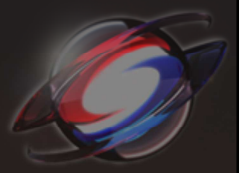

SIGGRAPH2010
The People Behind the Pixels

Siggraph 2010 – "Split Second Screen Space"



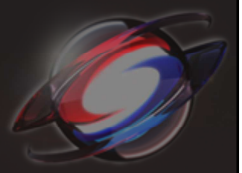
A Fast Deferred Shading Algorithm for Approximate Indirect Illumination

Olivier Hoël
Cyril Soler

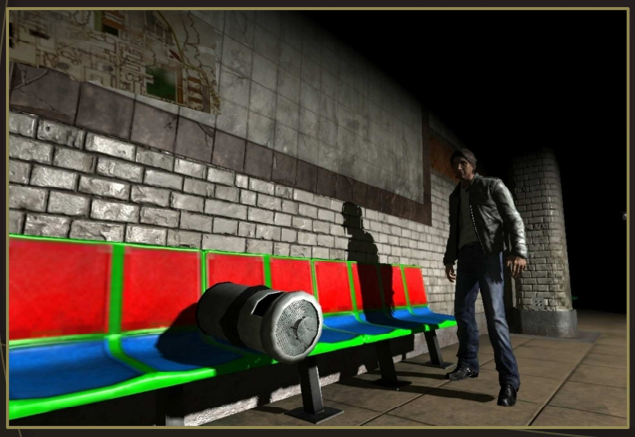
Frank Rochet

This work has been done in collaboration with the game studio Eden Games.

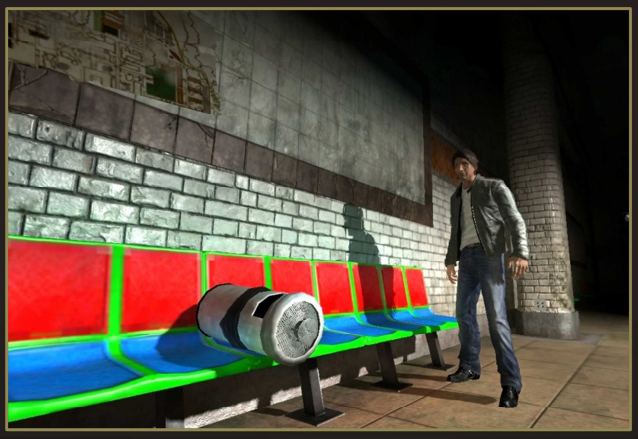
Motivations



- Greatly enhance realism and immersion
- But expensive to compute



Direct lighting only

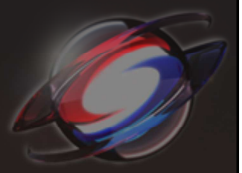


With indirect lighting

2

For this work, we choose to focus on indirect lighting.

Video games constraints

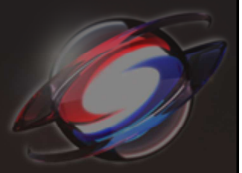


- 30fps => at most ~10ms for indirect lighting
- Dynamic environment
- Independent of geometry / lighting complexity
- Any kind of illumination source
- Artifact free result

3

Important point is that the cost must be independent of scene complexity, and we'd like to be compatible with any kind of light source like environment maps...

Video games constraints

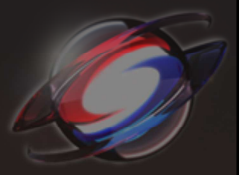


- ✓ 30fps => at most ~10ms for indirect lighting
- ✓ Dynamic environment
- ✓ Independent of geometry / lighting complexity
- ✓ Any kind of illumination source
- ✓ Artifact free result
- ✓ Deferred shading pipeline

4

Another important is that it should run using a deferred shading pipeline.

Deferred shading



- Two steps
  - Render Geometry Buffer (*G-Buffer*)
  - Compute lighting as post-process
- Pros / Cons
  - Hidden objects => approximate ☹
    - » Included solution to remove artifact ☺
  - Screen Space => FAST ☺

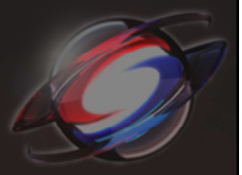
5

The big limitation of this approach is that the geometry buffer contains data only for visible objects. Back faces, culled objects or objects that are not in the view frustrum can't appear in the G-Buffer.

This is not a problem for direct lighting but it will be for indirect lighting.

As we inherit from this limitation, we do not pretend to compute exact indirect lighting, but only an approximate solution. All our computations are done in screen space, so at first our results will be inaccurate, but that's also the reason why it can be computed in real time.

Outline



I. Related Work
II. Screen Space sampling
III. Pipeline
IV. Results

Related Work

*Imperfect Shadow Map [RGK*08]*

Multi-resolution splatting of Indirect Illumination [NW09]

Cascaded Light Propagation Volumes [KD10]

Hierarchical Image-space Radiosity [NSW09]

7

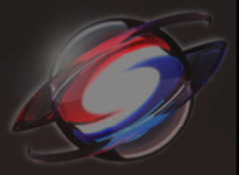
I will just cite the most recent techniques which can achieve real time performances in a dynamic environment.

