OpenSG Symposium (2003)
D. Reiners (Editor)

The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing –

Andreas Dietrich, Ingo Wald, Carsten Benthin, Philipp Slusallek

{dietrich, wald, benthin, slusallek}@graphics.cs.uni-sb.de

Abstract

For more than a decade now, interactive graphics has been shaped by triangle rasterization technology and the corresponding OpenGL graphics API. Since recently, however, interactive ray tracing is becoming a reality, and is slowly becoming available on several different hardware platforms. Due to its superior scalability, usability and efficiency, it is likely to play an increasingly important role in future interactive graphics applications.

Though it would be desirable to drive this technology with a well-known API such as todays quasi-standard OpenGL interface, this would be complicated due to OpenGL's tight coupling to rasterization technology, which makes it less suitable for ray tracing. In this paper, we propose a new application programming interface called OpenRT. This new API is designed to be as similar to OpenGL as possible, while emphasizing the strength of interactive ray tracing. While being as simple to learn and use as OpenGL, OpenRT offers all the advantages of ray tracing, like implicit visibility culling, instantiation, the freedom to shoot arbitrary rays, and fully programmable shading.

1. Introduction

Interactive computer graphics has become an important part in such fields as science, engineering, and entertainment to mention only a few. Tremendous progress in development of graphics hardware made interactive rendering performance widely available even on standard PCs.

Although every new generation of graphics hardware provides richer features and improved performance, high-quality image generation is still dominated by offline rendering systems, which in contrast usually are far from being interactive.

1.1. Rasterization

The majority of today's interactive rendering systems builds on rasterization-based techniques. These approaches process polygons independently of each other by projecting them onto the image plane. Once a triangle is *issued*, it is immediately transformed and lighted, clipped, rasterized, and z-buffered. After all these operations have been performed, the rasterizer can move on to the next triangle.

As each triangle is processed purely locally and without relation to any other triangle, this process can be efficiently performed in a pipelined manner, which results in the *rasterization pipeline*.

While this purely local pipeline model allows for highly efficient hardware implementations, the absence of global information has certain drawbacks: First, the approach is inherently linear in the number of triangles, as each triangle has to be processed on its own. Even more importantly, the renderer has to rely on purely local information when it comes to shading. Hence producing high-quality images requires multiple rendering passes and on top of that extensive manual tuning. Although recent graphics hardware features programmable processing units, multi-pass rendering is still required e.g. to produce shadows. Even then physically-accurate global lighting effects such as multiple reflections and refractions let alone diffuse interreflections are beyond the capabilities of such rendering tricks.

1.2. Ray Tracing

Conversely, most architectures incorporating sophisticated illumination perform *ray tracing*. Ray tracing closely models the physical process of light propagation by shooting imaginary rays into the scene to be visualized [4]. As a consequence, there is a number of advantages over rasterization algorithms:

Physical Correctness. The ability to shoot arbitrary rays allows to accurately compute global and advanced lighting and shading effects, such as shadows, reflections, and refractions on arbitrary surfaces even in complex environments (see Figures 1a and 1d).

Plug and Play Shading. Furthermore, ray tracing automatically combines shading effects from multiple objects in the correct order. This allows for building the individual objects and their *shaders* (a function that characterizes the light leaving a point to be shaded as a function of the light arriving there) independently and have the ray tracer automatically take care of correctly rendering the resulting combinations of shading effects (see Figure 1b).

Complex Scenes. Finally, it efficiently supports huge models with billions of polygons as it exhibits logarithmic time complexity with respect to scene size, i.e. the number of primitives in a scene [5]. This efficiency is due to inherent pixel-accurate *occlusion culling* and *demand driven* and *output-sensitive* processing that computes only actually visible results (see Figures 1c and 1d).

However, ray tracing techniques have also been infamous for their extensive rendering times, since millions of rays have to be intersected with the virtual scene's geometry. In order to obtain interactive frame rates highly optimized implementations are necessary in conjunction with significant computational horse power.

That interactive ray tracing is in fact possible was first demonstrated by Muuss et al. [8] and Parker et al. [12] using massive parallelization on large shared memory supercomputers. By employing algorithmic improvements together with low-level optimizations tailored to the capabilities of modern CPUs Wald et al. [19] were able to achieve high ray tracing performance even on a single processor. They also showed how efficient parallelization can be implemented on commodity PC clusters exhibiting linear scalability [21]. More recently, Schmittler et al. [17] and Purcell et al. [15] demonstrated the feasibility of efficient hardware support either by using dedicated ray tracing chips or utilizing the features of programmable GPUs respectively.

