
SIGGRAPH2012
The 39th International Conference and Exhibition
on Computer Graphics and Interactive Techniques

Scalable High Quality Motion Blur and Ambient Occlusion

Michael Bukowski – Vicarious Visions
Padraic Hennessy – Vicarious Visions
Morgan McGuire – Vicarious Visions / Williams College
Brian Osman – Vicarious Visions

Advances in Real-Time Rendering in 3D Graphics and Games Course

Agenda

- Designing scalable algorithms
- Examples
  - Motion blur
  - Ambient occlusion
- Closing

Designing Scalable Algorithms

Advances in Real-Time Rendering in 3D Graphics and Games

Why we go for scalable

- Ability to share effects across platforms with different performance constraints
- Share across games that are going for different art styles

At Vicarious Visions we develop on platforms ranging from mobile through console to dx11.

Since we often have two or three projects running at the same time, each on multiple platforms, it is important that we invest in shared technologies.

Just as import as being able to scale in performance, our techniques must also scale in design, giving them the ability to compliment multiple art styles.

Benefits of scalable effects

- Visual consistency
- Development time savings
- Consistent content requirements
- “Free” quality gains

When talking about scalable effects in the context of one project there are a number of benefits we get:

- Visual Consistency – Using the same algorithm on every platform results in similar contributions
  - The visual consistency means that our artists will get the same results on each platform, saving them the time choice of which platform to optimize for.
- Development time savings - designing and implementing robust algorithms take time and effort, we have found that it is worth it to spend more time upfront designing for scalability than to implement 2 algorithms
- Consistent Content Requirements – if we find that we need to make some requirements on how assets are authored do support an algorithm we don’t want to have multiple requirements for different platforms
- “Free” quality gains – When you have scalability designed into an algorithm you can maintain visual consistency but increase quality as you move to new platforms and performance restrictions are lifted.

We ask ourselves the following

- Will it work on platform X?
- Where can we change parameters for increased quality?
- Where can we change parameters for increased speed?
- Does it work if we change the resolution?
- Does it depend on a hardware or API feature?
- Does it dictate an art style?

When we start working on an algorithm we have a mental checklist that we continually measure or ideas against.

An interesting exercise is to think about some algorithms that you currently have in your engine and ask these questions.

If you are a multi-project studio or part of a central tech group the most important of the questions here in terms of scalability is “Does it dictate an art style?” since this will limit the application of the approach.

Potential Scalable Elements

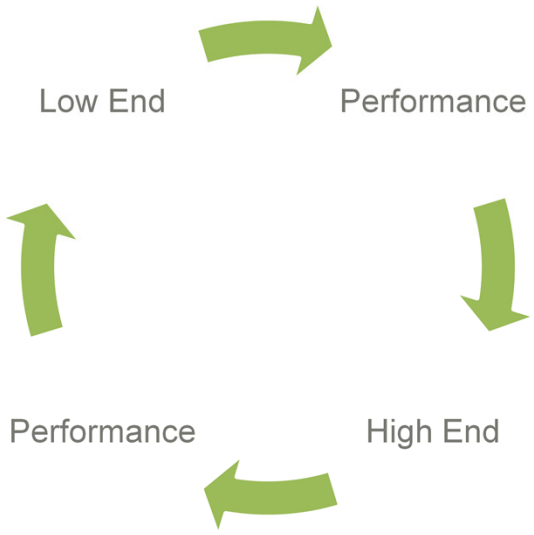
- Quality knobs
  - Maximum radius
  - Sample count
  - Clamps on inputs
- Resolution independence
  - Allow normalized screen units instead of pixels
- Modular functions
  - Accuracy vs speed
- Additional processing
  - Additional passes, iterative algorithms

Some of the common elements we considered when designing the algorithms were:

- Quality knobs – these are simply adjustable parameters that can yield higher quality. All your usual suspects are here like radius, sample counts, adjusting restrictions like clamps
- Resolution independence – pretty self explanatory, this one can be hard to do if you perform a lot of bilateral blurs or nearest neighbor sampling
- Modular Functions – Especially for lower end hardware we often have to replace some function with and approximation for performance reasons, for the quality improvements it is important to leave the option of going back to the full formula on platforms where you have some cycles to spare.
- Additional Processing – This is probably one that hasn't got much thought on the real-time side of things, but this could give you lots of quality gains if for instance you change a random seed and re-run an algorithm to reduce aliasing or if you design a function that converges on the correct solution each iteration.

Our Process

- Design for quality on low end
- Extend to high end
- Iterate on quality and performance
- Ample time between iterations



Our process has evolved over the past 2 years into something that we regularly employ now.

1) Design for the lower end platform using the types of questions asked earlier
2) Optimize the algorithm for performance and integrate it into the game
3) We then revisit the higher end platforms and implement there
    - By performing the optimization in between the low end and high end implementations we give ourselves time to take a step back and re-evaluate our approach, this often leads to major quality and performance gains

4) We take any fundamental improvements made on the higher end platform and integrate them back into the lower end targets

5) Rinse and repeat, often changing project directions and requirements will expose or remove constraints from the design

Agenda

- ~~Designing Scalable Algorithms~~
- Examples
  - Motion Blur
  - Ambient Occlusion
- Closing

Two of the first algorithms that really helped us develop this process were Motion Blur and Ambient Occlusion. We are going to take a closer look at where those algorithms are at right now.

Motion Blur

Introduction

- Convey sense of speed
- Reduces strobe artifacts
  - Especially important @ 30 fps
- Give cinematic feel of a real camera

Motion blur is a great effect to invest in to get a more cinematic feel for your games.

This is because motion blur is emulating the non-instantaneous shutter time of actual cameras.

