

CHAPTER

WEBRAYS: RAY TRACING ON THE WEB

Nick Vitsas,^{1,2} Anastasios Gkaravelis,^{1,2} Andreas A. Vasilakis,^{1,2} and

Georgios Papaioannou¹

¹Athens University of Economics and Business

²Phasmatic

ABSTRACT

This chapter introduces WebRays, a GPU-accelerated ray intersection engine for the World Wide Web. It aims to offer a flexible and easy-to-use programming interface for robust and high-performance ray intersection tests on modern browsers. We cover design considerations, best practices, and usage examples for several ray tracing tasks.

1 INTRODUCTION

Traditionally, ray tracing has been employed in rendering algorithms for production and interactive visualization running on high-end desktop or server platforms, relying on dedicated, native, and close-to-the-metal libraries for optimal performance [2, 5]. Nowadays, the Web is the most ubiquitous, collaborative, convenient, and platform-independent conveyor of visual information. The only requirement for accessing visual content online is a web browser on an Internet-enabled device, dispensing with the dependence on additional software.

The potential of the Web has driven Khronos to release the WebGL specifications in order to standardize a substantial portion of graphics acceleration capabilities on the majority of new consumer devices and browsers. As a result, frameworks like ThreeJS and BabylonJS emerged, enabling various visualization applications that take advantage of the Web as a platform [4]. However, these solutions have been explicitly designed for widespread commodity rasterization-based graphics, before ray tracing was popular. There is currently no functionality exposed that accommodates client-side GPU-accelerated ray tracing on this platform.

We developed WebRays to fill this gap, by aiming for an as-thin-as-possible abstraction over any underlying graphics, or potentially compute, application programming interface (API). Following the successful design of modern ray



Figure 1. The Space Station scene rendered using a unidirectional path tracer implemented in WebRays (trace depth 4).

tracing engines, WebRays exposes an easy-to-use and explicit API with lightweight support for acceleration data structures, to enable ray/triangle intersection functionality in Web-based applications. It is by design not intended to implement a specific image synthesis pipeline. Instead, it offers a framework for ray tracing support that can either act as a core API for implementing diverse rendering pipelines and algorithms or as a complementary toolkit to harmoniously coexist with and enhance rasterization-based graphics solutions (see the example in Figure 1). This is achieved by allowing access to ray tracing functionality both with stand-alone intersection procedures and individual ray tracing function calls from within a standard shader program.

A prototype version of the API was successfully utilized for the implementation of the Rayground platform [7]. Rayground is an open, cross-platform, online integrated development sandbox and educational resource for fast prototyping and interactive demonstration of ray tracing algorithms. It offers a programming experience similar to a standard, GPU-accelerated ray tracing pipeline that allows customization of the basic ray tracing stages directly in the browser.

In this chapter, we cover the system architecture and programming interface design as well as provide representative usage examples via code snippets for a gentle introduction to WebRays.

2 FRAMEWORK ARCHITECTURE

Modern APIs that support ray intersection acceleration, such as DXR and OptiX [2, 5], are designed to give explicit control to developers. Users are given as much freedom as possible over resource management and control flow. WebRays sets a similar goal.

2.1 DESIGN GOALS

We want to bring ray tracing functionality to modern browsers, with no specific conditions or exceptions and irrespective of platform. We strive for independence from a particular specialized or low-level graphics API. Users refer to engine resources using opaque handles, and information about internal storage types is only communicated to the user in order to help achieve optimal performance. Having often been on the user end of similar frameworks, we recognized the need for a modern API that simultaneously supports two different approaches to ray tracing:

- > *Wavefront*: A simple, general-purpose execution model for tracing arbitrary ray batches in bulk.
- > *Megakernel*: A programming interface for ray tracing within a GPU shader.

In wavefront ray tracing, the entire process is divided into small, discrete phases, corresponding to specific kernels, which are executed successively and process data in parallel (Figure 2, left). On the other hand, in the megakernel approach, a single kernel is launched. All operations including intersections, visibility determination, shading, etc. take place within this single kernel (Figure 2, right). This allows user code to closely follow the underlying algorithm and can therefore

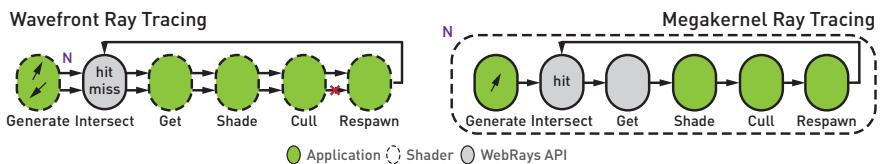


Figure 2. Wavefront and megakernel ray tracing conceptual models. Left: ray tracing consists of simple parallel steps. Note that all stages (except Intersect) can be executed on either host side or device side. Right: A monolithic kernel is responsible for the entire ray tracing process.

be considered more programmer-friendly. However, given the nature of modern streaming processors, it can lead to serious occupancy and divergence issues as the potential execution paths and access patterns may vary significantly.

