“SHADOWS” OF THE TOMB RAIDER - A RAY TRACING DEEP DIVE

Holger Gruen (NVIDIA), Jon Story (NVIDIA), Michiel Roza (Nixxes)
03/19/2019
AGENDA

“Shadows” of the Tomb Raider

Why ray traced shadows?

DXR shaders (ray generation)

GameWorks spatial denoiser

DXR acceleration structure

Integration in render pipeline

Results

Future work
AGENDA

“Shadows” of the Tomb Raider

Shadow of the Tomb Raider

Why ray traced shadows?

DXR shaders (ray generation)

GameWorks spatial denoiser

DXR acceleration structure

Integration in render pipeline

Results

Future work
AGENDA

“Shadows” of the Tomb Raider

Why ray traced shadows?

DXR shaders (ray generation)

GameWorks spatial denoiser

DXR acceleration structure

Integration in render pipeline

Results

Future work
Shadow Mapped
Shadow Mapped
WHY RAY TRACED SHADOWS?

There’s so much that shadow mapping can’t do!

- Pixel perfect shadows
- Translucent shadows
- Point lights
  - Currently faked by using two spot lights
- Area lights
AGENDA

“Shadows” of the Tomb Raider

Why ray traced shadows?

DXR shaders (ray generation)

GameWorks spatial denoiser

DXR acceleration structure

Integration in render pipeline

Results

Future work
DXR SHADERS

Mini Agenda

• Noise / random number generation
• Ray generation
• Hit shaders
• Adaptive raytracing
• Translucency
• TAA and jittering
DXR SHADERS

DXR Mini intro

Shaders

Raygen()

Anyhit()

ClosestHit()

Acceleration Structures

TLAS

BLAS

BLAS
DXR SHADERS

Mini Agenda

• Noise / random number generation
• Ray generation
• Hit shaders
• Adaptive raytracing
• Translucency
• TAA and jittering
DXR SHADERS

Noise / random number generation

Ideally we want to trace many rays to find out how much of the light source a point can see.
• For great performance we want to shoot only 1 ray per pixel

• So instead of one pixel shooting many rays, a neighborhood of pixels samples ‘enough’ random positions on the light source

DXR SHADERS
Noise / random number generation
For great performance we want to shoot only 1 ray per pixel.

So instead of one pixel shooting many rays, a neighborhood of pixels samples ‘enough’ random positions on the light source.
DXR SHADERS
Noise / random number generation

• For great performance we want to shoot only 1 ray per pixel

• So instead of one pixel shooting many rays, a neighborhood of pixels samples ‘enough’ random positions on the light source
For great performance we want to shoot only 1 ray per pixel

So instead of one pixel shooting many rays, a neighborhood of pixels samples ‘enough’ random positions on the light source
DXR SHADERS

Noise / random number generation

• For great performance we want to shoot only 1 ray per pixel

• So instead of one pixel shooting many rays, a neighborhood of pixels samples ‘enough’ random positions on the light source
DXR SHADERS

Noise / random number generation

• For great performance we want to shoot only 1 ray per pixel

• So instead of one pixel shooting many rays, a neighborhood of pixels samples ‘enough’ **random** positions on the light source
DXR SHADERS
Noise / random number generation

• For great performance we want to shoot only 1 ray per pixel

• So instead of one pixel shooting many rays, a neighborhood of pixels samples ‘enough’ random positions on the light source
DXR SHADERS

Noise / random number generation

• For great performance we want to shoot only 1 ray per pixel
• So instead of one pixel shooting many rays, a neighborhood of pixels samples ‘enough’ random positions on the light source
DXR SHADERS
Noise / random number generation

• For great performance we want to shoot only 1 ray per pixel

• So instead of one pixel shooting many rays, a neighborhood of pixels samples ‘enough’ random positions on the light source
DXR SHADERS
Noise / random number generation

• Random positions on the light are based on generating pseudo random numbers
  

• The trick is to choose the right random seed for the generator

• We use a seed that is based on the 2D position of the pixel

  Seed( ( pixel_2d_pos ) % TILE_SIZE_2D );
Noisy shadows
DXR SHADERS

Noise / random number generation
DXR SHADERS

Mini Agenda

• Noise / random number generation
• Ray generation
• Hit shaders
• Adaptive raytracing
• Translucency
• TAA and jittering
We use specialized raygen shaders for each light type for optimal performance.