2. An Interactive Ray Tracing API

With the recent advances in interactive ray tracing, it seems likely that ray tracing will play a larger role in future interactive graphical applications. Thus, there naturally arises the need for a flexible and powerful yet easy to use programming interface, which provides an abstraction layer between the application and the underlying ray tracing library.

2.1. Previous Work

Ideally, one would simply adopt an already standardized, widespread and well-known graphics interface. Currently, there already exists a number of APIs covering various kinds of abstraction levels. However, none of these is well suited for interactive ray tracing.

Pixar's well-known RenderMan [13] API is oriented towards high-quality rendering and has been successfully used for ray tracing. Unfortunately it is missing support for interactive applications. Choosing a high-level API from scene graph libraries such as OpenGL Performer [16], OpenInventor [22] or OpenSG [11] would be difficult because of their large number and the fact that each of them is used mostly for a specific application domain. Instead, we have chosen to provide a low-level API in the spirit of OpenGL on which we could then layer any high-level scene graph interface. At first sight OpenGL [9, 1] itself seems suitable due to its popularity as well as its flexibility. However, as we will see below, it is too tightly coupled to the rasterization pipeline.

2.2. OpenRT

APIs appropriate for interactive ray tracing are simply not yet available. Because considering only extensions to existing interfaces might have caused to much limitations, we decided to come up with a new ray tracing specific design, which we called *OpenRT*. Though design and development of OpenRT are not yet finalized, it is already used in practice. Currently, it is used to drive the Saarland University's Real-Time Ray Tracing system [19, 21]. This implementation demonstrates the capabilities of the new API: As can be seen from Figure 1, OpenRT allows to drive applications ranging from physical simulation, over arbitrary programmable shading, to highly complex scenes, and even interactive lighting simulation in huge models. However, the OpenRT API is not limited to this specific implementation, but is currently also being evaluated to drive other ray tracing systems, such as for example the SaarCOR architecture [17].

As the name suggests, OpenRT is syntactically oriented towards OpenGL as a simple, easy-to-use, well-known, and widely accepted programming interface. While staying as close as possible, OpenRT is neither a simple extension nor a subset of OpenGL due to the fundamental differences between rasterization and ray tracing as outlined in Sections 1.1 and 1.2.

2.3. Relation to OpenGL

In this section we will now take a closer look at the similarities and differences between OpenRT and OpenGL.

2.3.1. Rendering semantics

Resulting from OpenGL's rasterization-based pipeline architecture, two basic operations are performed: Drawing something and changing the state that determines how drawing is performed. OpenGL includes two drawing modes. During *immediate mode* each triangle that has been fully specified is sent to the pipeline and gets rasterized right away with

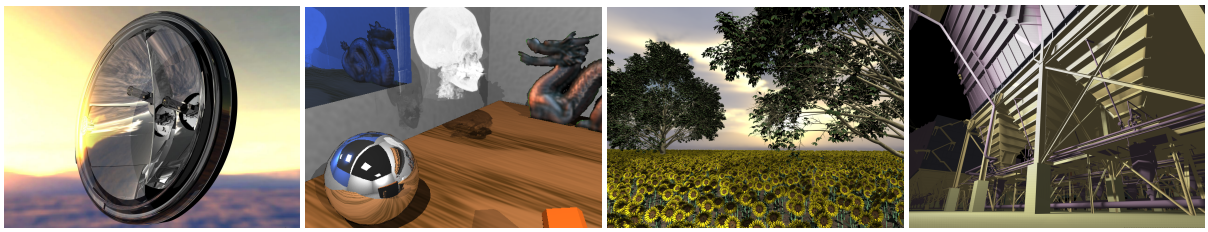

Figure 1: *Examples of rendering complex and dynamic scenes in real time using our OpenRT API. From left to right: (a) A car headlight with physically-correct glass simulation, (b) An office environment with procedural shading, a lightfield, and a volume object, (c) An outdoor scene consisting of roughly one billion triangles featuring shadows and transparency, and finally (d) interactive global illumination in a scene with 37.5 million triangles. All examples make use of individual rendering front-end applications, each built on top of the OpenRT API driving our interactive software ray tracing engine.*

respect to the currently active state (e.g. the current texturing state like filter and wrapping modes). Later changes to this state will only affect successive triangles and rendering a new frame requires sending all primitive data anew. Although these semantics could also be applied by combining a ray *casting* process with z-buffering similar to the REYES Image Rendering Architecture [3], it would limit the rendering system to local illumination.