Each approach operates under different design considerations, and we wanted to properly support both. In WebRays, this is completely up to application logic, and the user can also combine the two strategies. To this end, the API offers both a *host-side* interface implemented in JavaScript and a *device-side* interface implemented in the backend's shading language (GLSL). Performance-sensitive parts of the library are written in C++ and compiled to WebAssembly.

WebRays is not intended as a stand-alone pipeline for image synthesis. It does not introduce features such as specific shading models and material properties, nor does it enforce a particular image synthesis algorithm. In the same spirit, with WebRays you do not write kernels that respond to specific events. The developer is free to implement their own pipeline, which of course comes at the cost of more user-defined "glue" code. With this generic, yet complete, ray tracing support, one can even build wrapper APIs on top of WebRays to conform to popular ray tracing frameworks. Finally, capturing the typical needs of practical rendering implementations, it currently supports triangle primitives.

2.2 HOST-SIDE API

The JavaScript API is accessible through the standard web development workflow. Working with this host-side API is very similar to most ray intersection engines. To start, users submit triangle meshes to build acceleration data structures over their geometric data. These acceleration data structures are built on the CPU and later uploaded to the GPU for fast intersections. The API uses handles to refer to these structures on both the host and the device sides. Users "submit" rays for intersection by allocating and populating appropriately formatted ray buffers. Similarly, intersection or occlusion results are returned to the user through appropriately formatted intersection buffers. These allocations are handled by the user to allow for full control over the application's GPU memory management. To facilitate a wavefront approach, the host-side API offers functions that take a batch of rays, identify ray/geometry intersections in parallel, and finally store the results in properly formatted buffers. Nothing needs to leave the GPU in this process, and the results are ready to be consumed in a shader or fetched back to the CPU.

2.3 DEVICE-SIDE API

The device-side API enables users to write megakernels and hybrid rendering solutions. The biggest challenge is to be able to provide access to geometric data and ray tracing functionality in a flexible and unobtrusive manner. Like early OpenGL, current programmable graphics pipelines for the Web are configured with string-based shaders provided in source form. In WebRays, the shader code that constitutes the device-side API for ray intersection functionality is a simple string, automatically generated by the engine. Developers can acquire this string via an API call and prepend it to their own source code.

The attached code gives access to in-shader intersection routines and helper functions for accessing geometric properties of intersected objects. We did not want to offer yet another shader abstraction layer as this would not fit well with existing graphics applications that want to add ray intersection functionality. All calls and variables are appropriately isolated in their own namespace, in order to avoid clashes with users' code.

The WebRays intersection engine allocates GPU resources for internal data structures, which need to be bound to the user's program for the device-side API to function properly. These resources are commonly passed from the host to the device through shader binding locations. The API offers a function that returns a binding table containing the name, type, and value of the resource that needs to be bound to the shader before execution.

WebRays is currently implemented on top of WebGL 2.0. The remainder of the text exposes several WebGL-specific implementation details and best practices. Familiarity with any OpenGL version will greatly help the reader.

2.4 ENGINE CORE

Device-side computation in WebGL 2.0 is available via plain fragment shaders. Also, memory allocation and sharing must be handled through standard textures. When memory needs to be exchanged between WebRays and the rest of the application, e.g., passing a ray buffer for intersection and receiving the results, it is passed using appropriately formatted textures. Thankfully, the support of multiple render targets in WebGL 2.0 makes this process quite streamlined.

The intersection engine and the application achieve interoperability by both operating on the same WebGL context. In order to help users manage their GPU resources, WebRays is transparent about its memory requirements, layouts, and

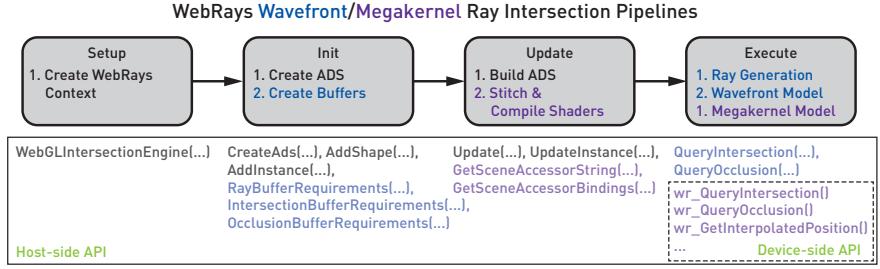


Figure 3. Top: basic WebRays application control flow for wavefront and megakernel ray tracing. Steps in black are common for both pipelines. Bottom: a summary of the basic functions exposed by the WebRays API and how these map to each application stage.

bindings. These design decisions help the intersection engine work closely with the application and allow for maximum performance by keeping ray tracing resources resident on the GPU every step of the way.

