Optimized pixel-projected reflections for planar reflectors

a.k.a. Pixel-projected reflections

Adam Cichocki

@multisampler
Screen space reflections: Overview

• Widely adopted technique
• Algorithm based on ray-marching
• Reflection generated from:
  - color buffer
  - depth buffer
• Result usually contains multiple areas with missing data
• Often performed at half resolution for efficiency [Wronski14]
  (bandwidth heavy ray-marching)
• [McGuire14]
Pixel-projected reflections: Overview

• Constrained form of SSR
• Offers high performance and quality
• Can be selectively used where applicable [Stachowiak15]
  - through compute shader indirect dispatch
• Algorithm based on data scattering
• Reflection generated from:
  - color buffer
  - depth buffer
  - analytical reflective areas
Pixel-projected reflections: Concept

- Reverse the reflection tracing
- Instead of performing ray-marching … … calculate exactly where pixels are reflected
- Approximate reflective areas of the scene with flat analytical shapes
- Reflective areas will be provided to shader through constants
- We’ll just use rectangles to approximate reflective areas
  - this is the most practical shape
  - other shapes also possible
Limitations

• Non-glossy reflections
• Reflections only on planar surfaces
• Normalmaps not supported
  - can be approximated, see "Bonus slides"
• Requires reflector shapes that approximate reflective areas
  - shapes are coplanar with reflective areas
  - enclose reflective areas
  - defined by artists on the scene or in prefabs
Simplified algorithm in 2D

• Just to grasp the pixel-projection concept
• We’ll switch to 3D afterward
Example 2D scene

- Single object
- Puddle on the floor
- Puddle approximated with line segment
Projection pass

• First pass of the algorithm is the „projection pass”
• For every pixel calculate where it is reflected on the screen
  - single pixel in the example
Projection pass

- Mirror pixel position against puddle’s plane
Projection pass

• Test if pixel’s reflection is visible in the puddle
  - ray-cast toward mirrored pixel position
  - test if intersection point is in puddle’s bounds

• Reject pixel if bounds test fails

• In this example result is positive
Projection pass

- We just found the reflection point
- Calculate mirrored pixel position on the screen
- Write reflected color data in that place
Projection pass: overlapping shapes

- What if the puddle was occluded by some other shapes?
- Test other shapes bounds during ray-cast
- Reject pixel if occluded
Reflection pass

- Second pass of the algorithm is the „reflection pass”
- We want to obtain reflection for given pixel
- Read data that was encoded for this pixel in „projection pass”
Reflection pass

- This gives us reflected color without doing any search.
Intermediate buffer introduction

- Pixel-data grid
- Single value per pixel
- Filled in the „projection pass”
- Read from in the „reflection pass”
Full algorithm

• Clear the „intermediate buffer”

• Projection pass
  - use rectangles (3D) instead of line segments (2D)
  - for every pixel find rectangles that reflect it
  - project pixels on those rectangles
  - write pixel-data to „intermediate buffer”

• Reflection pass
  - read pixel-data from „intermediate buffer” (simply using ‘vpos’)
  - decode pixel-data and obtain reflected color
  - write color to reflection buffer
Rendered image

- Pillar is partially visible through the flowerpot
Intermediate buffer pixel-data

- What kind of data should we store here?
- Value ideally just a reflected pixel color, but ...
  … reflection tracing is reversed, so ...
  … multiple pixels can be written to the same place
Intermediate buffer pixel-data

- Both green pixels are projected to the same place
- We are processing pixels independently
- Write order depends on GPU scheduling
- We want the closest pixel to be reflected
Intermediate buffer pixel-data

• Let's just encode screen-space offset
• For 3D scene, offset is a 2D screen-space vector
• Obtaining the offset is simple
• While writing pixel-data we already know:
  - reflected pixel coordinate (current pixel position)
  - reflecting pixel coordinate (determines where to write)
• Ensure that stored pixel-data has the smallest offset
• Write pixel-data using InterlockedMin
Intermediate buffer encoding