It also helps reduces the strobe effect that happens when you render at frame rates under 60 fps, giving your game a much smoother experience.

Artistically motion blur can be used very effectively to communicate a sense of speed to the player.

Project Motivations

- Highly dynamic characters
- Dense organic environments
- Lots of scene motion

So why did we go for motion blur?

When the concepts for our game started coming in we saw many characters with exaggerated movements, dense “living” environments, and that they would require lots scene motion to sell them. At roughly the same time we decided that we would be targeting 30fps to support more flexibility in our lighting. Deciding to go for motion blur at this point aligned with so many of our project goals that it was an easy call.

Here is just a quick sample of what we were able to achieve with the algorithm I'm about to present

Project Constraints

- No asset requirements
- Consistent for all geometry types
- Independent of scene complexity
- Minimal G-buffer encoding

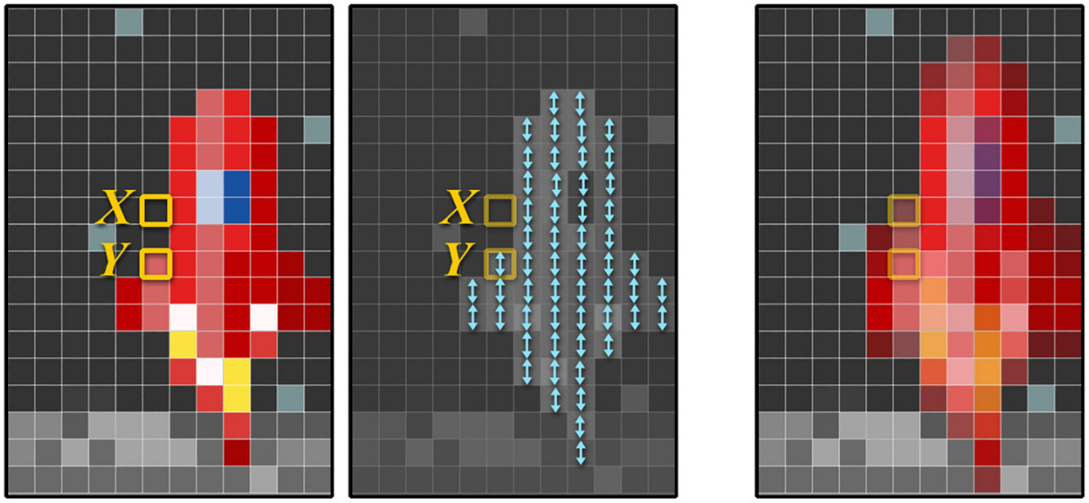
These constraints eliminated most existing techniques

When we approached motion blur we decided early that we would impose some restrictions on our solution.

- No asset requirements – We wanted to avoid as much as possible imposing new restrictions on the content teams. At this point in the project we had already redesigned our core lighting algorithm, material system, and animation system, we felt that any more requirements would have just slowed down the entire team.
- Consistent for all geometry types – This speaks to the robustness of the solution, we didn't want to handle many different classes of geometry or potentially have visual inconsistencies between two pieces of geo.
- Independent of scene complexity – When we started evaluating motion blur we didn't have a clear picture of where art was going to spend the main scene budget, we wanted to avoid algorithms that didn't scale well with differing scene complexities
- Minimal G-buffer encoding – With our g-buffer layouts we could really only spare 2 channels to encode the velocity and we couldn't spare any additional space for per-geometry data

Once we had these project constraints we evaluated numerous existing techniques but we found that we couldn't find one that met our list of restrictions

Closer Look At Motion Blur



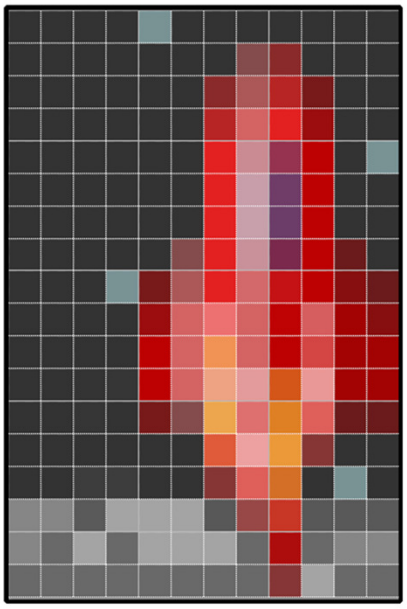
If look at what is going on under the good when something is blurring due to motion we can represent it fairly simply.

We have an input image and a velocity at every pixel for objects in that image. We see that the contribution of a pixel is spread out over distance represented by the velocity.

So if I can describe what is happening in 2 sentences why have a talk about how to solve it?

Core Problems

- Naturally scatter effect
- Need to represent as gather
- Objects blur outside their extents
- Results in transparent like effect



As I just described motion blur it is extremely simple to represent it as a scattering effect. The contribution of a pixel is distributed over the area described by the velocity.

Unfortunately we need to describe motion blur as a gathering effect. Independent of performance we could say that for each pixel loop over the entire image and find what pixels potentially overlap the pixel in question and we should end up with a very similar results. Again this is far to slow work in practice so we have to come up with something to make that portion of the algorithm faster.

Another problem associated with the gather technique is that objects moving no longer have the same silhouette, their shape becomes distorted as they move faster.

To put the icing on the cake, we end up seeing what was originally occluded by the object resulting in a transparent like effect.