OpenRT's ray tracing back-end on the other hand involves global lighting and therefore requires a more object-oriented approach: The user specifies geometric *objects*, which encapsulate primitives and are organized in an efficient low-level scene graph. Light transport simulation can be explicitly defined using programmable shader plugins. Geometric objects are then bound to shader objects that contain specific attributes e.g. material colors. This comes close to the *retained mode* of OpenGL where primitives are stored in *display lists* in a compiled form. The binding of individual shader instances may be regarded as *local* state changes, but two issues have to be kept in mind: First of all OpenGL display lists also depend on global state changes and can be rendered differently even if remaining unaltered themselves. Secondly, because geometry and shader instances are linked by references, manipulating shader parameters will affect *all* primitives affiliated with them. Apart from environmental settings (e.g. atmospheric effects like fog), shader attributes provide the only way to affect geometric appearance.

Even though these semantic differences require careful attention during porting of existing OpenGL applications, especially scene graph packages like OpenInventor should not cause problems as they typically too incorporate an object-oriented point of view.

2.3.2. Multi-pass Rendering vs. Programmable Shading

It has already been mentioned in Section 1.1 that rasterization-based systems like OpenGL may approximate more complex lighting situations by applying *multi-pass* rendering techniques. Blending enables applications to combine images resulting from multiple drawing passes. However, these approximations are always much coarser than ray tracing solutions, as e.g. reflection mapping methods often do not account for inter-object reflections.

Of course, ray tracing is also capable of multi-pass procedures, yet they are not required due to the ability to simulate physically-correct lighting during one single rendering step. It has already been mentioned in the previous sections that specialized shader routines are responsible for this task. OpenRT provides a fully programmable shading model, making it possible to directly implement shaders for most optical effects. Writing such a shader is straightforward and much easier than hand tuning complicated OpenGL code. For example, adding reflection is one of the basic tasks of ray tracing. It only requires the shader to shoot one additional ray and can be specified by just a few lines of code as we will see in more detail in Section 3.2.5.

Independently written OpenRT shaders may also be simultaneously assigned to individual geometric objects in a simple plug and play manner. The desired effects are automatically combined and simulated in the correct order during ray tracing (see Figure 1b).

In our current software implementation OpenRT handles user programmable shading via loadable shader libraries. Shading routines may be coded and compiled separately and are dynamically linked at run-time. C/C++ serves as programming language and thus offers high-level language support. Our implementation features diverse shader types ranging from simple Phong models as used by OpenGL, procedural effects (e.g. marble) up to global illumination algorithms including path tracing or photon mapping. Future hardware implementations might have limited programmability but this is not a restriction of the API but rather of the underlying hardware and it would only affect shader code but not the application.

2.3.3. Objects and Instantiation

As pointed out in Section 2.3.1 OpenGL and OpenRT rendering semantics are quite different. As a consequence

OpenRT does not offer immediate mode rendering. Nevertheless its object definition scheme behaves mostly like OpenGL's display list handling. Primitives are grouped into objects, providing a collection of geometry. Once an object has been fully defined an acceleration structure is built, inevitable for efficient ray surface intersection calculations. Note, however, that there are neither state changes nor side effects; objects serve as simple geometry containers and primitives can never be specified outside an object.

Similar to OpenGL display lists, objects have to be instantiated in order to be effective. Nevertheless there remains the difference that visibility computations will only take place once all objects have been specified. On the other hand, instancing works most efficient for a ray tracer. Not only that a single object can be reused multiple times, but because of inherent occlusion culling no overdraw operations take place, there may be thousands of instances without suffering a major hit in rendering performance.

2.3.4. Fragment and 2D Operations

OpenRT serves as a pure 3D graphics library. However, several low-level OpenGL applications (e.g. games) also use 2D imaging and additional *fragment* operations like stencil tests, alpha tests, blending etc. to perform special effects like masking certain image regions or alpha-blending (e.g. explosions). Fragments, i.e. the output of OpenGL's actual rasterization stage can be regarded as partial color results in analogy to radiance values that travel along single rays and therefore contribute to a pixel's final color.

OpenRT does not offer a direct equivalent to these operations but according to the just mentioned analogy they could be performed by programmable shaders, although this might sometimes be un-intuitive. For example, blending could be realized using transparently textured polygons.

One way of solving that problem would be to mix OpenGL and OpenRT calls: In a first pass, OpenGL functions could be used to control the way that primary rays are issued, e.g. by using stencil buffers to define which pixels should be traced and by using clipping planes to clip the primary rays. Then, the ray tracer could use this information to compute the required pixels, setting both the color value and the z-buffer value for each pixel. Finally, an OpenGL pass could perform 2D imaging operations, like copying pixels, or activating alpha-blending to achieve the desired imaging effects.