• We’d like to handle all possible offset orientations
• And at the same time in the „reflection pass” rely only on the „intermediate buffer” - to keep it simple
• Offset is 2D, so how can we encode it without precision loss …
• … while still being able to use InterlockedMin?
Intermediate buffer encoding

- Offset orientation defines a coordinate system
- Offset length is the ‘Y’ coordinate in this system
- ‘X’ coordinate is always zero
Intermediate buffer encoding

• We can also use a different coordinate system, …
• … as long as ‘Y’ coordinates are positive
• For InterlockedMin to work, encode ‘Y’ in most significant bits
Intermediate buffer encoding

- Turns out it’s enough to use just four coordinate systems
- Coordinate system is chosen based on offset orientation
- In the „intermediate buffer” we need to encode:
  - ‘Y’ coordinate (in most significant bits)
  - ‘X’ coordinate
  - coordinate system index (0 .. 3)
- In the „reflection pass” we’ll restore original offset based on encoded values
- „Reflection pass” based only on the „intermediate buffer”
Intermediate buffer layout

- Most → least significant bits:
  - 12 bits: 'Y' integer (unsigned)
  - 3 bits: 'Y' fraction (signed, flipped)
  - 12 bits: 'X' integer (signed)
  - 3 bits: 'X' fraction (signed, not flipped)
  - 2 bits: coordinate system index (0 .. 3)

- Offset fractions used for filtering – covered later
Rendered image

- Resolved writing order issue
Multiple simultaneous orientations

- Three different orientations encoded in the „projection pass”
Reflection pass

- We have covered the „projection pass“
- „Intermediate buffer“ contains reflected pixels data
- Now it's time for „reflection pass“
  - let's read the „intermediate buffer“ data
  - retrieve the offsets
  - generate reflected color
- „Reflection pass“ contains four logical steps
- All steps are performed in a single shader
Holes

- Original image can get stretched when reflected
- This results in missing data in the "intermediate buffer"
- Fortunately holes do not form big groups
Holes

- There are two types of holes
  - holes without any data
  - holes only in the first reflection layer
Holes patching

- Find neighboring pixel-data with smallest offset
- Calculate reflection the same way as for selected neighbor ...  
  ... but only if it's offset is significantly smaller than original offset
Distortion

- Reflections are slightly distorted
- Because holes are filled with neighboring pixel reflection
- Also because “intermediate buffer” is a discrete grid
Filtering

- Filtered color sampling
- Make use of fixed point fractions encoded in the „intermediate buffer“
Color bleeding

- Filtered color sampling artifact
- Visible in high contrast areas
- Doesn't happen with non-filtered sampling
Color bleeding reduction

• Combine filtered and non-filtered samples
• Ensure combined result is close enough to non-filtered sample
• Hue and luminance handled separately [Karis14]
Reflection pass: step by step

• All steps are performed in a single shader execution
Reflection pass: step by step

- All steps are performed in a single shader execution

Raw pixel-data
Reflection pass: step by step

• All steps are performed in a single shader execution

Raw pixel-data

Holes filling
Reflection pass: step by step

- All steps are performed in a single shader execution

Raw pixel-data  Holes filling  Filtering
Reflection pass: step by step

- All steps are performed in a single shader execution

Raw pixel-data | Holes filling | Filtering | Anti-bleeding
Reflection pass: step by step

- Magnified images

Raw pixel-data  Holes filling  Filtering  Anti-bleeding
Performance and quality

• Comparison of selected techniques
• Measured on nVidia GTX 1070
• 4K resolution (3840x2160)
• Full resolution reflection
Performance and quality: Techniques

**Brute force SSR** [McGuire14]
- pixel granularity

**Low quality SSR** [McGuire14]
- skip every 64 pixels
- binary refinement

**Hierarchical depth SSR** [Uludag14]
- no tracing behind geometry implemented
- on the other hand that makes it cheaper

**Pixel-projected reflections**
- presented technique
Performance and quality: Brute force SSR

20,6 ms
Performance and quality: Low quality SSR

0.95 ms
Performance and quality: Hierarchical depth SSR

3.2 ms
(+0.35ms for hiZ generation)
Performance and quality: Pixel-projected reflections