Our Approach

- Dilate velocity using tiles
  - Allowing blurring outside of object boundaries
- Sample along dilated velocity
  - Scatter → Gather
- Mix samples based on velocity and depth
  - Background estimation

Our approach is three pronged:

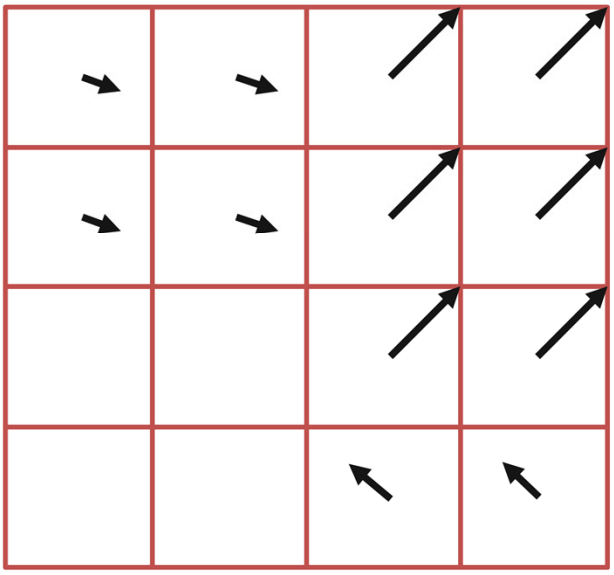
1. Dilate the velocity using screen space tiles : We chose to dilate the velocity using a tile based approach. We found that we could do a large radius dilation efficiently
2. Sample along dilated velocity : By preprocessing the velocity into the dilated form we have also given our selves the primary direction that we should look in when looking for contributing pixels
3. Mix samples based on velocity and depth : Finally we combine the samples in an interesting way to simulate background estimation for achieving the transparent aspect of motion blur

When we combine the 3 things mentioned above we get a robust motion blur solution

Walkthrough

Render Velocity

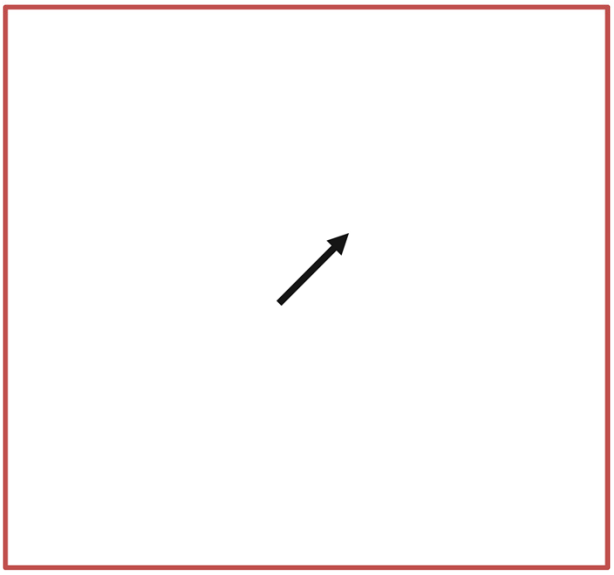
- Calculate velocity in screen space
- Clamp to max blur
- Rescale to 0→1



First you will want to render a pretty traditional velocity buffer. You will want to calculate velocity in terms of pixels in this step and then clamp it to the maximum allowable blur, you will see why this is important as the walk through progresses. Finally you will want to take the –max to max velocity and scale it into the 0-1 range so you can store it in 2 channels.

Tile Maximum Velocity

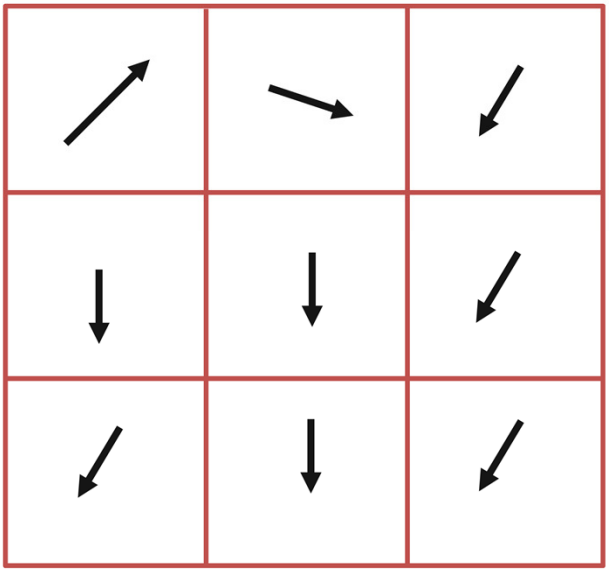
- NxN Down-sample
  - N = max blur radius
- Record maximum by magnitude



The first step of the velocity preprocessing is to find the maximum velocity in each tile on the screen. A tile is represented by NxN pixels where N is the maximum allowable blur. As you perform this down sample make sure you account for the remapping of the velocity when testing magnitude.

Neighborhood Maximum Velocity

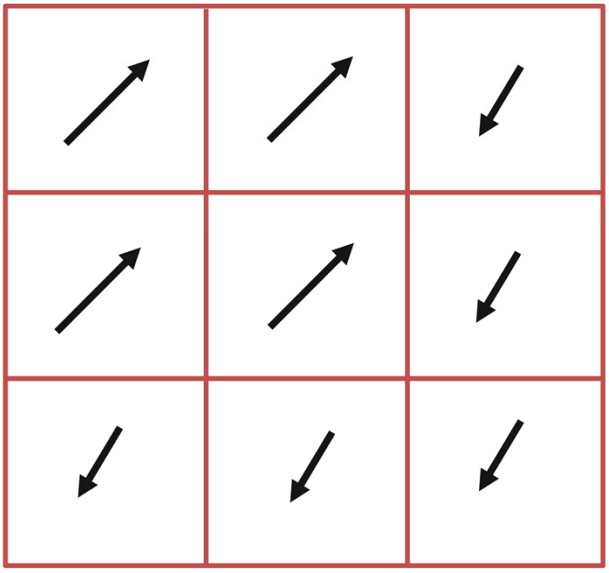
- Input tile maximum results
- Apply 3x3 box filter
- Find the maximum by magnitude of center and surrounding tiles



The second step of the velocity preprocess is to apply a 3x3 box filter on the tile max velocity texture to find the maximum velocity in a particular region. Once this stage is done we now have a conservative map of the regions of influence for some large scale velocities. Finally we can pass this texture on to the reconstruction shader as you will see next.

