RADEON

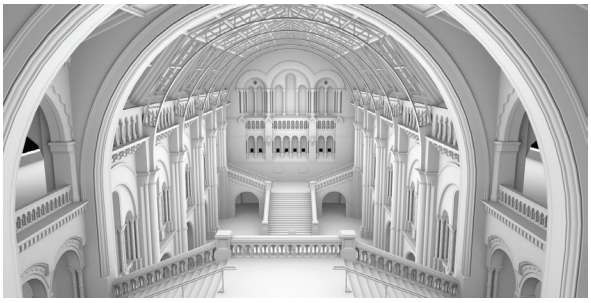
INTRODUCING THE

Radeon Rays SDK

High-Efficiency, High-Performance Heterogeneous Ray Tracing Intersection Library for GPU and APU on Any Platform with OpenCL.

INTRODUCTION

Radeon Rays is a GPU intersection acceleration library with basic support for heterogeneous systems. AMD developed Radeon Rays to help developers get the most out of AMD GPUs and APUs, as well as save them from maintaining hardware-dependent code. Radeon Rays exposes a well-defined C++ API for scene construction and performing asynchronous ray intersection queries. The current implementation is based on OpenCL, which means Radeon Rays supports execution on all platforms conforming to the OpenCL 1.2 standard. It is not limited to AMD hardware or a specific operating system. Radeon Rays is released under the GPUOpen license, which means you can modify the library to your specific needs if necessary. However, using Radeon Rays through its API guarantees compatibility and best performance across the entire line of current and future AMD products.



Ambient Occlusion

GI and Shadows

AMD

FirePro W9100			
Sponza (273K triangles)	Primary (ms / Mrays/s)	Shadow (2 x primary, ms / Mrays/s)	Secondary (ms / Mrays/s)
VP1 (easy)	6 / 384	20 / 230	27 / 85
VP2 (medium)	7.8 / 295	26 / 177	32.8 / 70
VP3 (hard)	10 / 230	29.8 / 154	39.7 / 58

Rungholt (6.7M triangles)			
VP1 (easy)	5 / 460	6 / 768	17.9 / 95
VP2 (medium)	11.6 / 199	22.8 / 202	33.4 / 70
VP3 (hard)	6.6 / 351	20.4 / 225	73 / 31.5

MacPro (D700 x2)			
Sponza (273K triangles)	Primary	Shadow (double number)	Secondary
VP1 (easy)	8 / 288	17 / 271	26 / 88
VP2 (medium)	9 / 256	24 / 200	30 / 77
VP3 (hard)	13 / 177	22 / 208	34 / 67

Rungholt (6.7M triangles)			
VP1 (easy)	6 / 384	5.5 / 837	21 / 83
VP2 (medium)	7 / 329	18 / 242	56 / 40
VP3 (hard)	6 / 384	22.5 / 204	70 / 31

SYSTEM REQUIREMENTS:

- Apple Mac Pro
- HP Z820 - Intel Xeon E5-2660 @ 2.2 GHz - 32Gb RAM - FirePro W9100 16Gb - Windows 7 x64 15.20 driver

GETTING STARTED WITH THE API

The concept of API and its workflow is relatively simple. To use the API, include the following header:

```
#include <intersection/radeonrays.h>
```

All classes and functions in Radeon Rays are defined in the corresponding namespace, so to access them the user explicitly specifies the namespace or inserts the following `using` statement:

```
using namespace RadeonRays;
```

Device enumeration

Next the user configures the intersection devices by means of the Radeon Rays device enumeration API. Radeon Rays exposes all devices capable of performing intersection queries with the following static methods of `IntersectionApi` class:

```
// Get the number of devices available in the system
static int GetIntersectionDeviceCount();
// Get the information for the specified device
static IntersectionDeviceInfo const& GetIntersectionDeviceInfo(int devidx);
```