0.95 ms
Make sure to see „Bonus slides”!
Summary

• Simple technique
• Easy to integrate
• High quality / cost ratio
• More limited than SSR
• Can be selectively used where applicable
Special thanks

• Natalya Tatarchuk
• Michał Iwanicki
• Tomasz Jonarski
• Havok Germany team
Thank you!
References

  http://jcgt.org/published/0003/04/04
  http://advances.realtimerendering.com/s2015/
  https://www.slideshare.net/MicheleGiacalone1/screen-space-reflections-in-the-surge
- [Valient13] Michal Valient “Killzone Shadow Fall Demo Postmortem”
  http://www.guerrilla-games.com/presentations/Valient_Killzone_Shadow_Fall_Demo_Postmortem.html
- [Karis14] Brian Karis "High-quality Temporal Supersampling"
  http://advances.realtimerendering.com/s2014/
- [Uludag14] Yasin Uludag "Hi-Z Screen-Space Cone-Traced Reflections“
  GPU Pro 5
Content credits

• Stanford Dragon:
  - Stanford 3D Scanning Repository

• Atrium Sponza Palace:
  - Marko Dabrovic
  - Frank Meinl
  - Morgan McGuire
  - Alexandre Pestana
  - Crytek
Bonus slides
Previous work

• Just a quick memory refresher
• Screen space techniques only
• Sharp reflections only
Previous work: Basic raymarching

- Calculate reflection vector in screen space (3D)
- Sample depth buffer along this vector
- Four samples per loop iteration
- Stop if current tap behind depth buffer
- Sample color buffer with last tap coordinates
- Thickness value to raymarch behind geometry
- Very slow at pixel granularity
- [McGuire14]
Previous work: Basic optimization

• Sample every n-th pixel
• Lower depth resolution
  (we're skipping pixels anyway)
• Reversed 16bit float depth buffer
• Heavy banding
• [McGuire14]
Previous work: Binary search refinement

- Obtain reflection coordinate through raymarching
- Binary search around last step segment
- Only a few refinement taps needed
- Good if reflecting flat geometry
- Bad if reflecting depth discontinuity
- Simple and fast
- [McGuire14]
Previous work: Dithered raymarching

- Per pixel noise
- Bayer pattern works well
- Offset samples by noise * stepSize
- No banding, but reflections noisy
- Smooth with temporal filter and animated noise
- Binary search refinement to reduce bluriness
- Lowered memory coherence
- Deinterleaving possible
- [Giacaone16]
Previous work: Hierarchical raymarching

- Optimized search
- Per-pixel granularity result
- Needs hierarchical depth buffer (hi-Z)
- Traverse hi-Z mipchain during raymarching
- Step size based on current mip level
- Power of two hi-Z for simpler shader
- Don't bother with full mip chain
- [Uludag14]
Offset fractions mirroring

- Filtered reflections 'shake' under motion, because pixel-data is stored in a pixel grid
- Offset fractions need to be flipped
- Flip fractions along offset direction
- XY coordinates in screen-space (2D)
- Performed in „projection pass”
- Solves the shakiness
Multiple parallel shapes

- Find shapes that reflect given pixel
- Write offsets projected onto selected shape
Multiple parallel shapes

• Example: stairway with mirror steps
• All steps share the same normal
• Every pixel reflected at most once
• „Intermediate buffer” written at most once per pixel
Multiple parallel shapes

- Ignore shapes for which pixel is below shape’s plane
- Ignore shapes not reflecting the pixel (mirrored bounds test)
- Pick shape with smallest plane → pixel distance
- „Intermediate buffer” written at most once
Multiple parallel shapes

