B SCENE DESCRIPTION INTERFACE

This appendix describes the application programming interface (API) that is used to describe the scene to be rendered to `pbrt`. Users of the renderer typically don't call the functions in this interface directly, but instead describe their scenes using the text file format described in the file `docs/fileformat.pdf` in the `pbrt` distribution. The statements in these text files have a direct correspondence to the API functions described here.

The need for such an interface to the renderer is clear: there must be a convenient way in which all of the properties of the scene to be rendered can be communicated to the renderer. The interface should be well defined and general purpose, so that future extensions to the system fit into its structure cleanly. It shouldn't be too complicated, so that it's easy to describe scenes, but it should be expressive enough that it doesn't leave any of the renderer's capabilities hidden.

A key decision to make when designing a rendering API is whether to expose the system's internal algorithms and structures or offer a high-level abstraction for describing the scene. These have historically been the two main approaches to scene description in graphics: the interface may specify *how* to render the scene, configuring a rendering pipeline at a low level using deep knowledge of the renderer's internal algorithms, or it may specify *what* the scene's objects, lights, and material properties are and leave it to the renderer to decide how to transform that description into the best possible image.

The first approach has been successfully used for interactive graphics. In APIs such as OpenGL® or Direct3D®, it is not possible to just mark an object as a mirror and have reflections appear automatically; rather, the user must choose an algorithm for rendering reflections, render the scene multiple times (e.g., to generate an environment map), store those images in a texture, and then configure the graphics pipeline to use the environment map when rendering the reflective object. The advantage of this approach is that the full flexibility of the rendering pipeline is exposed to the user, making it possible to carefully control the actual computation being done and to use the pipeline very efficiently. Furthermore, because APIs like these impose a very thin abstraction layer between the user and the renderer, the user can be confident that unexpected inefficiencies won't be introduced by the API.

The second approach to scene description, based on describing the geometry, materials, and lights at a higher level of abstraction, has been most successful for applications like high-quality offline rendering. There, users are generally willing to cede control of the low-level rendering details to the renderer in exchange for the ability to specify the scene's properties at a high level. An important advantage of the high-level approach is that the implementations of these renderers have greater freedom to make major changes to the internal algorithms of the system, since the API exposes less of them.

For `pbrt`, we will use an interface based on the descriptive approach. Because `pbrt` is fundamentally physically based, the API is necessarily less flexible in some ways than APIs for many nonphysically based rendering packages. For example, it is not possible to have some lights illuminate only some objects in the scene.

Another key decision to make in graphics API design is whether to use an immediate mode or a retained mode style. In an immediate mode API, the user specifies the scene via a stream of commands that the renderer processes as they arrive. In general, the user cannot make changes to the scene description data already specified (e.g., "change the material of that sphere I described previously from plastic to glass"); once it has been given to the renderer, the information is no longer accessible to the user. Retained mode APIs give the user some degree of access to the data structures that the renderer has built to represent the scene. The user can then modify the scene description in a variety of ways before finally instructing the renderer to render the scene.

Immediate mode has been very successful for interactive graphics APIs since it allows graphics hardware to draw the objects in the scene as they are supplied by the user. Since they do not need to build data structures to store the scene and since they can apply techniques like immediately culling objects that are outside of the viewing frustum without worrying that the user will change the camera position before rendering, these APIs have been key to high-performance interactive graphics.

For ray-tracing-based renderers like `pbrt`, where the entire scene must be described and stored in memory before rendering can begin, some of these advantages of an immediate mode interface aren't applicable. Nonetheless, we will use immediate mode semantics in our API, since it leads to a clean and straightforward scene description language. This choice makes it more difficult to use `pbrt` for applications like quickly rerendering a scene after making a small change to it (e.g., by moving a light source) and may make rendering animations less straightforward, since the entire scene needs to be redescribed for each frame of an animation. Adding a retained mode interface to `pbrt` would be a challenging but useful project.

`pbrt`'s rendering API consists of just over 40 carefully chosen functions, all of which are declared in the `core/api.h` header file. The implementation of these functions is in `core/api.cpp`. This appendix will focus on the general process of turning the API function calls into instances of the classes that represent the scenes.

B.1 PARAMETER SETS

A key problem that a rendering API must address is extensibility: if a developer has created new implementations of shapes, cameras, or the like for pbrt, it shouldn't be necessary to modify the API. In particular, the API should be as unaware as possible of what particular parameters these objects take and what their semantics are. pbrt uses the ParamSet class to address this issue. This class handles collections of named parameters and their values in a generic way. For example, it might record that there is a single floating-point value named "radius" with a value of 2.5, and an array of four color values named "specular" with various color values. The ParamSet provides methods for both setting and retrieving values from these kinds of generic parameter lists. It is defined in core/paramset.h and core/paramset.cpp.

Most of pbrt's API routines take a ParamSet as one of their parameters; for example, the shape creation routine, pbrtShape(), takes a string giving the name of the shape to make and a ParamSet with parameters for it. The creation routine of the corresponding shape implementation is called with the ParamSet passed along as a parameter. This style makes the API's implementation very straightforward.

```
⟨ParamSet Declarations⟩ ≡
    class ParamSet {
    public:
        ⟨ParamSet Public Methods 1049⟩
    private:
        ⟨ParamSet Private Data 1047⟩
    };
```

A ParamSet can hold nine types of parameters: Booleans, integers, floating-point values, points, vectors, normals, spectra, strings, and the names of Textures that are being used as parameters for Materials and other Textures. Internally, it stores a vector of named values for each of the different types that it stores; each parameter is represented by a ParamSetItem of the appropriate type. This representation means that searching for a given parameter takes $O(n)$ time, where n is the number of parameters of the parameter's type. In practice, there are just a handful of parameters to any function, so a more time-efficient representation isn't necessary.

Material	483
Normal	65
ParamSet	1047
ParamSetItem	1048
pbrtShape()	1065
Point	63
Reference	1011
Spectrum	263
Texture	519
Vector	57

```
⟨ParamSet Private Data⟩ ≡                                         1047
    vector<Reference<ParamSetItem<bool> > > bools;
    vector<Reference<ParamSetItem<int> > > ints;
    vector<Reference<ParamSetItem<float> > > floats;
    vector<Reference<ParamSetItem<Point> > > points;
    vector<Reference<ParamSetItem<Vector> > > vectors;
    vector<Reference<ParamSetItem<Normal> > > normals;
    vector<Reference<ParamSetItem<Spectrum> > > spectra;
    vector<Reference<ParamSetItem<string> > > strings;
    vector<Reference<ParamSetItem<string> > > textures;
```

B.1.1 THE `ParamSetItem` STRUCTURE

The `ParamSetItem` structure stores all of the relevant information about a single parameter, such as its name, its base type, and its value(s). For example (using the syntax from `pbrt`'s input files), the `foo` parameter

```
"float foo" [ 0 1 2 3 4 5 ]
```

has a base type of `float`, and six values have been supplied for it. It would be represented by a single `ParamSetItem<float>`. To simplify determining when the data a `ParmSetItem` stores can be freed, this structure is reference counted.

⟨*ParamSet Declarations*⟩ +≡

```
template <typename T> struct ParamSetItem : public ReferenceCounted {
    ⟨ParamSetItem Public Methods⟩
    ⟨ParamSetItem Data 1048⟩
};
```

The `ParamSetItem` directly initializes its members from the arguments and makes a copy of the values.

⟨*ParamSetItem Methods*⟩ ≡

```
template <typename T>
ParamSetItem<T>::ParamSetItem(const string &n, const T *v, int ni) {
    name = n;
    nItems = ni;
    data = new T[nItems];
    for (int i = 0; i < nItems; ++i) data[i] = v[i];
    lookedUp = false;
}
```

The Boolean value `lookedUp` is set to `true` after the value has been retrieved from the `ParamSet`. This makes it possible to print warning messages if any parameters were added to the parameter set but never used, which typically indicates a misspelling in the scene description file or other user error.

⟨*ParamSetItem Data*⟩ ≡ **1048**

```
string name;
int nItems;
T *data;
mutable bool lookedUp;
```

ParamSetItem 1048
ParamSetItem::data 1048
ParamSetItem::lookedUp 1048
ParamSetItem::name 1048
ParamSetItem::nItems 1048
ReferenceCounted 1010

B.1.2 ADDING TO THE PARAMETER SET

To add an entry to the parameter set, the user calls the appropriate `ParamSet` method, passing the name of the parameter, a pointer to its data, and the number of data items. These methods first remove previous values for this parameter, if any.

⟨*ParamSet Methods*⟩ ≡
```
void ParamSet::AddFloat(const string &name, const float *data,
                        int nItems) {
    EraseFloat(name);
    floats.push_back(new ParamSetItem<float>(name, data, nItems));
}
```

We won't include the rest of the methods to add data to the ParamSet, but we do include their prototypes here for reference. The erasure methods are also straightforward and won't be included here.

