining;
MethodType method;
mathvec< double > curPixel;
mathvec< double > allOnes;
```

23.1 Constructor

The constructor simply has to set the pointers defined above to null values.

⟨*ImageAccum Method Declarations*⟩≡
```
imageAccum( void ) : sub( 0 ), accumulator( 0 ) {
};
```

23.2 Destructor

The destructor here has to release any memory allocated for the outgoing file stream and the border-pixel buffer.

⟨*ImageAccum Method Declarations*⟩+≡
```
virtual ~imageAccum( void ) {
    delete this->sub;
    delete[] this->accumulator;
};
```

23.3 Setting The Image Information

This method overrides the one in the base class. It restricts the number of color channels to either three or one as those are the only values a Accum can use. After that, this method allocates a buffer it will use to output border pixels.

⟨*ImageAccum Method Declarations*⟩+≡

```
virtual void setSize(
    unsigned int ww, unsigned int hh,
    unsigned int dd,
    unsigned int cc
) {
    this->image::setSize( ww, hh, dd, cc );
    this->sub->setSize( ww, hh, 1, cc );
    this->pixelsRemaining = ww * hh * dd;

    unsigned int len = ww * hh * cc;

    this->accumulator = new double[ len ];
    this->curIndex = 0;

    double initialValue = 0.0;
    if ( this->method == MINIMUM ) {
        initialValue = 1.0;
    }

    for ( unsigned int ii=0; ii < len; ++ii ) {
        this->accumulator[ ii ] = initialValue;
    }

    this->curPixel.resize( cc, 0.0 );
    this->allOnes.resize( cc, 1.0 );
};
```

23.4 Adding Pixels

The following methods are used by the `view` class in 2 to place pixels in the image. The first method here adds a single pixel value to the output image. If the pixel size is smaller than expected, the last value is repeated to fill any remaining output channels. On the other hand, if the pixel size is larger than expected, the average value is used for all channels.

⟨*ImageAccum Method Declarations*⟩+≡

```
virtual void add( const mathvec< double >& pixel ) {
    if ( pixel.size() <= this->channelCount ) {
        double lastVal = 0.0;
        for ( unsigned int ii=0; ii < pixel.size(); ++ii ) {
            lastVal = pixel[ ii ];
            this->curPixel[ ii ] = lastVal;
        }
        for ( unsigned int ii=pixel.size(); ii < this->channelCount; ++ii ) {
            this->curPixel[ ii ] = lastVal;
        }
    } else {
        double total = 0.0;
        for ( unsigned int ii=0; ii < pixel.size(); ++ii ) {
            total += pixel[ ii ];
        }
        double val = ( total / (double)pixel.size() );
        for ( unsigned int ii=0; ii < this->channelCount; ++ii ) {
            this->curPixel[ ii ] = val;
        }
    }
    this->add();
};
```

To emit a border pixel, this method simply copies a single pixel from the border buffer.

⟨*ImageAccum Method Declarations*⟩+≡

```
virtual void addBorderPixel( void ) {
    this->curPixel = this->allOnes;
    this->add();
};
```

To emit a border row, this method copies a whole row from the border buffer.

⟨*ImageAccum Method Declarations*⟩+≡

```
virtual void addBorderRow( void ) {
    this->curPixel = this->allOnes;
    for ( unsigned int ii=0; ii < this->width; ++ii ) {
        this->add();
    }
};
```

This method is used to add the current pixel to the accumulator.

⟨*ImageAccum Protected Method Declarations*⟩≡

```
virtual void add( void ) {
    double* spot = &this->accumulator[ this->curIndex ];
    this->curIndex += this->channelCount;

    switch ( this->method ) {
    case MINIMUM: {
        double curTotal = ( this->curPixel ^ this->allOnes );
        double spotTotal = 0.0;
        ⟨ImageAccum get spot total⟩

        if ( curTotal < spotTotal ) {
            ⟨ImageAccum copy current pixel to spot⟩
        }
    } break;
    case MAXIMUM: {
        double curTotal = ( this->curPixel ^ this->allOnes );
        double spotTotal = 0.0;
        ⟨ImageAccum get spot total⟩

        if ( curTotal > spotTotal ) {
            ⟨ImageAccum copy current pixel to spot⟩
        }
    } break;
    case AVERAGE: {
        for ( unsigned int ii=0; ii < this->channelCount; ++ii ) {
            spot[ ii ] += this->curPixel[ ii ] / (double)this->depth;
        }
    } break;
    case FIRST: {
        double spotTotal = 0.0;
        ⟨ImageAccum get spot total⟩

        if ( spotTotal < 0.000001 ) {
            ⟨ImageAccum copy current pixel to spot⟩
        }
    } break;
    };

    unsigned int len = this->width * this->height * this->channelCount;

    if ( this->curIndex >= len ) {
        this->curIndex = 0;
    }
```

```
    if ( --this->pixelsRemaining == 0 ) {
        mathvec< double > pixel( this->channelCount );
        for ( unsigned int ii=0; ii < len; /*done*/ ) {
            for ( unsigned int kk=0; kk < this->channelCount; ++kk ) {
                pixel[ kk ] = this->accumulator[ ii++ ];
            }
            this->sub->add( pixel );
        }
    }
};
```

⟨*ImageAccum get spot total*⟩≡

```
for ( unsigned int ii=0; ii < this->channelCount; ++ii ) {
    spotTotal += spot[ ii ];
}
```

⟨*ImageAccum copy current pixel to spot*⟩≡

```
for ( unsigned int ii=0; ii < this->channelCount; ++ii ) {
    spot[ ii ] = this->curPixel[ ii ];
}
```

23.5 Reading In A Accum

This segment of code gets included in the image read function to create an instance of this class if it is needed.

⟨*Image input load particular type*⟩+≡

```
if ( tok == "accumulator" || tok == "accum" ) {
    image* nn = new imageAccum();
    nn->get( stream );
    return nn;
} else
```

With an accum image, the method and the subimage are the only available parameters.

⟨*ImageAccum Method Declarations*⟩+≡

```
virtual std::istream& get( std::istream& stream )
{
    input in( stream );

    this->method = AVERAGE;

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "method" ) {
            if ( in.get( token, sizeof(token) ) == 0 ) {
                return stream;
            }
            std::string meth( token );
            if ( meth == "minimum" || meth == "min" ) {
                this->method = MINIMUM;
            } else if ( meth == "maximum" || meth == "max" ) {
                this->method = MAXIMUM;
            } else if ( meth == "average" || meth == "avg" ) {
                this->method = AVERAGE;
            } else if ( meth == "first" ) {
                this->method = FIRST;
            }
        } else if ( tok == "image" ) {
            this->sub = this->image::read( stream );
            stream >> semicolon;
        } else {
            in.ignore( "ACCUM", tok.c_str() );
        }
    }

    return stream;
};
```

23.6 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Image Derived Classes*⟩+≡

```
class imageAccum : public image {
protected:
    ⟨ImageAccum Method Types⟩
public:
    ⟨ImageAccum Method Declarations⟩
protected:
    ⟨ImageAccum Protected Method Declarations⟩
    ⟨ImageAccum Data Members⟩
public:
    static const char* id;
};

const char* imageAccum::id = "NKVERSION: imageAccum( 2.1.2007.05.17 )";
```

Chapter 24

Portal

Most views will contain a viewport. That view is an instance of the color class defined in §25. For that color to actually represent a view into a universe, the color must contain a portal.

The portal refers to a universe by name. Additionally, it defines a point within that universe to be its center, it defines an orientation for the view into that universe, and it provides a scaling for the universe.

⟨*portal bnf*⟩≡
```
    portal_parameters ::=
            portal_parameters portal_param
        |   /* empty */
        ;

    portal_param ::=
            'scale' real_vector ';'
        |   'orientation' real_matrix ';'
        |   'center' real_vector ';'
        |   'universe' universe_name ';'
        ;
```

The portal, if it is to be useful, must refer to a universe by name. If the scale is omitted, it is assumed to be a scaling factor of one in every dimension. If the orientation is omitted, it is assumed to be the identity matrix. If the center is omitted, it is assumed to be the zero vector.

The portal class contains a pointer to the scene to which it belongs.

⟨*Portal Member Variables*⟩≡
```
const scene* myScene;
```

The portal class contains a pointer the universe into which it peers. But, until all of the universes have been loaded, it has to satisfy itself with just the name of the universe.

⟨*Portal Member Variables*⟩+≡
```
const universe* univ;
std::string universeName;
```

It contains a place to store its output pixel once evaluated.

⟨*Portal Member Variables*⟩+≡
```
mathvec< double > pixel;
```

Additionally, it contains places to track the scale, orientation, and center.

⟨*Portal Member Variables*⟩+≡
```
mathvec< double > scale;
basevec< mathvec< double > > orientation;
mathvec< double > center;
```

24.1 Constructor

The constructor for the portal class merely caches the scene pointer it is given and zeros the universe pointer.

⟨*Portal Method Declarations*⟩≡
```
portal( const scene* ss );
```

⟨*Portal Methods*⟩≡
```
portal::portal( const scene* ss ) : myScene( ss ), univ( 0 )
{
}
```

24.2 Destructor

The destructor for the portal does nothing.

⟨*Portal Method Declarations*⟩+≡
```
~portal( void );
```

⟨*Portal Methods*⟩+≡
```
portal::~portal( void )
{
}
```

24.3 Retrieving A Reference To Its Pixel Value

The expression subclass defined in §33 can be used to refer to a pixel value traced through this portal. As such, it needs to retrieve a reference to that pixel value.

⟨*Portal Method Declarations*⟩+≡

```
const mathvec< double >&  getPixel( void ) const {
    return this->pixel;
};
```

Note: for some complex scenes (or maybe even some not-so complex scenes) running multi-threaded could cause this buffer to be overwritten after it is calculated but before it is used. At the moment, I am not making any attempt to lock out this possibility.

24.4 Setting Up The Number Of Dimensions

The following method is invoked to bring the portal to the number of dimensions required by the enclosing extrusion, universe, or view. This is done after the portal is read and before any traces are performed so that once we get to tracing, all of the vectors and matrixen should be the correct size.

⟨*Portal Method Declarations*⟩+≡

```
void setDimensions( unsigned int dd );
```

This method resizes the scaling vector padding any new dimensions with ones. It orthogonalizes the orientation matrix. And, it resizes the center point padding any new dimensions with zeros.

⟨*Portal Methods*⟩+≡

```
void
portal::setDimensions( unsigned int dd )
{
    this->scale.resize( dd, 1.0 );
    ::orthogonalize( this->orientation, dd );
    this->center.resize( dd, 0.0 );
}
```

24.5 Casting A Ray Into The Portal

The following method allows one to cast a ray into a portal.

⟨*Portal Method Declarations*⟩+≡
```
bool evaluate(
    const mathvec< double >& position,
    const mathvec< double >& direction,
    double contribution, unsigned int height, unsigned int channelCount
);
```

This method ensures that the universe exists. Then, it reorients the incoming vectors, resizes those to the number of dimensions of the universe, clears out the pixel buffer, and sends the ray into the universe.

⟨*Portal Methods*⟩+≡
```
bool
portal::evaluate(
    const mathvec< double >& position,
    const mathvec< double >& direction,
    double contribution, unsigned int height, unsigned int channelCount
) {
    ⟨Portal make sure universe is loaded⟩
    ⟨Portal reorient incoming vectors⟩
    ⟨Portal resize incoming vectors⟩

    this->pixel.resize( channelCount, 0.0 );
    return this->univ->evaluate(
        rp, rd, contribution, height-1, this->pixel
    );
}
```

If the universe pointer is null but its name is not, then this method attempts to get a pointer to the universe from the enclosing scene. If it should fail, this method returns false.

⟨*Portal make sure universe is loaded*⟩≡
```
if ( this->univ == 0 && this->universeName != "" ) {
    this->univ = this->myScene->getUniverse( this->universeName );
}
if ( this->univ == 0 ) {
    return false;
}
```

To reorient the incoming vector, the direction is rotated, scaled, and then renormalized while the position is rotated, scaled, and recentered.

⟨*Portal reorient incoming vectors*⟩≡
```
mathvec< double > rd = direction;
rd.rotate( this->orientation );
rd /= this->scale;
rd.normalize();

mathvec< double > rp = position;
rp.rotate( this->orientation );
rp /= this->scale;
rp += this->center;
```

If the number of dimensions in the incoming vectors is not the same as that of the universe, the incoming vectors are rescaled. After this rescaling process, the direction vector may need to be renormalized. If the direction vector cannot be renormalized, this method bails.

⟨*Portal resize incoming vectors*⟩≡
```
unsigned int dims = this->univ->getDimensions();
if ( rp.size() != dims ) {
    rp.resize( dims, 0.0 );
    rd.resize( dims, 0.0 );

    double magdir = rd.magnitude();
    if ( magdir >= 0.000001 ) {
        rd /= magdir;
    } else {
        this->pixel *= 0.0;
        return false;
    }
}
```

24.6 Loading A Portal

The following method allows one to read a portal from an input stream.

⟨*Portal Method Declarations*⟩+≡
```
friend std::istream& operator >> ( std::istream& in, portal& pp );
```

There are four valid keywords within the portal: `scale`, `orientation`, `center`, and `universe`. For each of these, the appropriate parameter is read.

⟨*Portal Methods*⟩+≡

```
std::istream&
operator >> ( std::istream& stream, portal& pp )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "scale" ) {
            stream >> pp.scale >> semicolon;
        } else if ( tok == "orientation" ) {
            stream >> pp.orientation >> semicolon;
        } else if ( tok == "center" ) {
            stream >> pp.center >> semicolon;
        } else if ( tok == "universe" ) {
            if ( in.get( token, sizeof(token) ) == 0 ) {
                return stream;
            }
            pp.universeName = token;
        } else {
            in.ignore( "PORTAL", tok.c_str() );
        }
    }

    return stream;
}
```

24.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Portal Class Declaration*⟩≡
```
class portal {
protected:
    ⟨Portal Member Variables⟩
public:
    ⟨Portal Method Declarations⟩
public:
    static const char* id;
};
```

24.8 Source Files

The following sections assemble the source files for the portal class from the chunks above.

24.8.1 portal.h

The header file the portal class includes the `iostream` and `string` headers from the standard C++ libraries and the header file for the `mathvec<>` class. It predeclares the scene and universe classes, and then incorporates the class defined in §24.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*portal.h*⟩≡
```
#ifndef _NKLEIN_PORTAL_H_
#define _NKLEIN_PORTAL_H_ 1

#include <iostream>
#include <string>
#include "mathvec.h"

class scene;
class universe;

    ⟨Portal Class Declaration⟩

#endif /*_NKLEIN_PORTAL_H_*/
```

24.8.2 portal.cc

Herein, the implementations of the methods of the portal class are included. The implementation requires the `string` and `stringstream` classes from the standard C++ library, the header for the input subsystem, the header for the universe class, the header for the scene class, and the declaration of the portal class itself.

⟨*portal.cc*⟩≡
```
#include <string>
#include <sstream>
#include "input.h"
#include "universe.h"
#include "scene.h"
#include "portal.h"

const char* portal::id = "NKVERSION: portal( 2.1.2007.05.17 )";
```

⟨*Portal Methods*⟩

Chapter 25

Color

Most views will contain a viewport. That viewport is an instance of this color class. Additionally, all objects and all lights must specify a color. Technically, when one considers objects which contain other objects, this statement is not entirely precise. More precise would be that if one modeled the top-level object as a tree, there must be some color specified along every path to a leaf.

The color contains zero or more portal specifications. It contains an expression vector specifying the values of the pixel channels. It contains parameters defining the specularness, the reflectiveness, and the transparency of the object. It contains a parameter specifying the Phong shading attribute and a parameter defining the index of refraction of the object. It contains vectors to rescale and recenter the color coordinates and a matrix to reorient them.

⟨*color bnf*⟩≡

```
color_parameters ::=
        color_parameters color_param
    |   color_param
    ;

color_param ::=
        'portal' portal_name '{' portal_parameters '}' ';'
    |   color_channels
    |   'specularness' real ';'
    |   'transparency' real ';'
    |   'reflectiveness' real ';'
    |   'phong' real ';'
    |   color_ior
    |   'scale' real_vector ';'
    |   'orientation' real_matrix ';'
    |   'center' real_vector ';'
    ;

portal_name ::= string ;
```

```
color_channels ::=
        'channels' expr_vector ';'
    |   'color' expr_vector ';'

color_ior ::=
        'indexOfRefraction' real ';'
    |   'ior' real ';'
    ;
```

Portal parameters are described in §24. To be useful, each portal instance contained within this color class should have a name that is unique within this color instance. Expression vectors are described in §26.5. The color, if it is to be useful, must specify the channels vector. If the specularness is omitted, it is assumed to be `0.1`. If the transparency is omitted, it is assumed to be zero. If the reflectiveness is omitted, it is assumed to be zero. If the Phong factor is omitted, it is assumed to be `15.0`. If the index of refraction is omitted, it is assumed to be one.

The color instance keeps a pointer to the scene to which it belongs so that it can help any portals contained within the color to track down the universes they wish to use.

⟨*Color Member Variables*⟩≡

```
const scene* myScene;
```

The `color` class defines all aspects of the color of an object. This class tracks the color channel values.

⟨*Color Member Variables*⟩+≡

```
basevec< expr* > channels;
```

The channels above may need to reference portals. The following map tracks the portals for this color.

⟨*Color Member Variables*⟩+≡

```
typedef std::map< std::string, class portal* > PortalMap;
PortalMap portals;
```

In addition, this class tracks the specularness, transparency, reflectiveness, Phong factor, and index of refraction.

⟨*Color Member Variables*⟩+≡
```
double specularness;
double transparency;
double reflectiveness;
double phong;
double indexOfRefraction;
```
Note: It may be nice to allow these to be full-blown expressions at some point.

The color may vary with the position and normal at the point of the color and the direction from which the color is viewed. If any of those are true, the appropriate boolean is set here. If they are not true, then the color does not bother to reorient the unneeded vectors.

⟨*Color Member Variables*⟩+≡
```
bool usesPosition;
bool usesNormal;
bool usesDirection;
```

In order to allow flexibility in the position and orientation of the color field, the color keeps its own scale, orientation, and center.

⟨*Color Member Variables*⟩+≡
```
mathvec< double > scale;
basevec< mathvec< double > > orientation;
mathvec< double > center;
```

25.1 Constructor

The color constructor sets up default values for the color.

⟨*Color constructor defaults*⟩≡
```
specularness( 0.1 ), transparency( 0.0 ), reflectiveness( 0.0 ),
phong( 15.0 ), indexOfRefraction( 1.0 ),
usesPosition( false ), usesNormal( false ), usesDirection( false )
```

The default constructor does nothing more than prepare the defaults.

⟨*Color Method Declarations*⟩≡
```
color( const scene* ss );
```

⟨*Color Methods*⟩≡
```
color::color( const scene* ss ) : myScene( ss ), ⟨Color constructor defaults⟩
{
}
```

This constructor allows other objects to initialize a color from its string representation.

⟨*Color Method Declarations*⟩+≡
```
color( const char* str, const scene* ss = 0 );
```

⟨*Color Methods*⟩+≡
```
color::color( const char* str, const scene* ss )
    : myScene( ss ), ⟨Color constructor defaults⟩
{
    std::istringstream in( str );
    in >> *this;
}
```

25.2 Setting Up The Number Of Dimensions

When the number of dimensions are set on the object which contains this color, that information is passed along to the color class so that it can resize the appropriate vectors and inform its child portals.

⟨*Color Method Declarations*⟩+≡
```
void setDimensions( unsigned int dd );
```

⟨*Color Methods*⟩+≡
```
void
color::setDimensions( unsigned int dd )
{
    this->scale.resize( dd, 1.0 );
    ::orthogonalize( this->orientation, dd );
    this->center.resize( dd, 0.0 );

    PortalMap::iterator it;
    for ( it = this->portals.begin(); it != this->portals.end(); ++it ) {
        it->second->setDimensions( dd );
    }
}
```

25.3 Evaluating The Color

This method allows one to query the number of channels this color provides.

⟨*Color Method Declarations*⟩+≡

```
inline unsigned int getChannelCount( void ) const {
    return this->channels.size();
};
```

The following methods allow one to query basic properties of the color.

⟨*Color Method Declarations*⟩+≡

```
inline double getReflectiveness( void ) const {
    return this->reflectiveness;
};
inline double getTransparency( void ) const {
    return this->transparency;
};
inline double getIndexOfRefraction( void ) const {
    return this->indexOfRefraction;
};
inline double getSpecularness( void ) const {
    return this->specularness;
};
inline double getPhong( void ) const {
    return this->phong;
};
```

The following method allows one to query the color at a given position, with a given normal, and from a given direction.

⟨*Color Method Declarations*⟩+≡