- Simple with a few shapes (just iterate over them)
- Large amounts need preselection (otherwise heavy)
- Multiple solutions:
  - preselection in „projection pass“
  - world space grid shape lists
  - froxel based shape lists
  - BVH lists
  - etc.
Multiple parallel shapes

GPU based approach:

- Integrated into „projection pass”
- Build thread-group tile world space bounds
- Build per-tile shapes list in groupshared memory
  - every thread is processing one shape
  - project tile world space bounds to screen space
  - test against shape bounds
  - append to shared list (test positive)
- Pick best shape from global list
- Proceed as in single shape case
Multiple parallel shapes

GPU based approach:
- test case similar to other measurements
- 0.5 ms overhead
Transparent reflectors

- Glass panels, windows
- Barely used in games
- High cost because of raymarching
- Much cheaper with „Pixel-projected reflections”
Transparent reflectors

- Additional „intermediate buffer“ (atlas)
- For every glass panel:
  - calculate screen bounds
  - allocate region in atlas
  - use atlas as „intermediate buffer“
  - write pixel-data to allocated atlas region
- Fetch atlas while rendering glass panels
Transparent reflectors
Normalmaps

• Approximation
• Integrated into „reflection pass”
• Calculate normalmap based reflected direction
• Scale by flat reflection distance
• Add to world space position
• Project to screen-space
• Fetch color buffer
Normalmaps

- **Problem**: Disocclusion
- Noticeable on depth discontinuities
- Samples closer than approximated reflection
Normalmaps

- **Solution**: Raymarching (4 taps) toward reflecting pixel
- Accept sample if closer than depth buffer value
- Fallback to flat reflection if no sample found
Normalmaps

- Ground truth reference
- Raymarched reflection against normalmap
- Better quality on depth discontinuities
// Constants for 'intermediate buffer' values encoding.
// Lowest two bits reserved for coordinate system index.
#define PPR_CLEAR_VALUE (0xffffffff)
#define PPR_PREDICTED_DIST_MULTIPLIER (8)
#define PPR_PREDICTED_OFFSET_RANGE_X (2048)
#define PPR_PREDICTED_DIST_OFFSET_X (PPR_PREDICTED_OFFSET_RANGE_X)
#define PPR_PREDICTED_DIST_OFFSET_Y (0)

// Calculate coordinate system index based on offset
uint PPR_GetPackingBasisIndexTwoBits( const float2 offset )
{
    if ( abs( offset.x ) >= abs( offset.y ) )
    {
        return offset.x >= 0 ? 0 : 1;
        return offset.y >= 0 ? 2 : 3;
    }

    // Decode coordinate system based on encoded coordSystem index
    float2x2 PPR_DecodePackingBasisIndexTwoBits( uint packingBasisIndex )
    {
        float2 basis = 0;
        packingBasisIndex &= 3;
        basis.x += 0 == packingBasisIndex ? 1 : 0;
        basis.x += 1 == packingBasisIndex ? -1 : 0;
        basis.y += 2 == packingBasisIndex ? 1 : 0;
        basis.y += 3 == packingBasisIndex ? -1 : 0;
        return float2x2( float2( basis.y, -basis.x ), basis.xy );
    }
}
// Pack integer and fract offset value
uint PPR_PackValue( const float _whole, float _fract, const bool isY )
{
    uint result = 0;

    // pack whole part
    result += (uint)(_whole + (isY ? PPR_PREDICTED_DIST_OFFSET_Y : PPR_PREDICTED_DIST_OFFSET_X));
    result *= PPR_PREDICTED_DIST_MULTIPLIER;

    // pack fract part
    _fract *= PPR_PREDICTED_DIST_MULTIPLIER;
    result += (uint)min( floor( _fract + 0.5 ), PPR_PREDICTED_DIST_MULTIPLIER - 1 );