⟨*ParamSet Public Methods*⟩ ≡ **1047**
```
void AddInt(const string &, const int *, int nItems);
void AddBool(const string &, const bool *, int nItems);
void AddPoint(const string &, const Point *, int nItems);
void AddVector(const string &, const Vector *, int nItems);
void AddNormal(const string &, const Normal *, int nItems);
void AddString(const string &, const string *, int nItems);
void AddTexture(const string &, const string &);
```

A number of different methods for adding spectral data are provided, making it easy for the data to be supplied with a variety of representations.

⟨*ParamSet Public Methods*⟩ +≡ **1047**
```
void AddRGBSpectrum(const string &, const float *, int nItems);
void AddXYZSpectrum(const string &, const float *, int nItems);
void AddBlackbodySpectrum(const string &, const float *, int nItems);
void AddSampledSpectrumFiles(const string &, const char **, int nItems);
void AddSampledSpectrum(const string &, const float *, int nItems);
```

B.1.3 LOOKING UP VALUES IN THE PARAMETER SET

To retrieve a parameter value from a set, it is necessary to loop through the entries of the requested type and return the appropriate value, if any. There are two versions of the lookup method for each parameter type: a simple one for parameters that have a single data value, and a more general one that returns a pointer to the possibly multiple values of more complex types. The first method mostly serves to reduce the amount of code needed in routines that retrieve parameter values.

The methods that look up a single item (e.g., FindOneFloat()) take the name of the parameter and a default value. If the parameter is not found, the default value is returned. This makes it easy to write initialization code like

```
float radius = params.FindOneFloat("radius", 1.f);
```

Normal 65
ParamSet 1047
ParamSetItem 1048
Point 63
Vector 57

In this case, it is not an error if there isn't a "radius" parameter; the default value will be used instead. If the caller wants to detect a missing parameter and issue an error, the second variant of lookup method should be used, since they return a NULL pointer if the parameter isn't found.

⟨*ParamSet Methods*⟩ +≡

```
float ParamSet::FindOneFloat(const string &name, float d) const {
    for (uint32_t i = 0; i < floats.size(); ++i)
        if (floats[i]->name == name && floats[i]->nItems == 1) {
            floats[i]->lookedUp = true;
            return *(floats[i]->data);
        }
    return d;
}
```

As earlier, here are the declarations of the analogous methods for the remaining types:

⟨*ParamSet Public Methods*⟩ +≡ **1047**

```
int FindOneInt(const string &, int d) const;
bool FindOneBool(const string &, bool d) const;
Point FindOnePoint(const string &, const Point &d) const;
Vector FindOneVector(const string &, const Vector &d) const;
Normal FindOneNormal(const string &, const Normal &d) const;
Spectrum FindOneSpectrum(const string &,
                         const Spectrum &d) const;
string FindOneString(const string &, const string &d) const;
string FindTexture(const string &) const;
```

The second kind of lookup method returns a pointer to the data if the data is present and returns the number of data items in `nItems`.

⟨*ParamSet Methods*⟩ +≡

```
const float *ParamSet::FindFloat(const string &name, int *n) const {
    for (uint32_t i = 0; i < floats.size(); ++i)
        if (floats[i]->name == name) {
            *n = floats[i]->nItems;
            floats[i]->lookedUp = true;
            return floats[i]->data;
        }
    return NULL;
}
```

These are the prototypes for the rest of the lookup functions for the other types:

⟨*ParamSet Public Methods*⟩ +≡ **1047**

```
const int *FindInt(const string &, int *nItems) const;
const bool *FindBool(const string &, int *nItems) const;
const Point *FindPoint(const string &, int *nItems) const;
const Vector *FindVector(const string &, int *nItems) const;
const Normal *FindNormal(const string &, int *nItems) const;
const Spectrum *FindSpectrum(const string &, int *nItems) const;
const string *FindString(const string &, int *nItems) const;
```

Because the user may misspell parameter names in the scene description file, the `ParamSet` also provides a `ReportUnused()` function, not included here, that goes through the pa-

Normal	65
ParamSet	1047
ParamSet::floats	1047
ParamSetItem::data	1048
ParamSetItem::lookedUp	1048
ParamSetItem::name	1048
ParamSetItem::nItems	1048
Point	63
Spectrum	263
Vector	57

rameter set and reports if any of the parameters present were never looked up, checking the `ParamSetItem::lookedUp` member variable. For any items where this variable stores the value `false`, it is likely that the user has given an incorrect parameter.

⟨*ParamSet Public Methods*⟩ +≡ **1047**

```
void ReportUnused() const;
```

The `ParamSet::Clear()` method clears all of the individual parameter vectors. The corresponding `ParamSetItems` will in turn be freed if their reference count goes to zero.

⟨*ParamSet Public Methods*⟩ +≡ **1047**

```
void Clear();
```

B.2 INITIALIZATION AND RENDERING OPTIONS

We now have the machinery to describe the routines that make up the rendering API. Before any other API functions can be called, the rendering system must be initialized by a call to `pbrtInit()`. Similarly, when rendering is done, `pbrtCleanup()` should be called; this handles final cleanup of the system. The definitions of these two functions will be filled in at a number of points throughout the rest of this appendix.

A few systemwide options are passed to `pbrtInit()` using the `Options` structure here.

⟨*Global Forward Declarations*⟩ +≡

```
struct Options {
    Options() { nCores = 0;
                quickRender = quiet = openWindow = verbose = false;
                imageFile = ""; }
    int nCores;
    bool quickRender;
    bool quiet, verbose;
    bool openWindow;
    string imageFile;
};
```

⟨*API Function Definitions*⟩ ≡

```
void pbrtInit(const Options &opt) {
    PbrtOptions = opt;
    ⟨API Initialization 1053⟩
    SampledSpectrum::Init();
}
```

`Options` 1051
`ParamSet::Clear()` 1051
`ParamSetItem::lookedUp` 1048
`PbrtOptions` 1051
`SampledSpectrum::Init()` 271

The options are stored in a global variable for easy access by other parts of the system. This variable is only used in a read-only fashion by the system after initialization in `pbrtInit()`.

⟨*API Global Variables*⟩ ≡

```
Options PbrtOptions;
```

```
⟨API Function Definitions⟩ +≡
    void pbrtCleanup() {
        ProbesCleanup();
        ⟨API Cleanup 1053⟩
    }
```

After the system has been initialized, a subset of the API routines is available. Legal calls at this point are those that set general rendering options like the camera and sampler properties, the type of film to be used, and so on, but the user is not yet allowed to start to describe the lights, shapes, and materials in the scene.

After the overall rendering options have been set, the `pbrtWorldBegin()` function locks them in; it is no longer legal to call the routines that set them. At this point, the user can begin to describe the geometric primitives and lights that are in the scene. This separation of global versus scene-specific information can help simplify the implementation of the renderer. For example, consider a spline patch shape that tessellates itself into triangles. This shape might compute the required size of its generated triangles based on the area of the screen that it covers. If the camera's position and image resolution are guaranteed not to change after the shape is created, then the shape can potentially do the tessellation work immediately at creation time.

Once the scene has been fully specified, the `pbrtWorldEnd()` routine is called. At this point, the renderer knows that the scene description is complete and that rendering can begin. The image will be rendered and written to a file before `pbrtWorldEnd()` returns. The user may then specify new options for another frame of an animation, and then another `pbrtWorldBegin()`/`pbrtWorldEnd()` block to describe the geometry for the next frame, repeating as many times as desired. The remainder of this section will discuss the routines related to setting rendering options. Section B.3 describes the routines for specifying the scene inside the world block.

B.2.1 STATE TRACKING

There are three distinct states that the renderer's API can be in:

- *Uninitialized:* Before `pbrtInit()` has been called, no other API calls are legal.
- *Option block:* Outside a `pbrtWorldBegin()` and `pbrtWorldEnd()` pair, scenewide global options may be set.
- *World block:* Inside a `pbrtWorldBegin()` and `pbrtWorldEnd()` pair, the scene may be described.

A module static variable `currentApiState` starts out with value `STATE_UNINITIALIZED`, indicating that the API system hasn't yet been initialized. Its value is updated appropriately by `pbrtInit()`, `pbrtWorldBegin()`, and `pbrtCleanup()`.

```
⟨API Static Data⟩ ≡
    #define STATE_UNINITIALIZED  0
    #define STATE_OPTIONS_BLOCK  1
    #define STATE_WORLD_BLOCK    2
    static int currentApiState = STATE_UNINITIALIZED;
```

pbrtInit() 1051
pbrtWorldBegin() 1059
pbrtWorldEnd() 1071
STATE_UNINITIALIZED 1052

Now we can start to define the implementation of `pbrtInit()`. `pbrtInit()` first makes sure that it hasn't already been called and then sets the `currentApiState` variable to `STATE_OPTIONS_BLOCK` to indicate that the scenewide options can be specified.

⟨*API Initialization*⟩ ≡ 1051

```
if (currentApiState != STATE_UNINITIALIZED)
    Error("pbrtInit() has already been called.");
currentApiState = STATE_OPTIONS_BLOCK;
```

Similarly, `pbrtCleanup()` makes sure that `pbrtInit()` has been called and that we're not in the middle of a `pbrtWorldBegin()`/`pbrtWorldEnd()` block before resetting the state to the uninitialized state.