```
void evaluate(
    const mathvec< double >& position,
    const mathvec< double >& normal,
    const mathvec< double >& direction,
    double contribution, unsigned int height,
    mathvec< double >& pixel
) const;
```

⟨*Color Methods*⟩+≡
```
void
color::evaluate(
    const mathvec< double >& position,
    const mathvec< double >& normal,
    const mathvec< double >& direction,
    double contribution, unsigned int height,
    mathvec< double >& pixel
) const {
    ⟨Color prepare pixel⟩
    ⟨Color update position, normal, and direction⟩
    ⟨Color evaluate portals⟩
    ⟨Color assign pixel⟩
}
```

If the pixel vector is smaller than the channels vector, the pixel vector is resized.

⟨*Color prepare pixel*⟩≡
```
if ( pixel.size() < this->channels.size() ) {
    pixel.resize( this->channels.size() );
}
```

If the rendering has already reached its maximum depth (the bottom of its height) or if the contribution of this pixel is too small or if this pixel has no pixel expressions, this method stops early.

⟨*Color prepare pixel*⟩+≡
```
if ( height == 0 || contribution < 0.0001 || this->channels.size() == 0 ) {
    pixel *= 0.0;
    return;
}
```

If the position, normal, or direction are used, they will have to be rotated, translated, and scaled into this color's space.

⟨*Color update position, normal, and direction*⟩≡
```
mathvec<double> pos;
mathvec<double> norm;
mathvec<double> dir;
```

The position has to be scaled, rotated, and translated.

⟨*Color update position, normal, and direction*⟩+≡
```
if ( this->usesPosition ) {
    pos = position * this->scale;
    pos.rotate( this->orientation );
    pos += this->center;
}
```

The normal has to be scaled, rotated, and renormalized. It has to be scaled and renormalized because the scaling factor may be different in different directions.

⟨*Color update position, normal, and direction*⟩+≡
```
if ( this->usesNormal ) {
    norm = normal * this->scale;
    norm.rotate( this->orientation );
    norm.normalize();
}
```

⟨*Color update position, normal, and direction*⟩+≡
```
if ( this->usesDirection ) {
    dir = direction * this->scale;
    dir.rotate( this->orientation );
    dir.normalize();
}
```

⟨*Color evaluate portals*⟩≡
```
    PortalMap& pps( *(PortalMap*)&this->portals );
    PortalMap::iterator it;
    for ( it = pps.begin(); it != pps.end(); ++it ) {
        it->second->evaluate(
            pos, dir, contribution, height, this->channels.size()
        );
    }
```

To assign the pixel, this method loops through and evaluates each expression in the channels vector scaling it by the contribution. It also tracks the last stored value so that it can replicate it to the rest of the channels in the pixel.

⟨*Color assign pixel*⟩≡
```
double lastVal = 0.0;

for ( unsigned int ii=0; ii < this->channels.size(); ++ii ) {
    pixel[ii] = this->channels[ii]->evaluate( pos, norm, dir ) * contribution;
    lastVal = pixel[ ii ];
}

for ( unsigned int ii=this->channels.size(); ii < pixel.size(); ++ii ) {
    pixel[ii] = lastVal;
}
```

25.4 Loading A Color

The following method allows one to read in a color from a stream.

⟨*Color Method Declarations*⟩+≡

```
friend std::istream& operator >> ( std::istream& in, color& cc );
```

⟨*Color Methods*⟩+≡
```
std::istream&
operator >> ( std::istream& stream, color& cc )
{
    input in( stream );

    while ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return stream;
        }

        std::string tok( token );
        char semicolon;

        if ( tok == "portal" ) {
            ⟨Color read in portal⟩
        } else if ( tok == "channels" || tok == "color" ) {
            expr::read( stream, cc.channels, cc.portals );
            ⟨Color check for channels using pos, norm, dir⟩
            stream >> semicolon;
        } else if ( tok == "specularness" ) {
            stream >> cc.specularness >> semicolon;
        } else if ( tok == "transparency" ) {
            stream >> cc.transparency >> semicolon;
        } else if ( tok == "reflectiveness" ) {
            stream >> cc.reflectiveness >> semicolon;
        } else if ( tok == "phong" ) {
            stream >> cc.phong >> semicolon;
        } else if ( tok == "indexOfRefraction" || tok == "ior" ) {
            stream >> cc.indexOfRefraction >> semicolon;
        } else if ( tok == "scale" ) {
            stream >> cc.scale >> semicolon;
        } else if ( tok == "orientation" ) {
            stream >> cc.orientation >> semicolon;
        } else if ( tok == "center" ) {
            stream >> cc.center >> semicolon;
        } else {
            in.ignore( "COLOR", tok.c_str() );
        }
    }

    return stream;
}
```

To read in a portal, this code gets the name of the portal. Then, it allocates a new portal, reads it in, and puts it in the portal map.

⟨*Color read in portal*⟩≡
```
if ( in.get( token, sizeof(token) ) == 0 ) {
    break;
}
std::string name( token );
portal* pp = new portal( cc.myScene );
stream >> *pp >> semicolon;
cc.portals[ name ] = pp;
```

This section of code updates the top-level booleans that indicate whether or not this color uses the position, normal, and direction vectors that are passed into the color for evaluation.

⟨*Color check for channels using pos, norm, dir*⟩≡
```
cc.usesPosition = false;
cc.usesNormal = false;
cc.usesDirection = false;
for ( unsigned int ii=0; ii < cc.channels.size(); ++ii ) {
    cc.usesPosition  = cc.usesPosition  || cc.channels[ ii ]->usesPosition();
    cc.usesNormal    = cc.usesNormal    || cc.channels[ ii ]->usesNormal();
    cc.usesDirection = cc.usesDirection || cc.channels[ ii ]->usesDirection();
}
```

25.5 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Color Class Declaration*⟩≡
```
class color {
protected:
    ⟨Color Member Variables⟩
public:
    ⟨Color Method Declarations⟩
public:
    static const char* id;
};
```

25.6 Source Files

The following sections assemble the source files for the `color` class from the chunks above.

25.6.1 color.h

The header file the `color` class includes the `iostream` and `string` headers from the standard C++ libraries and the header file for the `mathvec<>` class and then incorporates the class defined in 25.5 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*color.h*⟩≡

```
#ifndef _NKLEIN_COLOR_H_
#define _NKLEIN_COLOR_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "expr.h"

⟨Color Class Declaration⟩

#endif /*_NKLEIN_COLOR_H_*/
```

25.6.2 color.cc

Herein, the implementations of the methods of the `color` class are included as well as the the declarations and implementations of the derived classes.

⟨*color.cc*⟩≡

```
#include <string>
#include <sstream>
#include <math.h>
#include "input.h"
#include "color.h"
#include "portal.h"

const char* color::id = "NKVERSION: color( 2.1.2007.05.17 )";

⟨Color Methods⟩
```

Chapter 26

Prefix Notation Expressions

Expressions are used in the color class to provide functional texturing. As an example,

⟨*expression bnf*⟩≡

```
expr ::=
        binary_operation
    |   unary_operation
    |   constant_expr
    |   pos_expr
    |   dir_expr
    |   norm_expr
    |   ref_expr
    ;
```

As a complicated example, the following color will place bands on a 4-dimensional ball that correspond to the polar coordinates.

⟨*sample expression*⟩≡

```
channels [3]<
    + 0.5 * 0.5 sign cos * 10 atan2 pos 3 pos 2 ,

    + 0.5 * 0.5 sign cos * 10 atan2
        sqrt + * pos 3 pos 3 * pos 2 pos 2
        pos 1 ,

    + 0.5 * 0.5 sign cos * 10 atan2
        sqrt + + * pos 3 pos 3 * pos 2 pos 2 * pos 1 pos 1
        pos 0
>;
```

Here, the polar coordinates are given by these expressions:

$$x = \cos\theta$$

$$y = \sin\theta\cos\phi$$

$$z = \sin\theta \sin\phi \cos\psi$$
$$w = \sin\theta \sin\phi \sin\psi$$

The red channel is banded according to ψ. The green channel is banded according to ϕ. The blue channel is banded according to θ.

$$\psi = \operatorname{atan}\frac{w}{z}$$
$$\phi = \operatorname{atan}\frac{\sqrt{z^2 + w^2}}{y}$$
$$\theta = \operatorname{atan}\frac{\sqrt{y^2 + z^2 + w^2}}{x}$$

26.1 Destructor

The destructor here in the base class is simply to allow the derived classes to be destructed.

⟨*Expr Method Declarations*⟩≡
```
virtual ~expr( void );
```

⟨*Expr Methods*⟩≡
```
expr::~expr( void ) {}
```

26.2 Checking For Constness

This method is an abstract method which allows the upper layers to check whether this expression is constant or not. If it is constant, it will be replaced by an `exprConst` instance.

⟨*Expr Method Declarations*⟩+≡
```
virtual bool isConstant( void ) const = 0;
virtual bool usesPosition( void ) const = 0;
virtual bool usesNormal( void ) const = 0;
virtual bool usesDirection( void ) const = 0;
```

26.3 Evaluating The Expression

This method is an abstract method which the expression overrides to evaluate the expression.

⟨*Expr Method Declarations*⟩+≡
```
virtual double evaluate(
        const mathvec<double>& position,
        const mathvec<double>& normal,
        const mathvec<double>& direction
    ) = 0;
```

26.4 Loading An Expression

There is a virtual method `get()` defined so that the `read()` methods below can read in derived expressions without having to worry about how that actually works.

⟨*Expr Method Declarations*⟩+≡
```
virtual std::istream& get(
    std::istream& in,
    const std::map< std::string, class portal* >& portals
) = 0;
```

The following methods are factory methods which allow the raytracer to read expressions in from a stream or a string.

⟨*Expr Method Declarations*⟩+≡
```
static expr* read(
        std::istream& in,
        const std::map<std::string, class portal*>& portals
    );
static expr* read(
        const char* str,
        const std::map<std::string, class portal*>& portals
    );
```

This is a rather complicated method as prefix notation can be a bit ugly.

⟨*Expr Methods*⟩+≡

```
expr*
expr::read(
    std::istream& stream,
    const std::map<std::string, class portal*>& portals
) {
    input in( stream );

    if ( in.check() ) {
        char token[ input::MAX_TOKEN_LEN ];

        if ( in.get( token, sizeof(token) ) == 0 ) {
            return 0;
        }

        std::string tok( token );

        if ( tok == "pos" ) {
            expr* nn = new exprPosition();
            nn->get( stream, portals );
            return nn;
        } else if ( tok == "dir" ) {
            expr* nn = new exprDirection();
            nn->get( stream, portals );
            return nn;
        } else if ( tok == "norm" ) {
            expr* nn = new exprNormal();
            nn->get( stream, portals );
            return nn;
        } else if ( tok == "@" ) {
            ⟨Expr read reference expression⟩
        }

        ⟨Expr check binary expressions⟩
        ⟨Expr check unary expressions⟩
        ⟨Expr check constant expressions⟩
    }

    return 0;
}
```

To parse a reference expression, this function retrieves the name of the portal, then it uses that portal name to snag the instance of the portal. Then, it calls the reference expression's constructor with a reference to that portal's pixel.

⟨*Expr read reference expression*⟩≡
```
if ( in.get( token, sizeof(token) ) == 0 ) {
    return 0;
}

std::string name( token );

std::map< std::string, class portal* >::const_iterator pp
    = portals.find( name );
if ( pp == portals.end() ) {
    return 0;
}

exprReference* rret = new exprReference( pp->second->getPixel() );
rret->get( stream, portals );
return rret;
```

This raytracer supports a variety of binary operations for its expressions. These operations are stored in a table. The table itself is defined below in 27.6.

⟨*Expr check binary expressions*⟩≡
```
static const BinaryExpressionTable binaries;
exprBinary* bret = binaries[ tok ];

if ( bret != 0 ) {
    bret->get( stream, portals );
    if ( bret->isConstant() ) {
        mathvec< double > dd;
        exprConst* cret = new exprConst( bret->evaluate( dd, dd, dd ) );
        delete bret;
        return cret;
    } else {
        return bret;
    }
}
```

This raytracer supports a variety of unary operations for its expressions. These operations are stored in a table. The table itself is defined below in 28.6.

⟨*Expr check unary expressions*⟩≡
```
static const UnaryExpressionTable unaries;
exprUnary* uret = unaries[ tok ];

if ( uret != 0 ) {
    uret->get( stream, portals );
    if ( uret->isConstant() ) {
        mathvec< double > dd;
        exprConst* cret = new exprConst( uret->evaluate( dd, dd, dd ) );
        delete uret;
        return cret;
    } else {
        return uret;
    }
}
```

This raytracer keeps a list of common constants along with their numerical value. These constants can be used directly by name. The `ConstantExpressionTable` is defined below in 29.5.

⟨*Expr check constant expressions*⟩≡
```
static const ConstantExpressionTable constants;
exprConst* cret = constants[ tok ];

if ( cret != 0 ) {
    cret->get( stream, portals );
    return cret;
}
```

Alternatively, if the expression has not been any of the named constants above, this raytracer attempts to read it as a numeric constant.

⟨*Expr check constant expressions*⟩+≡
```
    double value;
    std::istringstream str( token );
    str >> value;

    if ( ! str.fail() ) {
        return new exprConst( value );
    }
```

To read from a string, this method simply initializes a `istringstream` from the standard template library to provide the string as an `istream` for the previous method.

⟨*Expr Methods*⟩+≡
```
expr*
expr::read(
    const char* str,
    const std::map<std::string, class portal*>& portals
) {
    std::istringstream in( str );
    return read( in, portals );
}
```

26.5 Loading A Vector Of Expressions

The following methods are used to read in a vector of expressions. One would like to take advantage of the method in the `basevec<>` template class used for input. However, the `basevec<>` input method will not be able to invoke the `read()` method defined above.

⟨*Expr Method Declarations*⟩+≡
```
static void read(
    std::istream& in,
    basevec< expr* >& exprs,
    const std::map< std::string, class portal* >& portals
);
static void read(
    const char* str,
    basevec< expr* >& exprs,
    const std::map< std::string, class portal* >& portals
);
```

As in the `basevec<>` template class input method from 35.2, this method skips over the brackets and the opening less-than sign but saves the length. Then, it loops through that length reading in an expression for each spot in the vector skipping over separating commas. Once all of the expressions have been read, this method reads until the closing greater-than sign is found.

⟨*Expr Methods*⟩+≡

```
void
expr::read(
    std::istream& in,
    basevec< expr* >& exprs,
    const std::map<std::string, class portal*>& portals
) {
    char bogus;
    unsigned int len;

    in.setf( std::ios::skipws );

    in >> bogus;                // "["
    in >> len;
    in >> bogus >> bogus;       // "]<"

    exprs.resize( len, (expr*)0 );

    for (unsigned ii=0; ii < len; ++ii) {
        exprs[ ii ] = expr::read( in, portals );

        if (ii < len-1) {
            in >> bogus; // ","
        }
    }

    while (bogus != '>' && in.good() ) {
        in >> bogus; // ">"
    }
}
```

This read method simply employs the method defined above.

⟨*Expr Methods*⟩+≡

```
void
expr::read(
    const char* str,
    basevec< expr* >& exprs,
    const std::map<std::string, class portal*>& portals
) {
    std::istringstream in( str );
    expr::read( in, exprs, portals );
}
```

26.6 The Expression Table Initializers

And, in order to ensure that the static tables `binaries`, `unaries`, and `constants` above all get constructed before any threaded execution begins, this compilation unit defines a global instance of a class whose constructor simply makes sure that the the `read()` method above gets invoked in such a way as to ensure that all of the relevant tables are checked.

⟨*Expr TableInitializer*⟩≡

```
class TableInitializer {
public:
    TableInitializer( void );
public:
    static const char* id;
};
const char* TableInitializer::id = "NKVERSION: TableInitializer( 2.1.2007.05.17 )";
```

The constructor calls the read giving it an expression which has to exercise all three tables. Then, it deletes that expression.

⟨*Expr TableInitializer*⟩+≡

```
TableInitializer::TableInitializer( void ) {
    std::map<std::string, class portal*> portals;
    expr* ee = expr::read( "+ sin pi 1", portals );
    if ( ee != 0 ) {
        delete ee;
    }
};
```

As mentioned above, there is a single global instance of this class to ensure that the tables above have been initialzed before any threading begins.

⟨*Expr TableInitializer*⟩+≡

```
static TableInitializer __tableInitializer;
```

26.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Expr Class Declaration*⟩≡
```
class expr {
public:
    ⟨Expr Method Declarations⟩
public:
    static const char* id;
};
```

26.8 Source Files

The following sections assemble the source files for the `expr` class from the chunks above.

26.8.1 `expr.h`

The header file the `expr` class includes the `iostream` and `string` headers from the standard C++ libraries and the header file for the `mathvec<>` class and then incorporates the class defined in 26.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*expr.h*⟩≡
```
#ifndef _NKLEIN_EXPR_H_
#define _NKLEIN_EXPR_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "mathvec.h"

class scene;

    ⟨Expr Class Declaration⟩

#endif /*_NKLEIN_EXPR_H_*/
```

26.8.2 `expr.cc`

Herein, the implementations of the methods of the `expr` class are included as well as the the declarations and implementations of the derived classes and table classes.

⟨*expr.cc*⟩≡
```
#include <string>
#include <sstream>
#include <math.h>
#include "input.h"
#include "expr.h"
#include "portal.h"

const char* expr::id = "NKVERSION: expr( 2.1.2007.05.17 )";

⟨Expr Derived Classes⟩

⟨Expr BinaryExpressionTable Definition⟩
⟨Expr UnaryExpressionTable Definition⟩
⟨Expr ConstantExpressionTable Definition⟩
⟨Expr TableInitializer⟩

⟨Expr Methods⟩
```

Chapter 27

Binary Expressions

There are a few different binary operations one could employ in the raytracer.

⟨*binary operations bnf*⟩≡

```
binary_operation ::=
        '+' expr expr
    |   '-' expr expr
    |   '*' expr expr
    |   '/' expr expr
    |   '%' expr expr
    |   'div' expr expr
    |   'atan2' expr expr
    |   '^' expr expr
    |   'pow' expr expr
    |   'max' expr expr
    |   'min' expr expr
    ;
```

For example, if one wanted the red color to be the absolute value of the arctangent of the x- and y-coordinates, then one could specify:

⟨*sample binary expression*⟩≡

```
channels [3]< abs atan2 pos 0 pos 1, 0, 0 >;
```

The binary expression class tracks its evaluation function and its operand expressions.

⟨*ExprBinary Data Members*⟩≡

```
double (*func)( double, double );
expr* left;
expr* right;
```

27.1 Constructor

The binary expression is initialized with its evaluation function.

⟨*ExprBinary Method Declarations*⟩≡
```
exprBinary( double (*_func)( double, double ) )
    : func( _func ), left( 0 ), right( 0 )
{
};
```

27.2 Destructor

⟨*ExprBinary Method Declarations*⟩+≡
```
~exprBinary( void )
{
    delete this->right;
    delete this->left;
};
```

27.3 Checking For Constness

The binary expression is constant if and only if both of its operands are constant.

⟨*ExprBinary Method Declarations*⟩+≡
```
virtual bool isConstant( void ) const
{
    return this->left->isConstant() && this->right->isConstant();
};
```

⟨*ExprBinary Method Declarations*⟩+≡
```
virtual bool usesPosition( void ) const
{
    return this->left->usesPosition() || this->right->usesPosition();
};
```

⟨*ExprBinary Method Declarations*⟩+≡
```
virtual bool usesNormal( void ) const
{
    return this->left->usesNormal() || this->right->usesNormal();
};
```

⟨*ExprBinary Method Declarations*⟩+≡
```
virtual bool usesDirection( void ) const
{
    return this->left->usesDirection() || this->right->usesDirection();
};
```

27.4 Evaluating The Expression

To evaluate a binary operation, both of the operands are evaluated and passed to the function which evaluates this operation given the operands.

⟨*ExprBinary Method Declarations*⟩+≡
```
virtual double evaluate(
        const mathvec<double>& pos,
        const mathvec<double>& norm,
        const mathvec<double>& dir
    )
{
    return (*this->func)(
        this->left->evaluate( pos, norm, dir ),
        this->right->evaluate( pos, norm, dir )
    );
}
```

27.5 Reading In A Binary Expression

With a binary expression, there are two operands to read.

⟨*ExprBinary Method Declarations*⟩+≡
```
virtual std::istream& get(
    std::istream& in,
    const std::map< std::string, class portal* >& portals
) {
    delete this->left;
        this->left = expr::read( in, portals );
    delete this->right;
        this->right = expr::read( in, portals );
    return in;
};
```

27.6 The Binary Expression Table

The following class is used in reading expressions to efficiently store and return named binary expressions.

⟨*Expr BinaryExpressionTable Definition*⟩≡

```
class BinaryExpressionTable {
protected:
    typedef double (*FunctionType)( double, double );
    typedef std::map< std::string, FunctionType > TableType;
    typedef TableType::const_iterator const_iterator;
    TableType table;
public:
    BinaryExpressionTable( void );
    exprBinary* operator [] ( const std::string& name ) const;

    static double add( double aa, double bb );
    static double sub( double aa, double bb );
    static double mul( double aa, double bb );
    static double div( double aa, double bb );
    static double mod( double aa, double bb );
    static double idiv( double aa, double bb );
    static double min( double aa, double bb );
    static double max( double aa, double bb );
public:
    static const char* id;
};

const char* BinaryExpressionTable::id
    = "NKVERSION: BinaryExpressionTable( 2.1.2007.05.17 )";
```

The constructor initializes the table of binary expressions with pointers to functions which implement the expressions. These are the binary expressions known to this raytracer.

⟨*Expr BinaryExpressionTable Definition*⟩+≡

```
BinaryExpressionTable::BinaryExpressionTable( void )
{
    this->table[ "+" ] = BinaryExpressionTable::add;
    this->table[ "-" ] = BinaryExpressionTable::sub;
    this->table[ "*" ] = BinaryExpressionTable::mul;
    this->table[ "/" ] = BinaryExpressionTable::div;
    this->table[ "%" ] = BinaryExpressionTable::mod;
    this->table[ "div" ] = BinaryExpressionTable::idiv;
    this->table[ "atan2" ] = (double (*)( double, double ))::atan2;
    this->table[ "^" ] = (double (*)( double, double ))::pow;
    this->table[ "min" ] = BinaryExpressionTable::min;
    this->table[ "max" ] = BinaryExpressionTable::max;
}
```

The array accessor for this table checks to see if the element is in the table. If it is, then it returns a constant expression for that element. Otherwise, it returns null.

⟨*Expr BinaryExpressionTable Definition*⟩+≡

```
exprBinary*
BinaryExpressionTable::operator [] ( const std::string& name ) const
{
    const_iterator it = this->table.find( name );
    if ( it != this->table.end() ) {
        return new exprBinary( it->second );
    } else {
        return 0;
    }
}
```

Most of the binary operations which this class implements itself are very straightforward.

⟨*Expr BinaryExpressionTable Definition*⟩+≡
```
double BinaryExpressionTable::add( double aa, double bb ) {
    return aa + bb;
}

double BinaryExpressionTable::sub( double aa, double bb ) {
    return aa - bb;
}

double BinaryExpressionTable::mul( double aa, double bb ) {
    return aa * bb;
}

double BinaryExpressionTable::div( double aa, double bb ) {
    return aa / bb;
}
```

The `mod()` function deserves special attention, however. Here, the remainder is determined through the use of the `modf()` from the math library. The `modf()` splits the quotient `aa/bb` into an integer portion `ii` and a remainder between zero and one `rr`. That remainder is then returned scaled back up into the range zero to `bb`.

⟨*Expr BinaryExpressionTable Definition*⟩+≡
```
double BinaryExpressionTable::mod( double aa, double bb ) {
    double ii;
    double rr = ::modf( aa / bb, &ii );

    return rr * bb;
}
```

The `idiv()` function serves a similar purpose except that it returns the integer multiple of `bb` rather than the remainder.

⟨*Expr BinaryExpressionTable Definition*⟩+≡
```
double BinaryExpressionTable::idiv( double aa, double bb ) {
    double ii;
    (void) ::modf( aa / bb, &ii );

    return ii * bb;
}
```

The `min()` and `max()` functions are easy.

⟨*Expr BinaryExpressionTable Definition*⟩+≡

