Realtime Computer Graphics on GPUs

Realtime Raytracing

Jan Kolomazník

Department of Software and Computer Science Education
Faculty of Mathematics and Physics
Charles University in Prague

Computer Graphics Charles University

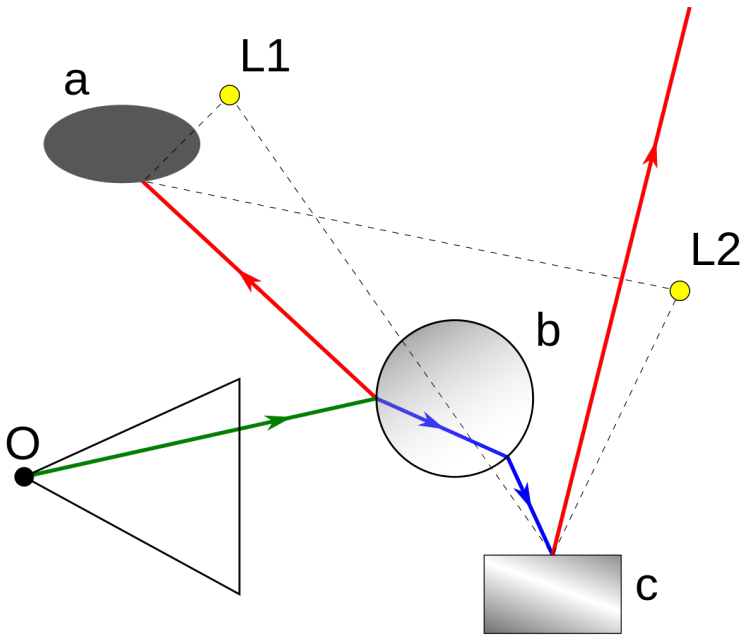
Terminology

RAYCASTING

- Arthur Appel in 1968
- Simplest ray-based method
- Shoot rays from the eye and find the closest object blocking the path of that ray

Whitted's Raytracing

- Turner Whitted in 1979
- Extending raycasting by using recursion
- New reflected/refracted rays are generated and color is recursively propagated back
- Perfect reflections/refractions
- Simple shadow computation – ray to light source



Distributed Raytracing

- Modeling soft effects:
  - Soft shadows
  - Depth of field
  - Soft reflections
- Shooting multiple rays
  - Sampling the domain (angle, lens)
  - Weighted average of the payloads
  - Suppression of alias

Path Tracing

- James Kajiya in 1986
- Systematic approach to solving global illumination problem
- Rendering equation:
  - Integral equation in which the radiance leaving a point is given as the sum of emitted plus reflected radiance under a geometric optics approximation
- Raytracing using Monte-Carlo method for the rendering equation solving
- Bidirectional path tracing – faster convergence

Comparison with Rasterization

- Advantages
  - General camera model
  - Reflection and refraction – exact computation with minimum code
  - Simple computation of hard and soft shadows
  - Usable for preprocessing and/or rendering pass in combination with rasterization (lightmaps, AO, exact reflections as postprocess, . . .)
  - Logarithmic acceleration structures
- Disadvantages
  - Expensive build of high quality acceleration structure
  - Worse support of dynamic scenes – acc. structure rebuild is not trivial
  - Suitable HW only recently
  - Computationally more expensive
  - Geometry cannot be streamed

Overview of Typical Shaders in Ray Tracing Engines

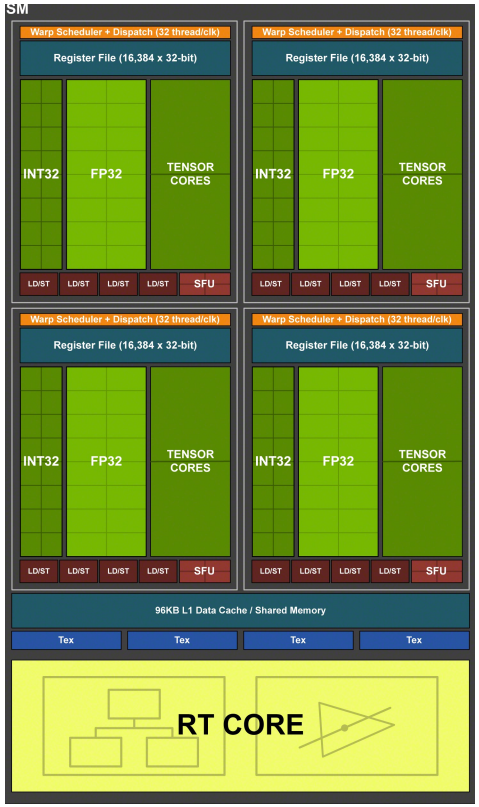
Ray tracing engines utilize various specialized shaders to achieve realistic rendering. The primary shaders used are:

- **Ray Generation Shader**
- **Intersection Shader**
- **Any-Hit Shader**
- **Closest-Hit Shader**
- **Miss Shader**
- **Callable Shader**

Nvidia RTX

New HW Architecture

- First GPUs with raytracing support (NVidia Turing)
- RT cores (ASIC –Application-specific integrated circuit)
- Usable in OptiX, MS DirectX Raytracing API, Vulcan raytracing extension



RT Cores in NVIDIA GPUs

- **RT Cores**:
  - **Bounding Volume Hierarchy (BVH) Traversal**:
    - RT Cores accelerate BVH traversal, a hierarchical structure that organizes scene geometry.
    - Efficiently skips large portions of the scene that rays do not intersect.
  - **Ray/Triangle Intersection Testing**:
    - RT Cores perform fast intersection tests between rays and triangles.
    - Reduces computational load on the main GPU cores (CUDA cores).
  - **Parallel Processing**:
    - RT Cores operate in parallel with traditional rasterization pipelines.
    - Allows simultaneous execution of rasterization and ray tracing workloads.

VULCAN EXTENSIONS

Raytracing available in Vulcan as an extension

- **VK_KHR_ray_tracing_pipeline**: Core ray tracing functionality.
- **VK_KHR_acceleration_structure**: Optimizes ray tracing performance.
- **VK_KHR_ray_query**: Allows ray queries within shaders.
- **VK_KHR_pipeline_library** and **VK_EXT_descriptor_indexing**: Manage complex configurations.

