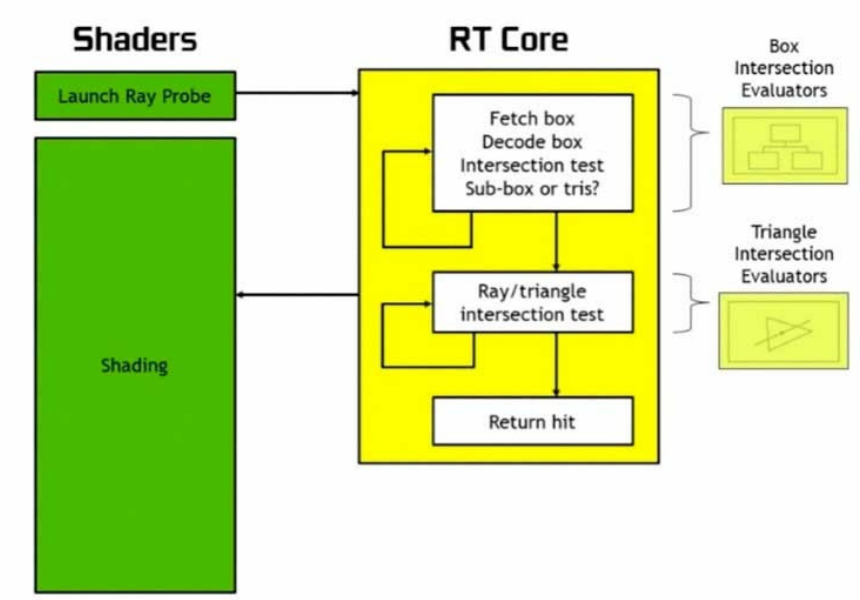
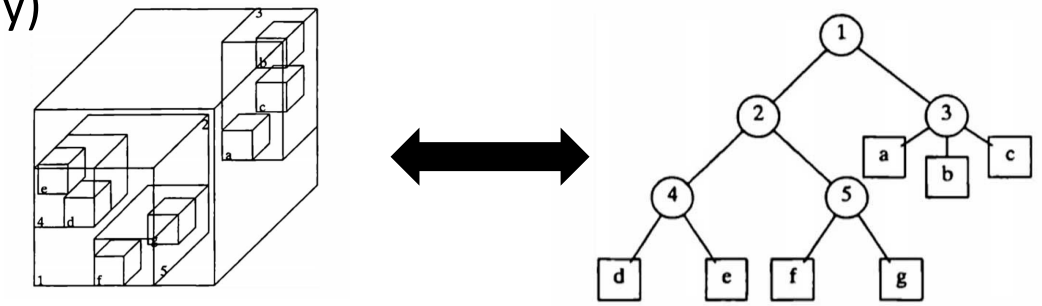
GPU Raytracing

Dieter Schmalstieg

GPU Hardware

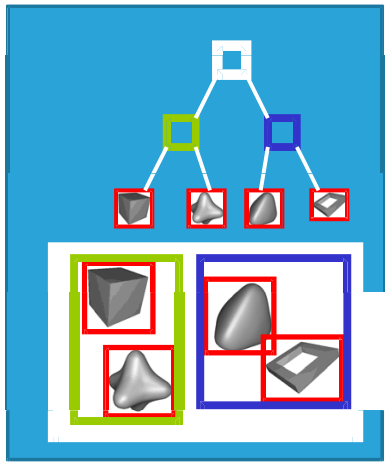
- Shader cores
  - General purpose code
- Fixed function units
  - Rasterizer
  - Texture sampler
  - Tensor core (matrix-matrix multiply)
  - …
- RT cores (**new**)
  - Intersection ray/box
  - Intersection ray/triangle