```
double BinaryExpressionTable::min( double aa, double bb ) {
    if ( aa < bb ) {
        return aa;
    } else {
        return bb;
    }
}
double BinaryExpressionTable::max( double aa, double bb ) {
    if ( aa < bb ) {
        return bb;
    } else {
        return aa;
    }
}
```

27.7 The Class Definition

This class inherits directly from the `expr` class. It includes all of the methods declared above. And, it includes some version identification information.

⟨*Expr Derived Classes*⟩≡

```
class exprBinary : public expr {
public:
    ⟨ExprBinary Method Declarations⟩
protected:
    ⟨ExprBinary Data Members⟩
public:
    static const char* id;
};

const char* exprBinary::id = "NKVERSION: exprBinary( 2.1.2007.05.17 )";
```

Chapter 28

Unary Expressions

There are a fair number of unary expressions available within the raytracer.

⟨*unary operations bnf*⟩≡

```
    unary_operation ::=
            'sin' expr
        |   'cos' expr
        |   'tan' expr
        |   'atan' expr
        |   'sinh' expr
        |   'cosh' expr
        |   'tanh' expr
        |   'sqrt' expr
        |   'cbrt' expr
        |   'exp' expr
        |   'log' expr
        |   'log10' expr
        |   'fabs' expr
        |   'floor' expr
        |   'ceil' expr
        |   'rint' expr
        |   'round' expr
        |   'gamma' expr
        |   'lgamma' expr
        |   'erf' expr
        ;
```

For example, if one wanted a blue and yellow checker pattern on an object, one might specify the color using:

⟨*sample unary expression*⟩≡

```
    channels [3]<
        0.5,
        0.5,
        + 0.5 * 0.5 sign * * cos * 10 pos 0 cos * 10 pos 1 cos * 10 pos 2
```

```
    >;
```

This says that the blue channel will be one half plus one half times the sign the products of ten times the cosines of the x, y, and z coordinates of the object.

The class maintains a pointer to a function which can evaluate this unary expression. It also tracks the operand expression.

⟨*ExprUnary Data Members*⟩≡
```
double (*func)( double );
expr* operand;
```

28.1 Constructor

⟨*ExprUnary Method Declarations*⟩≡
```
exprUnary( double (*_func)( double ) )
    : func( _func ), operand( 0 )
{
};
```

28.2 Destructor

⟨*ExprUnary Method Declarations*⟩+≡
```
~exprUnary( void )
{
    delete this->operand;
};
```

28.3 Checking For Constness

A unary expression is constant if and only if its argument is constant.

⟨*ExprUnary Method Declarations*⟩+≡
```
virtual bool isConstant( void ) const
{
    return this->operand->isConstant();
};
```

⟨*ExprUnary Method Declarations*⟩+≡
```
virtual bool usesPosition( void ) const
{
    return this->operand->usesPosition();
};
```

⟨*ExprUnary Method Declarations*⟩+≡
```
virtual bool usesNormal( void ) const
{
    return this->operand->usesNormal();
};
```

⟨*ExprUnary Method Declarations*⟩+≡
```
virtual bool usesDirection( void ) const
{
    return this->operand->usesDirection();
};
```

28.4 Evaluating The Expression

To evaluate a unary expression, the argument is evaluated, then the value is passed into the unary expression routine.

⟨*ExprUnary Method Declarations*⟩+≡
```
virtual double evaluate(
        const mathvec<double>& pos,
        const mathvec<double>& norm,
        const mathvec<double>& dir
    )
{
    return (*this->func)( this->operand->evaluate( pos, norm, dir ) );
}
```

28.5 Reading In A Unary Expression

With a unary expression, there is one operand to read.

⟨*ExprUnary Method Declarations*⟩+≡
```
virtual std::istream& get(
    std::istream& in,
    const std::map< std::string, class portal* >& portals
) {
    delete this->operand;
        this->operand = expr::read( in, portals );
    return in;
};
```

28.6 The Unary Expression Table

The following class is used in reading expressions to efficiently store and return `expr` instances which evaulate unary expressions.

⟨*Expr UnaryExpressionTable Definition*⟩≡

```
class UnaryExpressionTable {
protected:
    typedef double (*FunctionType)( double );
    typedef std::map< std::string, FunctionType > TableType;
    typedef TableType::const_iterator const_iterator;
    TableType table;
public:
    UnaryExpressionTable( void );
    exprUnary* operator [] ( const std::string& name ) const;

    static double sign( double );
public:
    static const char* id;
};
const char* UnaryExpressionTable::id
    = "NKVERSION: UnaryExpressionTable( 2.1.2007.05.17 )";
```

The constructor initializes the table of unary expressions with pointers to functions which perform the evaluation. In this case, all of these expressions are from the standard math libary. These are the unary expressions known to this raytracer.

⟨*Expr UnaryExpressionTable Definition*⟩+≡
```
UnaryExpressionTable::UnaryExpressionTable( void )
{
    this->table[ "sin" ] = (double (*)( double ))::sin;
    this->table[ "cos" ] = (double (*)( double ))::cos;
    this->table[ "tan" ] = (double (*)( double ))::tan;
    this->table[ "atan" ] = (double (*)( double ))::atan;
    this->table[ "sinh" ] = (double (*)( double ))::sinh;
    this->table[ "cosh" ] = (double (*)( double ))::cosh;
    this->table[ "tanh" ] = (double (*)( double ))::tanh;
    this->table[ "sqrt" ] = (double (*)( double ))::sqrt;
    this->table[ "cbrt" ] = (double (*)( double ))::cbrt;
    this->table[ "exp" ] = (double (*)( double ))::exp;
    this->table[ "log" ] = (double (*)( double ))::log;
    this->table[ "log10" ] = (double (*)( double ))::log10;
    this->table[ "fabs" ] = (double (*)( double ))::fabs;
    this->table[ "floor" ] = (double (*)( double ))::floor;
    this->table[ "ceil" ] = (double (*)( double ))::ceil;
    this->table[ "rint" ] = (double (*)( double ))::rint;
    this->table[ "round" ] = (double (*)( double ))::rint;
    this->table[ "sign" ] = UnaryExpressionTable::sign;
    this->table[ "gamma" ] = (double (*)( double ))::gamma;
    this->table[ "lgamma" ] = (double (*)( double ))::lgamma;
    this->table[ "erf" ] = (double (*)( double ))::erf;
}
```

The only unary expression not available directly from the standard math library is the `sign()` function:

⟨*Expr UnaryExpressionTable Definition*⟩+≡
```
double UnaryExpressionTable::sign( double vv ) {
    if ( vv > 0.000001 ) {
        return  1.0;
    } else if ( vv < -0.000001 ) {
        return -1.0;
    } else {
        return  0.0;
    }
}
```

The array accessor for this table checks to see if the element is in the table. If it is, then it returns a constant expression for that element. Otherwise, it returns null.

⟨*Expr UnaryExpressionTable Definition*⟩+≡
```
exprUnary*
UnaryExpressionTable::operator [] ( const std::string& name ) const
{
    const_iterator it = this->table.find( name );
    if ( it != this->table.end() ) {
        return new exprUnary( it->second );
    } else {
        return 0;
    }
}
```

28.7 The Class Definition

This class inherits directly from the `expr` class. It includes all of the methods declared above. And, it includes some version identification information.

⟨*Expr Derived Classes*⟩+≡
```
class exprUnary : public expr {
public:
    ⟨ExprUnary Method Declarations⟩
protected:
    ⟨ExprUnary Data Members⟩
public:
    static const char* id;
};

const char* exprUnary::id = "NKVERSION: exprUnary( 2.1.2007.05.17 )";
```

Chapter 29

Constant Expressions

There are several named constant expressions and numerical constants.
⟨*constant expression bnf*⟩≡
```
    constant_expr ::=
            'pi'
        |   '2pi'
        |   'e'
        |   'sqrt2'
        |   real
        ;
```

The numerical constant is simply stored within this expression.
⟨*ExprConst Data Members*⟩≡
```
double value;
```

29.1 Constructor

The expression is initialized with a value.
⟨*ExprConst Method Declarations*⟩≡
```
exprConst( double _value ) : value( _value ) {};
```

29.2 Checking For Constness

This expression is entirely constant.

⟨*ExprConst Method Declarations*⟩+≡
```
virtual bool isConstant( void ) const
{
    return true;
};
```

⟨*ExprConst Method Declarations*⟩+≡
```
virtual bool usesPosition( void ) const
{
    return false;
};
```

⟨*ExprConst Method Declarations*⟩+≡
```
virtual bool usesNormal( void ) const
{
    return false;
};
```

⟨*ExprConst Method Declarations*⟩+≡
```
virtual bool usesDirection( void ) const
{
    return false;
};
```

29.3 Evaluating The Expression

This method returns the value of the constant directly.

⟨*ExprConst Method Declarations*⟩+≡
```
virtual double evaluate(
        const mathvec<double>& /*position*/,
        const mathvec<double>& /*normal*/,
        const mathvec<double>& /*direction*/
    )
{
    return this->value;
}
```

29.4 Reading In A Constant Expression

With a constant expression, by the time it is initialized, it already has all of its information. This `get()` method doesn't have anything to do then.

⟨*ExprConst Method Declarations*⟩+≡
```
virtual std::istream& get(
    std::istream& in,
    const std::map< std::string, class portal* >& /*portals*/
) {
    return in;
};
```

29.5 The Constant Expression Table

The following class is used in reading expressions to efficiently store and return named constant values.

⟨*Expr ConstantExpressionTable Definition*⟩≡
```
class ConstantExpressionTable {
protected:
    typedef std::map< std::string, double > TableType;
    typedef TableType::const_iterator const_iterator;
    TableType table;
public:
    ConstantExpressionTable( void );
    exprConst* operator [] ( const std::string& name ) const;
public:
    static const char* id;
};

const char* ConstantExpressionTable::id
    = "NKVERSION: ConstantExpressionTable( 2.1.2007.05.17 )";
```

The constructor initializes the table of constant expressions with the known constants. These are the constant expressions known to this raytracer.

⟨*Expr ConstantExpressionTable Definition*⟩+≡
```
ConstantExpressionTable::ConstantExpressionTable( void )
{
    this->table[ "pi" ] = M_PI;
    this->table[ "2pi" ] = 2.0 * M_PI;
    this->table[ "e" ] = M_E;
    this->table[ "sqrt2" ] = M_SQRT2;
}
```

The array accessor for this table checks to see if the element is in the table. If it is, then it returns a constant expression for that element. Otherwise, it returns null.

⟨*Expr ConstantExpressionTable Definition*⟩+≡
```
exprConst*
ConstantExpressionTable::operator [] ( const std::string& name ) const
{
    const_iterator it = this->table.find( name );
    if ( it != this->table.end() ) {
        return new exprConst( it->second );
    } else {
        return 0;
    }
}
```

29.6 The Class Definition

This class inherits directly from the `expr` class. It includes all of the methods declared above. And, it includes some version identification information.

⟨*Expr Derived Classes*⟩+≡
```
class exprConst : public expr {
public:
    ⟨ExprConst Method Declarations⟩
protected:
    ⟨ExprConst Data Members⟩
public:
    static const char* id;
};

const char* exprConst::id = "NKVERSION: exprConst( 2.1.2007.05.17 )";
```

Chapter 30

Position Expressions

The color of an object can be a function of the position at that point. The `expr` operand to this call is used as the coordinate index within the position vector.

⟨*position expression bnf*⟩≡

```
    pos_expr ::=  'pos' expr ;
```

For example, if one wanted an object to be colored more red the further it was along the w-axis, one might specify the color with:

⟨*sample position expression*⟩≡

```
    channels [3]< + 0.5 * 0.5 pos 3, 0.2, 0.2 >;
```

The position expression needs a single operand telling which coordinate of the position to use.

⟨*ExprPosition Data Members*⟩≡

```
expr* operand;
```

30.1 Constructor

⟨*ExprPosition Method Declarations*⟩≡

```
exprPosition( void )
    : operand( 0 )
{
};
```

30.2 Destructor

⟨*ExprPosition Method Declarations*⟩+≡
```
~exprPosition( void )
{
    delete this->operand;
};
```

30.3 Checking For Constness

As this depends upon the position vector for the color, it cannot be taken as a constant.

⟨*ExprPosition Method Declarations*⟩+≡
```
virtual bool isConstant( void ) const
{
    return false;
};
```

⟨*ExprPosition Method Declarations*⟩+≡
```
virtual bool usesPosition( void ) const
{
    return true;
};
```

⟨*ExprPosition Method Declarations*⟩+≡
```
virtual bool usesNormal( void ) const
{
    return this->operand->usesNormal();
};
```

⟨*ExprPosition Method Declarations*⟩+≡
```
virtual bool usesDirection( void ) const
{
    return this->operand->usesDirection();
};
```

30.4 Evaluating The Expression

To evaluate the direction, the method here evaluates the operand expression, rounds the result to an integer, and returns that coordinate from the position vector.

⟨*ExprPosition Method Declarations*⟩+≡
```
virtual double evaluate(
        const mathvec<double>& pos,
        const mathvec<double>& norm,
        const mathvec<double>& dir
    )
{
    double op = this->operand->evaluate( pos, norm, dir );
    unsigned int index = (unsigned int)( op + 0.5 );
    return pos[ index ];
}
```

30.5 Reading In A Position Expression

With a position expression, there is an operand to read.

⟨*ExprPosition Method Declarations*⟩+≡
```
virtual std::istream& get(
    std::istream& in,
    const std::map< std::string, class portal* >& portals
) {
    delete this->operand;
        this->operand = expr::read( in, portals );
    return in;
};
```

30.6 The Class Definition

This class inherits directly from the `expr` class. It includes all of the methods declared above. And, it includes some version identification information.

⟨*Expr Derived Classes*⟩+≡

```
class exprPosition : public expr {
public:
    ⟨ExprPosition Method Declarations⟩
protected:
    ⟨ExprPosition Data Members⟩
public:
    static const char* id;
};

const char* exprPosition::id = "NKVERSION: exprPosition( 2.1.2007.05.17 )";
```

Chapter 31

Direction Expressions

The color of an object can be a function of the direction from which it is viewed. The `expr` operand to this call is used as the coordinate index within the direction vector.

⟨*direction expression bnf*⟩≡
```
    dir_expr ::=  'dir' expr ;
```
As an example, suppose one wants a particular object to look bright blue when seen directly along the x-axis and less blue the further away from the x-axis one gets. One might specify the color as:

⟨*sample direction expression*⟩≡
```
    channels [3]< 0.5, 0.5, + 0.5 * 0.5 abs dir 0 >;
```
That is to say that the red and green channels are always set at a half. But, the blue channel is a half plus half the absolute value of the x-coordinate in the direction vector.

The direction expression needs a single operand telling which coordinate of the direction to use.

⟨*ExprDirection Data Members*⟩≡
```
expr* operand;
```

31.1 Constructor

⟨*ExprDirection Method Declarations*⟩≡
```
exprDirection( void )
    : operand( 0 )
{
};
```

31.2 Destructor

⟨*ExprDirection Method Declarations*⟩+≡
```
~exprDirection( void )
{
    delete this->operand;
};
```

31.3 Checking For Constness

As this depends upon the direction vector for the color, it cannot be taken as a constant.

⟨*ExprDirection Method Declarations*⟩+≡
```
virtual bool isConstant( void ) const
{
    return false;
};
```

⟨*ExprDirection Method Declarations*⟩+≡
```
virtual bool usesPosition( void ) const
{
    return this->operand->usesPosition();
};
```

⟨*ExprDirection Method Declarations*⟩+≡
```
virtual bool usesNormal( void ) const
{
    return this->operand->usesNormal();
};
```

⟨*ExprDirection Method Declarations*⟩+≡
```
virtual bool usesDirection( void ) const
{
    return true;
};
```

31.4 Evaluating The Expression

To evaluate the direction, the method here evaluates the operand expression, rounds the result to an integer, and returns that coordinate from the direction vector.

⟨*ExprDirection Method Declarations*⟩+≡

```
virtual double evaluate(
        const mathvec<double>& pos,
        const mathvec<double>& norm,
        const mathvec<double>& dir
    )
{
    double op = this->operand->evaluate( pos, norm, dir );
    unsigned int index = (unsigned int)( op + 0.5 );
    return dir[ index ];
}
```

31.5 Reading In A Direction Expression

With a direction expression, there is one operand to read.

⟨*ExprDirection Method Declarations*⟩+≡

```
virtual std::istream& get(
    std::istream& in,
    const std::map< std::string, class portal* >& portals
) {
    delete this->operand;
        this->operand = expr::read( in, portals );
    return in;
};
```

31.6 The Class Definition

This class inherits directly from the `expr` class. It includes all of the methods declared above. And, it includes some version identification information.

⟨*Expr Derived Classes*⟩+≡

```
class exprDirection : public expr {
public:
    ⟨ExprDirection Method Declarations⟩
protected:
    ⟨ExprDirection Data Members⟩
public:
    static const char* id;
};

const char* exprDirection::id = "NKVERSION: exprDirection( 2.1.2007.05.17 )";
```

Chapter 32

Normal Expressions

The color of an object can be a function of the normal at that point. The `expr` operand to this call is used as the coordinate index within the normal.

⟨*normal expression bnf*⟩≡

```
    norm_expr ::=  'norm' expr ;
```

As an example, if one wanted to color a 3-D object based upon the normal of the object at the point of intersection with the x-coordinate giving the red channel, the y-coordinate giving the green channel, and the z-coordinate giving the blue channel, one would specify the color channels like this:

⟨*sample normal expression*⟩≡

```
    channels [3]< norm 0, norm 1, norm 2 >;
```

The normal expression needs a single operand telling which coordinate of the normal to use.

⟨*ExprNormal Data Members*⟩≡

```
expr* operand;
```

32.1 Constructor

⟨*ExprNormal Method Declarations*⟩≡

```
exprNormal( void )
    : operand( 0 )
{
};
```

32.2 Destructor

⟨*ExprNormal Method Declarations*⟩+≡
```
~exprNormal( void )
{
    delete this->operand;
};
```

32.3 Checking For Constness

As this depends upon the normal vector for the color, it cannot be taken as a constant.

⟨*ExprNormal Method Declarations*⟩+≡
```
virtual bool isConstant( void ) const
{
    return false;
};
```

⟨*ExprNormal Method Declarations*⟩+≡
```
virtual bool usesPosition( void ) const
{
    return this->operand->usesPosition();
};
```

⟨*ExprNormal Method Declarations*⟩+≡
```
virtual bool usesNormal( void ) const
{
    return true;
};
```

⟨*ExprNormal Method Declarations*⟩+≡
```
virtual bool usesDirection( void ) const
{
    return this->operand->usesDirection();
};
```

32.4 Evaluating The Expression

To evaluate the normal, the method here evaluates the operand expression, rounds the result to an integer, and returns that coordinate from the normal vector.

⟨*ExprNormal Method Declarations*⟩+≡
```
virtual double evaluate(
        const mathvec<double>& pos,
        const mathvec<double>& norm,
        const mathvec<double>& dir
    )
{
    double op = this->operand->evaluate( pos, norm, dir );
    unsigned int index = (unsigned int)( op + 0.5 );
    return norm[ index ];
}
```

32.5 Reading In A Normal Expression

With a normal expression, there is one operand to read.

⟨*ExprNormal Method Declarations*⟩+≡
```
virtual std::istream& get(
    std::istream& in,
    const std::map< std::string, class portal* >& portals
) {
    delete this->operand;
        this->operand = expr::read( in, portals );
    return in;
};
```

32.6 The Class Definition

This class inherits directly from the `expr` class. It includes all of the methods declared above. And, it includes some version identification information.

⟨*Expr Derived Classes*⟩+≡

```
class exprNormal : public expr {
public:
    ⟨ExprNormal Method Declarations⟩
protected:
    ⟨ExprNormal Data Members⟩
public:
    static const char* id;
};

const char* exprNormal::id = "NKVERSION: exprNormal( 2.1.2007.05.17 )";
```

Chapter 33

Reference Expressions

The reference expression refers to a channel of a pixel traced through a portal. The `portal_name` tells which portal this references and the `expr` operand is used to determine which channel to employ.

⟨*reference expression bnf*⟩≡

```
ref_expr ::=  '@' portal_name expr ;
```

For example, if each of the color channels (red, green, and blue) were to be pulled from a separate portal, the `channels` vector specified in the color may look like this:

⟨*sample reference expression*⟩≡

```
channels [3]< @ red_portal 0, @ green_portal 1, @ blue_portal 2 >;
```

Similarly, if one wished to use only the green channel from the trace into the poral as the output channel in a view's viewport, one might do something like this:

⟨*sample reference expression*⟩+≡

```
channels [1]< @ my_portal 1 >;
```

The reference expression needs a pixel to reference and a single operand telling which coordinate of the pixel to use.

⟨*ExprReference Data Members*⟩≡

