

Optimized pixel-projected reflections for planar reflectors

a.k.a. Pixel-projected reflections

Adam Cichocki

 @multisampler



private research

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]

- Let's start with a quick overview of screen-space reflections.
- This is a very widely adopted technique.
- The algorithm is based on ray-marching, and the reflection itself is obtained from already rendered color and depth buffer. This is an important property, because we don't have to pay extra cost for rendering the scene geometry from different points of view.
- The drawback of this approach is that the resulting reflection contains only information that is available in already rendered buffers. So typically there are multiple areas on the screen without available reflection data. This is usually addressed by using some kind of fallback system, most often reflection probes. [Valient13]
- Beside that the technique is complex especially in terms of memory bandwidth, so it's often applied at half resolution, to meet the frame budget.

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
-
- So how does it compare to the presented technique.
 - Pixel-projected reflections is a constrained form of raymarched SSR. It's more limited in terms of application, and requires additional content authoring,
 - But in return it offers high performance and quality.
 - What's important is that when the technique's limitations are met, then the result is comparable to ray-marched SSR. So pixel-projected reflections can be used selectively, instead of ray-marched reflections, through compute shader indirect dispatching.
 - The algorithm is based on data scattering instead of gathering, and the actual reflection is generated based on already rendered color and depth buffers (as in SSR), but additionally requires analytical information about reflective areas of the scene.

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

- As you might already suspect, the main concept behind the technique is to reverse the reflection tracing.
- Instead of trying to find for every pixel where does the reflected ray hit the depth buffer, we'll just calculate exactly where on the screen every pixel is being reflected. This is a powerful simplification because it doesn't involve reading the depth buffer multiple times.
- So in order to be able to perform the calculation, we'll just approximate the reflective areas of the scene with flat analytical shapes, and provide them to the shader through shader constants.
- The most practical shapes are rectangles, so we will use them, but other shapes can be also used if needed.

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

- As I already mentioned, pixel-projected reflections have some additional limitations when compared to SSR.
- We're able to generate only non-glossy reflections this way, and reflections can be applied only on planar surfaces,
- Surface normalmaps are not natively supported, but can be approximated in various ways. One of the solutions for this problem is proposed in “bonus slides”.
- The technique also requires additional content authoring, more specifically reflector shapes that approximate reflective areas of the scene. These are simple shapes that need to be aligned with reflective geometry, and should cover all parts of the geometry that are meant to use pixel-projected reflections. Authoring of these reflectors need to be manually done on a prefab level, but it seems as this process could be automated.

Simplified algorithm in 2D

- Just to grasp the pixel-projection concept
- We'll switch to 3D afterward



- Next few slides are going to cover a simplified version of the algorithm in two dimensions.
- This is just to get a good understanding of the pixel-projection concept.
- We're going to switch to 3D afterward.

Example 2D scene

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



- What you can see here is a simple two dimensional scene.
- It contains a single object, and the puddle on the floor.
- The puddle is approximated with a line segment.

Projection pass

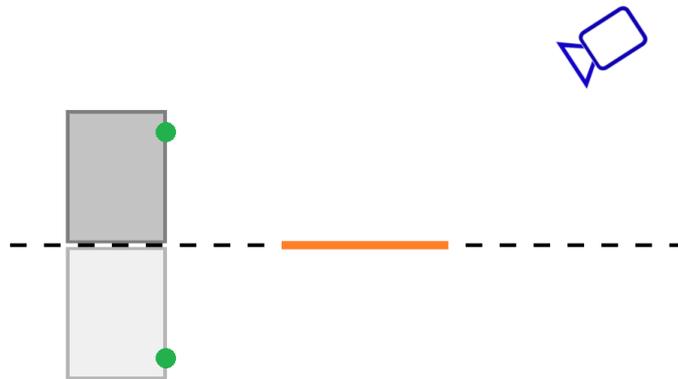
- First pass of the algorithm is the „projection pass”
- For every pixel calculate where it is reflected on the screen
 - single pixel in the example



- First pass of the algorithm is the “projection pass”.
- For all pixels on the screen we’re going to calculate where on the screen they are being reflected.
- In this simple example we will focus on a single pixel – marked in green on the object.

Projection pass

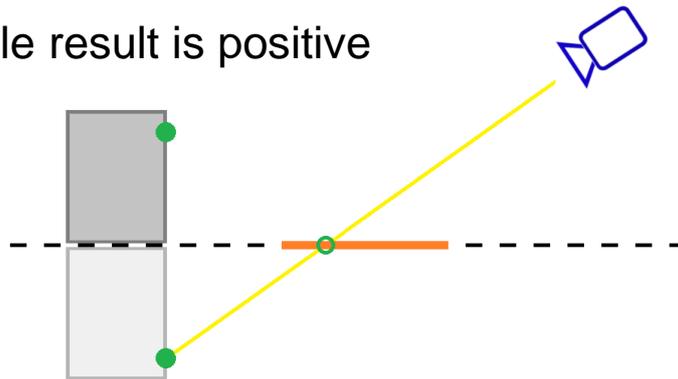
- Mirror pixel position against puddle's plane



- First we have to mirror pixel's position against puddle's plane.
- This is how the pixel would be seen from the camera's point of view, if reflected by puddle's plane.