2.5 ACCELERATION DATA STRUCTURES

Geometry loading from external sources is left to application code, similar to other modern APIs [2, 5]. WebRays is optimized for triangle meshes, which constitute the most common geometric representation for 3D models. The API supports two types of acceleration data structures (ADS): top-level (scene-wide) acceleration structure (TLAS) and bottom-level (per-object) acceleration structure (BLAS). A TLAS can contain one or more BLASs. Instancing is also supported by inserting the same BLAS into the TLAS multiple times with different transformation matrices.

The internals of the WebRays data structures are not exposed to the user. The library internally allocates and deallocates GPU resources for triangle data. We employ bounding volume hierarchies (BVHs) as acceleration data structures, as they have proven their worth in terms of balanced performance and construction cost. Specifically, we use a custom variant of a wide BVH [8], which has shown good performance for incoherent rays.

3 PROGRAMMING WITH WEBRAYS

The basic control flow of a WebRays application is outlined in Figure 3. The general steps are *setup*, *init*, *update*, and *execute*, all of which are determined by the context of the intended execution model and algorithm.

- > *Setup*: The application gets a handle to the WebRays host-side API functions (Section 3.1).
- > *Init*: The ADSs based on the scene's geometry are created (Section 3.2). In case of wavefront ray intersections (Section 3.5), ray and intersection buffers are also allocated and filled accordingly (Section 3.3).
- > *Update*: The ADSs are actually built (or updated), and data is synchronized between CPU and GPU memory (Section 3.2). In the case of megakernel ray tracing, the dynamically created WebRays device-side API code must be prepended to the user-provided fragment shader, before compilation.
- > *Execute*: Rays are generated (Section 3.4) and their intersections handled either within a viewport-filling rectangle draw pass (wavefront rendering) or directly within the user fragment shader in the megakernel approach (Section 3.6). After execution has finished, the resulting intersection data can be used by the application in a multi-pass manner by repeating the cycle.

3.1 SETUP

For WebRays to function in the HTML document, users specify a canvas object that will be enabled with WebGL capabilities at runtime. The WebRays API is accessible via a single JavaScript file, loaded as usual:

```
1 <body>
2   <canvas id="canvas" width="640" height="480"></canvas>
3 </body>
4 <script src="webrays.js"></script>
```

Because all access to device-side rendering is handled via WebGL, one also needs to access the WebGL context in one's own script:

```
1 var gl = document.querySelector("#canvas").getContext("webgl2");
```

The intersection engine context is initialized by providing the newly created WebGL context:

```
1 var wr = new WebRays.WebGLIntersectionEngine(gl);
```

From this point on, users can start using both the `gl` and the `wr` objects to perform rasterization and ray tracing, respectively.

3.2 POPULATING THE ACCELERATION DATA STRUCTURES

Before tracing any rays, a bottom-level acceleration structure must first be constructed for the mesh geometry. In the following code, we assume that the mesh has been loaded using an external library with the corresponding mesh class providing access to the vertex, normal, and texture coordinate buffers, along with their respective strides:

```

1  function build_blas_ads(mesh) {
2    let blas = wr.CreateAds({ type: "BLAS" });
3    let shape = wr.AddShape(blas,
4                           mesh.vertex_data, mesh.vertex_stride,
5                           mesh.normal_data, mesh.normal_stride,
6                           mesh.uv_data, mesh.uv_stride,
7                           mesh.face_data);
8    return [blas, shape];
9 }
```

Each vertex position and normal vector is a `float[3]` and each texture coordinate pair a `float[2]`. For the index buffer, the API expects an `int[4]`. The `xyz` components contain the offsets to each of the triangle's vertices within the attribute buffers. The `w` component is user-provided and is always available during intersections. This can be used to store application-specific data like material indices, texture indices, and more.

The `blas` variable is an opaque handle that can later be passed to host-side or device-side WebRays API functions in order to refer to this specific structure. The user can add multiple shapes to a single BLAS. The returned `shape` variable is a handle that is used by the engine to identify the specific geometry group within the BLAS.