⟨*API Cleanup*⟩ ≡ 1052

```
if (currentApiState == STATE_UNINITIALIZED)
    Error("pbrtCleanup() called without pbrtInit().");
else if (currentApiState == STATE_WORLD_BLOCK)
    Error("pbrtCleanup() called while inside world block.");
currentApiState = STATE_UNINITIALIZED;
```

All API procedures that are only valid in particular states invoke a state verification macro like `VERIFY_INITIALIZED()`, to make sure that `currentApiState` holds an appropriate value. If the states don't match, an error message is printed and the function immediately returns.

⟨*API Macros*⟩ ≡

```
#define VERIFY_INITIALIZED(func) \
if (currentApiState == STATE_UNINITIALIZED) { \
    Error("pbrtInit() must be before calling \"%s()\". " \
          "Ignoring.", func); \
    return; \
} else /* swallow trailing semicolon */
```

The implementations of `VERIFY_OPTIONS()` and `VERIFY_WORLD()` are analogous.

B.2.2 TRANSFORMATIONS

As the scene is being described, `pbrt` maintains *current transformation matrices* (CTMs), one for each of a number of points in time. If the transformations are different, then they describe an animated transformation. (Recall, for example, that the `AnimatedTransform` class defined in Section 2.9.3 stores two transformation matrices for two given times.) A number of API calls are available to modify the CTMs; when objects like shapes, cameras, and lights are created, the CTMs are passed to their constructor to define the transformation from their local coordinate system to world space.

The code below stores two CTMs in the module-local `curTransform` variable. They are represented by the `TransformSet` class, to be defined shortly, which stores a fixed number of transformations. The `activeTransformBits` variable is a bit-vector indicating which of the CTMs are active; the active transforms are updated when the transformation-related API calls are made, while the others are unchanged. This mechanism allows the user to modify some of the CTMs selectively in order to define animated transformations.

`activeTransformBits` 1054
`AnimatedTransform` 96
`currentApiState` 1052
`curTransform` 1054
`Error()` 1005
`pbrtInit()` 1051
`STATE_OPTIONS_BLOCK` 1052
`STATE_UNINITIALIZED` 1052
`STATE_WORLD_BLOCK` 1052
`TransformSet` 1054

⟨*API Static Data*⟩ +≡
```
static TransformSet curTransform;
static int activeTransformBits = ALL_TRANSFORMS_BITS;
```

The implementation here just stores two transformation matrices, one that defines the CTM for the starting time (provided via the `pbrtTransformTimes()` call, defined in a few pages), and the other for the ending time.

⟨*API Local Classes*⟩ ≡
```
#define MAX_TRANSFORMS 2
#define START_TRANSFORM_BITS (1 << 0)
#define END_TRANSFORM_BITS   (1 << 1)
#define ALL_TRANSFORMS_BITS  ((1 << MAX_TRANSFORMS) - 1)
```

`TransformSet` is a small utility class that stores an array of transformations and provides some utility routines for managing them.

⟨*API Local Classes*⟩ +≡
```
struct TransformSet {
    ⟨TransformSet Public Methods 1054⟩
private:
    Transform t[MAX_TRANSFORMS];
};
```

An accessor function is provided to access the individual `Transform`s.

⟨*TransformSet Public Methods*⟩ ≡ **1054**
```
Transform &operator[](int i) {
    return t[i];
}
```

The `Inverse()` method returns a new `TransformSet` that holds the inverses of the individual `Transform`s.

⟨*TransformSet Public Methods*⟩ +≡ **1054**
```
friend TransformSet Inverse(const TransformSet &ts) {
    TransformSet t2;
    for (int i = 0; i < MAX_TRANSFORMS; ++i)
        t2.t[i] = Inverse(ts.t[i]);
    return t2;
}
```

The actual transformation functions are straightforward. Because the CTM is used both for the rendering options and the scene description phases, these routines only need to verify that `pbrtInit()` has been called.

⟨*API Function Definitions*⟩ +≡
```
void pbrtIdentity() {
    VERIFY_INITIALIZED("Identity");
    FOR_ACTIVE_TRANSFORMS(curTransform[i] = Transform();)
}
```

`ALL_TRANSFORMS_BITS` 1054
`curTransform` 1054
`FOR_ACTIVE_TRANSFORMS()` 1055
`Inverse()` 1021
`MAX_TRANSFORMS` 1054
`pbrtInit()` 1051
`pbrtTransformTimes()` 1057
`Transform` 76
`TransformSet` 1054
`VERIFY_INITIALIZED` 1053

The `FOR_ACTIVE_TRANSFORMS()` macro encapsulates the logic for determining which of the CTMs is active. The given expression is evaluated only for the active transforms.

⟨*API Macros*⟩ +≡
```
#define FOR_ACTIVE_TRANSFORMS(expr) \
    for (int i = 0; i < MAX_TRANSFORMS; ++i) \
        if (activeTransformBits & (1 << i)) { expr }
```

⟨*API Function Definitions*⟩ +≡
```
void pbrtTranslate(float dx, float dy, float dz) {
    VERIFY_INITIALIZED("Translate");
    FOR_ACTIVE_TRANSFORMS(curTransform[i] =
        curTransform[i] * Translate(Vector(dx, dy, dz));)
}
```

Most of the rest of the functions are similarly defined, so we will not show their definitions here. `pbrt` also provides `pbrtConcatTransform()` and `pbrtTransform()` functions to allow the user to specify an arbitrary matrix to postmultiply or replace the active CTM(s), respectively.

⟨*API Function Declarations*⟩ ≡
```
void pbrtRotate(float angle, float ax, float ay, float az);
void pbrtScale(float sx, float sy, float sz);
void pbrtLookAt(float ex, float ey, float ez,
                float lx, float ly, float lz,
                float ux, float uy, float uz);
void pbrtConcatTransform(float transform[16]);
void pbrtTransform(float transform[16]);
```

It can be useful to make a named copy of the CTM so that it can be referred to later. For example, to place a light at the camera's position, it is useful to first apply the transformation into the camera coordinate system, since then the light can just be placed at the origin (0, 0, 0). This way, if the camera position is changed and the scene is rerendered, the light will move with it. The `pbrtCoordinateSystem()` function copies the current `TransformSet` into the `namedCoordinateSystems` associative array, and `pbrtCoordSysTransform()` loads a named set of CTMs.

⟨*API Static Data*⟩ +≡
```
static map<string, TransformSet> namedCoordinateSystems;
```

⟨*API Function Definitions*⟩ +≡
```
void pbrtCoordinateSystem(const string &name) {
    VERIFY_INITIALIZED("CoordinateSystem");
    namedCoordinateSystems[name] = curTransform;
}
```

⟨*API Function Definitions*⟩ +≡

```
void pbrtCoordSysTransform(const string &name) {
    VERIFY_INITIALIZED("CoordSysTransform");
    if (namedCoordinateSystems.find(name) !=
        namedCoordinateSystems.end())
        curTransform = namedCoordinateSystems[name];
    else
        Warning("Couldn't find named coordinate system \"%s\"",
                name.c_str());
}
```

B.2.3 OPTIONS

All of the rendering options that are set before the `pbrtWorldBegin()` call are stored in a `RenderOptions` structure. This structure contains public data members that are set by API calls and methods that help create objects used by the rest of `pbrt` for rendering.

⟨*API Local Classes*⟩ +≡

```
struct RenderOptions {
    ⟨RenderOptions Public Methods 1072⟩
    ⟨RenderOptions Public Data 1057⟩
};
```

⟨*API Local Classes*⟩ +≡

```
RenderOptions::RenderOptions() {
    ⟨RenderOptions Constructor Implementation 1057⟩
}
```

A single `static` instance of a `RenderOptions` structure is available to the rest of the API functions:

⟨*API Static Data*⟩ +≡

```
static RenderOptions *renderOptions = NULL;
```

When `pbrtInit()` is called, it allocates a `RenderOptions` structure that initially holds default values for all of its options:

⟨*API Initialization*⟩ +≡ 1051

```
renderOptions = new RenderOptions;
```

The `renderOptions` variable is freed by `pbrtCleanup()`:

⟨*API Cleanup*⟩ +≡ 1052

```
delete renderOptions;
renderOptions = NULL;
```

A few calls are available to indicate which of the CTMs are active.

curTransform 1054
namedCoordinateSystems 1055
pbrtInit() 1051
pbrtWorldBegin() 1059
RenderOptions 1056
renderOptions 1056
VERIFY_INITIALIZED 1053
Warning() 1005

⟨*API Function Definitions*⟩ +≡
```
void pbrtActiveTransformAll() {
    activeTransformBits = ALL_TRANSFORMS_BITS;
}
void pbrtActiveTransformEndTime() {
    activeTransformBits = END_TRANSFORM_BITS;
}
void pbrtActiveTransformStartTime() {
    activeTransformBits = START_TRANSFORM_BITS;
}
```

The two times at which the two CTMs are defined are provided via `pbrtTransformTimes()`. By default, the start time is zero and the end time is one.