```
const mathvec< double >& pixel;
expr* operand;
```

33.1 Constructor

The constructor initializes the pixel reference.

⟨*ExprReference Method Declarations*⟩≡
```
exprReference( const mathvec< double >& pp )
    : pixel( pp ), operand( 0 )
{
};
```

33.2 Destructor

When this expression is destroyed, it must destroy its operand expression.

⟨*ExprReference Method Declarations*⟩+≡
```
~exprReference( void )
{
    delete this->operand;
};
```

33.3 Checking For Constness

As this depends upon the reference vector for the color, it cannot be taken as a constant.

⟨*ExprReference Method Declarations*⟩+≡
```
virtual bool isConstant( void ) const
{
    return false;
};
```

⟨*ExprReference Method Declarations*⟩+≡
```
virtual bool usesPosition( void ) const
{
    return true;
};
```

⟨*ExprReference Method Declarations*⟩+≡
```
virtual bool usesNormal( void ) const
{
    return true;
};
```

⟨*ExprReference Method Declarations*⟩+≡
```
virtual bool usesDirection( void ) const
{
    return true;
};
```

33.4 Evaluating The Expression

To evaluate this expression, the operand is evaluated. Then, the operand is rounded to the nearest integer. The rounded integer is used to select the channel from the pixel.

⟨*ExprReference Method Declarations*⟩+≡
```
virtual double evaluate(
        const mathvec<double>& pos,
        const mathvec<double>& norm,
        const mathvec<double>& dir
    )
{
    double op = this->operand->evaluate( pos, norm, dir );
    unsigned int index = (unsigned int)( op + 0.5 );
    return this->pixel[ index ];
}
```

33.5 Reading In A Reference Expression

With a reference expression, there is one operand to read.

⟨*ExprReference Method Declarations*⟩+≡
```
virtual std::istream& get(
    std::istream& in,
    const std::map< std::string, class portal* >& portals
) {
    delete this->operand;
        this->operand = expr::read( in, portals );
    return in;
};
```

33.6 The Class Definition

This class inherits directly from the `expr` class. It includes all of the methods declared above. And, it includes some version identification information.

⟨*Expr Derived Classes*⟩+≡

```
class exprReference : public expr {
public:
    ⟨ExprReference Method Declarations⟩
protected:
    ⟨ExprReference Data Members⟩
public:
    static const char* id;
};

const char* exprReference::id = "NKVERSION: exprReference( 2.1.2007.05.17 )";
```

Chapter 34

Mathematical Vectors

An integer vector is expressed as an integer in square brackets which specifies the length of the vector. Following that, the vector is a comma-separated list of integers enclosed in angle brackets.

⟨*integer vector bnf*⟩≡

```
int_vector ::=
        '[' integer ']' '<' integer_list '>'
    ;

integer_list ::=
        integer_list ',' integer
    |   integer
    ;
```

The number of elements in the list must be at least as great as the number given in the square brackets at the start.

A real vector is expressed as an integer in square brackets which specifies the length of the vector. Following that, the vector is a comma-separated list of reals enclosed in angle brackets.

⟨*real vector bnf*⟩≡

```
real_vector ::=
        '[' int ']' '<' real_list '>'
    ;

real_list ::=
        real_list ',' real
    |   real
    ;
```

The number of elements in the list must be at least as great as the number given in the square brackets at the start.

A real matrix is expressed as an integer in square brackets which specifies the number of rows in the vector. Following that, the matrix is a comma-separated list of real vectors enclosed in angle brackets.

⟨*real matrix bnf*⟩≡
```
    real_matrix ::=
            '[' int ']' '<' real_vector_list '>'
        ;

    real_vector_list ::=
            real_vector_list ',' real_vector
        |   real_vector
        ;
```

The number of vectors in the list must be at least as great as the number given in the square brackets at the start.

This template class extends the `basevec<>` class from section 35 to include operations for vector mathematics.

The code in this raytracer uses two different types of these vectors. One is simply a vector of `double`s. The second type is a vector of `unsigned int`s used for counters.

⟨*MathVec Template Instantiations*⟩≡
```
template class mathvec< double >;
template class mathvec< unsigned int >;
template class basevec< mathvec< double > >;
template class basevec< mathvec< unsigned int > >;
```

The code in this raytracer also needs to orthogonalize matrixen of double-vectors.

⟨*MathVec Template Instantiations*⟩+≡
```
template bool orthogonalize(
    basevec< mathvec< double > >& matrix,
    unsigned int dims
);
```

And, in one instance, we need to transpose a matrix.

⟨*MathVec Template Instantiations*⟩+≡
```
template void transpose(
    basevec< mathvec< double > >& matrix
);
```

34.1 Creating Vectors

There are several ways to create vectors. The most basice way is using the default constructor which simply initializes a zero-length vector.

⟨*MathVec Template Method Declarations*⟩≡
```
mathvec();
```

⟨*MathVec Template Methods*⟩≡
```
template < typename TT >
mathvec< TT >::mathvec() : basevec< TT >()
{
}
```

One could alternatively allocate a zero-vector of a fixed length.

⟨*MathVec Template Method Declarations*⟩+≡
```
mathvec( unsigned int len );
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
mathvec< TT >::mathvec( unsigned int len ) : basevec< TT >()
{
    TT zero( 0U );
    this->resize( len, zero );
}
```

One could initialize a vector from a given value and length.

⟨*MathVec Template Method Declarations*⟩+≡
```
mathvec( const TT value, unsigned int len );
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
mathvec< TT >::mathvec( const TT value, unsigned int len )
    : basevec< TT >( value, len )
{
}
```

One could initialize a vector from a given array and given length.

⟨*MathVec Template Method Declarations*⟩+≡
```
mathvec( const TT array[], unsigned int len );
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
mathvec< TT >::mathvec( const TT array[], unsigned int len )
    : basevec< TT >( array, len )
{
}
```

One could copy an existing vector. This method makes use of the assignment operator defined next.

⟨*MathVec Template Method Declarations*⟩+≡
```
mathvec( const mathvec< TT >& other );
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
mathvec< TT >::mathvec( const mathvec< TT >& other )
    : basevec< TT >( other )
{
}
```

One can also, craftily enough, assign a vector from a string representation of the vector. For example, one might assign a unit 3-vector using the string `"[3]< 1, 0, 0 >"`

⟨*MathVec Template Method Declarations*⟩+≡
```
mathvec( const char* str );
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
mathvec< TT >::mathvec( const char* str )
    : basevec< TT >( str )
{
}
```

34.2 Comparing Vectors

It will frequently be necessary to see if two vectors are equal. As it's less computationally expensive to check to see if vectors are unequal, the following code implements the not-equal operator and then simply negates the result of the not-equal operator to get the equal-to operator.

⟨*MathVec Template Method Declarations*⟩+≡
```
bool operator != ( const mathvec< TT >& other ) const;
bool operator == ( const mathvec< TT >& other ) const;
```

For two vectors to be unequal, they must differ in size or in one of the components:

$$\vec{v} \neq \vec{w} \leftrightarrow (\dim(\vec{v}) \neq \dim(\vec{w})) \vee \bigvee_i (v_i \neq w_i)$$

⟨*MathVec Template Methods*⟩+≡

```
template < typename TT >
bool
mathvec< TT >::operator != ( const mathvec< TT >& other ) const
{
    if ( this->size() != other.size() ) {
        return true;
    }

    unsigned int len( this->size() );

    for ( unsigned int ii=0; ii < len; ++ii ) {
        if ( (*this)[ ii ] != other[ ii ] ) {
            return true;
        }
    }

    return false;
}
```

For two vectors to be equal, they must not be unequal.

$$\vec{v} = \vec{w} \leftrightarrow \neg(\vec{v} \neq \vec{w})$$

⟨*MathVec Template Methods*⟩+≡

```
template < typename TT >
bool
mathvec< TT >::operator == ( const mathvec< TT >& other ) const
{
    return ! ( (*this) != other );
}
```

34.3 Scaling A Vector

One often needs to scale a vector. One can do this by multiplying by a constant: $\vec{v}' = \alpha\vec{v}$ or dividing by a constant: $\vec{v}' = \vec{v}/\beta = \frac{1}{\beta}\vec{v}$.

There is a method whereby one can scale the current vector by multiplying each component by a scalar.

⟨*MathVec Template Method Declarations*⟩+≡
```
const mathvec< TT >& operator *= ( const TT& alpha );
```

To scale a vector by a scalar, one simply multiplies each component of the vector by the scalar:

$$\alpha\vec{v} = \sum_i \alpha v_i \hat{v}_i$$

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >&
mathvec< TT >::operator *= ( const TT& alpha )
{
    unsigned int len( this->size() );

    for ( unsigned int ii=0; ii < len; ++ii ) {
        (*this)[ ii ] *= alpha;
    }

    return *this;
}
```

The above method is employed to implement methods whereby one can multiply a seperate vector by a constant. These methods use the copy constructor to copy the vector and then the method above to effect the scaling. There are methods here for pre- and post-multiplication by a scalar.

⟨*MathVec Template Friend Pre-Declarations*⟩≡
```
template < typename TT > const mathvec< TT > operator * (
    const TT& alpha, const mathvec< TT >& vv
);
```

⟨*MathVec Template Method Declarations*⟩+≡
```
friend const mathvec< TT > operator * <TT> (
        const TT& alpha, const mathvec< TT >& vv
    );
const mathvec< TT > operator * ( const TT& alpha ) const;
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >
operator * ( const TT& alpha, const mathvec< TT >& vv )
{
    mathvec< TT > result( vv );
    result *= alpha;

    return result;
}
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >
mathvec< TT >::operator * ( const TT& alpha ) const
{
    mathvec< TT > result( *this );
    result *= alpha;

    return result;
}
```

There is a method whereby one can scale the current vector by dividing each component by a scalar.

⟨*MathVec Template Method Declarations*⟩+≡
```
const mathvec< TT >& operator /= ( const TT& other );
```

To scale a vector by a dividing scalar, one simply divides each component of the vector by the scalar:

$$\vec{v}/\alpha = \sum_i (v_i/\alpha)\hat{v}_i$$

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >&
mathvec< TT >::operator /= ( const TT& alpha )
{
    unsigned int len( this->size() );

    TT zero( 0U );
    if ( alpha != zero ) {
        for ( unsigned int ii=0; ii < len; ++ii ) {
            (*this)[ ii ] /= alpha;
        }
    } else {
        for ( unsigned int ii=0; ii < len; ++ii ) {
            (*this)[ ii ] = zero;
        }
    }

    return *this;
}
```

This code only supports post-division of a vector by a scalar.

⟨*MathVec Template Method Declarations*⟩+≡
```
const mathvec< TT > operator / ( const TT& alpha );
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >
mathvec< TT >::operator / ( const TT& alpha )
{
    mathvec< TT > result( *this );
    result /= alpha;

    return result;
}
```

34.4 The Sum of Vectors

There are many occassions in a raytracer in which to add two vectors.

⟨*MathVec Template Method Declarations*⟩+≡
```
const mathvec< TT >& operator += ( const mathvec< TT >& other );
```

To sum two vectors, one simply sums the components:

$$\vec{v} + \vec{w} = \sum_i (v_i + w_i)\hat{v}_i = \sum_i (v_i + w_i)\hat{w}_i$$

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >&
mathvec< TT >::operator += ( const mathvec< TT >& other )
{
    unsigned int len( this->size() );

    for ( unsigned int ii=0; ii < len; ++ii ) {
        (*this)[ ii ] += other[ ii ];
    }

    return *this;
}
```

⟨*MathVec Template Method Declarations*⟩+≡
```
const mathvec< TT > operator + ( const mathvec< TT >& other ) const;
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >
mathvec< TT >::operator + ( const mathvec< TT >& other ) const
{
    mathvec< TT > result( *this );
    result += other;

    return result;
}
```

34.5 The Difference of Vectors

There are many occasions in a raytracer in which one subtracts two vectors.

⟨*MathVec Template Method Declarations*⟩+≡
```
const mathvec< TT >& operator -= ( const mathvec< TT >& other );
```

To subtract a vector from another, one simply subtracts its compenents from the corresponding components of the other.

$$\vec{v} - \vec{w} = \sum_i (v_i - w_i)\hat{v}_i = \sum_i (v_i - w_i)\hat{w}_i$$

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >&
mathvec< TT >::operator -= ( const mathvec< TT >& other )
{
    unsigned int len( this->size() );

    for ( unsigned int ii=0; ii < len; ++ii ) {
        (*this)[ ii ] -= other[ ii ];
    }

    return *this;
}
```

⟨*MathVec Template Method Declarations*⟩+≡
```
const mathvec< TT > operator - ( const mathvec< TT >& other ) const;
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >
mathvec< TT >::operator - ( const mathvec< TT >& other ) const
{
    mathvec< TT > result( *this );
    result -= other;

    return result;
}
```

34.6 The Elementwise Product of Vectors

To scale vectors by a different amount in each dimension, the raytracer uses elementwise products and quotients.

⟨*MathVec Template Method Declarations*⟩+≡
```
const mathvec< TT >& operator *= ( const mathvec< TT >& other );
```

To multiply two vectors elementwise, one simply multiplies the corresponding components.

$$\vec{v} \circ \vec{w} = \sum_i (v_i \circ w_i)\hat{v}_i = \sum_i (v_i \circ w_i)\hat{w}_i$$

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >&
mathvec< TT >::operator *= ( const mathvec< TT >& other )
{
    unsigned int len( this->size() );

    for ( unsigned int ii=0; ii < len; ++ii ) {
        (*this)[ ii ] *= other[ ii ];
    }

    return *this;
}
```

⟨*MathVec Template Method Declarations*⟩+≡
```
const mathvec< TT > operator * ( const mathvec< TT >& other ) const;
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
const mathvec< TT >
mathvec< TT >::operator * ( const mathvec< TT >& other ) const
{
    mathvec< TT > result( *this );
    result *= other;

    return result;
}
```

34.7 The Elementwise Quotient of Vectors

To scale vectors by a different amount in each dimension, the raytracer uses elementwise products and quotients.

⟨*MathVec Template Method Declarations*⟩+≡
```
const mathvec< TT >& operator /= ( const mathvec< TT >& other );
```

To divide one vector by another, one simply divides each component of the vector by the corresponding component of the other.

$$\vec{v}/\vec{w} = \sum_i (v_i/w_i)\hat{v}_i = \sum_i (v_i/w_i)\hat{w}_i$$

⟨*MathVec Template Methods*⟩+≡

```
template < typename TT >
const mathvec< TT >&
mathvec< TT >::operator /= ( const mathvec< TT >& other )
{
    TT zero( 0U );
    unsigned int len( this->size() );

    for ( unsigned int ii=0; ii < len; ++ii ) {
        if ( other[ ii ] != zero ) {
            (*this)[ ii ] /= other[ ii ];
        } else {
            (*this)[ ii ] = zero;
        }
    }

    return *this;
}
```

⟨*MathVec Template Method Declarations*⟩+≡

```
const mathvec< TT > operator / ( const mathvec< TT >& other ) const;
```

⟨*MathVec Template Methods*⟩+≡

```
template < typename TT >
const mathvec< TT >
mathvec< TT >::operator / ( const mathvec< TT >& other ) const
{
    mathvec< TT > result( *this );
    result /= other;

    return result;
}
```

34.8 The Dot Product of Two Vectors

To determine the angle between two vectors, one must use the dot-product of the vectors because:

$$cos\theta = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}| \cdot |\vec{w}|}$$

⟨*MathVec Template Method Declarations*⟩+≡

```
const TT operator ^ ( const mathvec< TT >& other ) const;
```

To calculate the dot product of two vectors one simply sums the products of the corresponding components.

$$\vec{v} \cdot \vec{w} = \sum_i v_i \cdot w_i$$

⟨*MathVec Template Methods*⟩+≡

```
template < typename TT >
const TT
mathvec< TT >::operator ^ ( const mathvec< TT >& other ) const
{
    unsigned int len( this->size() );
    TT result( 0U );

    for ( unsigned int ii=0; ii < len; ++ii ) {
        result += (*this)[ ii ] * other[ ii ];
    }

    return result;
}
```

34.9 Normalizing A Vector

Many operations in a raytracer are simplest if the vectors are normalized. As such, the following methods can be used to determine the magnitude of a vector and normalize the vector.

⟨*MathVec Template Method Declarations*⟩+≡

```
TT magnitude( void ) const;
mathvec< TT >& normalize( void );
```

The magnitude is simply the square root of the dot product of the vector with itself:

$$|\vec{v}| = \sqrt{\vec{v} \cdot \vec{v}}$$

⟨*MathVec Template Methods*⟩+≡

```
template < typename TT >
TT
mathvec< TT >::magnitude( void ) const
{
    return (TT)std::sqrt( (double)( (*this) ^ (*this) ) );
}
```

The magnitude is simply the square root of the dot product of the vector with itself:

$$\hat{v} = \frac{\vec{v}}{|\vec{v}|}$$

⟨*MathVec Template Methods*⟩+≡

```
template < typename TT >
mathvec< TT >&
mathvec< TT >::normalize( void )
{
    *this /= this->magnitude();
    return *this;
}
```

34.10 Orthogonalizing A Matrix

Rather than require all input matrices to be fully specified and orthogonal, this method allows the matrices to be resized and orthogonalized. It uses the Gramm–Schmidt process.

This is a summary of the Gramm–Schmidt process. Given linearly independent vectors $\vec{a}_0$, $\vec{a}_1$, $\vec{a}_2$, $\ldots \vec{a}_{n-1}$ and orthogonal vectors $\vec{v}_0$, $\vec{v}_1$, $\vec{v}_2$, $\ldots \vec{v}_{m-1}$, generate orthogonal vectors $\vec{y}_0$, $\vec{y}_1$, $\vec{y}_2$, $\ldots \vec{y}_{n-1}$ with the constraint that $\vec{y}_k = \vec{v}_k$ for $k < m$.

To accomplish this, set $\vec{y}_k = \vec{v}_k$ for $k = 0, 1, 2, \ldots m - 1$. Then,

$$\vec{y}_j = \vec{a}_j - \sum_{k=0}^{j-1} \frac{\vec{y}_k \cdot \vec{a}_j}{\vec{y}_k \cdot \vec{y}_k}$$

This gives almost the whole picture. In reality, the j subscript on the $\vec{y}$ items is somewhat independent of the j subscript on the $\vec{a}$ items. This happens when the $\vec{a}_j$ is not linearly independent of the vectors $\vec{y}_0$, $\vec{y}_1$, $\vec{y}_2$, $\ldots \vec{y}_{j-1}$. This results in the calculated $\vec{y}_j$ being a zero vector. When this happens, we need to recalculate $\vec{y}_j$ using the next $\vec{a}$. So, the j subscript on the $\vec{a}$ may increment faster than it does on the $\vec{y}$.

In this scenario, all of the $\vec{y}_k$ are unit vectors, so the denominator of the fraction is always one. Additionally, in this scenario, all of the $\vec{a}_j$ are columns of the identity matrix, so the numerator is easy to calculate.

⟨*MathVec Template Function Declarations*⟩≡

```
template < typename TT > bool orthogonalize(
    basevec< mathvec< TT > >& matrix,
    unsigned int dims
);
```

⟨*MathVec Template Functions*⟩≡

```
template < typename TT >
bool orthogonalize(
    basevec< mathvec< TT > >& matrix,
    unsigned int dims
)
{
    unsigned int originalSize = matrix.size();
    ⟨MathVec orthogonalize – rework given vectors⟩
    ⟨MathVec orthogonalize – fabricate remaining vectors⟩
    return true;
}
```

⟨*MathVec orthogonalize – rework given vectors*⟩≡
```
for ( unsigned int ii=0; ii < originalSize; ++ii ) {
    matrix[ ii ].resize( dims, 0.0 );
    ⟨MathVec orthogonalize – make perpendicular to preceding vectors⟩
    ⟨MathVec orthogonalize – normalize vector⟩
    if ( zero ) {
        return false;
    }
}
```

⟨*MathVec orthogonalize – make perpendicular to preceding vectors*⟩≡
```
mathvec< double > vv( matrix[ ii ] );
for ( unsigned int jj=0; jj < ii; ++jj ) {
    matrix[ ii ] -= ( vv ^ matrix[ jj ] ) * matrix[ jj ];
}
```

⟨*MathVec orthogonalize – normalize vector*⟩≡
```
double mag = matrix[ ii ].magnitude();
bool zero = ( mag < 0.000001 );
if ( ! zero ) {
    matrix[ ii ] /= mag;
}
```

⟨*MathVec orthogonalize – fabricate remaining vectors*⟩≡
```
matrix.resize( dims );
unsigned int currentGuessColumn = 0;
for ( unsigned int ii=originalSize; ii < dims; ++ii ) {
    matrix[ ii ].resize( dims, 0.0 );
    ⟨MathVec orthogonalize – make guess⟩
    ⟨MathVec orthogonalize – make perpendicular to preceding vectors⟩
    ⟨MathVec orthogonalize – normalize vector⟩
    if ( zero ) {
        --ii;
    }
}
```

⟨*MathVec orthogonalize – make guess*⟩≡
```
matrix[ ii ] *= 0.0;
matrix[ ii ][ currentGuessColumn++ ] = 1.0;
```

34.11 Orthogonalizing A Matrix

It is comparatively easy to transpose a matrix.

⟨*MathVec Template Function Declarations*⟩+≡
```
template < typename TT > void transpose(
    basevec< mathvec< TT > >& matrix
);
```

⟨*MathVec Template Functions*⟩+≡
```
template < typename TT >
void transpose(
    basevec< mathvec< TT > >& matrix
)
{
    unsigned int originalSize = matrix.size();
    for ( unsigned int ii=0; ii < originalSize-1; ++ii ) {
        for ( unsigned int jj=ii+1; jj < originalSize; ++jj ) {
            TT tt = matrix[ ii ][ jj ];
            matrix[ ii ][ jj ] = matrix[ jj ][ ii ];
            matrix[ jj ][ ii ] = tt;
        }
    }
}
```

34.12 Rotating A Vector

⟨*MathVec Template Method Declarations*⟩+≡
```
mathvec< TT >& rotate( const basevec< mathvec< TT > >& matrix );
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
mathvec< TT >&
mathvec< TT >::rotate( const basevec< mathvec< TT > >& matrix )
{
    mathvec< TT >& res( *this );
    mathvec< TT > orig( *this );

    for ( unsigned int ii=0; ii < this->size(); ++ii ) {
        TT value = 0;
        for ( unsigned int jj=0; jj < this->size(); ++jj ) {
            value += orig[ jj ] * matrix[ jj ][ ii ];
        }
        res[ ii ] = value;
    }


    return res;
}
```

34.13 Antirotating A Vector

⟨*MathVec Template Method Declarations*⟩+≡
```
mathvec< TT >& antirotate( const basevec< mathvec< TT > >& matrix );
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
mathvec< TT >&
mathvec< TT >::antirotate( const basevec< mathvec< TT > >& matrix )
{
    mathvec< TT >& res( *this );
    mathvec< TT > orig( *this );

    for ( unsigned int ii=0; ii < this->size(); ++ii ) {
        const mathvec< TT >& col( matrix[ ii ] );
        TT value = 0;
        for ( unsigned int jj=0; jj < this->size(); ++jj ) {
            value += orig[ jj ] * col[ jj ];
        }
        res[ ii ] = value;
    }

    return res;
}
```

34.14 Solving A System Of Linear Equations

This is Gaussian elimination with partial pivoting as described in *Numerical Methods and Analysis* by James L. Buchanan and Peter R. Turner.