A TLAS is created similarly to a BLAS, but the user adds *instances* of existing BLASs to the TLAS via their handles:

```

1 let blases = [];
2 let tlas = wr.CreateAds({ type: "TLAS" });
3 for (mesh in meshes) {
4   let [blas, shape] = build_blas_ads(mesh);
5   let instance = wr.AddInstance(tlas, blas, mesh.transform);
6   blases.push(blas);
7 }
```

An instance's transformation can later be updated using the returned `instance` handle with the following code:

```
1 wr.UpdateInstance(tlas, instance, transform);
```

Up to this point, none of the provided scene geometry resources have been

submitted to the GPU. The user must call the following function in order to commit and finalize the created structures:

```
1 let flags = wr.Update();
```

The returned flags indicate whether the device-side accessor code string or the bindings were affected by the update operation. It is advised to call this function every frame, as it does not incur an additional cost when no actual update is required and it enables the programmer to react on a significant change.

For example, if the returned flags indicate a change in the device-side API, the user is expected to get the new device-side API code as well as the updated bindings using the following snippet:

```
1 let accessor_code = wr.GetSceneAccessorString();
2 let bindings      = wr.GetSceneAccessorBindings();
```

The accessor code is a plain string that needs to be prepended to the user's shader code before compilation. Bindings are mostly relevant at program invocation time during rendering.

3.3 RAY AND INTERSECTION BUFFERS

The ray structure declaration that WebRays uses internally resembles the one described in the *Ray Tracing Gems* chapter “What Is a Ray?” [6]. Intersection data use a packed representation and are not intended to be directly read by the user. However, they can be passed to API functions within the shader in order to get intersection-specific attributes. Occlusion data simply indicate if any geometric object was found between the user-defined ray extents.

Ray buffers have memory requirements for two `vec4` entries. Users are required to fill two buffers corresponding to a `vec4(origin, tmin)` and a `vec4(direction, tmax)` in order to submit rays for intersection. Intersection and occlusion buffers use integer buffers. Because ray tracing commonly requires high-precision arithmetic operations, the API provides helper functions to give a hint to the user about which is the optimal storage and internal texture format.

```
1 let dimensions      = [width, height];
2 let ray_req         = wr.RayBufferRequirements(dimensions);
3 let intersect_req  = wr.IntersectionBufferRequirements(dimensions);
4 let occlusion_req  = wr.OcclusionBufferRequirements(dimensions);
5 let origins          = tex2d_alloc(ray_req);
6 let directions       = tex2d_alloc(ray_req);
7 let intersections    = tex2d_alloc(intersect_req);
8 let occlusions       = tex2d_alloc(occlusion_req);
```

The preceding code allocates two `vec4` buffers for storing ray information and two integer ones for storing intersection and occlusion results. The returned requirement structure contains the type of the buffer along with the expected format and dimensions. The engine supports both 1D and 2D buffers, depending on the dimensions of the passed array. Note that 2D buffers fit naturally to image synthesis. For example, in the current WebGL implementation, the most common buffer type is backed by an appropriately formatted 2D texture. Thus, the `tex2d_alloc` call is an application helper function for allocating such buffers in WebGL.

3.4 RAY GENERATION

Ray buffers can be populated either in the host-side JavaScript code or by utilizing a GPU shader. Each design choice depends on the performance characteristics of the application. The simplest and most efficient method to populate ray buffers is by using the native graphics API. Because WebGL does not have access to compute shaders, launching a shader for ray generation and/or handling of intersection data on the device side is simply performed via a fragment shader over a viewport-filling rectangle in a specific viewport matching the ray batch size. Although one can fill ray data and read back intersections in the host-side code via a read pixels operation, it is advantageous to always perform such operations on the GPU.

For example in WebGL, the textured-backed ray buffers can be attached as render targets in a framebuffer object, and ray properties can be written using a fragment shader. This way, setting the ray direction, origin, and valid ray interval is trivial:

```
1 origin_OUT      = vec4(origin,      tmin);
2 direction_OUT = vec4(direction,  tmax);
```

The newly populated textures can then be passed to WebRays as ray buffers for intersection or used as input in any stage of the application's rendering pipeline. It is important to note that intermediate storage of rays is not a requirement in order to perform intersections. Using the device-side intersection API, a ray can be generated, intersected, and consumed within a single shader. A typical use case is that of shadow rays (Section 4.3.2), where a new ray can be sampled and queried directly within the shader with the result being accessible immediately.

3.5 HOST-SIDE INTERSECTIONS

WebRays offers host-side API functions for ray intersection and occlusion queries. Intersection queries return closest-hit information encoded in an `ivec4`. Occlusion

queries are useful for binary visibility queries. As expected, they are faster than regular intersection queries, due to their any-hit termination criterion.

```
1 let rays = [origins, directions];
2 wr.QueryIntersection(ads, rays, intersections, dimensions);
3 wr.QueryOcclusion(ads, rays, occlusions, dimensions);
```

The ads that the API expects is the same handle that is created on the host side during acceleration data structure creation (Section 3.2). origins and directions are the properly formatted and populated ray buffer textures, whereas intersections and occlusions are appropriately formatted intersection buffer textures (Section 3.3) that will receive intersection and occlusion results, respectively.