- Directional light source with an angular extent
- Area Cone light
- Point light with a spherical area
- Rectangular area cone light
DXR SHADEERS
Ray generation

All light types

Move some small distance along the normal to prevent self-shadowing!
DXR SHADERS

Ray generation

All light types

No rays for pixels that face away from the current light!
Directional lights

DXR SHADERS
Ray generation

Angular extent of the sun

z-buffer

transform to WS

WS Pixels
DXR SHADERS

Ray generation

Spot lights

Rays only get generated for pixels:
• Inside the cone of the light
• Within reach of the light

z-buffer
Point lights

Rays only get generated for pixels:
  - Within reach of the light
**DXR SHADERS**

**Ray generation**

Rectangular lights

Rays only get generated for pixels:
- Inside the cone of the light
- Within reach of the light
DXR SHADERS

Mini Agenda

• Noise / random number generation
• Ray generation
• Hit shaders
• Adaptive raytracing
• Translucency
• TAA and jittering
DXR SHADERS

Hit shaders

• We don‘t use the closest hit along the ray

• Instead we use the first opaque one that gets reported to anyhit()
  • Translucency for DXR ultra quality is an exception

• We fetch texture coords for alpha tested prims to carry out the alpha test
DXR SHADERS
Hit Shaders

• Opaque Geometry

```c
void OpaqueClosestHit(...) {
    payload.hitT = RayTCurrent();
    payload.visibility = 0.0f;
}
```

```c
void OpaqueAnyHit(...) {
    AcceptHitAndEndSearch();
}
```

• Alpha-tested Geometry

```
void AlphaClosestHit(...) {
    payload.hitT = RayTCurrent();
}
```

```
void AlphaAnyHit(...) {
    float alpha = GetHitAlpha(bary);
    if( alpha < g_fAlphaThreshold )
        IgnoreHit();
    else {
        payload.visibility = 0.0f;
        AcceptHitAndEndSearch();
    }
}
```
DXR SHADERS

Mini Agenda

- Noise / random number generation
- Ray generation
- Hit shaders
- Adaptive raytracing
- Translucency
- TAA and jittering
DXR SHADERS
Adaptive raytracing

• Adaptive raytracing is only used for ultra DXR quality

• In this mode we cast more than 1 ray for some pixels

• Let’s dive into the details ...
Phase 1 - Cast 1 ray per pixel

Phase 2 - Cast up to 2 more rays, ‘where necessary’

**Visibility:**
- Black => can’t see the light
- White => can see the light

**hitT:**
- Distance to blocker along ray
- White means ‘infinity’
- Dark means close-by blocker

**DXR SHADERS**
Adaptive raytracing
DXR SHADERS
Adaptive raytracing

Phase 1 - Cast 1 ray per pixel
Phase 2 - Cast up to 2 more rays ‘where necessary’
DXR SHADERS
Adaptive raytracing

Phase 1 - Cast 1 ray per pixel
Phase 2 - Cast up to 2 more rays ‘where necessary’

Visibility
hitT
Neighborhood of pixel

No additional rays
DXR SHADERS
Adaptive raytracing

Phase 1 - Cast 1 ray per pixel
Phase 2 - Cast up to 2 more rays 'where necessary'

Visibility

hitT

Neighborhood of pixel

Visibility = (Visibility0 + Visibility1) / 2
Phase 1 - Cast 1 ray per pixel

Phase 2 - Cast up to 2 more rays 'where necessary'

Visibility = \frac{(Visibility_0 + Visibility_1 + Visibility_2)}{3}
DXR SHADERS

Mini Agenda

- Noise / random number generation
- Ray generation
- Hit shaders
- Adaptive raytracing
- Translucency
- TAA and jittering
DXR SHADERS
Translucency

• Ultra DXR quality also features translucent shadows
• We support up to 3 layers of translucency
  • Mainly to keep performance at acceptable levels
• Translucency should be straightforward with DXR right?
  • Want to keep on using anyhit() instead of iterated closesthit()
  • Let’s look at the details ...
DXR SHADERS

Translucency

WS Pixel

anyhit() order non-deterministic
DXR SHADERS

Translucency

anyhit() order non-deterministic

WS Pixel
Subtraction is order independent:

- Let each layer subtract 1/3 of the light
- Pixel in full shadow after 3 order independent hits
void TranslucentAnyHit(...) {
    float alpha = GetHitAlpha(bary, PrimID);

    if( alpha >= g_fAlphaThreshold )
        payload.visibility -= ( 1.0f / 3.0f );

    if( payload.visibility < 0.01f ) {
        payload.visibility = 0.0f;
        AcceptHitAndEndSearch();
    }
    else
        IgnoreHit();
}
Opaque raytraced shadows
Translucent raytraced shadows
DXR SHADERS

Mini Agenda

- Noise / random number generation
- Ray generation
- Hit shaders
- Adaptive raytracing
- Translucency
- TAA and jittering
DXR SHADERS

TAAS + Jittering

• Like many games SotTR uses jittered TAA

• Each frame adds a ‘random’ subpixel offset to all geometry

• Surprisingly this creates problems with flickering shadows!
DXR SHADERS
TAA + Jittering

a 2x2 pixel grid

• The red dots are the pixel centers
• This is where rasterized geometry is sampled
DXR SHADERS
TAA + Jittering

Rasterizing a 3D quadrangle

The intermediate positions and the grid are shown to help understand how 3D positions change across the quad.
Jitter somewhat ...

DXR SHADERS

TAAS + Jittering
**DXR SHADERS**

**TAO + Jittering**

- Jittering changes the WS position that is sampled at pixel centers
  - It also changes the depth values at the pixel centers
- Jittering changes the reconstructed world space positions

Shadow ray origins jitter as well
Jittered ray positions are not problematic in general, but:

- We typically shoot only one ray per pixel
  - Which is equivalent to ‘point sampling‘ of the visibility signal

- Large areas of flat ground are problematic
  - Vertical jittering leads to large differences in WS positions
  - Also visible with shadow maps but less because of SM filtering

**DXR SHADERS**

**TAAS + Jittering**
DXR SHADERS
TAA + Jittering

Solutions:

1. Currently we render an extra depth pass without jittering
   • Use non-jittered depth to reconstruct WS ray origins

2. Future: Render $ddx/ddy(1/z_{buffer\_depth})$ with depth pass
   • Reconstruct non-jittered depth
   • Use non-jittered depth to construct WS ray origins
   • $1/z_{buffer\_depth}$ is linear in screen space
AGENDA

“Shadows” of the Tomb Raider

Why ray traced shadows?

DXR shaders (ray generation)

GameWorks spatial denoiser

DXR acceleration structure

Integration in render pipeline

Results

Future work
INPUT / OUTPUT

Visibility
HitT
Normals
Depth

Light Desc Params

GameWorks Spatial Denoiser
Edward Liu & Jon Story
ISOTROPIC KERNEL
ANISOTROPIC KERNEL
OVERLAPPING PENUMBRA #1
OVERLAPPING PENUMBRA #2
BLEEDING ARTIFACTS
CUSTOMIZED BOUNDARY DETECTION
COULD WE DO LESS WORK?
PENUMBRA MASK
IMPORTANT FEATURES

- Half resolution denoising
  - Drastically improves performance
  - SOTTR uses this mode for ALL light types
- MSAA input Depth & Normal buffers supported
  - Still only requires single sample Visibility & HitT buffers
  - Produces MSAA shadow mask
- Sub-viewport supported for local light sources
  - Just need to figure out screen area affected
“Shadows” of the Tomb Raider

AGENDA

Shadow of the Tomb Raider
Why ray traced shadows?
DXR shaders (ray generation)
GameWorks spatial denoiser
DXR acceleration structure
Integration in render pipeline
Results
Future work
BLAS
“a mesh”

- Vertex and index buffers for each geometry
- Straightforward for static geometry

```c
struct D3D12_RAYTRACING_GEOMETRY_TRIANGLES_DESC
{
    DXGI_FORMAT IndexFormat;
    DXGI_FORMAT VertexFormat;
    UINT IndexCount;
    UINT VertexCount;
    D3D12_GPU_VIRTUAL_ADDRESS IndexBuffer;
    D3D12_GPU_VIRTUAL_ADDRESS VertexBuffer;
}
```
**BLAS**

What about skinned objects and vertex animations?

