Filtering Approaches for Real-Time Anti-Aliasing

http://www.iryoku.com/aacourse/

Filtering Approaches for Real-Time Anti-Aliasing

Jimenez's MLAA & SMAA: Subpixel Morphological Anti-Aliasing

Jorge Jimenez
Universidad de Zaragoza

jorge@iryoku.com

Hi,

I'm Jorge Jimenez, from the Universidad de Zaragoza, Spain.

Up to now, Alex has been talking about his original, CPU-based MLAA.

My presentation is about how we transformed this technique into a very efficient GPU shader, and the improvements we introduced.

SIGGRAPH2011 VANCOUVER

The Team





Jorge Jimenez
Universidad de Zaragoza

Belen Masia
Universidad de Zaragoza

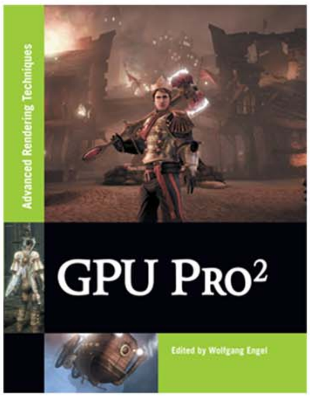
Jose I. Echevarria
Universidad de Zaragoza

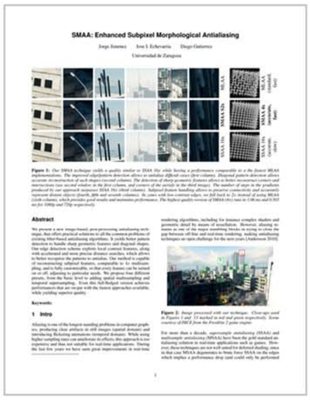
Fernando Navarro
Lionhead Studios

Diego Gutierrez
Universidad de Zaragoza

Practical Morphological Anti-Aliasing
In GPU Pro 2: Advanced Rendering Techniques

But, first of all, here you have all the team members that made this technique possible.





Practical Morphological Anti-Aliasing
In GPU Pro 2: Advanced Rendering Techniques

SMAA: Subpixel Morphological Anti-Aliasing
Technical Report

It has been published in GPU Pro 2, and there has been some improvements ever since.

So, I'll be giving you a sneak preview [click] of our latest-latest technique, which we have just published as a tech report (but more on that later).

SIGGRAPH2011 VANCOUVER

Key Features

- **High Quality**
  - ★ 16× gradients (or more!)
  - ★ Noise proof → **Temporally Stable**
  - ★ Sharpness preservation
- **Fast**
  - ★ 0.28ms@720p (GeForce GTX 470)
  - ★ Beats MSAA by about a 1180% (GeForce 9800 GTX+)
- **Low Memory Footprint**
  - ★ 2× the backbuffer size
- **Portable**
- **Customizable Edge Detection**

Practical Morphological Anti-Aliasing
In GPU Pro 2: Advanced Rendering Techniques

During the development of our technique we had to decide on several tradeoffs.

SIGGRAPH2011 VANCOUVER

Key Features

- **High Quality**
  - ★ 16× gradients (or more!)
  - ★ Noise proof → **Temporally Stable**
  - ★ Sharpness preservation
- **Fast**
  - ★ 0.28ms@720p (GeForce GTX 470)
  - ★ Beats MSAA by about a 1180% (GeForce 9800 GTX+)
- **Low Memory Footprint**
  - ★ 2× the backbuffer size
- **Portable**
- **Customizable Edge Detection**

Practical Morphological Anti-Aliasing
In GPU Pro 2: Advanced Rendering Techniques

First of all, we put our maximum emphasis, on achieving the highest quality possible:

- We obtain gradients in the silhouette of objects that usually surpass those produced by MSAA 16x.

- In our latest version, we are more temporally stable, with a much better management of noise.

- We are really consevative with the image, as we only touch were it's really needed, which translates to a better sharpness preservation.

SIGGRAPH2011 VANCOUVER

Key Features

- High Quality
  - ★ 16× gradients (or more!)
  - ★ Noise proof → **Temporally Stable**
  - ★ Sharpness preservation
- **Fast**
  - ★ 0.28ms@720p (GeForce GTX 470)
  - ★ Beats MSAA by about a 1180% (GeForce 9800 GTX+)
- Low Memory Footprint
  - ★ 2× the backbuffer size
- Portable
- Customizable Edge Detection

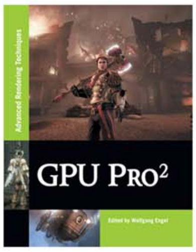
Practical Morphological Anti-Aliasing
In GPU Pro 2: Advanced Rendering Techniques

Second to high quality, we tried to achieve the best performance possible.

We are quite fast on medium to high-end range of GPUs, running in 0.28 on a GeForce GTX 470 and beating MSAA by a factor of twelve in our older test machine.

Key Features

- High Quality
  - ★ 16× gradients (or more!)
  - ★ Noise proof ⟶ **Temporally Stable**
  - ★ Sharpness preservation
- Fast
  - ★ 0.28ms@720p (GeForce GTX 470)
  - ★ Beats MSAA by about a 1180% (GeForce 9800 GTX+)
- **Low Memory Footprint**
  - ★ **2× the backbuffer size**
- **Portable**
- Customizable Edge Detection



Practical Morphological Anti-Aliasing
In GPU Pro 2: Advanced Rendering Techniques

Also, we have a low memory footprint and we are easily portable.

SIGGRAPH2011 VANCOUVER

Key Features

- High Quality
  - ★ 16× gradients (or more!)
  - ★ Noise proof → **Temporally Stable**
  - ★ Sharpness preservation
- Fast
  - ★ 0.28ms@720p (GeForce GTX 470)
  - ★ Beats MSAA by about a 1180% (GeForce 9800 GTX+)
- Low Memory Footprint
  - ★ 2× the backbuffer size
- Portable
- **Customizable Edge Detection**

Practical Morphological Anti-Aliasing
In GPU Pro 2: Advanced Rendering Techniques

The edge detection step can be customized to use color, depth, instance ids, normals, primitive ids, or any combination of them, as Alexander already mentioned.

This allows to select the best method for a particular scenario.

We believe the method used in Killzone 3, described later on by Tobias, could be one of the best edge detection approaches.

SIGGRAPH2011 VANCOUVER

Key Ideas

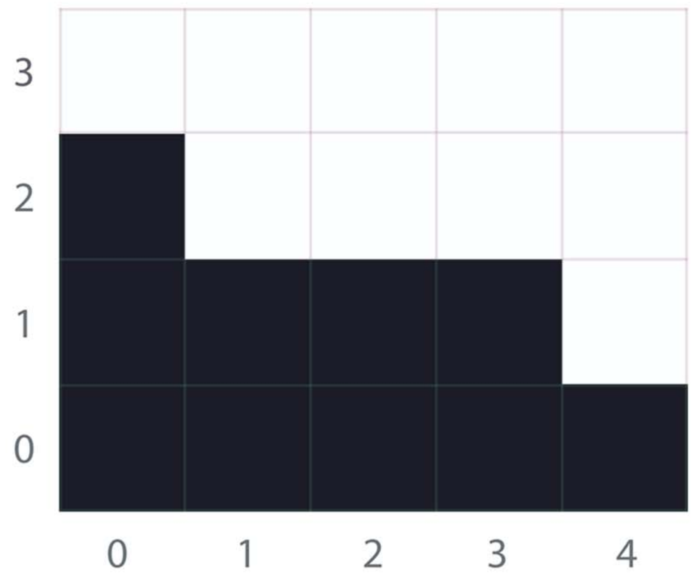
- Translate MLAA to use simple textures
- Use pre-computed textures:
  - ★ Avoid dynamic branching
  - ★ Avoid calculating areas on the fly
- Leverage bilinear filtering to the limit
- Share calculations between pixels (pixels share edges!)
- Mask operations by using the stencil buffer

I would like to begin with the key, high-level ideas of our technique.

For this, let's forget about this boring slide and begin with what the original CPU based approach does, and how we replace each component into a more GPU-friendly form.

SIGGRAPH2011 VANCOUVER

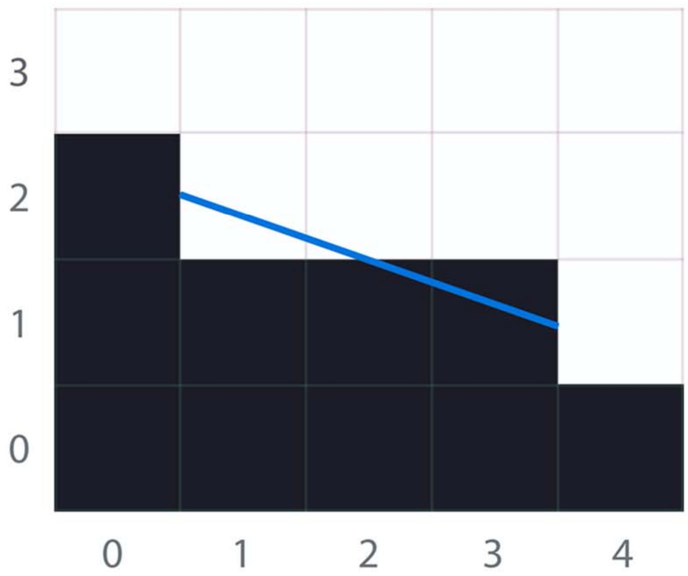
MLAA High-Level Algorithm



So, say we want to antialias this image.

SIGGRAPH2011 VANCOUVER

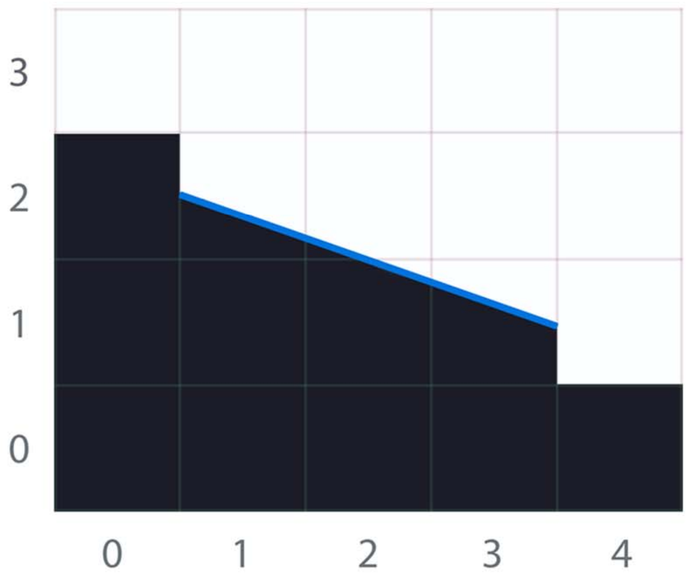
MLAA High-Level Algorithm



For this we have to figure out the blue line, which represents the revectorization of this pixel pattern.

SIGGRAPH2011 VANCOUVER

MLAA High-Level Algorithm

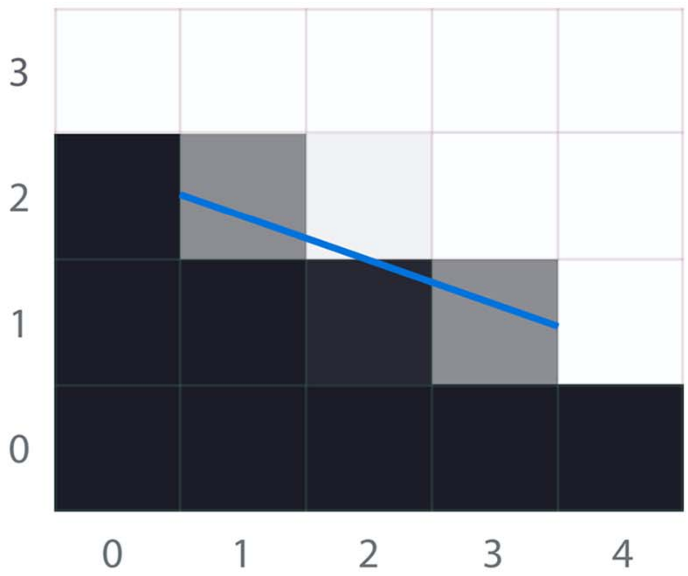


Using this revectorization, we will fill the areas under the line using the opposite color at each pixel.

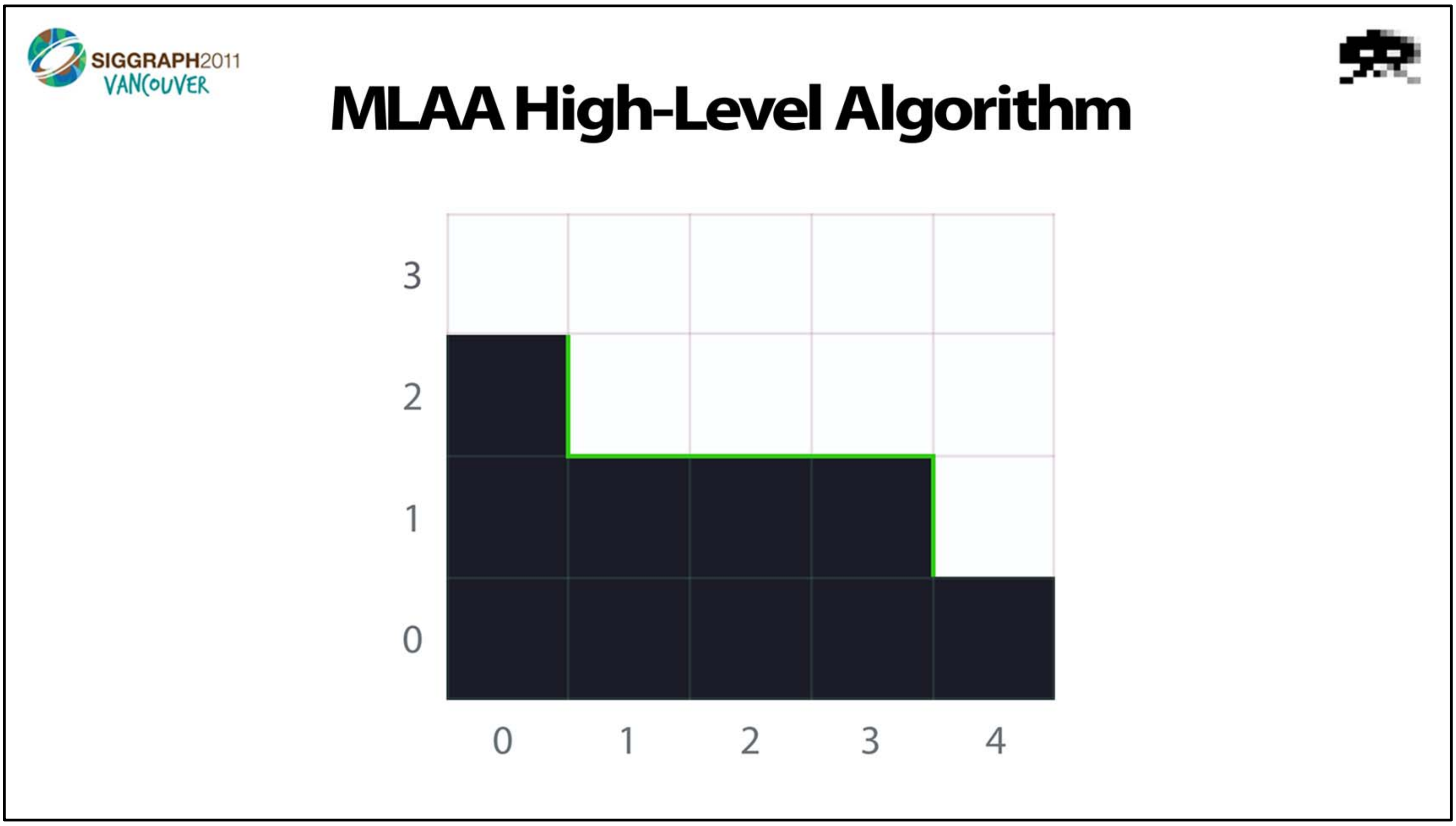
So, in the left we fill with black, and on the right, we fill with white.