Neighborhood Maximum Velocity

- Input tile maximum results
- Apply 3x3 box filter
- Find the maximum by magnitude of center and surrounding tiles



The second step of the velocity preprocess is to apply a 3x3 box filter on the tile max velocity texture to find the maximum velocity in a particular region. Once this stage is done we now have a conservative map of the regions of influence for some large scale velocities. Finally we can pass this texture on to the reconstruction shader as you will see next.

Reconstruction

- Center Depth & Color

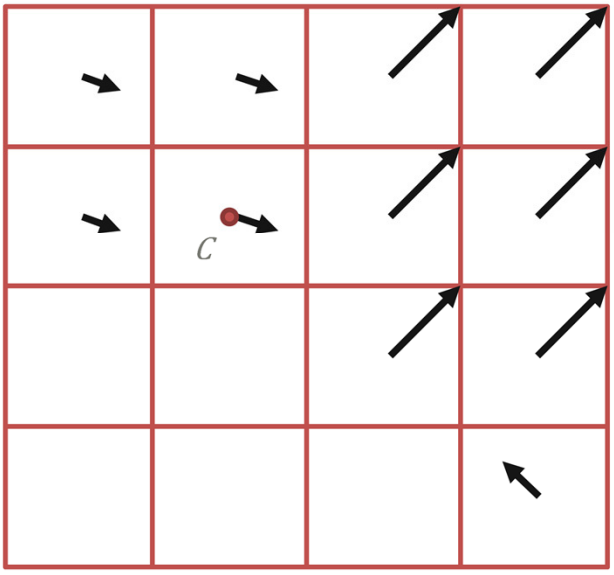
z_C , $Color_C$

- Center Velocity

$\vec{v}_C$

- Neighborhood maximum

$\vec{v}_N$



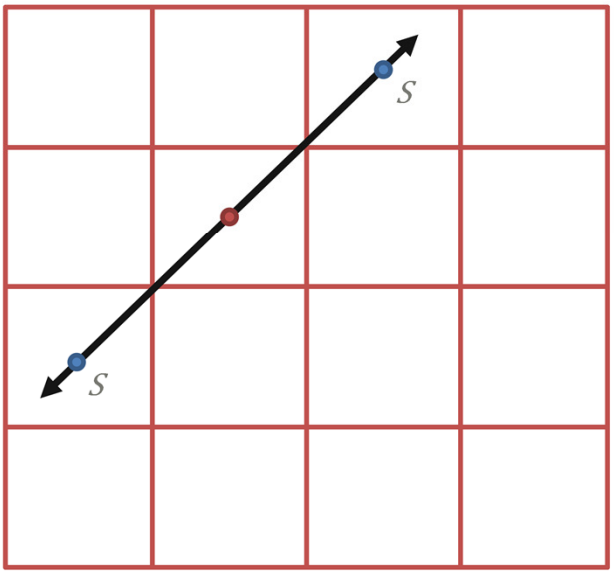
Now that we have pre processed our velocity into a W/NxH/N sized buffer it's time to use the information.

Looking at center sample C we first grab the depth and color of that pixel. Next we grab the high resolution velocity from our source velocity buffer, we hold on to this because we will use it in the comparisons in the next step. Finally we grab the maximum neighborhood velocity this will come in handy as we choose sampling locations in the next step.

Samples

- Calculate samples along $\vec{v}_N$ in both directions
- Sample

  z_S , $Color_S$, $\vec{v}_S$
- Determine foreground or background
- Compare with center



For this step you will want to re-expand the neighborhood velocity back into the –max – max range and then generate some sample points along that vector (both positive and negative from center).

At each sample point you will want to get the depth, color, and high resolution velocity. We then compare the depths to see if the sample is in the foreground or background of the center and then apply some comparisons based on that finding.

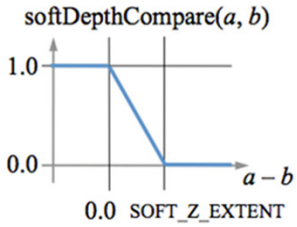
Foreground vs. Background

Desired:

$$z_S \gg z_C : 1$$

$$z_S \ll z_C : 0$$

$$0 < (z_C - z_S) < soft\ extent : 1 \rightarrow 0$$



Formula:

$$B_S = 1 - \frac{(z_C - z_S)}{soft\ extent}$$

To determine the foreground or background of a pixel we just applied a ramped step curve that gave us some intermediate values when the depths were close. If you do anything with soft particles this will look familiar.