Because in essence all these buffers are basic textures, the programmer can pass them to shaders as regular uniform sampler variables. This way, users can bind the already-filled intersection texture and apply application-specific operations in parallel, on the GPU. This approach completely decouples intersection operations from application logic, giving users full control of how they manage ray queries and process the results. The same holds for the consumption of intersection results.

3.6 DEVICE-SIDE INTERSECTIONS

The device-side API enables intersection queries within the shader as well as access to geometric properties of intersected objects. For intersection queries, WebRays provides two function variants, `wr_QueryIntersection` and `wr_QueryOcclusion`. These take as a parameter the handle of a previously created acceleration data structure ads. `wr_QueryOcclusion` returns the result of a visibility test, whereas `wr_QueryIntersection` provides the closest intersection point encoded in a single `ivec4`. This value can be subsequently passed to other intersection-specific API accessor functions to obtain interpolated position, normal, texture parameters, barycentric coordinates, etc.

The following shader code, which implements a basic renderer using ray casting, demonstrates the usage of those two functions. Primary ray hits can either be calculated in the shader (lines 8–10) or obtained from a previous separate shader invocation (line 12). The ads uniform can be treated as a single `int` and passed to the shader with the appropriate `glUniform` variant.

```
1 uniform int ads;
2 uniform sampler2D origins;
3 uniform sampler2D directions;
4 uniform isampler2D intersections;
```

```

5 ...
6 ivec2 coords    = ivec2(gl_FragCoord.xy);
7 // Case 1: Query primary ray intersections.
8 vec4 origin     = texelFetch(origins, coords, 0);
9 vec4 direction  = texelFetch(directions, coords, 0);
10 ivec4 hit       = wr_QueryIntersection(ads, origin, direction, tmax);
11 // Case 2: Or obtain already calculated primary ray hits.
12 ivec4 hit       = texelFetch(intersections, coords, 0);
13 ...
14 // Miss
15 if (!wr_IsValidIntersection(hit)) {
16     color_OUT = ...
17     return;
18 }
19 // Intersect
20 ivec4 face      = wr_GetFace(ads, hit);
21 vec3 geom_normal = wr_GetGeomNormal(ads, hit);
22 vec3 normal     = wr_GetInterpolatedNormal(ads, hit);
23 vec3 position   = wr_GetInterpolatedPosition(ads, hit);
24 ...
25 // Shade intersected point
26 color_OUT = ...

```

4 USE CASES

This section describes how a number of fundamental and modern applications of ray tracing to image synthesis can be implemented with WebRays. We demonstrate how the versatile design and high-performance implementation of WebRays can be used in pure ray tracing implementations (Sections 4.1 and 4.2) as well as hybrid rendering (Section 4.3). Timings across several devices are provided for all experiments in Figure 4, to give a feeling of the expected performance. The complete source code of the WebRays examples is provided in the accompanying repository of the book. The source code aims to get developers started with WebRays, moving from trivially simple to moderately complex ray tracing examples. Furthermore, we briefly describe Rayground (Section 4.4), an interactive authoring platform for ray tracing algorithms based on WebRays.

4.1 AMBIENT OCCLUSION

Ambient occlusion is a non-physically-accurate illumination technique, highly popularized in the games industry due to its relative simplicity and efficiency. Using ray tracing, it can be estimated by stochastically sampling the visibility of the shaded point over a hemisphere centered at its normal vector.

The following code demonstrates how WebRays visibility queries can be used to estimate ambient occlusion in a fragment shader. The `sample_count` corresponds to the number of directions selected by importance sampling, and `dmax` the near-field extent. The `position` and `normal` of the shaded point come from the

Scene (triangles)	(35k)	(143k)	(143k)	(109k)	(51k)	(4.6k)	(123k)
Device	(35k)	(143k)	(143k)	(109k)	(51k)	(4.6k)	(123k)
nVidia RTX 2080	21	5.3	51	12.2	6.2	4.3	44
nVidia GTX 1060 [†]	50	5.1	140	35.0	19.5	3.0	147
nVidia GTX 970	60	5.8	188	41.0	25.0	4.5	193
AMD RX 580	54	6.0	200	37.0	9.0	5.0	192
Intel HD 630 [†]	290	34.2	626	131.0	93.0	14.0	618

Figure 4. Expected performance on a variety of desktop and laptop[†] GPU devices. Timings show the total frame time, in milliseconds, of each use case, including shading, intersections, and post-processing. Scenes are rendered at a native 1024×768 resolution.