The main difference with our approach is that they all use another view of the scene which is reflective shadow maps. They can get accurate result but the cost is then dependent of lighting complexity.

Comparison

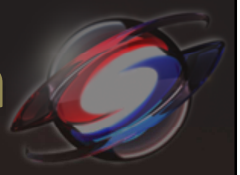
	[RGK*08]	[NW09]	[NSW09]	[KD10]	Ours
No pre-computation	✗	✓	✓	✓	✓
Constant cost w.r.t lighting complexity	✗	✗	✗	✗	✓
Any kind of light source	✓	✗	✗	✗	✓
Multiple bounces	✓	✗	✗	✓	✓
Visibility	✓	✗	✗	✓	✓

Outline



I. Related Work
II. Screen Space sampling
III. Pipeline
IV. Results

Screen Space Indirect Illumination

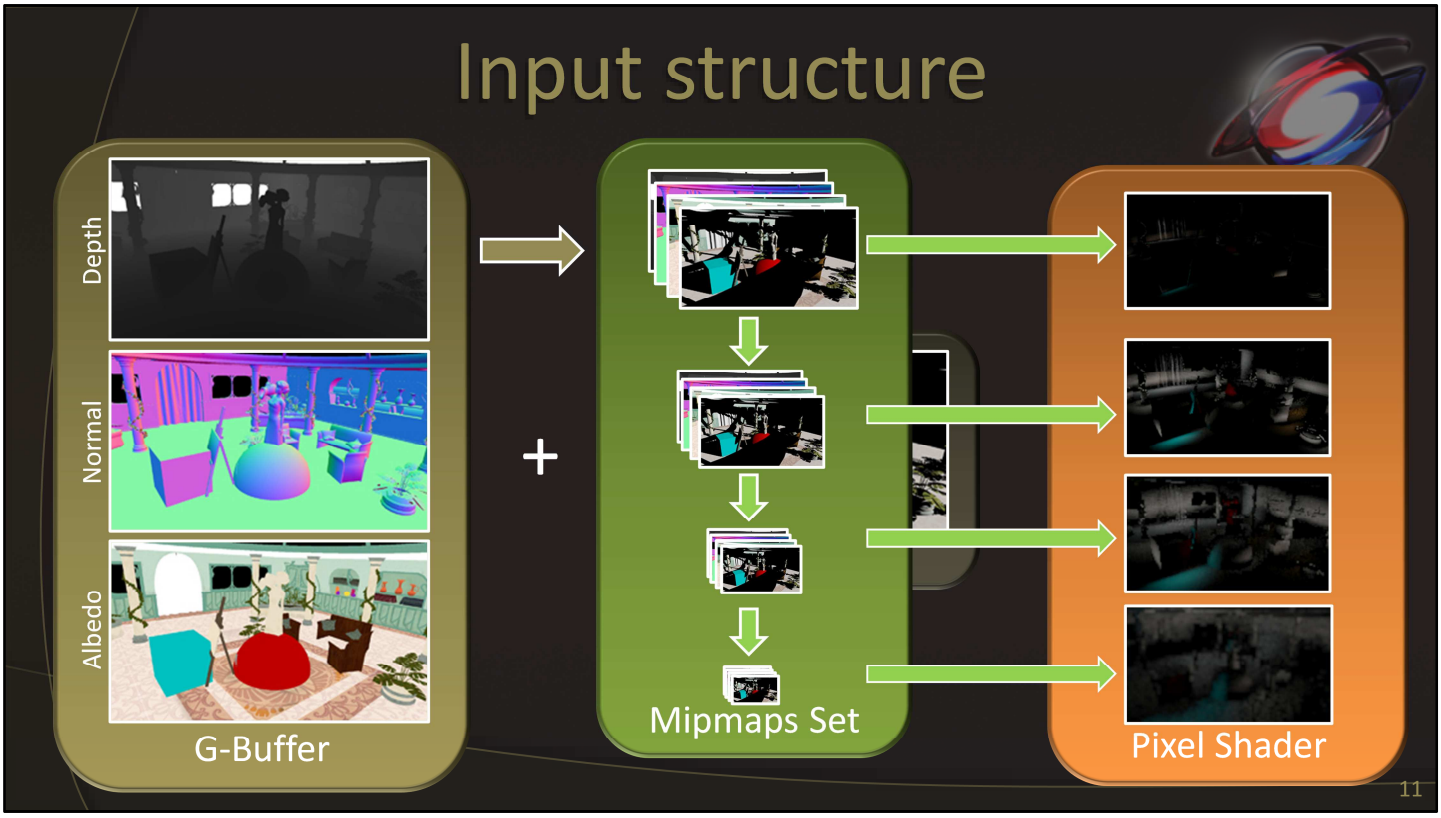


- Monte-Carlo integration on G-Buffer
- Hierarchical computation using mipmaps
  - Arbitrary distance
  - Save performances

10

The G-Buffer provides all relevant data we need. We can then perform a Monte-Carlo integration to compute indirect illumination per pixel.