⟨*API Function Definitions*⟩ +≡
```
void pbrtTransformTimes(float start, float end) {
    VERIFY_OPTIONS("TransformTimes");
    renderOptions->transformStartTime = start;
    renderOptions->transformEndTime = end;
}
```

⟨*RenderOptions Public Data*⟩ ≡ **1056**
```
float transformStartTime, transformEndTime;
```

The API functions for setting the rest of the rendering options are mostly similar in both their interface and their implementation. For example, `pbrtPixelFilter()` specifies the kind of `Filter` to be used for filtering image samples. It takes two parameters: a string giving the name of the filter to use and a `ParamSet` giving the parameters to the filter. When this function is called, all that needs to be done is to verify that the API is in an appropriate state for `pbrtPixelFilter()` to be called and store the name of the filter and its parameters in `renderOptions`.

⟨*API Function Definitions*⟩ +≡
```
void pbrtPixelFilter(const string &name, const ParamSet &params) {
    VERIFY_OPTIONS("PixelFilter");
    renderOptions->FilterName = name;
    renderOptions->FilterParams = params;
}
```

`activeTransformBits`	1054
`ALL_TRANSFORMS_BITS`	1054
`END_TRANSFORM_BITS`	1054
`Filter`	393
`ParamSet`	1047
`pbrtPixelFilter()`	1057
`pbrtTransformTimes()`	1057
`renderOptions`	1056
`RenderOptions::FilterName`	1057
`RenderOptions::FilterParams`	1057
`START_TRANSFORM_BITS`	1054

⟨*RenderOptions Public Data*⟩ +≡ **1056**
```
string FilterName;
ParamSet FilterParams;
```

The default filter function is set to the Box filter. If no specific filter is specified in the scene description file, then, since the default `ParamSet` has no key-value pairs, the filter will be created based on its default parameter settings.

⟨*RenderOptions Constructor Implementation*⟩ ≡ **1056**
```
FilterName = "box";
```

Most of the rest of the rendering option setting API calls are similar; they simply store their arguments in `renderOptions`. Therefore, we will only include the declarations of these functions here. The options controlled by each function should be apparent from its name; more information about the legal parameters to each of these routines can be found in the `docs/fileformat.pdf` file.

⟨*API Function Declarations*⟩ +≡
```
void pbrtFilm(const string &type, const ParamSet &params);
void pbrtSampler(const string &name, const ParamSet &params);
void pbrtAccelerator(const string &name, const ParamSet &params);
void pbrtSurfaceIntegrator(const string &name, const ParamSet &params);
void pbrtVolumeIntegrator(const string &name, const ParamSet &params);
void pbrtRenderer(const string &name, const ParamSet &params);
```

`pbrtCamera()` is slightly different than the other options, since the world-to-camera transformation needs to be recorded. The CTM is used by `pbrtCamera()` to initialize this value, and the camera coordinate system transformation is also stored for possible future use by `pbrtCoordSysTransform()`.

⟨*RenderOptions Public Data*⟩ +≡ **1056**
```
string CameraName;
ParamSet CameraParams;
TransformSet CameraToWorld;
```

⟨*API Function Definitions*⟩ +≡
```
void pbrtCamera(const string &name, const ParamSet &params) {
    VERIFY_OPTIONS("Camera");
    renderOptions->CameraName = name;
    renderOptions->CameraParams = params;
    renderOptions->CameraToWorld = Inverse(curTransform);
    namedCoordinateSystems["camera"] = renderOptions->CameraToWorld;
}
```

The default camera is set to use a perspective projection:

⟨*RenderOptions Constructor Implementation*⟩ +≡ **1056**
```
CameraName = "perspective";
```

B.3 SCENE DEFINITION

After the user has set up the overall rendering options, the `pbrtWorldBegin()` call marks the start of the description of the shapes, materials, and lights in the scene. It sets the current rendering state to `STATE_WORLD_BLOCK`, resets the CTMs to identity matrices, and enables all of the CTMs.

`curTransform` 1054
`Inverse()` 1021
`namedCoordinateSystems` 1055
`ParamSet` 1047
`pbrtCamera()` 1058
`pbrtCoordSysTransform()` 1056
`pbrtWorldBegin()` 1059
`renderOptions` 1056
`RenderOptions::CameraName` 1058
`RenderOptions::CameraParams` 1058
`RenderOptions::CameraToWorld` 1058
`STATE_WORLD_BLOCK` 1052
`TransformSet` 1054

⟨*API Function Definitions*⟩ +≡
```
void pbrtWorldBegin() {
    VERIFY_OPTIONS("WorldBegin");
    currentApiState = STATE_WORLD_BLOCK;
    for (int i = 0; i < MAX_TRANSFORMS; ++i)
        curTransform[i] = Transform();
    activeTransformBits = ALL_TRANSFORMS_BITS;
    namedCoordinateSystems["world"] = curTransform;
}
```

B.3.1 HIERARCHICAL GRAPHICS STATE

As the scene's lights, geometry, and participating media are specified, a variety of attributes can be set as well. In addition to the CTMs, these include information about textures and the current material. When a geometric primitive or light source is then added to the scene, the current attributes are used when creating the corresponding object. These data are all known as the *graphics state.*

It is useful for a rendering API to provide some functionality for managing the graphics state. `pbrt` has API calls that allow the current graphics state to be managed with an *attribute stack*; the user can push the current set of attributes, make changes to their values, and then later pop back to the previously pushed attribute values. For example, a scene description file might contain the following:

```
Material "matte"
AttributeBegin
  Material "plastic"
  Translate 5 0 0
  Shape "sphere" "float radius" [1]
AttributeEnd
Shape "sphere" "float radius" [1]
```

The first sphere is affected by the translation and is bound to the plastic material, while the second sphere is matte and isn't translated. Changes to attributes made inside a `pbrtAttributeBegin()`/`pbrtAttributeEnd()` block are forgotten at the end of the block. Being able to save and restore attributes in this manner is a classic idiom for scene description in computer graphics.

The graphics state is stored in the `GraphicsState` structure. As was done previously with `RenderOptions`, we'll be adding members to it throughout this section.

⟨*API Local Classes*⟩ +≡
```
struct GraphicsState {
    ⟨Graphics State Methods 1067⟩
    ⟨Graphics State 1064⟩
};
```

activeTransformBits 1054
ALL_TRANSFORMS_BITS 1054
currentApiState 1052
curTransform 1054
GraphicsState 1059
MAX_TRANSFORMS 1054
namedCoordinateSystems 1055
pbrtAttributeBegin() 1060
pbrtAttributeEnd() 1060
RenderOptions 1056
STATE_WORLD_BLOCK 1052
Transform 76

```
⟨API Local Classes⟩ +≡
    GraphicsState::GraphicsState() {
        ⟨GraphicsState Constructor Implementation 1064⟩
    }
```

When `pbrtInit()` is called, the current graphics state is initialized to hold default values.

```
⟨API Initialization⟩ +≡                                                    1051
    graphicsState = GraphicsState();
```

A `vector` of `GraphicsStates` is used as a stack to perform hierarchical state management. When `pbrtAttributeBegin()` is called, the current `GraphicsState` is copied and pushed onto this stack. `pbrtAttributeEnd()` then simply pops the state from this stack.

```
⟨API Function Definitions⟩ +≡
    void pbrtAttributeBegin() {
        VERIFY_WORLD("AttributeBegin");
        pushedGraphicsStates.push_back(graphicsState);
        pushedTransforms.push_back(curTransform);
        pushedActiveTransformBits.push_back(activeTransformBits);
    }
```

```
⟨API Static Data⟩ +≡
    static GraphicsState graphicsState;
    static vector<GraphicsState> pushedGraphicsStates;
    static vector<TransformSet> pushedTransforms;
    static vector<uint32_t> pushedActiveTransformBits;
```

`pbrtAttributeEnd()` also verifies that we do not have stack underflow by checking to see if there are any entries on the stack.

```
⟨API Function Definitions⟩ +≡
    void pbrtAttributeEnd() {
        VERIFY_WORLD("AttributeEnd");
        if (!pushedGraphicsStates.size()) {
            Error("Unmatched pbrtAttributeEnd() encountered. "
                  "Ignoring it.");
            return;
        }
        graphicsState = pushedGraphicsStates.back();
        pushedGraphicsStates.pop_back();
        curTransform = pushedTransforms.back();
        pushedTransforms.pop_back();
        activeTransformBits = pushedActiveTransformBits.back();
        pushedActiveTransformBits.pop_back();
    }
```

activeTransformBits	1054
curTransform	1054
Error()	1005
GraphicsState	1059
graphicsState	1060
pbrtAttributeBegin()	1060
pbrtAttributeEnd()	1060
pbrtInit()	1051
pbrtTransformBegin()	1061
pbrtTransformEnd()	1061
pushedActiveTransformBits	1060
pushedGraphicsStates	1060
pushedTransforms	1060
TransformSet	1054

We also provide the `pbrtTransformBegin()` and `pbrtTransformEnd()` calls. These functions are similar to `pbrtAttributeBegin()` and `pbrtAttributeEnd()`, except that they only push and pop the CTMs. We frequently want to apply a transformation to a texture, but since the list of named textures is stored in the graphics state, we cannot use pbrtAttributeBegin() to save the transformation matrix. Since the implementations of pbrtTransformBegin() and pbrtTransformEnd() are very similar to pbrtAttributeBegin() and pbrtAttributeEnd(), respectively, they are not shown here.

```
⟨API Function Declarations⟩ +≡
    void pbrtTransformBegin();
    void pbrtTransformEnd();
```

B.3.2 TEXTURE AND MATERIAL PARAMETERS

Recall that all of the materials in pbrt use textures to describe all of their parameters. For example, the diffuse color of the matte material class is always obtained from a texture, even if the material is intended to have a constant reflectivity (in which case a ConstantTexture is used).