This is then translated to…

MLAA High-Level Algorithm



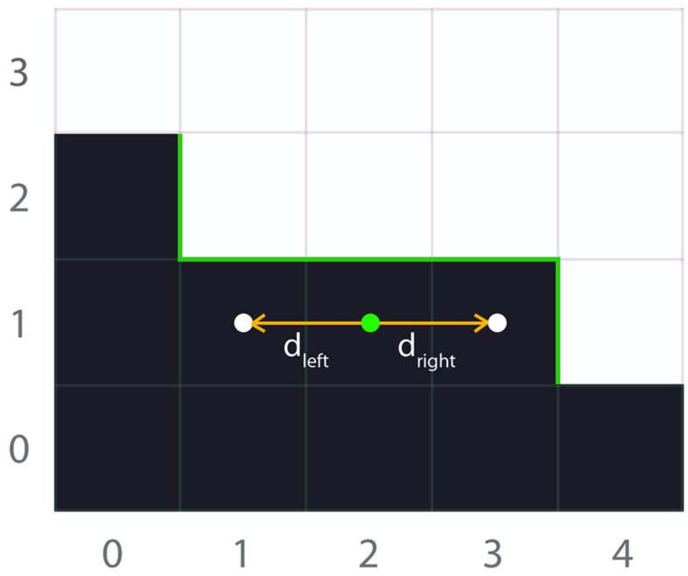
…gray levels, which approximate the real shape.



But, ok, let's rewind.

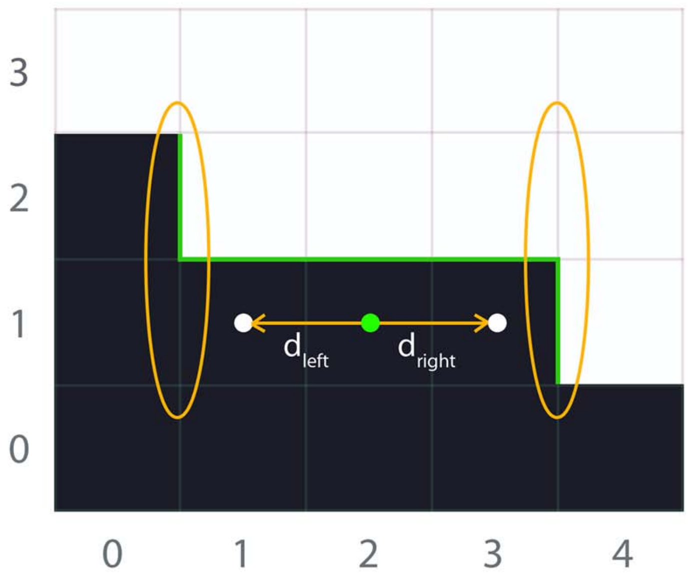
The first step is detecting where the edges are, which are the lines marked on green.

MLAA High-Level Algorithm



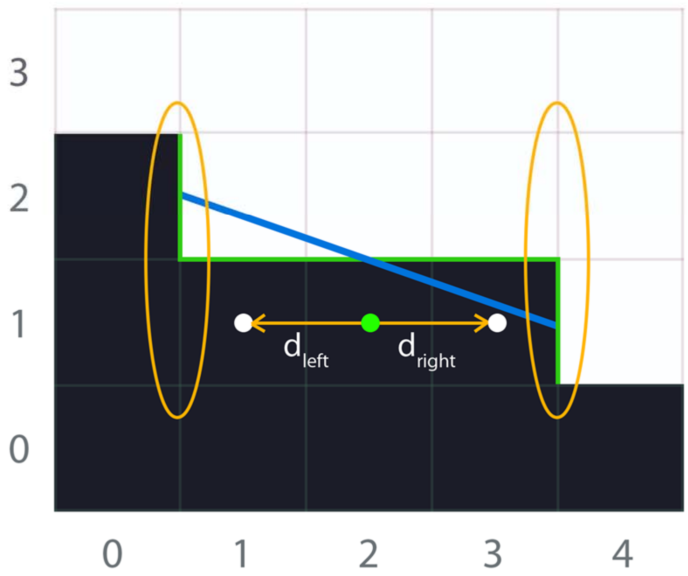
And now, we have to search for the line ends to the left and to the right.

MLAA High-Level Algorithm



And, obtain the crossing edges at each side of the line.

MLAA High-Level Algorithm



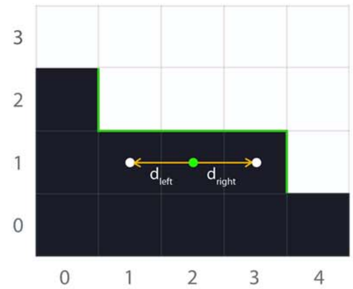
With the distances and crossing edges at hand, we have enough information for calculating the areas under the revectorized line.

Easy, isn't it?

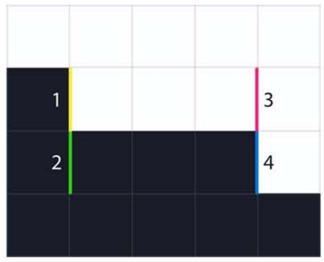
Problems

- Searching for ends is slow
- Fetching for crossing edges is costly
- Revectorization is branchy:

  $2^4 = 16$ cases!
- Area calculation is not cheap
- Up to 4 lines per pixel!



Searchs



Crossing Edges

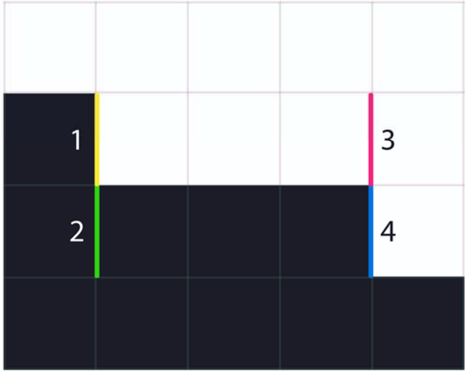
But the beauty doesn’t come without problems:

Fetching edges for the searches and crossing edges is slow, as it requires lots of memory bandwidth.

Problems

- Searching for ends is slow
- Fetching for crossing edges is costly
- Revectorization is branchy:

  $2^4 = 16$ cases!
- Area calculation is not cheap
- Up to 4 lines per pixel!

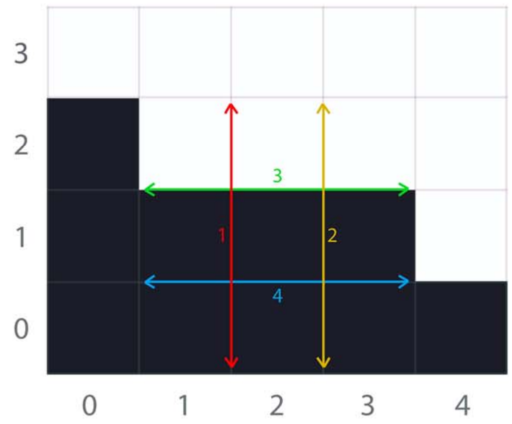


And even with the crossing edges at hand, the revectorization is not trivial given the high number of possible patterns.

Problems

- Searching for ends is slow
- Fetching for crossing edges is costly
- Revectorization is branchy:

  $2^4 = 16$ cases!
- Area calculation is not cheap
- **Up to 4 lines per pixel!**



Furthermore, we have to repeat these calculations up to four times per pixel, one per boundary, which introduces a very huge performance penalty.

So, how can we, solve these problems, without reducing the quality of MLAA?

SIGGRAPH2011 VANCOUVER

Solutions

- Searching for line ends is slow
- Fetching for crossing edges is costly

★ **Solution: introduce bilinear filtering to post processing antialiasing**

This allows to fetch multiple values in a single access!

To improve the searches for line ends and the fetching of crossing edges, which is the most expensive component of MLAA, we introduce bilinear filtering to accelerate post processing antialiasing.

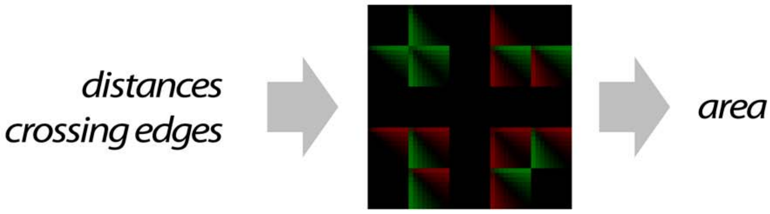
This allows to fetch multiple values in a single access.

SIGGRAPH2011 VANCOUVER

Solutions

- Calculating the revectorization is not easy nor fast
- Accurate area calculation is not cheap

★ **Solution: avoid branchy code by using a precomputed texture**



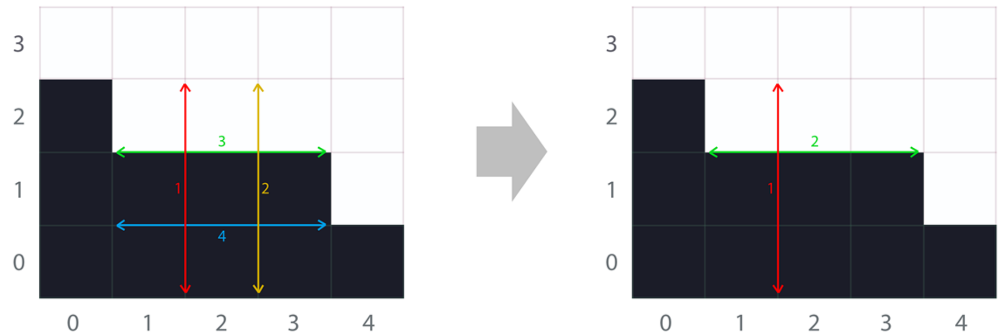
Then, for easing the revectorization and area calculation we built a texture which takes as input the distances and crossing edges and, outputs the area under the line.

This transforms the whole branchy code to discern between the sixteen cases, and the area calculation into a single texture access.

Solutions

- Up to 4 lines can pass through a pixel

★ **Solution: stupid observation; pixels share edges, don't repeat calculations!**



While it's true that four lines can pass through a pixel, they are shared with the neighbors.

So, instead of searching for the four lines, we search just for the top and left lines,

Then, we calculate the corresponding areas, and store them into a temporal buffer.

This allows to share this information with the neighbors, at the cost of introducing another pass.

New Problem!

- We now require three full-screen passes

★ **Solution: use the stencil buffer!**

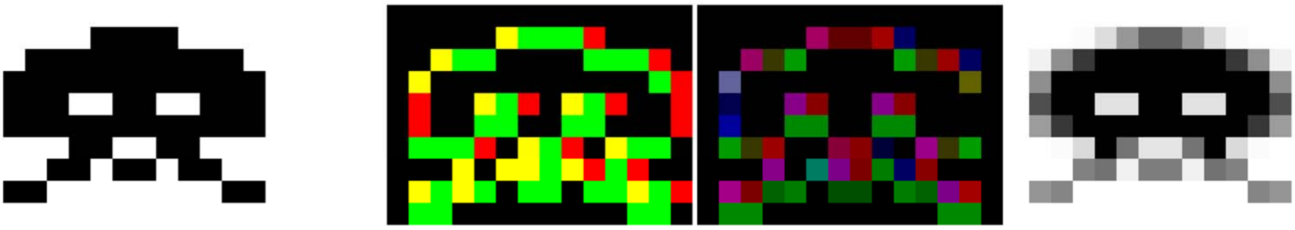
But, now we got a new problem: we require three fullscreen passes.

However, the solution is easy:

[click]

Mask pixels that need procesing in the first pass using the stencil buffer.