SHADERS

- New GLSL shader types:
  - Any hit
  - Intersection
  - Miss
  - Closest hit
- All shaders must be available – ray may intersect any object

Handle Ray Tracing Shaders

```
// Example of a ray generation shader
#version 460
#extension GL_EXT_ray_tracing : require

layout(set = 0, binding = 0) uniform accelerationStructureEXT topLevelAS;
layout(location = 0) rayPayloadEXT vec3 hitValue;

void main() {
    vec3 origin = vec3(0.0, 0.0, 0.0);
    vec3 direction = vec3(0.0, 0.0, -1.0);
    traceRayEXT(topLevelAS, 0, 0xFF, 0, 1, 0, origin, 0.0, direction, 10000.0, 0);
}
```

Bottom-Level Acceleration Structures (BLAS)

- **Purpose**:
  - Represent individual objects or meshes in the scene.
- **Structure**:
  - Consist of geometric primitives like triangles.
  - Optimized using a Bounding Volume Hierarchy (BVH).
- **Usage**:
  - Built once and reused for multiple frames.
  - Can be updated if the geometry changes.

Top-Level Acceleration Structures (TLAS)

- **Purpose**:
  - Represent the entire scene, including instances of BLAS.
- **Structure**:
  - Contains instances of BLAS with their transformations.
  - Organized in a BVH for efficient traversal.
- **Usage**:
  - Built from instances of BLAS.
  - Updated when the scene layout changes.

Optix

OptiX I

- OptiX is not just a raytracer
- OptiX is framework for creating application using raytracing, independent from any specific method
- Build on CUDA architecture
- Most of the components programmable
- Usable not only for CG, but also for:
  - collision detection
  - visibility determination
  - sound propagation simulation
  - volume estimation of complicated objects
  - ...

OptiX II

- Abstract model of generic raytracer
- Future-proofed – build to scale with future development of powerful GPUs
- Similar abstraction to OpenGL/DirectX/Vulcan
- Mechanism for execution of custom CUDA C code
  - Shading + recursive rays
  - Camera model, ray generator
  - Ray payload
  - Intersection with arbitrary geometry (e.g. exact sphere without tesselation)
  - ...

OptiX Versions

- **OptiX 6 and earlier**
  - High-level API: automatic memory & launch handling
  - Built-in program linking and variable scoping
- **OptiX 7+**
  - Low-level GPU programming model, RTX optimized
  - Full control: manual SBT, memory, and ray dispatch
  - Backward-incompatible with earlier versions

PROGRAMS I – SHADER STAGES

- 8 types of GPU programs supported by OptiX
- **Ray Generation** – Launch entry point, invoked per-pixel/sample
- **Exception** – Called on invalid state (e.g., stack overflow)
- **Closest Hit** – Executed on closest geometry hit (used for shading)
- **Any Hit** – Invoked on all intersections (used for early-out, transparency, shadows)
- **Intersection** – User-defined geometry intersection logic (procedural primitives)

Programs II – Shader Stages Continued

- **Miss** – Executed when ray misses the scene (e.g., skybox sampling)
- **Direct Callable** – Fast inlined function calls via SBT (e.g., shading models)
- **Continuation Callable** – May trigger new ray traces or shading logic
- PTX or CUBIN – Programs written in CUDA C++, compiled with `nvcc`
- OptiX SDK provides API and helpers for C/C++ pipeline integration

VARIABLES VS SBT

- Optix 6 and older
  - System for declaration and definition of variables, so they are correctly passed into and between different programs
  - Program reference via specific set of rules
    - Internal semantics maybe specified
    - Variable color can be connected to *Material*
    - Specific geometry instances can redefine the variable
- Optix 7 and newer
  - Ray payload only 32bit integers (can be a pointer)
  - Shaders/programs and data store in *Shader Binding Table*

Acceleration Structures

- OptiX uses RTX hardware to accelerate BVH traversal
- Two-level hierarchy:
  - **GAS** – Geometry Acceleration Structure (individual meshes / primitives)
  - **IAS** – Instance Acceleration Structure (transforms & references to GAS)
- Built using optixAccelBuild() on the GPU
- Traversal happens via optixTrace() at runtime

EXECUTION

- All informations and data passed to context instance
- Specify dimensions and execution parameters
- Ray generator is executed, once for each element(pixel)
- Results stored into output buffer

Examples

EXAMPLE 1 - NORMAL SHADER

- Most used program – **closest hit**

Normal shader

```
extern "C" __global__ void __closesthit__normal_shader() {
    float3 shading_normal = make_float3(
 __uint_as_float(optixGetAttribute_0()),
 __uint_as_float(optixGetAttribute_1()),
 __uint_as_float(optixGetAttribute_2())
    );

    float3 world_normal = normalize(
        optixTransformNormalFromObjectToWorldSpace(shading_normal));
    float3 color = 0.5f * world_normal + 0.5f;

    setPayload(color);
}
```

EXAMPLE 1 - VARIABLES

- The shading normal is passed via attributes from the intersection program

Intersection Attribute Passing

```
// In __intersection__ program
float3 shading_normal = calculate_normal();
optixReportIntersection(t_hit, 0,
    __float_as_uint(shading_normal.x),
    __float_as_uint(shading_normal.y),
    __float_as_uint(shading_normal.z));
```

- Result is written via payload or a user-defined payload struct

```
// Launch parameters or inline payload (example: RGB packed)
static __forceinline__ void setPayload(float3 color) {
    optixSetPayload_0(__float_as_uint(color.x));
    optixSetPayload_1(__float_as_uint(color.y));
    optixSetPayload_2(__float_as_uint(color.z));
}
```

Example 1 - Miss Program

- No intersection — handled by the **miss** program
- Returns background color from launch parameters

Miss program

```
extern "C" __global__ void __miss__radiance() {
    float3 bg_color = params.bg_color;
    optixSetPayload_0(__float_as_uint(bg_color.x));
    optixSetPayload_1(__float_as_uint(bg_color.y));
    optixSetPayload_2(__float_as_uint(bg_color.z));
}
```

EXAMPLE 1 - RAY GENERATION PROGRAM

```
extern "C" __global__ void __raygen__pinhole_camera() {
    const uint3 idx = optixGetLaunchIndex();
    const uint3 dim = optixGetLaunchDimensions();

    float2 d = (make_float2(idx.x, idx.y) / make_float2(dim.x, dim.y)) *
    float3 origin = params.eye;
    float3 direction = normalize(d.x * params.U + d.y * params.V + param

    uint32_t p0, p1, p2;
    optixTrace(
     // ...
    );

    float3 color = make_float3(__uint_as_float(p0),
                               __uint_as_float(p1),
                               __uint_as_float(p2));
    params.output_buffer[idx.y * dim.x + idx.x] = make_color(color);
}
```