Before a material can be created, it is necessary to create these textures to pass to the material creation procedures. Textures can be either explicitly created and later referred to by name or implicitly created on the fly to encapsulate a constant parameter. These two methods of texture creation are hidden by the TextureParams class.

```
⟨TextureParams Declarations⟩ ≡
    class TextureParams {
    public:
        ⟨TextureParams Public Methods 1061⟩
    private:
        ⟨TextureParams Private Data 1061⟩
    };
```

The TextureParams class contains associative arrays of previously defined named float and Spectrum textures, as well as two ParamSets that will be searched for named textures.

```
⟨TextureParams Private Data⟩ ≡                                              1061
    map<string, Reference<Texture<float> > > &floatTextures;
    map<string, Reference<Texture<Spectrum> > > &spectrumTextures;
    const ParamSet &geomParams, &materialParams;
```

```
⟨TextureParams Public Methods⟩ ≡                                            1061
    TextureParams(const ParamSet &geomp, const ParamSet &matp,
                  map<string, Reference<Texture<float> > > &ft,
                  map<string, Reference<Texture<Spectrum> > > &st)
        : floatTextures(ft), spectrumTextures(st),
          geomParams(geomp), materialParams(matp) {
    }
```

ConstantTexture 520
ParamSet 1047
pbrtAttributeBegin() 1060
pbrtAttributeEnd() 1060
pbrtTransformBegin() 1061
pbrtTransformEnd() 1061
Reference 1011
Spectrum 263
Texture 519
TextureParams 1061
TextureParams::GetSpectrumTexture() 1062

Here we will show the code for finding a texture of Spectrum type; the code for finding a float texture is analogous. The TextureParams::GetSpectrumTexture() method takes a parameter name (e.g., “Kd”) , as well as a default Spectrum value. If no texture has been explicitly specified for the parameter, a constant texture will be created that returns the default spectrum value.

Finding the texture is performed in several stages; the order of these stages is crucial. First, the parameter list from the `Shape` for which a `Material` is being created is searched for a named reference to an explicitly defined texture. If no such texture is found, then the material parameters are searched. Finally, if no explicit texture has been found, the two parameter lists are searched in turn for supplied constant values. If no such constants are found, the default is used.

The order of these steps is crucial, because `pbrt` allows a shape to override individual elements of the material that is bound to it. For example, the user should be able to create a scene description that contains the lines

```
Material "matte" "color Kd" [ 1 0 0 ]
Shape "sphere" "color Kd" [ 0 1 0 ]
```

These two commands should create a green matte sphere. Because the shape's parameter list is searched first, the `Kd` parameter from the `Shape` command will be used when the `MatteMaterial` constructor is called.

⟨*TextureParams Method Definitions*⟩ ≡
```
Reference<Texture<Spectrum> >
TextureParams::GetSpectrumTexture(const string &n,
                                  const Spectrum &def) const {
    string name = geomParams.FindTexture(n);
    if (name == "") name = materialParams.FindTexture(n);
    if (name != "") {
        if (spectrumTextures.find(name) != spectrumTextures.end())
            return spectrumTextures[name];
        else
            Error("Couldn't find spectrum texture named \"%s\" "
                  "for parameter \"%s\"", name.c_str(), n.c_str());
    }
    Spectrum val = geomParams.FindOneSpectrum(n,
                            materialParams.FindOneSpectrum(n, def));
    return new ConstantTexture<Spectrum>(val);
}
```

Because an instance of the `TextureParams` class is passed to material creation routines that might need to access nontexture parameter values, we also provide ways to access the other parameter list types. These methods return parameters from the geometric parameter list, if found. Otherwise, the material parameter list is searched, and finally the default value is returned.

The `TextureParams::FindFloat()` method is shown here. The other access methods are similar and omitted.

⟨*TextureParams Public Methods*⟩ +≡ 1061
```
float FindFloat(const string &n, float d) const {
    return geomParams.FindOneFloat(n, materialParams.FindOneFloat(n, d));
}
```

ConstantTexture 520
Error() 1005
Material 483
MatteMaterial 484
ParamSet::FindOneFloat() 1050
ParamSet::FindOneSpectrum() 1050
ParamSet::FindTexture() 1050
Reference 1011
Shape 108
Spectrum 263
Texture 519
TextureParams 1061
TextureParams::FindFloat() 1062
TextureParams::geomParams 1061
TextureParams::materialParams 1061
TextureParams::spectrumTextures 1061

B.3.3 SURFACE AND MATERIAL DESCRIPTION

The pbrtTexture() method creates a named texture that can be referred to later. In addition to the texture name, its *type* is specified. Currently, pbrt supports only “float” and “color” as texture types. The supplied parameter list is used to create a TextureParams object, which will be passed to the desired texture’s creation routine.

⟨*API Function Definitions*⟩ +≡
```
void pbrtTexture(const string &name, const string &type,
                 const string &texname, const ParamSet &params) {
    VERIFY_WORLD("Texture");
    TextureParams tp(params, params, graphicsState.floatTextures,
                     graphicsState.spectrumTextures);
    if (type == "float")  {
        ⟨Create float texture and store in floatTextures 1063⟩
    }
    else if (type == "color")  {
        ⟨Create color texture and store in spectrumTextures⟩
    }
    else
        Error("Texture type \"%s\" unknown.", type.c_str());
}
```

Creating the texture is simple. This function first checks to see if a texture of the same name and type already exists and issues a warning if so. Then, the MakeFloatTexture() routine calls the creation function for the appropriate Texture implementation, and the returned texture class is added to the GraphicsState::floatTextures associative array. The code for creating a color texture is similar and not shown.

⟨*Create* float *texture and store in* floatTextures⟩ ≡ 1063
```
if (graphicsState.floatTextures.find(name) !=
    graphicsState.floatTextures.end())
    Warning("Texture \"%s\" being redefined", name.c_str());
WARN_IF_ANIMATED_TRANSFORM("Texture");
Reference<Texture<float> > ft = MakeFloatTexture(texname,
                                                 curTransform[0], tp);
if (ft) graphicsState.floatTextures[name] = ft;
```

curTransform 1054
Error() 1005
graphicsState 1060
GraphicsState::floatTextures 1064
GraphicsState::spectrumTextures 1064
ParamSet 1047
pbrtTexture() 1063
Reference 1011
Texture 519
TextureParams 1061
TransformSet::IsAnimated() 1064
Warning() 1005
WARN_IF_ANIMATED_TRANSFORM() 1063

Not all of the types in pbrt that take transformations support animated transformations. Textures are one example; it’s not worth the additional code complexity to support them, especially since the utility an animated texture transform brings isn’t obvious. The WARN_IF_ANIMATED_TRANSFORM() macro warns if the CTMs are different, indicating animated transformations, for classes that don’t actualy use more than a single transformation.

⟨*API Macros*⟩ +≡
```
#define WARN_IF_ANIMATED_TRANSFORM(func) \
do { if (curTransform.IsAnimated()) \
         Warning("Animated transformations set; ignoring for \"%s\"" \
                 "and using the start transform only", func); \
} while (false)
```

⟨*TransformSet Public Methods*⟩ +≡ 1054
```
bool IsAnimated() const {
    for (int i = 0; i < MAX_TRANSFORMS-1; ++i)
        if (t[i] != t[i+1]) return true;
    return false;
}
```

⟨*Graphics State*⟩ ≡ 1059
```
map<string, Reference<Texture<float> > > floatTextures;
map<string, Reference<Texture<Spectrum> > > spectrumTextures;
```

The current material is specified by a call to `pbrtMaterial()`. Its `ParamSet` is stored until a `Material` object needs to be created later when a shape is specified.

⟨*API Function Declarations*⟩ +≡
```
void pbrtMaterial(const string &name, const ParamSet &params);
```

⟨*Graphics State*⟩ +≡ 1059
```
ParamSet materialParams;
string material;
```

The default material for all objects in the scene is matte:

⟨*GraphicsState Constructor Implementation*⟩ ≡ 1060
```
material = "matte";
```

`pbrt` also supports the notion of creating a `Material` with a given set of parameters and then associating an arbitrary name with the combination of material and parameter settings. `pbrtMakeNamedMaterial()` creates such an association, and `pbrtNamedMaterial()` sets the current material and material parameters based on a previously defined named material.