    // return result;
}

// Unpack integer and fract offset value
float2 PPR_UnpackValue( uint v, const bool isY )
{
    // unpack fract part
    float _fract = (v % PPR_PREDICTED_DIST_MULTIPLIER + 0.5) / float(PPR_PREDICTED_DIST_MULTIPLIER) - 0.5;
    v /= PPR_PREDICTED_DIST_MULTIPLIER;

    // unpack whole part
    float _whole = int(v) - (isY ? PPR_PREDICTED_DIST_OFFSET_Y : PPR_PREDICTED_DIST_OFFSET_X);

    // return float2( _whole, _fract );
}
// Encode offset for 'intermediate buffer' storage
uint PPR_EncodeIntermediateBufferValue( const float2 offset )
{
    // build snapped basis
    const uint packingBasisIndex = PPR_GetPackingBasisIndexTwoBits( offset );
    const float2x2 packingBasisSnappedMatrix = PPR_DecodePackingBasisIndexTwoBits( packingBasisIndex );

    // decompose offset to _whole and _fract parts
    float2 _whole = floor(offset + 0.5);
    float2 _fract = offset - _whole;

    // mirror _fract part to avoid filtered result 'swimming' under motion
    const float2 dir = normalize( offset );
    _fract -= 2 * dir * dot( dir, _fract );

    // transform both parts to snapped basis
    _whole = mul( packingBasisSnappedMatrix, _whole );
    _fract = mul( packingBasisSnappedMatrix, _fract );

    // put _fract part in 0..1 range
    _fract *= 0.707;
    _fract += 0.5;

    // encode result
    uint result = 0;
    result += PPR_PackValue( _whole.y, _fract.y, true );
    result *= 2 * PPR_PREDICTED_OFFSET_RANGE_X * PPR_PREDICTED_DIST_MULTIPLIER;
    result += PPR_PackValue( _whole.x, _fract.x, false );
    result *= 4;
    result += packingBasisIndex;

    // return result;
}
Shader code

// Decode value read from 'intermediate buffer'
void PPR_DecodeIntermediateBufferValue( uint value, out float2 distFilteredWhole, out float2 distFilteredFract, out float2x2 packingBasis )
{
    distFilteredWhole = 0;
    distFilteredFract = 0;
    packingBasis = float4( 1, 0, 0, 1 );
    if ( value != PPR_CLEAR_VALUE )
    {
        const uint setFullValueRange = 2 * PPR_PREDICTED_OFFSET_RANGE_X * PPR_PREDICTED_DIST_MULTIPLIER;

        // decode packing basis
        packingBasis = PPR_DecodePackingBasisIndexTwoBits( value );
        value /= 4;

        // decode offsets along (y) and perpendicular (x) to snapped basis
        float2 x = PPR_UnpackValue( value & (setFullValueRange - 1), false );
        float2 y = PPR_UnpackValue( value / setFullValueRange, true );

        // output result
        distFilteredWhole = float2( x.x, y.x );
        distFilteredFract = float2( x.y, y.y );
        distFilteredFract /= 0.707;
    }
}
Shader code

// Combine filtered and non-filtered color sample to prevent color-bleeding.
// Current implementation is rather naive and may result in distortion artifacts
// (created by holes-filling) to become visible again at some extent.
// Anti-bleeding solution might use some further research to reduce artifacts.
// Note that for high resolution reflections, filtering might be skipped, making
// anti-bleeding solution unneeded.
float3 PPR_FixColorBleeding( const float3 colorFiltered, const float3 colorUnfiltered )
{
    // transform color to YCoCg, normalize chrominance
    float3 ycocgFiltered = mul( RGB_to_YCoCg(), colorFiltered );
    float3 ycocgUnfiltered = mul( RGB_to_YCoCg(), colorUnfiltered );
    ycocgFiltered.yz /= max( 0.0001, ycocgFiltered.x );
    ycocgUnfiltered.yz /= max( 0.0001, ycocgUnfiltered.x );