EXAMPLE 1 - TRACE CALL

- Most important function: `optixTrace` (replaces `rtTrace`)
  - `params.handle` — root of scene hierarchy (OptixTraversableHandle)
  - Origin and direction — define the current ray
  - Payloads passed via registers
- The call invokes a miss or hit program and returns control
- Stack depth is managed explicitly via launch parameters like `maxTraceDepth`

EXAMPLE 1

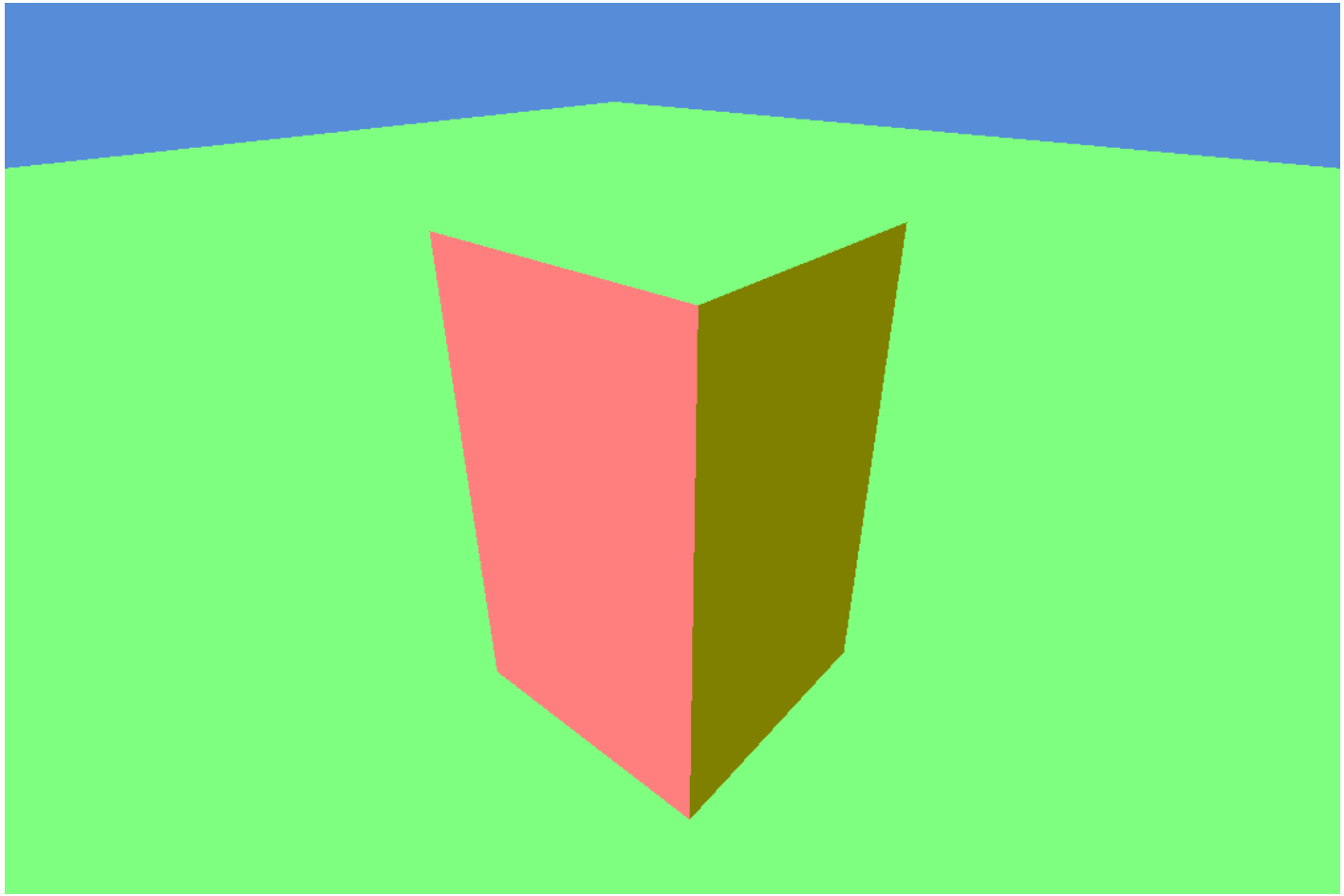


Figure: Normal shader example

EXAMPLE 2 - DIFFUSE SHADER

- Only *closest hit* program is modified
- Geometric and shading normals can differ (e.g., for bump mapping)
- Both normals are passed via attributes from the intersection program

Diffuse shader — part 1

```
extern "C" __global__ void __closesthit__diffuse() {
    float3 shading_normal = make_float3(
        __uint_as_float(optixGetAttribute_0()),
        __uint_as_float(optixGetAttribute_1()),
        __uint_as_float(optixGetAttribute_2()));
    float3 geometric_normal = make_float3(
        __uint_as_float(optixGetAttribute_3()),
        __uint_as_float(optixGetAttribute_4()),
        __uint_as_float(optixGetAttribute_5()));
    float3 world_geo_normal =
        normalize(optixTransformNormalFromObjectToWorldSpace(geometric_n
    float3 world_shade_normal =
        normalize(optixTransformNormalFromObjectToWorldSpace(shading_nor
```

EXAMPLE 2 - DIFFUSE SHADER (CONTINUED)

Diffuse shader — part 2

```
    float3 ffnormal = faceforward(world_shade_normal,
                                  -optixGetWorldRayDirection(),
                                  world_geo_normal);

    float3 hit_point = optixGetWorldRayOrigin() +
                       optixGetRayTmax() * optixGetWorldRayDirection();
    float3 color = params.Ka * params.ambient;

    for (int i = 0; i < params.num_lights; ++i) {
        BasicLight light = params.lights[i];
        float3 L = normalize(light.pos - hit_point);
        float nDl = dot(ffnormal, L);
        if (nDl > 0.f)
            color += params.Kd * nDl * light.color;
    }

    optixSetPayload_0(__float_as_uint(color.x));
    optixSetPayload_1(__float_as_uint(color.y));
    optixSetPayload_2(__float_as_uint(color.z));
}
```

EXAMPLE 2 - LIGHT SETUP

- Light info stored in a 1D array in launch parameters
- User-defined structure, allocated and uploaded by the host
- Diffuse reflectance (`Kd`) is part of the SBT or launch params

Light definition and access

```
struct BasicLight {
    float3 pos;
    float3 color;
    int    casts_shadow;
};

// In launch parameters
struct LaunchParams {
    ...
    BasicLight* lights;
    int         num_lights;
    float3      Ka, Kd, ambient;
    ...
};
extern "C" __constant__ LaunchParams params;
```