⟨*API Function Declarations*⟩ +≡
```
void pbrtMakeNamedMaterial(const string &name, const ParamSet &params);
void pbrtNamedMaterial(const string &name);
```

⟨*Graphics State*⟩ +≡ 1059
```
map<string, Reference<Material> > namedMaterials;
string currentNamedMaterial;
```

B.3.4 LIGHT SOURCES

`pbrt`'s API provides two ways for the user to specify light sources for the scene. The first, `pbrtLightSource()`, defines a light source that doesn't have geometry associated with it (e.g., a point light or a directional light). The second, `pbrtAreaLightSource()`, specifies an area light source. All shape specifications that appear between an area light source call up to the end of the current attribute block are treated as emissive.

⟨*API Function Definitions*⟩ +≡

```
void pbrtLightSource(const string &name, const ParamSet &params) {
    VERIFY_WORLD("LightSource");
    WARN_IF_ANIMATED_TRANSFORM("LightSource");
    Light *lt = MakeLight(name, curTransform[0], params);
    if (lt == NULL)
        Error("pbrtLightSource: light type \"%s\" unknown.", name.c_str());
    else
        renderOptions->lights.push_back(lt);
}
```

⟨*RenderOptions Public Data*⟩ +≡ **1056**

```
vector<Light *> lights;
```

When an area light is specified via `pbrtAreaLightSource()`, it can't be created immediately since the shapes to follow are needed to define the light source's geometry. Therefore, this function just saves the name of the area light source type and the parameters given to it.

⟨*Graphics State*⟩ +≡ **1059**

```
ParamSet areaLightParams;
string areaLight;
```

⟨*API Function Definitions*⟩ +≡

```
void pbrtAreaLightSource(const string &name,
                         const ParamSet &params) {
    VERIFY_WORLD("AreaLightSource");
    graphicsState.areaLight = name;
    graphicsState.areaLightParams = params;
}
```

B.3.5 SHAPES AND VOLUME REGIONS

The `pbrtShape()` function creates a new `Shape` object and adds it to the scene. This function is relatively complicated, in that it has to create area light sources for the shape if an area light is being defined, it has to handle animated and static shapes differently, and it also has to deal with creating object instances when needed.

⟨*API Function Definitions*⟩ +≡

```
void pbrtShape(const string &name, const ParamSet &params) {
    VERIFY_WORLD("Shape");
    Reference<Primitive> prim;
    AreaLight *area = NULL;
    if (!curTransform.IsAnimated()) {
        ⟨Create primitive for static shape 1066⟩
    } else {
        ⟨Create primitive for animated shape 1067⟩
    }
    ⟨Add primitive to scene or current instance 1068⟩
}
```

AreaLight 623
curTransform 1054
Error() 1005
graphicsState 1060
Light 606
MakeLight() 1073
ParamSet 1047
pbrtAreaLightSource() 1065
pbrtShape() 1065
Primitive 185
Reference 1011
renderOptions 1056
RenderOptions::lights 1065
Shape 108
TransformSet::IsAnimated() 1064

Shapes that are animated are represented with `TransformedPrimitives`, which include extra functionality to use `AnimatedTransforms`, while shapes that aren't animated use `GeometricPrimitives`. Therefore, there are two code paths here for those two cases.

The static shape case is mostly a matter of creating the appropriate `Shape` and `Material` to make a `GeometricPrimitive` and then allocating a `GeometricPrimitive`. `MakeShape()` handles the details of creating the shape corresponding to the given shape name, passing the `ParamSet` along to its creation routine. The code below uses a `TransformCache` (defined shortly), which allocates and stores a single `Transform` pointer for each unique transformation that is passed to its `Lookup()` method. In this way, if many shapes in the scene have the same transformation matrix, a single `Transform` pointer can be shared among all of them.

```
⟨Create primitive for static shape⟩ ≡                                    1065
   Transform *obj2world, *world2obj;
   transformCache.Lookup(curTransform[0], &obj2world, &world2obj);
   Reference<Shape> shape = MakeShape(name, obj2world, world2obj,
       graphicsState.reverseOrientation, params);
   if (!shape) return;
   Reference<Material> mtl = graphicsState.CreateMaterial(params);
   params.ReportUnused();
   ⟨Possibly create area light for shape 1067⟩
   prim = new GeometricPrimitive(shape, mtl, area);
```

`TransformCache` is a small wrapper around an associative array from transformations to pairs of `Transform` pointers; the first pointer is equal to the transform and the second is its inverse. The `Lookup()` method just looks for the given transformation in the cache, allocates space for it and stores it and its inverse if not found, and returns the appropriate pointers.

```
⟨TransformCache Private Data⟩ ≡
   map<Transform, std::pair<Transform *, Transform *> > cache;
   MemoryArena arena;
```

```
⟨API Static Data⟩ +≡
   static TransformCache transformCache;
```

The `MakeShape()` function takes the name of the shape to be created, the CTMs, and the `ParamSet` for the new shape. It calls an appropriate shape creation function based on the shape name provided. Here only the sphere creation code is shown; code for the rest of the shapes is equivalent.

```
⟨Object Creation Function Definitions⟩ ≡
   Reference<Shape> MakeShape(const string &name,
           const Transform *object2world, const Transform *world2object,
           bool reverseOrientation, const ParamSet &paramSet) {
       Shape *s = NULL;
```

AnimatedTransform	96
CreateSphereShape()	1073
curTransform	1054
GeometricPrimitive	188
graphicsState	1060
GraphicsState::CreateMaterial()	1067
MakeShape()	1066
Material	483
MemoryArena	1015
ParamSet	1047
ParamSet::ReportUnused()	1051
Reference	1011
Shape	108
Transform	76
TransformCache	1066
transformCache	1066
TransformCache::Lookup()	1066
TransformedPrimitive	190

```
        if (name == "sphere")
            s = CreateSphereShape(object2world, world2object,
                                  reverseOrientation, paramSet);
        ⟨Create remaining Shape types⟩
        paramSet.ReportUnused();
        return s;
    }
```

The `Material` for the shape is created by the `MakeMaterial()` call; its implementation is analogous to that of `MakeShape()`. If the specified material cannot be found (usually due to a typo in the material name), a matte material is created and a warning is issued.

```
⟨Graphics State Methods⟩ ≡                                                1059
    Reference<Material> CreateMaterial(const ParamSet &params);
```

If an area light has been set in the current graphics state by `pbrtAreaLightSource()`, the new shape is an emitter and an `AreaLight` needs to be made for it.

```
⟨Possibly create area light for shape⟩ ≡                                  1066
    if (graphicsState.areaLight != "") {
        area = MakeAreaLight(graphicsState.areaLight, curTransform[0],
                             graphicsState.areaLightParams, shape);
    }
```

If the transformation matrices are animated, the task is a little more complicated. A base `Shape` and `GeometricPrimitive` are first created. Recall from Section 4.1.2 that the `TransformedPrimitive` holds only a single `Primitive` and requires that it supports ray intersection queries; if the shape is not immediately intersectable, it is refined into intersectable primitives and a BVH is created, yielding a single primitive for the `TransformedPrimitive` constructor.

```
⟨Create primitive for animated shape⟩ ≡                                   1065
    ⟨Create initial Shape for animated shape 1068⟩
    ⟨Get animatedWorldToObject transform for shape 1068⟩
    Reference<Primitive> baseprim = new GeometricPrimitive(shape, mtl, NULL);
    if (!baseprim->CanIntersect()) {
        ⟨Refine animated shape and create BVH if more than one shape created 1068⟩
    }
    prim = new TransformedPrimitive(baseprim, animatedWorldToObject);
```

Because the `Shape` class doesn't handle animated transformations, here the initial `Shape` for animated primitives is created with an identity transform passed to the shape creation routine. All of the details related to the shape's transformation are managed with the `TransformedPrimitive` that ends up holding the shape. Animated transformations for light sources aren't currently supported in `pbrt`; thus, if an animated transform has been specified with an area light source, a warning is issued here.

AreaLight	623
curTransform	1054
GeometricPrimitive	188
graphicsState	1060
GraphicsState::areaLight	1065
GraphicsState::areaLightParams	1065
MakeShape()	1066
Material	483
ParamSet	1047
pbrtAreaLightSource()	1065
Primitive	185
Primitive::CanIntersect()	186
Reference	1011
Shape	108
TransformedPrimitive	190

⟨*Create initial* `Shape` *for animated shape*⟩ ≡ 1067

```
if (graphicsState.areaLight != "")
    Warning("Ignoring currently set area light when creating "
            "animated shape");
Transform *identity;
transformCache.Lookup(Transform(), &identity, NULL);
Reference<Shape> shape = MakeShape(name, identity, identity,
    graphicsState.reverseOrientation, params);
if (!shape) return;
Reference<Material> mtl = graphicsState.CreateMaterial(params);
params.ReportUnused();
```

Again the `TransformCache` is used, here to get transformations for the start and end time, which are then passed to the `AnimatedTransform` constructor.