    // calculate pixel sampling factors for luma/chroma separately
    float lumaPixelsamplingFactor = saturate( 3.0 * abs(ycocgFiltered.x - ycocgUnfiltered.x) );
    float chromaPixelsamplingFactor = saturate( 1.4 * length(ycocgFiltered.yz - ycocgUnfiltered.yz) );

    // build result color YCoCg space
    // interpolate between filtered and nonFiltered colors (luma/chroma separately)
    float resultY = lerp( ycocgFiltered.x, ycocgUnfiltered.x, lumaPixelsamplingFactor );
    float2 resultCoCg = lerp( ycocgFiltered.yz, ycocgUnfiltered.yz, chromaPixelsamplingFactor );
    float3 ycocgResult = float3( resultY, resultCoCg * resultY );

    // transform color back to RGB space
    return mul( YCoCg_to_RGB(), ycocgResult );
}
// Write projection to 'intermediate buffer'.
// Pixel projected from 'originalPixelVpos' to 'mirroredWorldPos'.
// Function called in 'projection pass' after ensuring that pixel projected into given
// place of the shape is not occluded by any other shape.

void PPR_ProjectionPassWrite(
    SSharedConstants globalConstants,
    RWStructuredBuffer<uint> uavIntermediateBuffer,
    const int2 originalPixelVpos,
    const float3 mirroredWorldPos
) {
    const float4 projPosOrig = mul( float4( mirroredWorldPos, 1 ),
        globalConstants.worldToScreen );
    const float4 projPos = projPosOrig / projPosOrig.w;
    if ( all( abs( projPos.xy ) < 1 ) )
    {
        const float2 targetCrd = (projPos.xy * float2(0.5, -0.5) + 0.5) *
            globalConstants.resolution.xy;
        const float2 offset = targetCrd - (originalPixelVpos + 0.5);
        const uint writeOffset = uint(targetCrd.x) +
            uint(targetCrd.y) * uint(globalConstants.resolution.x);
        // PPR_EncodeIntermediateBufferValue( offset );
        originalValue = 0;
        valueToWrite = PPR_EncodeIntermediateBufferValue( offset );
        InterlockedMin( uavIntermediateBuffer[ writeOffset ],
            valueToWrite, originalValue );
    }
}
Shader code

// 'Reflection pass' implementation.
float4 PPR_ReflectionPass(
    SSharedConstants globalConstants, const int2 vpos, StructuredBuffer<uint> srvIntermediateBuffer, Texture2D srvColor, SamplerState smpLinear,
    SamplerState smpPoint, const bool enableHolesFilling, const bool enableFiltering, const bool enableFilterBleedingReduction
)
{
    int2 vposread = vpos;

    // perform holes filling.
    // If we're dealing with a hole then find a closeby pixel that will be used
    // to fill the hole. In order to do this simply manipulate variable so that
    // compute shader result would be similar to the neighbor result.
    float2 holesOffset = 0;
    if (enableHolesFilling)
    {
        uint v0 = srvIntermediateBuffer[ vpos.x + vpos.y * int(globalConstants.resolution.x) ];
        {
            // read neighbors 'intermediate buffer' data
            const int2 holeOffset1 = int2( 1, 0 );
            const int2 holeOffset2 = int2( 0, 1 );
            const int2 holeOffset3 = int2( 1, 1 );
            const int2 holeOffset4 = int2(-1, 0 );
            const uint v1 = srvIntermediateBuffer[ (vpos.x + holeOffset1.x) + (vpos.y + holeOffset1.y) * int(globalConstants.resolution.x) ];
            const uint v2 = srvIntermediateBuffer[ (vpos.x + holeOffset2.x) + (vpos.y + holeOffset2.y) * int(globalConstants.resolution.x) ];
            const uint v3 = srvIntermediateBuffer[ (vpos.x + holeOffset3.x) + (vpos.y + holeOffset3.y) * int(globalConstants.resolution.x) ];
            const uint v4 = srvIntermediateBuffer[ (vpos.x + holeOffset4.x) + (vpos.y + holeOffset4.y) * int(globalConstants.resolution.x) ];
            // get neighbor closest reflection distance
            const uint minv = min( min( min( v0, v1 ), min( v2, v3 ) ), v4 );
        }
    }