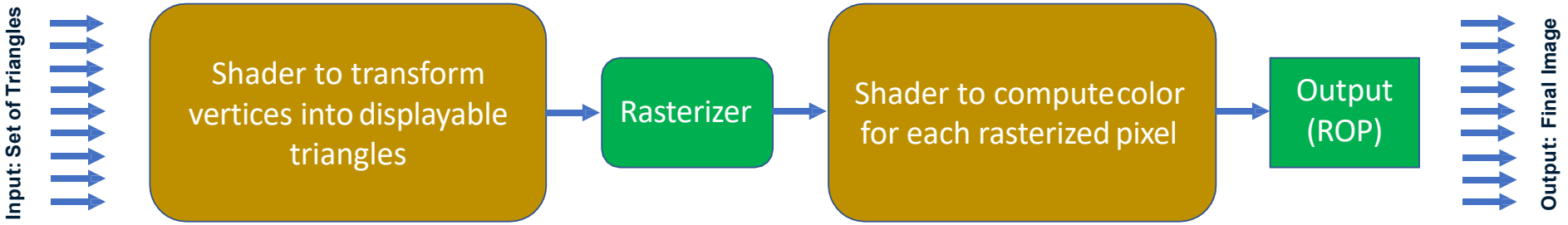
Bounding Volume Hierarchy

- Most important spatial data structure
- Common bounding volume types
  - Spheres
  - Axis-aligned bounding box (AABB)
  - Oriented bounding box (OBB)
- Encloses the bounded object
- Hierarchical data structure
  - Tree (binary or n-ary)
  - Leaves: store objects
  - Interior nodes: stores bounding volume enclosing all contained bounding volumes

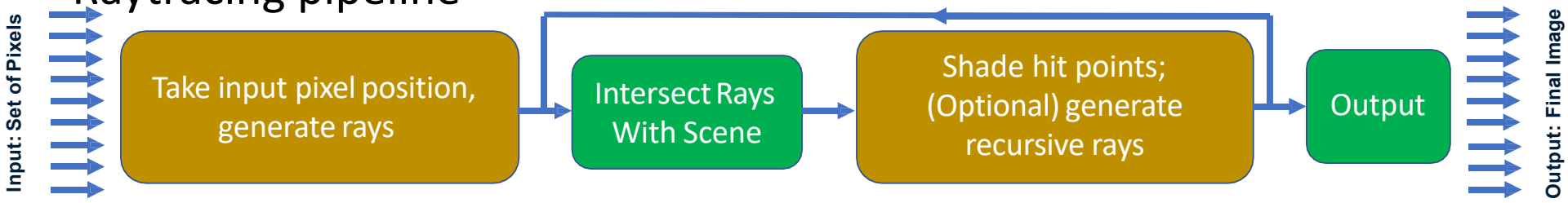


Rasterization vs Raytracing Pipeline

Rasterization pipeline

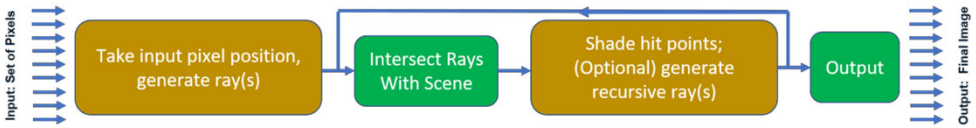


Raytracing pipeline



DirectX Ray Tracing Pipeline

- Pipeline is split into five new shaders:
  - **Ray generation shader** defines how to start ray tracing
  - **Intersection shader** define how rays intersect geometry
  - **Miss shader** define behavior when rays miss geometry
  - **Closest-hit shader** run once per ray (e.g., to shade final hit)
  - **Any-hit shader** run once per hit (e.g., to determine transparency)

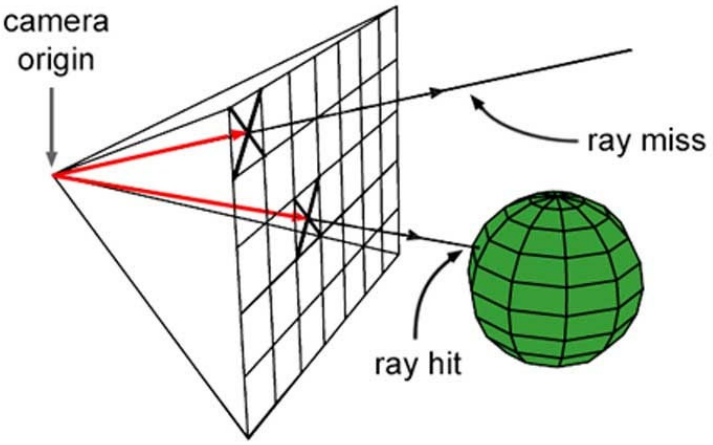


Runs once per algorithm (or per pass)