Projection pass

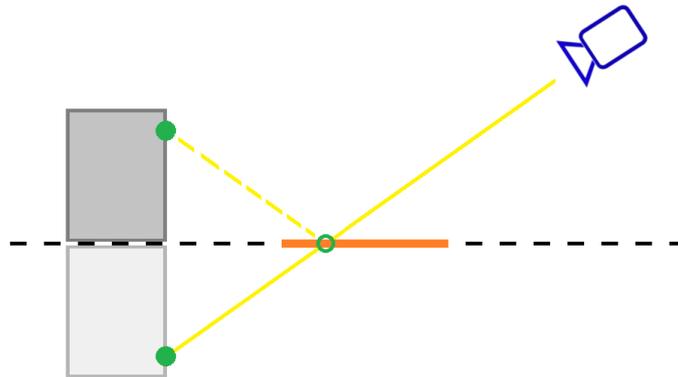
- Test if pixel's reflection is visible in the puddle
 - ray-cast toward mirrored pixel position
 - test if intersection point is in puddle's bounds
- Reject pixel if bounds test fails
- In this example result is positive



- Then we'll test whether the pixel is visible in the puddle. In order to do this, we just perform a ray-cast from the camera toward the mirrored pixel, obtain the intersection point with puddle's plane, and check whether this point lies inside the reflector shape.
- If the boundary test fails, then the original pixel is not reflected by the puddle,
- But if it's positive, as in this example, then we have found the reflection point.

Projection pass

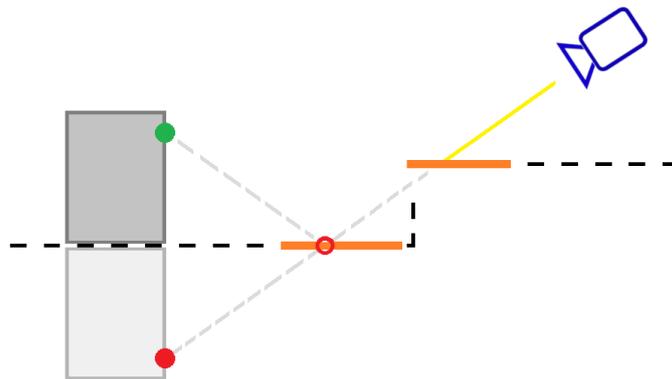
- We just found the reflection point
- Calculate mirrored pixel position on the screen
- Write reflected color data in that place



- What's left is to calculate mirrored pixel position on the screen, and we're going to do this simply by applying perspective projection.
- Then we'll just write the reflected color information to that place.

Projection pass: overlapping shapes

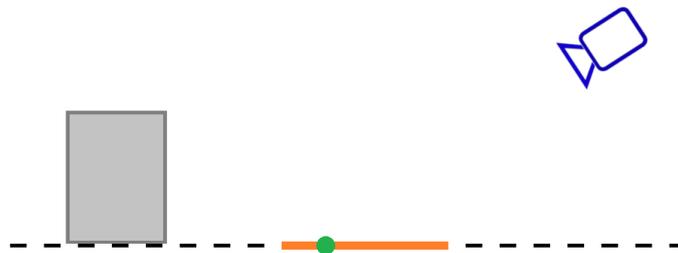
- What if the puddle was occluded by some other shapes?
- Test other shapes bounds during ray-cast
- Reject pixel if occluded



- But what would happen if the puddle, on which we have just projected our pixel, was occluded by some other reflective shape?
- Writing reflected color information in this case is not only redundant, but also leads to artifacts.
- To avoid this, we'll just additionally test other reflective shapes bounds during ray-casting.
- If our puddle is not the closest reflective shape hit by the ray, then we simply skip outputting reflected color information.
- Ray-casting can be accelerated, by building on the cpu-side per-shape potentially occluding shapes list, and passing these lists to the "projection pass" shader.

Reflection pass

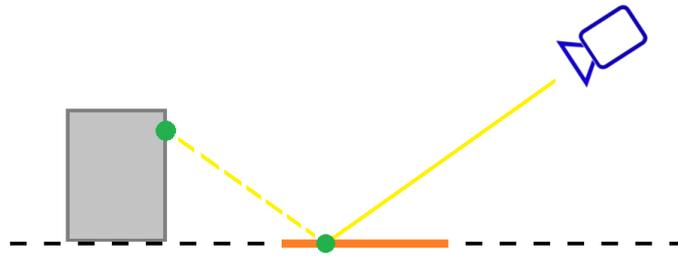
- Second pass of the algorithm is the „reflection pass”
- We want to obtain reflection for given pixel
- Read data that was encoded for this pixel in „projection pass”



- But let's get back to our simple case...
- We have finished the “projection pass”, and information about reflections have been scattered throughout the screen.
- So now it's time for the reflection pass.
- In this example we want to obtain the reflection for pixel marked in green, that is positioned on the puddle.
- All we need to do, is just read the data that was stored for this pixel by the “projection pass”.