⟨*MathVec Template Method Declarations*⟩+≡
```
mathvec< TT > solve( const basevec< mathvec< TT > >& _AA );
```

⟨*MathVec Template Methods*⟩+≡
```
template < typename TT >
mathvec< TT >
mathvec< TT >::solve( const basevec< mathvec< TT > >& _AA )
{
    unsigned int nn = this->size();
    mathvec< TT > xx( nn );

    ⟨MathVec solve – initialize⟩

    for ( unsigned int jj=0; jj < nn-1; ++jj ) {
        ⟨MathVec solve – find pivot⟩
        ⟨MathVec solve – forward elimination⟩
    }

    ⟨MathVec solve – back substitution⟩

    return xx;
}
```

⟨*MathVec solve – initialize*⟩≡
```
mathvec< unsigned int > perm( nn );
for ( unsigned int ii=0; ii < nn; ++ii ) {
    perm[ ii ] = ii;
}

basevec< mathvec< TT > > AA( _AA );
mathvec< TT > bb( *this );

static const TT zero = (TT)0;
static const TT minusOne = (TT)-1;
```

⟨*MathVec solve – find pivot*⟩≡

```
TT pivot = AA[ perm[ jj ] ][ jj ];
if ( pivot < zero ) {
    pivot *= minusOne;
}
unsigned int piv = jj;
unsigned int ptmp = perm[ jj ];
for ( unsigned int ii=jj+1; ii < nn; ++ii ) {
    TT cur = AA[ perm[ ii ] ][ jj ];
    if ( cur < zero ) {
        cur *= minusOne;
    }
    if ( cur > pivot ) {
        pivot = cur;
        piv = ii;
    }
}

perm[ jj ] = perm[ piv ];
perm[ piv ] = ptmp;

if ( (double)pivot < 0.000001 ) {
    return xx;
}
```

⟨*MathVec solve – forward elimination*⟩≡

```
unsigned int pj = perm[ jj ];
const mathvec< TT >& baseRow( AA[ pj ] );
for ( unsigned int ii=jj+1; ii < nn; ++ii ) {
    unsigned int pi = perm[ ii ];
    mathvec< TT >& curRow( AA[ pi ] );
    TT mm = curRow[ jj ] / baseRow[ jj ];
    curRow[ jj ] = zero;
    bb[ pi ] -= mm * bb[ pj ];
    for ( unsigned int kk=jj+1; kk < nn; ++kk ) {
        curRow[ kk ] -= mm * baseRow[ kk ];
    }
}
```

⟨*MathVec solve – back substitution*⟩≡
```
for ( unsigned int kk=0; kk < nn; ++kk ) {
    unsigned int ii = ( nn - 1 ) - kk;
    unsigned int pi = perm[ ii ];
    const mathvec< TT >& curRow( AA[ pi ] );

    xx[ ii ] = bb[ pi ];
    for ( unsigned int jj=ii+1; jj < nn; ++jj ) {
        xx[ ii ] -= curRow[ jj ] * xx[ jj ];
    }
    xx[ ii ] /= curRow[ ii ];
}
```

34.15 The Class Definition

This class inherits directly from the `vector<>` template class in the standard template library. It includes all of the methods declared above. And, it includes some version identification information.

⟨*MathVec Template Pre-Declaration*⟩≡
```
template < typename TT > class mathvec;
```

⟨*MathVec Template Declaration*⟩≡
```
template < typename TT >
class mathvec : public basevec< TT > {
public:
    ⟨MathVec Template Method Declarations⟩
public:
    static const char* id;
};
```

34.16 Source Files

The following sections assemble the source files for the `mathvec` template class from the chunks above.

34.16.1 mathvec.h

The header file the `mathvec` template class includes the `iostream`, `vector`, and `cmath` headers from the standard C++ libraries and then incorporates the template defined in 34.15 above. In order to make templates work out alright, this file also includes the implementations of the template methods. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*mathvec.h*⟩≡

```
#ifndef _NKLEIN_MATHVEC_H_
#define _NKLEIN_MATHVEC_H_ 1

#include <iostream>
#include <vector>
#include <cmath>
#include "basevec.h"

    ⟨MathVec Template Pre-Declaration⟩
    ⟨MathVec Template Friend Pre-Declarations⟩

    ⟨MathVec Template Declaration⟩
    ⟨MathVec Template Methods⟩

    ⟨MathVec Template Function Declarations⟩
    ⟨MathVec Template Functions⟩

#endif /*_NKLEIN_MATHVEC_H_*/
```

34.16.2 mathvec.cc

The implementation file for the `mathvec` template class instantiates the types of vectors used in the raytracer.

⟨*mathvec.cc*⟩≡

```
#include "mathvec.h"

template< typename TT >
    const char* mathvec<TT>::id = "NKVERSION: mathvec<>( 2.1.2007.05.17 )";

⟨MathVec Template Instantiations⟩
```

Chapter 35

Basic Vectors

This template class extends the `vector<>` class from the standard template library to include the small amount of code needed to read and write the vector format used in this raytracer. This class is extended by the `mathvec<>` class.

The code in this raytracer uses a variety of types of these vectors. They will be instantiated in the places where the element classes are defined.

⟨*BaseVec Template Instantiations*⟩≡

```
template class basevec< double >;
template class basevec< unsigned int >;
```

35.1 Creating Vectors

There are several ways to create vectors. The most basice way is using the default constructor which simply initializes a zero-length vector.

⟨*BaseVec Template Method Declarations*⟩≡

```
basevec();
```

⟨*BaseVec Template Methods*⟩≡

```
template < typename TT >
basevec< TT >::basevec() : std::vector< TT >()
{
}
```

One could initialize a vector from a given array and given length.

⟨*BaseVec Template Method Declarations*⟩+≡

```
basevec( const TT array[], unsigned int len );
```

⟨*BaseVec Template Methods*⟩+≡
```
template < typename TT >
basevec< TT >::basevec( const TT array[], unsigned int len )
    : std::vector< TT >( len )
{
    for ( unsigned int ii=0; ii < len; ++ii ) {
        (*this)[ ii ] = array[ ii ];
    }
}
```

One could initialize a vector from a given value and given length.

⟨*BaseVec Template Method Declarations*⟩+≡
```
basevec( const TT value, unsigned int len );
```

⟨*BaseVec Template Methods*⟩+≡
```
template < typename TT >
basevec< TT >::basevec( const TT value, unsigned int len )
    : std::vector< TT >( len )
{
    for ( unsigned int ii=0; ii < len; ++ii ) {
        (*this)[ ii ] = value;
    }
}
```

One could copy an existing vector. This method makes use of the assignment operator defined next.

⟨*BaseVec Template Method Declarations*⟩+≡
```
basevec( const basevec< TT >& other );
```

⟨*BaseVec Template Methods*⟩+≡
```
template < typename TT >
basevec< TT >::basevec( const basevec< TT >& other )
    : std::vector< TT >()
{
    *this = other;
}
```

One could also assign a vector from a given vector.

⟨*BaseVec Template Method Declarations*⟩+≡
```
const basevec< TT >& operator = ( const basevec< TT >& other );
```

⟨*BaseVec Template Methods*⟩+≡
```
template < typename TT >
const basevec< TT >&
basevec< TT >::operator = ( const basevec< TT >& other )
{
    unsigned int len( other.size() );
    this->resize( len );

    for ( unsigned int ii=0; ii < len; ++ii ) {
        (*this)[ ii ] = other[ ii ];
    }

    return *this;
}
```

One can also, craftily enough, assign a vector from a string representation of the vector. For example, one might assign a unit 3-vector using the string `"[3]< 1, 0, 0 >"`

⟨*BaseVec Template Method Declarations*⟩+≡
```
basevec( const char* str );
```

⟨*BaseVec Template Methods*⟩+≡
```
template < typename TT >
basevec< TT >::basevec( const char* str )
    : std::vector< TT >()
{
    std::istringstream in( str );
    in >> (*this);
    if ( ! in.good() ) {
        this->resize( 0 );
    }
}
```

35.2 Input and Output of Vectors

As seen above, this raytracer presents vectors as a comma-separated list of the component values surrounded by greater-than and less-than symbols and preceded by the length of the vector in square brackets. The following routines read and write them.

⟨*BaseVec Template Friend Pre-Declarations*⟩≡
```
template < typename TT >
    std::istream& operator >> ( std::istream&, basevec< TT >& );
template < typename TT >
    std::ostream& operator << ( std::ostream&, const basevec< TT >& );
```

⟨*BaseVec Template Method Declarations*⟩+≡
```
friend std::istream& operator >> <TT> ( std::istream&, basevec< TT >& );
friend std::ostream& operator << <TT> ( std::ostream&, const basevec< TT >& );
```

The input routine sets the input stream to skip white space. Then, it plucks off the opening bracket, the vector length, the closing bracket, and the greater-than sign. It initializes a zero vector of the given size. Then, it proceeds to read through the components being careful to pluck out the commas. Then, it continues plucking off any bogus characters until it comes to the less-than sign.

⟨*BaseVec Template Methods*⟩+≡
```
template < typename TT >
std::istream&
operator >> ( std::istream& in, basevec< TT >& vv )
{
    char bogus;
    unsigned int len;

    in.setf( std::ios::skipws );

    in >> bogus;                // "["
    in >> len;
    in >> bogus >> bogus;       // "]" "<"

    TT zero( 0U );
    vv.resize( len, zero );

    for (unsigned ii=0; ii < len; ++ii) {
        in >> vv[ii];

        if (ii < len-1) {
            in >> bogus; // ","
        }
    }

    while (bogus != '>' && in.good() ) {
        in >> bogus; // ">"
    }

    return in;
}
```

The output routine emits the opening bracket, the length of the vector, the closing bracket, the greater-than sign, and a bit of white space. Then, it emits each component value being careful to add a comma and some whitespace between components. Then, it emits the less-than sign.

⟨*BaseVec Template Methods*⟩+≡

```
template < typename TT >
std::ostream&
operator << ( std::ostream& out, const basevec< TT >& vv )
{
    unsigned int len = vv.size();

    out.put( '[' );
    out << len;
    out.put( ']' ).put( '<' ).put( ' ' );

    for (unsigned int ii=0; ii < len; ++ii) {
        out << vv[ii];

        if (ii < len-1) {
            out.put( ',' ).put( ' ' );
        }
    }

    out.put( ' ' ).put( '>' );

    return out;
}
```

35.3 The Class Definition

This class inherits directly from the `vector<>` template class in the standard template library. It includes all of the methods declared above. And, it includes some version identification information.

⟨*BaseVec Template Pre-Declaration*⟩≡

```
template < typename TT > class basevec;
```

⟨*BaseVec Template Declaration*⟩≡
```
template < typename TT >
class basevec : public std::vector< TT > {
public:
    ⟨BaseVec Template Method Declarations⟩
public:
    static const char* id;
};
```

35.4 Source Files

The following sections assemble the source files for the `basevec` template class from the chunks above.

35.4.1 basevec.h

The header file the `basevec` template class includes the `iostream` and `vector` headers from the standard C++ libraries and then incorporates the template defined in 35.3 above. In order to make templates work out alright, this file also includes the implementations of the template methods. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*basevec.h*⟩≡
```
#ifndef _NKLEIN_BASEVEC_H_
#define _NKLEIN_BASEVEC_H_ 1

#include <iostream>
#include <sstream>
#include <vector>

    ⟨BaseVec Template Pre-Declaration⟩
    ⟨BaseVec Template Friend Pre-Declarations⟩

    ⟨BaseVec Template Declaration⟩
    ⟨BaseVec Template Methods⟩
#endif /*_NKLEIN_BASEVEC_H_*/
```

35.4.2 basevec.cc

Herein, the instantiations of the `basevec` class are included. The implementation requires the definition of the class itself.

⟨*basevec.cc*⟩≡
```
#include "basevec.h"

template< typename TT >
    const char* basevec<TT>::id = "NKVERSION: basevec<>( 2.1.2007.05.17 )";

⟨BaseVec Template Instantiations⟩
```

Chapter 36

Intersection

The `intersection` class represents a single intersection between a ray and an object. The intersection class specifies the position and the normal at the point where the ray and object intersect, and whether the ray hit the object from the inside or not.

⟨*Intersection Member Variables*⟩≡
```
mathvec< double > position;
mathvec< double > normal;
mathvec< double > direction;
bool inside;
```

Additionally, the class holds the position, normal, and direction in the object's coordinates so that if the color of the object needs to be checked, the raytracer will not have to re-rotate the position and normal into the object's frame of reference.

⟨*Intersection Member Variables*⟩+≡
```
mathvec< double > objectPosition;
mathvec< double > objectNormal;
mathvec< double > objectDirection;
```

The class also holds a pointer to the color of the object and a record of the distance.

⟨*Intersection Member Variables*⟩+≡
```
const color* objectColor;
double distance;
```

Additionally, it contains a buffer into which the object color can be evaluated.

⟨*Intersection Member Variables*⟩+≡
```
mathvec< double > pixel;
```

36.1 Constructor

The default constructor does nothing except zero the color pointer.

⟨*Intersection Method Declarations*⟩≡
```
intersection( void );
```

⟨*Intersection Methods*⟩≡
```
intersection::intersection( void ) : objectColor( 0 )
{
}
```

36.2 Setting The Intersection Attributes

The following methods can be used to set the attributes of this intersection point. These attributes are in the global frame of reference.

⟨*Intersection Method Declarations*⟩+≡
```
inline void setPosition( const mathvec< double >& pp ) {
    this->position = pp;
};
inline void setNormal( const mathvec< double >& nn ) {
    this->normal = nn;
};
inline void setDirection( const mathvec< double >& dd ) {
    this->direction = dd;
};
inline void setInside( void ) {
    this->inside = true;
};
inline void setOutside( void ) {
    this->inside = false;
};
inline void setDistance( double dd ) {
    this->distance = dd;
};
```

36.3 Getting The Intersection Attributes

The following methods can be used to retrieve the attributes of this intersection point. These attributes are in the global frame of reference.

⟨*Intersection Method Declarations*⟩+≡

```
inline const mathvec< double >& getPosition( void ) const {
    return this->position;
};
inline const mathvec< double >& getNormal( void ) const {
    return this->normal;
};
inline const mathvec< double >& getDirection( void ) const {
    return this->direction;
};
inline bool isInside( void ) const {
    return this->inside;
}
inline bool isOutside( void ) const {
    return ! this->inside;
}
inline const color* getColor( void ) const {
    return this->objectColor;
}
inline double getDistance( void ) const {
    return this->distance;
}
```

36.4 Setting The Object Intersection Attributes

The following methods can be used to set the attributes of this intersection point. These attributes are in the object's frame of reference.

⟨*Intersection Method Declarations*⟩+≡

```
inline void setObjectPosition( const mathvec< double >& pp ) {
    this->objectPosition = pp;
};
inline void setObjectNormal( const mathvec< double >& nn ) {
    this->objectNormal = nn;
};
inline void setObjectDirection( const mathvec< double >& dd ) {
    this->objectDirection = dd;
};
inline void setObjectColor( const color* cc ) {
    this->objectColor = cc;
};
```

36.5 Updating The Object Intersection Attributes

At the time this intersection is filled in, only the coordinates in the object's reference frame are known. This method is used to re-orient the intersection attributes back toward the global reference frame.

⟨*Intersection Method Declarations*⟩+≡

```
void reorient(
    const mathvec< double >& scale,
    const basevec< mathvec< double > >& orientation,
    const mathvec< double >& center
);
```

⟨*Intersection Methods*⟩+≡
```
void
intersection::reorient(
    const mathvec< double >& scale,
    const basevec< mathvec< double > >& orientation,
    const mathvec< double >& center
) {
    this->position *= scale;
    this->position.rotate( orientation );
    this->position += center;

    this->direction *= scale;
    this->direction.rotate( orientation );
    this->direction.normalize();

    this->normal /= scale;
    this->normal.rotate( orientation );
    this->normal.normalize();
}
```

If there is a color specified in an object which contains the object intersected, then the color gets overridden. Along with the color, the position, normal, and direction must also be updated. As such, container objects should reorient objects using the above method before recoloring them.

⟨*Intersection Method Declarations*⟩+≡
```
void recolor(
    const color& cc
);
```

⟨*Intersection Methods*⟩+≡
```
void
intersection::recolor(
    const color& cc
) {
    this->setObjectColor( &cc );
    this->setObjectPosition( this->getPosition() );
    this->setObjectNormal( this->getNormal() );
    this->setObjectDirection( this->getDirection() );
}
```

36.6 Querying The Intersection Color

The following method allows one to query the color of the intersection from the intersection's point of view.

⟨*Intersection Method Declarations*⟩+≡
```
void evaluate(
    double contribution, unsigned int height,
    mathvec< double >& pp
) const;
```

⟨*Intersection Methods*⟩+≡
```
void
intersection::evaluate(
    double contribution, unsigned int height,
    mathvec< double >& pp
) const {
    if ( this->pixel.size() == 0 ) {
        this->objectColor->evaluate(
            this->objectPosition, this->objectNormal, this->objectDirection,
            contribution, height, *(mathvec< double >*)&this->pixel
        );
    }
    pp = this->pixel;
}
```

36.7 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Intersection Class Declaration*⟩≡
```
class intersection {
protected:
    ⟨Intersection Member Variables⟩
public:
    ⟨Intersection Method Declarations⟩
public:
    static const char* id;
};
```

36.8 Source Files

The following sections assemble the source files for the `intersection` class from the chunks above.

36.8.1 intersection.h

The header file the `intersection` class includes the `iostream` and `string` headers from the standard C++ libraries and the header file for the `mathvec<>` class and then incorporates the class defined in 36.7 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*intersection.h*⟩≡
```
#ifndef _NKLEIN_INTERSECTION_H_
#define _NKLEIN_INTERSECTION_H_ 1

#include <iostream>
#include <string>
#include <map>
#include "color.h"

    ⟨Intersection Class Declaration⟩

#endif /*_NKLEIN_INTERSECTION_H_*/
```

36.8.2 intersection.cc

Herein, the implementations of the methods of the `intersection` class are included as well as the the declarations and implementations of the derived classes.

⟨*intersection.cc*⟩≡
```
#include <string>
#include <sstream>
#include <math.h>
#include "input.h"
#include "intersection.h"

const char* intersection::id = "NKVERSION: intersection( 2.1.2007.05.17 )";

⟨Intersection Methods⟩
```

Chapter 37

Thread

This section implements the `thread` class which allows rendering to take place in multiple threads. The `thread` class is a singleton shared throughout the raytracer. It is used mainly within the `universe`'s `evaluate()` method to parallelize that.

To serve its purposes, the `thread` class needs something to actually do the work required in the thread. To this end, it uses a specialized base-class called `work`. A class hoping to get work done in a thread will submit an instance of the `work` class to the `thread` singleton.

⟨*Thread Public Member Types*⟩≡
  ⟨*Thread Work Class Declaration*⟩
 The `work` class is defined in §38.

That work is passed on to an instance of the `worker` class if there is one available. If there is not, then the `work` is performed inline.

⟨*Thread Member Types*⟩≡
  ⟨*Thread Worker Class Declaration*⟩
 The `worker` class is defined in §39.

The `thread` class keeps a list of available workers.

⟨*Thread Member Variables*⟩≡
```
#ifdef USE_PTHREADS
pthread_mutex_t mutex;
#endif
std::deque< worker* > workers;
```

The `thread` class is a singleton. This method passes along the pointer to this instance.

⟨*Thread Method Declarations*⟩≡
```
static thread* instance( void );
```

⟨*Thread Methods*⟩≡
```
thread*
thread::instance( void ) {
    static thread mgr;
    return &mgr;
};
```

37.1 Constructor

The default constructor initializes the mutex.

⟨*Thread Method Declarations*⟩+≡
```
thread( void );
```

⟨*Thread Methods*⟩+≡
```
thread::thread( void )
{
#ifdef USE_PTHREADS
    ::pthread_mutex_init( &this->mutex, 0 );
#endif
}
```

37.2 Destructor

The default constructor clears out all of the workers.

⟨*Thread Method Declarations*⟩+≡
```
~thread( void );
```

⟨*Thread Methods*⟩+≡
```
thread::~thread( void )
{
    this->setWorkerCount( 0 );
#ifdef USE_PTHREADS
    ::pthread_mutex_destroy( &this->mutex );
#endif
}
```

37.3 Setting The Number Of Workers

To increase the number of workers, this method simply allocates new worker threads. The workers are defined in §39. The workers add themselves to this manager from within their run-loop. To decrease the number of workers, the extra workers are removed and deleted.

⟨*Thread Method Declarations*⟩+≡
```
void setWorkerCount( unsigned int nn );
```

⟨*Thread Methods*⟩+≡
```
void
thread::setWorkerCount( unsigned int nn )
{
    std::vector< worker* > tmp;

    this->lock();
        unsigned int original = this->workers.size();
        if ( original < nn ) {
            for ( unsigned int ii=original; ii < nn; ++ii ) {
                new worker();
            }
        } else if ( original > nn ) {
            tmp.resize( original - nn );

            for ( unsigned int ii=nn; ii < original; ++ii ) {
                tmp[ ii - nn ] = this->workers.front();
                this->workers.pop_front();
            }
        }

#ifdef USE_PTHREADS
        ::pthread_setconcurrency( nn+1 );
#endif
    this->unlock();

    for ( unsigned int ii=0; ii < tmp.size(); ++ii ) {
        delete tmp[ ii ];
    }
}
```

37.4 Make A Worker Available Again

When a worker has completed its work, it invokes this method on that manager to put itself back in the queue of workers.

