Dynamic Temporal Antialiasing and Upsampling

in Call of Duty

Jorge Jimenez

Graphics R&D Technical Director - Activision Blizzard

SIGGRAPH Advances in Real-Time Rendering 2017 | Digital Dragons Programming and Technology Track 2018

30 JULY – 3 AUGUST Los Angeles SIGGRAPH2017 AT THE ♥ of COMPUTER GRAPHICS & INTERACTIVE TECHNIQUES

Digital Dragons

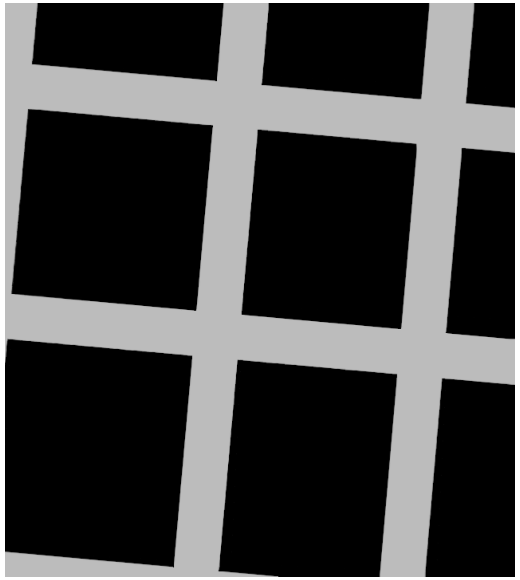
ACTIVISION | BLIZZARD


0.83333
AUTOPILOT: ON / OFF
HOVER
FLIGHT
FLARES
ECM
0
THRUSTERS ENGAGED
THRUSTERS ENGAGED
R-7 SKELTER
4592.4897
30mm Gren
30mm Gren

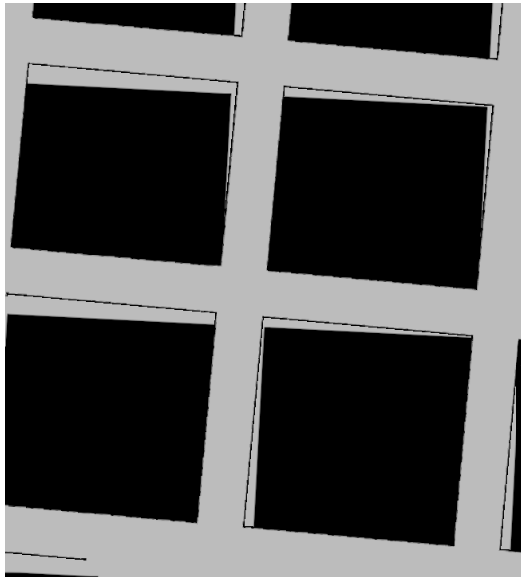

0.83333
AUTOPILOT: ON / OFF
HOVER
FLIGHT
FLARES
ECM
0
THRUSTERS ENGAGED
THRUSTERS ENGAGED
R-7 SKELTER
4579.1851
30mm Gren
30mm Gren
50mm Pathfinder


0.83333
AUTOPILOT: ON / OFF
HOVER
FLIGHT
FLARES
ECM
0
THRUSTERS ENGAGED
THRUSTERS ENGAGED
R-7 SKELTER
4579.1851
30mm Gren
30mm Gren

Reprojection Ghosting

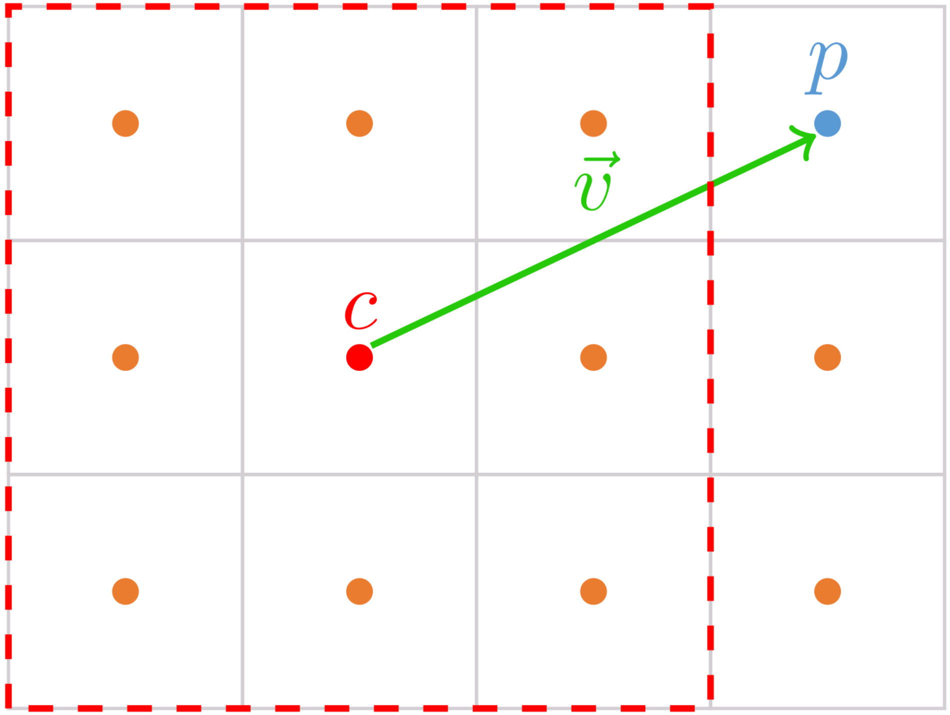


Without
Temporal Supersampling



With
Temporal Supersampling

Neighborhood Clamp [Lottes2011]



Filmic SMAA

- Morphological Component
- Temporal Supersampling Component
- Temporal Filtering Component

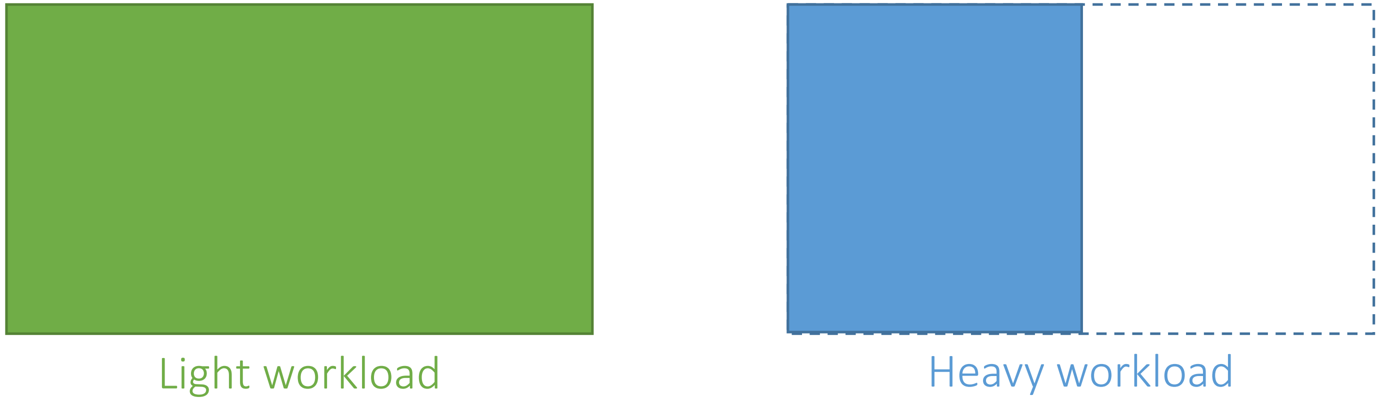
[Jimenez2016] Filmic SMAA: Sharp Morphological and Temporal Antialiasing

Dynamic Resolution and Infrastructure

- Adam Micciulla
- Akimitsu Hogge
- Angelo Pesce
- Michael Vance
- Michal Drobot
- Wade Brainerd

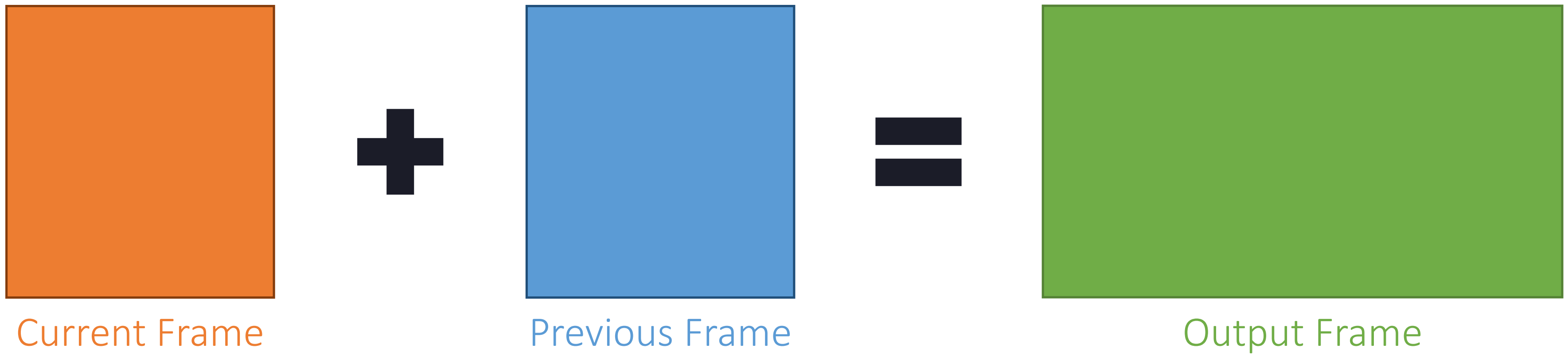
Intro

- Dynamic resolution widely used for 60fps games
  - Change output resolution according to load



Intro

- Temporal upsampling deployed on many PS4 Pro titles
  - Combine previous and current frame for a higher resolution image



Intro

- Dynamic Resolution + Temporal upsampling Problem

Light workload   Heavy workload

Intro

- Dynamic Resolution + Temporal upsampling Problem

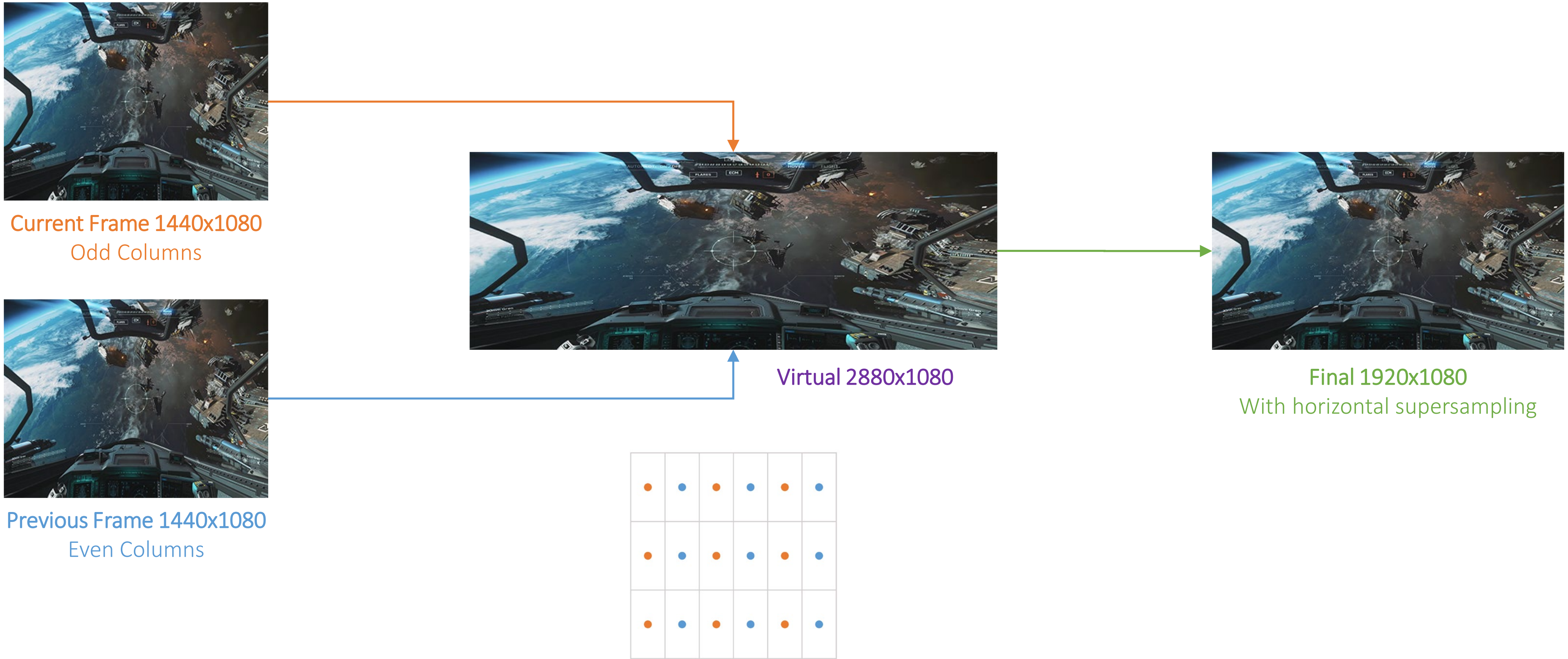
Filmic SMAA TU2X

Dynamic AA 2X

Filmic SMAA TU2x Highlights

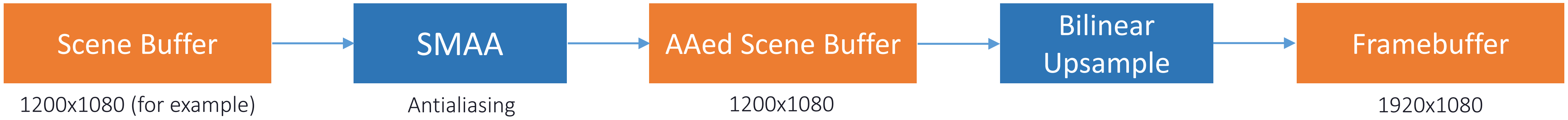
- Dynamic resolution + temporal upsample combo
  - Rather than virtually vary the resolution, vary the antialiasing quality
- Always outputs 1080p
  - On the upper end: 2x supersampling
  - On the lower end: 1x supersampling
  - In between: 1x to 2x supersampling
- Can directly upsample from any resolution to 1080p
  - From [960x1080, 1920x1080]
  - Rather than just from 960x1080

Big Picture


Current Frame 1440x1080
Odd Columns
Previous Frame 1440x1080
Even Columns
Virtual 2880x1080
Final 1920x1080
With horizontal supersampling

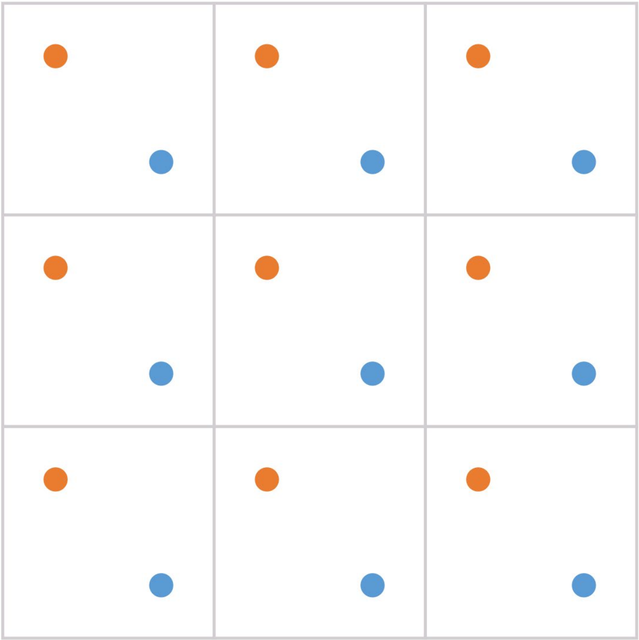
Big Picture

- **Dynamic resolution:** (our previous setup)

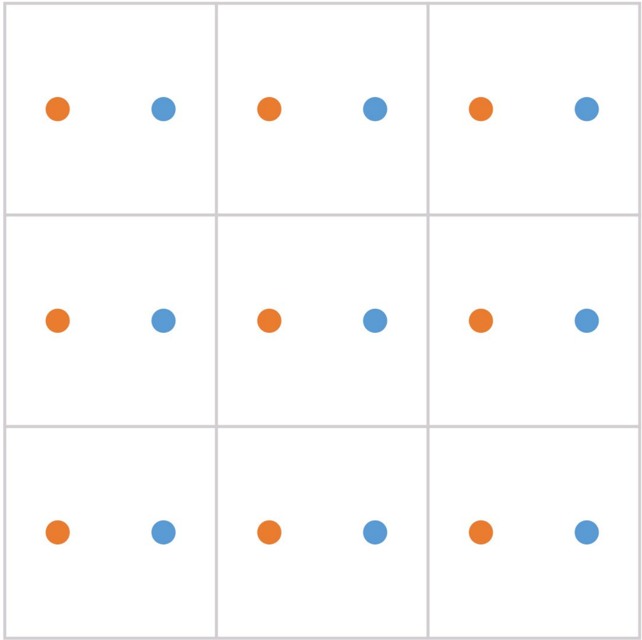


- **Dynamic AA:**

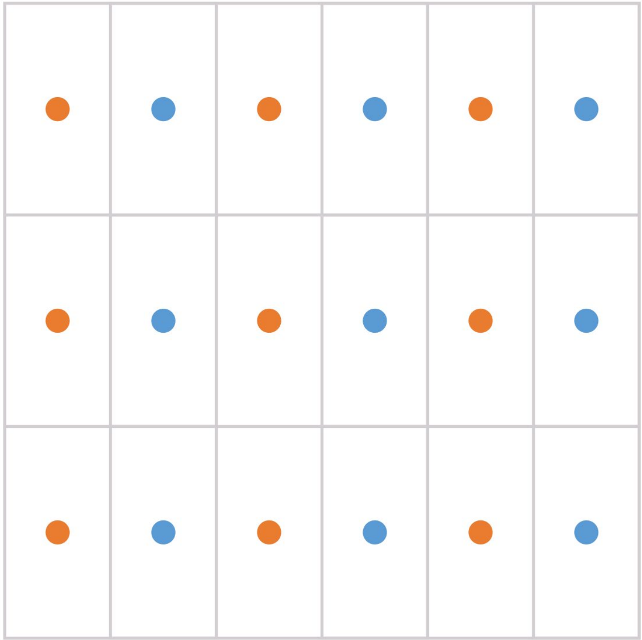
Subpixel Jittering



SMAA T2x
(Default Diagonal Jitter)



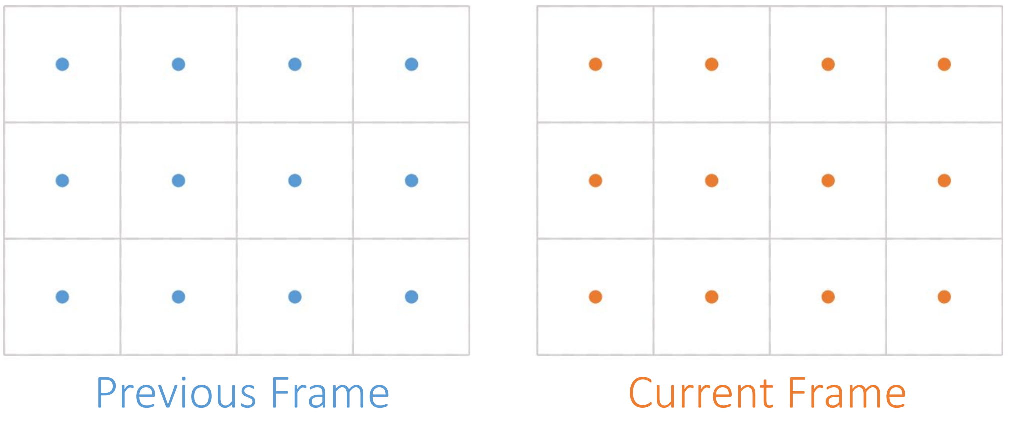
SMAA T2x
(Horizontal Jitter)