Devices are identified by their indices, so the user needs to check device capabilities and save each device's index somewhere if the device fits user needs. For example, you can use the following construct to choose the GPU OpenCL device using enumeration API:

```
int gpuidx = -1;
for (int idx=0; idx < IntersectionApi::GetIntersectionDeviceCount(); ++idx)
{
      IntersectionDeviceInfo const&
      devinfo(IntersectionApi::GetIntersectionDeviceInfo(idx));

      if (devinfo.type == IntersectionDeviceInfo::kGpu &&
      devinfo.apis & IntersectionDeviceInfo::kOpenCl)
      {
              gpuidx = idx;
      }
}
```

API Initialization

Next the user creates an API instance based on the device chosen in previous step, as shown in the following code:

```
int apitype = IntersectionDeviceInfo::kOpenCl;
api_ = IntersectionApi::Create(0, &gpuidx, &apitype, 1);
```

Note that the Create() function accepts an array of devices. However, only a single device is supported in the current library version. The first parameter is also reserved for future use and should be 0.

There is a chance an error will occur during library initialization (OpenCL runtime issue, access rights problem, etc.). To communicate the reason back to the user, the library uses a derivative of the Exception interface so the user can catch it and get the text description of the error.

Geometry creation

Now that the user has an instance of the API, he or she can move to the geometry creation stage. The current version of the API supports triangle, quad, and mixed meshes along with the instancing. The following snippet shows how to create a simple mesh consisting of a single triangle:

```
// Mesh vertices
float vertices[] = {
      0.f,0.f,0.f,
      0.f,1.f,0.f,
      1.f,0.f,0.f
};

int indices[] = {0, 1, 2};

// Number of vertices for the face
int numfaceverts[] = { 3 };
Shape* shape = api_->CreateMesh(vertices, 3, 3*sizeof(float), indices,
0, numfaceverts, 1));
```

Here 0 is used for index stride, meaning indices are densely packed. The method is blocking. However, it is safe to call from multiple threads as long as these calls are not interleaved with ray casting method calls.

The next step is to construct the scene from multiple meshes. To add/remove meshes from the current scene (which is implicitly defined by API instance), you can use the following methods:

```
api_->AttachShape(shape);
api_->DetachShape(shape);
```

These methods are fast since they are not going to launch any time-consuming operations. Instead actual data transfers and acceleration structure constructions are deferred till the call to Commit() method. This method should be called any time something any time something has changed in the scene:

```
api_->Commit()
```

The method is blocking and can't be called simultaneously with other API methods.

Instancing

The geometry can be instanced in the API, which means the same base geometry is used for different entities with different world transforms. The following code should be used to instantiate the mesh:

```
Shape* instance = api_->CreateInstance(shape);
```

Instance can be attached to the scene the same as any other regular shape. Instancing allows you to create overwhelmingly complex scenes with a moderate memory footprint by means of geometry reuse.

Simple intersection queries

As soon as the geometry is committed, the user can perform intersection queries. As the library is specifically designed for heterogeneous architectures, it accepts batches of rays rather than individual rays as an input to intersection query methods. As a general rule, the larger the batch the better because massively parallel devices can maintain better occupancy and perform better latency hiding.

There are basically two types of intersection queries:

- Closest hit query (intersection)
- Any hit query (occlusion)

The methods for querying closest hit are called IntersectBatch() and methods for occlusion are called IntersectBatch() (occlusion predicate).

The simplest version of an intersection query looks like this:

```
// Rays
ray rays[3];

// Prepare the ray
rays[0].o = float4(0.f,0.f,-10.f, 1000.f);
rays[0].d = float3(0.f,0.f,1.f);
rays[1].o = float4(0.f,0.5f,-10.f, 1000.f);
rays[1].d = float3(0.f,0.f,1.f);
rays[2].o = float4(0.5f,0.f,-10.f, 1000.f);
rays[2].d = float3(0.f,0.f,1.f);

// Intersection and hit data
Intersection isect[3];

// Intersect
ASSERT_NO_THROW(api_->IntersectBatch(rays, 3, isect));
```

This method accepts an array of ray structures. The layout of a ray structure is:

o.xyz	Ray origin
d.xyz	Ray direction
o.w	Ray maximum distance
d.w	Time stamp for motion blur

The resulting information for a closest hit query is returned as an array of Intersection structures with the following layout:

uvwt.xyz	Parametric coordinates of a hit (xy for triangles and quads)
uvwt.w	Hit distance along the ray
shapeid	ID of a shape
primid	ID of a primitive within a shape

Shape ID corresponds to a value which is either automatically assigned to a shape at creation time by the API or manually set by the user using Shape::SetId() method. Primitive ID is a zero-based index of a primitive within a shape (in the order they were passed to CreateMesh method). If no intersection is detected, they are both set to kNullId.