Defines geometric shapes, widely reusable

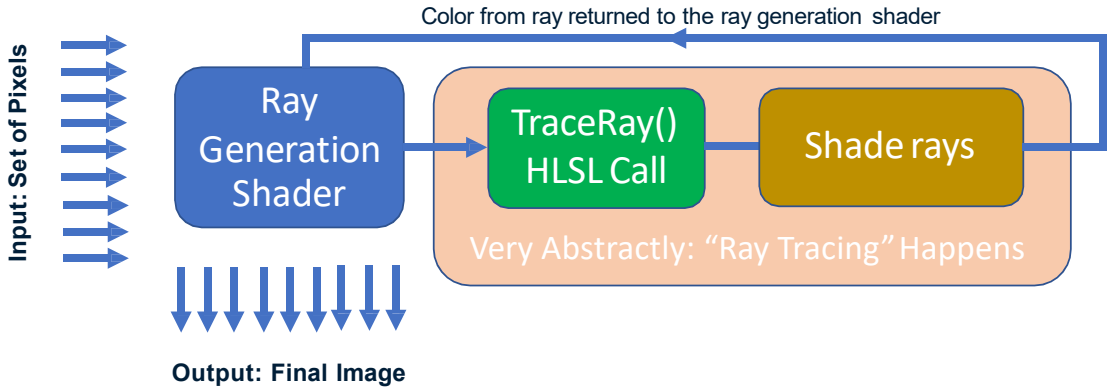
Defines behavior of rays

Different between shadow, primary, indirect rays



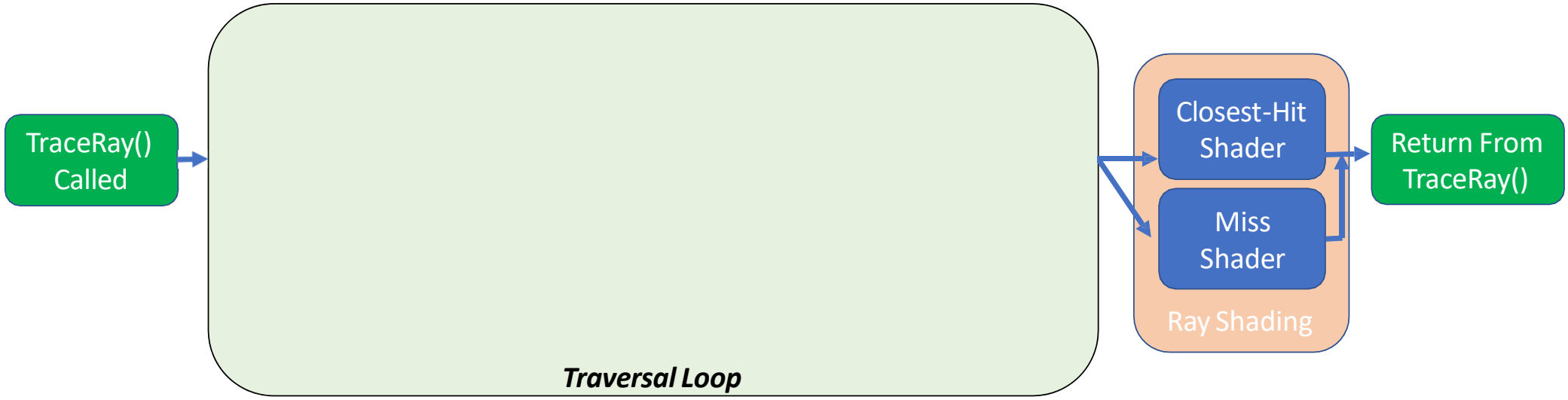
Ray Generation Shader

- Specify what rays to trace for each pixel
  - Launch rays by calling new HLSL TraceRay() intrinsic
  - Accumulate ray color into image after ray tracing finishes