New SMAA TU2x Upsampler
Based on [Valient2014]

Virtual Bilinear Downsample

- We start from current and previous frames
- Rendered with different horizontal subpixel offsets



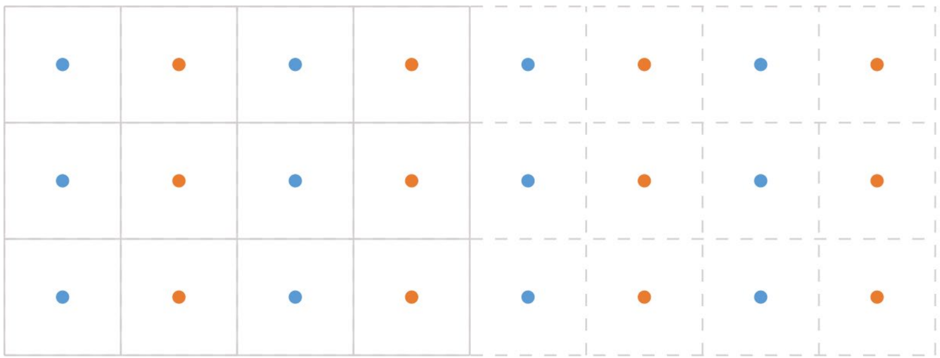
Virtual Bilinear Downsample

- We build a 2x virtual image by interlacing the previous and current frames

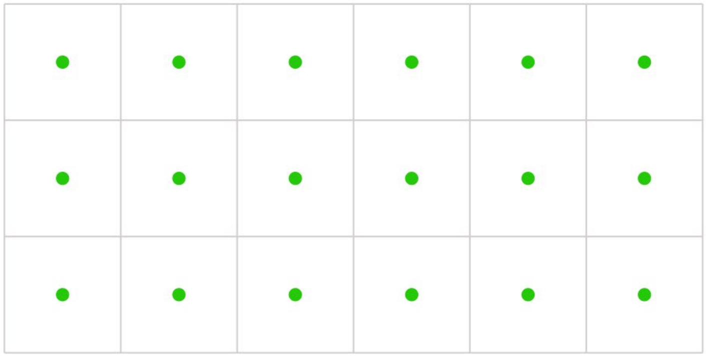
Virtual 2x Image

Virtual Bilinear Downsample

- However our output image is smaller



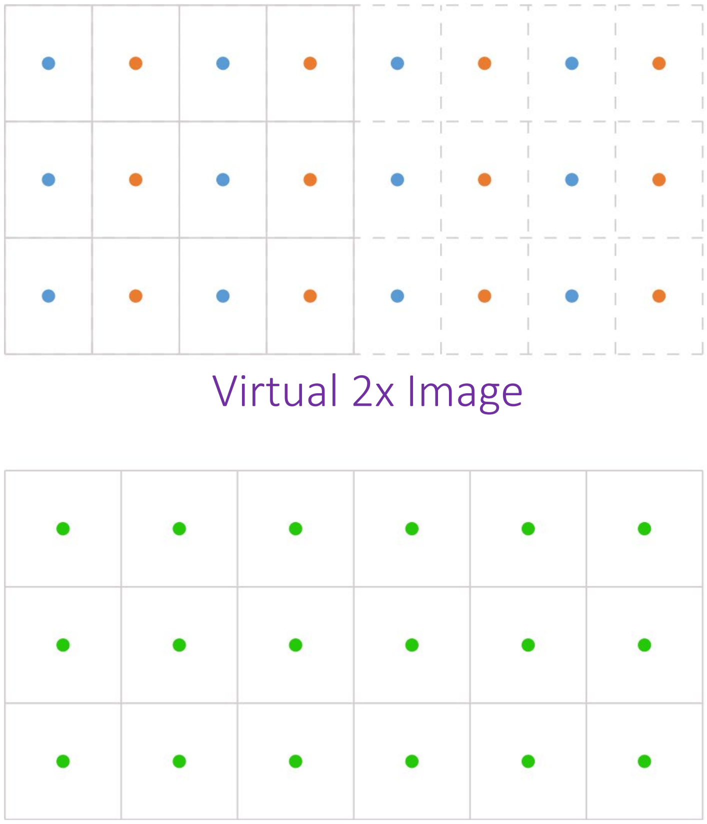
Virtual 2x Image



Output Image

Virtual Bilinear Downsample

- We downsample the virtual 2x image to output resolution

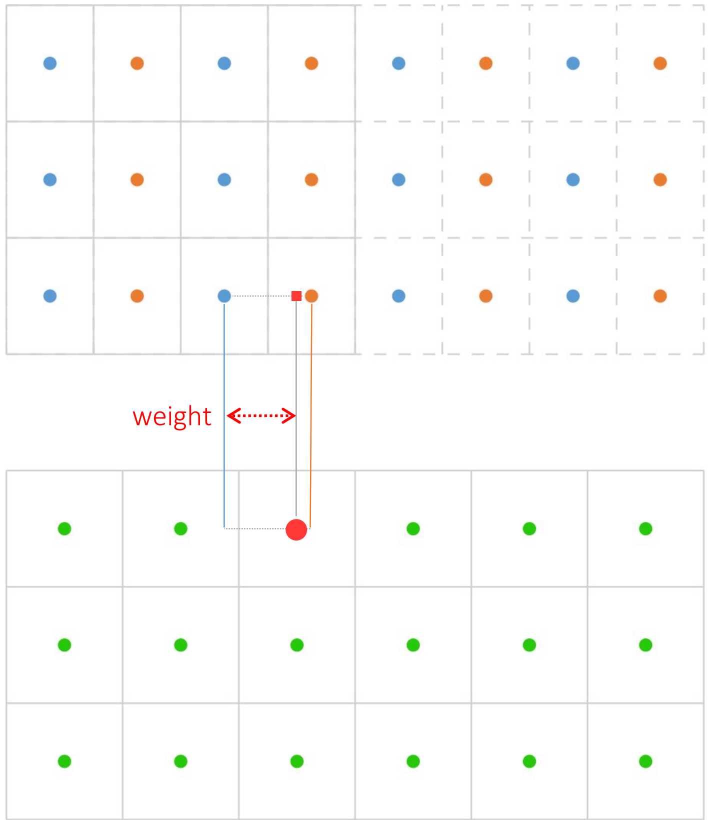


Virtual 2x Image

Output Image

Dynamic AA Algorithm

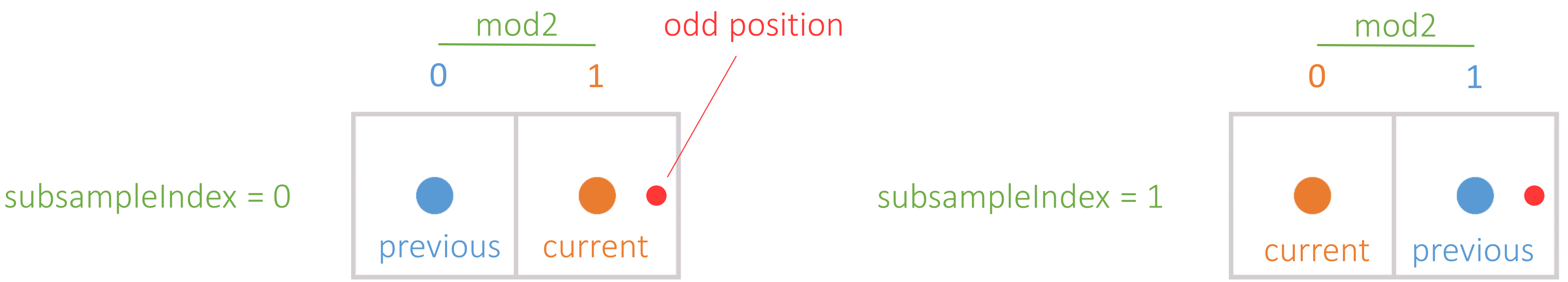
- First step:
  - Calculate position of output image pixel in the virtual 2x image
  - Bilinear weight will be the fractional of this position




```
float upsampledPosition = 2.0 * inputDimensions.x * texcoord . x - 0.5 ;
float weight = frac ( upsampledPosition );
```

Dynamic AA Algorithm

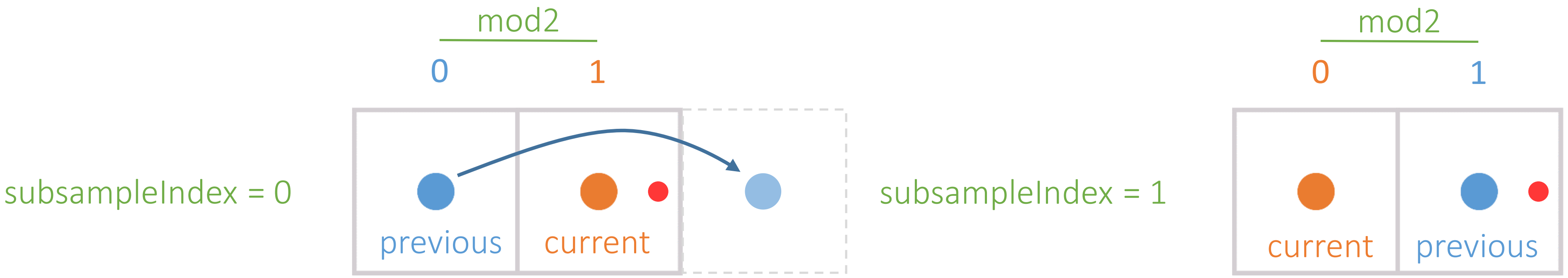
- Second step:
  - Odd positions need to lerp across 2-pixel block boundaries




```
int    mod2 = SMAAMod2(upsampledPosition   );
float    currentOffset      = mod2 &&   subsampleIndex  ? 1.0  :  0.0 ;  // | c | p |      -> | . | p | c |
float    previousOffset     = mod2 && ! subsampleIndex  ? 1.0  :  0.0 ;  // | p | c |      -> | . | c | p |
```

Dynamic AA Algorithm

- Second step:
  - Odd positions need to lerp across 2-pixel block boundaries
  - For them, offset current or previous frame colors




```
int    mod2  =  SMAAMod2(upsampledPosition);
float  currentOffset   = mod2 && subsampleIndex  ? 1.0 : 0.0; // |c|p|  -> |.|p|c|
float  previousOffset  = mod2 && !subsampleIndex ? 1.0 : 0.0; // |p|c|  -> |.|c|p|
```

Dynamic AA Algorithm

- Third step:
  - Apply previously calculated offsets
  - Snap to input texel centers
  - Convert to texture coordinates
  - Beware slightly optimized code below

```
texcoord         . x = (1.0 / inputDimensions.x)   * ( floor  ( 0.5  * ( floor  ( upsampledPosition   ) + currentOffset    ) + 0.25 )  + 0.5 );
previousTexcoord . x = (1.0 / inputDimensions.x)   * ( floor  ( 0.5  * ( floor  ( upsampledPosition   ) + previousOffset   ) + 0.25 )  + 0.5 );
```

Dynamic AA Algorithm

- Fourth step:
  - Blend of current and previous has current on the left
  - If previous frame color is on the left, reverse the weight


```
bool subsampleSwap = SMAAXor( subsampleIndex , mod2);
weight = subsampleSwap ? weight : 1.0 – weight ;
outputColor = lerp ( currentColor , previousColor , weight );
```

Filmic SMAA TU2x Highlights

- Dynamic resolution + temporal upsample combo
  - Rather than virtually vary the resolution, vary the antialiasing quality
- Always outputs 1080p:
  - On the upper end: 2x supersampling
  - On the lower end: 1x supersampling
  - In between: 1x to 2x supersampling
- Can directly upsample from any resolution to 1080p
  - From [960x1080, 1920x1080]
  - Rather than just from 960x1080

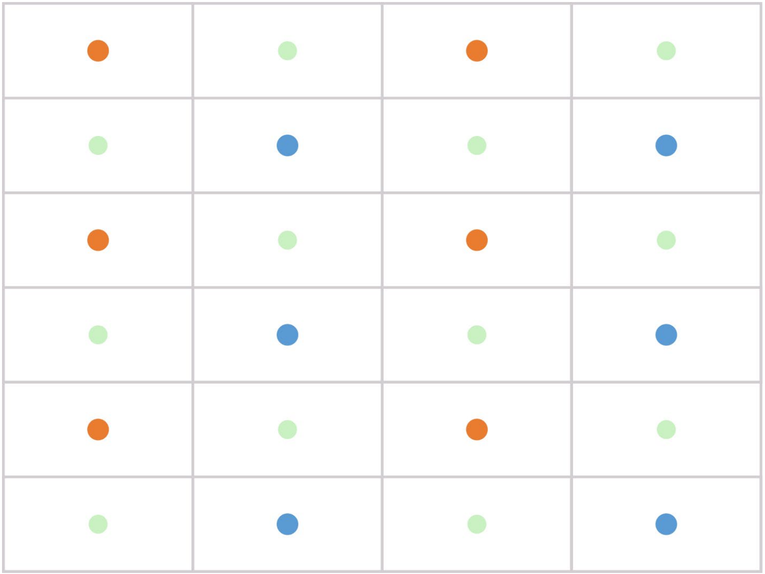
Filmic SMAA TU4X

Dynamic AA 4X

Filmic SMAA TU4x

- Presented method so far can do up to 2x upsampling
- We have an experimental method that can do up to 4x upsampling

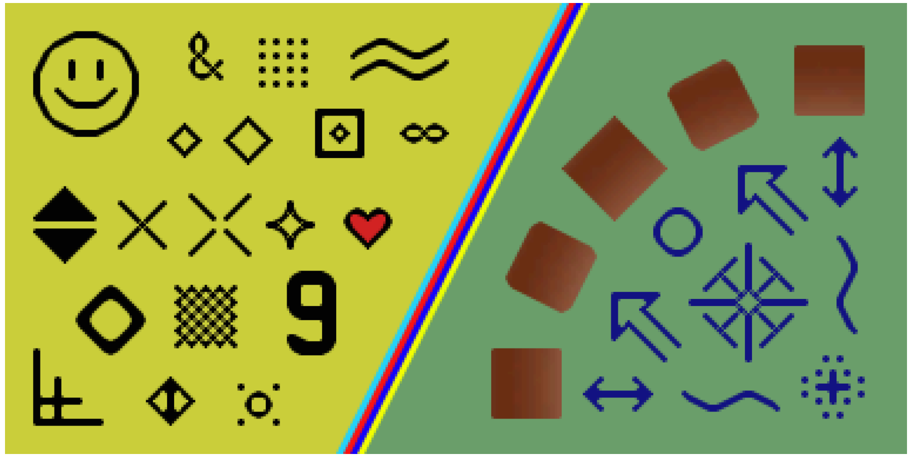
- Uses diagonal jitter rather than horizontal:
  - Current frame
  - Previous frame
- Reconstructs:
  - Missing samples from orange and blue samples



Filmic SMAA TU4x

- Pixel art upsampling algorithms have been doing similar reconstructions (hqx)
  - Core difference is that the input here is checkerboarded => Easier reconstruction
- Similar problem to demosaicing [Phelippeau2009]
- We extended the highly efficient [Berghoff2016] differential blend to a temporal checkerboard