    // next slide
// previous slide

// allow hole fill if we don't have any 'intermediate buffer' data for current pixel,
// or any neighbor has reflection significantly closer than current pixel's reflection
bool allowHoleFill = true;
if ( PPR_CLEAR_VALUE != v0 )
{
    float2 d0_filtered_whole;
    float2 d0_filtered_fract;
    float2x2 d0_packingBasis;
    float2 dmin_filtered_whole;
    float2 dmin_filtered_fract;
    float2x2 dmin_packingBasis;
    PPR_DecodeIntermediateBufferValue( v0, d0_filtered_whole, d0_filtered_fract, d0_packingBasis );
    PPR_DecodeIntermediateBufferValue( minv, dmin_filtered_whole, dmin_filtered_fract, dmin_packingBasis );
    float2 d0_offset = mul( d0_filtered_whole + d0_filtered_fract, d0_packingBasis );
    float2 dmin_offset = mul( dmin_filtered_whole + dmin_filtered_fract, dmin_packingBasis );
    float2 diff = d0_offset - dmin_offset;
    const float minDist = 6;
    allowHoleFill = dot( diff, diff ) > minDist * minDist;
}

// hole fill allowed, so apply selected neighbor's parameters
if ( allowHoleFill )
{
    if ( minv == v1 ) vposread = vpos + holeOffset1;
    if ( minv == v2 ) vposread = vpos + holeOffset2;
    if ( minv == v3 ) vposread = vpos + holeOffset3;
    if ( minv == v4 ) vposread = vpos + holeOffset4;
    holesOffset = vposread - vpos;
}

// next slide
// previous slide

// obtain offsets for filtered and non-filtered samples
float2 predictedDist = 0;
float2 predictedDistUnfiltered = 0;
{
    uint v0 = srvIntermediateBuffer[ vposread.x + vposread.y * int(globalConstants.resolution.x) ];
}

// decode offsets
float2 decodedWhole;
float2 decodedFract;
float2x2 decodedPackingBasis;
{
    PPR_DecodeIntermediateBufferValue( v0, decodedWhole, decodedFract, decodedPackingBasis );
    predictedDist = mul( decodedWhole + decodedFract, decodedPackingBasis );
    // fractional part ignored for unfiltered sample, as it could end up
    // sampling neighboring pixel in case of non axis aligned offsets.
    predictedDistUnfiltered = mul( decodedWhole, decodedPackingBasis );
}

// include holes offset in predicted offsets
if ( PPR_CLEAR_VALUE != v0 )
{
    const float2 dir = normalize( predictedDist );
    predictedDistUnfiltered -= float2( holesOffset.x, holesOffset.y );
    predictedDist -= 2 * dir * dot( dir, holesOffset );
}

// exit if reflection offset not present
if ( all( predictedDist == 0 ) )
{
    return 0;
}

// next slide
Shader code

// previous slide

// sample filtered and non-filtered color
const float2 targetCrd = vpos + 0.5 - predictedDist;
const float2 targetCrdUnfiltered = vpos + 0.5 - predictedDistUnfiltered;
const float3 colorFiltered = srvColor.SampleLevel( smpLinear, targetCrd * globalConstants.resolution.zw, 0 ).xyz;
const float3 colorUnfiltered = srvColor.SampleLevel( smpPoint, targetCrdUnfiltered * globalConstants.resolution.zw, 0 ).xyz;

// combine filtered and non-filtered colors
float3 colorResult;
if ( enableFiltering )
{
    if ( enableFilterBleedingReduction )
    {
        colorResult = PPR_FixColorBleeding( colorFiltered, colorUnfiltered );
    }
    else
    {
        colorResult = colorFiltered;
    }
}
else
{
    colorResult = colorUnfiltered;
}

// return float4( colorResult, 1 );
}