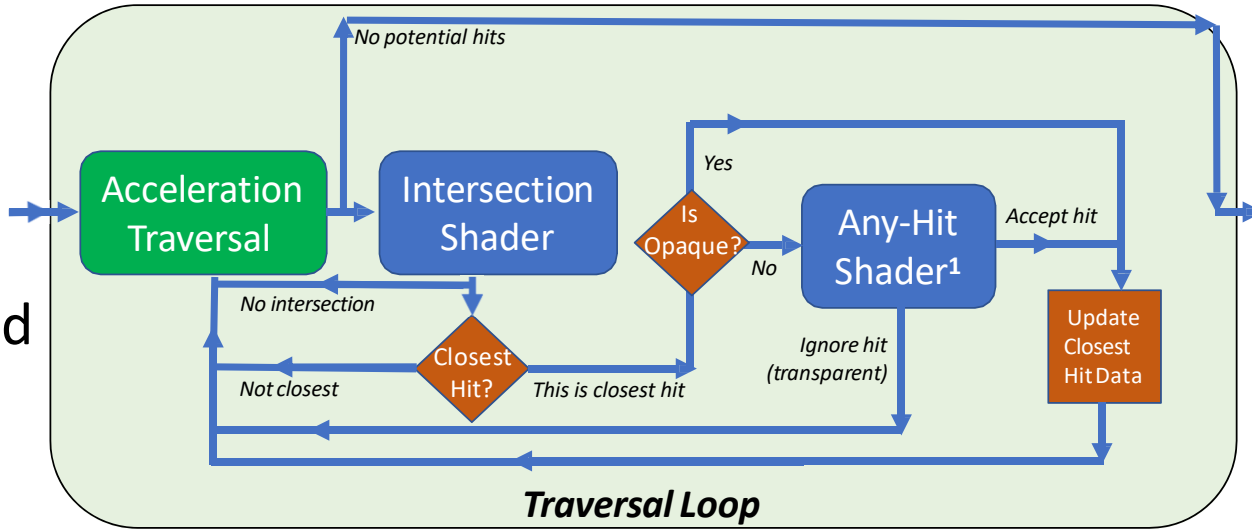


Tracing A Ray

- First, we traverse our scene to find what geometry ray hits
- When we find the closest hit, shade the point using **closest-hit shader**
- If ray misses all geometry (hits background), **miss shader** gets invoked



Traversal Loop



- If all geometry trivially ignored
  - Ray traversal ends
- For potential intersections
  - Call **intersection shader** is invoked
  - Triangles (special hardware), spheres, Bezier patches etc.
- If no intersection detected or not closest, continue traversal
- If hit
  - If transparent, run **any-hit shader** (call IgnoreHit() to continue)
  - Update closest hit data, continue to look for closer intersections

Starting a DXR Shader

- As any program, need an entry point where execution starts
  - Think `main()` in C/C++
- Shader entry points can be arbitrarily named
- Type specified by HLSL attribute: `[shader("shader-type")]`
  - Remember the **ray generation shader** is where ray tracing starts
- Starting other shader types look like this:
  - **`RayPayload`** is a user-defined (and arbitrarily named structure)
  - **`IntersectAttribs`** has data reported on hits (by intersection shader)

```
[shader("raygeneration")]
void PinholeCameraRayGen()
{ ... <Place code here> ... }

[shader("intersection")]
void PrimitiveIntersection ()
{ ... <Place code here> ... }

[shader("miss")]
void RayMiss(inout RayPayload data) // User-defined struct
{ ... <Place code here> ... }

[shader("anyhit")]
void RayAnyHit(inout RayPayload data,
               IntersectAttribs attribs)
{ ... <Place code here> ... }

[shader("closesthit")]
void RayClosestHit(inout RayPayload data,
                   IntersectAttribs attribs)
{ ... <Place code here> ... }
```

Ray Payload

- Ray payload is arbitrary user-defined struct
  - Contains intermediate data needed during ray tracing
  - Note: Keep ray payload as small as possible
    - Large payloads will reduce performance
    - Spills registers into memory
- A simple ray might look like this:
  - Sets color to blue when the ray misses
  - Sets color to red when the ray hits an object

```
struct SimpleRayPayload
{
    float3 rayColor;
};
```

```
[shader("miss")]
void RayMiss(inout SimpleRayPayload data)
{
    data.rayColor = float3( 0, 0, 1 ); // blue
}

[shader("closesthit")]
void RayClosestHit(inout SimpleRayPayload data,
                   IntersectAttribs attribs)
{
    data.rayColor = float3( 1, 0, 0 ); // red
}
```

Intersection Attributes

```
struct BuiltinIntersectionAttribs {
                            // Barycentric coordinates of hit in
     float2 barycentrics;  // the triangle are: (1-x-y, x, y)
}
struct PossibleSphereAttribs {
                          // Giving (theta,phi) of the hit on
     float2 thetaPhi;  // the sphere (thetaPhi.x, thetaPhi.y)
}

struct PossibleVolumeAttribs {
                    // Doing volumetric ray marching? Maybe
       float3 vox; // return voxel coord: (vox.x, vox.y, vox.z)
}
```

- Communications intersection information needed for shading
  - E.g., how do you look up textures for your primitive?
- Specific to each intersection type
  - One structure for triangles, one for spheres, one for Bezier patches
  - DirectX provides a built-in for the fixed function triangle intersector
  - Could imagine custom intersection attribute structures
- Limited attribute structure size: max 32 bytes

Data Needed for GPU Raytracing

Besides our shader, what data is needed on GPU to shoot rays?