Reflection pass

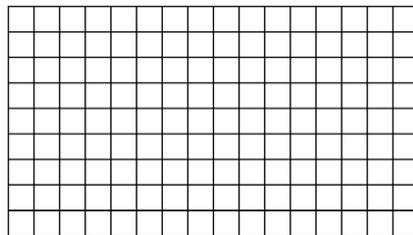
- This gives us reflected color without doing any search



- This gives us reflected color without doing any searching, and the reflection pass is done.

Intermediate buffer introduction

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



- So now it's time to introduce the "intermediate buffer".
- "Intermediate buffer" is just a regular pixel-data grid, similar to color and depth buffer.
- It stores a single value for every pixel on the screen.
- It's filled in the "projection pass" through data scattering, and read in the "reflection pass" to obtain final reflection.

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

- Full algorithm contains three passes.

- First pass is just to clear the intermediate buffer.

- Second pass is the “projection pass”.

** For 3D we’re going to use rectangles instead of line segments.

** For every pixel we’re going to find rectangles that reflect it, project pixels on those rectangles, and write corresponding pixel-data to the “intermediate buffer” in places, where current pixel is being reflected.

- Third pass is the “reflection pass”

** Here we’re going to read pixel-data from the “intermediate buffer” simply by using the vpos.

** We’ll then decode this pixel-data, and this way obtain the reflected color.

** Then we’re going to write this color to the final reflection buffer.

Rendered image

- Pillar is partially visible through the flowerpot



- Here is a scene rendered using this algorithm.
- Please take a look at the magnifying glass on the right side of the screen.
- As you can notice, the reflection of the flowerpot is broken.
- Some pixels instead reflecting the flowerpot, contain the reflection of the pillar behind it.
- There must be something wrong with the data stored in the intermediate-buffer...

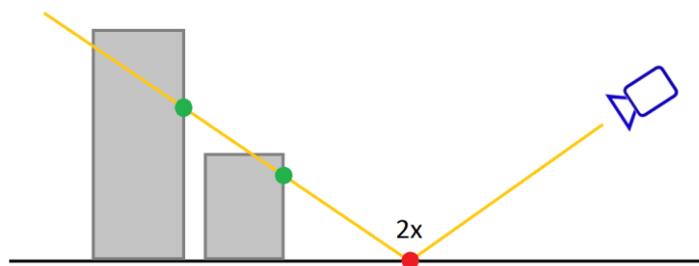
Intermediate buffer pixel-data

- 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

- So what kind of data should we store here?
- Ideally that would be just a reflected color, but as you remember the reflection tracing is reversed, so as a result, multiple pixels can be written to the same place of the intermediate buffer.

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



- Here you can see an illustration what happens.
- There are two pixels, both marked in green, that are projected to the same place, which is the red spot on the ground.
- The leftmost pixel should be occluded by the other one, but because we're processing pixels independently, we are unable to resolve this in the "projection pass" shader.
- So both pixels will get written to the intermediate buffer, which means that the last pixel that was written to the "intermediate buffer", will be later read by the "reflection pass" and interpreted as reflection. Data writing order is indeterministic and depends on GPU scheduling. This is exactly what happened in the 'flowerpot' screenshot.
- What we would like instead, is always the closest pixel ending up in the intermediate buffer.

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

- To solve the order issue, lets just encode a screen-space offset instead of color.
- For a 3D scene, offset is a 2D screen-space vector.
- Obtaining the offset is simple, because while writing pixel-data in the “projection pass” we already know the reflected pixel coordinate (this is the current pixel position), and the reflecting pixel coordinate (which is the place where we’re about to write the data).
- So all we need to do, is just ensure that pixel-data with the smallest offset is the one, that will end up in the intermediate buffer.
- In order to do this, we’ll just use the InterlockedMin operation.

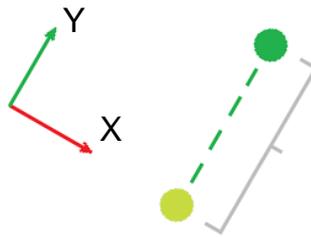
Intermediate buffer encoding

- 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?

- What's very important is that we need to handle all possible offset orientations.
- At the same time in the “reflection pass”, we would like to rely only on the “intermediate buffer”, just to keep it simple.
- But offsets have two dimensions, so how can we encode it without precision loss, and still be able to use the InterlockedMin?