3. The OpenRT Application Programming Interface

A complete interactive ray tracing system typically consists of three different processing parts: The application itself, shader programs, and the ray tracing core. User interaction, scene specification and display are handled by the front-end application, shaders implement surface dependent light reflection/scattering calculations, while the ray tracing core is responsible for actually transporting light within the scene. Consequently, the OpenRT API comprises three different sub-interfaces.

3.1. Interface Organization

First, the core *OpenRT application interface* is designed to allow an application to specify geometry, objects, transformations, texture objects, and so on, in a way similar to OpenGL. Secondly, the *OpenSRT programmable shader interface* provides ways to specify and load shading programs, and enables the application to communicate with them via shader parameters. Finally, the *OpenRTS shading language* controls core access by shaders.

As the actual shading language is but slightly coupled with the rest of the API, using another shading language (such as a variant of CG [10], RenderMan [13], or similar) should be possible without major changes. In the remainder of this paper we will concentrate on the core API. Due to brevity of space, we can just outline the major components of OpenRT.

3.2. A Brief Tour of OpenRT

For sake of rendering performance most ray tracing (and also rasterization) engines exclusively operate on triangles. Therefore, we have chosen to restrict our API to also support only polygonal primitives. Thus, in most cases for specifying geometry we can directly use the same syntax as OpenGL, except that the usual "gl" prefix has been changed to "rt". Even though OpenRT sometimes uses different semantics, even inexperienced OpenGL programmers should easily understand the examples presented in the following sections.

3.2.1. Geometry and Transformations

OpenGL supports a lot of functionality to specify geometric primitives and most of these functions can directly be adopted by OpenRT. So an application may issue its geometry in exactly the same way by implementing customary `rtBegin()/rtEnd()` statements with all their usual primitive types like `RT_TRIANGLE`, `RT_TRIANGLE_FAN`, `RT_POLYGON` etc. After a primitive mode has been chosen vertices are created by the familiar calls such as `rtVertex3f()`, `rtNormal3f()`, `rtColor3f()`, or `rtTexCoord3f()` where all OpenGL vertex attributes namely position, surface normal, color, and textures coordinates are available. Apart from these standard parameters, OpenRT also allows to keep arbitrary additional data with each vertex as per-vertex shader data.

Unlike OpenGL, OpenRT currently maintains only two transformation stacks to coordinate `RT_MODELVIEW` and `RT_TEXTURE` matrices. As expected, matrix modes are selected by invoking `rtMatrixMode()`. However, no projection mode like `GL_PROJECTION` exists. Rather than

backward projecting every single object, a ray tracer explicitly produces *primary rays* which are actually *projectors* i.e. projection rays that emanate from a *center of projection* and cross a *projection plane*. These rays are created by so-called *camera shaders* which are also capable of simulating complicated camera models including depth-of-field and other lens effects. Even without a projection matrix mode, camera parameters may be easily specified by `rtPerspective()` and `rtLookAt()`. It should also be pointed out that the ModelView matrix does not include viewing transformations because they are inherently handled by camera shaders.

Regardless of particular matrix modes OpenRT features the full set of OpenGL matrix operations such as `rtScalef()`, `rtRotatef()`, `rtTranslatef()` including `rtMultMatrixf()`, `rtLoadIdentity()`, `rtPushMatrix()`, `rtPopMatrix()` etc.

3.2.2. Geometry Objects

Section 2.3.1 outlined that OpenRT does not offer immediate mode rendering. As a ray tracer simulates the flow of light it needs to efficiently access geometric primitives multiple times during the generation of a single image frame. Therefore, OpenRT defines persistent geometry objects encapsulating primitives, similar to an OpenGL display list that is defined once, and can then be reused efficiently.

In analogy to OpenGL's `glGenLists()`, objects are allocated by `rtGenObjects()`. A new object is then opened using `rtNewObject()` followed by primitive definitions as presented in the last section. Finally, `rtEndObject()` has to be invoked, which additionally triggers construction of the object's acceleration structure.

3.2.3. Instantiation

Just like with display lists (if using `GL_COMPILE`), simply specifying an object does not automatically include it into the actual rendering process. After definition, objects first have to be *instantiated* similar to OpenGL's `glCallList()`. To avoid confusion, the appropriate function has been named `rtInstantiateObject()` because no immediate rendering occurs.

In fact, an instance is just a reference to a geometry object, combined with an associated transformation. Specification of such transformations is done using the same matrix functionality as for specifying primitives. Note, that as a side effect of this approach multiple instantiations are natively supported. Reusing a display list several times is also possible with OpenGL, but suffers from the disadvantage that the object has to be rerasterized with every activation.A ray tracer only has to handle geometry that is actually visible.

It is also possible to manage dynamic affine transformations by simply replacing the matrix of an instance with the help of `rtSetInstanceXfm()`. For more complicated changes an object has to be deleted and rebuilt.