Workflow

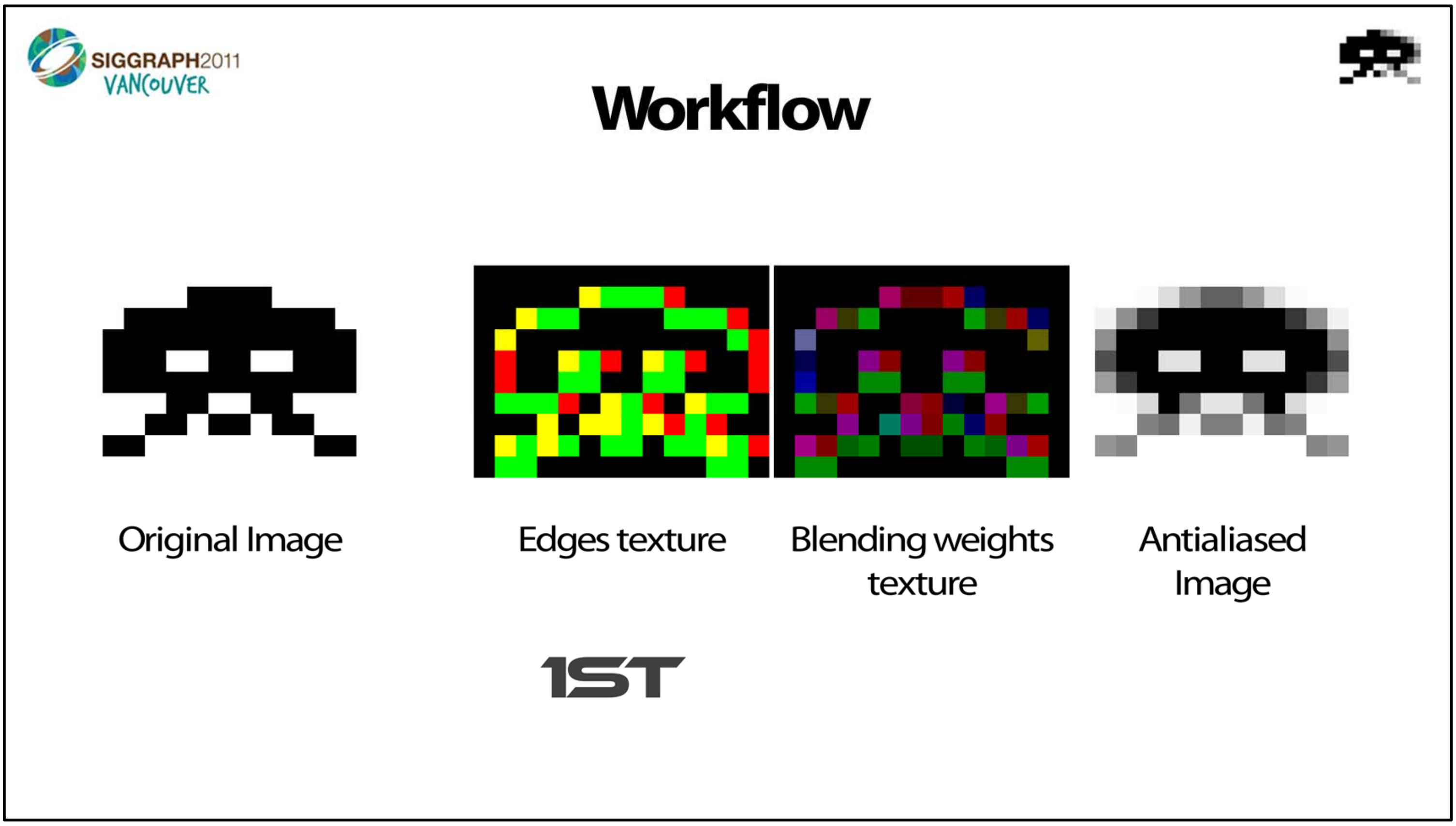


Original Image | Edges texture | Blending weights texture | Antialiased Image

Ok, so we finished with the high level idea, now, let’s dive into the details of our implementation.

Here you have the big picture of our technique.

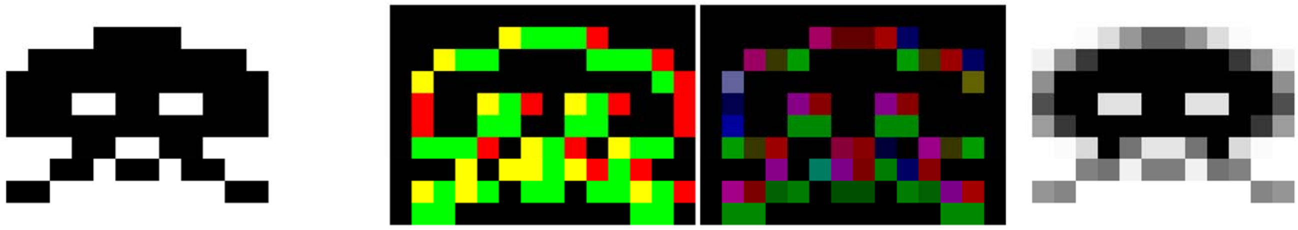
It consists on three passes.



In the first pass, we perform edge detection, obtaining the edges texture.

SIGGRAPH2011 VANCOUVER

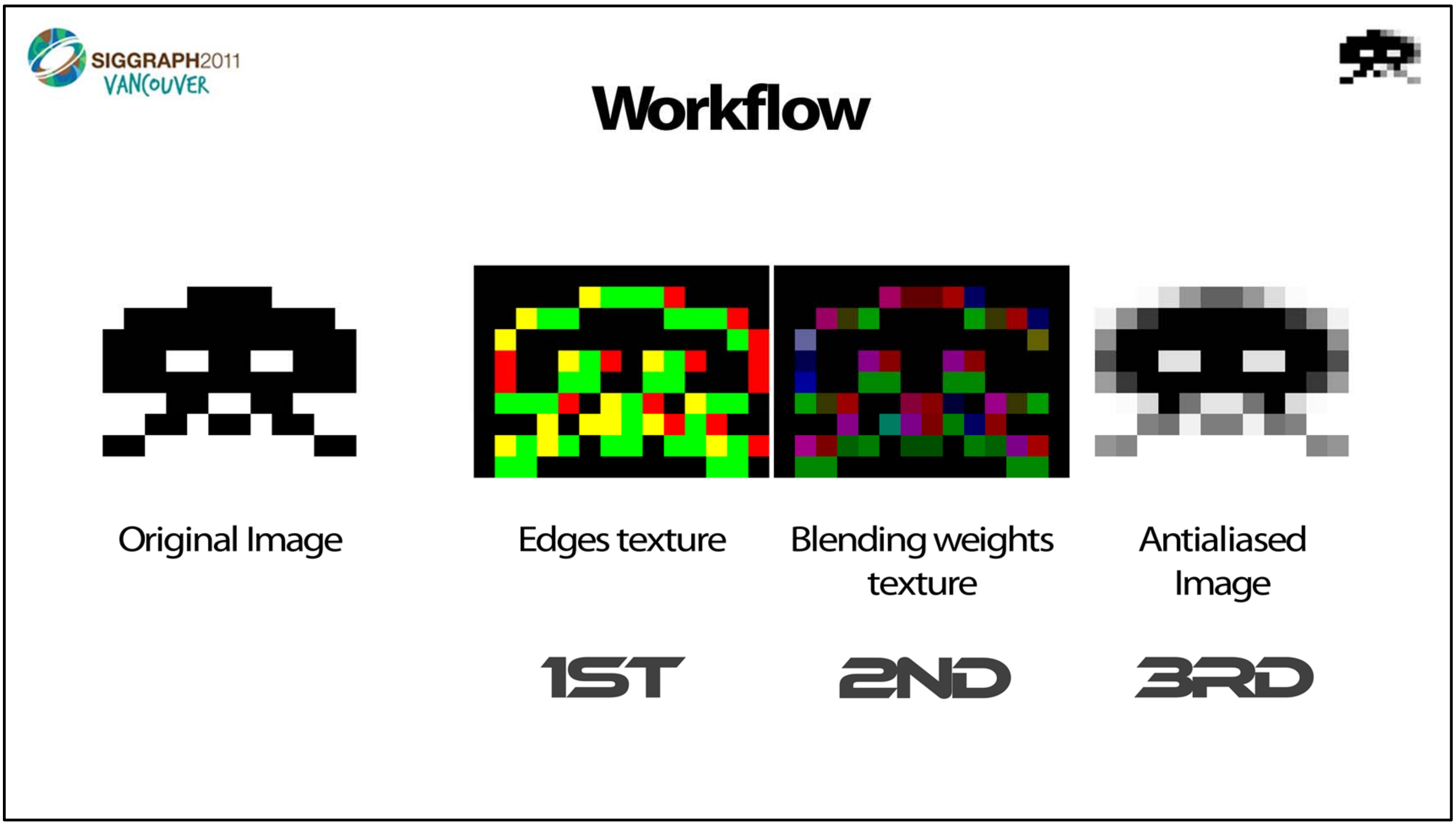
Workflow



Original Image | Edges texture | Blending weights texture | Antialiased Image

1ST 2ND

In the second pass, we process each edge, calculating the revectorizations and obtaining the corresponding areas.



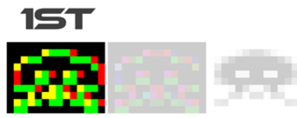
In the third and final pass, we blend each pixel with its four-neighborhood, using the areas from the second pass.

I’m going to skip the edge detection step and go straight to the second one, which has a more interesting implementation.

SIGGRAPH2011 VANCOUVER

Edge Detection 1st Pass

- **Color:** (ITU-R Recommendation BT.709)
  - $Y' = 0.2126 \cdot R' + 0.7152 \cdot G' + 0.0722 \cdot B'$
- **Depth:** cheaper and more clean edges, but cannot work at all object scales
- **Instance ids/depth + normals:** the best if you have this information available



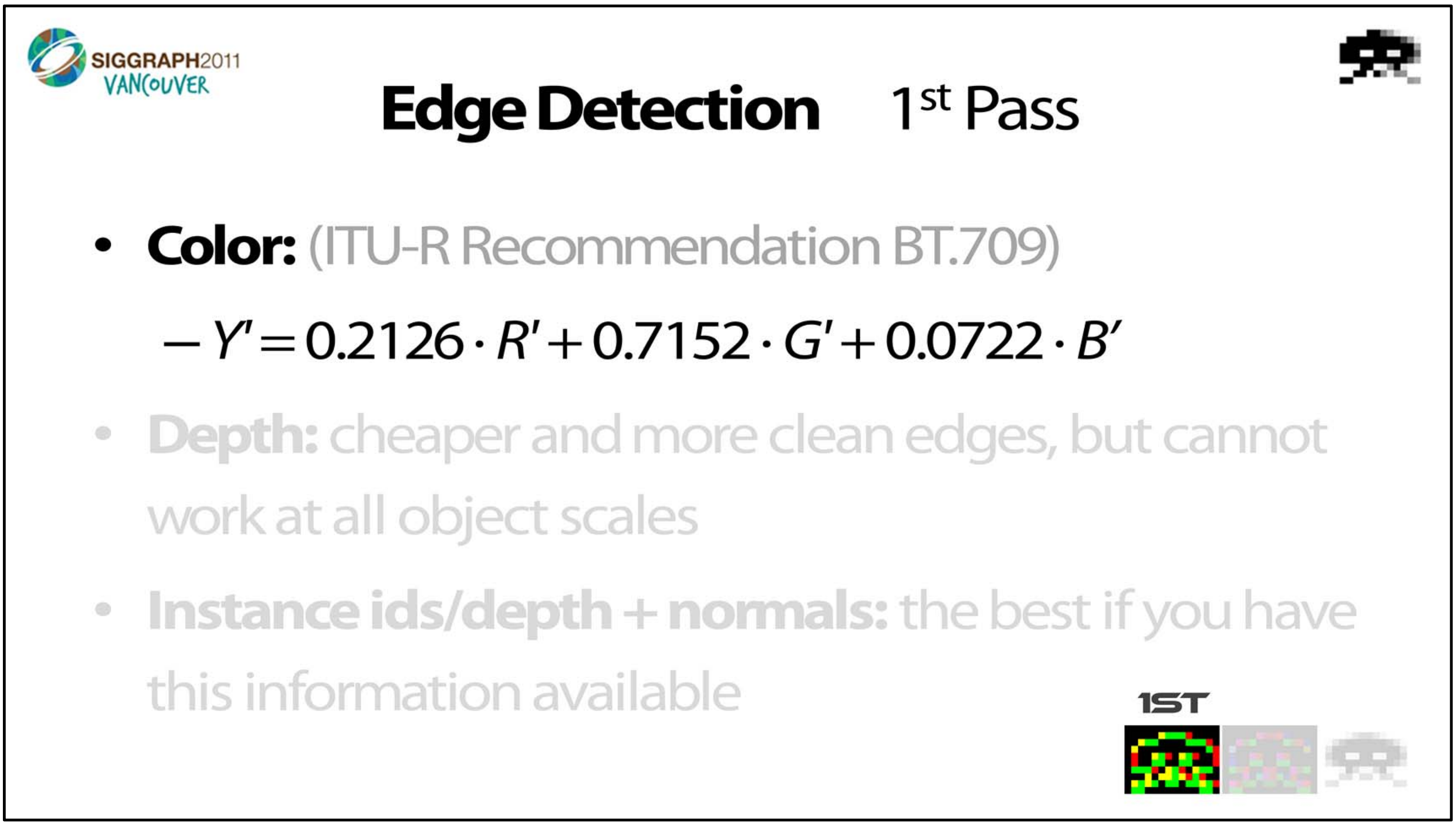
So, let's begin with the first pass.

Edge detection is a critical step for the quality of the final image.

Each undetected edge will remain aliased in the final image, so detecting all perceptible edges is crucial.

Robustness in this step is also desirable, given that good edge detection enhances temporal stability.

There are multiple options; which one is the best will depend largely on the particular scenario.



Color can be considered the most universal and easier solution, as it's always available.

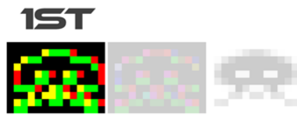
Working with color additionally provides seamless handling of shading aliasing, which may improve quality in some scenarios.

On the downside, it may introduce a slightly blur on text appearing on models, and other high-frequency features.

SIGGRAPH2011 VANCOUVER

Edge Detection 1st Pass

- **Color:** (ITU-R Recommendation BT.709)
  - $Y' = 0.2126 \cdot R' + 0.7152 \cdot G' + 0.0722 \cdot B'$
- **Depth:** cheaper and more clean edges, but cannot work at all object scales
- **Instance ids/depth + normals:** the best if you have this information available



Depth, normals or object IDs can also be used,

as they are better estimators for geometrical edges,

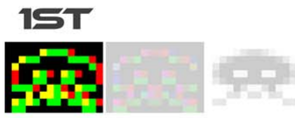
allowing to maintain the maximum image sharpness.

Using only depth is tricky as it’s really hard to manage all object scales properly, as Pete will show later on.

SIGGRAPH2011 VANCOUVER

Edge Detection 1st Pass

- **Color:** (ITU-R Recommendation BT.709)
  - $Y' = 0.2126 \cdot R' + 0.7152 \cdot G' + 0.0722 \cdot B'$
- **Depth:** cheaper and more clean edges, but cannot work at all object scales
- **Instance ids/depth + normals:** clean, sharp but more expensive



Combining depth and instance ids with normals leads to very good results in general,

As they produce really clean and complete edges,

Managing to preserve most of the image sharpness.

However the edge detection pass is more expensive given the extra work required.

Sometimes they also introduce artifacts that color doesn't, but explaining them are out of the scope of this talk.

SIGGRAPH2011 VANCOUVER

Edge Detection 1st Pass

- Color version:

```
float4 ColorEdgeDetectionPS(float4 position : SV_POSITION,
                            float2 texcoord : TEXCOORD0) : SV_TARGET {
  float3 weights = float3(0.2126,0.7152, 0.0722);

  /**
   * Luma calculation requires gamma-corrected colors, and thus 'colorTex' should
   * be a non-sRGB texture.
   */
  float L = dot(colorTex.SampleLevel(PointSampler, texcoord, 0).rgb, weights);
  float Lleft = dot(colorTex.SampleLevel(PointSampler, texcoord, 0, -int2(1, 0)).rgb, weights);
  float Ltop  = dot(colorTex.SampleLevel(PointSampler, texcoord, 0, -int2(0, 1)).rgb, weights);
  float Lright = dot(colorTex.SampleLevel(PointSampler, texcoord, 0, int2(1, 0)).rgb, weights);
  float Lbottom  = dot(colorTex.SampleLevel(PointSampler, texcoord, 0, int2(0, 1)).rgb, weights);

  float4 delta = abs(L.xxxx - float4(Lleft, Ltop, Lright, Lbottom));
  float4 edges = step(threshold.xxxx, delta);

  if (dot(edges, 1.0) == 0.0)
    discard;

  return edges;
}
```

1ST

Here you can see the simplest form of edge detection, using color input data.