An occlusion query has the same format but returns an array of integers, where each entry is either -1 (no intersection) or 1 (intersection).

Asynchronous queries

The methods discussed above are blocking, but there is an option to launch a ray query asynchronously using the following version of the method:

```
// Intersect
Event* event = nullptr;
api_->IntersectBatch(rays, numrays, intersections, nullptr, &event);
```

This method launches an asynchronous query returning the pointer to an Event object. This event can be used to track the status of execution or to build dependency chains. To track the status, the user can use the following methods:

```
event->complete();
event->Wait();
```

The first call returns true if the method has completed and the contents of the result buffer are available. The second waits until execution is complete.

In addition, you can pass the Event object to another ray query, thus establishing a dependency. The second ray query in this case would start only after the first one has finished:

```
api_->IntersectBatch(rays, numrays, intersections, event, nullptr);
```

Memory API

The previously described intersection queries operate on CPU memory. However, the library provides a memory interface that you can use to create buffers in device memory (for example GPU memory). The layout of the buffers is essentially the same as for their CPU counterparts. To create a buffer in remote device memory, use the following method of the `IntersectionApi` class:

```
virtual Buffer* CreateBuffer(size_t size, void* initdata) const = 0;
```

The buffer can be mapped and unmapped with the following calls:

```
virtual void MapBuffer(Buffer const* buffer, MapType type, size_t offset, size_t size, void** data,
Event** event) const = 0;

virtual void UnmapBuffer(Buffer const* buffer, void* ptr, Event** event) const = 0;
```

Note that these operations are asynchronous and you need to establish correct dependencies to intersection queries to ensure they work as intended.

OpenCL interop

There is a way to use existing OpenCL contexts in the API as well as to share existing OpenCL buffers with the application code. To create an API instance using an existing OpenCL context, the user can call the following:

```
Cl_int status = clGetDeviceIDs(platform[0], type, 1, &device, nullptr);
cl_context rawcontext = clCreateContext(nullptr, 1, &device, nullptr, nullptr, &status);
cl_command_queue queue = clCreateCommandQueue(rawcontext, device, 0, &status);
api_ = IntersectionApi::CreateFromOpenClContext(0, rawcontext, &device, &queue, 1);
```

The user must ensure that there is only one device present in the context passed into the API.

To share the buffer, you can use this code:

```
cl_mem rays_buffer = clCreateBuffer(rawcontext, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
sizeof(ray), &r, &status);

rays = api_->CreateFromOpenClBuffer(rays_buffer);
```

Global options

Options control various aspects of Radeon Rays behavior. Currently, they are mainly used to control acceleration structure construction and traversal algorithms. Refer to the header file for the complete set of supported options. For example, to set an option you use this call:

```
api_->SetOption("bvh.builder", "sah");
```

Acceleration strategies

The default acceleration structure is Bounding Volume Hierarchy (BVH) using spatial median splits. It maintains fast build times and provides decent intersection performance. You can enable SAH builder using the global option and trade off construction time for better intersection performance.

If you need faster refits (for example if geometry is frequently changing position), you can enable two level BVH, which doesn't get re-created every time the geometry transform is changed.

For scenes containing instances or motion blur, two level BVH is used by default.

Releasing entities

All the entities created via the Radeon™ Rays interface should be released when an application shuts down. The following methods are available to release shapes, buffers, and events:

```
api_->DeleteShape(shape);
api_->DeleteBuffer(buffer);
api_->DeleteEvent(event);
```

Destroy the API instance itself with:

```
IntersectionApi::Delete(api_);
```