EXAMPLE 2

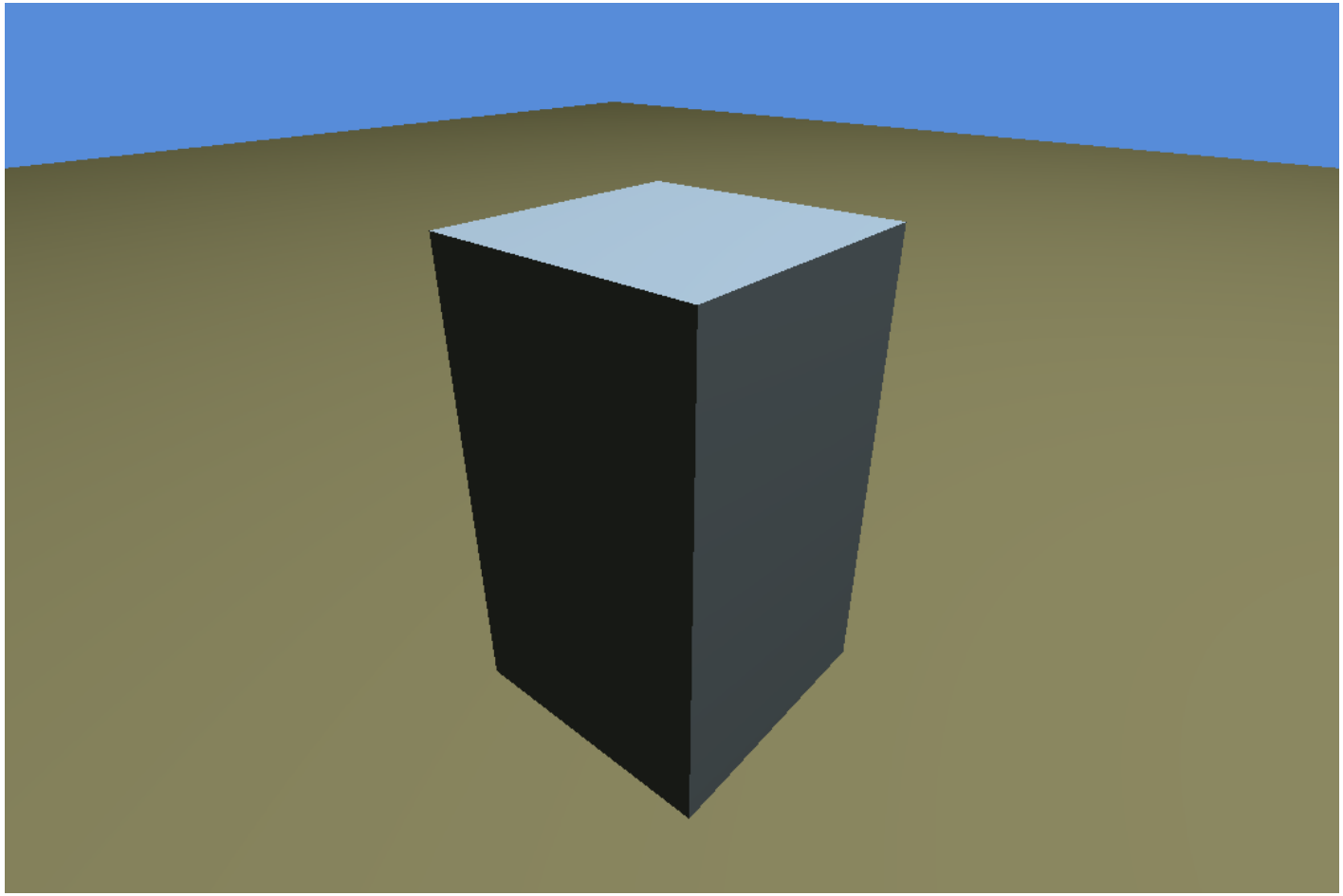


Figure: Diffuse shader

EXAMPLE 3 - PHONG LIGHT MODEL

- Adds specular highlights using Blinn-Phong model
- Uses a halfway vector between light and view direction

Phong shader

```
for (int i = 0; i < params.num_lights; ++i) {
    BasicLight light = params.lights[i];
    float3 L = normalize(light.pos - hit_point);
    float nDl = dot(ffnormal, L);

    if (nDl > 0.f) {
        float3 H = normalize(L - optixGetWorldRayDirection());
        float nDh = dot(ffnormal, H);

        float3 Lc = light.color;
        color += params.Kd * nDl * Lc;

        if (nDh > 0.f)
            color += params.Ks * pow(nDh, params.phong_exp) * Lc;
    }
}
```