Comparison Details

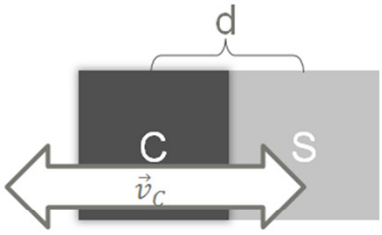
Interesting cases:

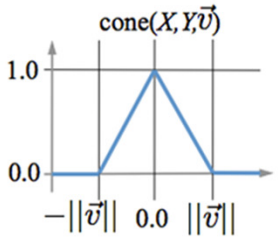
1. Background sample $(z_C < z_S)$
2. Potential background estimation
3. Foreground sample $(z_C > z_S)$
4. Potentially moving overtop of the center
5. Both samples are moving $(\|\vec{v}_S\| \neq 0 \;\&\&\; \|\vec{v}_C\| \neq 0)$
   - aka we want something blurry

Right now there are 3 cases that we really care about for the comparisons

1. Background sample
   The background sample case allows us to achieve the transparent effect by potentially treating the sample as an estimate for what is in the background of this pixel.
2. Foreground sample
   The foreground samples case allows us to blur an object past the extents of its own bounds because we allow samples from elsewhere blend overtop of the center.
3. Both samples moving
   This is the general bucket that we just want something blurry because everything is moving. We found that when you get to this point there is so much blurring going on that the eye can't really depict direction.

Background Sample





Case: $z_C < z_S$

Desired:

$|d| \approx 0$: 1 (Good estimate)

$|d| \gg \|\vec{v}_C\|$: 0 (Bad estimate)

$\|\vec{v}_C\| \approx 0$: 0 (No estimate needed)

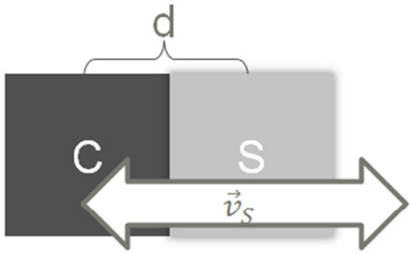
Formula:

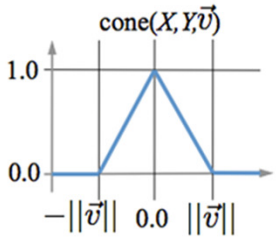
$$c_S = 1.0 - \frac{|d|}{\|\vec{v}_C\|}$$

Advances in Real-Time Rendering in 3D Graphics and Games

For the background estimation we only really need to look at the velocity of the center, determining if it needs a background estimate at all, and the pixel distance between the center and the sample to see if it would be feasible that they overlap. We represented this with a cone like fall off. We use the velocity of the center as the denominator because we get less picky about background samples the faster the object is moving.

Foreground Sample





Case: $z_C > z_S$

Desired:

$\|\vec{v}_S\| \gg |d| : 1$ (Blur overtop)

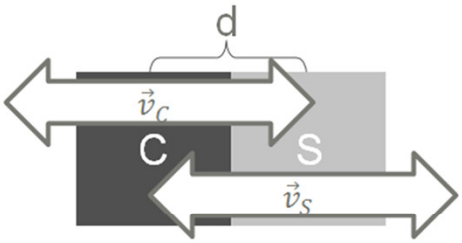
$|d| \gg \|\vec{v}_S\| : 0$ (Too far)

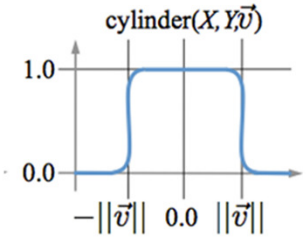
Formula:

$$c_S = 1.0 - \frac{|d|}{\|\vec{v}_S\|}$$

For the second case we want to allow object to blur out of their extents so all we really need to do here is ensure that the samples are overlapping. We apply a simple fall off so we get a nice fade at the ends.

Mutually Blurry Samples





Case: $\|\vec{v}_S\| \geq d \;\&\&\; \|\vec{v}_C\| \geq d$

Desired:

$\|\vec{v}_S\| \geq d \;\&\&\; \|\vec{v}_C\| \geq d : 1$

(Overlapping)

$\|\vec{v}_S\| < d \;||\; \|\vec{v}_C\| < d \quad : 0$

(Non-Overlapping)

Formula:

$$c_B = (\|\vec{v}_S\| \geq d \;\&\&\; \|\vec{v}_C\| \geq d)\;?\;1 : 0$$

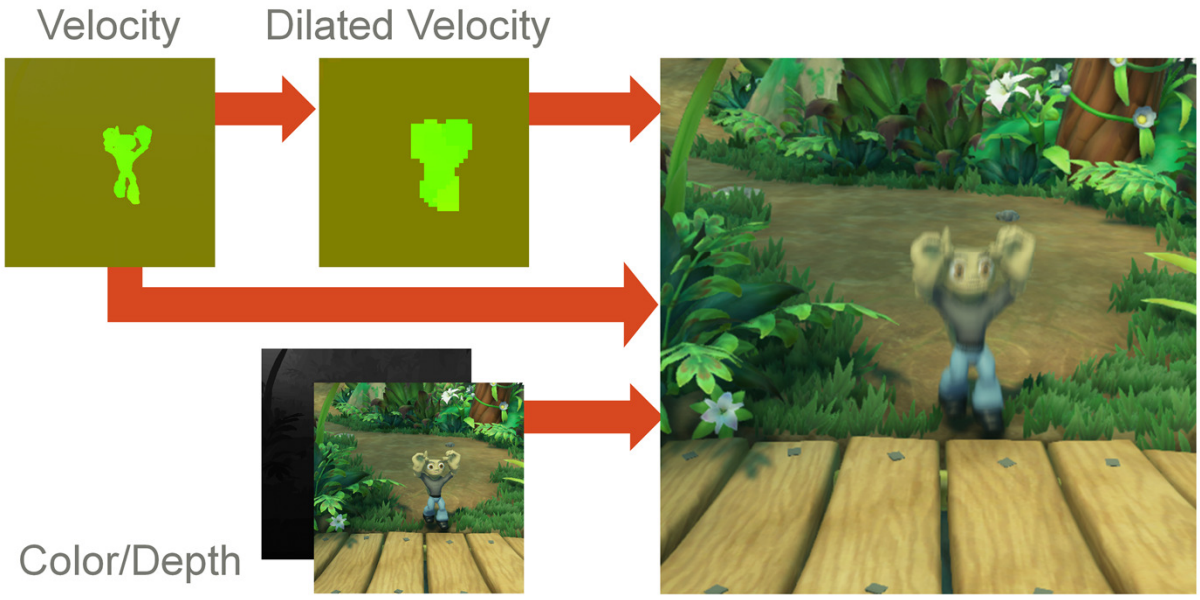
The last stage we simple need to validate the samples could feasibly overlap, notice how we don't check direction just magnitude, we found that the results were better doing this and the incorrect blurring was not noticeable at all.