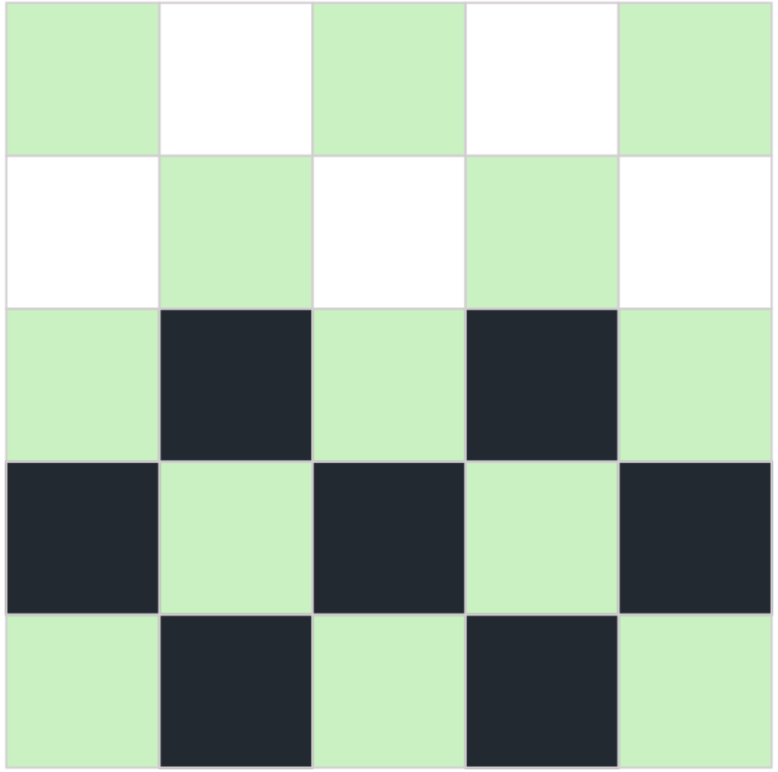
HQX Input



HQX Output

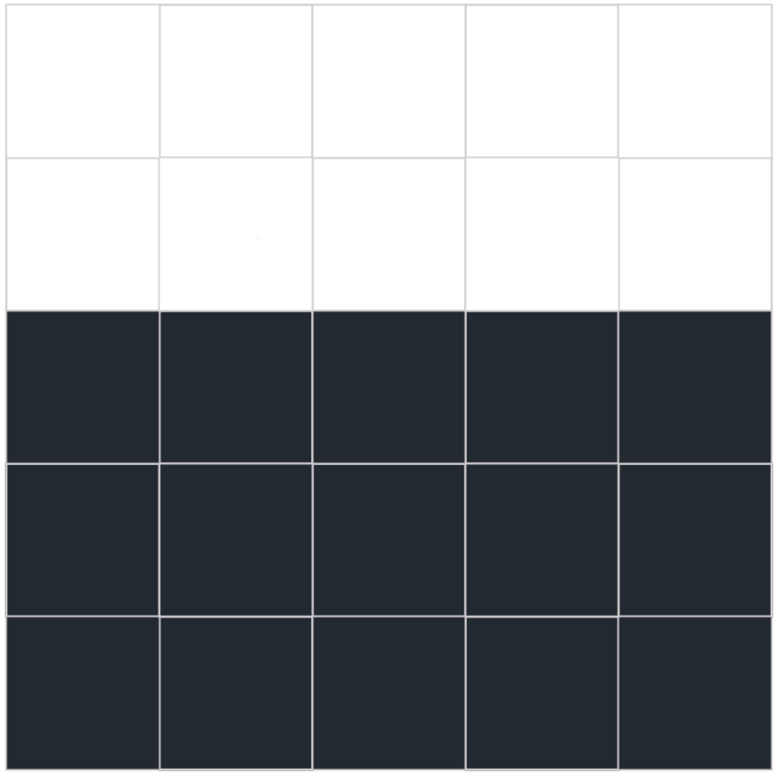
[Berghoff2016] Differential Blend Recap

- This is the checkerboarded input
- Green pixels are missing



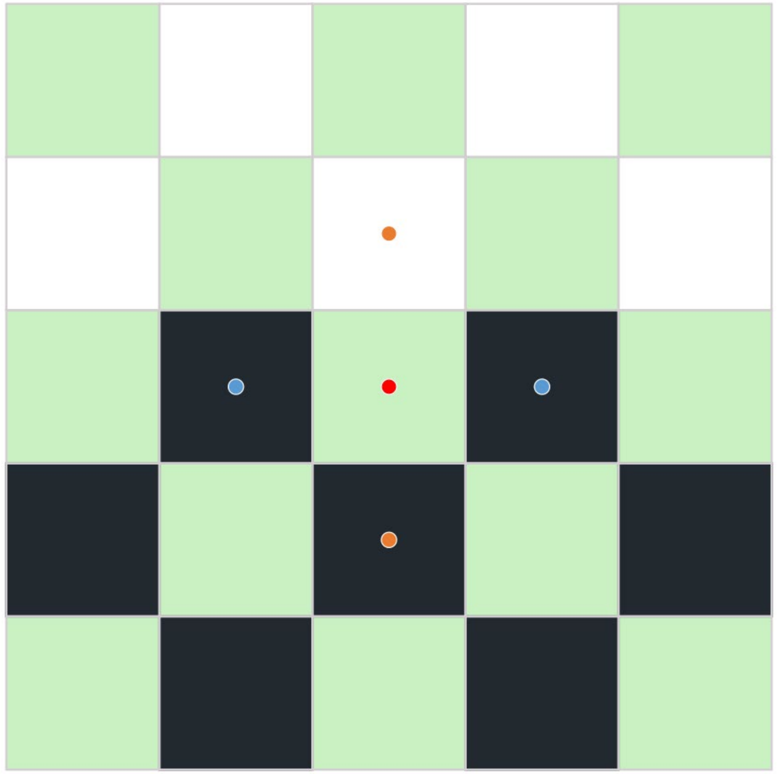
[Berghoff2016] Differential Blend Recap

- This is the target ideal reconstruction



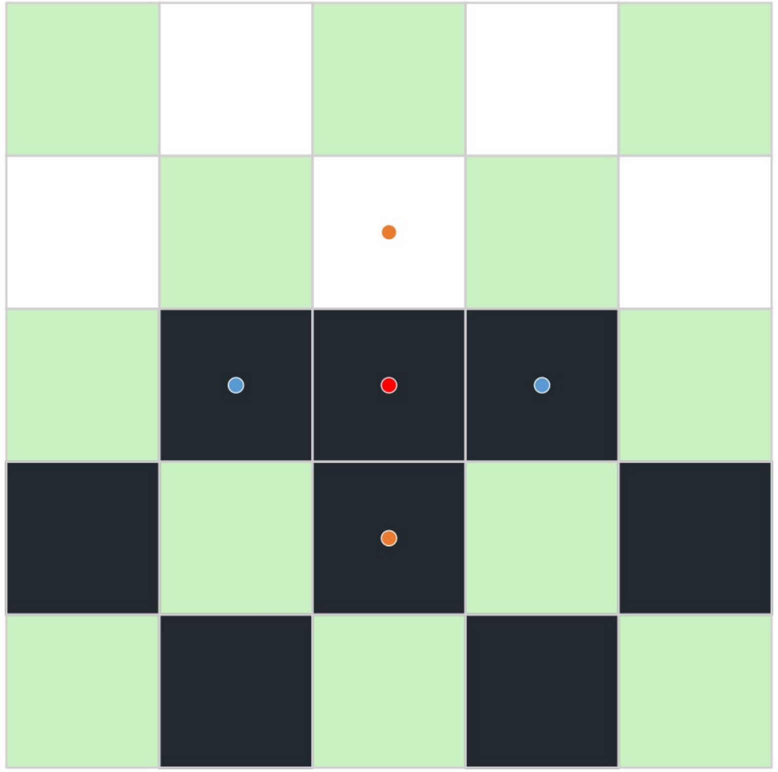
[Berghoff2016] Differential Blend Recap

- Determine color of current pixel by checking horizontal and vertical neighbor blends
- Keep the neighbor blend with the lowest color **difference**

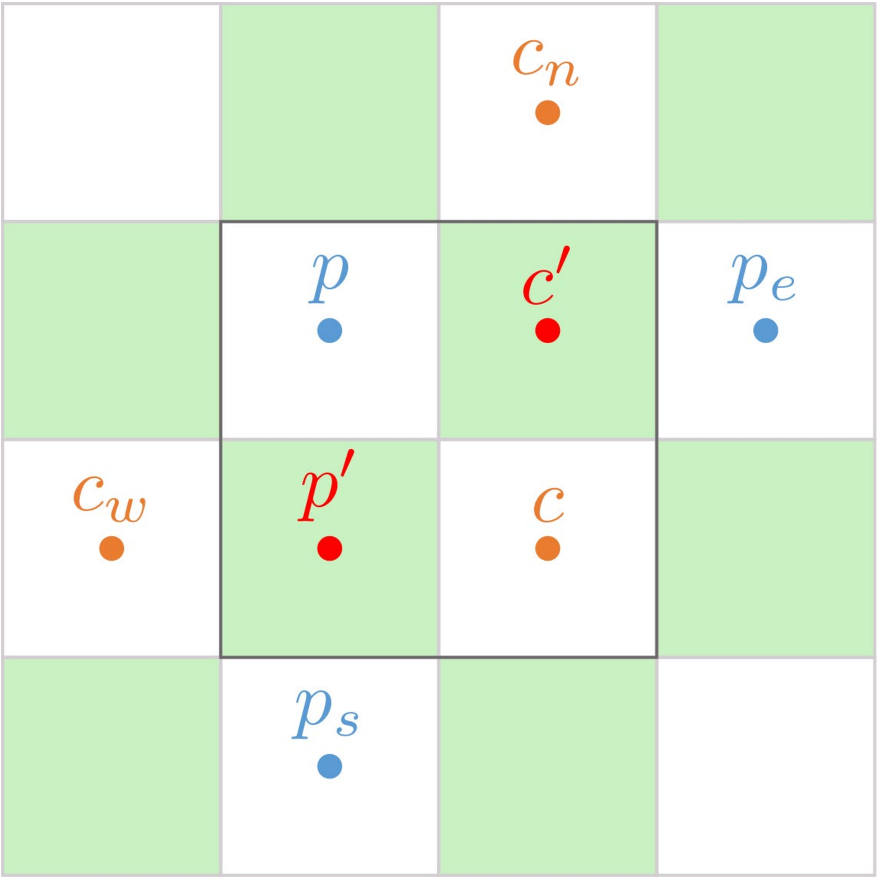


[Berghoff2016] Differential Blend Recap

- Determine color of current pixel by checking horizontal and vertical neighbor blends
- Keep the neighbor blend with the lowest color **difference**
- For the current pixel, that would be the horizontal blend



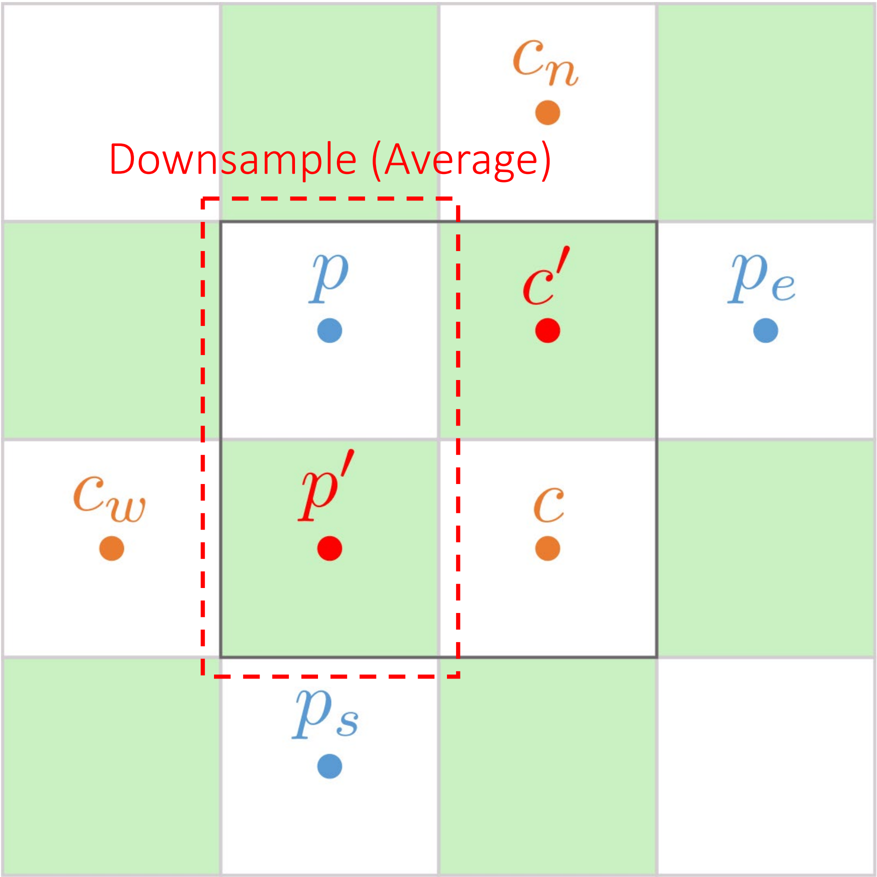
Temporal Checkerboard




```
float3  neighborhood1[4] = {
    currentNeighborhood[SMAA_NEIGHBORHOOD_WEST],   // SMAA_NEIGHBORHOOD_WEST
    currentColor,                                  // SMAA_NEIGHBORHOOD_EAST
    previousColor,                                 // SMAA_NEIGHBORHOOD_NORTH
    previousNeighborhood[SMAA_NEIGHBORHOOD_SOUTH]  // SMAA_NEIGHBORHOOD_SOUTH
};

float3 neighborhood2[4] = {
    previousColor,                                 // SMAA_NEIGHBORHOOD_WEST
    previousNeighborhood[SMAA_NEIGHBORHOOD_EAST],  // SMAA_NEIGHBORHOOD_EAST
    currentNeighborhood[SMAA_NEIGHBORHOOD_NORTH],  // SMAA_NEIGHBORHOOD_NORTH
    currentColor                                   // SMAA_NEIGHBORHOOD_SOUTH
};

float3 weights = SMAADifferentialBlendCalculateWeight(neighborhood1, neighborhood2);
float3 previousReconstructedColor = SMAADifferentialBlend(neighborhood1, weights); // p'
float3 currentReconstructedColor  = SMAADifferentialBlend(neighborhood2, weights); // c‘

previousColor = lerp(previousColor, previousReconstructedColor, 0.5);
currentColor  = lerp(currentColor,  currentReconstructedColor,  0.5);
```

```
float3 SMAADifferentialBlend(float3 neighborhood[4], float3 weights)
{
    float4 color = 0.0;
    color += float4(neighborhood[SMAA_NEIGHBORHOOD_WEST]  + neighborhood[SMAA_NEIGHBORHOOD_EAST],  1.0) * weights.x;
    color += float4(neighborhood[SMAA_NEIGHBORHOOD_NORTH] + neighborhood[SMAA_NEIGHBORHOOD_SOUTH], 1.0) * weights.y;
    return (0.5 * weights.z) * color.rgb;
}
```

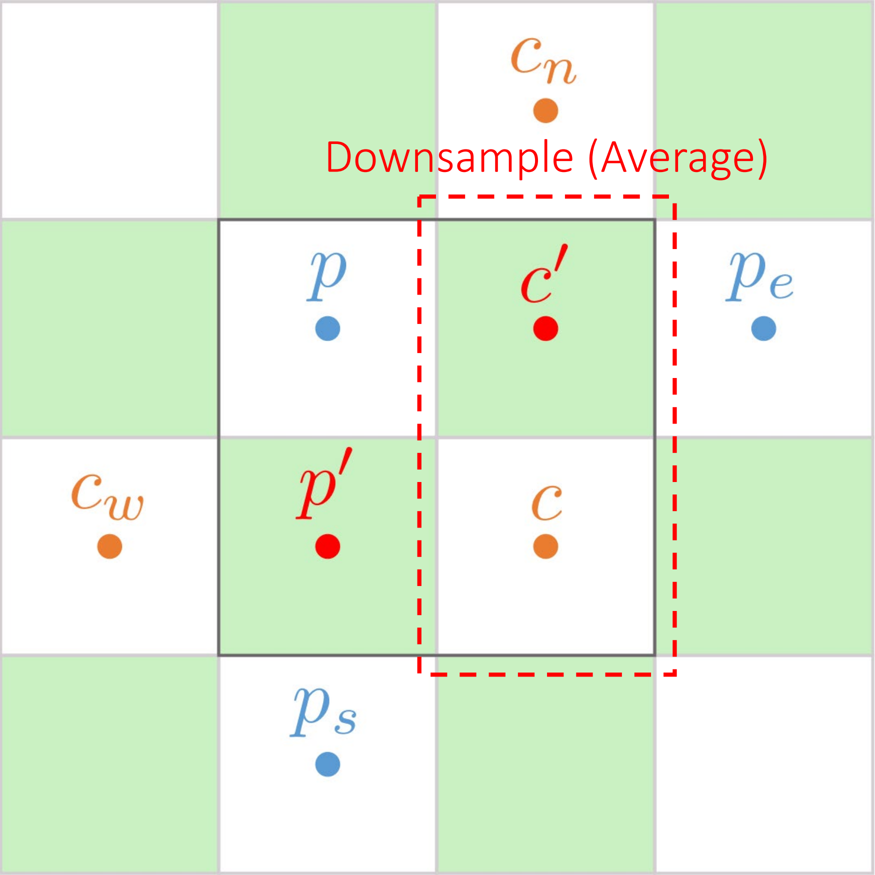
Temporal Checkerboard



Filmic SMAA TU2x
960x1080 to 1920x1080

Filmic SMAA TU4x
960x1080 to 1920x1080 + 2xAA

Temporal Checkerboard



Filmic SMAA TU2x
960x1080 to 1920x1080

Filmic SMAA TU4x
960x1080 to 1920x1080 + 2xAA

[Berghoff2016] Checkerboard

- Algorithm:
  - Render to EQAA spatial checkboard
  - Fills checkerboard with temporal information
  - If it fails, uses differential blend over the spatial checkerboard to fill missing pixels
  - Temporal and spatial reconstruction overlap, spatial one backs up temporal one
  - **2x output**
- Our approach:
  - Render temporal checkerboard
  - Upsample horizontally using temporal information
  - Upsample vertically using spatial checkerboard reconstruction
  - **4x output**

960x1080 + FXAA

960x1080 + Filmic SMAA TU4x

1440x1080 + Filmic SMAA TU4x

1920x1080 + Filmic SMAA T2x

960x1080 + FXAA

960x1080 + Filmic SMAA TU4x

1440x1080 + Filmic SMAA TU4x

1920x1080 + Filmic SMAA T2x

960x1080 + FXAA

960x1080 + Filmic SMAA TU4x

1440x1080 + Filmic SMAA TU4x

1920x1080 + Filmic SMAA T2x

960x1080 + FXAA

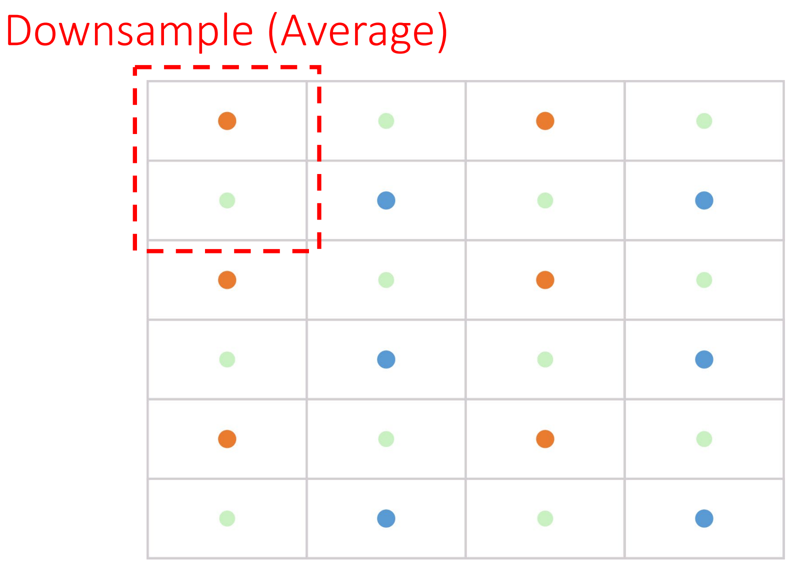
960x1080 + Filmic SMAA TU4x

1440x1080 + Filmic SMAA TU4x

1920x1080 + Filmic SMAA T2x

4x Upsampler

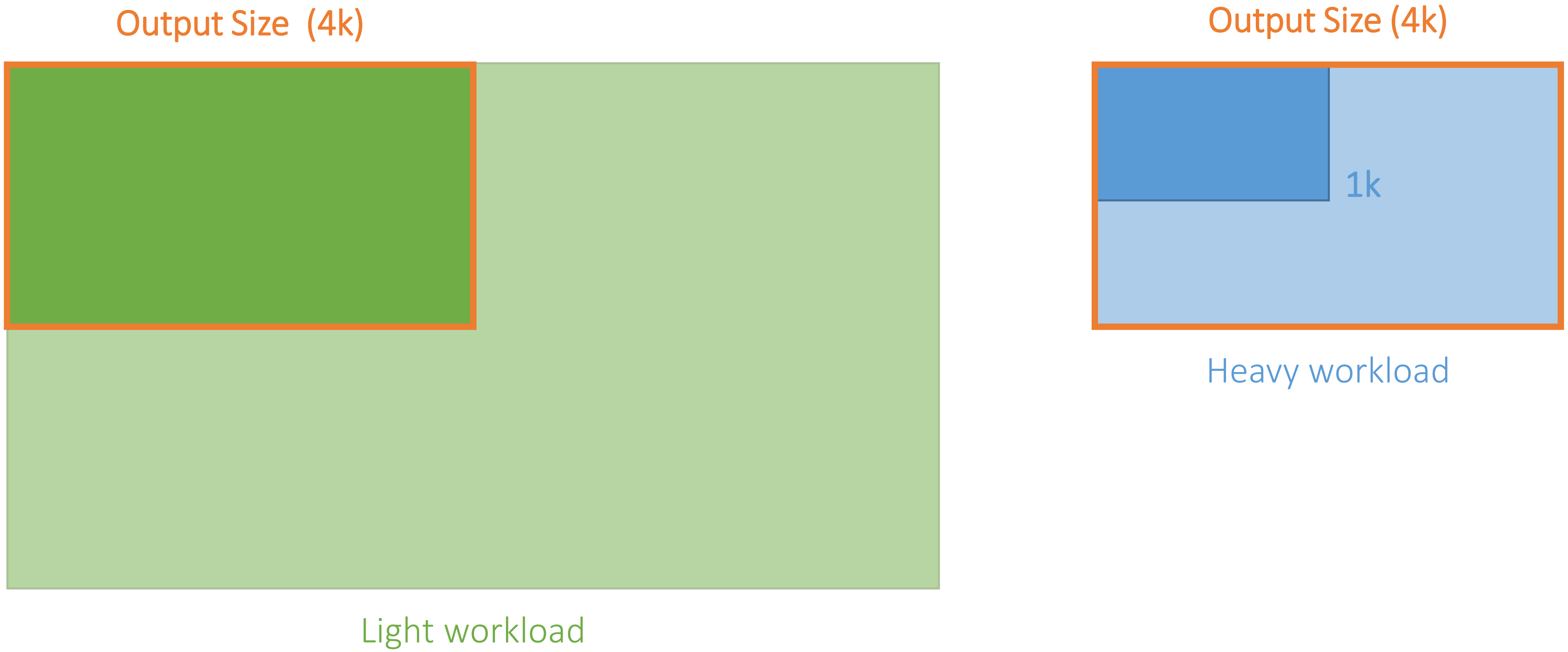
- **So far used for 2x vertical AA**
  - Horizontal is dynamic bilinear as Dynamic AA 2x
  - Fixed vertical bilinear downsample
- **Full 4x output**
  - Doubles state-of-the-art upsampler capabilities
  - Dynamic AA from 1920x1080 to 3840x2160 (with no bilinear upscale involved)
  - Full 4k pixels regardless of input resolution
    - 2k texture details when starting from quarter res
    - 4k texture details when starting from half res