It uses five [click] memory accesses and a few arithmetic operations.

In platforms where Gather4 is available…

Edge Detection 1st Pass

- Color version:

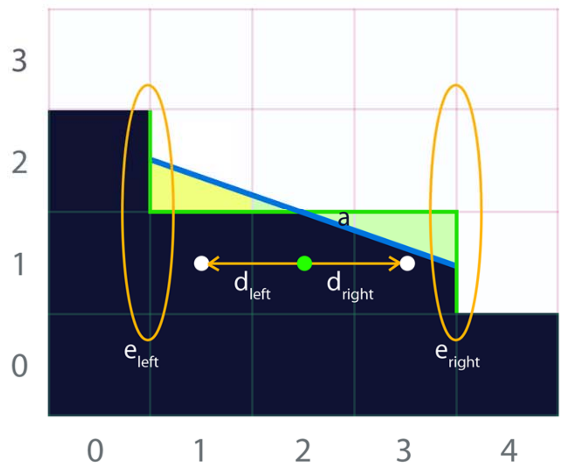
```
float4 ColorEdgeDetectionPS(float4 position : SV_POSITION,
                            float2 texcoord : TEXCOORD0) : SV_TARGET {


  /**
   * Luma calculation requires gamma-corrected colors, and thus 'colorTex' should
   * be a non-sRGB texture.
   */
  float4 topLeft = lumaTex.Gather(LinearSampler, texcoord + PIXEL_SIZE * float2(-0.5, -0.5), 0);
  float4 bottomRight = lumaTex.Gather(LinearSampler, texcoord + PIXEL_SIZE * float2(0.5, 0.5), 0);
  float L = topLeft.g;
  float Lleft = topLeft.r;       float Ltop = topLeft.b;
  float Lright = bottomRight.b;  float Lbottom = bottomRight.r;

  float4 delta = abs(L.xxxx - float4(Lleft, Ltop, Lright, Lbottom));
  float4 edges = step(threshold.xxxx, delta);

  if (dot(edges, 1.0) == 0.0)
    discard;

  return edges;
}
```

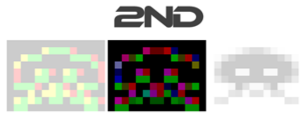
1ST

…we can reduce the number of accesses to 3 and remove all the dot products, given the lumas are precomputed

Blending Weights Calculation 2nd Pass

- It consists on three parts:
  - ★ Searching for distances d_{left} and d_{right} to the end of current line
  - ★ Fetching *crossing* edges e_{left} and e_{right}
  - ★ Calculating the coverage a of this pixel, using d and e





So, in this second pass we want to calculate the areas under the revectorized line.

In this pass, we have to search for the ends of the line, to fetch the crossing edges and to use this information to calculate the coverage area of this pixel.

Searching for Distances

- Done by exploiting bilinear filtering:

Searching line ends is a memory-intensive task.

So, to improve the bandwidth usage, we leverage the fact that bilinear filtering is free on most platforms.

In this image you have the edge marked on blue, with the colors of the dots representing the values of the edge buffer.

So, starting from here [click], we are going to jump two pixels at time, just between them…

Searching for Distances

- Done by exploiting bilinear filtering:

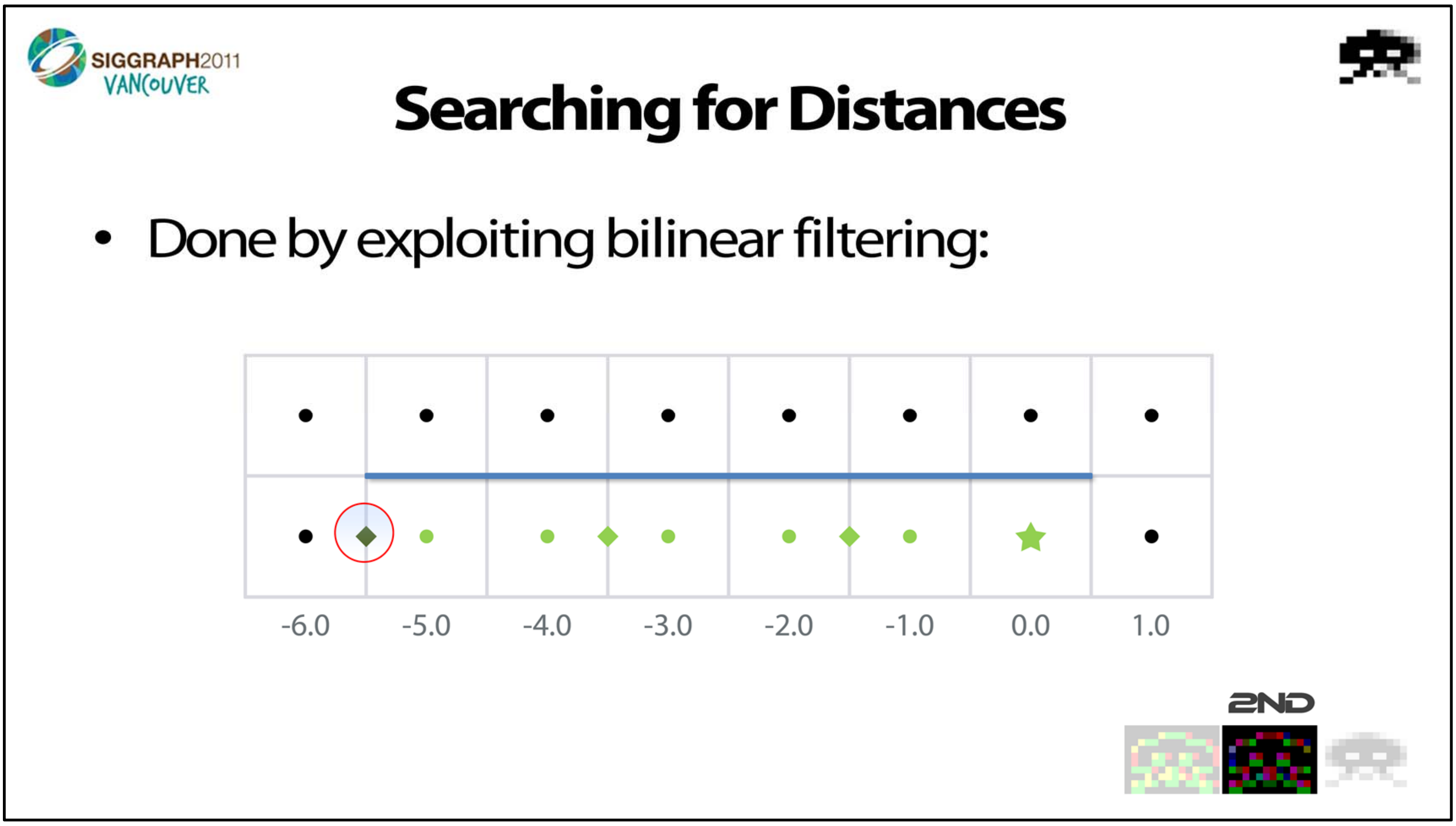
This rombus represent the first fetch.

Bilinear filtering returns one, so there is an edge in both pixels.

Searching for Distances

- Done by exploiting bilinear filtering:

The same here…



But, in this fetch we obtain 0.5, which means that one of the edges is not active, and thus the search has finished.

By using bilinear filtering in this way, we are able to accelerate searches by a factor of two, allowing to reach really long distance searches without performance drop outs.

Searching for Distances

- Search example code:

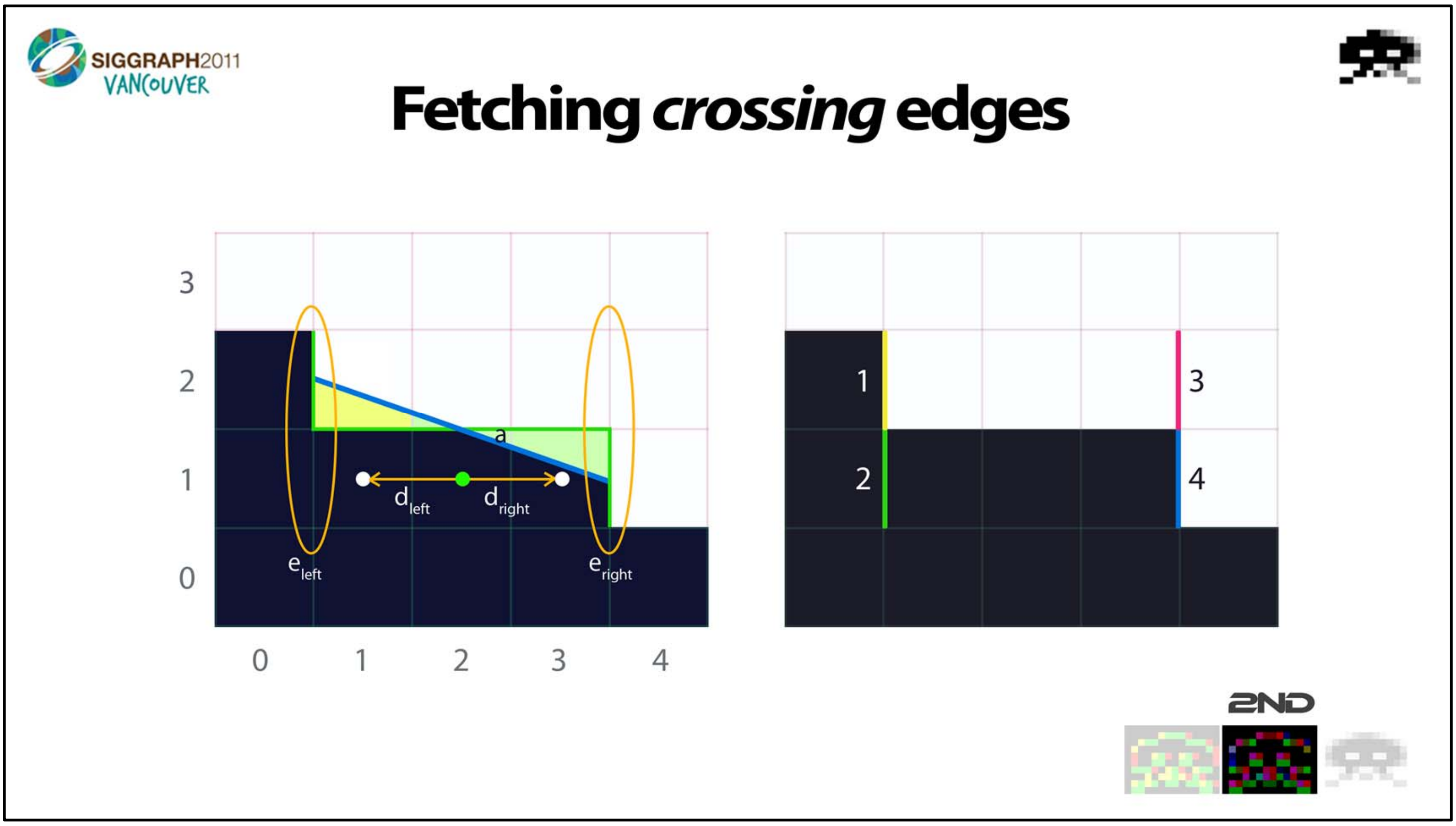
```
float SearchXLeft(float2 texcoord) {
  texcoord -= float2(1.5, 0.0) * PIXEL_SIZE;
  float e = 0.0;

  // We offset by 0.5 to sample between edgels, thus fetching two in a row
  for (int i = 0; i < maxSearchSteps; i++) {
      e = edgesTex.SampleLevel(LinearSampler, texcoord, 0).g;

      // We compare with 0.9 to prevent bilinear access precision problems
      [flatten] if (e < 0.9) break;
      texcoord -= float2(2.0, 0.0) * PIXEL_SIZE;
  }

  // When we exit the loop without founding the end, we want to return
  // -2 * maxSearchSteps
  return max(-2.0 * i - 2.0 * e, -2.0 * maxSearchSteps);
}
```

2ND

Here, you have the simple code that handles the searches to the left.

You can see how we jump two pixels at time [click].

And, how we stop when the fetch returns something else than one [click]

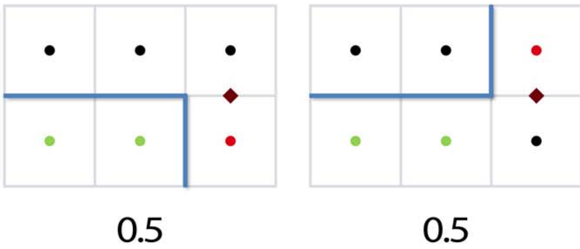


Once we have the distances to the ends of the line, we use them to obtain the crossing edges.

A naïve approach for fetching the crossing edges would imply querying the four edges.

Fetching *crossing* edges

- Again, done by exploiting bilinear filtering



Instead, a more efficient approach is to use bilinear filtering for fetching both edges at a time, similar to the distance search.

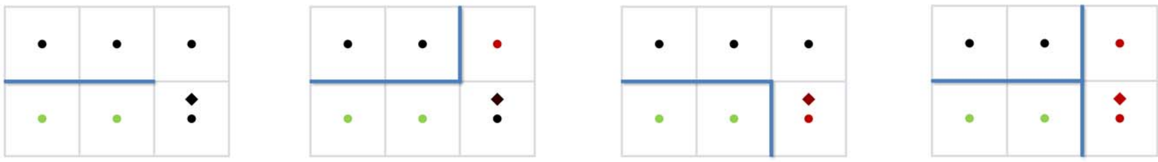
But, now there is a little problem.

When the return value is 0.5, we are dealing with this case [click]… or this other one [click]?