EXAMPLE 3

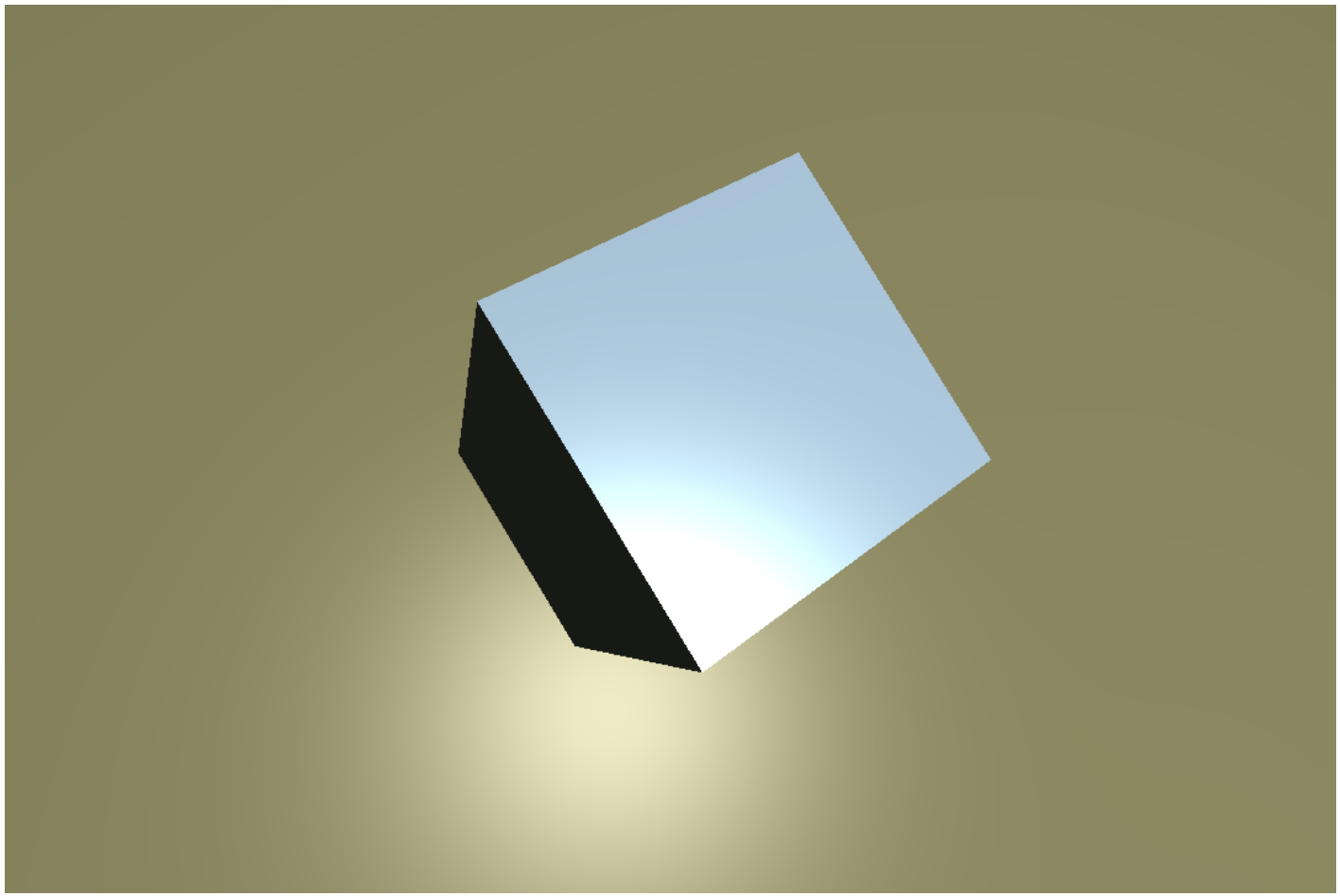


Figure: Phong shader

EXAMPLE 4 - SHADOWS

- Exact shadows for point light sources
- Shadow ray checks if geometry blocks light

Shadow ray dispatch

```
if (nDl > 0.f) {
    float3 L = normalize(light.pos - hit_point);
    float Ldist = length(light.pos - hit_point);
    uint32_t shadow = __float_as_uint(1.0f);  // visible by default
    optixTrace(
        params.handle,
        hit_point,
        L,
        1e-3f, Ldist - 1e-3f,  // avoid self-intersection
        0.0f,
        OptixVisibilityMask(255),
        OPTIX_RAY_FLAG_TERMINATE_ON_FIRST_HIT,
        SHADOW_RAY_TYPE, RAY_TYPE_COUNT, SHADOW_RAY_TYPE,
        shadow
    );
    float light_attenuation = __uint_as_float(shadow);
```

EXAMPLE 4 - SHADOWS CONTINUED

Shadowed lighting

```
    if (light_attenuation > 0.0f) {
        float3 Lc = light.color * light_attenuation;
        color += params.Kd * nDl * Lc;

        float3 H = normalize(L - optixGetWorldRayDirection());
        float nDh = dot(ffnormal, H);
        if (nDh > 0.f)
            color += params.Ks * pow(nDh, params.phong_exp) * Lc;
    }
}
```

EXAMPLE 4 - ANY-HIT SHADOW

- Shadow rays only need to know if something is hit
- **anyhit** terminates immediately upon any hit

Shadow any-hit program

```
extern "C" __global__ void __anyhit__shadow() {
    optixSetPayload_0(__float_as_uint(0.0f)); // shadowed
    optixTerminateRay();
}
```