⟨*Get* `animatedWorldToObject` *transform for shape*⟩ ≡ 1067

```
Transform *world2obj[2];
transformCache.Lookup(curTransform[0], NULL, &world2obj[0]);
transformCache.Lookup(curTransform[1], NULL, &world2obj[1]);
AnimatedTransform
     animatedWorldToObject(world2obj[0], renderOptions->transformStartTime,
                           world2obj[1], renderOptions->transformEndTime);
```

⟨*Refine animated shape and create BVH if more than one shape created*⟩ ≡ 1067

```
vector<Reference<Primitive> > refinedPrimitives;
baseprim->FullyRefine(refinedPrimitives);
if (refinedPrimitives.size() == 0) return;
if (refinedPrimitives.size() > 1)
    baseprim = new BVHAccel(refinedPrimitives);
else
    baseprim = refinedPrimitives[0];
```

If the user is in the middle of defining an object instance, `pbrtObjectBegin()` (defined in the following section) will have set the `currentInstance` member of `renderOptions` to point to a `vector` that is collecting the shapes that define the instance. In that case, the new shape is added to that array. Otherwise, the shape is stored in the `RenderOptions::primitives` array. This array will eventually be passed to the `Scene` constructor. If this primitive is an area light, it is also added to the `RenderOptions::lights` array, just as the `pbrtLightSource()` call also does.

⟨*RenderOptions Public Data*⟩ +≡ 1056

```
vector<Reference<Primitive> > primitives;
```

⟨*Add primitive to scene or current instance*⟩ ≡ 1065

```
if (renderOptions->currentInstance) {
    if (area)
        Warning("Area lights not supported with object instancing");
    renderOptions->currentInstance->push_back(prim);
}
```

```
else {
    renderOptions->primitives.push_back(prim);
    if (area != NULL) {
        renderOptions->lights.push_back(area);
    }
}
```

B.3.6 OBJECT INSTANCING

All shapes that are specified between a `pbrtObjectBegin()` and `pbrtObjectEnd()` pair are used to create a named object instance (see the discussion of object instancing and the `TransformedPrimitive` class in Section 4.1.2). `pbrtObjectBegin()` sets `RenderOptions::currentInstance` so that subsequent `pbrtShape()` calls can add the shape to this instance's vector of primitive references. This function also pushes the graphics state, so that any changes made to the CTMs or other state while defining the instance don't last beyond the instance definition.

⟨*API Function Definitions*⟩ +≡

```
void pbrtObjectBegin(const string &name) {
    VERIFY_WORLD("ObjectBegin");
    pbrtAttributeBegin();
    if (renderOptions->currentInstance)
        Error("ObjectBegin called inside of instance definition");
    renderOptions->instances[name] = vector<Reference<Primitive> >();
    renderOptions->currentInstance = &renderOptions->instances[name];
}
```

⟨*RenderOptions Public Data*⟩ +≡ **1056**

```
map<string, vector<Reference<Primitive> > > instances;
vector<Reference<Primitive> > *currentInstance;
```

⟨*RenderOptions Constructor Implementation*⟩ +≡ **1056**

```
currentInstance = NULL;
```

⟨*API Function Definitions*⟩ +≡

```
void pbrtObjectEnd() {
    VERIFY_WORLD("ObjectEnd");
    if (!renderOptions->currentInstance)
        Error("ObjectEnd called outside of instance definition");
    renderOptions->currentInstance = NULL;
    pbrtAttributeEnd();
}
```

When an instance is used in the scene, the instance's vector of `Primitives` needs to be found in the `RenderOptions::instances` map, a `TransformedPrimitive` created, and the instance added to the scene. Note that the `TransformedPrimitive` constructor takes the current transformation matrix from the time when `pbrtObjectInstance()` is called. The instance's complete world transformation is the composition of the CTM when it is instantiated with the CTM when it was originally created.

pbrtObjectInstance() first does some error checking to make sure that the instance is not being used inside the definition of another instance and also that the named instance has been defined. The error checking is simple and not shown here.

⟨*API Function Definitions*⟩ +≡
```
void pbrtObjectInstance(const string &name) {
    VERIFY_WORLD("ObjectInstance");
    ⟨Object instance error checking⟩
    vector<Reference<Primitive> > &in = renderOptions->instances[name];
    if (in.size() == 0) return;
    if (in.size() > 1 || !in[0]->CanIntersect()) {
        ⟨Refine instance Primitives and create aggregate 1070⟩
    }
    Transform *world2instance[2];
    transformCache.Lookup(curTransform[0], NULL, &world2instance[0]);
    transformCache.Lookup(curTransform[1], NULL, &world2instance[1]);
    AnimatedTransform animatedWorldToInstance(world2instance[0],
        renderOptions->transformStartTime,
        world2instance[1], renderOptions->transformEndTime);
    Reference<Primitive> prim =
        new TransformedPrimitive(in[0], animatedWorldToInstance);
    renderOptions->primitives.push_back(prim);
}
```

If there is more than one shape in an instance, or if the instance's shape must be refined before it can be intersected, then an aggregate needs to be built for it. This must be done here rather than in the TransformedPrimitive constructor so that the resulting aggregate will be reused if this instance is used multiple times in the scene.

⟨*Refine instance* Primitives *and create aggregate*⟩ ≡ 1070
```
Reference<Primitive> accel =
     MakeAccelerator(renderOptions->AcceleratorName,
                     in, renderOptions->AcceleratorParams);
if (!accel) accel = MakeAccelerator("bvh", in, ParamSet());
if (!accel) Severe("Unable to create \"bvh\" accelerator");
in.erase(in.begin(), in.end());
in.push_back(accel);
```

AnimatedTransform 96
curTransform 1054
ParamSet 1047
pbrtObjectInstance() 1070
Primitive 185
Primitive::CanIntersect() 186
Reference 1011
Renderer 24
Renderer::Render() 24
renderOptions 1056
RenderOptions::instances 1069
RenderOptions::primitives 1068
Severe() 1005
Transform 76
transformCache 1066
TransformedPrimitive 190

B.3.7 WORLD END AND RENDERING

When pbrtWorldEnd() is called, the scene has been fully specified and rendering can begin. This routine makes sure that there aren't excess graphics state structures pushed on the state stack (issuing a warning if so), creates the Scene and Renderer objects, and then calls the Renderer::Render() method.

⟨*API Function Definitions*⟩ +≡
```
void pbrtWorldEnd() {
    VERIFY_WORLD("WorldEnd");
    ⟨Ensure there are no pushed graphics states 1071⟩
    ⟨Create scene and render 1071⟩
    ⟨Clean up after rendering 1071⟩
}
```

If there are graphics states and/or transformations remaining on the respective stacks, a warning is issued for each one:

⟨*Ensure there are no pushed graphics states*⟩ ≡ 1071
```
while (pushedGraphicsStates.size()) {
    Warning("Missing end to pbrtAttributeBegin()");
    pushedGraphicsStates.pop_back();
    pushedTransforms.pop_back();
}
while (pushedTransforms.size()) {
    Warning("Missing end to pbrtTransformBegin()");
    pushedTransforms.pop_back();
}
```

Now the `RenderOptions::MakeRenderer()` and `RenderOptions::MakeScene()` methods can create the corresponding objects based on the settings provided by the user, and rendering can occur.

⟨*Create scene and render*⟩ ≡ 1071
```
Renderer *renderer = renderOptions->MakeRenderer();
Scene *scene = renderOptions->MakeScene();
if (scene && renderer) renderer->Render(scene);
TasksCleanup();
delete renderer;
delete scene;
```

Once rendering is complete, the API transitions back to the "options block" rendering state, prints out any statistics gathered by probes during rendering, and clears the CTMs and named coordinate systems so that the next frame, if any, starts with a clean slate.

⟨*Clean up after rendering*⟩ ≡ 1071
```
currentApiState = STATE_OPTIONS_BLOCK;
ProbesPrint(stdout);
for (int i = 0; i < MAX_TRANSFORMS; ++i)
    curTransform[i] = Transform();
activeTransformBits = ALL_TRANSFORMS_BITS;
namedCoordinateSystems.erase(namedCoordinateSystems.begin(),
                             namedCoordinateSystems.end());
```

Creating the Scene object is mostly a matter of creating the Aggregate for all of the primitives and calling the Scene constructor. The MakeAccelerator() function isn't included here; it's similar in structure to MakeShape() as far as using the string passed to it to determine which accelerator construction function to call.

⟨*API Function Definitions*⟩ +≡

```
Scene *RenderOptions::MakeScene() {
    ⟨Initialize volumeRegion from volume region(s) 1072⟩
    Primitive *accelerator = MakeAccelerator(AcceleratorName,
        primitives, AcceleratorParams);
    if (!accelerator)
        accelerator = MakeAccelerator("bvh", primitives, ParamSet());
    if (!accelerator)
        Severe("Unable to create \"bvh\" accelerator.");
    Scene *scene = new Scene(accelerator, lights, volumeRegion);
    ⟨Erase primitives, lights, and volume regions from RenderOptions 1072⟩
    return scene;
}
```

If more than one VolumeRegion was specified, an AggregateVolume is created to store them.

⟨*Initialize* volumeRegion *from volume region(s)*⟩ ≡ 1072

```
VolumeRegion *volumeRegion;
if (volumeRegions.size() == 0)
    volumeRegion = NULL;
else if (volumeRegions.size() == 1)
    volumeRegion = volumeRegions[0];
else
    volumeRegion = new AggregateVolume(volumeRegions);
```

After the scene has been created, RenderOptions clears the vectors of primitives, lights, and volume regions. This ensures that if a subsequent scene is defined then the scene description from this frame isn't inadvertently included.