SIGGRAPH2011 VANCOUVER

Fetching *crossing* edges

★ **Solution: offset the coordinates!**



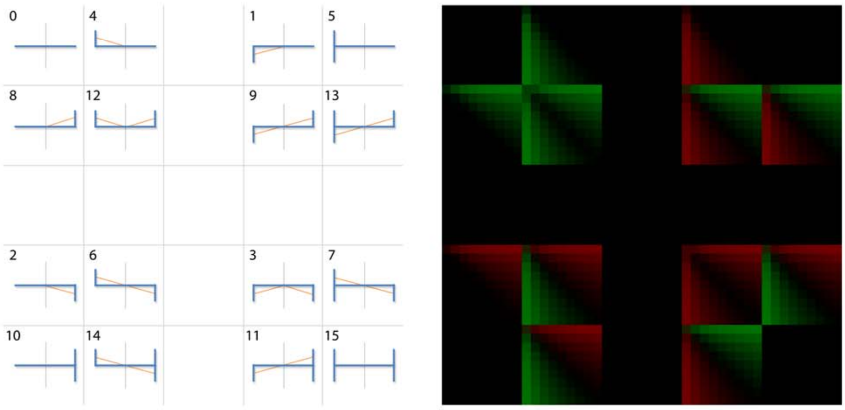
0.0 0.25 0.75 1.0

2ND

The solution is to offset the query by 0.25 allowing to distinguish them, as the returned value is different in each case.

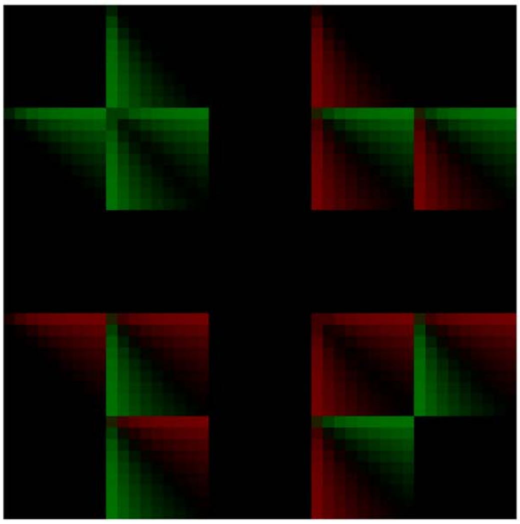
Calculating the coverage

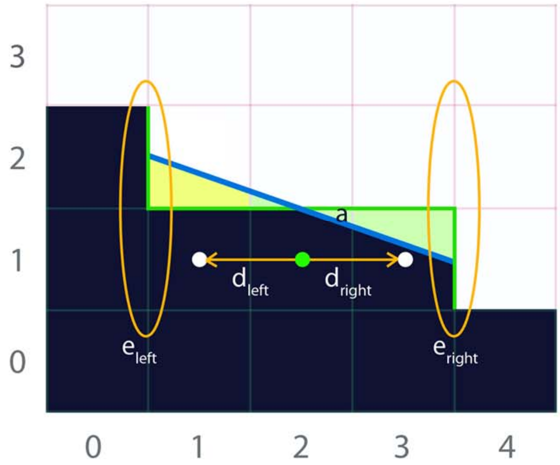
- We use a pre-computed texture to avoid:
  - ★ Dynamic branching
  - ★ Expensive area calculations



A key contribution in the coverage calculation is that, instead of handling sixteen different patterns, we precompute them in a 4D texture [click], which avoids branching code.

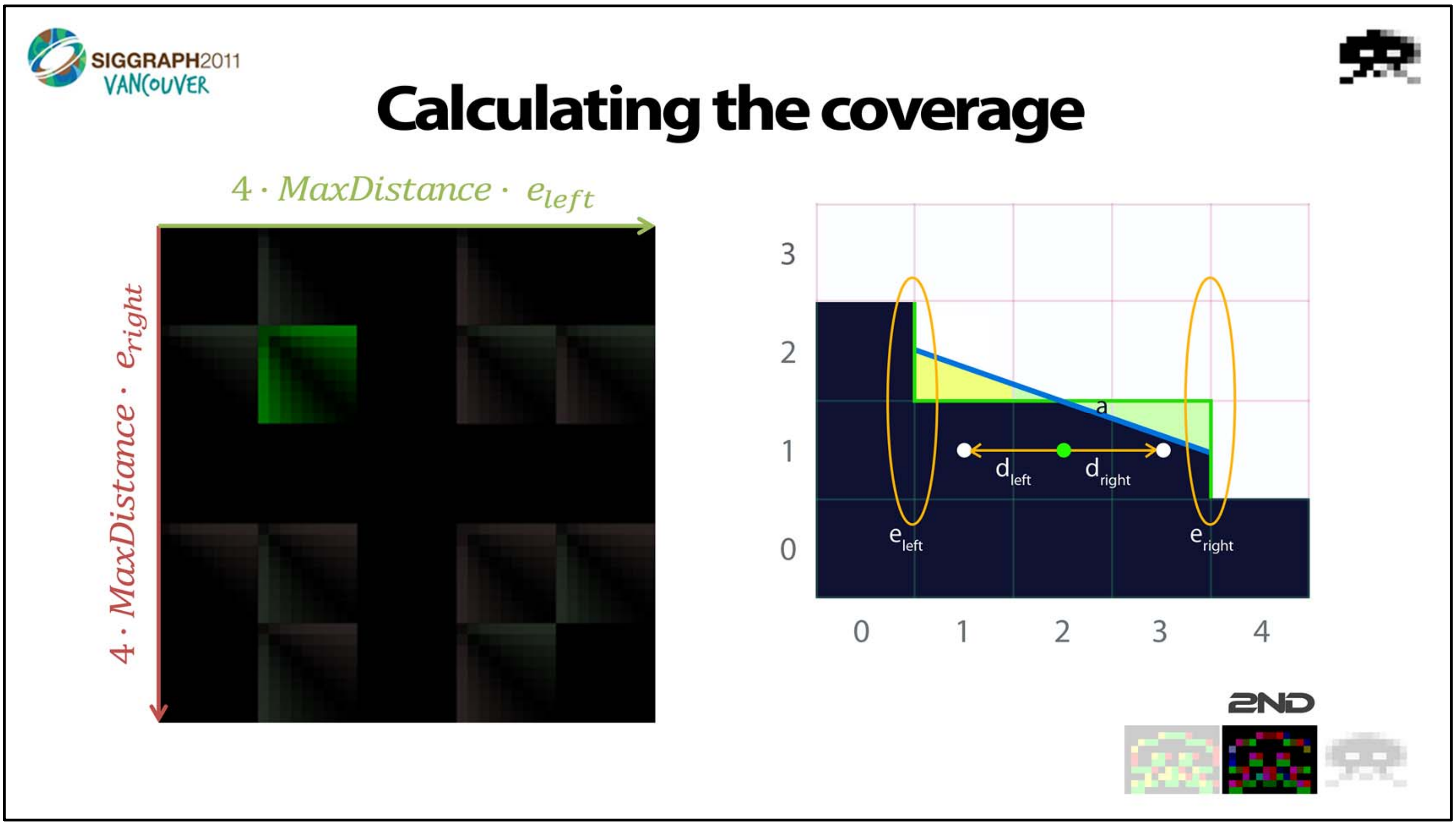
Calculating the coverage



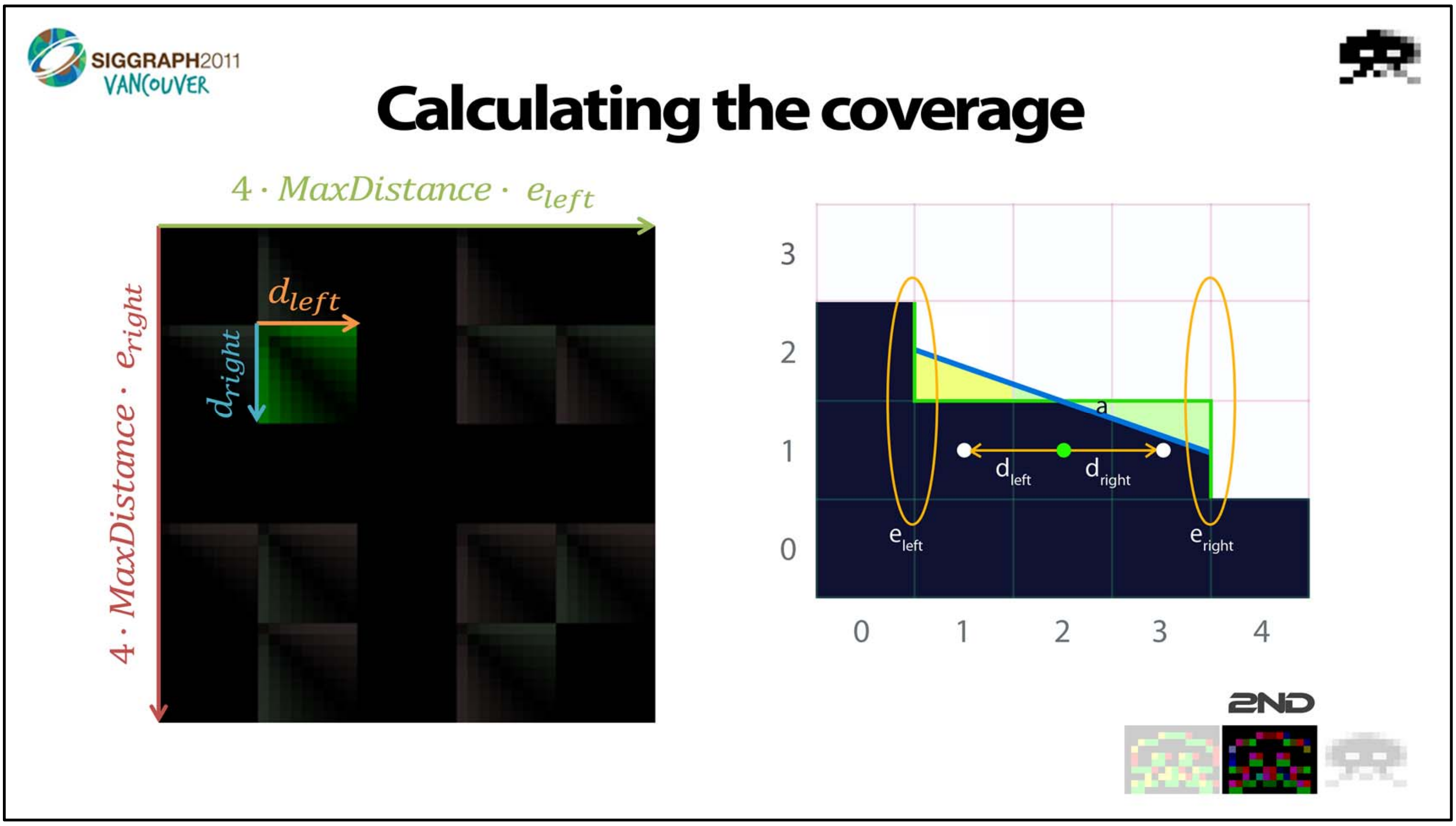


Accessing this texture is as follows.

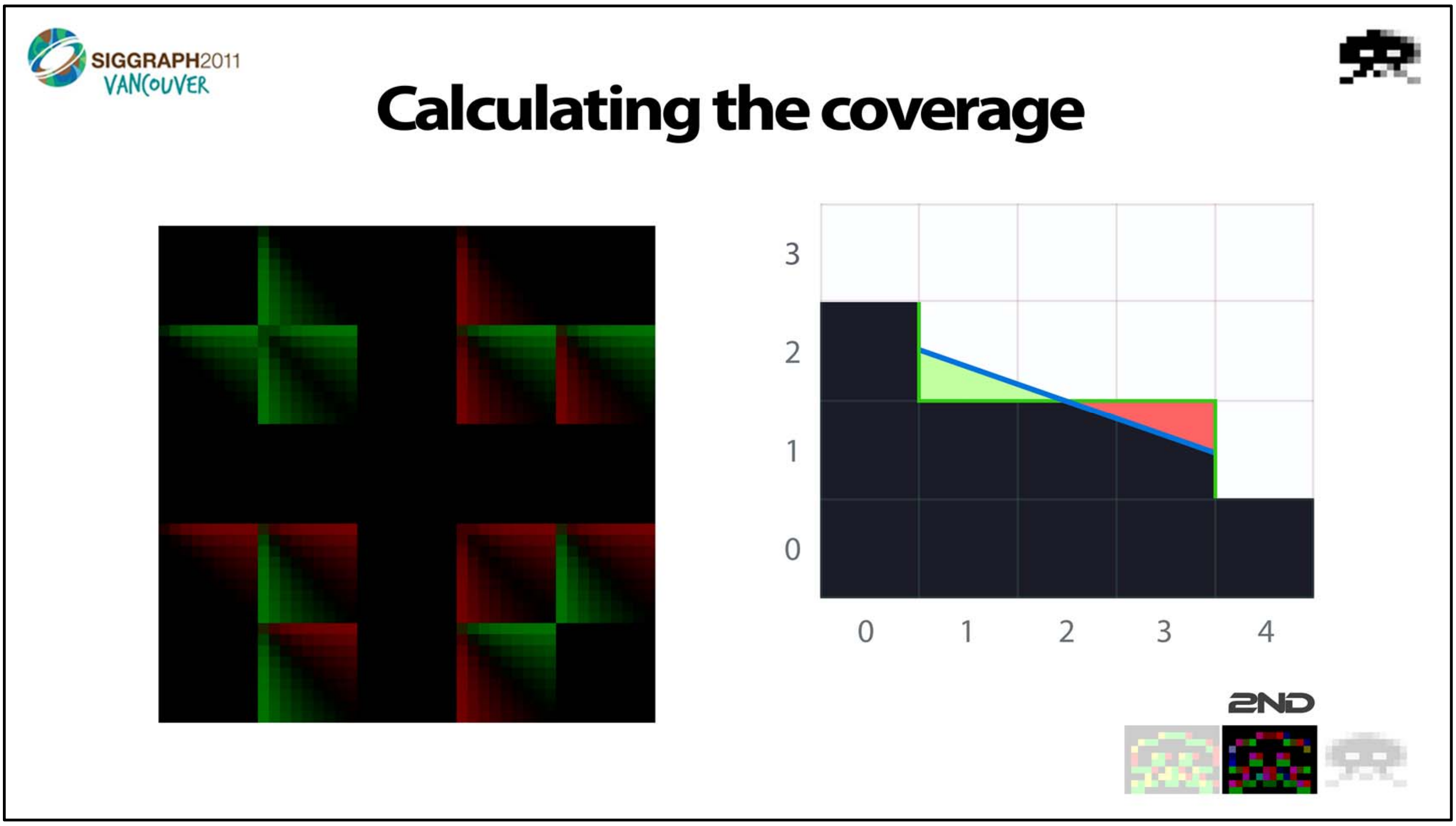
First, the pattern type, or block, is selected…



…by using the crossing edges information.



Then, the proper area is selected by using the distances to the end of the line.