Figure 5. Monte Carlo integration in WebRays. Left: the Fireplace scene using ray traced near-field ambient occlusion. Middle and right: a unidirectional path tracer (trace depth 4) with next event estimation used for the Fireplace scene (middle) and the ToyCar model (right).

previous ray casting example of Section 3.6, but they can be derived by other means, such as a forward rasterization pass (Section 4.3). The shaded result using the Fireplace scene and a near-field extent of 1.5 m is shown in Figure 5, left.

```

1 ...
2 float occlusion = 0.0;
3 for (int s = 0; s < sample_count; s++) {
4     vec3 direction = CosineSampleHemisphere(normal);
5     bool occluded = wr_QueryOcclusion(ads, position, direction, dmax);
6     occlusion += occluded ? 1.0 : 0.0;
7 }
8 occlusion /= float(sample_count);
9 // Ambient lighting visualization
10 color_OUT = vec4(vec3(1.0 - occlusion), 1.0);

```

4.2 PATH TRACING

Path tracing is an elegant method that estimates the integral of the rendering equation using Monte Carlo simulation. It produces accurate photorealistic images because all kinds of light transport paths are supported and sampled.

Using WebRays, a path tracer can be developed in either a wavefront or a megakernel manner. Due to the lack of bindless resources in WebGL, materials should be packed into a shared buffer and textures should be packed into a texture atlas, in order to minimize texture bindings. In the following code, a unidirectional path tracer with next event estimation is implemented using a wavefront approach, where rays are generated using a pinhole camera and intersections are resolved using the primitive and material ID of the intersected primitive.

```

1 [origins, dirs] = Generate(camera); // Get primary rays.
2 while(depth < depthmax) {
3     // Ray intersection test
4     wr.QueryIntersection(ads, [origins, dirs], hits, dimensions);
5     // Get new shadow and outgoing rays.
6     [origins, occ_dirs, dirs] = ResolveIntersections(hits);
7     // Shadow rays test
8     wr.QueryOcclusion(ads, [origins, occ_dirs], occs, dimensions);
9     // Compute local illumination.
10    ResolveDirectIllumination(hits, occs);
11    depth++; // Increase path length.
12 }
```

The `ResolveIntersections` call is responsible for populating the next outgoing ray buffer and the shadow ray buffer and is listed in the following code. This can be done efficiently by launching a single 2D kernel. Using multiple render targets, the kernel is responsible for populating the ray texture buffers. The last two textures correspond to the shadow rays and the outgoing rays leaving the intersection point. The first texture stores the ray origin, which is the same for both rays.

```

1 vec3 wo      = -direction_IN.xyz;
2 vec3 origin = wr_GetInterpolatedPosition(ads, hit);
3 vec3 normal = wr_GetGeomNormal(ads, hit);
4
5 float light_distance;
6 vec3 shadow_ray, wi;
7 LightSample(origin, /*out*/shadow_ray, /*out*/light_distance);
8 BxDF_Sample(origin, wo, /*out*/wi, /*out*/scattering_pdf);
9
10 direction_OUT      = vec4(wi, tmax);
11 shadow_direction_OUT = vec4(shadow_ray, light_distance - RAY_EPSILON);
12
13 origin      += normal * RAY_EPSILON;
14 origin_OUT   = vec4(origin, RAY_EPSILON);
```

After executing this kernel, the textures are forwarded to `QueryIntersection` and `QueryOcclusion` respectively. Direct illumination is computed in `ResolveDirectIllumination` based on the results produced from `QueryOcclusion` and the bidirectional scattering distribution function BxDF of the hit point.

```

1 vec3 origin      = wr_GetInterpolatedPosition(ads, hit);
2 vec3 Li          = LightEval(origin, light, wi, light_pdf, light_dis);
3 ...
4 vec3 Ld          = vec3(0.0);
5 int occlusion   = texelFetch(occlusions, coords, 0).r;
6 if(occlusion == 0) {
7     vec3 BxDF    = BxDF_Eval(origin, wo, wi);
8     Ld           += throughput * BxDF * NdL * Li / light_pdf;
9 }
10 color_OUT = vec4(Ld, 0.0);

```

Figures 1 and 5 (middle and right) show how this efficient WebRays path tracing implementation can be used to compute global illumination images for interior and outdoor scenes.

4.3 HYBRID RENDERING

Rendering on the Web is usually performed using a rasterization-based pipeline via either a deferred or a forward rendering approach. With WebRays, ray tracing becomes available to both desktop and mobile browsers and can be used to replace or complement some components of the typical rasterization pipeline, enhancing the quality of the rendered image.