3.2.4. Shading And Lighting

As discussed above, OpenRT does not support a fixed material and lighting model but exclusively uses shaders to determine the appearance of geometric surfaces. Therefore `glMaterial()` functions are not available. Instead, an application controls shaders via the OpenSRT programmable shader interface, similar to the Stanford shader API [14, 7].

Shader Specification. All shaders are dynamically loaded from shared library files that may also comprise several independent classes; `srtShaderFile()` selects the file to load as well as the requested shader class. To create an instance `srtCreateShader()` must be called afterwards. Each such shader object has to be assigned a unique ID by which it can be referenced later. Once created, a shader is also bound, so that all subsequently defined primitives will use this instance as a surface shader. Other shaders my be bound later again by invoking `srtBindShader()`.

Shader Parameters. When a shader class is being loaded, a set of parameters that is needed by the shader code to perform its lighting calculations is declared via `rtsDeclareParameter()`. Such data may be located in different scopes: Parameters declared as `PER_SHADER` are encapsulated inside shader instances. This typically involves settings which do not change over a shader's surface like numerical precision values (e.g. maximum recursion depth). Features declared as `PER_TRIANGLE` are stored with individual triangles (e.g. color information). Finally, if declared as `PER_VERTEX`, a parameter belongs to a single vertex (e.g. special texture coordinates).

Another important declaration argument is a unique identifier string, under which an application can refer to the parameter's value. The user can then register handles to so exported shader data, and is able to alter it with a call to a generic `srtParameter{1234}{ifdv}()` method. Of course, one has to know the names and types of the desired parameters.

Algorithm 1 presents a simple example of how a shader is being used: After first loading the shader class from a shared object file, the application program registers handles to the exported parameters. Then it instantiates a shader object and finally sets diffuse color values and a texture ID, which is needed by the shader program for texture lookup. (see also Section 3.2.6).

Light, Environment and Camera Shaders. OpenRT also offers light, environment, and camera shaders, i.e. custom programs for calculating light source characteristics, environment illumination and per-pixel postprocessing, respectively. For these shader types the initialization process works similarly. Instead of binding a shader to geometry, it can also be declared to act as a light, environment or camera shader by calling `srtUseLight()`,

Simple.cpp:

```
struct SimpleShader : public RTShader {
  RTfloat diffuse[3]; // diffuse color
  RTint   texture;    // texture ID

  RTvoid Register() {
    // export shader parameters
    rtsDeclareParameter("texture",
      PER_SHADER,memberoffset(texture),
      sizeof(texture));
    rtsDeclareParameter("diffuse",
      PER_SHADER,memberoffset(diffuse),
      sizeof(diffuse));
  }
  RTvoid Shade(RTState *state) {
    RTfloat color[3], reflect[3], normal[3];
    RTState reflection;

    rtsFindShadingNormal(state,normal);

    // calculate diffuse term
    DiffuseTerm(color,diffuse,normal);

    // modulate with texture
    rtsApplyTexture(state,texture,color);

    // trace reflection ray
    rtsReflectionRay(&reflection,state,normal);
    rtsTrace(&reflection);
    rtsGetColor(&reflection,reflect);

    // calculate reflective term
    ReflectiveTerm(color,reflect);

    rtsReturnColor(color);
  }
};
```

Application.cpp:

```
int main(int argc, char *argv[]) {
  const RTuint SIMPLE_DIFFUSE = 0;
  const RTuint SIMPLE_TEXTURE = 1;
  ...
  srtShaderFile(0,"Simple",NULL);
  srtParameterHandle("diffuse",SIMPLE_DIFFUSE);
  srtParameterHandle("texture",SIMPLE_TEXTURE);
  ...
  srtCreateShader(1);
  srtParameter3f(SIMPLE_DIFFUSE,1,0,0);
  RTint texID = LoadTexture("tex.ppm");
  srtParameter1i(SIMPLE_TEXTURE,texID);
  ...
}
```

Algorithm 1: *A simple shader. The `Register()` function declares two parameters: A diffuse color vector and a texture ID. The `Shade()` callback then uses these parameters to compute a local color for the incident ray, and finally adds reflection to it. Inside the `main()` part some simple code loads the given shader, declares handles to its parameters, and sets the parameters to appropriate values.*

`srtUseEnvironment()` or `srtUseCamera()` with the respective instance IDs. Currently supported are a single environment and one camera shader as well as an arbitrary number of light shaders.