- We need somewhere to write our output
- Where are we looking? – camera data
- Need to know about our scene geometry
- Also need information how to shade scene
  - Depends on material format
  - Depends on shading models

```
RWTexture<float4> outTex;
```

```
// HLSL "constant buffer", populated by host C++ code
cbuffer RayGenData {
    float3 wsCamPos; // World space camera position
    float3 wsCamU, wsCamV, wsCamW;//right/up/forward
};
```

```
// Our scene's ray acceleration structure, setup via the C++ DirectX API
RaytracingAccelerationStructure sceneAccelStruct;
```

Ray Generation Shader Code

```
[shader("raygeneration")]
void PinholeCamera() {
  uint2 curPixel      = DispatchRaysIndex().xy;
  uint2 totalPixels   = DispatchRaysDimensions().xy;
  float2 pixelCenter  = (curPixel+float2(0.5,0.5))/totalPixels;
  float2 ndc          = float2(2,-2)*pixelCenter+float2(-1,1);
  float3 pixelRayDir  = ndc.x*wsCamU+ndc.y*wsCamV+wsCamZ;
  RayDesc ray;
  ray.Origin          = wsCamPos;
  ray.Direction       = normalize( pixelRayDir );
  ray.TMin            = 0;
  ray.Tmax            = 999999999;
  SimpleRayPayload payload = { float3(0, 0, 0) };
  TraceRay(
    sceneAccelStruct,
    RAY_FLAG_NONE,
    0xFF,
    HIT_GROUP, NUM_HIT_GROUPS, MISS_SHADER,
    ray,
    payload );
  outTex[curPixel] = float4( payload.color, 1.0f );
}
```

What pixel are we currently computing?

How many rays, in total, are we generating?

Find pixel center in [0..1] x [0..1]

Compute normalized device coordinate (as in raster)

Convert NDC into pixel's ray direction (using camera)

Setup our ray

Setup our ray's payload

new intrinsic function in HLSL, can be called from ray generation, miss, and closest-hit

Our scene acceleration structure

Special traversal behavior for this ray? (Here: no)

mask 0xFF = test all geometry (could ignore some geometry)

Which intersection, any-hit, closest-hit, and miss shaders to use?

What ray are we shooting?

What is the ray payload? Stores intermediate, per-ray data

Write ray query result into our output texture

RayDesc is a new HLSL built-in type:

```
struct RayDesc {
    float3 Origin;    // Where the ray starts
    float  TMin;      // Min distance for a valid hit
    float3 Direction; // Direction the ray goes
    float  TMax;      // Max distance for a valid hit
};
```

Hit and Miss Shader Code

- Returns red if rays hit geometry

```
[shader("closesthit")]
void RayClosestHit(inout SimpleRayPayload data,
                   BuiltinIntersectionAttribs attribs)
{
    data.color = float3( 1, 0, 0 ); // red
}
```

- Returns blue on background

```
[shader("miss")]
void RayMiss(inout SimpleRayPayload data)
{
   data.color = float3( 0, 0, 1 ); // blue
}
```

- This is a complete DirectX Raytracing shader
  - Both intersection shader and any-hit shader are optional
  - Shoots rays from app-specified camera

Performance optimization

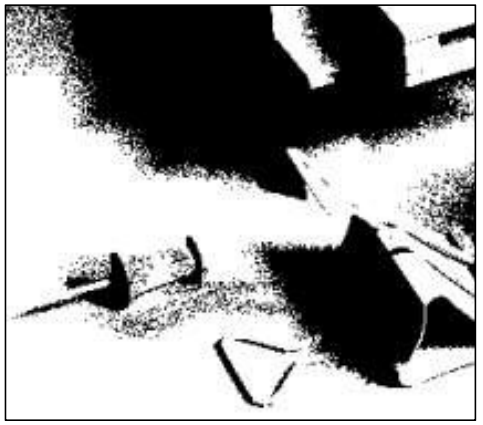
- Exploit ray coherence
  - Trace rays with similar direction at once
  - Coherent rays performing similar operations and memory access
  - Arbitrary reflection/refraction rays are bad
- Adaptive raytracing
  - Trace only where necessary
  - Shoot rays on demand
  - Filter/denoise the results

Hybrid Rasterization + Raytracing

- Pure raytracing is usually too expensive
- Game engines use a mixture
  - Main rendering done with rasterization
  - Special effects added with raytracing
- DirectX offers easy interoperability between raster and raytracing
  - Raytracing and rasterization shaders can share code and data types
  - E.g., HLSL material shaders - directly usable for a hybrid raytracing
  - Raytrace primary rays to fill G-buffer in deferred rendering
  - Raytracing of shadows, ambient occlusion, reflections