⟨*Thread Protected Method Declarations*⟩≡
```
void makeWorkerAvailable( worker* ww );
```

⟨*Thread Methods*⟩+≡
```
void
thread::makeWorkerAvailable( worker* ww )
{
    this->lock();
        this->workers.push_back( ww );
    this->unlock();
}
```

37.5 Starting Work

A piece of work invokes this method in order to get itself going. This method attempts to pull an available worker from the queue. If it succeeds, then this work is assigned to that worker and started. Otherwise, the work is done entirely in-line.

⟨*Thread Method Declarations*⟩+≡
```
void start( thread::work* ww );
```

⟨*Thread Methods*⟩+≡
```
void
thread::start( thread::work* ww )
{
    worker* wr = 0;

    this->lock();
        if ( this->workers.size() > 0 ) {
            wr = this->workers.front();
            this->workers.pop_front();
        }
    this->unlock();

    if ( wr == 0 ) {
        ww->perform();
        ww->finish();
    } else {
        wr->lock();
            wr->setWork( ww );
            wr->wakeup();
        wr->unlock();
    }
}
```

37.6 Locking And Unlocking The Mutex

These methods are used to lock and unlock the manager. They simply invoke the corresponding POSIX mutex methods.

⟨*Thread Method Declarations*⟩+≡
```
void lock( void );
void unlock( void );
```

⟨*Thread Methods*⟩+≡
```
void
thread::lock( void )
{
#ifdef USE_PTHREADS
    ::pthread_mutex_lock( &this->mutex );
#endif
}
```

⟨*Thread Methods*⟩+≡
```
void
thread::unlock( void )
{
#ifdef USE_PTHREADS
    ::pthread_mutex_unlock( &this->mutex );
#endif
}
```

37.7 Waiting On A Condition

This method is invoked in the worker's main loop to wait for work to become available to it. The worker's `wakeup()` method signals that the worker's thread should now have some work to do.

⟨*Thread Method Declarations*⟩+≡
```
void wait( void* cond );
```

⟨*Thread Methods*⟩+≡
```
void
thread::wait( void* cond )
{
#ifdef USE_PTHREADS
    ::pthread_cond_wait( (pthread_cond_t*)cond, &this->mutex );
#endif
}
```

37.8 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Thread Class Declaration*⟩≡
```
class thread {
public:
⟨Thread Public Member Types⟩
⟨Thread Member Types⟩
protected:
    ⟨Thread Member Variables⟩
public:
    ⟨Thread Method Declarations⟩
    friend class thread::worker;
protected:
    ⟨Thread Protected Method Declarations⟩
public:
    static const char* id;
};
```

37.9 Source Files

The following sections assemble the source files for the `thread` class from the chunks above.

37.9.1 nkthread.h

The header file the `thread` class includes the `deque` header from the standard C++ libraries and the header file for the POSIX threads interface. Then, it goes on to include the thread manager class defined in 37.8 above. All of this is wrapped in syntactic sugar to avoid problems should this file be included more than once.

⟨*nkthread.h*⟩≡

```
#ifndef _NKLEIN_THREAD_H_
#define _NKLEIN_THREAD_H_ 1

#include <deque>

#ifdef USE_PTHREADS
#  include <pthread.h>
#endif

⟨Thread Class Declaration⟩

#endif /*_NKLEIN_THREAD_H_*/
```

37.9.2 nkthread.cc

Herein, the implementations of the methods of the `thread` class are included as well as the the declarations and implementations of its support classes.

⟨*nkthread.cc*⟩≡

```
#include <iostream>
#include <vector>
#include "nkthread.h"

⟨Thread Worker Loop Starter Declaration⟩
⟨Thread Work Methods⟩
⟨Thread Worker Methods⟩

const char* thread::id = "NKVERSION: thread( 2.1.2007.05.17 )";
const char* thread::work::id = "NKVERSION: thread::work( 2.1.2007.05.17 )";
const char* thread::worker::id = "NKVERSION: thread::worker( 2.1.2007.05.17 )";

⟨Thread Methods⟩
```

Chapter 38

Work

This section implements the `work` class used by the thread. The class contains a mutual exclusion lock and a condition. The mutex and condition are used to wait until the work has been performed.

⟨*Thread Work Member Variables*⟩≡

```
#ifdef USE_PTHREADS
pthread_mutex_t mutex;
pthread_cond_t condition;
#endif
bool done;
```

38.1 Constructor

The constructor initializes the mutex and condition.

⟨*Thread Work Method Declarations*⟩≡

```
work( void );
```

⟨*Thread Work Methods*⟩≡

```
thread::work::work( void ) : done( true )
{
#ifdef USE_PTHREADS
    ::pthread_mutex_init( &this->mutex, 0 );
    ::pthread_cond_init( &this->condition, 0 );
#endif
}
```

38.2 Destructor

The destructor destroys the mutex and condition.

⟨*Thread Work Method Declarations*⟩+≡
```
virtual ~work( void );
```

⟨*Thread Work Methods*⟩+≡
```
thread::work::~work( void )
{
#ifdef USE_PTHREADS
    ::pthread_cond_destroy( &this->condition );
    ::pthread_mutex_destroy( &this->mutex );
#endif
}
```

38.3 Performing The Work

To actually perform the work, the derived class must override the `perform()` method.

⟨*Thread Work Method Declarations*⟩+≡
```
virtual void perform( void ) = 0;
```

To get things going, the caller should invoke the `start()` method.

⟨*Thread Work Method Declarations*⟩+≡
```
void start( void );
```

⟨*Thread Work Methods*⟩+≡
```
void
thread::work::start( void )
{
    this->done = false;

    thread* mgr = thread::instance();
    mgr->start( this );
}
```

One can wait for the work to be completed.

⟨*Thread Work Method Declarations*⟩+≡
```
void wait( void );
```

⟨*Thread Work Methods*⟩+≡
```
void
thread::work::wait( void )
{
#ifdef USE_PTHREADS
    ::pthread_mutex_lock( &this->mutex );
        while ( !this->done ) {
            ::pthread_cond_wait( &this->condition, &this->mutex );
        }
    ::pthread_mutex_unlock( &this->mutex );
#endif
}
```

One can declare that the work is finished. This method notifies all parties which are waiting on the condition that the work has been completed.

⟨*Thread Work Method Declarations*⟩+≡
```
void finish( void );
```

⟨*Thread Work Methods*⟩+≡
```
void
thread::work::finish( void )
{
#ifdef USE_PTHREADS
    ::pthread_mutex_lock( &this->mutex );
#endif
        this->done = true;
#ifdef USE_PTHREADS
        ::pthread_cond_broadcast( &this->condition );
    ::pthread_mutex_unlock( &this->mutex );
#endif
}
```

38.4 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Thread Work Class Declaration*⟩≡

```
class work {
protected:
⟨Thread Work Member Variables⟩
public:
    ⟨Thread Work Method Declarations⟩
public:
    static const char* id;
};
```

Chapter 39

Worker

This section implements the `worker` class used by the thread. The class contains a condition variable used to sit and wait for more work.

⟨*Thread Worker Member Variables*⟩≡
```
#ifdef USE_PTHREADS
pthread_mutex_t mutex;
pthread_cond_t condition;
pthread_t myThread;
#endif
```

The class also contains a pointer to the work it is supposed to do.

⟨*Thread Worker Member Variables*⟩+≡
```
thread::work* todo;
bool done;
```

39.1 Constructor

The constructor initializes the condition variable and the work pointer. It also creates a thread for this worker.

⟨*Thread Worker Method Declarations*⟩≡
```
worker( void );
```

⟨*Thread Worker Methods*⟩≡
```
thread::worker::worker( void ) : todo( 0 ), done( false )
{
#ifdef USE_PTHREADS
    ::pthread_mutex_init( &this->mutex, 0 );
    ::pthread_cond_init( &this->condition, 0 );
    ::pthread_create( &this->myThread, 0, ThreadWorkerLoopStarter, this );
#endif
}
```

39.2 Destructor

The destructor signals that the thread should end. Then, it waits for the thread to end. Then, it destroys the condition.

⟨*Thread Worker Method Declarations*⟩+≡
```
virtual ~worker( void );
```

⟨*Thread Worker Methods*⟩+≡
```
thread::worker::~worker( void )
{
    this->lock();
        this->done = true;
        this->wakeup();
    this->unlock();

#ifdef USE_PTHREADS
    ::pthread_join( this->myThread, 0 );
    ::pthread_cond_destroy( &this->condition );
    ::pthread_mutex_destroy( &this->mutex );
#endif
}
```

39.3 Assigning Work

The manager uses this method to assign work to this worker.

⟨*Thread Worker Method Declarations*⟩+≡
```
void setWork( thread::work* ww ) {
    this->todo = ww;
};
```

39.4 Waking Up

Once the work has been assigned, this worker is awakened.

⟨*Thread Worker Method Declarations*⟩+≡
```
void wakeup( void );
```

⟨*Thread Worker Methods*⟩+≡
```
void
thread::worker::wakeup( void )
{
#ifdef USE_PTHREADS
    ::pthread_cond_signal( &this->condition );
#endif
}
```

⟨*Thread Worker Method Declarations*⟩+≡
```
void lock( void );
void wait( void );
void unlock( void );
```

⟨*Thread Worker Methods*⟩+≡
```
void
thread::worker::lock( void )
{
#ifdef USE_PTHREADS
    ::pthread_mutex_lock( &this->mutex );
#endif
}
```

⟨*Thread Worker Methods*⟩+≡
```
void
thread::worker::wait( void )
{
#ifdef USE_PTHREADS
    ::pthread_cond_wait( &this->condition, &this->mutex );
#endif
}
```

⟨*Thread Worker Methods*⟩+≡
```
void
thread::worker::unlock( void )
{
#ifdef USE_PTHREADS
    ::pthread_mutex_unlock( &this->mutex );
#endif
}
```

39.5 Loop For Work

This first part here is a simple trampoline method used to get the thread started.

⟨*Thread Worker Loop Starter Declaration*⟩≡
```
extern "C" void* ThreadWorkerLoopStarter( void* arg );
```

⟨*Thread Worker Methods*⟩+≡
```
void*
ThreadWorkerLoopStarter( void* arg )
{
    thread::worker* me = (thread::worker*)arg;
    return me->loop();
}
```

⟨*Thread Worker Method Declarations*⟩+≡
```
void* loop( void );
```

⟨*Thread Worker Methods*⟩+≡
```
void*
thread::worker::loop( void )
{
#ifdef USE_PTHREADS
    thread* mgr = thread::instance();

    for (;;) {
        this->todo = 0;
        mgr->makeWorkerAvailable( this );

        this->lock();
            do {
                if ( this->done ) {
                    this->unlock();
                    return 0;
                }
                this->wait();
            } while ( this->todo == 0 );
        this->unlock();

        this->todo->perform();
        this->todo->finish();
    }
#endif
    return 0;
}
```

39.6 The Class Definition

This class includes all of the methods declared above. And, it includes some version identification information.

⟨*Thread Worker Class Declaration*⟩≡
```
class worker {
protected:
⟨Thread Worker Member Variables⟩
public:
    ⟨Thread Worker Method Declarations⟩
public:
    static const char* id;
};
```

Chapter 40

Input Class

This input class allows one to use an input stream while tracking the brace-level and filtering out comments.

⟨*Input Data Members*⟩≡
```
std::istream& in;
unsigned int braceLevel;
```

40.1 Constructor

The constructor simply takes a reference to an input stream. It saves this reference and clears the brace level.

⟨*Input Method Declarations*⟩≡
```
input( std::istream& _in );
```

⟨*Input Methods*⟩≡
```
input::input( std::istream& _in ) : in( _in ), braceLevel( 0 )
{
}
```

40.2 Checking An Input Stream

Most items will read things in a normal way. They don't particularly care about brace-level. And, they want an `EOF` when they get to the end of the information about them. That's what this function does. It keeps track of the brace-level, sucks up comments, and returns false when we're at the end of the object.

⟨*Input Method Declarations*⟩+≡
```
bool check( bool _ignoreEverything = false );
```

This routine leaves the `istream` in a state such that the next time we call it, we will get the next useful thing. Particularly, it skips over comments and tracks the current level of braces.

⟨*Input Methods*⟩+≡

```
bool
input::check( bool _ignoreEverything )
{
    unsigned int startBraceLevel = this->braceLevel;

    for (;;) {
        int ch = in.peek();

        if ( ch < 0 ) {
            break;
        } else if ( ch == '{' ) {
            this->in.get();
            ++this->braceLevel;
        } else if ( ch == '}' ) {
            this->in.get();
            --this->braceLevel;
        } else if ( ch == ';' ) {
            if ( this->braceLevel == 0
            || ( _ignoreEverything && this->braceLevel == startBraceLevel ) ) {
                break;
            } else {
                this->in.get();
            }
        } else if ( ch == ',' ) {
            if ( this->braceLevel == 0 ) {
                break;
            } else {
                this->in.get();
            }
        } else if ( ch == '#' ) {
            while ( ch >= 0 && ch != '\n' ) {
                ch = this->in.get();
            }
        } else if ( isspace( ch ) ) {
            this->in.get();
        } else if ( ! _ignoreEverything ) {
            return true;
        } else {
            this->in.get();
        }
    }
    return false;
```

```
}
```

40.3 Ignoring Part Of An Input Stream

⟨*Input Method Declarations*⟩+≡
```
void ignore( const char* prefix, const char* token );
```

When an entity comes upon an unknown token, it usually wishes to ignore everything after up to and including the following semicolon.

⟨*Input Methods*⟩+≡
```
void
input::ignore( const char* prefix, const char* token )
{
    char ch;

    std::cerr << prefix << ": Ignoring " << token;
    while ( this->check( true ) ) {
        ch = this->in.get();
#ifndef NDEBUG
        std::cerr << ch;
#endif
    };
    ch = this->in.get();
#ifndef NDEBUG
    std::cerr << ch;
#endif
    std::cerr << std::endl;
}
```

40.4 Getting A Token

This routine reads in a token from the stream. If the token is longer than the buffer, the remaining portion of the token is sucked up, as well.

⟨*Input Method Declarations*⟩+≡
```
enum { MAX_TOKEN_LEN = 64 };
unsigned int get( char token[], unsigned int tokenLen );
```

⟨*Input Methods*⟩+≡

```
unsigned int
input::get( char token[], unsigned int tokenLen )
{
    if ( ! this->check() ) {
        return 0;
    }

    unsigned int len = 0;

    int ch = this->in.peek();

    while ( this->in.good() && !isspace( ch )
    && ch != ',' && ch != ';' && ch != '>'
    && len < tokenLen-1 ) {
        ch = this->in.get();
        *token++ = ch;
        ++len;
        ch = this->in.peek();
    }

    *token = '\0';

    while ( this->in.good() && !isspace( ch )
    && ch != ',' && ch != ';' && ch != '>' ) {
        ch = this->in.get();
        ch = this->in.peek();
    }

    return len;
}
```

40.5 Source Files

The following section assembles the header file and source file for the input routines from the chunks above.

40.5.1 input.h

The main routine requires no include files at the moment, but it will very soon.

⟨*input.h*⟩≡
```
#ifndef _NKLEIN_INPUT_H_
#define _NKLEIN_INPUT_H_ 1
#include <iostream>

class input {
protected:
    ⟨Input Data Members⟩
public:
    ⟨Input Method Declarations⟩
public:
    static const char* id;
};
#endif /* _NKLEIN_INPUT_H_ */
```

40.5.2 input.cc

The main routine requires no include files at the moment, but it will very soon.

⟨*input.cc*⟩≡
```
#include "input.h"

const char* input::id = "NKVERSION: input( 2.1.2007.05.17 )";

⟨Input Methods⟩
```

Chapter 41

Main Routine

The main routine of the raytracer is responsible for loading in the scene files from the command-line and rendering them. If no scenes are given on the command-line, it is assumed that one will be coming on standard-input.

⟨*Main Routine*⟩≡

```
int
main( int argc, const char* const argv[] )
{
    int ii;

    ⟨Main parse arguments⟩

    if ( ii < argc ) {
        ⟨Main read files specified on command-line⟩
    } else {
        ⟨Main read scene from standard-input⟩
    }

    return 0;
}
```

If the argument, does not start with a dash (`-'), then it must be a file name. The only recognized argument at the moment is "-t" which specifies the number of threads to place in the thread pool. Unfortunately, at the moment, using any threads slows the system down immensely for some reason. So, this is not a wholly useful option.

⟨*Main parse arguments*⟩≡
```
for ( ii=1; ii < argc; ++ii ) {
    std::string arg( argv[ ii ] );

    if ( arg[0] != '-' ) {
        break;
    } else if ( arg == "-t" ) {
        ⟨Main handle thread-count argument⟩
    } else {
        std::cerr << "Unknown argument: " << arg << std::endl;
        ⟨Main handle print usage statement⟩
        return __LINE__;
    }
}
```

The thread-count argument will be a simple integer specifying the number of threads to employ. Here, this code arbitrarily keep the count from exceeding 100.

⟨*Main handle thread-count argument*⟩≡
```
std::istringstream str( argv[++ii] );
unsigned int count = 0;
str >> count;
if ( count > 100 ) {
    count = 100;
}
thread::instance()->setWorkerCount( count );
```

If there is an unrecognized argument, the usage statement is emitted to standard-error.

⟨*Main handle print usage statement*⟩≡
```
std::cerr << "USAGE: " << argv[0];
std::cerr << " [-t threadCount]";
std::cerr << " file1 file2 ..." << std::endl;
```

To read the files from the command-line, this code loops through the arguments on the command-line. With each argument, it attempts to open that as a file and read a scene from it. Once the scene is read, the scene's render method is invoked. The `scene` class is specified in 1.

⟨*Main read files specified on command-line*⟩≡
```
while ( ii < argc ) {
    std::ifstream stream( argv[ii++] );
    if ( stream.good() ) {
        scene ss;
        stream >> ss;
        ss.render();
    } else {
        std::cerr << "Couldn't open: " << argv[ii-1] << std::endl;
    }
}
```

To read in a scene from standard-input, the following code simply reads a scene from `std::cin` and invokes the scene's render method. The `scene` class is specified in 1.

⟨*Main read scene from standard-input*⟩≡
```
if ( std::cin.good() ) {
    scene ss;
    std::cin >> ss;
    ss.render();
} else {
    std::cerr << "Couldn't read standard input" << std::endl;
}
```

41.1 Source Files

The following section assembles the source file for the main routine from the chunks above.

41.1.1 main.cc

The main routine utilizes `fstream`s to read scenes in from files. It uses `iostream`s to read scenes in from standard-input. It uses `sstream`s to parse integer command-line arguments. Further, it directly invokes the construcor and the render method of the `scene` class. And, it uses the `nkthread` system to set the number of threads in the thread pool.

⟨*main.cc*⟩≡

```
#include <fstream>
#include <iostream>
#include <sstream>
#include "scene.h"
#include "nkthread.h"

⟨Main Routine⟩
```

Sample Input Files

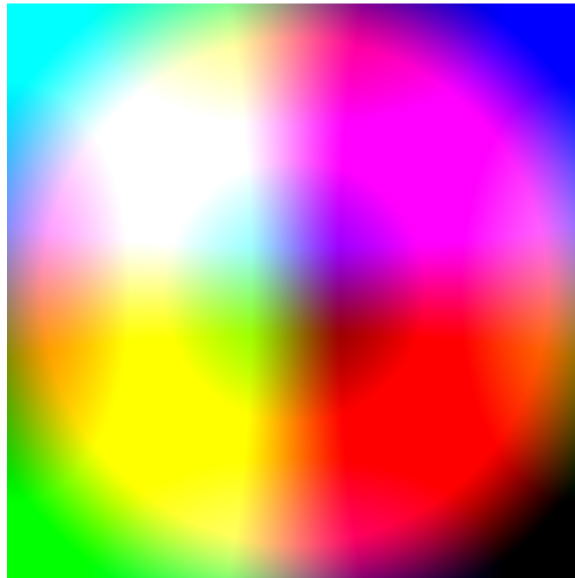

Figure 41.1: A simple rendering without the need for a portal

41.2 Simple Rendering Without Raytracing

This sample renders a view which does not depend upon any geometric objects or lighting of them. The color is determined solely as a function of the view direction. The resulting image appears in figure 41.1. Sadly, this image reminds the author of a glaucoma test.

⟨*justColor.rt*⟩≡
```
view { ⟨justColor view⟩ };
```

The view is set to wrap around if the fish-eye effect is great enough, although there is no fish-eye effect here. The gamma value is set to neutral for speed. And, the border is set to zero because this is a single-frame image.

⟨*justColor view*⟩≡
```
wrap;
gamma 1.0;
border 0;
```

The width and height of the image are both specified to be 256 units at 3 dots per unit without any fish-eye effect. The viewer is expected to be six units away from the image.

⟨*justColor view*⟩+≡
```
units [2]< 256, 256 >;
dotsPerUnit [2]< 3, 3 >;
fishEye [2]< 0, 0 >;
viewDepth 6;
```

The viewport is used to specify the color. Notably, the channel array in the viewport shows that the color is a function of the viewing direction. If the view direction is a unit vector $\vec{d}$, then the color channels are an RGB vector: $\left\langle \frac{1}{2} + \frac{3}{4}\sin\pi d_x, \frac{1}{2} + \frac{3}{4}\sin\pi d_y, \frac{1}{2} + \frac{3}{4}\sin\pi d_z \right\rangle$. The color values get clipped to the range $[0, 1]$.

⟨*justColor view*⟩+≡
```
viewport {
    channels [3]<
        + 0.5 * 0.75 sin * pi dir 0 ,
        + 0.5 * 0.75 sin * pi dir 1 ,
        + 0.5 * 0.75 sin * pi dir 2
    >;
    scale [3]< 1, 1, 1 >;
    orientation [3]<
        [3]< 1, 0, 0 >,
        [3]< 0, 1, 0 >,
        [3]< 0, 0, 1 >
    >;
    center [3]< 0, 0, 0 >;
};
```

The default scale, orientation, and center are shown explicitly here for reference. This viewport is an exceptional case. For most images, one will want the viewport to contain a portal into a universe. This, however, simply functionally color the image.