- Each vertex needs to be fully transformed!
- Foundation Engine uses shader graphs
- Added a shader permutation in VS template for exporting a transformed vertex buffer
- Run a pass for all dynamic objects before building

```cpp
#if ExportVertexBuffer
    RWStructuredBuffer<float3> OutVertexBuffer;
#endif

VertexOutput main(
    in VertexInput vi,
    uint vertID : SV_VertexID)
{
    VertexOutput vo;
    %ShaderGraph%

#if ExportVertexBuffer
    OutVertexBuffer[vertID] = vo.OutPosition;
#endif

    return vo;
}
```
Skinning gone wrong: Inner demon 😊
BLAS
Lara’s hair

- PureHair, an evolution of TressFX
  - Simulates control points
  - Renders strands of hair as camera facing quads

- Everything needs to be actual geometry in the AS
  - Make the simplest cylinder possible for every strand
BLAS
Rebuild/refit strategy

- Two modes of updating dynamic BLASes in DXR:
  - Rebuild, essentially “replacing” the old one (~100M tris/sec)
  - Refit, for “small” model changes (~1000M tris/sec, 10x as fast!)
    - Catch: ray trace performance might degrade!
    - Top refitting throughput only for large enough workloads

- We chose to always refit BLAS unless # vertices change
TLAS
“A scene”

- Static BLASes can be instanced
- Always rebuilding TLAS seems to be fast enough (<1ms)

```c
struct D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS {
    UINT NumDescs;
    D3D12_GPU_VIRTUAL_ADDRESS InstanceDescs;
}
```
ACCELERATION STRUCTURE

About LODs of meshes

- Every LOD level is stored in a separate BLAS
- Using LOD 0 for everything caused self-shadowing artifacts!
- Just use the same LOD we use for rendering
- What about LOD fading?
  - Use “most visible” LOD
AGENDA

“Shadows” of the Tomb Raider

Why ray traced shadows?

DXR shaders (ray generation)

GameWorks spatial denoiser

DXR acceleration structure

Integration in render pipeline

Results

Future work
RENDER PIPELINE
Forward+ renderer

- Depth pass
- Shadow map pass
- Shadow resolve
- Forward opaque pass
RENDER PIPELINE

Now with ray traced shadows!

Vertex transform → Build AS → Depth pass Jittered/Non jittered → Shadow map pass → (Ray traced) Shadow resolve → Forward opaque pass
RENDER PIPELINE

Now with ray tracing!

- Vertex transform
- Build AS
- Depth pass Jittered/Non jittered
- Shadow map pass
- (Ray traced) Shadow resolve
- Forward opaque pass
Can run async with depth and shadow map passes! 😊
RENDER PIPELINE
Async compute

Vertex transform

Depth pass
Jittered/Non jittered

Shadow map pass

(Ray traced) Shadow resolve

Forward opaque pass

Build AS

0.5ms

2ms

5ms

3ms

4ms completely hidden!
WHY DO WE STILL NEED SHADOW MAPPING?

Translucent rendering

- Translucent rendering has no depth write
- Can’t use shadow resolve pass!
- We cannot shoot rays from pixel shaders
WHY DO WE STILL NEED SHADOW MAPPING?

Performance!

- Updating entire scene full of dynamic objects costs up to 20ms of BLAS refits 😞
- AS culling using existing shadow map culling
WHY DO WE STILL NEED SHADOW MAPPING?

Performance!

- For directional lights:
  - Replace only nearest cascades with ray traced shadows
- For local lights:
  - Distance based fade to shadow map
- How do we choose these distances?
ARTIST TOOLS
Let lighting artists decide!
AGENDA

“Shadows” of the Tomb Raider

Why ray traced shadows?

DXR shaders (ray generation)

GameWorks spatial denoiser

DXR acceleration structure

Integration in render pipeline

Results

Future work
No point light shadows
Raytraced point light shadows
Shadow mapped area light shadows
Raytraced area light shadows
Shadow mapped sun light shadows
Raytraced sun light shadows
AGENDA

Shadow of the Tomb Raider

Why ray traced shadows?

DXR shaders (ray generation)

GameWorks spatial denoiser

DXR acceleration structure

Integration in render pipeline

Results

Future work
FUTURE WORK

- Reconstruct non-jittered depth
- Ray traced shadows on translucent geometry
- Tessellation
- Content authoring with ray tracing in mind
- Use vertex transform pass for rasterization as well
- GI / Reflections / AO / ... ?
QUESTIONS

Holger Gruen (hgruen@nvidia.com)
Jon Story (jons@nvidia.com)
Michiel Roza (mroza@nixxes.com)  @Paramike86