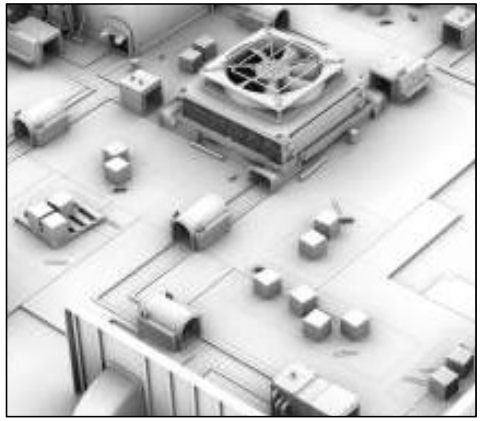
Raytraced Shadows

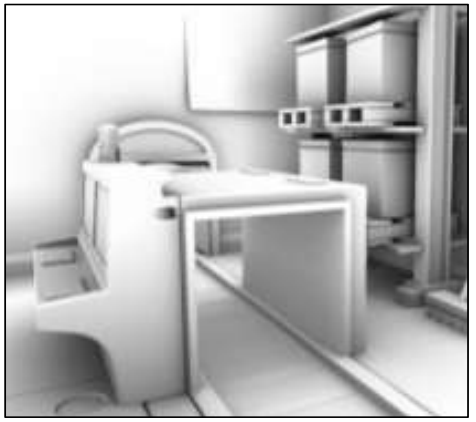
- Launch ray towards the light
  - If ray misses → not in shadow
  - Skip closest hit shader
  - Use miss shader
- Soft shadows
  - Choose random direction from cone
  - Cone width defines penumbra
  - Sample 1 ray per frame
  - Accumulate and filter over time (like TAA)



Raytraced Ambient Occlusion

- Randomly chose directions over hemisphere at a surface point
- Like shadows, miss shader tells us if a point is occluded over not
- Compute average





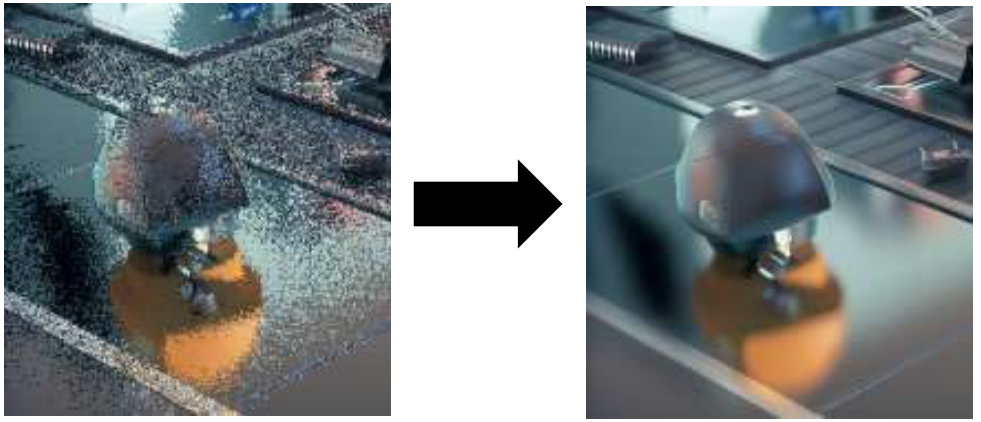
Raytraced Reflections

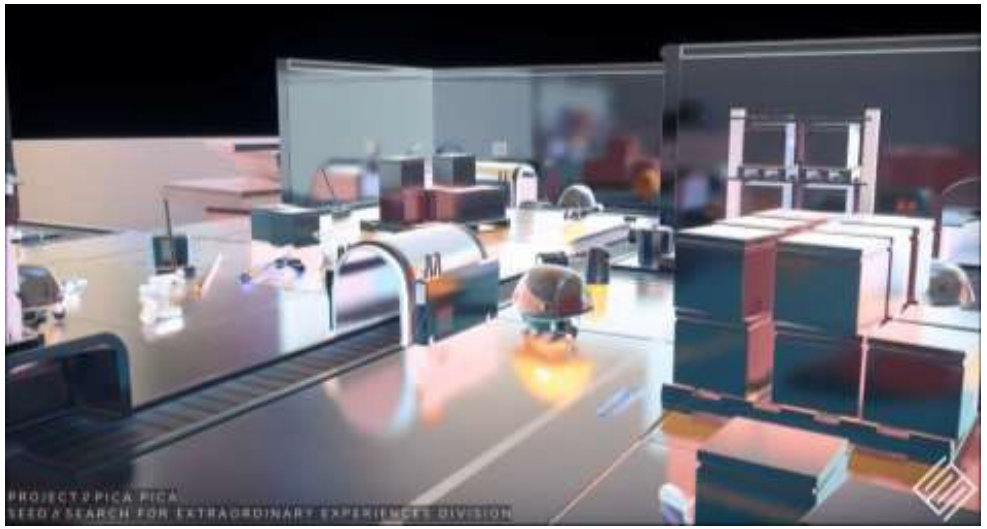
- Launch rays from G-Buffer
  - Trace at 50% resolution
  - 25% ray/pixel for reflection
  - 25% ray/pixel for reflected shadow
- Reconstruct at full resolution
- Combine reflection methods