You may be thinking why the precomputed area texture has two channels:

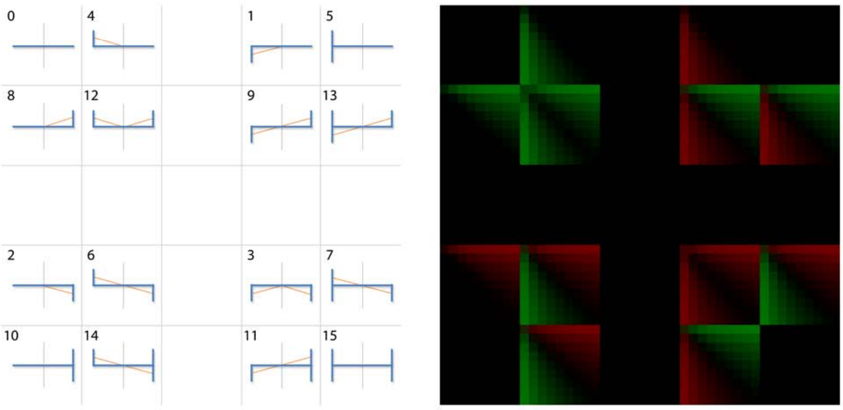
The answer is rather easy, having two channels allows to disambiguate the blend direction.

Red values means the bottom pixel blends with the top one.

While the green values just the opposite.

Area Texture Advatanges

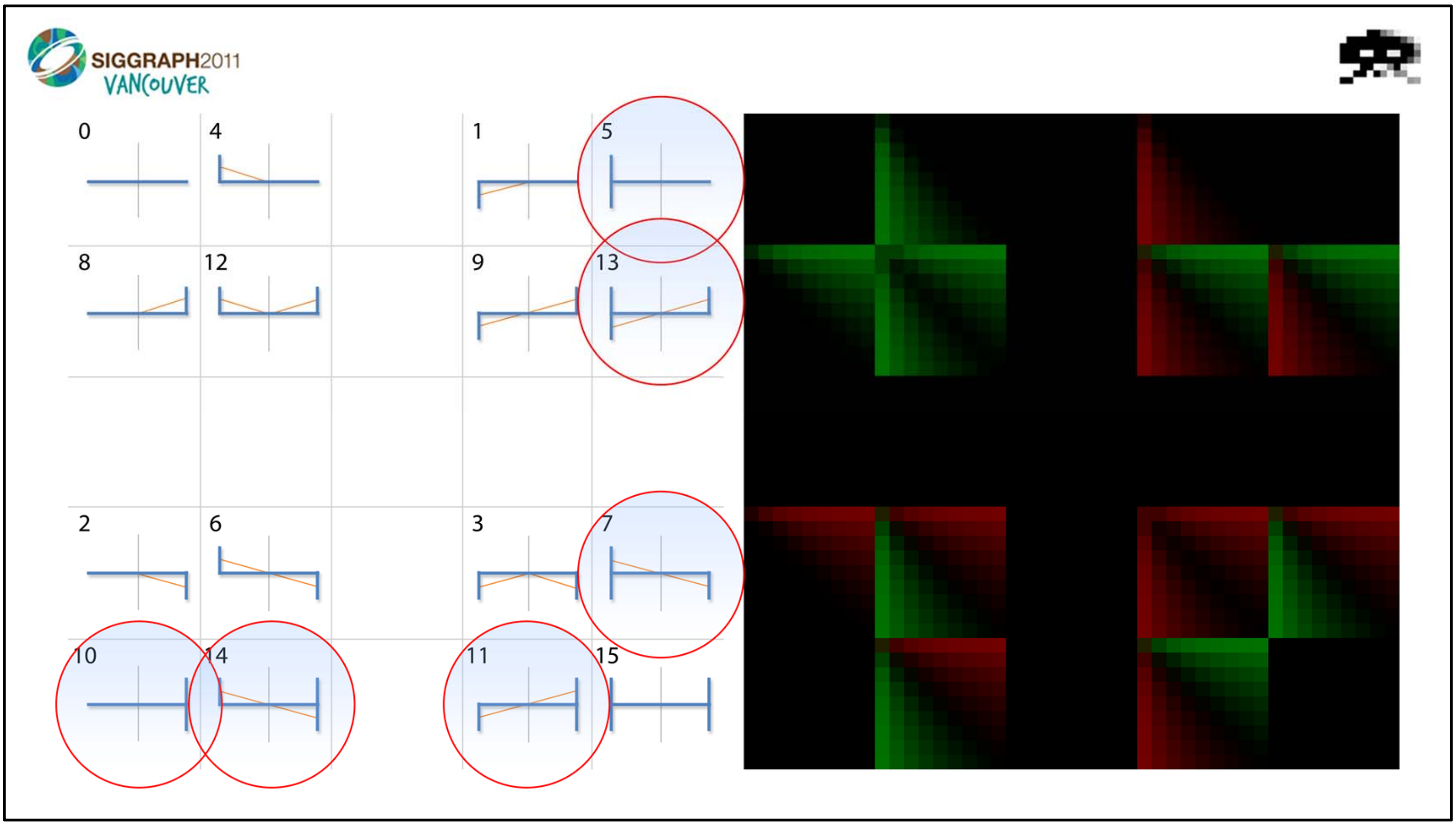
- Symetrical handling of patterns
- Asymetric revectorization → patterns fine tuning



Using a precomputed area has multiple advantages,

Including the fact that all patterns are handled in the same way: a simple texture access.

On the other hand, the texture itself can be customized, allowing to fine tune certain patterns.



We avoid filtering these patterns [click], as their revectorization usually introduces artifacts.

We also customized the revectorization of others [click], triting them as simple Z patterns.

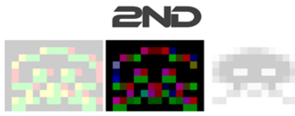
This fine tunings help to maintain the image as pristine as possible,

while still delivering high quality antialiasing where needed.

Blending Weights Calculation 2nd Pass

- Shader code (I):

```
float4 BlendingWeightCalculationPS(float4 position : SV_POSITION,
                                   float2 texcoord : TEXCOORD0) : SV_TARGET {
  float4 weights = 0.0;

  float2 e = edgesTex.SampleLevel(PointSampler, texcoord, 0).rg;

  [branch]
  if (e.g) { // Edge at north

    // Search distances to the left and to the right:
    float2 d = float2(SearchXLeft(texcoord), SearchXRight(texcoord));

    // Now fetch the crossing edges. Instead of sampling between edgels, we
    // sample at -0.25, to be able to discern what value has each edgel:
    float4 coords = mad(float4(d.x, -0.25, d.y + 1.0, -0.25),
                        PIXEL_SIZE.xyxy, texcoord.xyxy);
    float e1 = edgesTex.SampleLevel(LinearSampler, coords.xy, 0).r;
    float e2 = edgesTex.SampleLevel(LinearSampler, coords.zw, 0).r;

    // Ok, we know how this pattern looks like, now it is time for getting
    // the actual area:
    weights.rg = Area(abs(d), e1, e2);
  }
```



Here you have the code for the whole pass.

We first search for line ends [click].

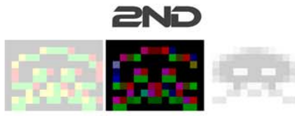
Then fetch the crossing edges [click].

And, finally obtain the coverage area [click].

Blending Weights Calculation 2nd Pass

- Shader code (II):

```
  [branch]
  if (e.r) { // Edge at west

    // Search distances to the top and to the bottom:
    float2 d = float2(SearchYUp(texcoord), SearchYDown(texcoord));

    // Now fetch the crossing edges (yet again):
    float4 coords = mad(float4(-0.25, d.x, -0.25, d.y + 1.0),
                        PIXEL_SIZE.xyxy, texcoord.xyxy);
    float e1 = edgesTex.SampleLevel(LinearSampler, coords.xy, 0).g;
    float e2 = edgesTex.SampleLevel(LinearSampler, coords.zw, 0).g;

    // Get the area for this direction:
    weights.ba = Area(abs(d), e1, e2);
  }

  return weights;
}
```

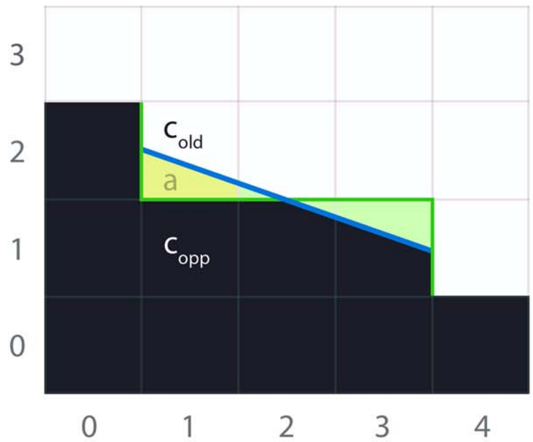


This shows the code for the vertical case, which is quite similar, as you can see.

SIGGRAPH2011 VANCOUVER

Neighborhood Blending 3rd Pass

- We blend with the neighborhood using the areas calculated in previous pass



$$c_{new} = (1 - a) \cdot c_{old} + a \cdot c_{opp}$$

3RD

Finally, we want to blend pixel on edges using the areas calculated on the previous pass.

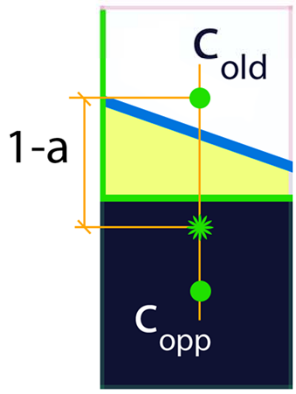
For example, for the case of the pixel c_old, we got the area a from the edge on the bottom.

The blending required [click] is similar to 1D bilinear filtering.

Neighborhood Blending 3rd Pass

- We leverage bilinear filtering (yet again):

$$c_{new} = (1 - a) \cdot c_{old} + a \cdot c_{opp}$$



So, we again leverage bilinear filtering, using it to implement the blending equation.

Neighborhood Blending 3rd Pass

- Shader code (I):

```
float4 NeighborhoodBlendingPS(float4 position : SV_POSITION,
                              float2 texcoord : TEXCOORD0) : SV_TARGET {
  // Fetch the blending weights for current pixel:
  float4 topLeft = blendTex.SampleLevel(PointSampler, texcoord, 0);
  float bottom = blendTex.SampleLevel(PointSampler, texcoord, 0, int2(0, 1)).g;
  float right = blendTex.SampleLevel(PointSampler, texcoord, 0, int2(1, 0)).a;
  float4 a = float4(topLeft.r, bottom, topLeft.b, right);

  // Up to 4 lines can be crossing a pixel (one in each edge). So, we perform
  // a weighted average, where the weight of each line is 'a' cubed, which
  // favors blending and works well in practice.
  float4 w = a * a * a;
```

3RD

Here [click], we can see how we fetch the areas of the four possible lines.

Neighborhood Blending 3rd Pass

- Shader code (II):

```
  // There is some blending weight with a value greater than 0.0?
  float sum = dot(w, 1.0);
  [branch]
  if (sum > 0.0) {
    float4 o = a * PIXEL_SIZE.yyxx;
    float4 color = 0.0;

    // Add the contributions of the possible 4 lines that can cross this
    // pixel:
    color = mad(colorTex.SampleLevel(LinearSampler, texcoord + float2( 0.0, -o.r), 0), w.r, color);
    color = mad(colorTex.SampleLevel(LinearSampler, texcoord + float2( 0.0,  o.g), 0), w.g, color);
    color = mad(colorTex.SampleLevel(LinearSampler, texcoord + float2(-o.b,  0.0), 0), w.b, color);
    color = mad(colorTex.SampleLevel(LinearSampler, texcoord + float2( o.a,  0.0), 0), w.a, color);

    // Normalize the resulting color and we are finished!
    return color / sum;
  } else {
    return colorTex.SampleLevel(LinearSampler, texcoord, 0);
  }
}
```

And here [click], we blend with the neighbors, averaging the result of the four possible lines that can cross the pixel.

SRGB and linear blending

- Blending should be done in linear space

MLAA
(Gamma Blend)

MSAA 8x

For the most accurate results, this blending should be done in linear space.

[back and forth] You can see that the differences between gamma and linear are subtle… yet apparent.

Using SRGB buffers and DirectX 10 will ensure blending is done in linear space.

SRGB and linear blending

- Blending should be done in linear space.

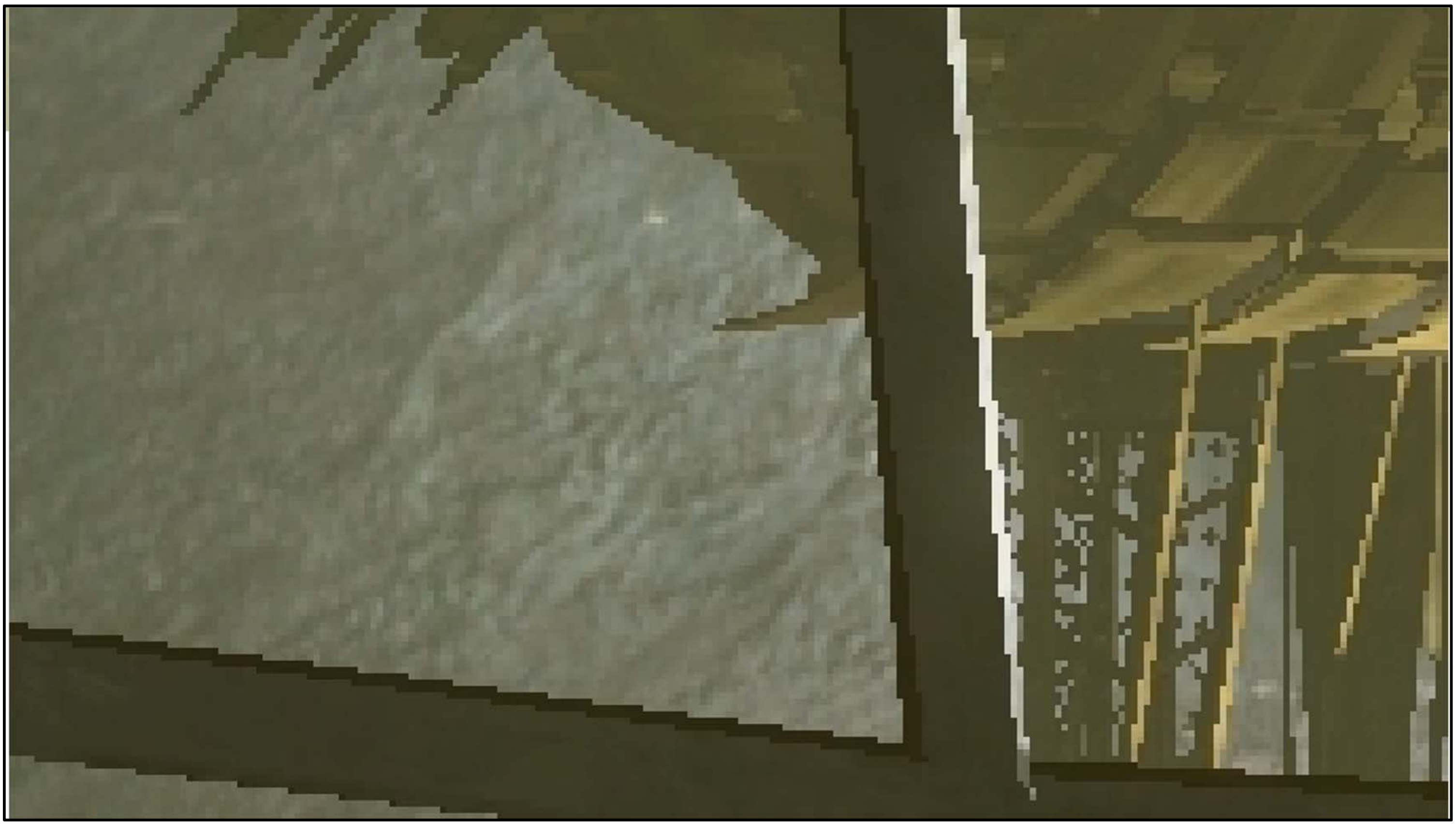
MLAA
(Linear Blend)

MSAA 8x

For the most accurate results, this blending should be done in linear space.