3.2.5. Shading Language

All shaders are currently written in C++ working on top of our OpenRTS shading language API. Each shader is derived from a base class `RTShader`, and has to implement a number of callback functions: Typically a `Register` method to register the shader and to declare its parameters, and also a `Shade` callback, which incorporates the actual light transport calculation routines. Algorithm 1 shows a simple surface shader that uses our shading language to acquire data from the renderer (e.g. the surface normal), to shoot reflection rays, and to return the final color value that is visible along an incident ray. Note, however, that we do not consider the actual shading language to be an integral part of the OpenRT API. Other shading languages like CG [10], RenderMan [13] etc. should be possible to support as long as the same concepts (like shader parameters) are being used.

3.2.6. Texturing

In contrast to OpenGL – which knows both texture objects and immediate texturing – the use of texture objects is mandatory in OpenRT. Regarding their definition, however, we have adopted exactly the same syntax as in OpenGL: Texture objects first have to be allocated by calling `rtGenTextures()`, can later on be (re)bound using `rtBindTexture()` and get their pixel data specified via `rtTexImage{12}D()`, with all the usual texture formats being supported (`RT_RGB`, `RT_INTENSITY` etc.). Currently, only 1D and 2D textures are included, but 3D texture implementation is straightforward.

Texture parameters can also be set exactly like in OpenGL using `rtTexParameteri()`. All OpenGL texture parameters are available, e.g. wrapping modes (`RT_CLAMP`, `RT_REPEAT` etc.) and also minification/magnification filter modes including linear filtering and MIP maps.

Once a texture object has been created, a reference is passed to a shader by sending the ID of the texture object to a shader parameter of type `RTint`. The shader code can then access the texture using the given ID. Of course, an arbitrary number of textures can be assigned to each shader using individual parameters. Similarly, each of these textures can have different types, resolutions, formats, filter modes etc. The usual way to access a texture is to execute `rtsApplyTexture()` along with the texture's ID. This call automatically considers all the specified parameters of the texture, and accordingly modifies the color "fragment" given to it (e.g. replacement, modulation, or addition). Note, that a shader can bypass this mechanism, and may directly read individual texels from the texture using its own mode settings.

3.3. A Detailed Example Application

After having outlined the most basic concepts of OpenRT, we can discuss their actual application on a simple example (see Algorithms 2 and 3):

Algorithm 2 first specifies a new geometry object containing a vertex-colored RGB cube along with a shader of class `VertexColor` applied to it. First, this code loads the shader class from a shared object file (by using `srtShaderFile()`) and creates an instance of this shader (`srtCreateShader()`). Then, a new geometry object is allocated (`rtGenObjects()`) and opened for definition (`rtNewObject()`). After setting the correct transformations (`rtMatrixMode()` etc.), some vertex-colored polygons are placed inside (`rtVertex3f()`, `rtColor3f()` etc.). Upon finalizing the object definition, the routine returns the object's ID, by which it can be referenced later for instantiation. In fact, this example looks almost exactly the same as defining a display list in OpenGL.

The object just defined is further being used in the second example. As can be seen in Algorithm 3, the `main` procedure makes use of the *OpenRT Utility Toolkit*, *RTUT*. RTUT (written by Markus Wagner [18]) provides a window system independent library that aids the process of creating windows, handling events, communicating with input devices etc. It provides almost exactly the same functionality as the OpenGL Utility Toolkit [6] and may be virtually operated the same way.

Our example invokes RTUT calls for opening a window, and for registering the display and idle callbacks. This code is identical to corresponding GLUT code. Of course, usage of RTUT is not obligatory. It would just as well be possible to open and use display windows independently of RTUT helper functions.

After the main window has been successfully created, the `createColorCubeObject()` procedure from Algorithm 2 is activated to define an RGB color cube object. After that, we set up the camera, and enter the RTUT event loop. As with GLUT, this main event loop independently calls back the previously registered `Display()` and `Idle()` functions. The `Display()` procedure just executes `rtutSwapBuffers()`, which displays the current image and triggers rendering of the next frame. The actual specification of the scene graph takes place during the `Idle()` callback: After first clearing all instances, it creates eight shifted and rotated instances of the RGB color cube object. Here, transformation calls (`rtTranslatef()`, `rtScalef()` etc.) affect these instances. By changing the ModelView matrix before invocation of `rtInstantiateObject()`, each of the eight cubes is positioned at a different place, even though they are all instances of exactly the same object.