EXAMPLE 4

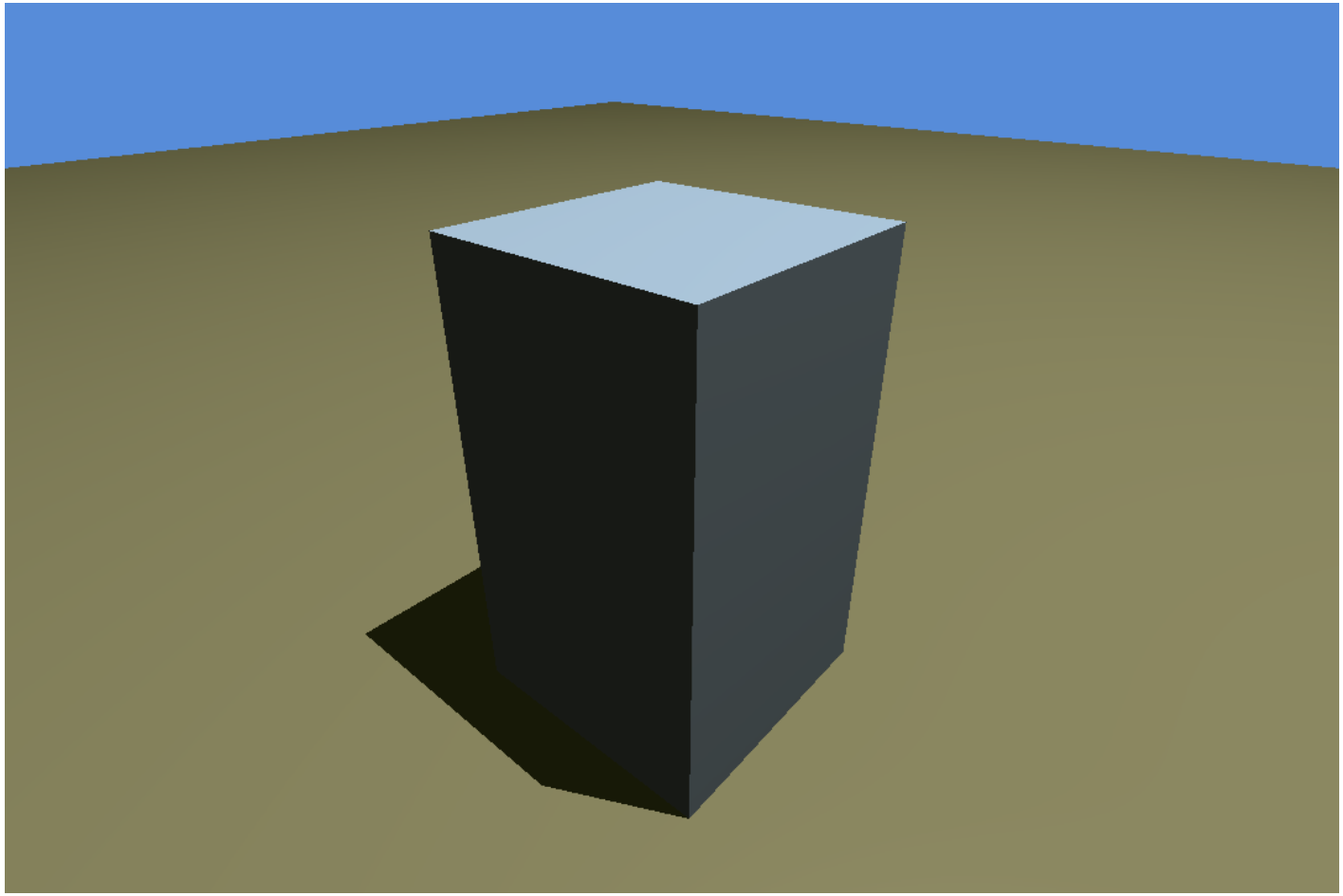


Figure: Exact shadow for point light

EXAMPLE 5 - REFLECTIONS

- Mirror reflection uses a secondary ray
- Color of reflected ray is combined with surface shading

Mirror reflection

```
float3 R = reflect(optixGetWorldRayDirection(), ffnormal);
uint32_t p0, p1, p2, depth = optixGetPayload_3() + 1;

optixTrace(
    params.handle,
    hit_point,
    R,
    1e-3f, 1e16f,
    0.0f,
    OptixVisibilityMask(255),
    OPTIX_RAY_FLAG_NONE,
    RADIANCE_RAY_TYPE, RAY_TYPE_COUNT, RADIANCE_RAY_TYPE,
    p0, p1, p2, depth
);

float3 reflected = make_float3(__uint_as_float(p0),
                               __uint_as_float(p1),
                                 uint as float(p2));
```

EXAMPLE 5 - REFLECTIONS WITH RECURSION LIMIT

- Use a per-ray depth counter
- Prevent stack overflows by bounding recursion

With recursion depth check

```
if (optixGetPayload_3() < params.max_depth) {
    float3 R = reflect(optixGetWorldRayDirection(), ffnormal);
    uint32_t p0, p1, p2;
    uint32_t next_depth = optixGetPayload_3() + 1;

    optixTrace(
        params.handle, hit_point, R,
        1e-3f, 1e16f,
        0.0f, OptixVisibilityMask(255), OPTIX_RAY_FLAG_NONE,
        RADIANCE_RAY_TYPE, RAY_TYPE_COUNT, RADIANCE_RAY_TYPE,
        p0, p1, p2, next_depth
    );
    float3 refl = make_float3(__uint_as_float(p0),
                              __uint_as_float(p1),
                              __uint_as_float(p2));
    color += params.reflectivity * refl;
}
```

EXAMPLE 5

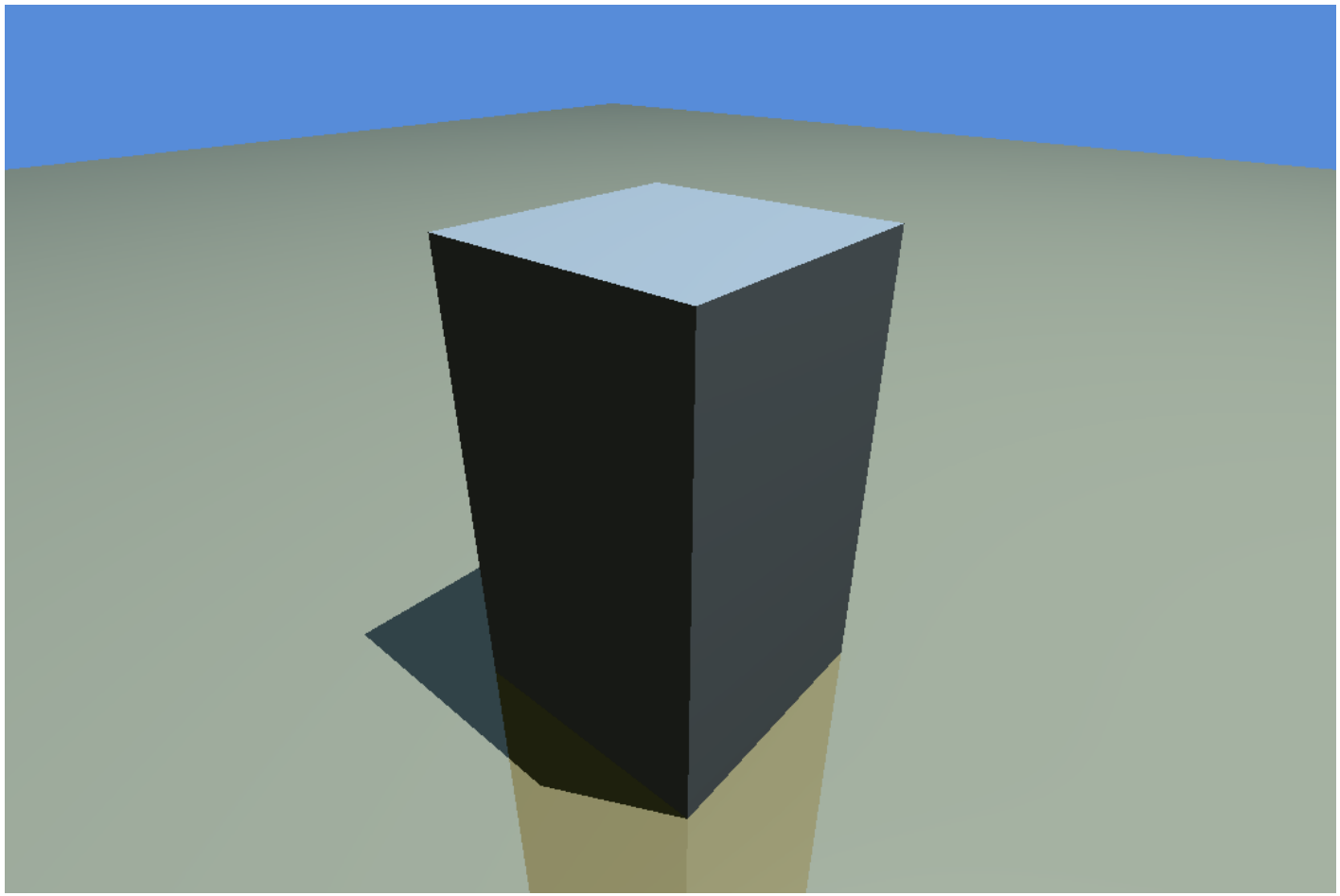


Figure: Mirror reflections

EXAMPLE 6 - ENVIRONMENT MAPPING

- Background color fetched from a 2D texture (e.g., lat-long environment map)
- Accessed in the **miss** program