Final Details

- Gather samples
- Sum weights from comparisons
- Sum color contributions
- Normalize for result

Finally we can generate the output pixel color. When performing the comparisons we don't actually treat them as mutually exclusive cases instead we determine the foreground and background weights and use that to weight the first two contributions and for the third we just mix it in. Because we do this we need to renormalize at the end. For the center sample you can fool around with the initial weights, a higher weight will result in less pronounced blur regions and a lower weight will result in a more "thick" looking motion blur.

General Flow



So putting it all together the high level flow we capture velocity depth and color. Then we preprocess the velocity and then finally run the reconstruction using all the source textures and the new processed velocity to produce the final image.

Scalability and Implementation

Scale for Design

- Tune relative weights of contributions
- Artificial velocity
  - Particles
  - Full screen masks
  - Game events
  - Special moves

1. Tune weights - The look of the motionblur is heavily determined by the weights of each of the comparison cases and the initial weight of the center sample. This can lead to a very subtle effect or a way over the top blur.
2. Artificial velocity – because this algorithm just operates on a normal velocity buffer you can inject any kind of velocity you want into the buffer, this artificial velocity could come from particles, full screen effects, special game events, or even character moves.

Scale for Quality

- Change max radius
- Increase number of samples
- Change comparison functions
- Mix in samples along tile max and center velocity
- More selective dilate

Quality scaling for this algorithm includes some usual suspects like adjusting the maximum radius, increasing the number of samples etc.

But there is also room for other quality improvements such as sampling not only in the direction of the neighborhood velocity but also in the direction of the tile velocity and center velocity.

I feel that you could also swap out the tile based dilate with any other dilation method to achieve different quality results.

Current Gen Implementation

- Branch on velocity
- 7 taps (6 samples + center)
- Checkerboard jitter
- Packed depth (velocity = xy, depth = zw)
- Max Blur Radius = 10px
- Alternate samples between Vc and Vn

We found it worth it to pay the branch cost on current gen because we had a lot of scene with not much motion. It would be interesting to change shaders when the engine detects that the camera is moving since that would likely result in more motion make the branch irrelevant

We chose a checker board pattern so that the post process AA would smooth out the motion blur contribution.

Packing the depth into the zw of the velocity texture will save you a ton on the reconstruction phase so you should do it… really… just do it.

We wanted the effect to be relatively subtle so we chose a max blur radius of around 10px at current gen resolutions

Even on current gen we decided to alternate samples between the center and neighbor hood velocity to avoid artifacts at the boundaries of tiles.

DX11 Implementation

- Branch on velocity
- 9 taps (6 samples + center)
- 4x4 randomized jitter
- Packed depth (velocity = xy, depth = zw)
- Max Blur Radius = 18px
- Alternate samples between Vc and Vn

We kept the branch on velocity since it was much less overhead in DX11

We only increase the sample count slightly because we didn't see much of a difference when really increasing that number

At a higher resolution we found we could get better results with a 4x4 randomized jitter so we removed the checkerboard

We only increased the radius to maintain visual consistency at the higher resolution.

Once again we alternated samples between the center and neighborhood velocities to avoid tile boundary artifacts

Results

This video shows a typical game camera and you can see how the motion blur enhances the characters movements. We have turned up the exposure time so it is more excessive that we normally have to better show the blurring. The game is rendering at 30fps

This video is more of a comparison of with and without motion blur.

Notice the strobing on the right hand side as well as the perceived lower frame rate.

Agenda

- ~~Designing Scalable Algorithms~~
- Examples
  - ~~Motion Blur~~
  - Ambient Occlusion

Scalable Ambient Occlusion

Introduction

- Greatly benefits lighting in shadow
- Contact and crease shadowing

AO is a pretty standard effect in rendering at this point. There has been some great work involved in approximating the ambient occlusion GI term. We wanted a solution that could move between platforms for visual consistency so we decided to throw our hat in the ring.

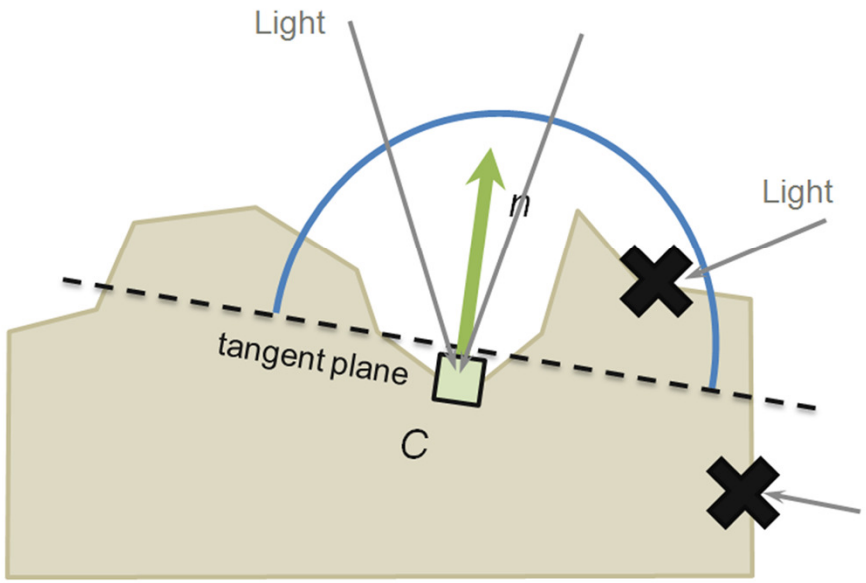
Here is the kind of contribution for AO that we can achieve with the algorithm I'm about to present