Intermediate buffer encoding

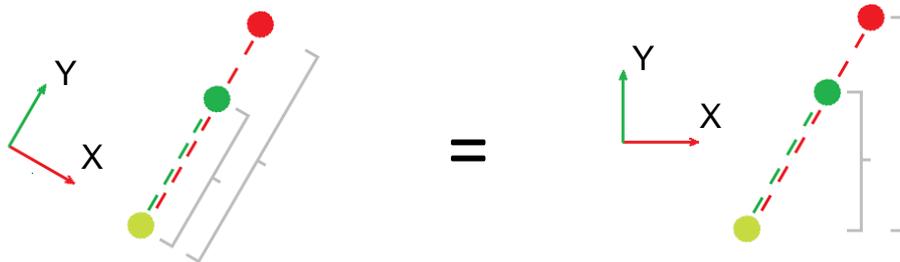
- Offset orientation defines a coordinate system
- Offset length is the 'Y' coordinate in this system
- 'X' coordinate is always zero



- Let's take a look at what is really needed for InterlockedMin to work.
- In this illustration you can see a green pixel, which is projected on the yellow pixel, forming an offset.
- Offset orientation defines a coordinate system, in which offset length is the same as 'Y' coordinate, and 'X' coordinate is always zero.

Intermediate buffer encoding

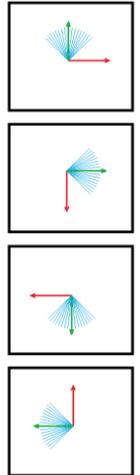
- We can also use a different coordinate system, ...
- ... as long as 'Y' coordinates are positive
- For InterlockedMin to work, encode 'Y' in most significant bits



- But we could also express this offset in a different coordinate system, so that the InterlockedMin operation would still work in our use case.
- Please take a look at the illustration... Both red and green pixels are projected on the yellow pixel.
- Offsets are expressed in two different coordinate systems – one for each illustration.
- Green pixel is closer to the projection point than the red pixel, but also green pixel's 'Y' coordinate is smaller than the red pixel's one.
- What's important here is that this property is true for both coordinate systems.
- This means that we can choose any coordinate system, in which 'Y' coordinates are still positive. But only as long as we are able to encode transformed offsets with high enough precision.

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”



- As it turns out, for our needs it's enough to just use one of four coordinate systems.
- On the right side of the screen you can see which coordinate system is chosen for different offset orientations.
- We're going to choose the coordinate system that is closest to the offset's orientation, transform offset by this coordinate system, and encode the intermediate buffer value in such way, so that the 'Y' coordinate occupies most significant bits. The 'X' coordinate and the coordinate system index will be encoded in less significant bits. This is enough for InterlockedMin to work.
- Later in the 'reflection pass' we're going to decode this data, transform offset back to screen-space, and use it to fetch color from the color buffer.
- This way we are able to encode any offset orientation and length in the intermediate buffer, and ensure that the closest pixel will be reflected.
- Additionally "reflection pass" is now 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

- Here is the final layout of the “intermediate buffer”, capable to encode every offset in 4K resolution.
 - As I already mentioned, ‘Y’ coordinate is just a slightly scaled offset length, so it’s stored in the most significant bits, to have biggest influence on the InterlockedMin operation.
 - In less significant bits we just encode the ‘X’ coordinate and the coordinate system itself.
- (pause)
- You may notice, that we’re also encoding offset fractions. They will be used later for reflections filtering.

Rendered image

- Resolved writing order issue



- As you can see, proper “intermediate buffer” encoding, and using InterlockedMin, fixed the problem with broken flowerpot.

Multiple simultaneous orientations

- Three different orientations encoded in the „projection pass”



- And here are three reflectors with different orientations, handled simultaneously.

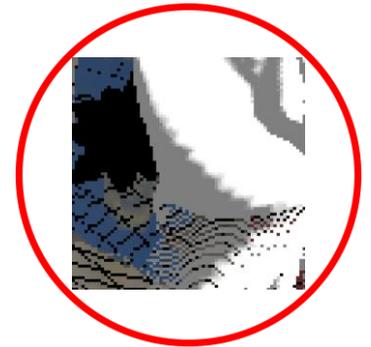
Reflection pass

- 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

- We have covered the “projection pass”, so let’s move on to the “reflection pass”
- The “reflections pass” is responsible for generating final reflected color.
- For this to happen, we’ll have to read the “intermediate buffer” data, decode the offsets, and compose the reflected color.
- In practice it’s more complicated than it sounds, and the “reflection pass” contains four logical steps, all of them performed in a single shader execution.

Holes

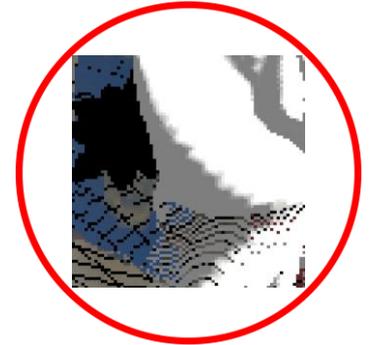
- Original image can get stretched when reflected
- This results in missing data in the „intermediate buffer”
- Fortunately holes do not form big groups



- When we decode the “intermediate buffer” without any processing, we get reflections with holes.
- This happens because the original image can get stretched in the reflection, which results in some data missing in the “intermediate buffer”.
- Fortunately holes are not forming large groups, so they can be easily fixed.