Dynamic Resolution

2x Upsampler

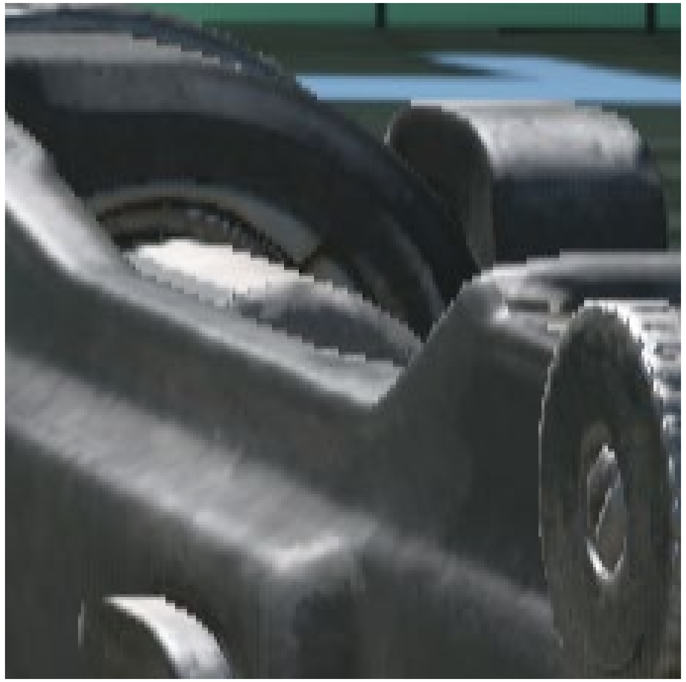
4x Upsampler



Results

No upsampling

Results



Filmic SMAA TU2x

(Morphological removed to see upsampling effect in isolation)

Results

Filmic SMAA TU4x

(Morphological removed to see upsampling effect in isolation)

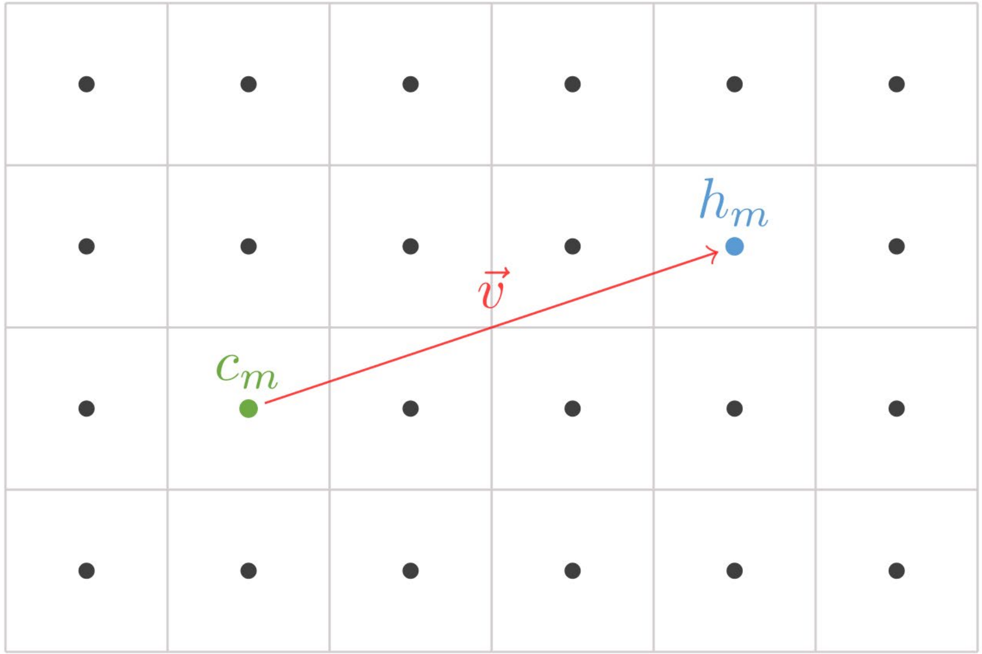
Results

No upsampling

New Temporal Toolset

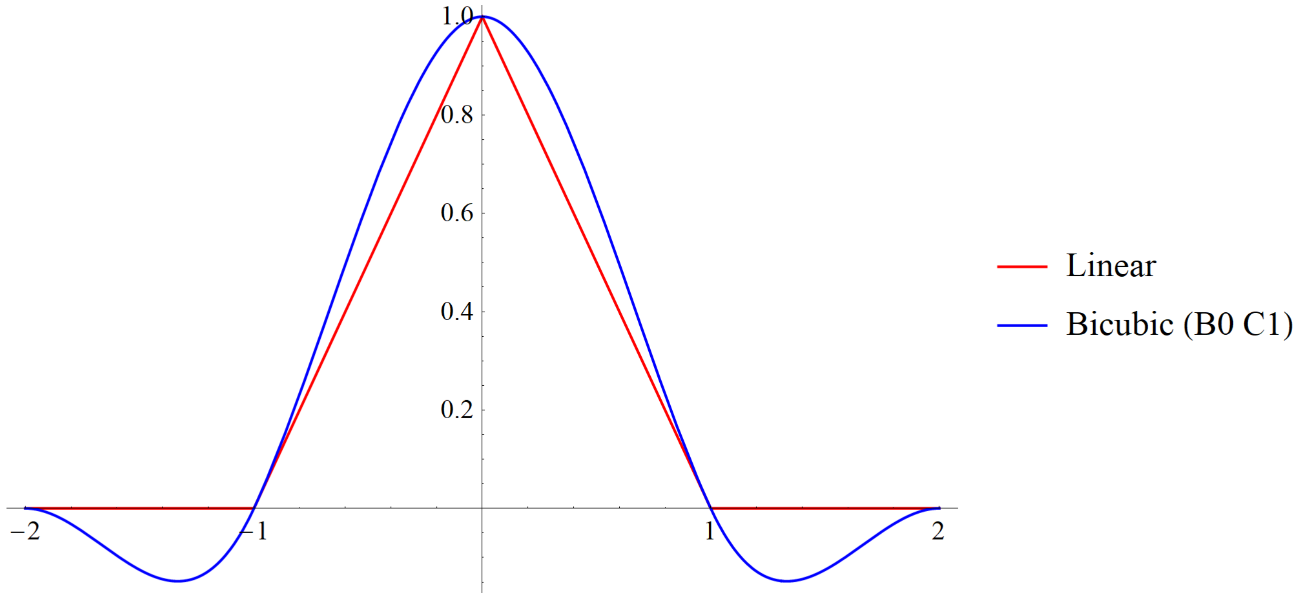
1-Sample Spatio-Temporal Bicubic Resampling Intro

- History buffer resampling leads to numerical diffusion error
  - Manifests as blur



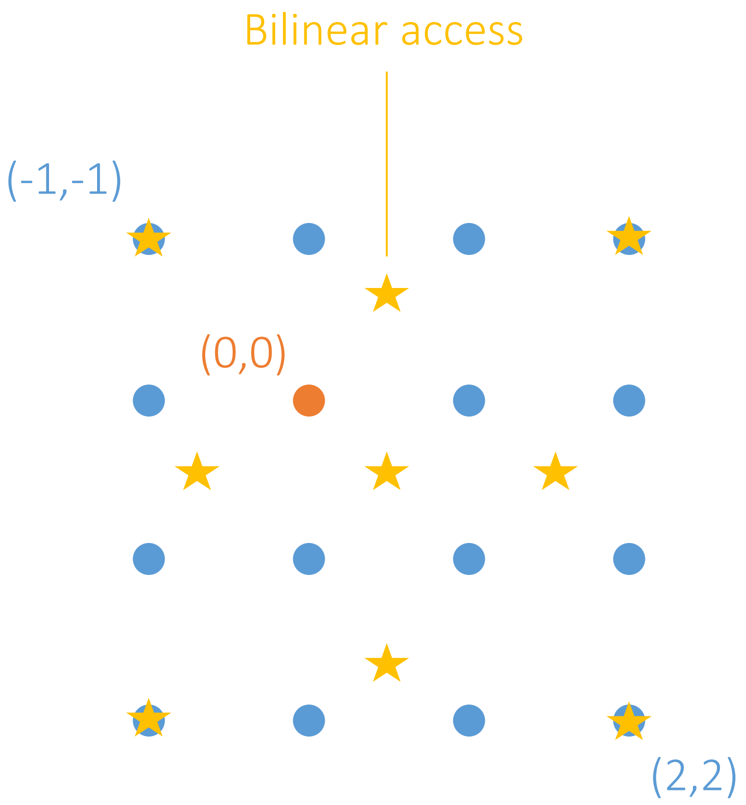
1-Sample Spatio-Temporal Bicubic Resampling Intro

- History buffer resampling leads to numerical diffusion error
  - Manifests as blur
- Bicubic filtering mitigates this problem [Jimenez2016]



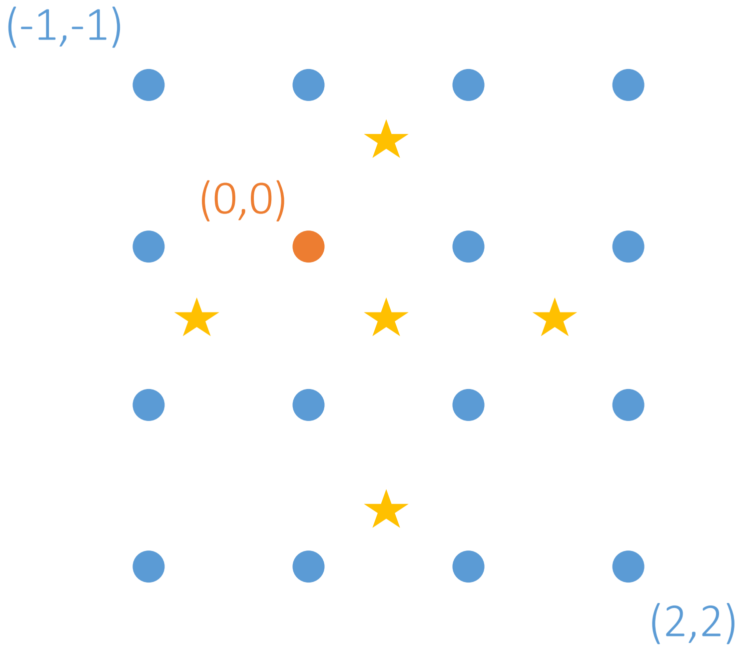
5-Sample Bicubic Resampling [Jimenez2016]

- Optimized Catmull-Rom uses 9 bilinear samples to filter the 4x4 area
  - http://vec3.ca/bicubic-filtering-in-fewer-taps/
  - http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter20.html



5-Sample Bicubic Resampling [Jimenez2016]

- Ignoring the 4 corners yields very similar results
- Reduces from 9 to 5 samples



Bicubic Resampling

- Mitchell-Netravali bicubic equation is computationally expensive:

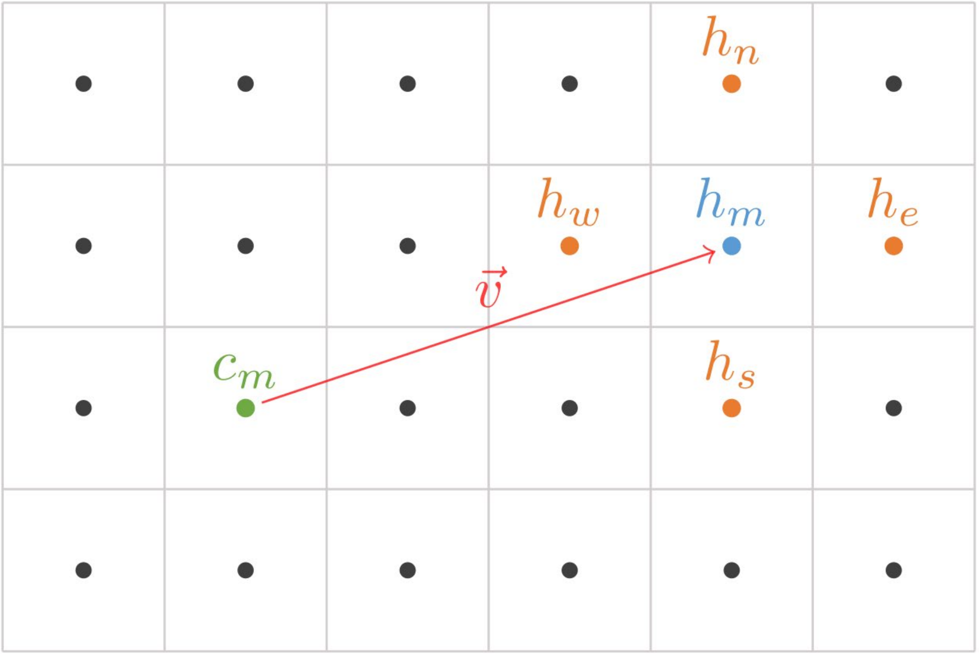
$$f(x) = \begin{cases} (12 - 9B - 6C)|x|^3 + (-18 + 12B + 6C)|x|^2 + (6 - 2B), |x| < 1 \\ (-B - 6C)|x|^3 + (6B + 30C)|x|^2 + (-12B - 48C)x + (8B + 24C), 1 \le |x| \le 2 \\ 0, otherwise \end{cases}$$

- Temporal effects are more forgiving than spatial ones with respect to quality

- What is really needed for temporal resampling?
  - Spatio-Temporal Optimization
  - Computation Optimization

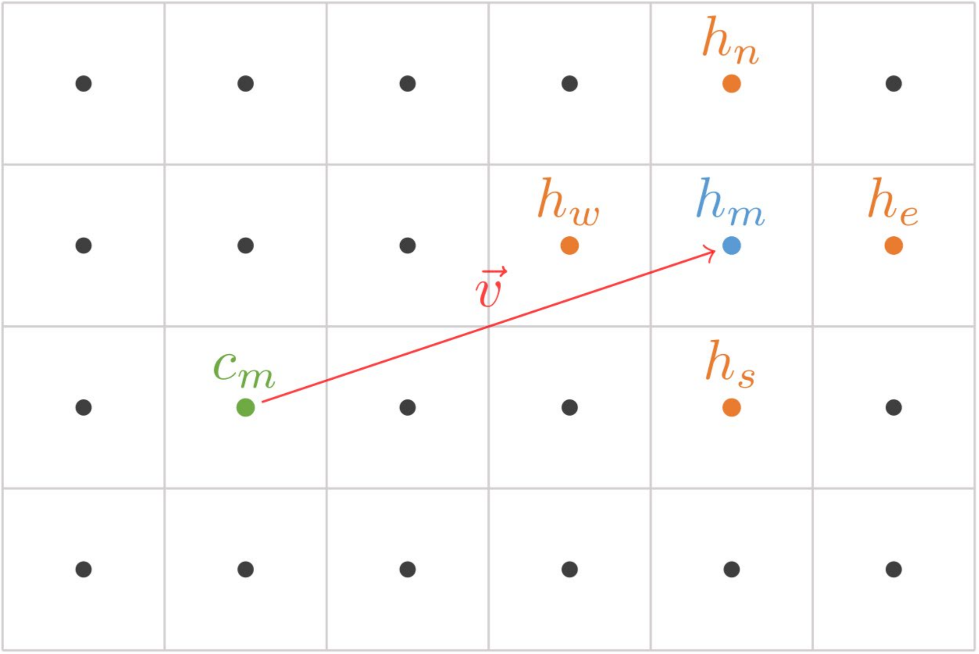
Spatio-Temporal Optimization

- The color information for the 5-sample bicubic is the following:
  - History color at texture coordinate
    - h_m
  - History color neighborhood around texture coordinate
    - h_w
    - h_e
    - h_n
    - h_s



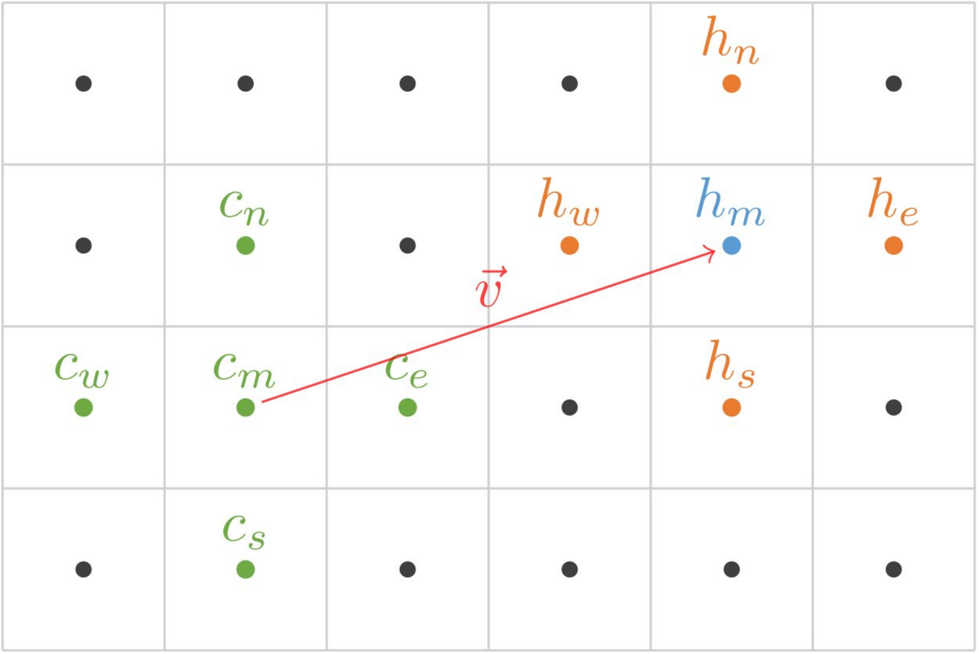
Spatio-Temporal Optimization

- The color information for the 5-sample bicubic is the following:
    - History color at texture coordinate
        - h_m
    - History color neighborhood around texture coordinate
        - h_w
        - h_e
        - h_n
        - h_s
- **Idea:** perform bicubic filtering across time



Spatio-Temporal Optimization

- Estimate history neighborhood colors h with current frame colors c:
  - $h_w \approx h_m + (c_w - c_m)$
  - $h_e \approx h_m + (c_e - c_m)$
  - $h_n \approx h_m + (c_n - c_m)$
  - $h_s \approx h_m + (c_s - c_m)$
- We already have them for the neighborhood clamp
- Very good match if reprojecting inside of an object
- On edges they will be different
  - Slightly reintroduces some aliasing
  - But much sharper details
  - It actually brings back real details rather than sharpening the history buffer
- **1-sample spatio-temporal bicubic -> we just sample h_m**



Computation Optimization

```
float3 SMAABicubicFilter(SMAATexture2D colorTex, float2 texcoord, float4 rtMetrics)
{
    float2 position = rtMetrics.zw * texcoord;
    float2 centerPosition = floor(position - 0.5) + 0.5;
    float2 f = position - centerPosition;
    float2 f2 = f * f;
    float2 f3 = f * f2;

    float c = SMAA_FILMIC_REPROJECTION_SHARPNESS / 100.0;
    float2 w0 =        -c  * f3 +  2.0 * c         * f2 - c * f;
    float2 w1 =  (2.0 - c) * f3 - (3.0 - c)        * f2         + 1.0;
    float2 w2 = -(2.0 - c) * f3 + (3.0 -  2.0 * c) * f2 + c * f;
    float2 w3 =          c  * f3 -                c * f2;

    float2 w12 = w1 + w2;
    float2 tc12 = rtMetrics.xy * (centerPosition + w2 / w12);
    float3 centerColor = SMAASample(colorTex, float2(tc12.x, tc12.y)).rgb;

    float2 tc0 = rtMetrics.xy * (centerPosition - 1.0);
    float2 tc3 = rtMetrics.xy * (centerPosition + 2.0);
    float4 color = float4(SMAASample(colorTex, float2(tc12.x, tc0.y )).rgb, 1.0) * (w12.x * w0.y ) +
                   float4(SMAASample(colorTex, float2(tc0.x,  tc12.y)).rgb, 1.0) * (w0.x  * w12.y) +
                   float4(centerColor,                                      1.0) * (w12.x * w12.y) +
                   float4(SMAASample(colorTex, float2(tc3.x,  tc12.y)).rgb, 1.0) * (w3.x  * w12.y) +
                   float4(SMAASample(colorTex, float2(tc12.x, tc3.y )).rgb, 1.0) * (w12.x * w3.y );
    return color.rgb * rcp(color.a);
}
```