Finally, the output image information is specified. Here, a `ppm` image is used since the `png` features may not be availabe on all systems.

⟨*justColor view*⟩+≡
```
image ppm { common { basename justColor; }; };
```

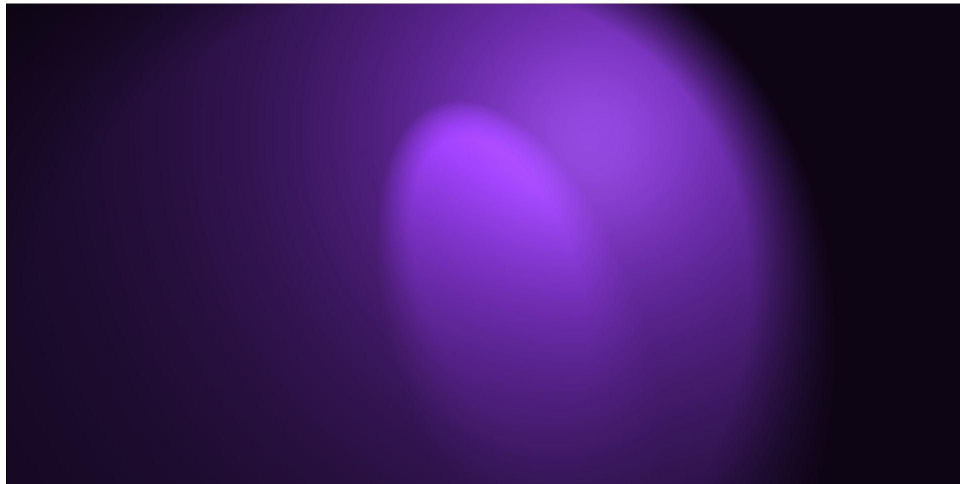

Figure 41.2: Two directional spot lights shining on a plane

41.3 Rendering Of Directional Spot Lights

This sample shows two directional spot lights shining on a plane. The resulting image is in figure 41.2.

⟨*spot.rt*⟩≡
```
view { ⟨spot view⟩ };
universe myuniverse { ⟨spot universe⟩ };
```

The view is 2.56 units wide and 1.28 units high at 300 dots per unit. There is no fish-eye effect. The viewer is expected to be six units from the image.

⟨*spot view*⟩≡
```
units [2]< 2.56, 1.28 >;
dotsPerUnit [2]< 300, 300 >;
viewDepth 6;
```

The viewport places the viewer a fair distance back the x-axis so that the full spread of the spotlights is visible.

⟨*spot view*⟩+≡
```
viewport {
    portal myportal {
        universe myuniverse ;
        center [3]< -100, 0, 0 >;
    };
    channels [3]< @ myportal 0 , @ myportal 1 , @ myportal 2 >;
};
```

Then, the output image is specified.

⟨*spot view*⟩+≡
```
image ppm { common { basename spot; }; };
```

Both spotlights are positioned.

⟨*spot universe*⟩≡
```
dimensions 3;
ambientLight {
    channels [3]< 0.1 , 0.1 , 0.1 >;
};
skyColor {
    channels [3]< 0.2 , 0.2 , 0.6 >;
};
light {
    position [3]< -10, 5, 15 >;
    falloff 0.05;
    direction [3]< 10, -5, -15 >;
    angles 10, 15;
    color {
        channels [3]< 0.4 , 0.4 , 0.4 >;
    };
};
light {
    position [3]< -15, -5, 5 >;
    falloff 0.05;
    direction [3]< 15,  5, -5 >;
    angles 40, 50;
    color {
        channels [3]< 0.5 , 0.5 , 0.5 >;
    };
};
```

And, a plane is positioned for them to shine upon.

⟨*spot universe*⟩+≡
```
object halfspace {
    base {
        orientation [1]< [1]< -1 > >;
        color {
            channels [3]< 0.5 , 0.2 , 0.8 >;
        };
    };
};
```

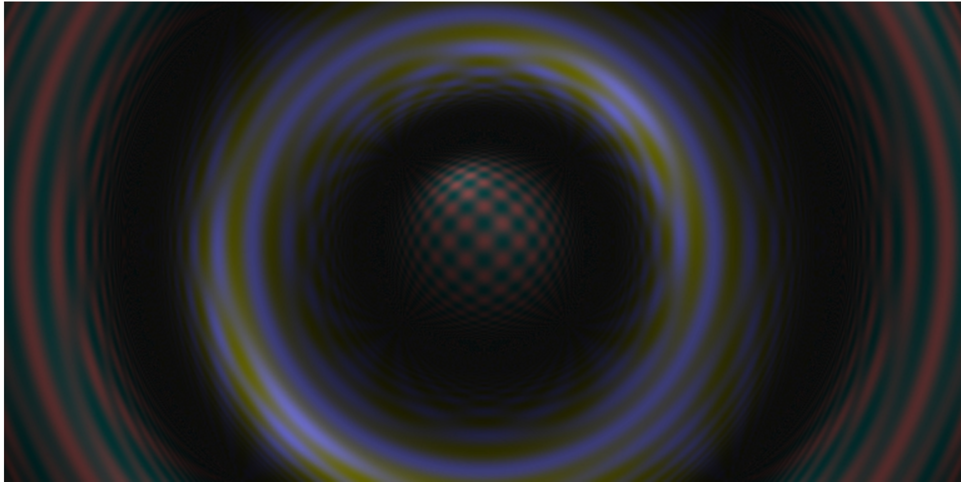

Figure 41.3: A fish-eyed rendering of two halfspaces

41.4 Rendering Of Two Halfspaces

This sample renders a view into a universe where the viewpoint is directly between two halfspaces. The halfspace directly in front of the viewpoint is checkered blue and peach. The halfspace directly behind the viewpoint is checkered indigo and greenish yellow. The view is fish-eyed and wrapped so that one can see more than the whole way around so that the plane in front shows up again on the fringes of the image. The resulting image is in figure 41.3.

```
⟨space.rt⟩≡
  view { ⟨halfspace view⟩ };
  universe myuniverse { ⟨halfspace universe⟩ };
```

The view is set to wrap around to provide a prettier image. The gamma value is set to neutral for speed. And, the border is set to zero because this is a single frame image.

```
⟨halfspace view⟩≡
  wrap;
  gamma 1.0;
  border 0;
```

The width of the image is specified to be 2.56 units at 300 dots per unit. The height of the image is specified to be 1.28 units at 300 dots per unit. There is a huge fish-eye effect which causes the plane in front of the viewpoint to actually show up again. If `wrap` had not been set above, the outer edge would be white instead of wrapping around. And, the viewer is expected to be sitting six units from the image.

⟨*halfspace view*⟩+≡
```
units [2]< 2.56, 1.28 >;
dotsPerUnit [2]< 300, 300 >;
fishEye [2]< 30, 30 >;
viewDepth 6;
```

The viewport into the universe is straightforward. It starts slightly down the x-axis.

⟨*halfspace view*⟩+≡
```
viewport {
    portal myportal {
        universe myuniverse ;
        center [3]< -12, 0, 0 >;
    };
    channels [3]< @ myportal 0 , @ myportal 1 , @ myportal 2 >;
};
```

then, the output image information is specified.

⟨*halfspace view*⟩+≡
```
image ppm { common { basename space; }; };
```

This is a three-dimensional universe with a little bit of ambient light, a slightly blue sky, and a couple of lights.

⟨*halfspace universe*⟩≡
```
dimensions 3;
ambientLight {
    channels [3]< 0.1 , 0.1 , 0.1 >;
};
skyColor {
    channels [3]< 0.2 , 0.2 , 0.6 >;
};
light {
    position [3]< -10, 5, 15 >;
    falloff 0.1;
    color {
        channels [3]< 0.4 , 0.4 , 0.4 >;
    };
};
light {
    position [3]< -15, -5, 5 >;
    falloff 0.1;
    color {
        channels [3]< 0.5 , 0.5 , 0.5 >;
    };
};
```

There is one plane at the origin oriented so its normal points at the viewpoint. Technically, this isn't a plane. It is the entire halfspace beyond the plane. But, the rays here will only hit the surface of the plane.

⟨*halfspace universe*⟩+≡
```
object halfspace {
    base {
        orientation [1]< [1]< -1 > >;
        color {
            channels [3]<
                + 0.5 * 0.5 * cos pos 1 cos pos 2 ,
                0.5 ,
                0.5
            >;
        };
    };
};
```

The second plane sits directly behind the viewpoint as far from the viewpoint as the viewpoint is from the other plane. This plane is also oriented so its normal points toward the viewpoint.

⟨*halfspace universe*⟩+≡

```
object halfspace {
    base {
        center [1]< -24 >;
        orientation [1]< [1]< 1 > >;
        color {
            channels [3]<
                0.5 ,
                0.5 ,
                + 0.5 * 0.5 * cos pos 1 cos pos 2
            >;
        };
    };
};
```

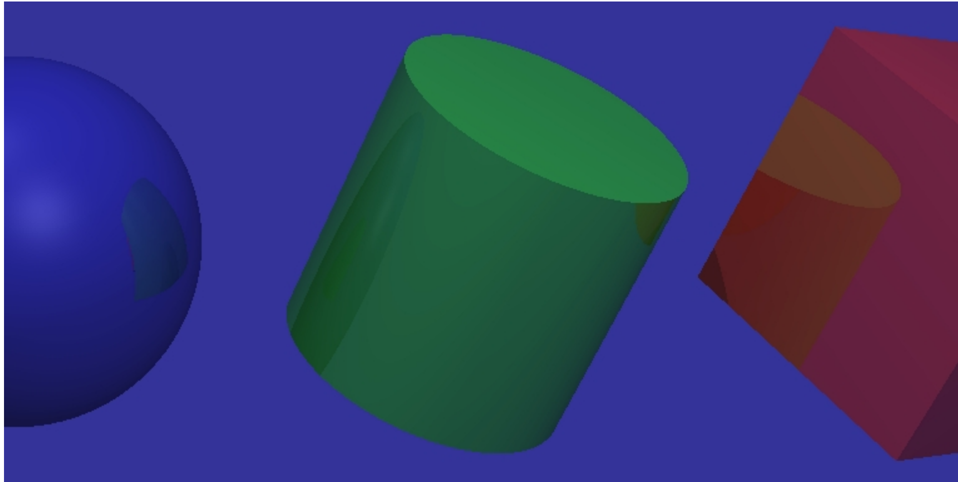

Figure 41.4: A rendering of a sphere, a cylinder, and a cube

41.5 Rendering Of 3-Dimensional Cylinders

This sample renders the three three-dimensional cylinder objects. The resulting image appears in figure 41.4. There is a red cube, next to a green cylinder, next to a blue sphere. Each is slightly reflective.

⟨*cylinders.rt*⟩≡
```
view { ⟨cylinders view⟩ };
universe myuniverse { ⟨cylinders universe⟩ };
```

The width of the image is specified to be 2.56 units at 300 dots per unit. The height of the image is specified to be 1.28 units at 300 dots per unit. This is no fish-eye effect. And, the viewer is expected to be six units from the image.

⟨*cylinders view*⟩≡
```
units [2]< 2.56, 1.28 >;
dotsPerUnit [2]< 300, 300 >;
fishEye [2]< 0, 0 >;
viewDepth 6;
```

The view into the universe starts at twelve units down the x-axis and peers into the universe called `myuniverse`.

⟨*cylinders view*⟩+≡
```
viewport {
    portal myportal {
        universe myuniverse ;
        center [3]< -12, 0, 0 >;
    };
    channels [3]< @ myportal 0 , @ myportal 1 , @ myportal 2 >;
};
```

The output image information is specified. Here, a `ppm` image is used since the `png` features may not be availabe on all systems.

⟨*cylinders view*⟩+≡
```
image ppm { common { basename cylinders; }; };
```

This is a three-dimensional universe with a little bit of ambient light, a slightly blue sky, and a couple of lights.

⟨*cylinders universe*⟩≡
```
dimensions 3;
ambientLight {
    channels [3]< 0.1 , 0.1 , 0.1 >;
};
skyColor {
    channels [3]< 0.2 , 0.2 , 0.6 >;
};
light {
    position [3]< -10, 5, 15 >;
    falloff 0.1;
    color {
        channels [3]< 0.8 , 0.8 , 0.8 >;
    };
};
light {
    position [3]< -15, -5, 5 >;
    falloff 0.1;
    color {
        channels [3]< 1.0 , 1.0 , 1.0 >;
    };
};
```

There is a red cube, next to a green cylinder, next to a slightly reflective blue sphere.

⟨*cylinders universe*⟩+≡

```
object cylinder {
    roundDimensions 0;
    base {
        scale [3]< 0.8, 0.8, 0.8 >;
        ⟨cylinders default orientation⟩
        center [3]< 0, -2.5, 0 >;
        color {
            channels [3]< 1.0 , 0.2 , 0.2 >;
            specularness 0.3;
            reflectiveness 0.3;
        };
    };
};
object cylinder {
    roundDimensions 2;
    base {
        scale [3]< 0.8, 0.8, 0.8 >;
        ⟨cylinders default orientation⟩
        center [3]< 0, 0, 0 >;
        color {
            channels [3]< 0.2 , 1.0 , 0.2 >;
            specularness 0.3;
            reflectiveness 0.3;
        };
    };
};
object cylinder {
    roundDimensions 3;
    base {
        scale [3]< 1.0, 1.0, 1.0 >;
        ⟨cylinders default orientation⟩
        center [3]< 0, 2.5, 0 >;
        color {
            channels [3]< 0.2 , 0.2 , 1.0 >;
            specularness 0.3;
            reflectiveness 0.3;
        };
    };
};
```

Each of the cylinders is tipped slightly for a more interesting image.

⟨*cylinders default orientation*⟩≡

```
orientation [2]<
     [3]<  1, 1, 1 >,
     [3]< -1, 1, 0 >
>;
```

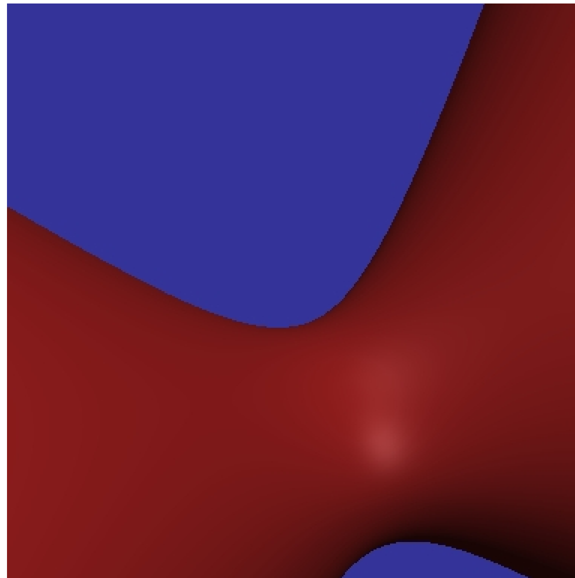

Figure 41.5: A rendering of a quadratic surface

41.6 Rendering Of A Quadratic Solid

This sample renders a quadratic solid. In this case, it is the solid described by:

$$x^2 + xy - xz - y^2 + yz + z^2 + x + y + z + \frac{1}{2} \leq 0$$

The resulting image appears in figure 41.5.

⟨*quadratic.rt*⟩≡
```
view { ⟨quadratic view⟩ };
universe myuniverse { ⟨quadratic universe⟩ };
```

The size of the image is specified to be 1.28 units in each direction at three hundred dots per unit without any fish-eye effect. The viewer is expected to be six units from the image.

⟨*quadratic view*⟩≡
```
units [2]< 1.28, 1.28 >;
dotsPerUnit [2]< 300, 300 >;
viewDepth 6;
```

The viewport puts the viewer a little ways down the x-axis.

⟨*quadratic view*⟩+≡
```
viewport {
    portal myportal {
        universe myuniverse ;
        center [3]< -15, 0, 0 >;
    };
    channels [3]< @ myportal 0 , @ myportal 1 , @ myportal 2 >;
};
```

Then, the output image information is specified.

⟨*quadratic view*⟩+≡
```
image ppm { common { basename quadratic; }; };
```

This is a three-dimensional universe with a little bit of ambient light, a slightly blue sky, and a couple of lights.

⟨*quadratic universe*⟩≡
```
dimensions 3;
ambientLight {
    channels [3]< 0.1 , 0.1 , 0.1 >;
};
skyColor {
    channels [3]< 0.2 , 0.2 , 0.6 >;
};
light {
    position [3]< -10, 5, 15 >;
    falloff 0.1;
    color {
        channels [3]< 0.6 , 0.6 , 0.6 >;
    };
};
light {
    position [3]< -15, -5, 5 >;
    falloff 0.1;
    color {
        channels [3]< 0.8 , 0.8 , 0.8 >;
    };
};
```

This specifies the quadratic object. The matrix of `squares` specifies the coefficients for the terms of order two. The `linears` vector specifies the coefficients for the terms of order one. And, the scalar specifies an offset.

⟨*quadratic universe*⟩+≡

```
object quadratic {
    squares [3]<
        [3]<  1,  1, -1 >,
        [3]<  0, -1,  1 >,
        [3]<  0,  0,  1 >
    >;
    linears [3]< 1, 1, 1 >;
    scalar 0.5;

    base {
        color {
            channels [3]< 1.0 , 0.2 , 0.2 >;
            specularness 0.4;
            phong 10;
        };
    };
};
```

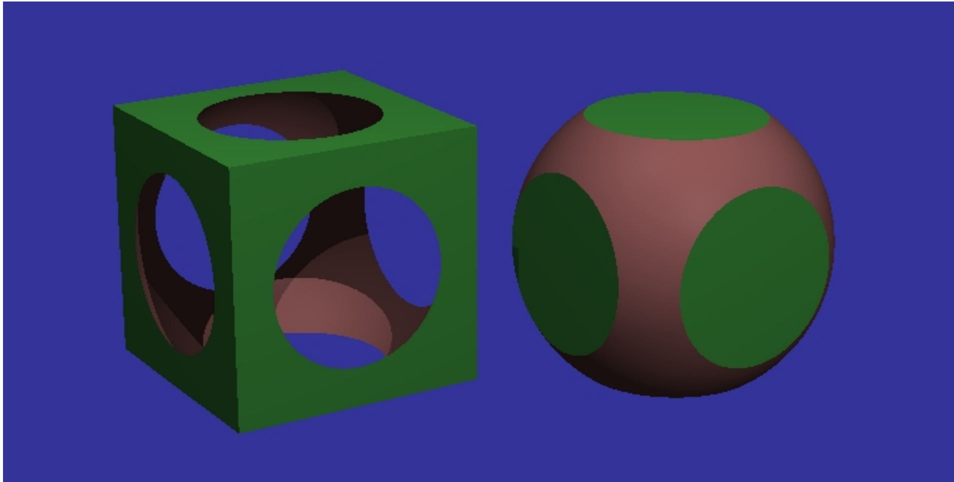

Figure 41.6: A rendering of the intersection of a cube and the complement of a sphere versus the intersection of a cube and a sphere

41.7 Rendering Of A Complement Of A Sphere

This sample renders the intersection of a cube with the complement of a sphere as well as the intersection of a cube with a sphere for comparison. The resulting image appears in figure 41.6.

⟨*compl.rt*⟩≡
```
view { ⟨compl view⟩ };
universe myuniverse { ⟨compl universe⟩ };
```

The size of the image is specified to be 256 units at one dot per unit without any fish-eye effect.

⟨*compl view*⟩≡
```
units [2]< 2.56, 1.28 >;
dotsPerUnit [2]< 300, 300 >;
viewDepth 6;
```

The viewport simply places the viewer slightly down the x-axis.

⟨*compl view*⟩+≡
```
viewport {
    portal myportal {
        universe myuniverse ;
        center [3]< -12, 0, 0 >;
    };
    channels [3]< @ myportal 0 , @ myportal 1 , @ myportal 2 >;
};
```

Then, the output image information is specified.

⟨*compl view*⟩+≡
```
image ppm { common { basename compl; }; };
```

This is a three-dimensional universe with a little bit of ambient light, a slightly blue sky, and a couple of lights.

⟨*compl universe*⟩≡
```
dimensions 3;
ambientLight {
    channels [3]< 0.1 , 0.1 , 0.1 >;
};
skyColor {
    channels [3]< 0.2 , 0.2 , 0.6 >;
};
light {
    position [3]< -10, 5, 15 >;
    falloff 0.1;
    color {
        channels [3]< 0.6 , 0.6 , 0.6 >;
    };
};
light {
    position [3]< -15, -5, 5 >;
    falloff 0.1;
    color {
        channels [3]< 0.8 , 0.8 , 0.8 >;
    };
};
```

```
⟨compl universe⟩+≡
  object intersection {
      ⟨compl cube⟩
      object complement { ⟨compl sphere⟩ };
      base {
          center [3]< 0, 1, 0 >;
          orientation [2]<
              [3]< 3, 2, 1 >,
              [3]< 1, 0, -3 >
          >;
      };
  };
  object intersection {
      ⟨compl cube⟩
      ⟨compl sphere⟩
      base {
          center [3]< 0, -1, 0 >;
          orientation [2]<
              [3]< 3, 2, 1 >,
              [3]< 1, 0, -3 >
          >;
      };
  };

⟨compl sphere⟩≡
  object cylinder {
      roundDimensions 3;
      base {
          scale [3]< 0.85, 0.85, 0.85 >;
          color { channels [3]< 1.0 , 0.6 , 0.6 >; };
      };
  };

⟨compl cube⟩≡
  object cylinder {
      roundDimensions 0;
      base {
          scale [3]< 0.7, 0.7, 0.7 >;
          color { channels [3]< 0.4 , 1.0 , 0.4 >; };
      };
  };
```

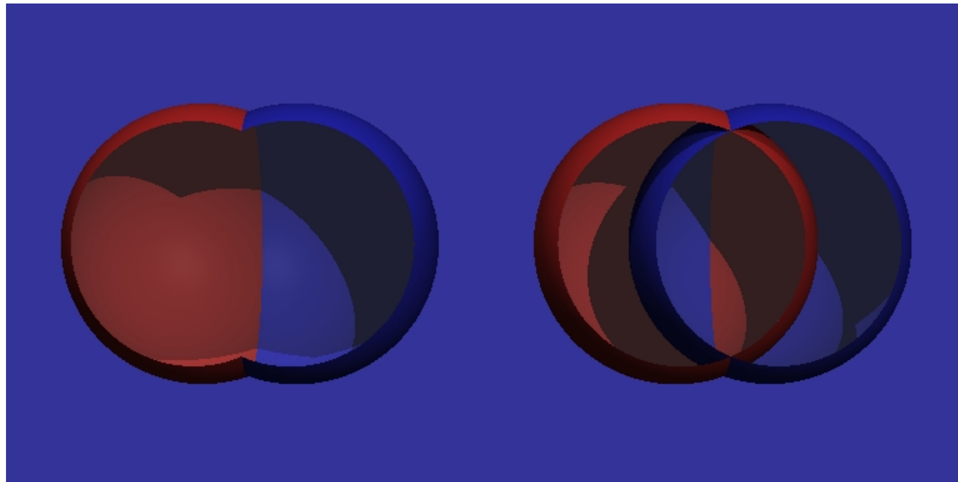

Figure 41.7: A cutaway rendering of a union of two spheres

41.8 Rendering Of The Union Of Two Spheres

This sample two spheres along with their union. The entire scene is shown cutaway so that one can see what really goes on inside the objects. The resulting image appears in figure 41.7.