Miss program with texture

```
extern "C" __global__ void __miss__envmap() {
    float3 dir = optixGetWorldRayDirection();
    float theta = atan2f(dir.x, dir.z);
    float phi   = M_PIf * 0.5f - acosf(dir.y);
    float u = (theta + M_PIf) * (0.5f * M_1_PIf);
    float v = 0.5f * (1.0f + sinf(phi));

    float4 tex = tex2D<float4>(params.envmap, u, v);
    float3 color = make_float3(tex);

    optixSetPayload_0(__float_as_uint(color.x));
    optixSetPayload_1(__float_as_uint(color.y));
    optixSetPayload_2(__float_as_uint(color.z));
}
```

EXAMPLE 6

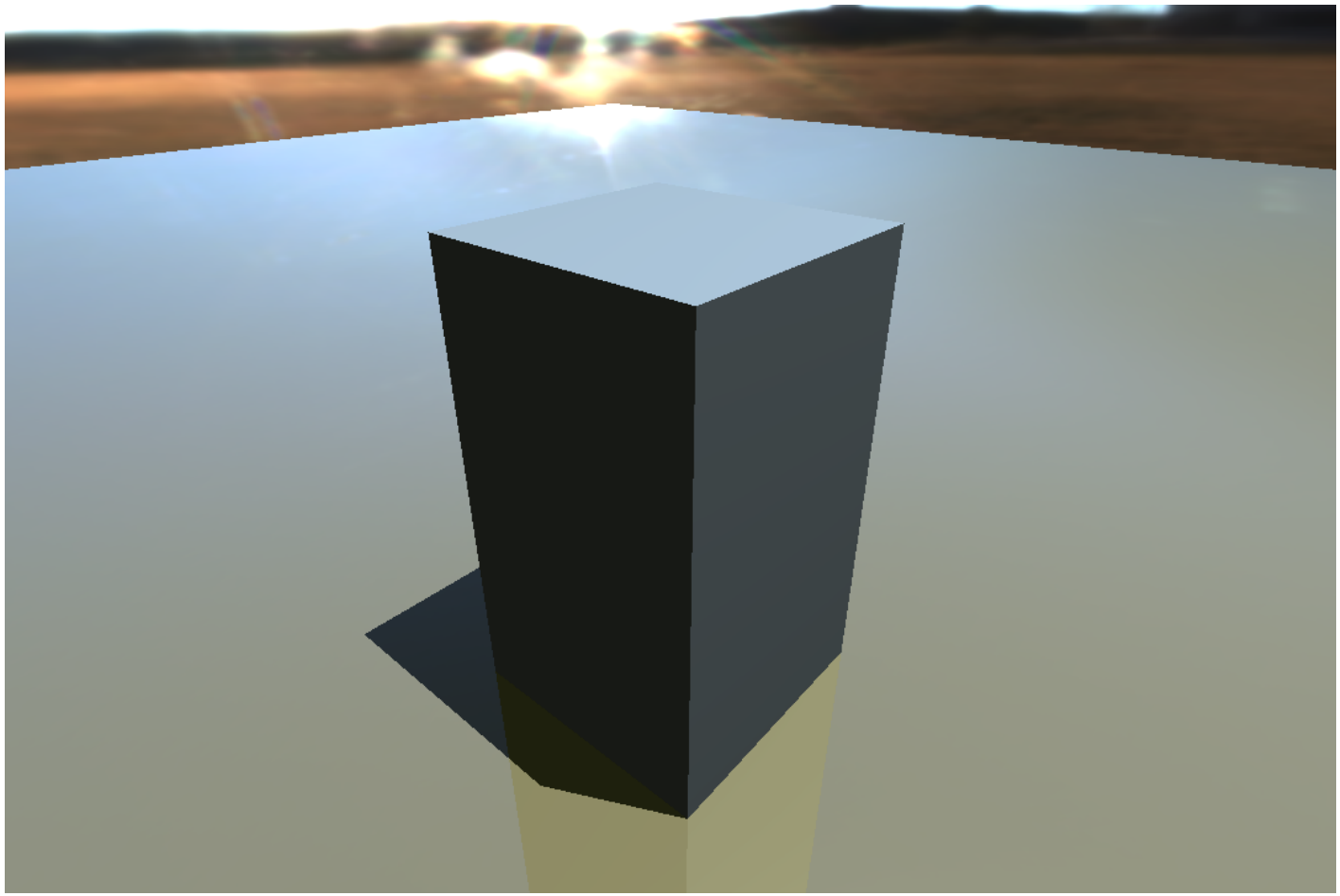


Figure: Environment mapping

EXAMPLE 7 - PROCEDURAL GEOMETRY

- Procedural primitives: e.g. convex hulls from plane equations
- Intersection program computes hit analytically

Convex hull intersection

```
extern "C" __global__ void __intersection__convex() {
    const float3 ray_origin = optixGetObjectRayOrigin();
    const float3 ray_dir = optixGetObjectRayDirection();

    int n = params.num_planes;
    float t0 = -FLT_MAX;
    float t1 =  FLT_MAX;
    float3 t0_normal = make_float3(0);
    float3 t1_normal = make_float3(0);

    for (int i = 0; i < n && t0 < t1; ++i) {
        float4 plane = params.planes[i];
        float3 n = make_float3(plane);
        float d = plane.w;
```

EXAMPLE 7 - CONVEX HULL CONTINUED

Convex hull intersection – continued

```
        float denom = dot(n, ray_dir);
        float t = -(d + dot(n, ray_origin)) / denom;
        if (denom < 0.f) {
            if (t > t0) { t0 = t; t0_normal = n; }
        } else {
            if (t < t1) { t1 = t; t1_normal = n; }
        }
    }
    if (t0 > t1) return;
    float3 normal;
    float t_hit;
    if (optixReportIntersection(t0, 0)) {
        normal = t0_normal;
        t_hit = t0;
    } else if (optixReportIntersection(t1, 0)) {
        normal = t1_normal;
        t_hit = t1;
    }
    optixSetAttribute_0(__float_as_uint(normal.x));
    optixSetAttribute_1(__float_as_uint(normal.y));
    optixSetAttribute_2(__float_as_uint(normal.z));
```