Project Constraints

- Fast
- Surface aware
- Reduce aliasing

Our project constraints for this effect were pretty generic, it needed to be about as fast as our current VO implementation, we wanted it to be more surface aware, and we wanted to reduce aliasing as much as possible.

A closer look at AO



So I'm sure you have probably seen a slide like this in the past but I'm going to quickly re-iterate the goal with AO.

We basically want to estimate the amount of ambient incoming light that is blocked by the complexity of the geometry.

Core problem

- Didn't want two different algorithms
- Wanted visual consistency

The core problems in this were much more localized to scalability than motion blur. We just wanted an algorithm that had visual consistency and we didn't want to support and maintain two separate algorithms

.

Our Approach

- Less but higher quality samples
- Base contribution on two things:
  - Distance of point to center
  - Projected length of the sample distance onto the normal
- Experiment with falloff function until we liked the results

We decided that to keep up in terms of speed with our existing algorithm we would take less samples but make them higher quality

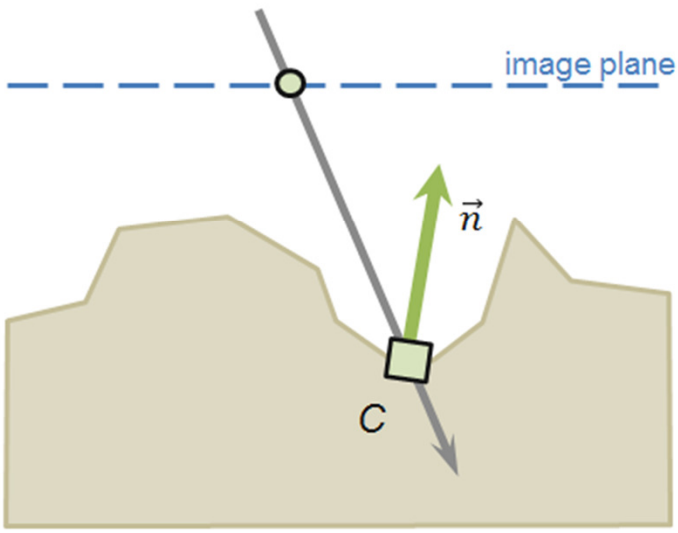
We would base the contributions on the normal of the surface being shaded to get better surface aware behavior

Finally we wanted to get it to a point where we could start manipulating the fall off function to get the exact look we wanted.

Walkthrough

Center

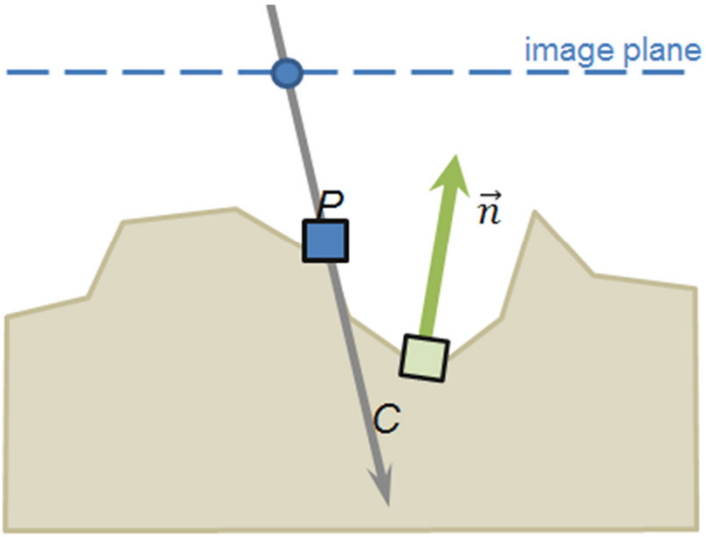
- Sample Depth and Normal
- Reconstruct Position



For the center sample we simply reconstruct the position and normal

Sample Properties

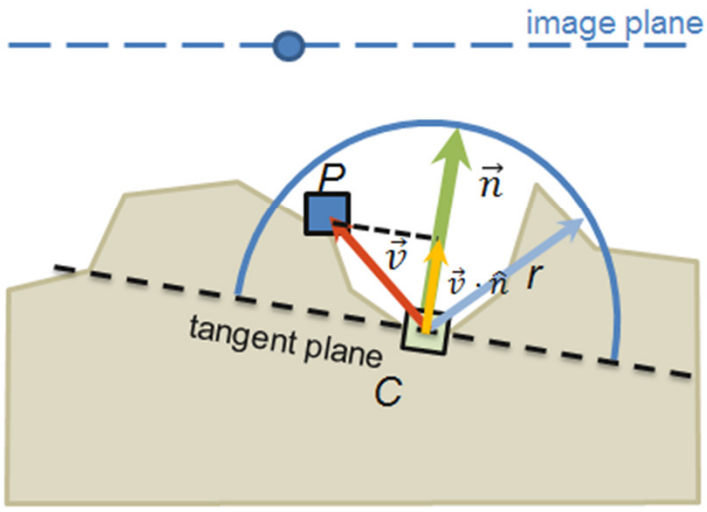
- Choose Sample Location
- Sample Depth
- Reconstruct Position



When gather sample properties we chose to just reconstruct the position of the sample.