[Jimenez2016]

Computation Optimization

```
float3 SMAABicubicFilter(SMAATexture2D colorTex, float2 texcoord, float4 rtMetrics)
{
    float2 position = rtMetrics.zw * texcoord;
    float2 centerPosition = floor(position - 0.5) + 0.5;
    float2 f = position - centerPosition;
    float2 f2 = f * f;
    float2 f3 = f * f2;

    float c = SMAA_FILMIC_REPROJECTION_SHARPNESS / 100.0;
    float2 w0 =        -c   * f3 +  2.0 * c        * f2 - c * f;
    float2 w1 =  (2.0 - c) * f3 - (3.0 - c)        * f2         + 1.0;
    float2 w2 = -(2.0 - c) * f3 + (3.0 -  2.0 * c) * f2 + c * f;
    float2 w3 =         c   * f3 -              c * f2;

    float2 w12 = w1 + w2;
    float2 tc12 = rtMetrics.xy * (centerPosition + w2 / w12);
    float3 centerColor = SMAASample(colorTex, float2(tc12.x, tc12.y)).rgb;

    float2 tc0 = rtMetrics.xy * (centerPosition - 1.0);
    float2 tc3 = rtMetrics.xy * (centerPosition + 2.0);
    float4 color = float4(SMAASample(colorTex, float2(tc12.x, tc0.y )).rgb, 1.0) * (w12.x * w0.y ) +
                   float4(SMAASample(colorTex, float2(tc0.x,  tc12.y)).rgb, 1.0) * (w0.x  * w12.y) +
                   float4(centerColor,                                  1.0) * (w12.x * w12.y) +
                   float4(SMAASample(colorTex, float2(tc3.x,  tc12.y)).rgb, 1.0) * (w3.x  * w12.y) +
                   float4(SMAASample(colorTex, float2(tc12.x, tc3.y )).rgb, 1.0) * (w12.x * w3.y );
    return color.rgb * rcp(color.a);
}
```

[Jimenez2016]

Computation Optimization

```
float4    color = float4(SMAASample(colorTex, float2(tc12.x, tc0.y )).rgb, 1.0) * (wl2.x * w0.y ) +
                  float4(SMAASample(colorTex, float2(tc0.x,  tc12.y)).rgb, 1.0) * (w0.x  * wl2.y) +
                  float4(centerColor,                                   1.0) * (wl2.x * wl2.y) +
                  float4(SMAASample(colorTex, float2(tc3.x,  tc12.y)).rgb, 1.0) * (w3.x  * wl2.y) +
                  float4(SMAASample(colorTex, float2(tc12.x, tc3.y )).rgb, 1.0) * (wl2.x * w3.y );
                  return color.rgb * rcp(color.a);
```

Computation Optimization

```
float4 color =
        float4(SMAASample(colorTex, float2(tc0.x, tc12.y)).rgb, 1.0) * w0.x  +
        float4(centerColor,                                      1.0) * w12.x +
        float4(SMAASample(colorTex, float2(tc3.x, tc12.y)).rgb, 1.0) * w3.x  +

        return color.rgb * rcp(color.a);
```

Computation Optimization

```
float4 color =
        float4(SMAASample(colorTex, float2(tc0.x, tc12.y)).rgb, 1.0) * (w0.x  / w12.x) +
        float4(centerColor,                                      1.0) * (w12.x / w12.x) +
        float4(SMAASample(colorTex, float2(tc3.x, tc12.y)).rgb, 1.0) * (w3.x  / w12.x) +

        return color.rgb * rcp(color.a);
```

Computation Optimization

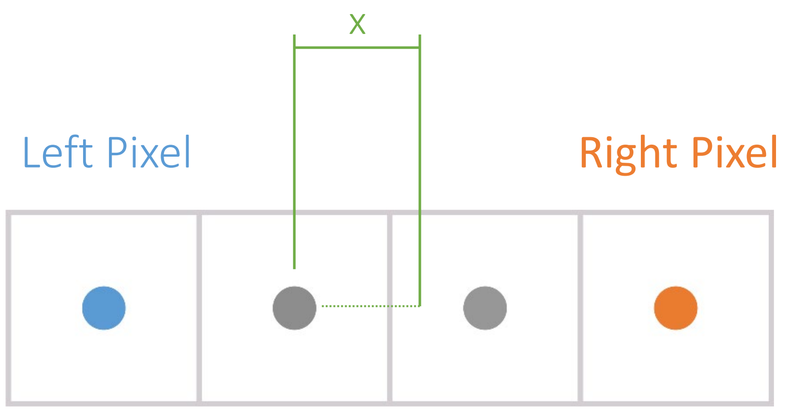
```
float4  color =
        float4(SMAASample(colorTex, float2(tc0.x,  tc12.y)).rgb, 1.0) * (w0.x /  w12.x) +
        float4(centerColor,                                      1.0) * 1.0 +
        float4(SMAASample(colorTex, float2(tc3.x,  tc12.y)).rgb, 1.0) * (w3.x /  w12.x) +

        return color.rgb * rcp(color.a);
```

Computation Optimization

```
float4  color =
        float4(SMAASample(colorTex, float2(tc0.x,  tc12.y)).rgb, 1.0) * (w0.x / w12.x) +
        float4(centerColor,                                     1.0) * 1.0 +
        float4(SMAASample(colorTex, float2(tc3.x,  tc12.y)).rgb, 1.0) * (w3.x / w12.x) +

        return color.rgb * rcp(color.a);
```

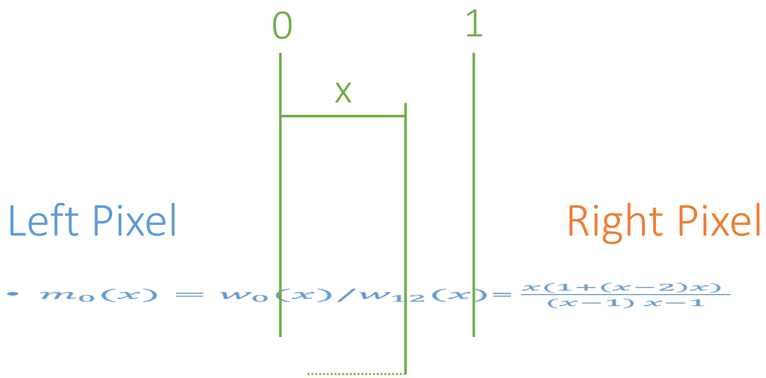
Computation Optimization

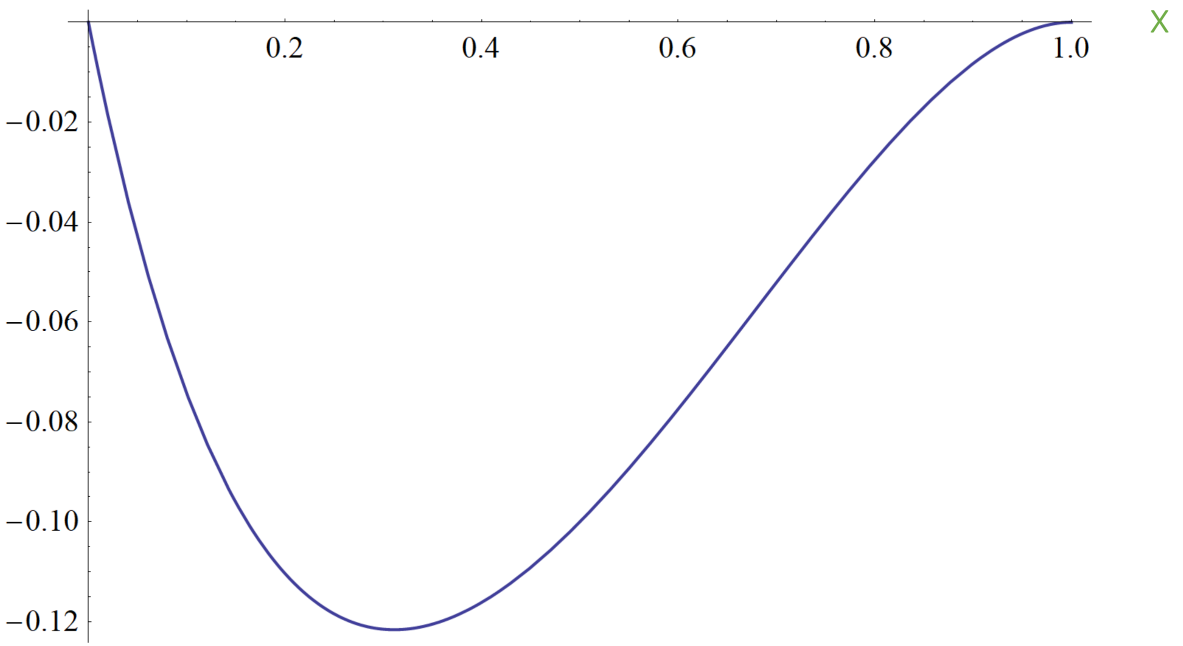
```
float4  color =
        float4(SMAASample(colorTex, float2(tc0.x,  tc12.y)).rgb, 1.0) * (w0.x / w12.x) +
        float4(centerColor,                                      1.0) * 1.0 +
        float4(SMAASample(colorTex, float2(tc3.x,  tc12.y)).rgb, 1.0) * (w3.x / w12.x) +

        return color.rgb * rcp(color.a);
```



Plotting the weights (left pixel): m_0

- $m_0(x) = w_0(x)/w_{12}(x) = \frac{x(1+(x-2)x)}{(x-1)\,x-1}$

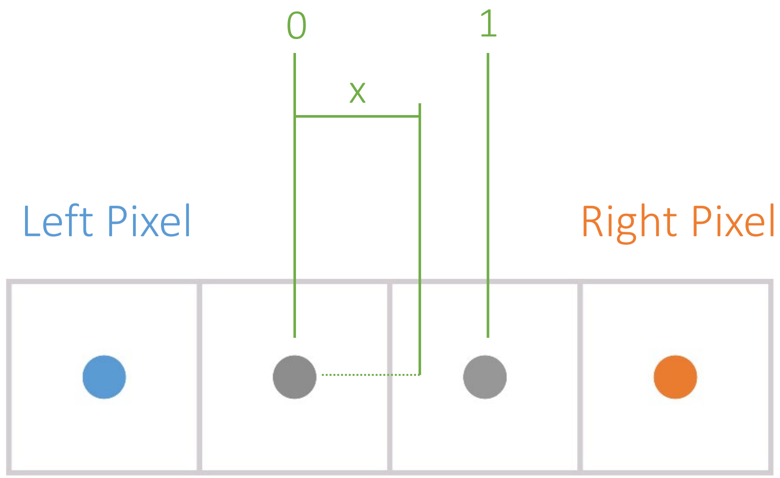


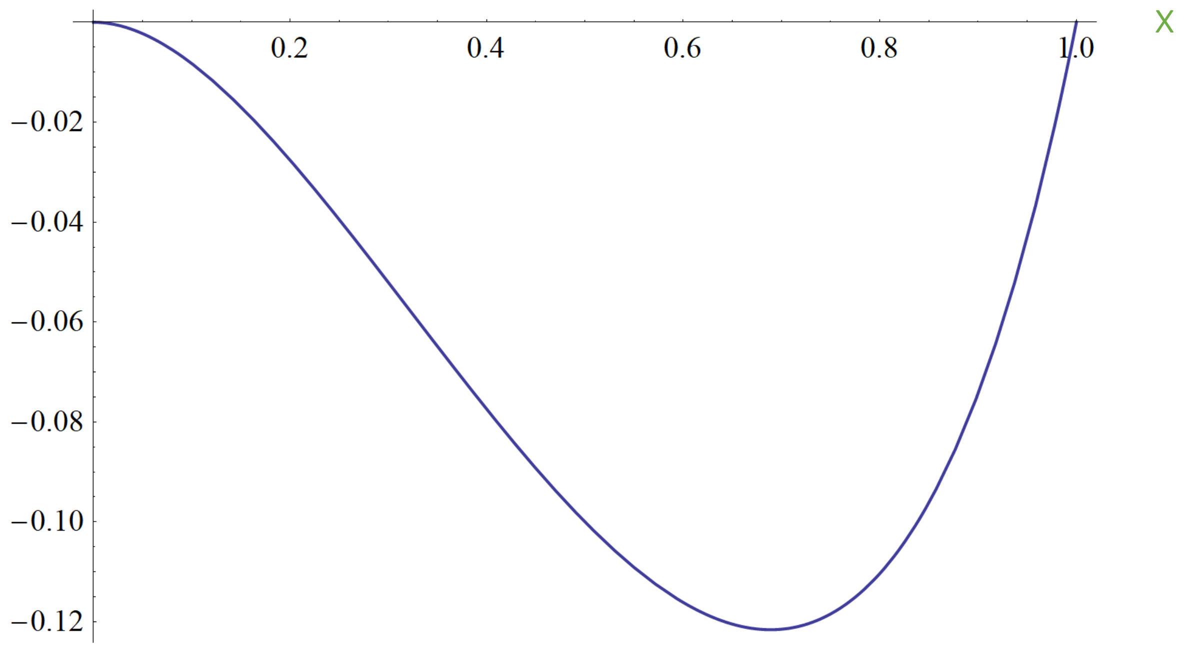


Weight
for left pixel

Plotting the weights (right pixel): m_3

- $m_3(x) = w_3(x)/w_{12}(x) = \frac{(1-x)\,x}{(x-1)\,x-1}$



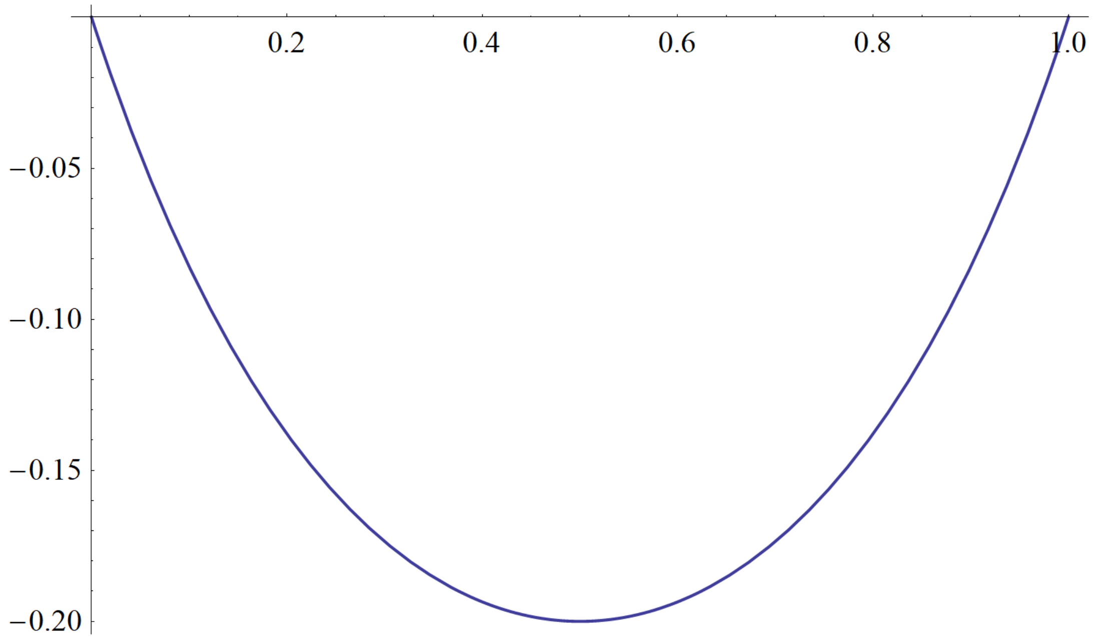


Weight
for right pixel

Plotting the weights: $m_{03}(x)$

- **Assumption:**
  - Left and right colors are the same
- Single weight:

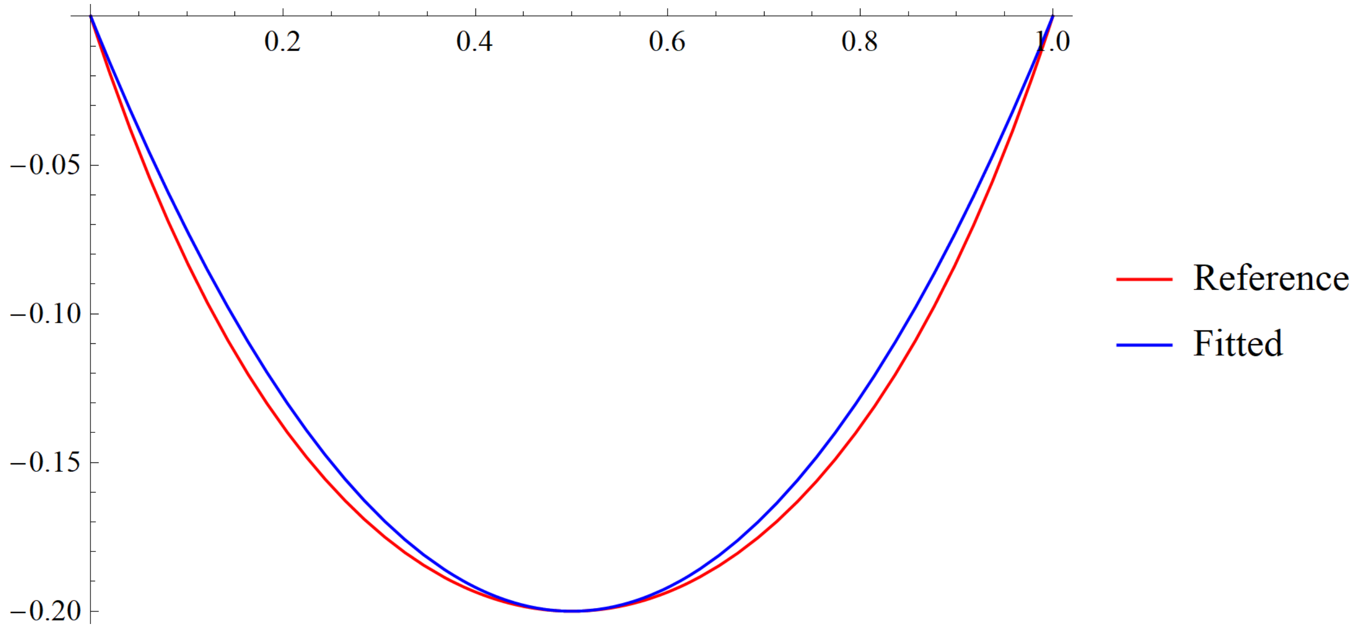
$$m_{03}(x) = m_0(x) + m_3(x)$$



Plotting the weights: $m_{03}'(x)$

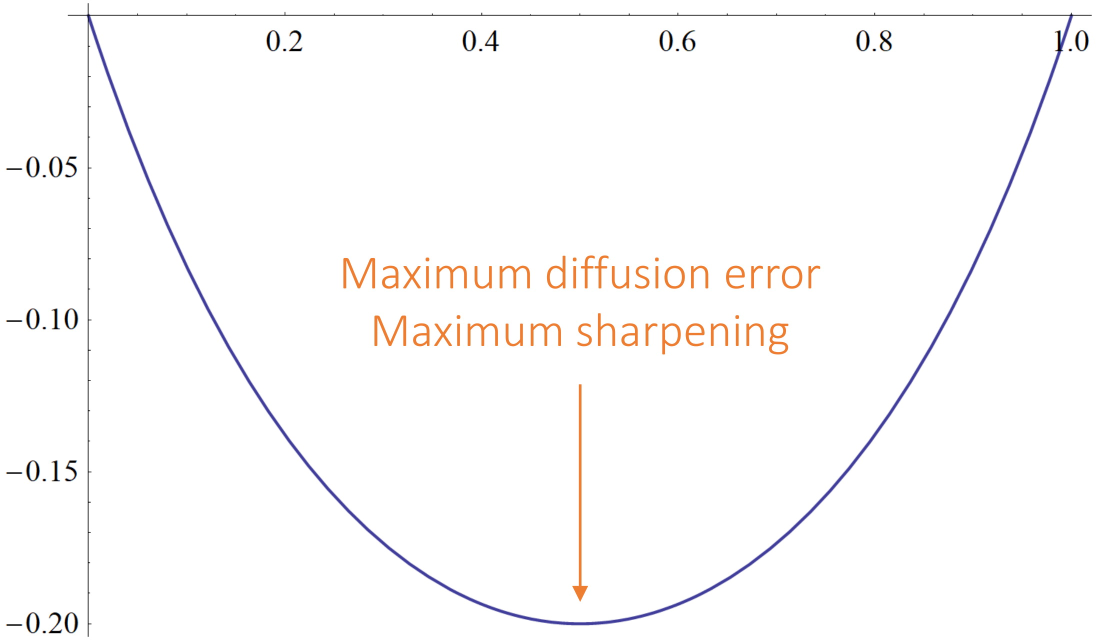
- Which can be fitted:

$$m_{03}'(x) = x(0.8x - 0.8)$$



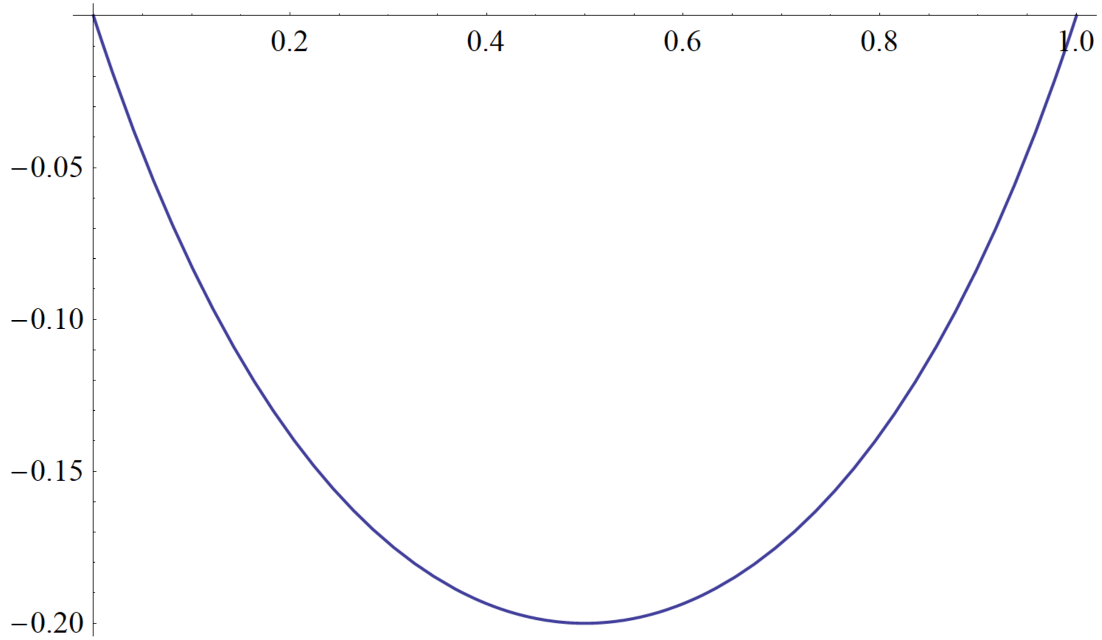
Insight 1

- Why bicubic filtering works so well for temporal resampling?
- When fractional of the position is near 0.5:
  - Maximum numerical diffusion error (blurriest)
  - Bicubic filtering sharpens the most



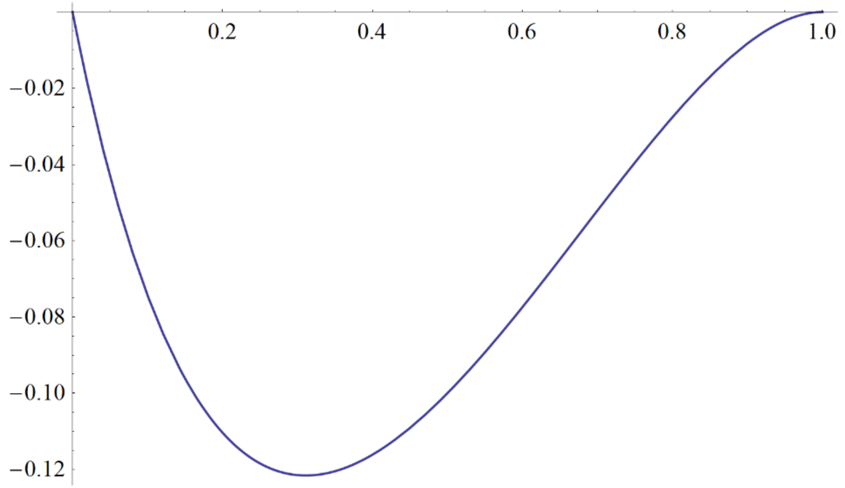
Insight 2

- Bicubic can be seen as:
  - Directional unsharp mask
  - Sharpness dependent on fractional of position
  - Direction dependent on the fractional of the position

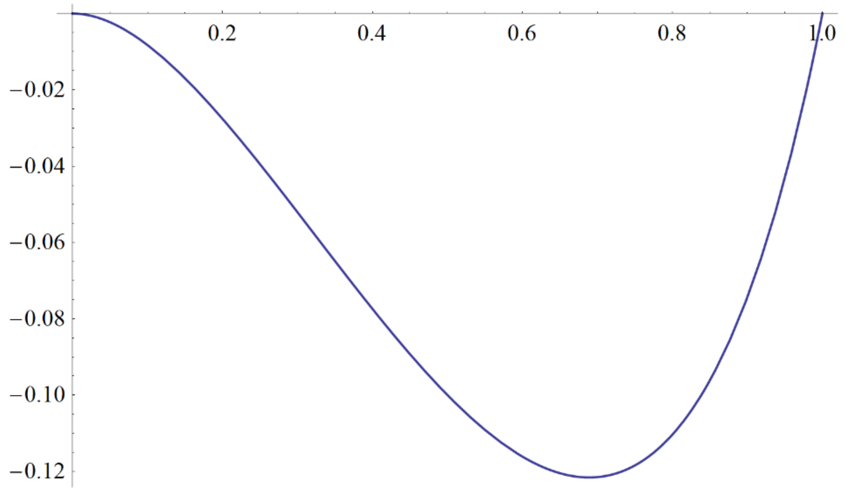


Handling the direction

- We made the assumption of left and right colors being the same
- When this assumption does not hold, they are pulled asymmetrically



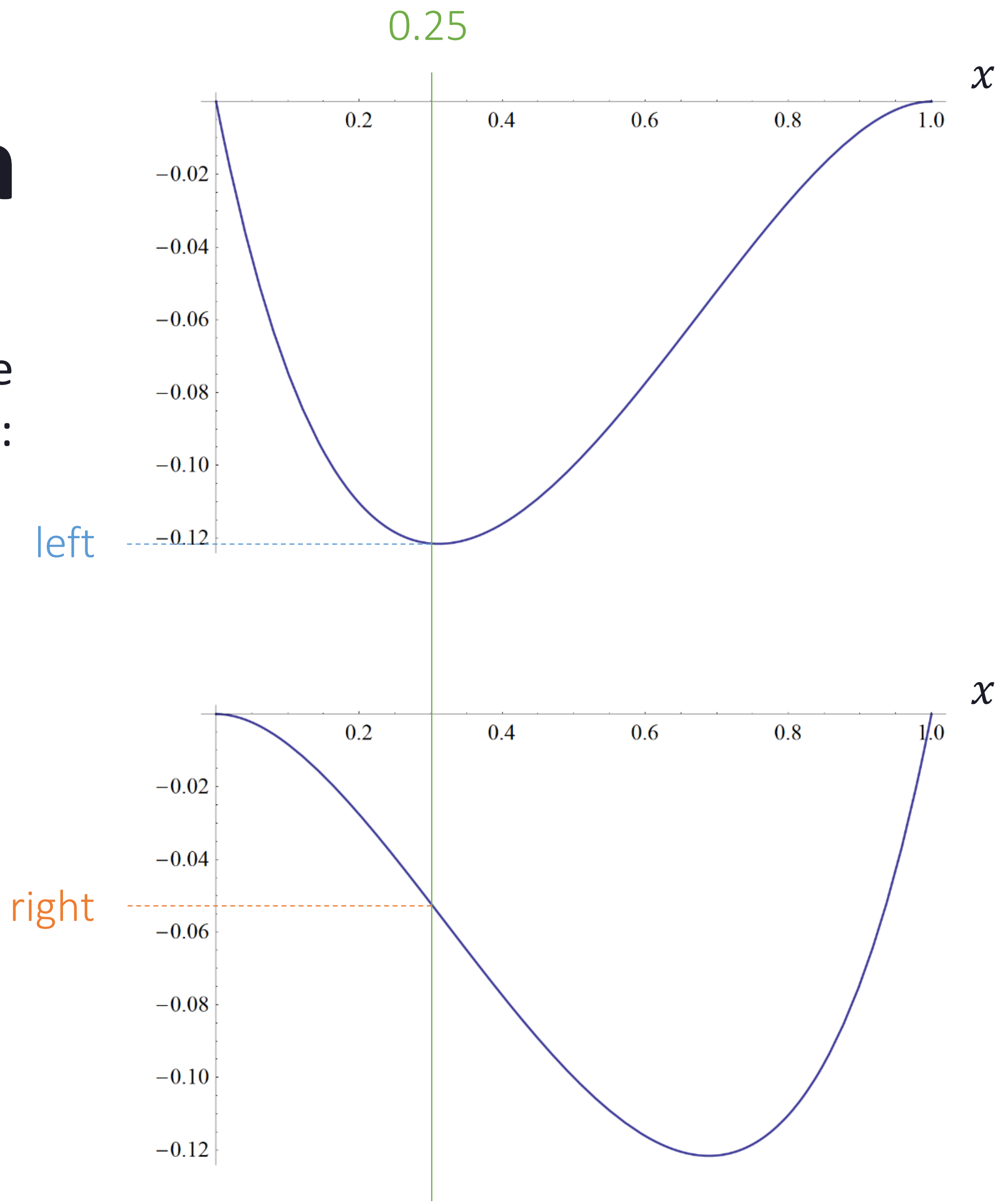
Left



Right

Handling the direction

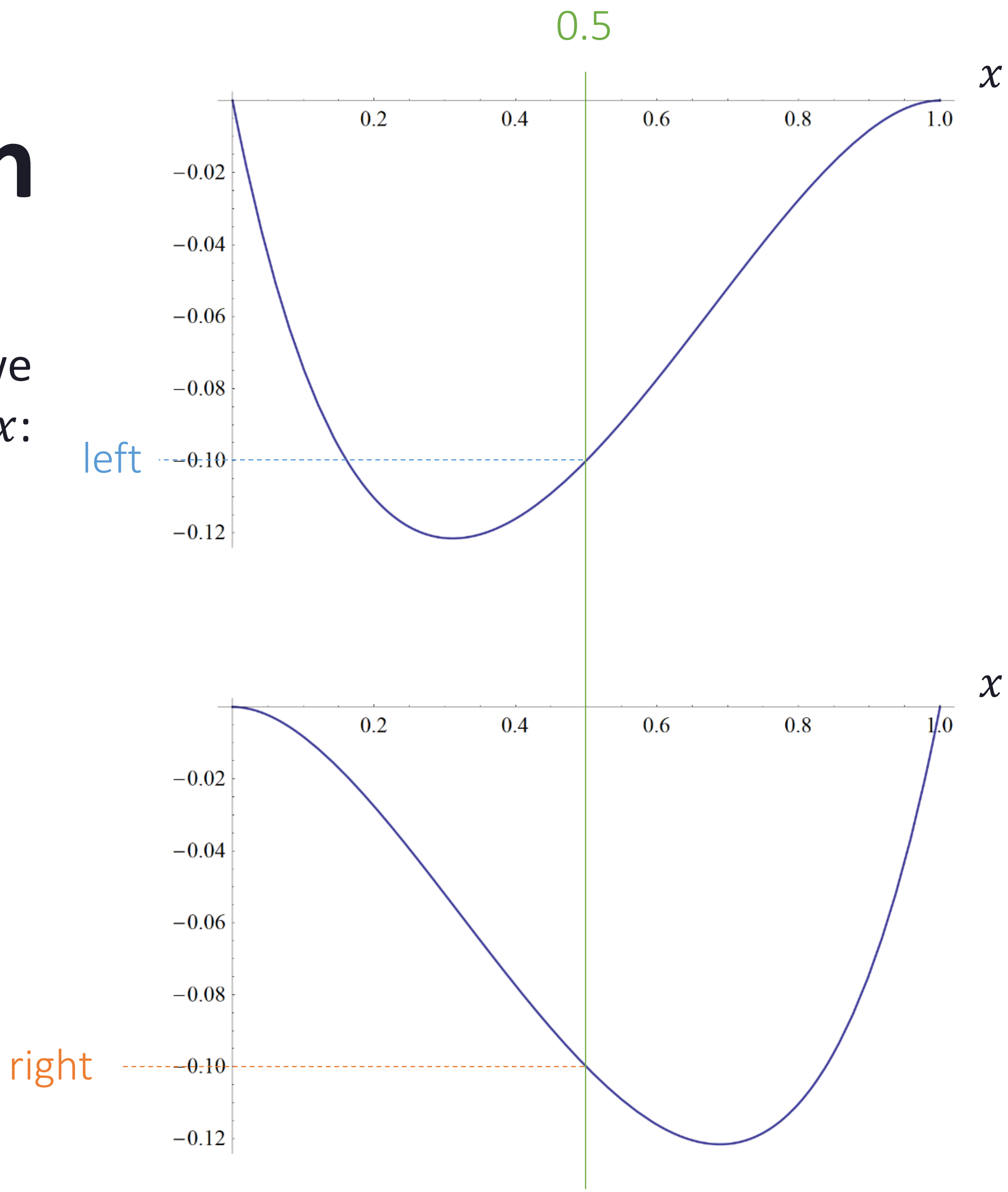
- Looking at relative weights for left and right, we can observe the following from the fractional x:
  - If $x = 0.25$: left $= 2.5 \times$ right



Handling the direction

- Looking at relative weights for left and right, we can observe the following from the fractional x:

  - If $x = 0.25$: left $= 2.5 \times$ right
  - If $x = 0.5$: left $=$ right

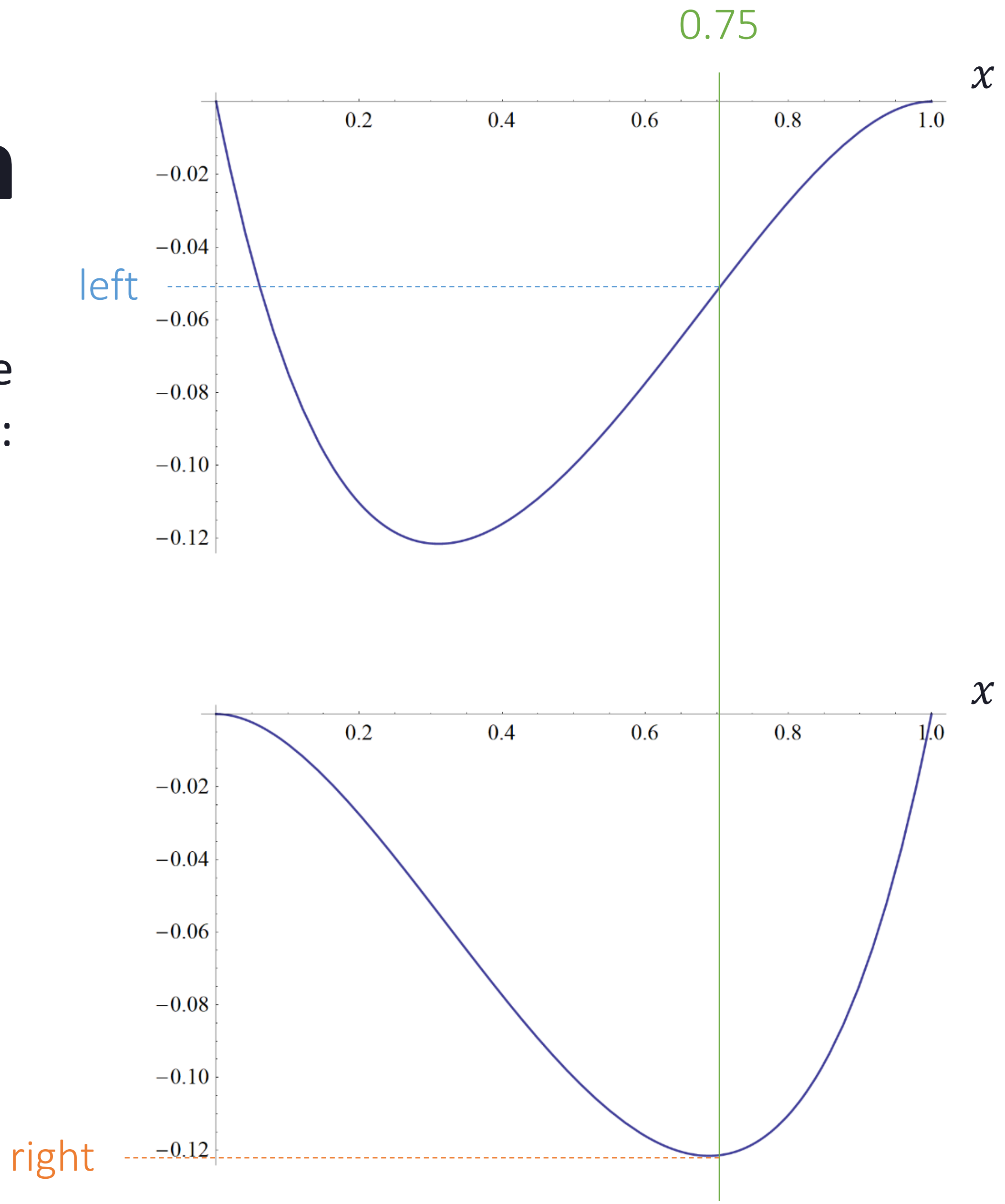


Handling the direction

- Looking at relative weights for left and right, we can observe the following from the fractional x:

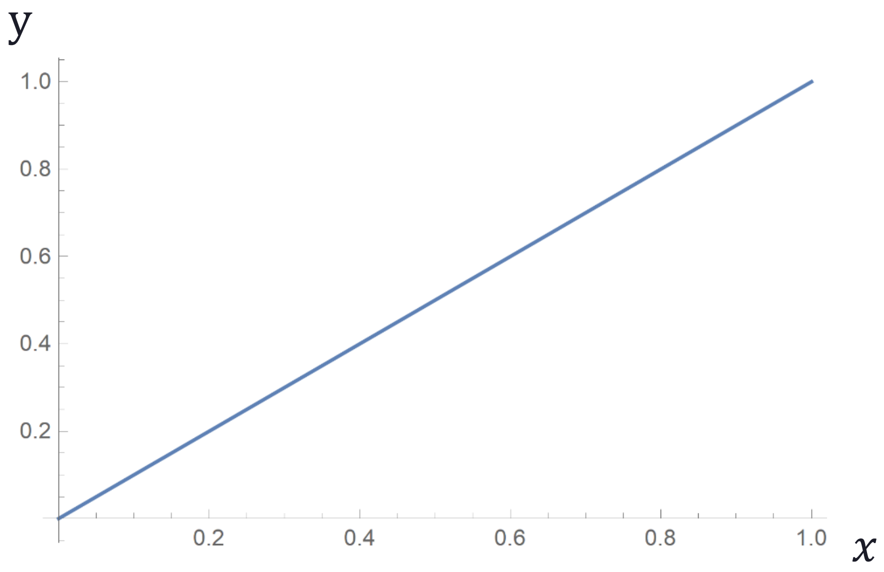
  - If $x = 0.25$: left $= 2.5 \times$ right
  - If $x = 0.5$: left $=$ right
  - If $x = 0.75$: right $= 2.5 \times$ left

- Linear?