Holes

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



- It's worth to keep in mind, that there are two types of holes, which makes the problem a bit harder.
- Simplest case is just missing data in the "intermediate buffer".
- The other case is when the data was written to the "intermediate buffer", but geometry that is actually closer got stretched, and wasn't stored for some pixels.

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



- In order to patch the holes, we're just going to find neighboring pixel-data with the smallest offset, and calculate the reflection the same way, as for selected neighbor.
- But we're going to use this data only if there is no "intermediate buffer" data for current pixel, or selected neighbor offset is significantly smaller than our current offset.
- This condition is tested, because we want to limit the patching only to holes, since other pixels already contain optimal data.

Distortion

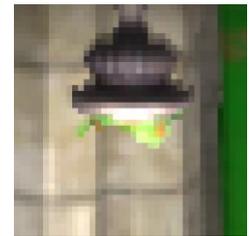
- Reflections are slightly distorted
- Because holes are filled with neighboring pixel reflection
- Also because „intermediate buffer” is a discrete grid



- Holes are now fixed, but we can observe that reflections are slightly distorted.
- Depending on the camera angle, rendered scene buffers when reflected, can get either stretched or squeezed.
- Squeezing results in some of the pixels not ending up in the “intermediate buffer”, which forms a color discontinuity.
- Stretching, on the other hand, results in holes, which we’re patching with neighboring pixel’s reflection.
- So if the reflection is squeezed, then we’re missing out some pixels from the original image, and if it’s stretched, we get pixels duplication.

Filtering

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



- In order to fix the distortion, we're going to perform filtered color sampling, which is possible because of offset fractions, that we encoded in the intermediate buffer.
- As you can see in the image, with filtered color sampling, there are no distortion artifacts.

Color bleeding

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



- Unfortunately, filtered sampling introduced a new problem – color bleeding.
- It occurs because of parallax effect in the reflection.
- So we'll make use of two very basic properties here.
- First property is that color bleeding is visible only in high contrast areas, and if the bleeding is low, then it's not really a problem.
- Second property is that the bleeding doesn't occur at all, when using point 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]



- We're going to make use of those observations to fix the color bleeding.
- We'll just take non-filtered color sample along with the filtered sample, and combine them, so that the result is as close as possible to the filtered sample. But with one restriction: combined color can't be perceptually too far from the point sampled color.
- In order to do this, we'll just decompose both colors to hue and luminance, and combine those components separately, based on their values differences.
- This is similar to neighborhood clamping in temporal antialiasing solutions, but here in order to stay within the "reflection pass" shader, filtered sampled color is toned down based on a single non-filtered sample instead of the whole neighborhood.

Reflection pass: step by step

- All steps are performed in a single shader execution

- At this point we have the final reflection result, so let's just quickly take a look back at the reflection pass, step by step.

Reflection pass: step by step

- All steps are performed in a single shader execution



Raw pixel-data

- This is what we get with simple usage of the “intermediate buffer” without any processing.
- Regular lines are visible because of missing data in the “intermediate buffer”, due to reflection stretching.

Reflection pass: step by step

- All steps are performed in a single shader execution



Raw pixel-data



Holes filling

- Next we perform the holes patching, which results in slight image distortion.

Reflection pass: step by step

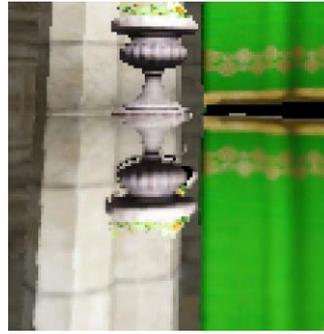
- All steps are performed in a single shader execution



Raw pixel-data



Holes filling



Filtering

- Then, in order to smooth out the distortion, we apply the filtering, which may result in color bleeding, in high contrast areas.

Reflection pass: step by step

- All steps are performed in a single shader execution



Raw pixel-data



Holes filling



Filtering



Anti-bleeding

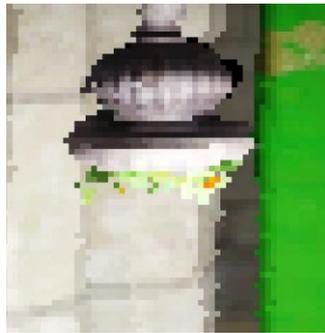
- As the last step, we apply the anti bleeding solution, which tones down the color bleeding, to perceptually acceptable level.
- We can also see here that applying the anti-bleeding, can result in some distortion artifacts to become noticeable again at some extent.
- Anti-bleeding could use some additional research. Obtaining better results might require an additional pass.