AMBIENT OCCLUSION

In real-time applications, either ambient occlusion is precomputed, for static scenes, or near-field ambient occlusion is computed in real time, using screen-space information stored in the G-buffer. Screen-space techniques, due to their view-dependent nature, fail to capture occlusion from objects that are offscreen or occluded by other geometric objects from the camera's point of view. Ray tracing can be utilized to compute accurate ambient occlusion without these drawbacks. The shaded point's position and normal can be reconstructed from the G-buffer, and ambient occlusion can be estimated as described in Section 4.1.

SHADOWS

Ray-traced shadows provide accurate and crisp boundaries and can work in complex lighting conditions such as arbitrarily shaped light sources. They offer a superior method for shadow calculations than the prevalent method for computing

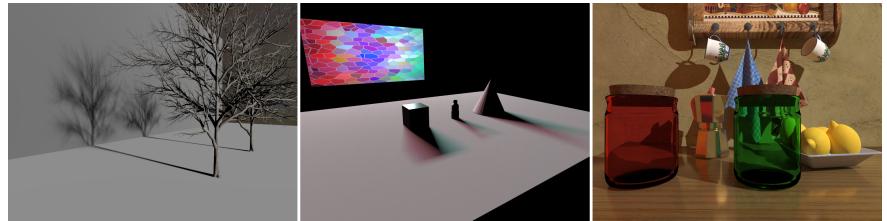


Figure 6. Hybrid rendering in WebRays. Left and middle: accurate soft shadows in the Tree and Mosaic scenes. Right: reflection and transmission events (trace depth 6) captured in the Kitchen Table scene.

shadows in real-time graphics that uses shadow maps [1].

In a hybrid renderer, the shaded point position can be reconstructed from the depth buffer, populated during the G-buffer rasterization or a depth prepass step. Then, any number of occlusion rays can be cast toward the light sources and checked for visibility.

```

1 // Shadow ray computation
2 vec3 origin      = ReconstructPositionFromDepthBuffer();
3 float distance;
4 vec3 direction;
5 LightSample(origin, /*out*/direction, /*out*/distance);
6 // Shadow visibility test
7 bool occluded   = wr_QueryOcclusion(ads, origin, direction, distance);
8 float visibility = occluded ? 0.0 : 1.0;
```

Occlusion queries are very fast as they terminate on the first encountered primitive intersection. Alpha-tested geometry would require a more complex hit kernel, where transparency of the hit point on the primitive should be taken into account. Soft shadows can be easily computed by sampling directions toward the light source surface area. Figure 6, left, presents soft shadows of thin geometry produced from a distant light source, whereas in the middle of the figure, colored soft shadows produced from a textured area light are demonstrated.

REFLECTION AND REFRACTION

Complex light paths resulting from reflected and transmitted light are very difficult to compute using a rasterization pipeline. Ray tracing offers a robust way to handle such difficult phenomena.

In a hybrid rendering pipeline, rays are spawned from the reconstructed position from the depth buffer toward a random direction inside the reflection or

transmission lobe defined by the material properties and traced into the scene. Each hit point is then evaluated and shaded, and a new ray is spawned and traced into the scene. This recursive procedure is usually performed up to a maximum number of iterations `depthmax` based on the number of light paths that the specific scenario requires. An example using WebRays is presented in the following code, where the recursive procedure is performed inside a single shader and rays are traced using the in-shader API call `wr_QueryIntersection`. Each hit point is evaluated and a new ray is spawned, or the procedure terminates in the event of a miss. Accurate reflection and transmission results from the above shader are presented in Figure 6, right.

```

1 // Geometry information computation
2 vec3 position = ReconstructPositionFromDepthBuffer();
3 vec3 normal   = ReconstructNormalFromGBuffer();
4 BxDF bxdf    = ReconstructMaterialFromGBuffer();
5 vec3 wo       = normalize(u_camera_pos - position);
6
7 // Compute new ray based on the material.
8 vec3 wi, throughput = vec3(1.0), color = vec3(0.0);
9 Sample_bxdf(bxdf, normal, wo, /*out*/wi, /*out*/throughput);
10
11 while(depth < depthmax) {
12     ivec4 hit  = wr_QueryIntersection(ads, position, wi, tmax);
13     if (!wr_IsValidIntersection(hit)) { // Miss
14         color += throughput * EvaluateEnvironmentalMap(wi);
15         break;
16     }
17     else { // Surface hit
18         position = wr_GetInterpolatedPosition(ads, hit);
19         normal   = wr_GetInterpolatedNormal(ads, hit);
20         bxdf     = GetMaterialFromIntersection(hit);
21         wo       = -wi;
22
23         vec3 Ld  = EvaluateDirectLight(position, normal, bxdf, wo);
24         color += throughput * Ld;
25
26         Sample_bxdf(bxdf, normal, wo, /*out*/wi, /*out*/throughput);
27     }
28     depth++;
29 }
30 color_OUT = color;