Altogether, this simple example demonstrates the most important aspects of OpenRT: Definition of geometry objects and their instantiation, loading shader classes, opening windows and rendering a frame, and finally the use of transformations to modify both geometry and instances. This program also shows the syntactic similarity to OpenGL. Even though Algorithms 2 and 3 form a "classical" OpenRT example program, each experienced OpenGL programmer should be able to read, understand, and if necessary, to extend it.

```
#include <openrt/rt.h>
#define VERTEX_COLOR_SHADER_ID     0
#define VERTEX_COLOR_SHADERFILE_ID 1

RTint createColorCubeObject()
{
  srtShaderFile(VERTEX_COLOR_SHADERFILE_ID,
                "VertexColor",
                "libVertexColor.so");
  srtCreateShader(VERTEX_COLOR_SHADER_ID);

  RTint objId = rtGenObjects(1);
  rtNewObject(objId, RT_COMPILE);

    rtMatrixMode(RT_MODELVIEW);
    rtPushMatrix();
    rtTranslatef(-1, -1, -1);
    rtScalef(2, 2, 2);

    // first cube side
    rtBegin(RT_POLYGON);
      rtColor3f(0, 0, 0);
      rtVertex3f(0, 0, 0);
      rtColor3f(0, 1, 0);
      rtVertex3f(0, 1, 0);
      rtColor3f(1, 1, 0);
      rtVertex3f(1, 1, 0);
      rtColor3f(1, 0, 0);
      rtVertex3f(1, 0, 0);
    rtEnd();

    // other cube sides
    ...
    rtPopMatrix();

  rtEndObject();
  return objId;
}
```

Algorithm 2: *Allocating and defining an object containing a vertex-colored cube. This code fragment loads and activates a `VertexColor` shader, allocates a new geometry object, and puts a vertex-colored cube inside. Finally, the routine returns the ID of the generated object by which it can be instantiated later.*

4. Summary and Conclusion

In this paper, we have motivated and proposed OpenRT, a new interactive graphics application programming interface

based on ray tracing. Though being semantically different, the new API is designed to be as similar to OpenGL as possible (see Section 3.3 as well as Algorithms 2 and 3).

```
#include <rtut/rtut.h>
#include <openrt/rt.h>

RTint objId;

void Display()
{ rtutSwapBuffers(); }

void Idle() {
  static int rot = 0; rot++;
  rtDeleteAllInstances();
  for (int i=0; i<8; i++) {
    int dx = (i&1)?-1:1;
    int dy = (i&2)?-1:1;
    int dz = (i&4)?-1:1;

    // position individual objects
    rtTranslatef(dx,dy,dz);
    rtRotatef(4*rot*dx,dz,dy,dx);
    rtScalef(.5,.5,.5);
    rtInstantiateObject(objId);
  }
  rtutPostRedisplay();
}

int main(int argc, char *argv[]) {
  rtutInit(&argc, argv);
  rtutInitWindowSize(640, 480);
  rtutCreateWindow("Eight Cubes");
  rtutIdleFunc(Idle);
  rtutDisplayFunc(Display);

  objId = createColorCubeObject();
  rtPerspective(65, 1, 1, 100000);
  rtLookAt(2,4,3, 0,0,0, 0,0,1);

  rtutMainLoop();
  return 0;
}
```

Algorithm 3: *An example application displaying eight rotating instances of a vertex-colored cube. The* `main()` *procedure initializes a display window, defines a vertex-colored cube object, and activates a perspective camera before starting the main event loop. Inside the* `Idle()` *part, eight transformed instances of the cube are specified.*

Although some parts of the API are still under active development, it is mostly stable, and is already being used for several practical applications. To demonstrate the capabilities of OpenRT, we have developed rendering applications that exclusively use the OpenRT API to drive the interactive ray tracing back-end. As our interface clearly exposes all the advantages of ray tracing, these applications are capable of rendering scenes with virtually any optical effect, as can be seen in Figure 1b, where even complex features such as volume and lightfield rendering could be seamlessly integrated into a scene using programmable shaders. Furthermore, our system is capable of rendering massively complex models of more than 12 million individual triangles. Making heavy use of instantiation, even models consisting of up to one billion triangles can be rendered (see Figure 2).

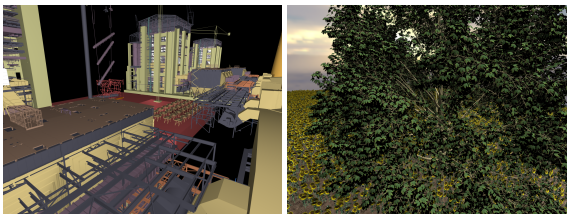

Figure 2: *Complex Models: (a) Three power plants containing 12.5 million individual triangles each, rendering interactively utilizing our API. (b) An outdoor scene consisting of roughly 28,000 instances of 10 different kinds of sunflowers with 36,000 triangles each, together with several multi-million-triangle trees, summing up to roughly one billion triangles. The latter scene features complex shaders using textures, shadows and transparencies.*

Being designed to be similar to OpenGL also allows to easily port existing OpenGL applications to OpenRT. Instead of using only applications that have been exclusively written for OpenRT, we have also taken an existing VRML97 library [2] that was originally based on OpenGL, and have successfully ported it to OpenRT. As VRML does not originally support programmable shading, we had to extend this VRML library to support shaders, after which we can now render VRML models with more sophisticated lighting.