Reflection pass: step by step

- Magnified images



Raw pixel-data



Holes filling



Filtering

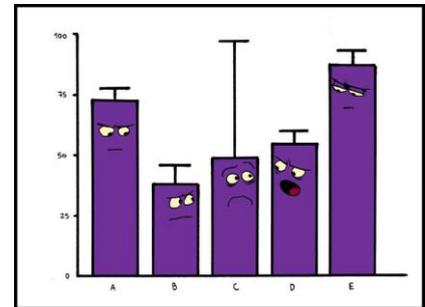


Anti-bleeding

- And here are the same images, only magnified.

Performance and quality

- Comparison of selected techniques
- Measured on nVidia GTX 1070
- 4K resolution (3840x2160)
- Full resolution reflection



- At this point we already know how the reflection is generated, so let's compare performance and quality.
- We're going to compare four real-time reflection algorithms.
- All measurements were taken on nVidia GTX 1070, at 4K resolution, with the reflection generated at full resolution.

Performance and quality: Techniques

Brute force SSR [McGuire14]

- pixel granularity

Low quality SSR [McGuire14]

- skip every 64 pixels
- binary refinement

Hierarchical depth SSR [Uludag14]

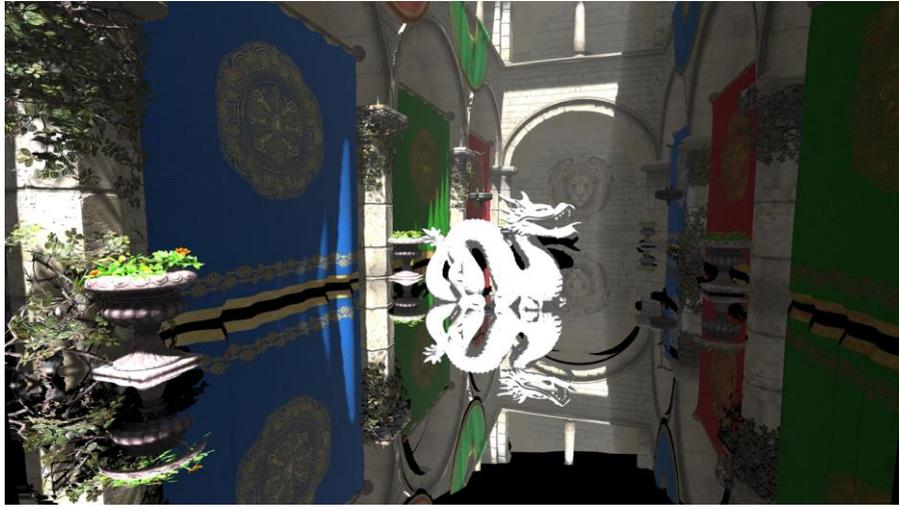
- no tracing behind geometry implemented
- on the other hand that makes it cheaper

Pixel-projected reflections

- presented technique

- First technique is brute force SSR, with reflection ray-marched at pixel granularity.
- Second technique is low-quality SSR. Here we're going to skip every 64 pixels during raymarching, and at the end of the tracing, we'll perform binary refinement of the intersection point.
- Third technique is hierarchical depth buffer SSR. Unfortunately measured implementation doesn't support tracing behind the geometry, but on the other hand, this makes the technique more efficient.
- Fourth technique is the presented technique, Pixel-projected reflections.

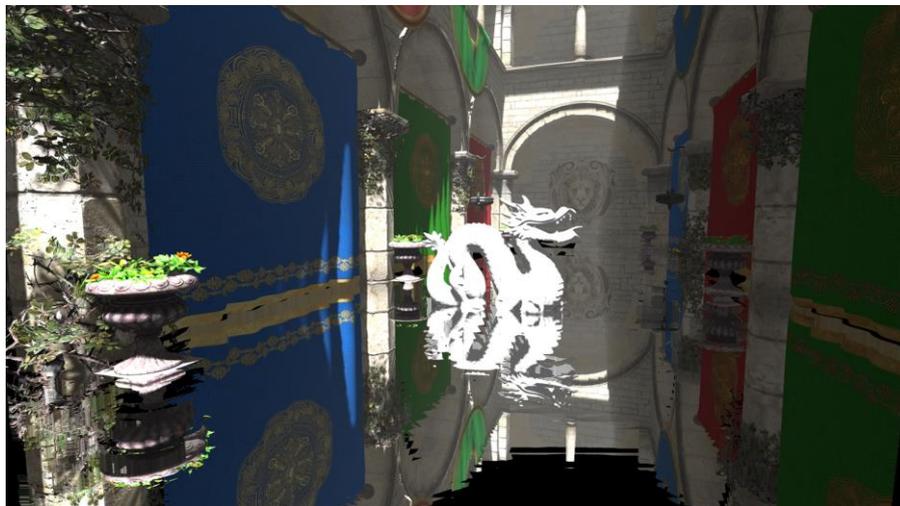
Performance and quality: Brute force SSR