```

4.4 RAY TRACING PROTOTYPING PLATFORM

WebRays has been deployed at the core of the Rayground platform, which is hosted at <https://www.rayground.com>. The Rayground pipeline offers a high-level framework for easy and rapid prototyping of ray tracing algorithms. More specifically, it exposes one declarative stage and four programmable ones. In the declarative stage, users describe their scene using simple shape primitives and material properties. The scene is then submitted to WebRays in order to build and traverse the acceleration data structure. The four programmable stages are

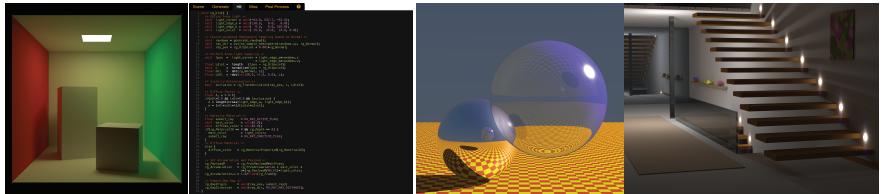


Figure 7. Left: the Rayground interface, with the preview window and the shader editor. Right: two representative ray traced projects created online on this platform: Whitted ray tracing and stochastic path tracing.

the standard *generate*, *hit*, *miss*, and *post-process* events. The graphical user interface consists of two discrete sections, the preview window and the shader editor. Visual feedback is interactively provided in the preview canvas, and the user performs live source code modifications as shown in Figure 7.

5 CONCLUSIONS AND FUTURE WORK

We have presented WebRays, a GPU-accelerated ray intersection framework able to provide photorealistic 3D graphics on the web. The versatile design of WebRays allows it to adapt to a broad range of application requirements, ranging from simple intersection queries to complex visualization tasks.

Admittedly, the low-level nature of the API requires a good amount of boilerplate code from the user. In the future, we intend to provide wrapper abstraction APIs that model more constrained yet popular programmable ray tracing pipelines. Keeping an eye toward future advances on Web-based accelerated graphics, we plan to offer a WebGPU backend implementation of WebRays, as soon as the latter becomes available to browsers, in order to provide a standard way to express ray tracing ubiquitously, so it ultimately benefits the entire 3D online graphics industry.

ACKNOWLEDGMENTS

This work was funded by the Epic MegaGrants grant program from Epic Games Inc. The ToyCar model, created by Guido Odendahl, was downloaded from Khronos Group repository. The San Miguel (whose Tree model was used) and Fireplace scenes were downloaded from McGuire’s Computer Graphics Archive [3]. The remaining scenes were created by the authors.

REFERENCES

- [1] Eisemann, E., Schwarz, M., Assarsson, U., and Wimmer, M. *Real-Time Shadows*. A K Peters/CRC Press, 1st edition, 2011. DOI: [10.1201/b11030](https://doi.org/10.1201/b11030).
- [2] Haines, E. and Akenine-Möller, T. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, 2019. DOI: [10.1007/978-1-4842-4427-2](https://doi.org/10.1007/978-1-4842-4427-2).
- [3] McGuire, M. Computer graphics archive. <https://casual-effects.com/data>, 2017.
- [4] Mwalongo, F., Krone, M., Reina, G., and Ertl, T. State-of-the-art report in Web-based visualization. *Computer Graphics Forum*, 35(3):553–575, 2016. DOI: [10.1111/cgf.12929](https://doi.org/10.1111/cgf.12929).
- [5] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, 29(4):66:1–66:13, July 2010. DOI: [10.1145/1778765.1778803](https://doi.org/10.1145/1778765.1778803).
- [6] Shirley, P., Wald, I., Akenine-Möller, T., and Haines, E. What is a ray? In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 15–19. Apress, 2019.
- [7] Vitsas, N., Gkaravelis, A., Vasilakis, A.-A., Vardis, K., and Papaioannou, G. Rayground: An online educational tool for ray tracing. In *Eurographics 2020—Education Papers*, pages 1–8, 2020. DOI: [10.2312/eged.20201027](https://doi.org/10.2312/eged.20201027).
- [8] Ylitie, H., Karras, T., and Laine, S. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of High Performance Graphics*, HPG ’17, 4:1–4:13, 2017. DOI: [10.1145/3105762.3105773](https://doi.org/10.1145/3105762.3105773).

Open Access This book is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this book or parts of it. The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.