Sample Contribution

- Calculate $\vec{v}$
- Calculate$\|\vec{v}\|$
- Calculate $\vec{v} \cdot \hat{n}$
- Apply falloff based on radius *r*



We then calculate the vector between the center and sample points.

We gather information that we can apply in fall off functions by projecting the vector onto the surface normal and getting the length of the vector.

Falloff Function

- We use

$$AO_s(\vec{v},\hat{n}) = \max\left(1 - \frac{\|\vec{v}\|}{\mathrm{r}}, 0\right) * \max\left(\left(\frac{\vec{v}}{\|\vec{v}\|}\right) \cdot \hat{n} - bias, 0\right)$$

Distance Falloff Angular Falloff

$bias$ = small bias to avoid self shadowing

- Try some of your own here!

*See Morgan et al. 12 for more falloff options

The last part for the sample is figuring out the contributions of the various parameters. We experimented a lot with this phase and determined that we liked the following function the most.

The distance portion of it is simply a linear fall off

The angular portion is really just a dot product but before normalizing the vector we apply a small bias to avoid self shadowing

You can check out the follow up paper for more falloff options.

The important thing here is that you should try some of your own function here to get the exact look you are going for.

Total Occlusion

- Sum Contributions and Normalize

$$AO_T = \left(\frac{2 * scale}{S} * \sum_{i=0}^{S} AO_{s_i}(\vec{v}, \hat{n}) \right)^2$$

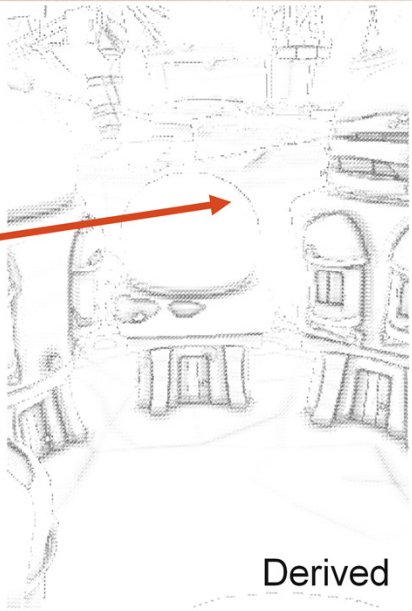
$scale$ = Artist tuned contribution scale

S = Number of samples

The last step is to accumulate all the samples and introduce any artistic controls you want before normalizing the contribution.

Normals

- G-buffer if you got it
- Derive from depth otherwise
  - Some artifacts
- Two different looks for the AO contribution



For the normals you have a couple options, the first is that you use g-buffer normals. Obviously this only works if you have a g-buffer, if you have a forward rendered engine then you might consider using derived normals from the depth buffer. You get some artifacts but they are lessened when you put the contributions through a blur.

AO Contribution (G-Buffer Normals)

AO Contribution (Derived Normals)

Blur Results

- Depth aware bilateral blur
- Softens contribution
- Wide filter to hide pattern
- Combined with shadows to amortize cost

To make the ao contribution silky smooth push it through a depth aware bilateral blur. You should make the filter fairly wide to avoid patterns. Even if you pack the depth in the same buffer as the AO you still have an extra channel so you should use it to soften some of your shadows.

Final AO

Scalability and Implementation

Scale for Design

- Increased radius
- Change falloff functions
  - Emphasize contact shadows
  - Wider influence
- Change application
  - Modulate everything
  - Modulate ambient

The biggest one here is the fall off functions

Depending on your falloff functions you can favor contact shadows or larger influence and everything inbetween.

Scale for Quality

- Increased radius
- Increased sample count
- Better sampling pattern
- Wider blur

Because we can apply the same fall off functions just on less samples to the different platforms the quality scalability is really good for this approach. Depending on how wide your blur is it might be more worth it to blur the results better.

Current Gen Implementation

- 4 taps at full resolution
- Sample around a spiral
- Rotate around random vector
- Use mirrored taps
- Clamp Radius to ~20 pixels
- Transpose math to calculate all at once
- Encode depth in blue/alpha for blur

We get pretty great results using only 4 taps at full resolution, to make it fast we use mirrored taps and transpose all the math to compute the 4 samples all at once. Our final shader is about 56 instructions, very comparable with VO.

DX11 Implementation

- 9 Samples
- Procedural spiral sampling pattern
  - Introduces some noise
- No radius clamp
- Mip depth for unclamped radius
- Encode depth in blue/alpha for blur
- See Morgan et al. 2012 for more performance tips

For dx11 we up the samples and sample patterns. We also employ the depth mip trick outline in Morgan et al. to effectively remove the screen space velocity constraint.

Clamped radius

Agenda

- ~~Designing Scalable Algorithms~~
- ~~Examples~~
  - ~~Motion Blur~~
  - ~~Ambient Occlusion~~
- Closing

Closing

Closing Remarks

- Demonstrated power of scalability
- Effects that bridge the gap between current gen and DX11
- We have experienced excellent results with this approach

Thanks

Activision

HPG and I3D

Vicarious Visions

Michael Mara and David Luebke – Nvidia

Natalya “Natasha” Tatarchuk

References

Original Papers and Source:

http://www.vvisions.com/research/whitepapers.cfm

http://graphics.cs.williams.edu/papers/

Extension to original AO

http://graphics.cs.williams.edu/papers/SAOHPG12/

Join our team!