20,6 ms

- Brute force SSR is here only as an introduction to practical solutions.
- It's not expected to be fast, but the tracing result can be considered as a reference.
- Brute force SSR cost is 20.6 ms

Performance and quality: Low quality SSR



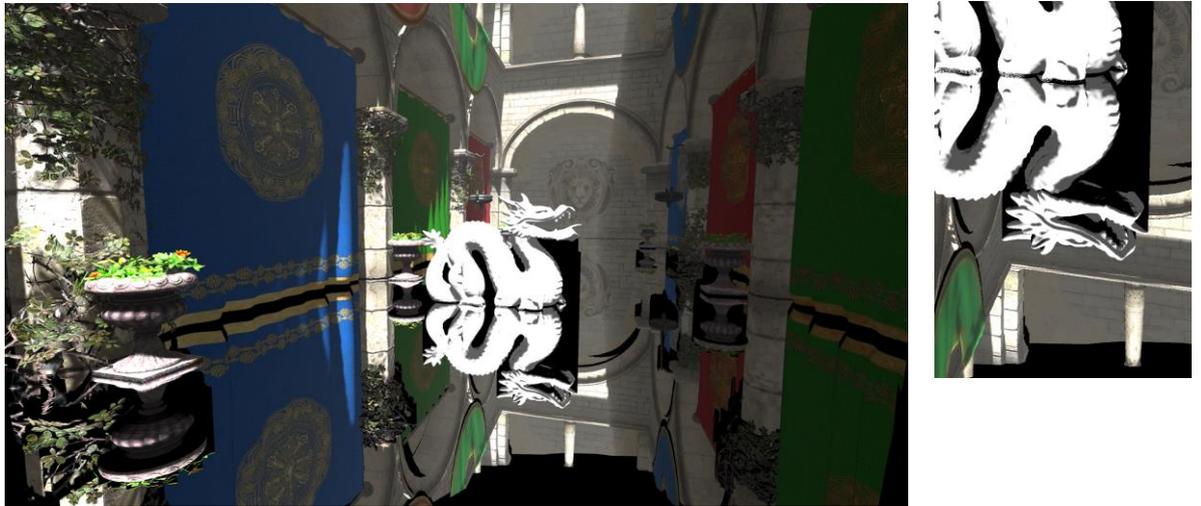
0.95 ms

- Low quality SSR offers great performance, but the result is bad on depth discontinuities in the reflection.
- You can see the artifacts in the magnifying glass, on the right side of the slide.
- Cost of applying this algorithm is 0.95 ms.

----- [skip] -----

- It's possible to improve the quality through dithering and temporal filter. But that only makes the problems a bit less obvious, because sharpness of reflection wouldn't be preserved this way, on depth discontinuities in the reflection.
- Beside that, dithering makes the memory reads less efficient, due to lowered coherency.
- Deinterleaving the algorithm could be as option to improve efficiency of dithered tracing.

Performance and quality: Hierarchical depth SSR

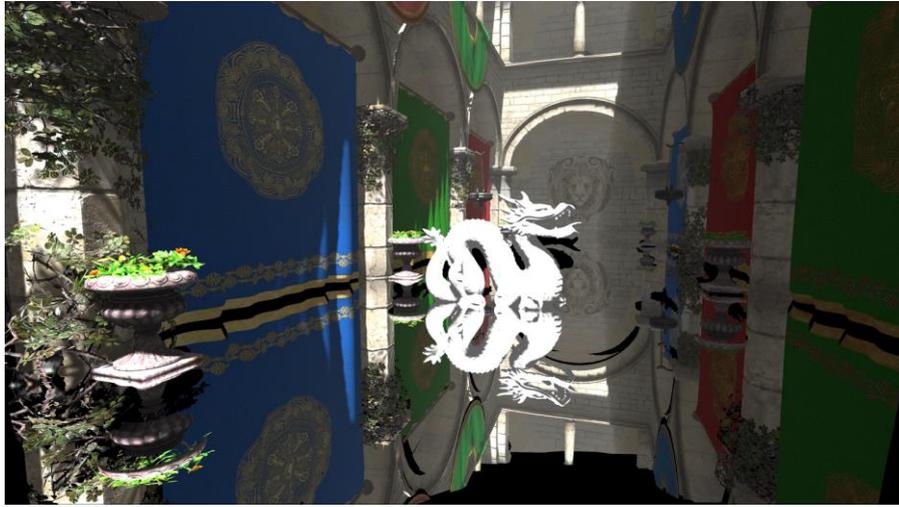


3.2 ms

(+0.35ms for hiZ generation)

- HiZ tracing offers very good quality, and has more reasonable performance than “brute force SSR”. Cost of reflections for this scene is 3.2 ms.
- Generation of hiZ depth buffer additionally takes 0.35 ms of GPU time. This kind of data can be also used by other algorithms in graphics pipeline, so this cost could be shared between them.
- ‘Shadow’ behind the dragon is because measured algorithm implementation doesn't perform tracing behind the geometry, and makes it more efficient.