Handling the direction

- Plotting $y = \dfrac{m_0(x)}{m_0(x) + m_3(x)}$ confirmed linearity
- Just calculate blended color with a lerp over the fractional x
  - color = lerp(left, right, x)



Summary

- We went from the computationally expensive Mitchell-Netravali Bicubic equation:

$$f(x) = \begin{cases} (12 - 9B - 6C)|x|^3 + (-18 + 12B + 6C)|x|^2 + (6 - 2B), |x| < 1 \\ (-B - 6C)|x|^3 + (6B + 30C)|x|^2 + (-12B - 48C)x + (8B + 24C), 1 \le |x| \le 2 \\ 0, otherwise \end{cases}$$

Summary

- We went from the computationally expensive Mitchell-Netravali Bicubic equation:

$$f(x) = \begin{cases} (12 - 9B - 6C)|x|^3 + (-18 + 12B + 6C)|x|^2 + (6 - 2B), |x| < 1 \\ (-B - 6C)|x|^3 + (6B + 30C)|x|^2 + (-12B - 48C)x + (8B + 24C), 1 \leq |x| \leq 2 \\ 0, otherwise \end{cases}$$

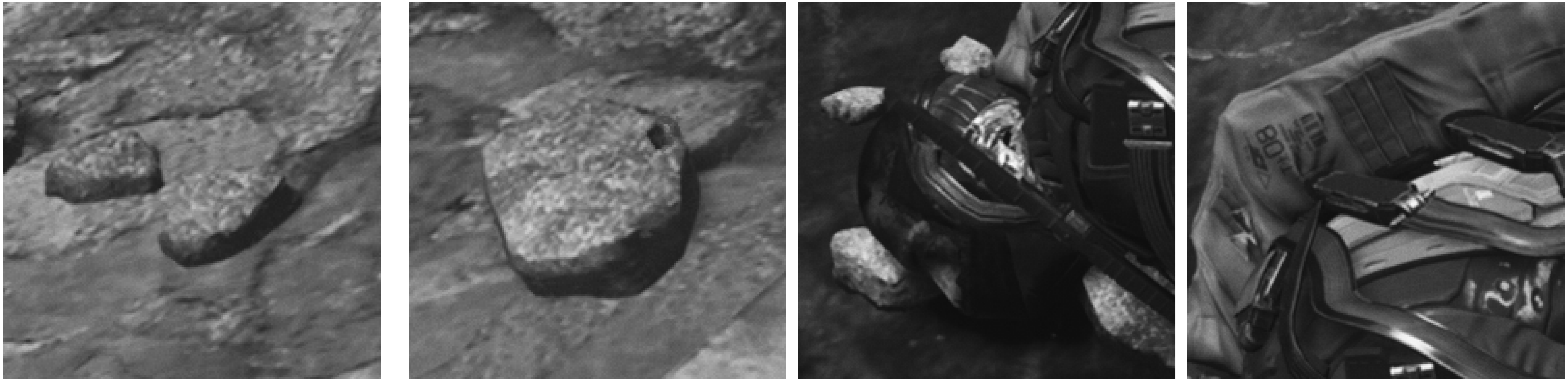
- To this simple shader snippet:

```
m03 = x * ( 0.8 * x - 0.8 )
color = lerp( left, right, x )
filteredColor = ( m03 * color + 1.0 * historyColor ) / ( m03 + 1.0 )
```

Shader Code Statistics

- 9-sample spatial bicubic:
  - Vector ALU: 78
  - Vector memory: 9
  - Estimated cost (cycles): 1868
- 5-sample spatial bicubic:
  - Vector ALU: 69
  - Vector memory: 4 + 1 already available
  - Estimated cost (cycles): 978 (1.91x)
- 1-sample spatio-temporal bicubic:
  - Vector ALU: 51
  - Vector memory: 1 + 4 already available
  - Estimated cost (cycles): 372 (5x)

Results



9-Sample Bicubic Resampling

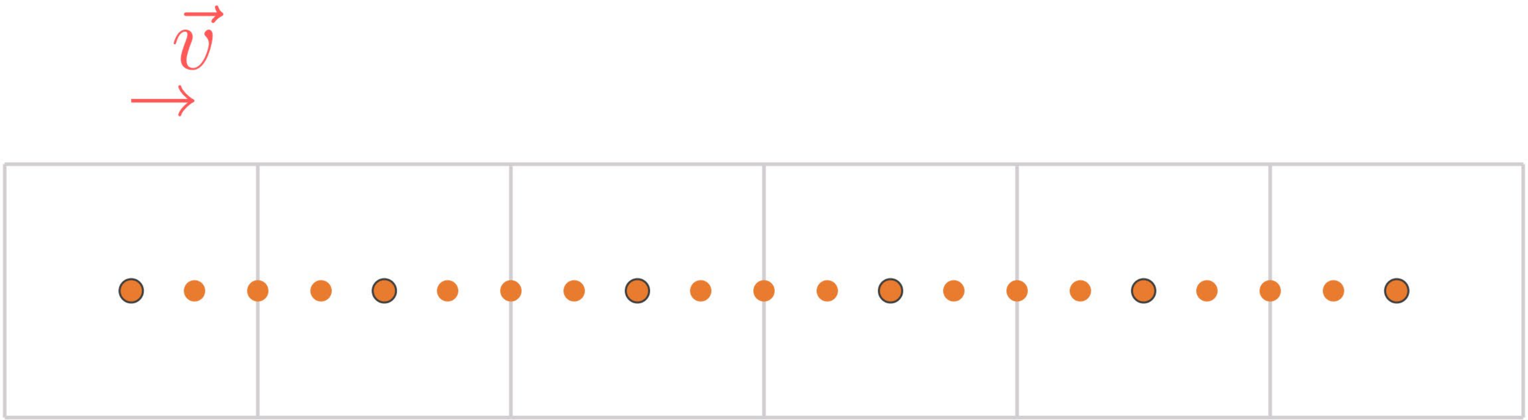
Results

Our 1-Sample Spatio-Temporal Bicubic Resampling

1x Antialiasing Dynamics

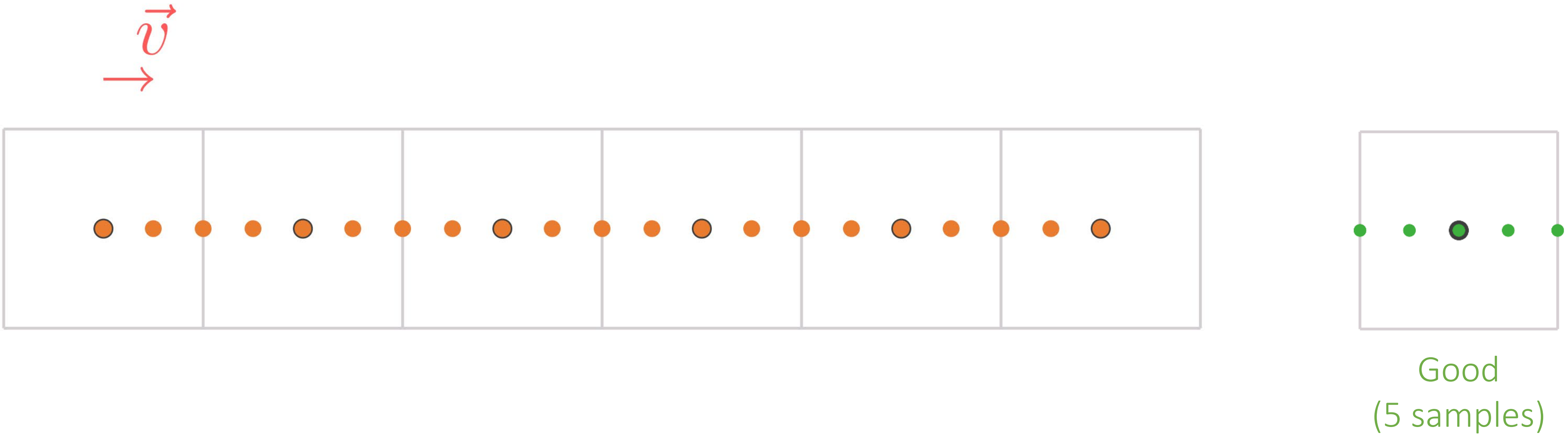
Dynamic Subpixel Jittering

- 1x antialiasing subpixel landing for 0.25px velocity sequence



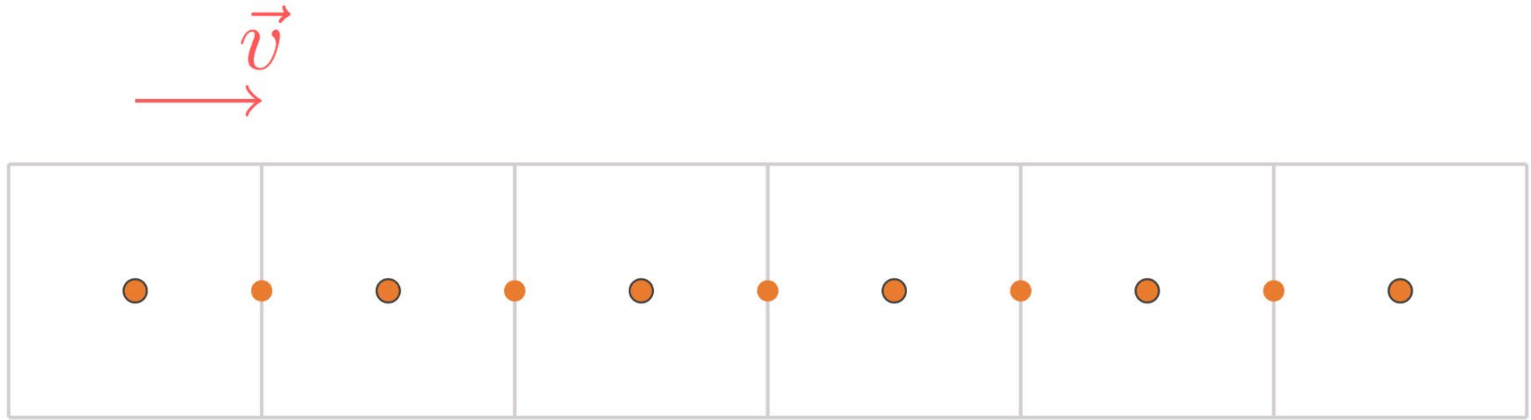
Dynamic Subpixel Jittering

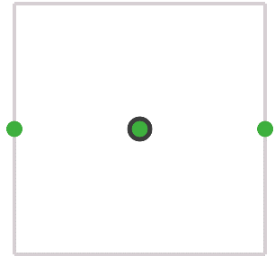
- 1x antialiasing subpixel landing for 0.25px velocity sequence



Dynamic Subpixel Jittering

- 1x antialiasing subpixel landing for 0.5px velocity sequence

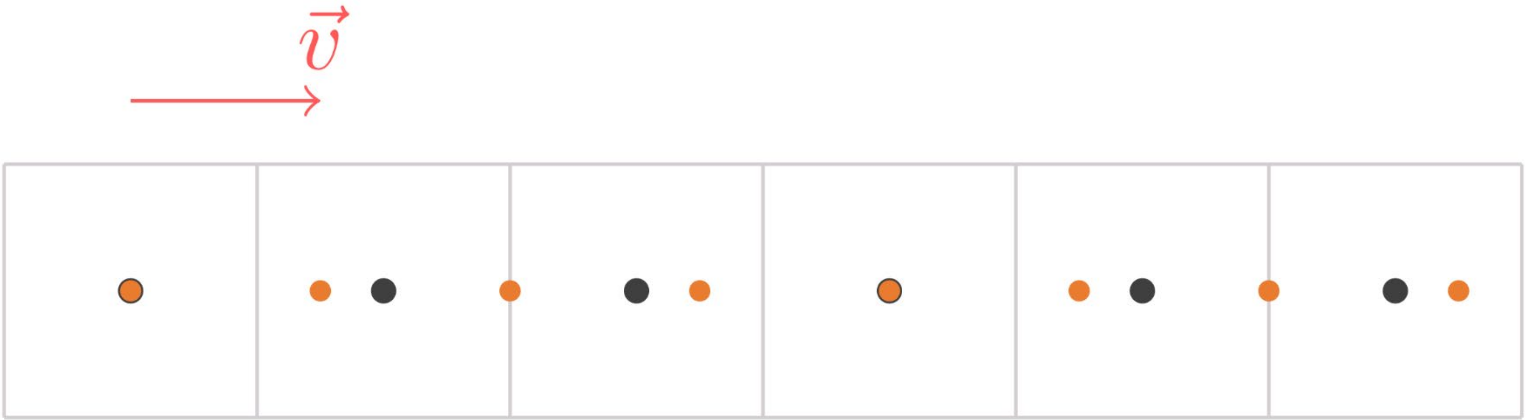


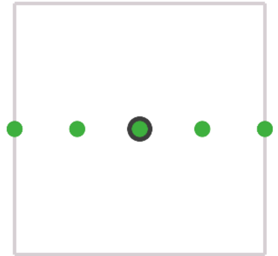


Good
(3 samples)

Dynamic Subpixel Jittering

- 1x antialiasing subpixel landing for 0.75px velocity sequence





Good
(5 samples)

Dynamic Subpixel Jittering

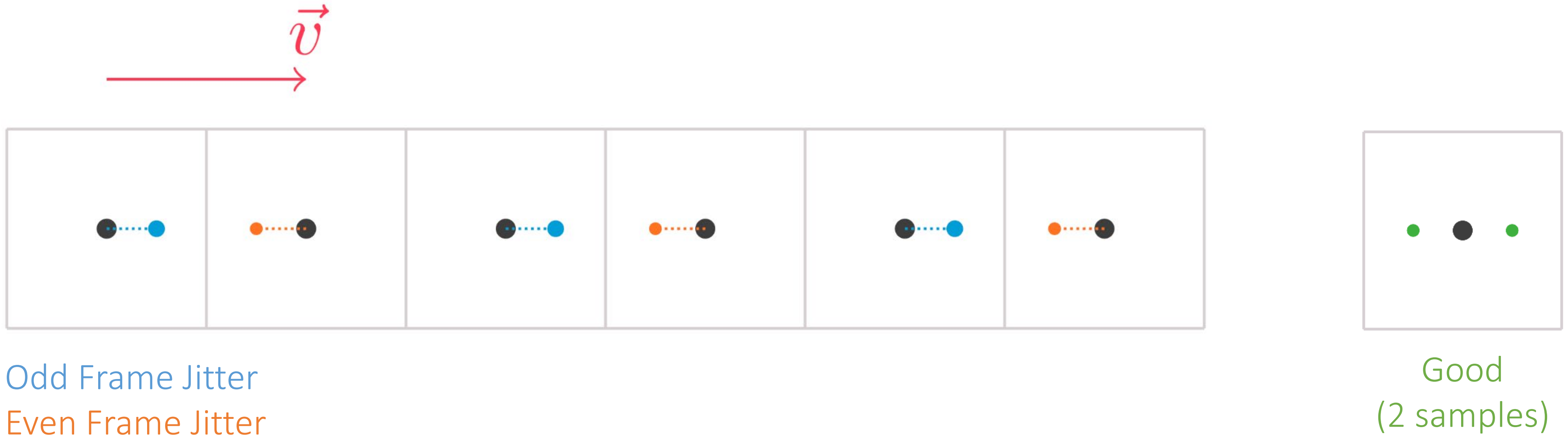
- 1x antialiasing subpixel landing for 1px velocity sequence

Bad
(1 sample)

Temporal 2x Antialiasing Dynamics

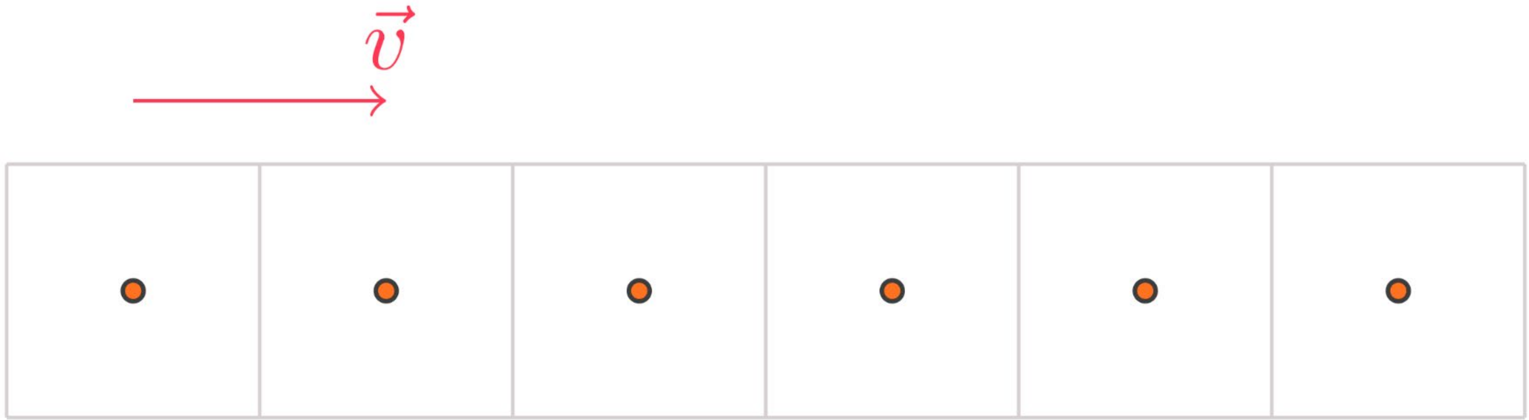
Dynamic Subpixel Jittering

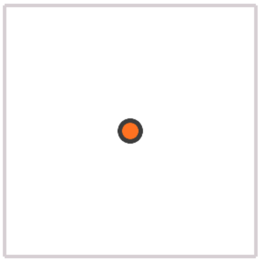
- Temporal 2x antialiasing subpixel landing for 1px velocity sequence



Dynamic Subpixel Jittering

- 1x antialiasing subpixel landing for 1px velocity sequence

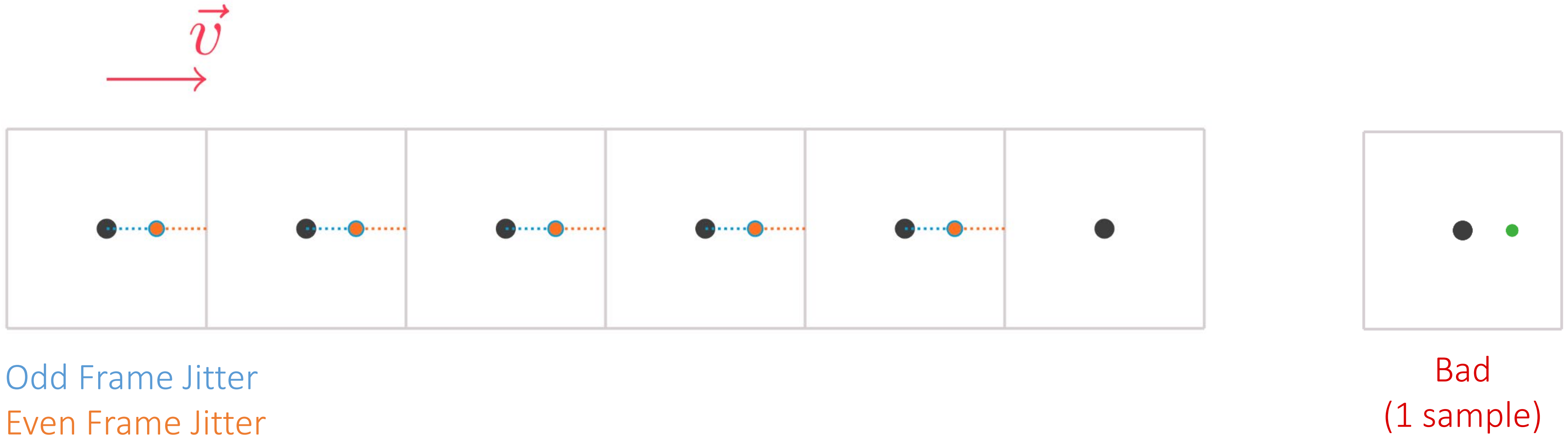




Bad
(1 sample)