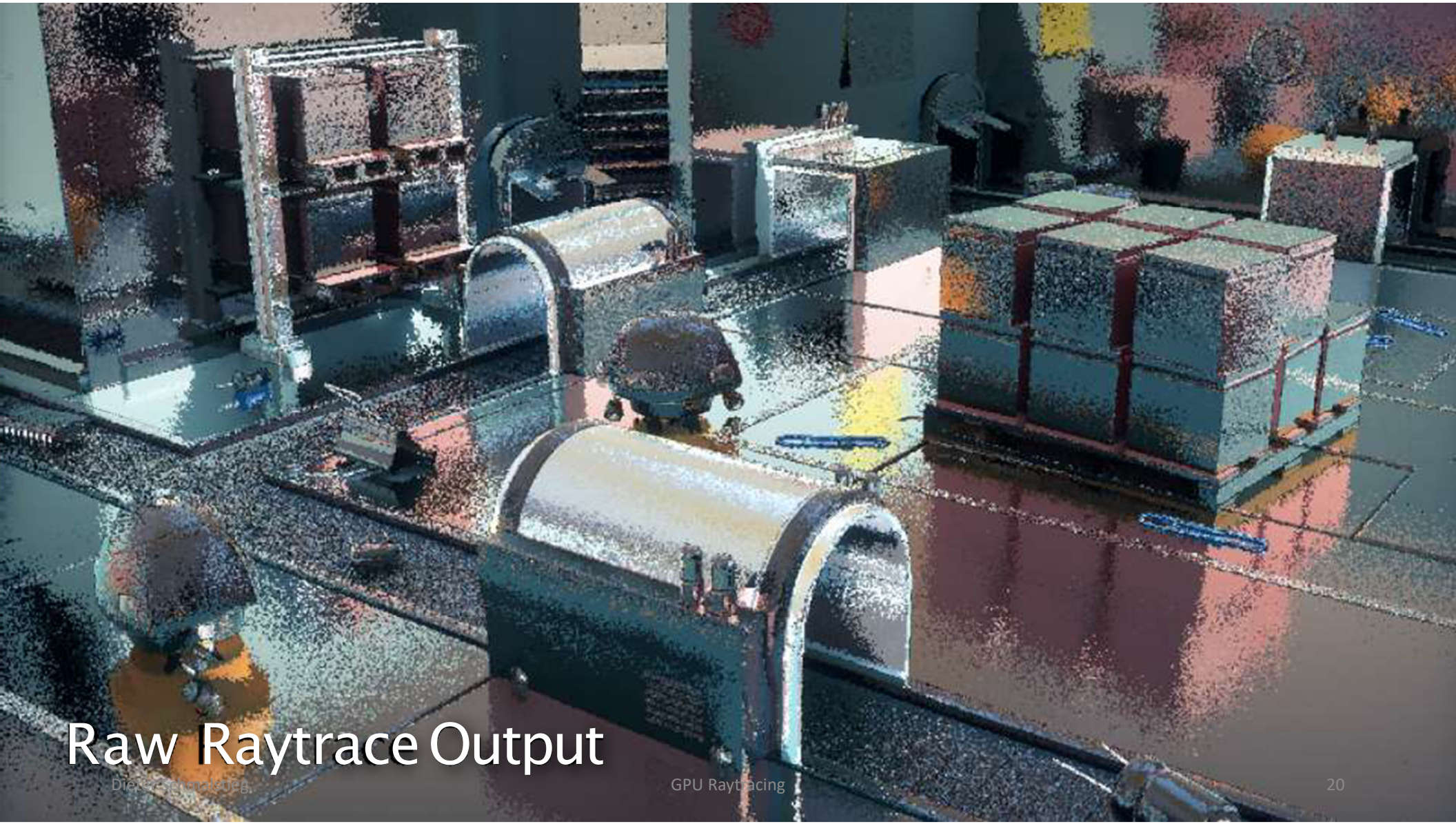
  Screen-space reflections (SSR)