EXAMPLE 7 - AABB

- Procedural primitives require user-defined bounding boxes
- AABB data is passed from host as parameters

AABB bounds program

```
extern "C" __global__ void __bounds__convex(int primIdx, float result[6]
    result[0] = params.convex_min.x;
    result[1] = params.convex_min.y;
    result[2] = params.convex_min.z;
    result[3] = params.convex_max.x;
    result[4] = params.convex_max.y;
    result[5] = params.convex_max.z;
}
```

EXAMPLE 7

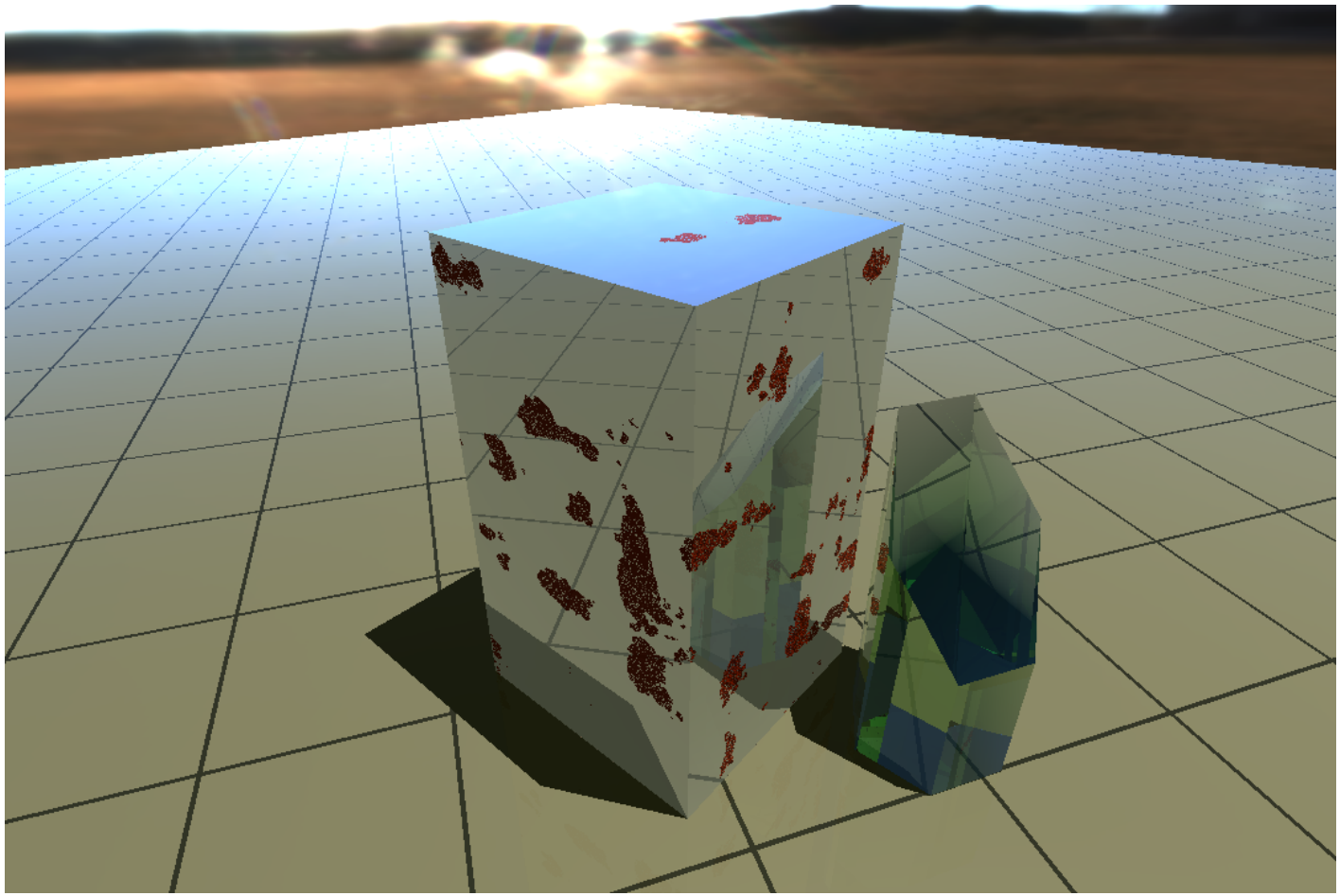


Figure: Glass convex hull

Denoising

NVIDIA DENOISING

- AI-powered denoiser trained on real path-traced images
- Removes Monte Carlo noise from ray-traced images
- Useful for real-time and interactive rendering

Benefits:

- High-quality output with fewer rays
- Works in real-time on RTX hardware
- Optional temporal support for stable animations

NVIDIA DENOISING

- AI-powered denoiser trained on real path-traced images
- Removes Monte Carlo noise from ray-traced images
- Useful for real-time and interactive rendering

Benefits:

- High-quality output with fewer rays
- Works in real-time on RTX hardware
- Optional temporal support for stable animations

Denoiser Inputs

- Denoiser can use up to 3 buffers:

Color Noisy path traced image
Albedo Diffuse color w/o lighting
Normal World-space surface normals

More inputs → better denoising quality!

Denoiser Inputs

- Denoiser can use up to 3 buffers:

Color	Noisy path traced image	
Albedo	Diffuse color w/o lighting	More inputs $\rightarrow$ better
Normal	World-space surface normals	

denoising quality!

OptiX Denoising Flow

```
// 1. Create denoiser
optixDenoiserCreate(context, OPTIX_DENOISER_MODEL_KIND_LDR, &denoiser);

// 2. Setup memory
optixDenoiserComputeMemoryResources(denoiser, ...);
cudaMalloc(&scratch, scratchSize);

// 3. Configure
optixDenoiserSetup(denoiser, stream, width, height, state, ...);

// 4. Fill guide and layer structures
OptixDenoiserGuideLayer guide = {...};
OptixDenoiserLayer layer = {
    .input = noisy_image,
    .output = denoised_image,
    ...
};

// 5. Run denoising
optixDenoiserInvoke(denoiser, stream, &params, state, ...);
```

Performance Tips

- Use normals & albedo for best quality
- Use shared scratch/state across frames (reuse!)
- Denoising latency: 1–5 ms at 1080p (RTX 30/40 series)
- Works on OptiX render outputs or CUDA buffers