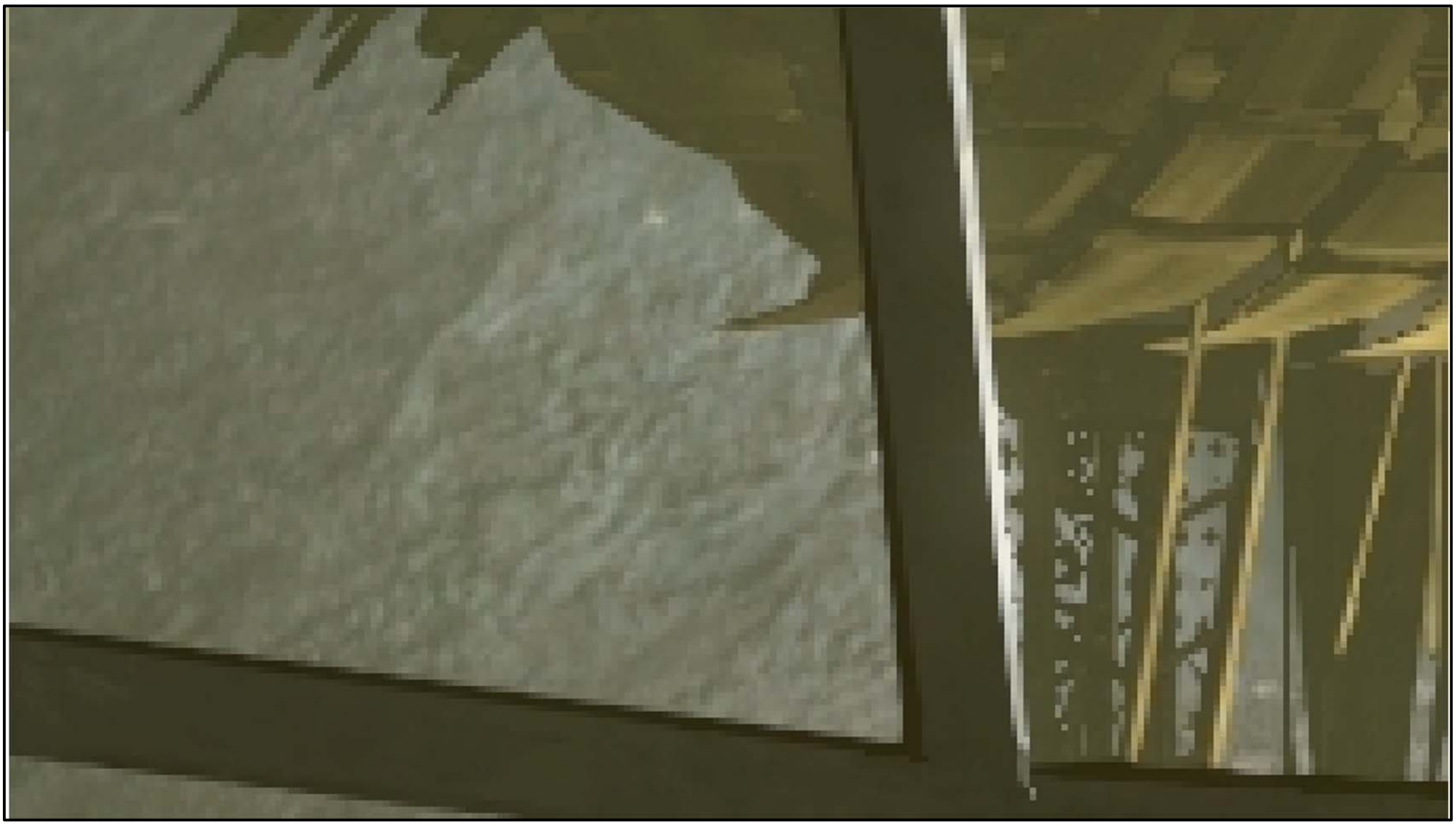
[back and forth] You can see that the differences between gamma and linear are subtle… yet apparent.

Using SRGB buffers and DirectX 10 will ensure blending is done in linear space.



I encourage you to visit our page for looking at the results, but nevertheless, I’m going to show some of them.

Here you have the first one.



Where you can appreciate the accuracy of color edge detection.

And here, an interesting image from Unigine.

As you can see [back and forth], some texture detail is lost when using color edge detection.

And here, an interesting image from Unigine.

As you can see [back and forth], some texture detail is lost when using color edge detection.

[redacted]

On the other hand, when using depth based edge detection…

The texture detail is perfectly preserved.

On the other hand, when using depth based edge detection…

The texture detail is perfectly preserved.

However, some zones with little depth deltas will not be antialiased.

How to combine the best of both worlds, will be covered by Tobias later on.

SIGGRAPH2011 VANCOUVER

Performance

0.28ms@720 on a GeForce GTX 470

0.37ms@720 if Gather4 is not used

Assasin's Creed	0,368	0,10
Bioshock	0,352	0,12
Crysis	0,348	0,12
Dead Space	0,312	0,12
Devil May Cry 4	0,256	0,05
GTA IV	0,234	0,01
Modern Warfare 2	0,248	0,02
NFS Shift	0,26	0,04
Split / Second	0,276	0,04
S.T.A.L.K.E.R.	0,29	0,04
Grand Average	0,2944	0,08

Measured with screen captures

So, this is all good, it’s really fast, it works fine [click], and this technique is already been used in several games.

However we have much better news, we have just made public a technical report about…

SMAA

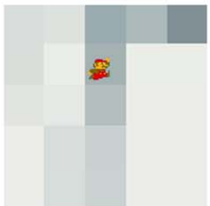
Subpixel Morphological Antialiasing

What we called Subpixel Morphological Antialiasing.

SMAA:
Subpixel Morphological Antialiasing

Temporal AA
Great subpixel features!

MLAA
Great gradrients!



MSAA
Great subpixel features!

Some months ago we were thinking:

Morphological Antialiasing is good [click]

Temporal Antialiasing is also good thing [click]

And multisampling is very good [click]

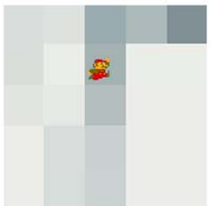
So, why not combine them into a single technique?

SIGGRAPH2011 VANCOUVER

SMAA:
Subpixel Morphological Antialiasing

Temporal AA
Good subpixel features!

MLAA
Great gradients!



MSAA
Great subpixel features!

…

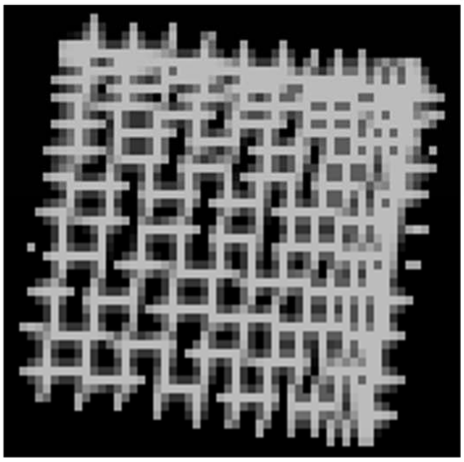
But, well… it’s not as easy as it sounds.

SMAA:
Subpixel Morphological Antialiasing

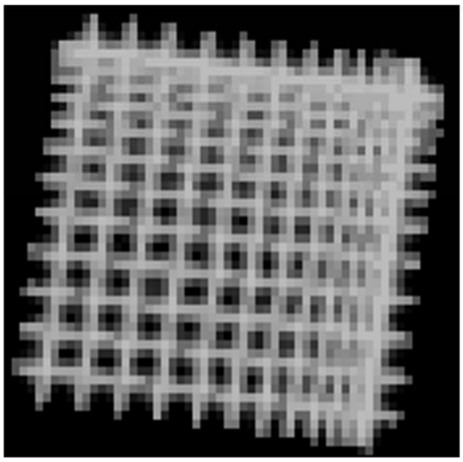
MLAA | **Post-Resolve MLAA** | **MSAA 2x**

If you try to apply morphological antialiasing after resolving, it won't be able to generate proper gradients, given the edges are now smoother and harder to detect.

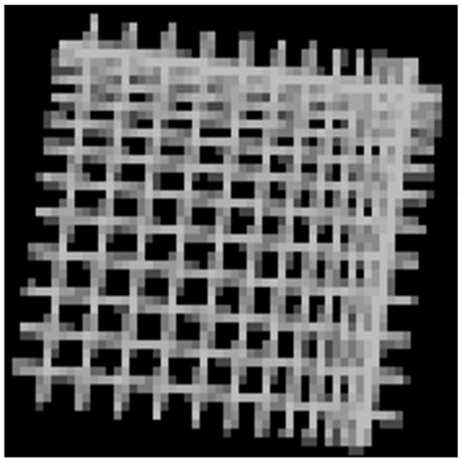
SMAA:
Subpixel Morphological Antialiasing



MLAA



Pre-Resolve
MLAA

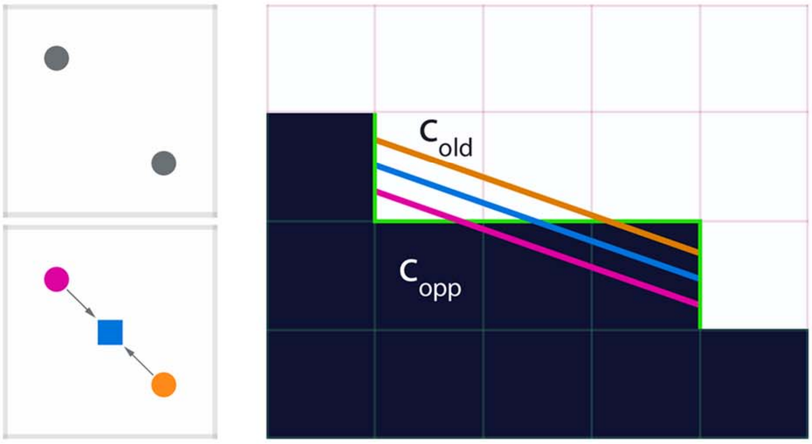


MSAA 8x

And if you try to apply it before resolving, it's an improvement over MLAA but it is still not that good, as there is too much blur.

SIGGRAPH2011 VANCOUVER

SMAA:
Subpixel Morphological Antialiasing



I won’t enter in details, but by offseting the revectorizations to match the subpixel positions, we managed to obtain much better results.

Figure 13: *Comparison of the features (rows) of our approach with a selection of anti-aliasing techniques (columns). FXAA I (preset 3) has been used for all the images with the exception of accurate searches, where preset 2 has been selected to exemplify accelarated searches failure cases. Green, orange and red dots mark accurate, regular and innacurate handling of a feature. Zoom the electronic version of this paper to see the details. Close-ups on last row taken from Assassin's Creed Brotherhood, courtesy of Ubisoft.*

11

We encourage you to download the technical report for a very exhaustive details.

SMAA:
Subpixel Morphological Antialiasing

- Better fallbacks: everything is processed
  - ★ Zones with low contrast are antialiased by multi/supersampling

Being composed of three components, when one of them fails, the other two serve as fallback.

In this example, we can see how low contrast zones, usually ignored by morpholohical approches are handled by our spatial multisampling component.

SMAA:
Subpixel Morphological Antialiasing

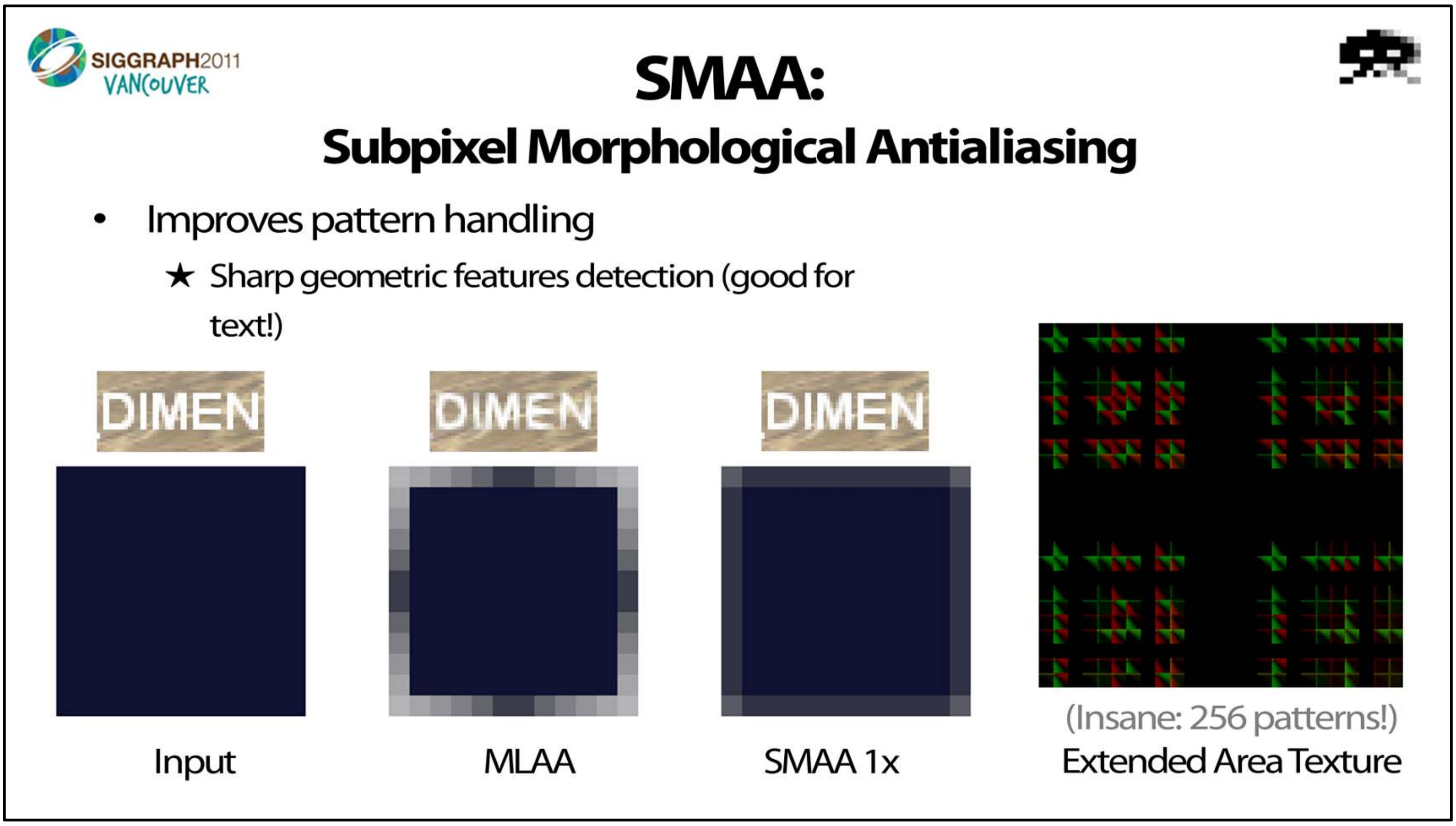
- Improves pattern handling
  ★Diagonals