Figure 3: *Left: A shader performing interactive global illumination computations using our own renderer front-end. Right: The same shader being used from within a ported VRML97 renderer, displaying a VRML97 animation with full global illumination.*

Another good example for the benefits of using a well defined API is our interactive lighting simulation application [20]. The entire lighting simulation code has been implemented as programmable shaders. The first obvious advantage is that this code can totally abstract from issues such as scene file formats, light source descriptions or even handling

dynamic scenes, which are all handled transparently through the shader API. Furthermore, as the global illumination code is being used like any other shader, it can be accessed from within different applications. For example, the left image in Figure 3 shows the global illumination shader running in our own front-end application, whereas the right image shows the same shader running in the ported VRML browser, rendering a VRML97 animation with global illumination.

5. Future Work

As there is currently only a single implementation of the OpenRT API available (the Saarland University's Real-Time Ray Tracing engine), an obvious next step is to use this interface also to drive other ray tracing architectures. We are already working on applying OpenRT to drive the SaarCOR [17] as well as other architectures (such as GPU-based systems [15]).

In terms of the API itself, we are currently investigating how to seamlessly integrate handling of huge models using demand-loading and reordering in the spirit of [21], which is not currently being supported. Furthermore, it might make sense to support other shading languages, like e.g. CG [10] or RenderMan [13]. Especially the use of shading language compilers seems promising.

Based on our experiences with porting the VRML97 library we are now investigating how other scene graph libraries such as OpenInventor [22] or OpenSG [11] could be plugged into OpenRT. Even more important, we are evaluating methods for merging OpenGL and OpenRT by modifying OpenRT in a way to use it as an OpenGL extension for interactive ray tracing.

References

1. Kurt Akeley. Reality Engine Graphics. *Computer Graphics*, pages 109–116, July 1993. (Proceedings of ACM SIGGRAPH 1993). 2
2. Philippe Bekaert. Extensible scene graph manager. http://www.cs.kuleuven.ac.be/∼graphics/XRML/, August 2001. 8
3. Robert L. Cook, Loren Carpenter, and Edwin Catmull. The REYES Image Rendering Architecture. *Computer Graphics*, pages 95–102, July 1987. (Proceedings of ACM SIGGRAPH 1987). 3
4. Andrew Glassner. *An Introduction to Raytracing*. Academic Press, 1989. 1
5. Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, 2001. 2
6. Mark J. Kilgard. The OpenGL Utility Toolkit GLUT Programming Interface API Version 3. http://www.opengl.org/developers/documentation/glut.html. 7
7. William Mark. Shading System Immediate-Mode API, v2.1. In *SIGGRAPH 2001 Course 24 Notes – Real-Time Shading*, August 2001. 5
8. Michael J. Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium '95*, June 1995. 2
9. Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993. 2
10. NVIDIA. *NVIDIA OpenGL Extension Specifications*. March 2001. 4, 6, 9
11. OpenSG-Forum. http://www.opensg.org, 2001. 2, 9
12. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics (I3D)*, pages 119–126, April 1999. 2
13. Pixar. *The RenderMan Interface*. San Rafael, September 1989. 2, 4, 6, 9
14. Kekoa Proudfoot, Willim Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Computer Graphics (Proceedings of SIGGRAPH 2001)*, pages 159–170, August 2001. 5
15. Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. (Proceedings of ACM SIGGRAPH 2002). 2, 9
16. John Rohlf and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Computer Graphics*, 28(Annual Conference Series):381–394, July 1994. 2
17. Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of Eurographics Workshop on Graphics Hardware*, pages 27–36, September 2002. 2, 9
18. Markus Wagner. Development of a Ray-Tracing-Based VRML Browser and Editor. Master's thesis, Computer Graphics Group, Saarland University, 2002. 7
19. Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, September 2001. (Proceedings of Eurographics 2001). 2
20. Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques 2002*, pages 15–24, June 2002. (Proceedings of the 13th Eurographics Workshop on Rendering). 8
21. Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. *Rendering Techniques 2001*, pages 274–285, June 2001. (Proceedings of the 12th Eurographics Workshop on Rendering). 2, 9
22. Josie Wernecke. *The Inventor Mentor*. Addison-Wesley, Reading, Massachusetts, U.S.A., 1994. 2, 9