⟨*union.rt*⟩≡
```
view { ⟨union view⟩ };
universe myuniverse { ⟨union universe⟩ };
```

The width of the image is specified to be 2.56 units at 300 dots per unit. The height of the image is specified to be 1.28 units at 300 dots per unit. The viewer is expected to be six units away from the image.

⟨*union view*⟩≡
```
units [2]< 2.56, 1.28 >;
dotsPerUnit [2]< 300, 300 >;
viewDepth 6;
```

The viewport places the viewer slightly down the x-axis.

⟨*union view*⟩+≡
```
viewport {
    portal myportal {
        universe myuniverse ;
        center [3]< -12, 0, 0 >;
    };
    channels [3]< @ myportal 0 , @ myportal 1 , @ myportal 2 >;
};
```

Then, the output image information is specified.

⟨*union view*⟩+≡
```
image ppm { common { basename union; }; };
```

This is a three-dimensional universe with a little bit of ambient light, a slightly blue sky, and a couple of lights.

⟨*union universe*⟩≡
```
dimensions 3;
ambientLight {
    channels [3]< 0.1 , 0.1 , 0.1 >;
};
skyColor {
    channels [3]< 0.2 , 0.2 , 0.6 >;
};
light {
    position [3]< -10, 5, 15 >;
    falloff 0.1;
    color {
        channels [3]< 0.8 , 0.8 , 0.8 >;
    };
};
light {
    position [3]< -15, -5, 5 >;
    falloff 0.1;
    color {
        channels [3]< 1.0 , 1.0 , 1.0 >;
    };
};
```

This is the box used to cutaway the real objects.

⟨*union universe*⟩+≡
```
object intersection {
    object cylinder {
        roundDimensions 0;
        base {
            scale [3]< 1.4, 5, 1 >;
            center [3]< 1, 0, 0 >;
            color {
                channels [3]< 1.0 , 1.0 , 1.0 >;
                transparency 1.0;
                specularness 0.0;
                reflectiveness 0.0;
            };
        };
    };
    object set {
        ⟨union real objects⟩
    };
};
```

This is the union of the two spheres.

⟨*union real objects*⟩≡
```
object union {
    object cylinder {
        roundDimensions 3;
        base {
            scale [3]< 0.75, 0.75, 0.75 >;
            center [3]< 0, 1.5, 0 >;
            color {
                channels [3]< 1.0 , 0.2 , 0.2 >;
            };
        };
    };
    object cylinder {
        roundDimensions 3;
        base {
            scale [3]< 0.75, 0.75, 0.75 >;
            center [3]< 0, 1.0, 0 >;
            color {
                channels [3]< 0.2 , 0.2 , 1.0 >;
            };
        };
    };
};
```

This is the two spheres without the unioning.

⟨*union real objects*⟩+≡

```
object set {
    object cylinder {
        roundDimensions 3;
        base {
            scale [3]< 0.75, 0.75, 0.75 >;
            center [3]< 0, -1.0, 0 >;
            color {
                channels [3]< 1.0 , 0.2 , 0.2 >;
            };
        };
    };
    object cylinder {
        roundDimensions 3;
        base {
            scale [3]< 0.75, 0.75, 0.75 >;
            center [3]< 0, -1.5, 0 >;
            color {
                channels [3]< 0.2 , 0.2 , 1.0 >;
            };
        };
    };
};
```

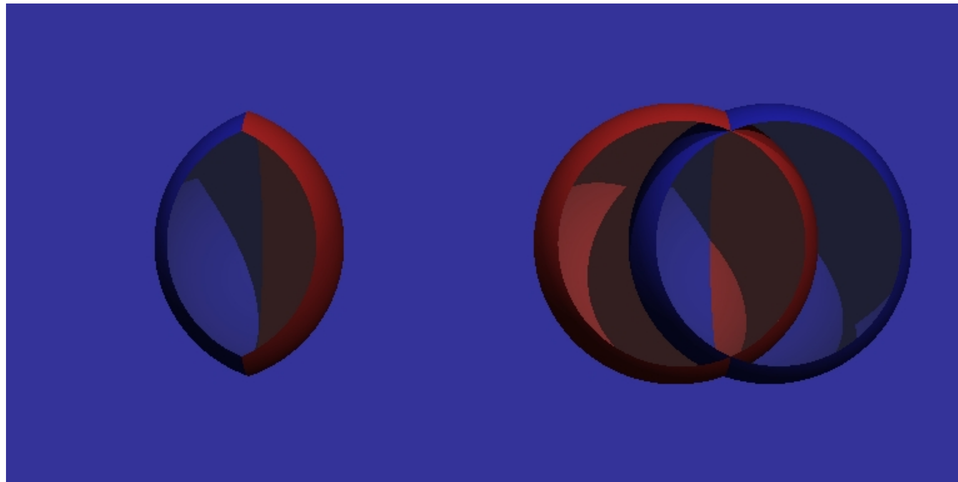

Figure 41.8: A cutaway rendering of a intersection of two spheres

41.9 Rendering Of The Union Of Two Spheres

This sample two spheres along with their intersection. The entire scene is shown cut-away so that one can see what really goes on inside the objects. The resulting image appears in figure 41.8.

⟨*inter.rt*⟩≡
```
view { ⟨intersection view⟩ };
universe myuniverse { ⟨intersection universe⟩ };
```

The width of the image is specified to be 2.56 units at 300 dots per unit. The height of the image is specified to be 1.28 units at 300 dots per unit. The viewer is expected to be six units away from the image.

⟨*intersection view*⟩≡
```
units [2]< 2.56, 1.28 >;
dotsPerUnit [2]< 300, 300 >;
viewDepth 6;
```

The viewport places the viewer slightly down the x-axis.

⟨*intersection view*⟩+≡
```
viewport {
    portal myportal {
        universe myuniverse ;
        center [3]< -12, 0, 0 >;
    };
    channels [3]< @ myportal 0 , @ myportal 1 , @ myportal 2 >;
};
```

Then, the output image information is specified.

⟨*intersection view*⟩+≡
```
image ppm { common { basename inter; }; };
```

This is a three-dimensional universe with a little bit of ambient light, a slightly blue sky, and a couple of lights.

⟨*intersection universe*⟩≡
```
dimensions 3;
ambientLight {
    channels [3]< 0.1 , 0.1 , 0.1 >;
};
skyColor {
    channels [3]< 0.2 , 0.2 , 0.6 >;
};
light {
    position [3]< -10, 5, 15 >;
    falloff 0.1;
    color {
        channels [3]< 0.8 , 0.8 , 0.8 >;
    };
};
light {
    position [3]< -15, -5, 5 >;
    falloff 0.1;
    color {
        channels [3]< 1.0 , 1.0 , 1.0 >;
    };
};
```

This is the box used to cutaway the real objects.

⟨*intersection universe*⟩+≡

```
object intersection {
    object cylinder {
        roundDimensions 0;
        base {
            scale [3]< 1.4, 5, 1 >;
            center [3]< 1, 0, 0 >;
            color {
                channels [3]< 1.0 , 1.0 , 1.0 >;
                transparency 1.0;
                specularness 0.0;
                reflectiveness 0.0;
            };
        };
    };
    object set {
        ⟨intersection real objects⟩
    };
};
```

This is the intersection of the two spheres.

⟨*intersection real objects*⟩≡

```
object intersection {
    object cylinder {
        roundDimensions 3;
        base {
            scale [3]< 0.75, 0.75, 0.75 >;
            center [3]< 0, 1.5, 0 >;
            color {
                channels [3]< 1.0 , 0.2 , 0.2 >;
            };
        };
    };
    object cylinder {
        roundDimensions 3;
        base {
            scale [3]< 0.75, 0.75, 0.75 >;
            center [3]< 0, 1.0, 0 >;
            color {
                channels [3]< 0.2 , 0.2 , 1.0 >;
            };
        };
    };
};
```

This is the two spheres without the intersectioning.

⟨*intersection real objects*⟩+≡

```
object set {
    object cylinder {
        roundDimensions 3;
        base {
            scale [3]< 0.75, 0.75, 0.75 >;
            center [3]< 0, -1.0, 0 >;
            color {
                channels [3]< 1.0 , 0.2 , 0.2 >;
            };
        };
    };
    object cylinder {
        roundDimensions 3;
        base {
            scale [3]< 0.75, 0.75, 0.75 >;
            center [3]< 0, -1.5, 0 >;
            color {
                channels [3]< 0.2 , 0.2 , 1.0 >;
            };
        };
    };
};
```

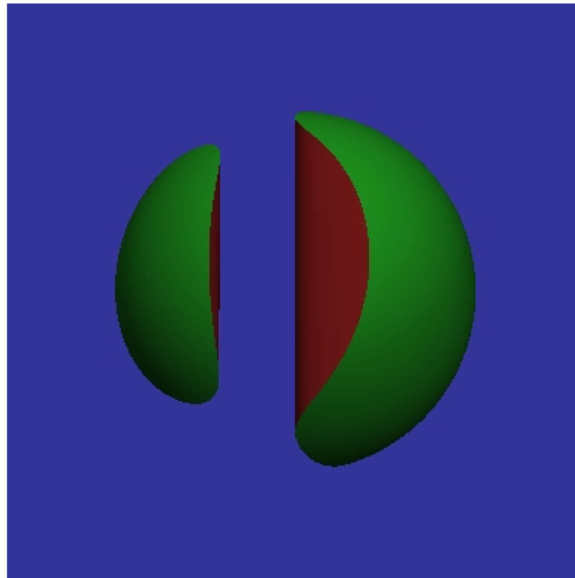

Figure 41.9: A rendering of the intersection of a sphere and an extrusion of a quadratic

41.10 Rendering Of An Extrusion Of A Quadratic

This sample renders the intersection of a unit sphere with the extrusion into three dimensions of the quadratic solid:

$$x^2 + xy - y^2 + x + y + \frac{1}{4} \leq 0$$

The resulting image appears in figure 41.9.

⟨*extrusion.rt*⟩≡

```
view { ⟨extrusion view⟩ };
universe myuniverse { ⟨extrusion universe⟩ };
```

The size of the image is specified to be 1.28 units at three hundred dot per unit without any fish-eye effect. The viewer is expected to be six units from the image.

⟨*extrusion view*⟩≡

```
units [2]< 1.28, 1.28 >;
dotsPerUnit [2]< 300, 300 >;
viewDepth 6;
```

The viewport moves the viewer down the x-axis a small way.

⟨*extrusion view*⟩+≡
```
viewport {
    portal myportal {
        universe myuniverse ;
        center [3]< -15, 0, 0 >;
    };
    channels [3]< @ myportal 0 , @ myportal 1 , @ myportal 2 >;
};
```

Then, the output image information is specified.

⟨*extrusion view*⟩+≡
```
image ppm { common { basename extrusion; }; };
```

This is a three-dimensional universe with a little bit of ambient light, a slightly blue sky, and a couple of lights.

⟨*extrusion universe*⟩≡
```
dimensions 3;
ambientLight {
    channels [3]< 0.1 , 0.1 , 0.1 >;
};
skyColor {
    channels [3]< 0.2 , 0.2 , 0.6 >;
};
light {
    position [3]< -10, 5, 15 >;
    falloff 0.1;
    color {
        channels [3]< 0.6 , 0.6 , 0.6 >;
    };
};
light {
    position [3]< -15, -5, 5 >;
    falloff 0.1;
    color {
        channels [3]< 0.8 , 0.8 , 0.8 >;
    };
};
```

Here is the intersection of the objects. The extrusion says that the items inside the extrusion are 2-dimensional. The universe that the extrusion sits in is three-dimensional.

⟨*extrusion universe*⟩+≡

```
object intersection {

    object extrusion {
        dimensions 2;
        object quadratic {
            squares [2]<
                [2]<  1,  1 >,
                [2]<  0, -1 >
            >;
            linears [2]< 1, 1 >;
            scalar 0.25;
        };
        base {
            color {
                channels [3]< 1.0 , 0.2 , 0.2 >;
                specularness 0.4;
                phong 5;
            };
        };
    };

    object cylinder {
        roundDimensions 3;
        base {
            scale [3]< 1, 1, 1 >;
            color {
                channels [3]< 0.2 , 1.0 , 0.2 >;
            };
        };
    };

    base {
        orientation [2]<
            [3]< 5, 0, -1 >,
            [3]< 0, 1, 0 >
        >;
    };
};
```

Figure 41.10: A rendering of a cube, an icosahedron, and an dodecahedron

41.11 Rendering Of 3-Dimensional Coxeters

This sample shows how to generate a regular polytope based upon the Coxeter-Dynkin diagram of the symmetry group of the vertexes. The resulting image appears in figure 41.10.

⟨*coxeter.rt*⟩≡
```
view { ⟨coxeter view⟩ };
universe myuniverse { ⟨coxeter universe⟩ };
```

The width of the image is specified to be 2.56 units at 300 dots per unit. The height of the image is specified to be 1.28 units at 300 dots per unit. The viewer is expected to be six units from the image.

⟨*coxeter view*⟩≡
```
units [2]< 2.56, 1.28 >;
dotsPerUnit [2]< 300, 300 >;
viewDepth 6;
```

The viewpoint sets the viewer back down the x-axis a ways.

⟨*coxeter view*⟩+≡
```
viewport {
    portal myportal {
        universe myuniverse ;
        center [3]< -12, 0, 0 >;
    };
    channels [3]< @ myportal 0 , @ myportal 1 , @ myportal 2 >;
};
```

Then, the output image information is specified.

⟨*coxeter view*⟩+≡
```
image ppm { common { basename coxeter; }; };
```

This is a three-dimensional universe with a little bit of ambient light, a slightly blue sky, and a couple of lights.

⟨*coxeter universe*⟩≡
```
dimensions 3;
ambientLight {
    channels [3]< 0.1 , 0.1 , 0.1 >;
};
skyColor {
    channels [3]< 0.2 , 0.2 , 0.6 >;
};
light {
    position [3]< -10, 5, 15 >;
    falloff 0.1;
    color {
        channels [3]< 0.8 , 0.8 , 0.8 >;
    };
};
light {
    position [3]< -15, -5, 5 >;
    falloff 0.1;
    color {
        channels [3]< 1.0 , 1.0 , 1.0 >;
    };
};
```

There is a red dodecahedron, next to a green icosahedron, next to a blue cube.

⟨*coxeter universe*⟩+≡
```
object coxeter {
        #
        # o--5--*-----*
        #
    matrix [3]<
        [3]< 1, 5, 2 >,
        [3]< 5, 0, 3 >,
        [3]< 2, 3, 0 >
    >;

    ⟨coxeter edge and vertex colors⟩

    base {
        scale [3]< 0.8, 0.8, 0.8 >;
        ⟨coxeter default orientation⟩
        center [3]< 0, -1.5, 0 >;
        color {
            channels [3]< 1.0 , 0.2 , 0.2 >;
            specularness 0.3;
            reflectiveness 0.1;
            transparency 0.4;
        };
    };
};
object coxeter {
        #
        # *--5--*-----o
        #
    matrix [3]<
        [3]< 0, 5, 2 >,
        [3]< 5, 0, 3 >,
        [3]< 2, 3, 1 >
    >;

    ⟨coxeter edge and vertex colors⟩

    base {
        scale [3]< 0.8, 0.8, 0.8 >;
        ⟨coxeter default orientation⟩
        center [3]< 0, 0, 0 >;
        color {
            channels [3]< 0.2 , 1.0 , 0.2 >;
            specularness 0.3;
```

```
            reflectiveness 0.1;
            transparency 0.4;
        };
    };
};
object coxeter {
        #
        # o--4--*-----*
        #
    matrix [3]<
        [3]< 1, 4, 2 >,
        [3]< 4, 0, 3 >,
        [3]< 2, 3, 0 >
    >;

    ⟨coxeter edge and vertex colors⟩

    base {
        scale [3]< 0.8, 0.8, 0.8 >;
        ⟨coxeter default orientation⟩
        center [3]< 0, 1.5, 0 >;
        color {
            channels [3]< 1.0 , 1.0 , 0.2 >;
            specularness 0.3;
            reflectiveness 0.1;
            transparency 0.4;
        };
    };
};
```

⟨coxeter edge and vertex colors⟩≡

```
colorDistance 0.05;
specialColor 1 {
    channels [3]< 0.2 , 0.2 , 0.2 >;
    specularness 0.3;
    reflectiveness 0.1;
};
specialColor 0 {
    channels [3]< 1.0 , 0.5 , 0.5 >;
    specularness 0.3;
    reflectiveness 0.1;
};
```

All of these are shown over a reflective plane to add to the ambience.

⟨*coxeter universe*⟩+≡
```
object halfspace {
    base {
        center [3]< 0, 0, -0.75 >;
        orientation [1]< [3]< 0, 0, 1 > >;
        color {
            channels [3]< 0.8 , 0.8 , 1.0 >;
            reflectiveness 0.8;
        };
    };
};
```

All of these shapes are tipped the same way.

⟨*coxeter default orientation*⟩≡
```
orientation [2]<
    [3]<  1, 2, 5 >,
    [3]< -2, 1, 0 >
>;
```

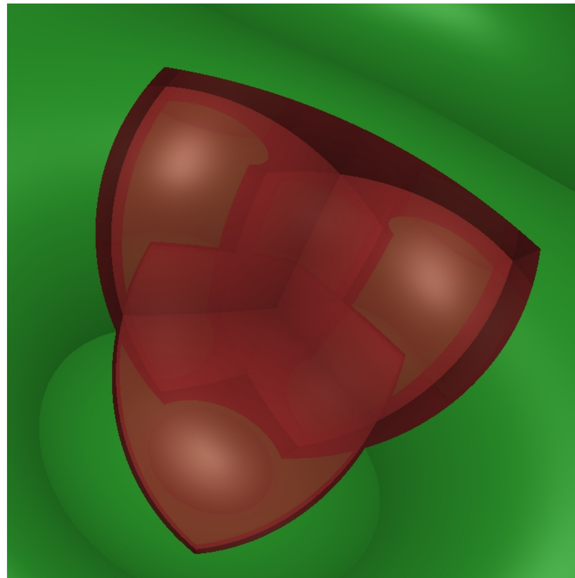

Figure 41.11: A rendering from inside the union of a cube and a cylinder

41.12 Rendering From Inside The Union Of A Cube And A Cylinder

This sample renders a fish-eyed view from inside the intersection of a reflective cube and a cylinder. The resulting image appears in figure 41.11.

⟨*inside.rt*⟩≡
```
view { ⟨inside view⟩ };
universe myuniverse { ⟨inside universe⟩ };
```

The size of the image is specified to be 2.56 units at one 300 dots per unit with some fish-eye effect.

⟨*inside view*⟩≡
```
units [2]< 2.56, 2.56 >;
dotsPerUnit [2]< 300, 300 >;
fishEye [2]< 8, 8 >;
viewDepth 6;
traceDepth 18;
```

The viewport sets the viewer right at the center of the universe.

⟨*inside view*⟩+≡
```
viewport {
    portal myportal {
        universe myuniverse ;
        center [3]< 0, 0, 0 >;
    };
    channels [3]< @ myportal 0 , @ myportal 1 , @ myportal 2 >;
};
```

Then, the output image information is specified.

⟨*inside view*⟩+≡
```
image ppm { common { basename inside; }; };
```

This is a three-dimensional universe with a little bit of ambient light, a slightly blue sky, and a couple of lights.

⟨*inside universe*⟩≡
```
dimensions 3;
ambientLight {
    channels [3]< 0.1 , 0.1 , 0.1 >;
};
skyColor {
    channels [3]< 0.2 , 0.2 , 0.6 >;
};
light {
    position [3]< -10, 5, 15 >;
    falloff 0.0;
    color {
        channels [3]< 0.3 , 0.3 , 0.3 >;
    };
};
light {
    position [3]< -15, -5, 5 >;
    falloff 0.0;
    color {
        channels [3]< 0.4 , 0.4 , 0.4 >;
    };
};
```

Here are the cylinder and the cube.

⟨*inside universe*⟩+≡
```
object union {
    object cylinder {
    roundDimensions 0;
        base {
            scale [3]< 30.0, 30.0, 30.0 >;
            orientation [2]<
                [3]<  1, 1, 1 >,
                [3]< -1, 1, 0 >
            >;
            center [3]< 15, 0, 0 >;
            color {
                channels [3]< 0.7 , 0.2 , 0.2 >;
                specularness 0.3;
                reflectiveness 0.2;
            };
        };
    };
    object cylinder {
        roundDimensions 2;
        base {
            scale [3]< 40.0, 40.0, 40.0 >;
            orientation [2]<
                [3]<  1, 1, 1 >,
                [3]< -1, 1, 0 >
            >;
            center [3]< -15, 0, 0 >;
            color {
                channels [3]< 0.2 , 0.7 , 0.2 >;
                specularness 0.3;
            };
        };
    };
};
```