  + Raytraced reflections (at SSR gaps)

  + Environment map (at remaining gaps)
- Both spatial + temporal filtering







Raw Raytrace Output

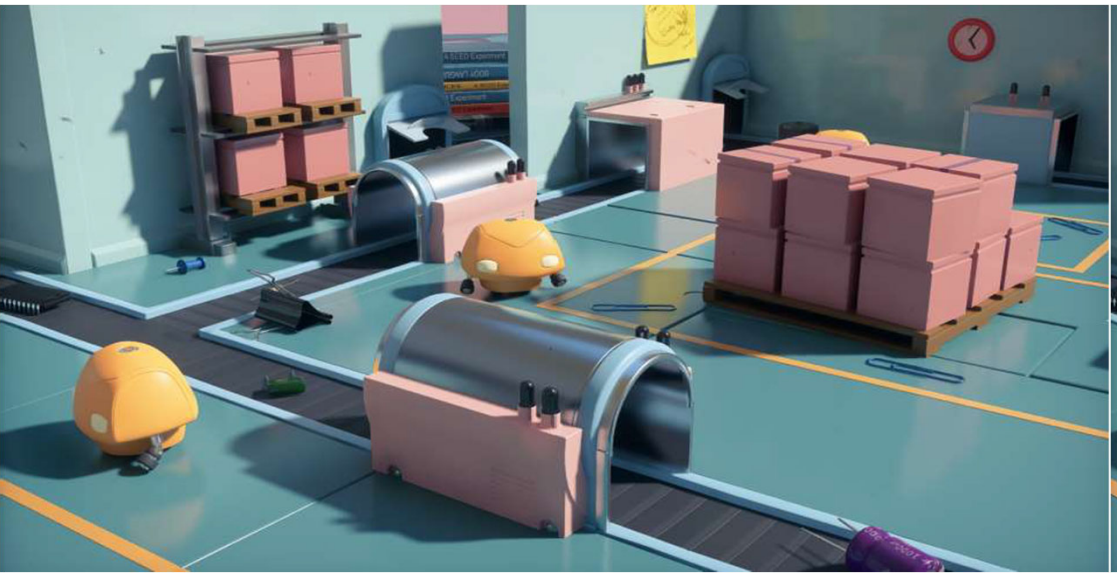
Spatial Reconstruction

+Temporal Accumulation

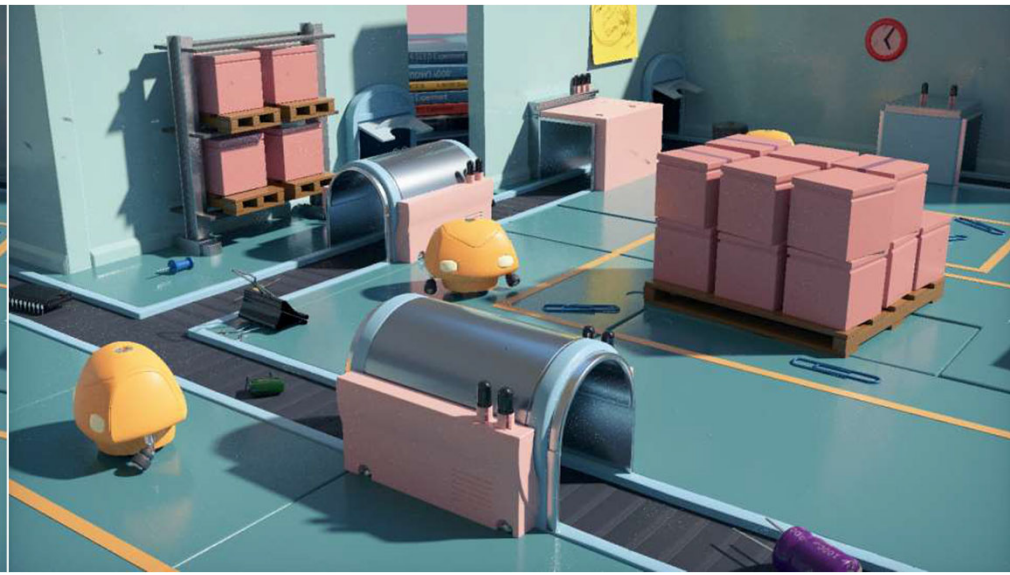
+Bilateral cleanup


+Temporal Anti-Aliasing

Comparison



Real-time raytracing



Pathtracing, ~15 seconds accumulation