⟨*Erase primitives, lights, and volume regions from* RenderOptions⟩ ≡ 1072

```
primitives.erase(primitives.begin(), primitives.end());
lights.erase(lights.begin(), lights.end());
volumeRegions.erase(volumeRegions.begin(), volumeRegions.end());
```

Renderer creation in the MakeRenderer() method is again similar to how shapes and other named objects are created; the string name is used to dispatch to an object-specific creation function.

⟨*RenderOptions Public Methods*⟩ ≡ 1056

```
Renderer *MakeRenderer() const;
```

B.4 ADDING NEW OBJECT IMPLEMENTATIONS

We will briefly review the overall process that pbrt uses to create instances of implementations of the various abstract interface classes like Shapes, Cameras, SurfaceIntegrators, etc., at run time. We will focus on the details for the Shape class, since the other types are handled similarly.

When the API needs to create a shape, it has the string name of the shape and the ParamSet that represents the corresponding information in the input file. These need to be used together to create a specific instance of the named shape. MakeShape() has a series of if tests to determine which shape creation function to call; the one for Spheres, CreateSphereShape(), is shown here:

⟨*Sphere Method Definitions*⟩ +≡
```
Sphere *CreateSphereShape(const Transform *o2w, const Transform *w2o,
        bool reverseOrientation, const ParamSet &params) {
    float radius = params.FindOneFloat("radius", 1.f);
    float zmin = params.FindOneFloat("zmin", -radius);
    float zmax = params.FindOneFloat("zmax", radius);
    float phimax = params.FindOneFloat("phimax", 360.f);
    return new Sphere(o2w, w2o, reverseOrientation, radius,
                      zmin, zmax, phimax);
}
```

The appropriate named parameter values are extracted from the parameter list, sensible defaults are used for ones not present, and the appropriate values are passed to the Sphere constructor. As another alternative, we could have written the Sphere constructor to just take a ParamSet as a parameter and extracted the parameters there. We followed this approach instead in order to make it easier to create spheres for other uses without having to create a full ParamSet to specify the parameters to it.

Thus, adding a new implementation to pbrt requires adding the new source files to the build process so they are compiled and linked to the executable, modifying the appropriate creation function (MakeShape(), MakeLight(), etc.) in core/api.cpp to look for the new type's name and call its creation function, in addition to implementing the creation function (like CreateSphereShape()) that extracts parameters and calls the object's constructor, returning a new instance of the object.

Camera 302
CreateSphereShape() 1073
MakeShape() 1066
ParamSet 1047
ParamSet::
  FindOneFloat() 1050
Shape 108
Sphere 115
SurfaceIntegrator 740
Transform 76

FURTHER READING

Foley et al. (1990) have discussed hierarchical modeling and retained mode versus immediate mode rendering APIs in depth; their chapter on these issues (Chapter 7) is a good starting point for more information about these topics. The *OpenGL Programming Guide* (Woo et al. 1999) is a good reference for learning the OpenGL interface. The RenderMan® API is described in a number of books (Upstill 1989; Apodaca and Gritz 2000) and in its specification document, which is available online (Pixar Animation Studios 2000). A different approach to rendering APIs is taken by the *mental ray* rendering system, which exposes much more internal information about the system's implementation to users, allowing it to be extensible in more ways than is possible in other systems. See Driemeyer and Herken (2002) for further information about it.

EXERCISES

③ B.1 `pbrt`'s scene file parser is written using the standard `lex` and `yacc` tools. While these are an easy way to develop such a parser, carefully implemented handwritten parsers can be substantially more efficient. Replace `pbrt`'s parser with a handwritten parser that parses the same file format. Measure the change in performance with your parser. Profile `pbrt` when rendering scenes with large scene description files. What fraction of execution time is spent in parsing?

② B.2 `pbrt`'s scene description format makes it easy for other programs to export `pbrt` scenes and for users to edit the scene files to make small adjustments to them. However, for complex scenes (such as the ecosystem scene in Figure 4.1), the large text files that are necessary to describe them can take a long time to parse.

Investigate extensions to `pbrt` to support compact binary file formats for scene files. One possible approach is to support binary encoding for large meshes: since most of the complexity in detailed scenes comes from large polygon and subdivision surface meshes, providing specialized encodings for just those shapes may be almost as effective as a binary encoding for the entire scene description format and has the advantage of not requiring that the scene file parser be rewritten. You could extend `TriangleMesh` and/or `LoopSubdiv` to take an optional string parameter that gives the filename for a binary file that holds some or all of the mesh vertex positions and normals and the array of integers that describes which triangles use which vertices.

A more complex alternative is to develop a complete binary encoding for the text scene description file format. Each directive in the file (`WorldBegin`, `Shape`, etc.) could be encoded with a unique byte-sized value, with string values following them (such as `"sphere"`) encoded as NUL-terminated strings, and then their parameter lists encoded with a similar mix of strings and binary-encoded integer and floating-point values.

③ B.3 Another approach for reducing renderer startup time is to support a binary representation of internal data structures that can be written to disk. For example, for complex scenes, creating the ray acceleration aggregates may take more time than the initial parsing of the scene file. An alternative is to modify the system to have the capability of dumping out a representation of the acceleration structure and all of the primitives inside it after it is first created. The resulting file could then be subsequently read back into memory much more quickly than rebuilding the data structure from scratch. However, because C++ doesn't have native support for saving arbitrary objects to disk and then reading them back during a subsequent execution of the program (a capability known as *serialization* or *pickling* in other languages), adding this feature effectively requires extending many of the objects in `pbrt` to support this capability on their own.

`LoopSubdiv` 151
`TriangleMesh` 135

One additional advantage of this approach is that substantial amounts of computation can be invested in creating high-quality acceleration structures, with the knowledge that this cost doesn't need to be paid each time the scene is loaded into memory. This approach is more complex than the one in Exercise B.2, but should give better performance.

❷ B.4 The material assigned to object instances in `pbrt` is the current material when the instance was defined inside the `pbrtObjectBegin()/pbrtObjectEnd()` block. This can be inconvenient if the user wants to use the same geometry multiple times in a scene, giving it a different material. Fix the API and the implementation of the `TransformedPrimitive` class so that the material assigned to instances is the current material when the instance is instantiated, or, if the user hasn't set a material, the current material when the instance was created.

❸ B.5 Generalize `pbrt`'s API for specifying animation; the current implementation only allows the user to provide two transformation matrices, only at the start and end of a fixed time range. For specifying more complex motion, a more flexible approach may be useful. One improvement is to allow the user to specify an arbitrary number of *keyframe* transformations, each associated with an arbitrary time.

More generally, the system could be extended to support transformations that are explicit functions of time. For example, a rotation could be described with an expression of the form `Rotate (time * 2 + 1) 0 0 1` to describe a time-varying rotation about the z axis. Extend `pbrt` to support a more general matrix animation scheme and render images showing the visual improvement. What is the performance impact of your changes for scenes with animated objects that don't need the generality of your improvements?

❸ B.6 Extend `pbrt`'s API to have some retained mode semantics so that animated sequences of images can be rendered without needing to respecify the entire scene for each frame. Make sure that it is possible to remove some objects from the scene, add others, modify objects' materials and transformations from frame to frame, and so on.

❷ B.7 In the current implementation, a unique `TransformedPrimitive` is created for each `Shape` with an animated transformation. If many shapes have exactly the same animated transformation, this turns out to be a poor choice. Consider the difference between a million-triangle mesh with an animated transformation versus a million independent triangles, all of which happen to have the same animated transformation.

In the first case, all of the triangles in the mesh are stored in a single instance of a `TransformedPrimitive` with an animated transformation. If a ray intersects the conservative bounding box that encompasses all of the object's motion over the frame time, then it is transformed to the mesh's object space according to the interpolated transformation at the ray's time. At this point, the intersection computation is no different than the intersection test with a static primitive; the only overhead due to the animation is from the larger bounding box and rays

`pbrtObjectBegin()` 1069
`pbrtObjectEnd()` 1069
`Shape` 108
`TransformedPrimitive` 190

that hit the bounding box but not the animated primitive and the extra computation for matrix interpolation and transforming each ray once, according to its time.

In the second case, each triangle is stored in its own `TransformedPrimitive`, all of which happen to have the same `AnimatedTransform`. Each instance of `TransformedPrimitive` will have a large bounding box to encompass each triangle's motion, giving the acceleration structure a difficult set of inputs to deal with: many primitives with substantially overlapping bounding boxes. The impact on ray–primitive intersection efficiency will be high: the ray will be redundantly transformed many times by what happens to be the same recomputed interpolated transformation, and many intersection tests will be performed due to the large bounding boxes. Overall performance will be much worse than the first case.

To address this case, modify the code that implements the `pbrt` API calls so that if many independent primitives are provided with the same animated transformation, they're all collected into a single acceleration structure with a single animated transformation. What is the performance improvement for the worst-case outlined above? Is there an impact for more typical scenes with animated primitives?

`AnimatedTransform` 96
`TransformedPrimitive` 190