Performance and quality: Pixel-projected reflections



0.95 ms

- Last measured algorithm is “Pixel-projected reflections”
- Presented technique generates high quality image, comparable to “brute force SSR”.
- Cost of applying PPR for this scene is 0.95 ms.

Make sure to see „Bonus slides”!



- We have reached the end of this talk.
- On the image you can see real-time reflections applied to a glass panel.
Given the high performance of presented technique, achieving this kind of effect is more affordable than with regular raymarching.
- I'd like to encourage you to check the "bonus slides", to get some additional information about "pixel projected reflections".

Summary

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



- As a summary..
- I presented rather simple technique for generating dynamic real-time reflections.
- It's easy to integrate, and offers high quality to cost ratio.
- It's more limited in application than raymarched SSR, but can be selectively used where applicable.

Special thanks

- Natalya Tatarchuk
- Michał Iwanicki
- Tomasz Jonarski
- Havok Germany team

- I'd like to thank Natalya Tatarchuk, Michal Iwanicki, Tomasz Jonarski, and the whole Havok Germany team, for their great feedback and support on the slides.

Thank you!

- Thank you!

References

- **[McGuire14] Morgan McGuire, Michael Mara** "Efficient GPU Screen-Space Ray Tracing"
<http://jcgt.org/published/0003/04/04>
- **[Wronski14] Bart Wronski** "The future of screenspace reflections"
<https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/>
- **[Stachowiak15] Tomasz Stachowiak, Yasin Uludag** "Stochastic Screen-Space Reflections"
<http://advances.realtimerendering.com/s2015/>
- **[Giacalone16] Michele Giacalone** "Screen Space Reflections in The Surge"
<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
- [Giaccone16]



no temporal filter

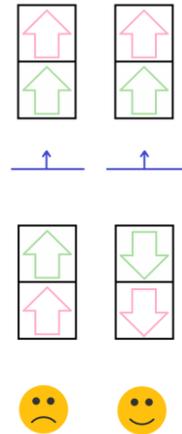
Previous work: Hierarchical raymarching

- 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



See the code snippets for implementation details.

Multiple parallel shapes

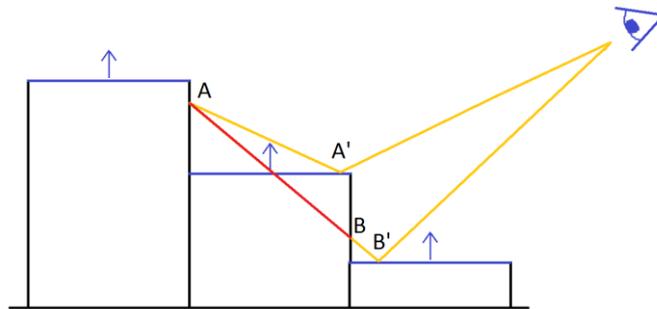
- Find shapes that reflect given pixel
- Write offsets projected onto selected shape



„Multiple parallel shapes” section highlights one nice property of parallel shapes.

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

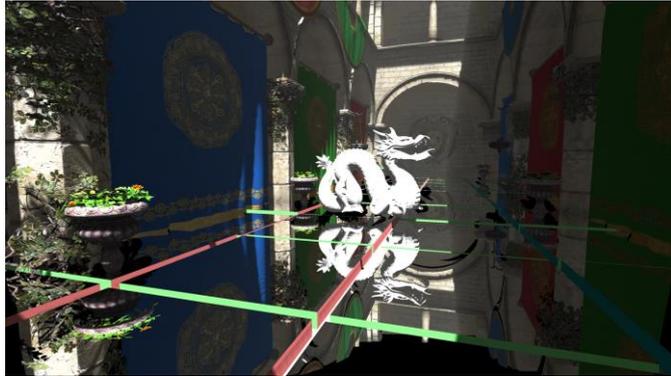
GPU based approach:

- 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



Shader code

```
// 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_DecomposePackingBasisIndexTwoBits( 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 );
}
```

Shader code

```
// 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 );
}
```

Shader code

```
// 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 settFullValueRange = 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 & (settFullValueRange - 1), false );
        float2 y = PPR_UnpackValue( value / settFullValueRange, 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 );
}
```

Shader code

```
// Write projection to 'intermediate buffer'.
// Pixel projected from 'originalPixelVpos' to 'mirroredWorldPos'.
// Function called in 'projectio 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);

        uint originalValue = 0;
        uint 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
}
```

Shader code

```
// 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
```

Shader code

```
// 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 targetCrđ = vpos + 0.5 - predictedDist;
const float2 targetCrđUnfiltered = vpos + 0.5 - predictedDistUnfiltered;
const float3 colorFiltered = srvColor.SampleLevel( smpLinear, targetCrđ * globalConstants.resolution.zw, 0 ).xyz;
const float3 colorUnfiltered = srvColor.SampleLevel( smpPoint, targetCrđUnfiltered * 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 );
}
```

end of file