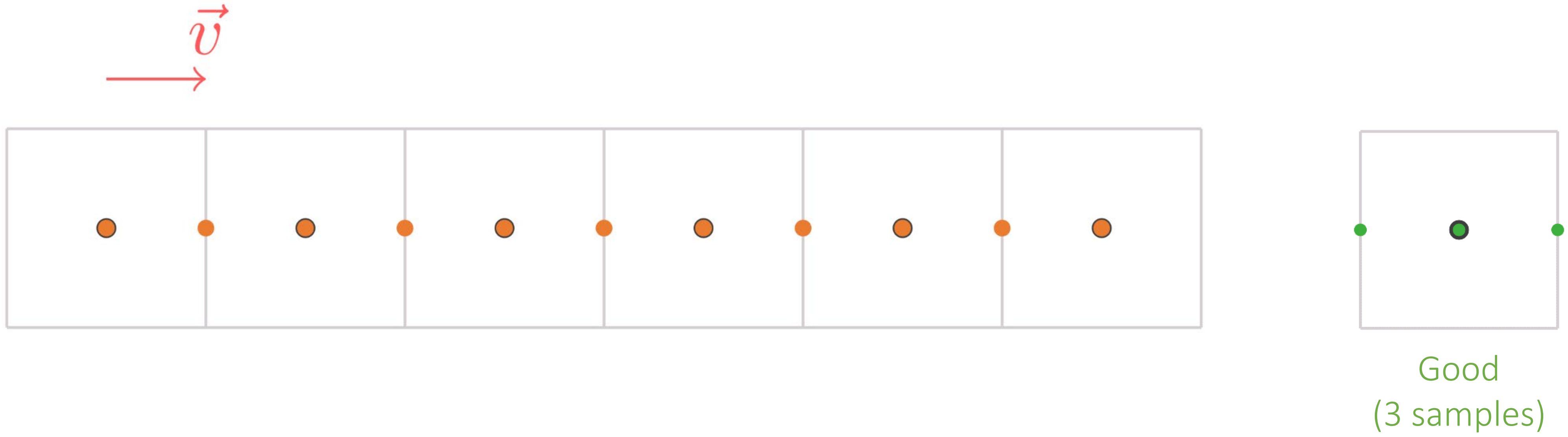
Dynamic Subpixel Jittering

- Temporal 2x antialiasing subpixel landing for 0.5px velocity sequence



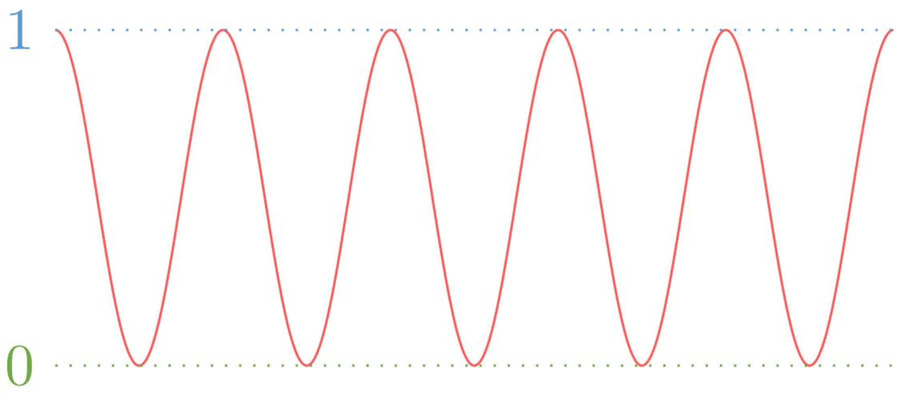
Dynamic Subpixel Jittering

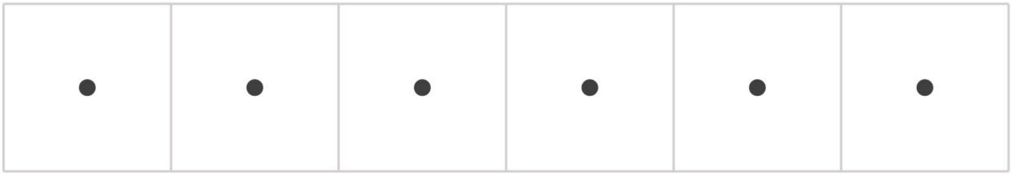
- 1x antialiasing subpixel landing for 0.5px velocity sequence



Dynamic Subpixel Jittering

- **Idea:** alter subpixel position on the vertex shader according to velocity

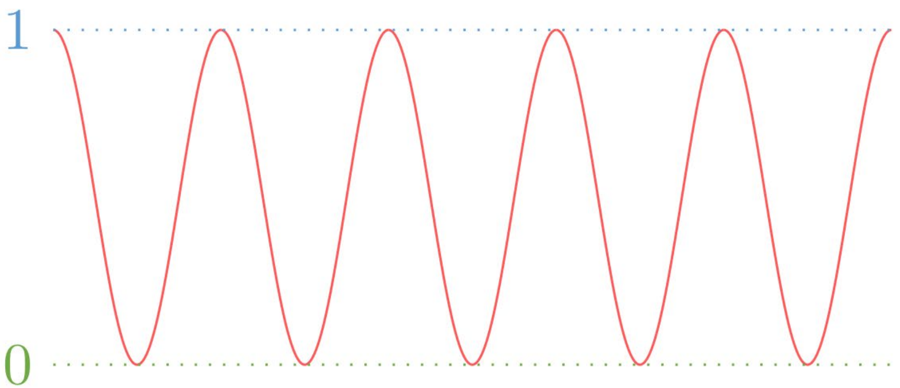




```
float2 scale = 0.5f + 0.5f *
               cos((3.141592f / jitterDistance) *
                   velocity);
svPosition.xy += scale * jitter.xy * svPosition.w;
```

Dynamic Subpixel Jittering

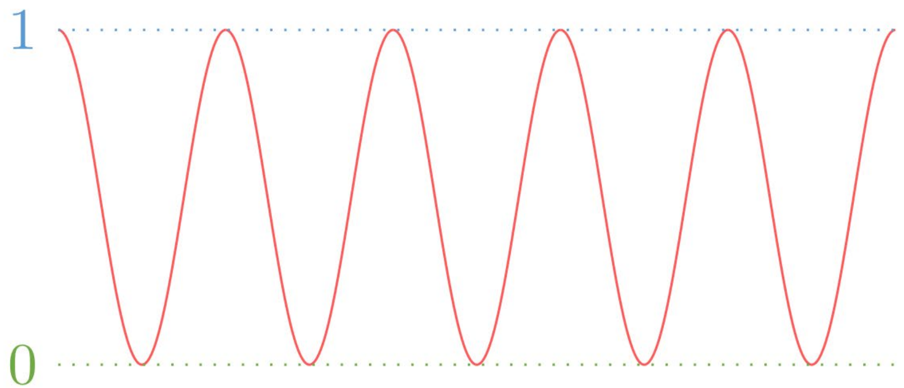
- **Idea:** alter subpixel position according to velocity on the vertex shader:
  - Velocity ~0.5px: no jitter

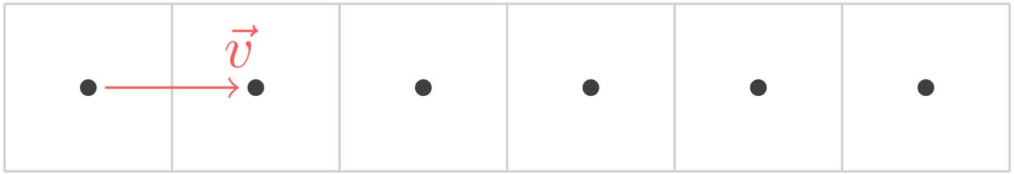



```
float2 scale = 0.5f + 0.5f *
        cos(( 3.141592f / jitterDistance ) *
          velocity );
svPosition.xy += scale * jitter.xy * svPosition.w;
```

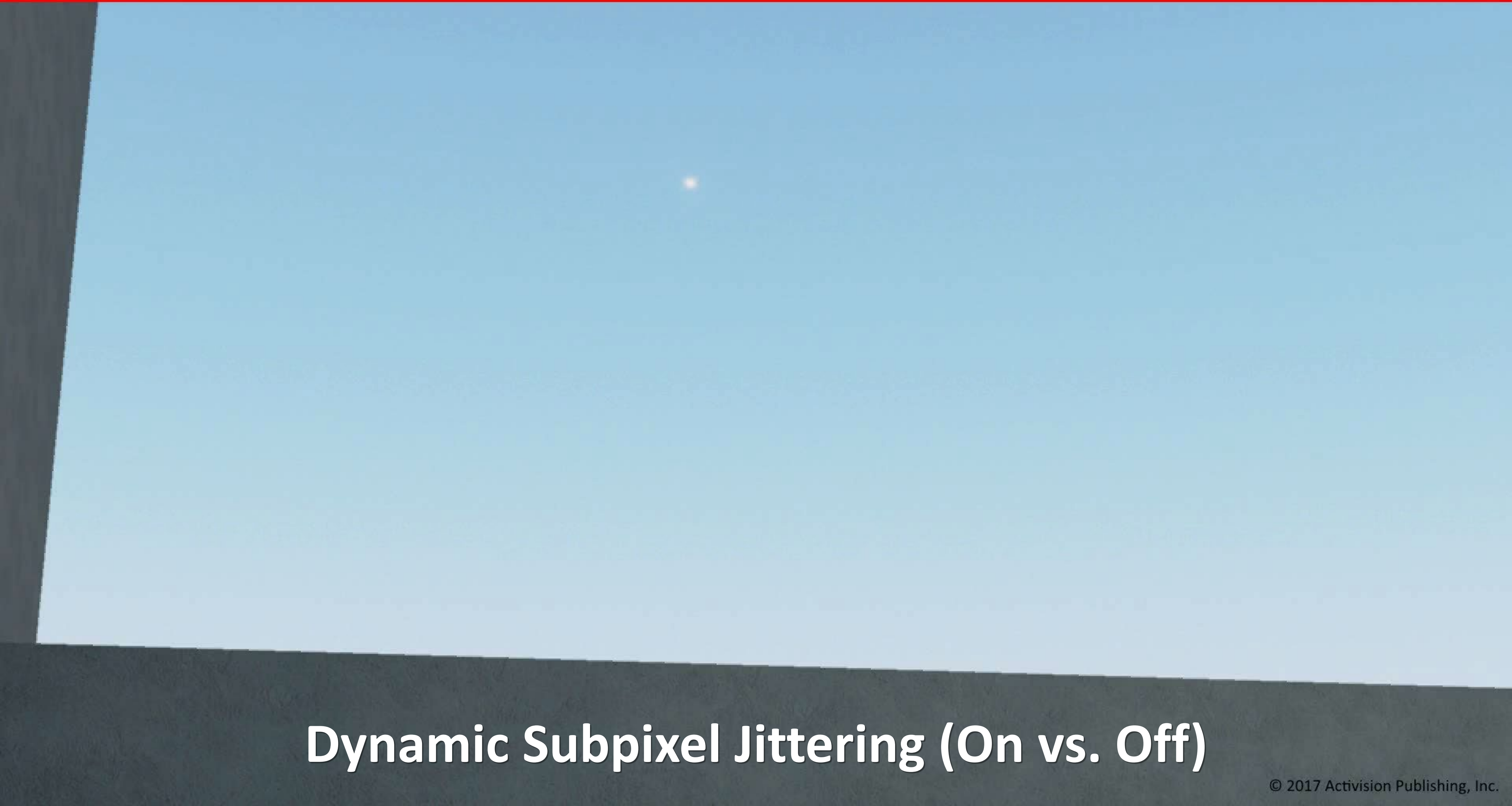
Dynamic Subpixel Jittering

- **Idea:** alter subpixel position according to velocity on the vertex shader:
  - Velocity ~0.5px: no jitter
  - Velocity ~1px: 2x jitter

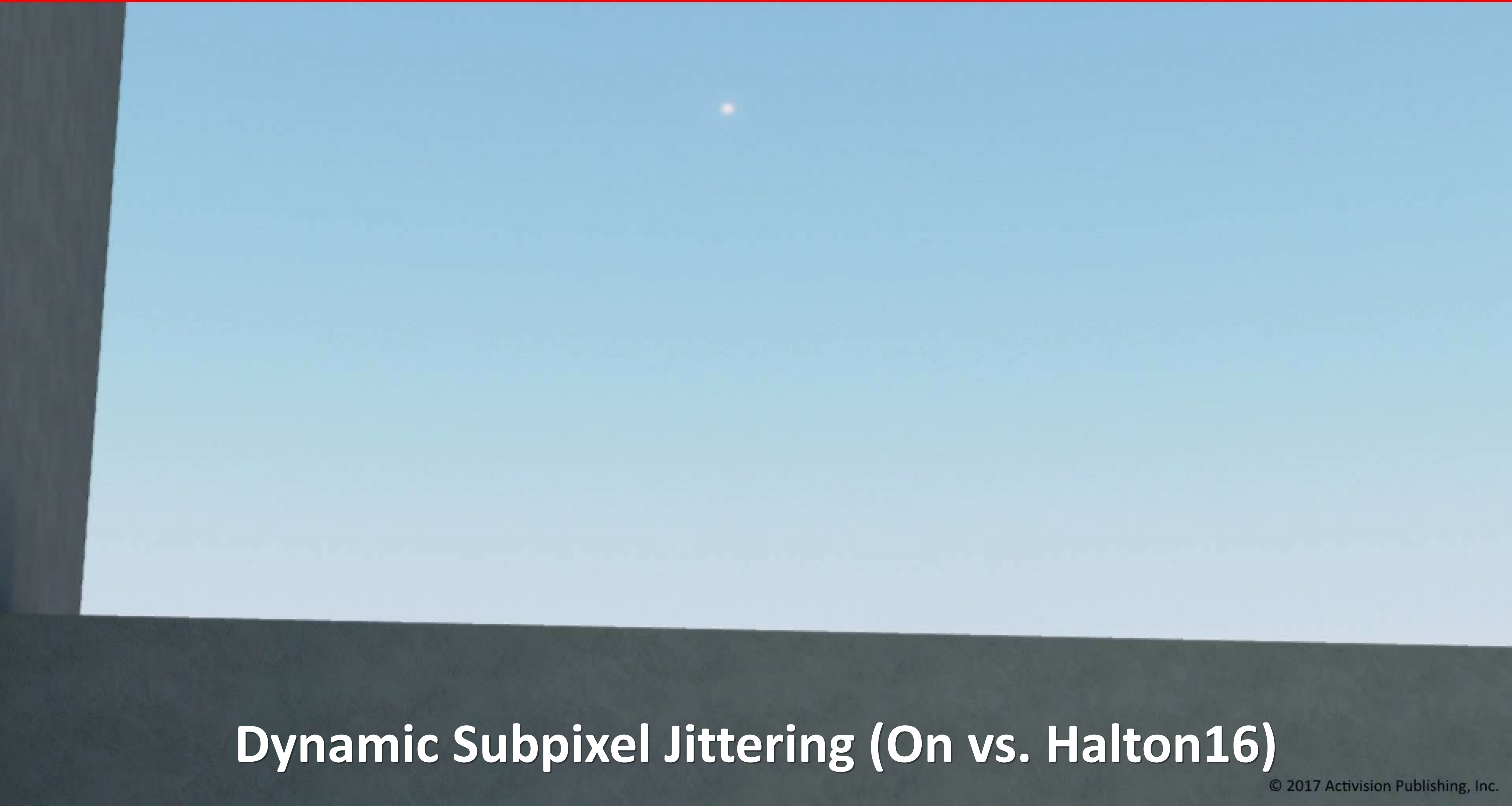





```
float2 scale = 0.5f + 0.5f *
               cos((3.141592f / jitterDistance) *
                   velocity);
svPosition.xy += scale * jitter.xy * svPosition.w;
```



Dynamic Subpixel Jittering (On vs. Off)



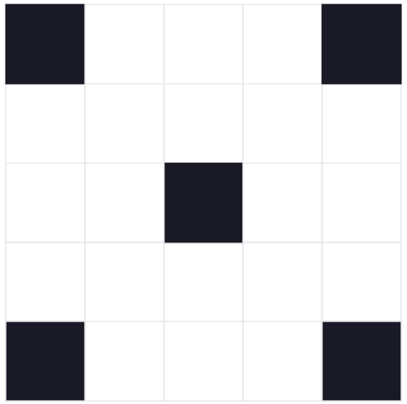
Dynamic Subpixel Jittering (On vs. Halton16)

Insight

- Sequences of 16x Halton offsets are not necessarily better than 1x nor 2x jittering in motion
  - Optimal sampling scheme depends on velocity

Halfres Nearest Velocities [Jimenez2016]

- **Idea:** precalculate nearest velocity
- Dynamic velocities typically composited with camera ones in a full screen pass
- Composite and downsample to half resolution using compute shader
  - Pick closest to camera velocity
  - Amortized if async
- Original 10-sample closest velocity reduced to:
  - 2 gathers for the velocity (GatherRed and GatherGreen)
  - 1 gather for depth
- Half resolution velocities and 8-bit:
  - Reduce velocity buffers footprint from 7.91MB to 0.98MB



Halfres Velocity Packing

- Pack depths and velocities into UINTs
- Fetch both of them at once
- Reduces Gathers from 3 to 1
- Depths strategically positioned on the most significant bits

Halfres Velocity Packing

- Selecting closest velocity, do for each sample:
  - closestVelocity = depth1 < depth2? velocity1 : velocity2;
- Requires a significant amount of conditional moves (cmov)

Halfres Velocity Packing

- Floating point values are lexicographically ordered:
  - float x, y;
  - min(x, y) == asfloat(min(asuint(x),asuint(y)))

Halfres Velocity Packing

- Exploit it:
  - uint velDepth1, velDepth2;
  - nearestVel = DecodeVel(min(velDepth1, velDepth2))
- Just an instruction per velocity sample

Performance on the PlayStation 4

- 1920x1080 input (no temporal upsample)
  - T2x: 0.8 – 0.99ms
- 1440x1080 input
  - TU2x: 0.75 – 0.94ms
  - TU4x: 1.0 – 1.26ms
- 960x1080 input
  - TU2x: 0.67 – 0.9ms
  - TU4x: 0.95 – 1.2ms

Summary

- Dynamic resolution + temporal upsampling
- Temporal upsampling + Spatial reconstruction = 4x upsample
- Analysis of the subpixel jittering
- New temporal AA tools

Q&A - Acknowledgements

Special thanks to SIGGRAPH Advances on Real-Time Rendering organizer Natasha Tatarchuk and Digital Dragons track chair Michal Drobot

- Angelo Pesce
- Adam Micciulla
- Akimitsu Hogge
- Christer Ericson
- Jennifer Velazquez
- Michael Vance
- Wade Brainerd

References

- [Berghoff2016] Tobias Berghoff, Tim Dann, et al. PlayStation®4 Pro High Resolution Technologies, Sony Interactive Entertainment, PlayStation®4 SDK
- [Jimenez2012] SMAA: Enhanced Subpixel Morphological Antialiasing
- [Jimenez2016] Filmic SMAA: Sharp Morphological and Temporal Antialiasing
- [Lottes2011] Temporal Supersampling (blog post, no longer available)
- [Phelippeau2009] Green Edge Directed Demosaicing Algorithm
- [Valient2014] Taking Killzone: Shadow Fall Image Quality Into the Next Generation