MLAA

SMAA S2x

SSAA 16x

We also achieve better handling of diagonals…

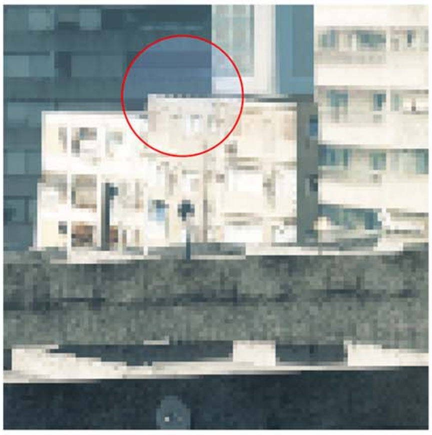


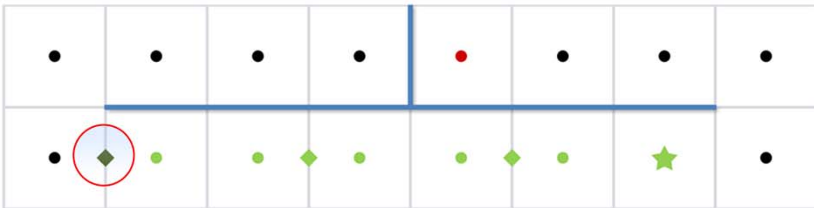
and improved sharp geometric features, which allows to avoid the general roundness introduced by MLAA.

It also enables better text handling, by the cost of just two additional memory accesses.

SMAA:
Subpixel Morphological Antialiasing

- Accurate searches



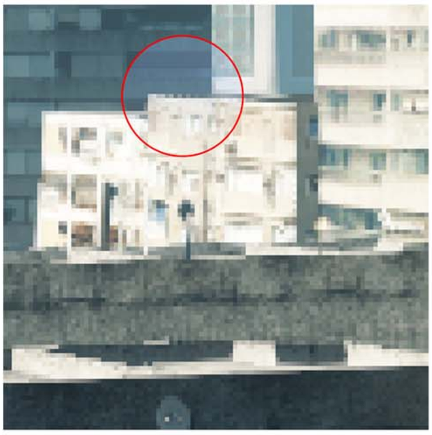


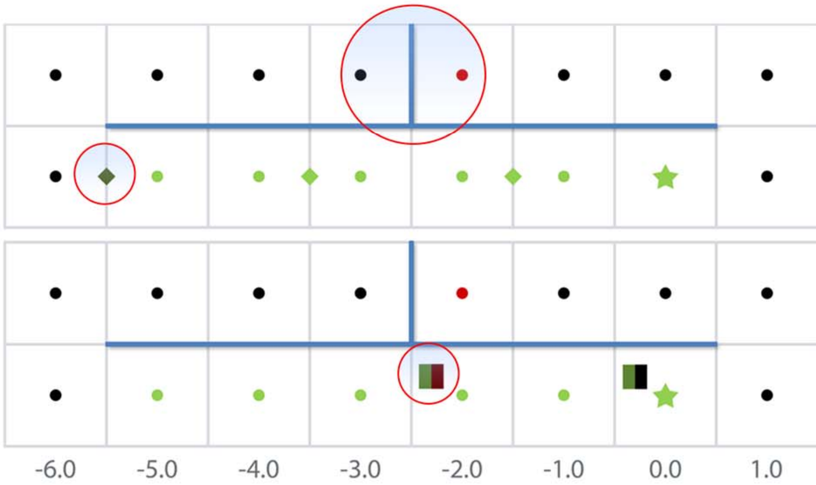
Our simplified search scheme, while very fast, sometimes introduced artifacts in form of dithering.

This is, due to the fact that, as we sample in the middle between pixels [click], the last step is ambiguous.

SMAA:
Subpixel Morphological Antialiasing

- Accurate searches





So by sampling at different offsets in the x and y directions [click], we stop at the appropriate moment without incurring into any additional overhead.

SIGGRAPH2011 VANCOUVER

SMAA:
Subpixel Morphological Antialiasing

- Take neighborhood lumas into account

Another feature we introduced is what we call local contrast awareness.

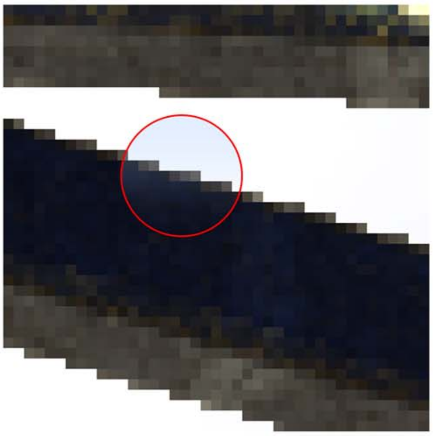
A big drawback of MLAA approaches is that they usually consider edges binary: they are either on or off.

Sometimes this can be an issue, because the actual strength of an edge is important.

For example, in this image we can see there are gradients in the silhouette of this object…

SMAA:
Subpixel Morphological Antialiasing

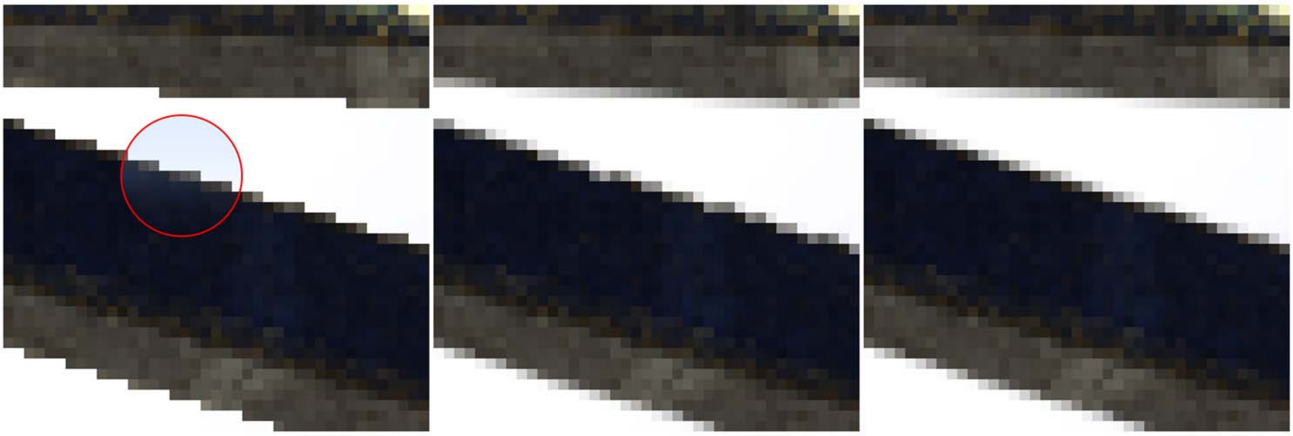
- Take neighborhood lumas into account



…which confuse MLAA's search scheme [click], as it will see crossing edges that will make it stop earlier than desired.

SMAA:
Subpixel Morphological Antialiasing

- Take neighborhood lumas into account



However, perceptually we ignore these crossing edges because the contrast with the white background is much, much, higher.

So, what we did is to mimic our visual system and ignore edges which have low contrast with respect to the neighbors.

SIGGRAPH2011 VANCOUVER

What's under the hood

- **High Quality**
  - ★ 16× gradients (or more!)
  - ★ Noise proof → **Temporally Stable**
  - ★ Sharpness preservation

- **What's under the hood**
  - ★ The fast and accurate distance searches
  - ★ The local contrast awareness
  - ★ Specific patterns tuning
  - ★ Calculating the four possible lines that can cross a pixel, and smartly average them

We believe the key component of our technique is the heuristics we are using for determining which is the best pattern revectorization for a pixel, and to stick with it over time.

SIGGRAPH2011 VANCOUVER

SMAA:
Subpixel Morphological Antialiasing

- Modular
  - ★ SMAA 1x — improved pattern handling
  - ★ SMAA T2x — 1x + temporal supersampling
  - ★ SMAA S2x — 1x + spatial multisampling
  - ★ SMAA 4x — 1x + 2x temporal supersampling + 2x spatial multisampling
- Performance of SMAA 4x (on a GTX 580)
  - ★ **1.06** ms @1080p
  - ★ **0.5** ms@720p — *both not taking into account 2x render overhead*

The technique is modular, so you can turn on features selectively, easily adapting it to the available budget.

In fact, most of the improvements just add a few lines to our current MLAA shader.

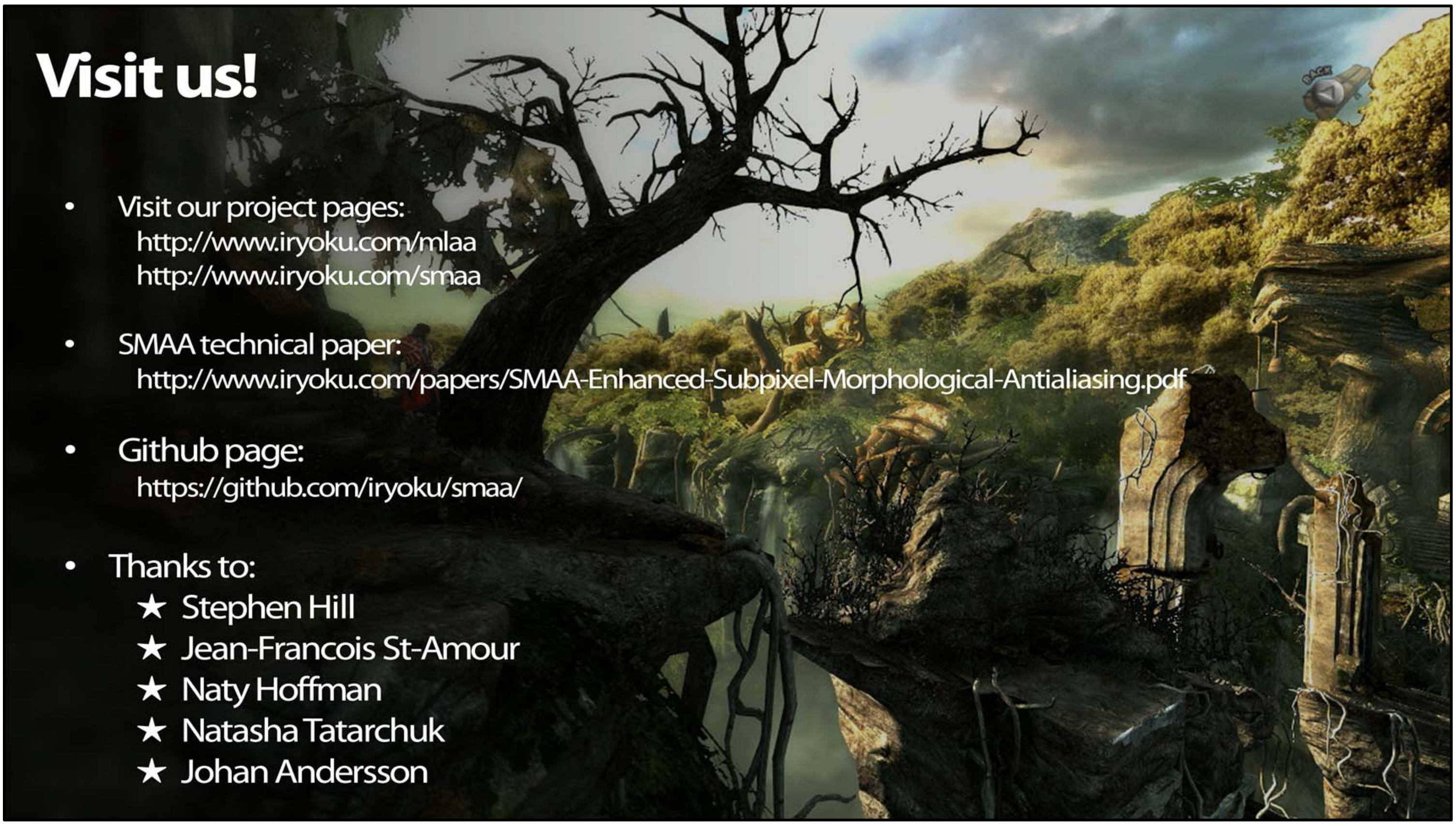
So, upgrading to SMAA is going to be a rather easy task.

SIGGRAPH2011 VANCOUVER

SMAA:
Subpixel Morphological Antialiasing

- Modular
  - ★ SMAA 1x — improved pattern handling
  - ★ SMAA T2x — 1x + temporal supersampling
  - ★ SMAA S2x — 1x + spatial multisampling
  - ★ **SMAA 4x — 1x + 2x temporal supersampling + 2x spatial multisampling**
- Performance of SMAA 4x (on a GTX 580)
  - ★ **1.06** ms @1080p
  - ★ **0.5** ms@720p — *both not taking into account 2x render overhead*

And finally, the performance of the highest quality profile, runs in 1 millisecond for a 720p buffer.



Sorry that we didn't give a detailed description of SMAA, but you have the technical report online, and I will be posting about each feature on my blog in the next weeks =]

My colleague Pete from Double Fine, will continue with the presentation of his Hybrid MLAA approach.

Thank you for your attention, and do not forget to visit us :-)