The key idea is to make it hierarchically, by using mipmapped version of G-Buffer. This allows us to compute distant light interactions, without loosing performances.



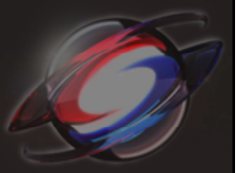
We start from a classical G-Buffer.
But to compute indirect lighting we also need the direct lighting.

We then add this into an extended G-Buffer.

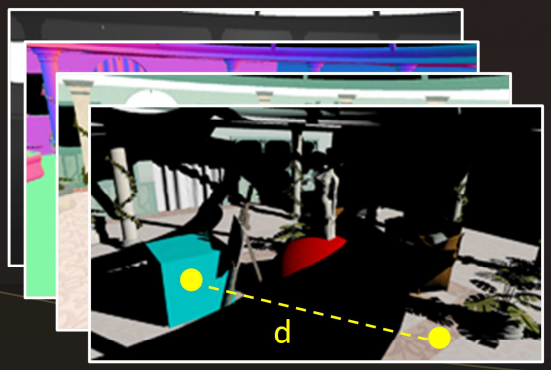
We compute a set of mipmaps until a specified level.

Then each level will be an input of a pixel shader, which compute indirect illumination at different scale.

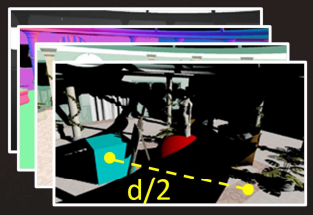
Sampling scheme (1/3)



- Random screen space samples on disk
- Efficient use of texture cache
- Constant radius over all set of mipmap



Mipmap set at level *l*



Mipmap set at level *l+1*

12

For sampling we use random samples distributed on a screen space disk.
Most of the time, objects far from each other in world space, are also distant in screen space. But not anymore when using mipmaps.

For instance if I place two samples, they lie at a certain screen space distance. But if I place them in the next mipmap, the distance is divided by two.

So it means that to compute lighting interaction between these objects, it's better to sample coarse mipmap; so that texture lookups are local, and then an optimal use of texture cache.

We keep the same sampling radius over all level to efficiently compute indirect lighting, for close and distant objects.

Sampling scheme (2/3)

13

Let’s say we’re computing indirect lighting for the center pixel. We start by sampling the highest mipmap.

We then samples the next level using the same radius. For illustrative purpose, instead of dividing the image by 2 I multiply the radius by 2. As you can see, the inner part of the disk has already been sampled, so we distribute them on the outer crown.

And we keep going the same way for the next levels.

So, using this scheme, we can sample nearly the whole buffer in a very efficient way.

Importance sampling (3/3)

14

It works well with diffuse material, but introduces some variance when computing glossy reflection, because most of samples won't fit into the reflectance lobe.

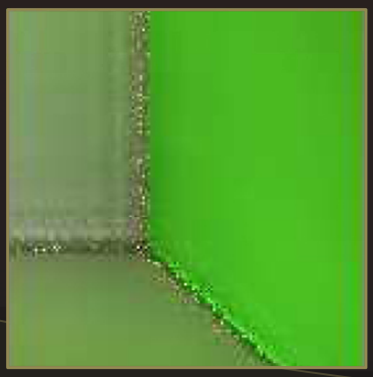
We solve this issue by introducing an importance sampling scheme. We compute the screen-space projection of the lobe, and then distribute samples into the bounding cone, to ensure that most of them would fit in the lobe.

Here are comparative results using a Phong BRDF for the ground. You can clearly notice the quality gain of using importance sampling.

Monte-Carlo reformulation

$$L'(x,\omega) = \sum_{i=1}^{N} \rho_i L_i \frac{\cos_+ \theta_i \cos_+ \theta'_i}{r_i^2} v_i \frac{z_i^2 4 \tan f_h \tan f_v dP_i}{WH} \frac{\cos\alpha}{\cos\beta}$$

$$\frac{\cos\theta' ds}{r^2} \approx \frac{R^2}{r^2 + R^2} \quad \text{with} \quad R^2 = \frac{1}{\pi}\cos\theta' ds$$



Using point to point form factor

Using point to disc form factor

15

Although we only account for visible light, we still need to do it accurately – so that different levels combine correctly.

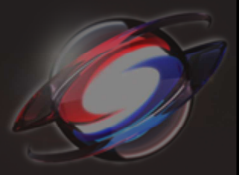
We use a Monte Carlo method to collect the incoming energy under each pixel, using a traditional point-to-point form factor.

But because we sample pixels instead of directions in 3D, we have to weight their contributions, with a jacobian.
For all details, you can refer to the technical report.

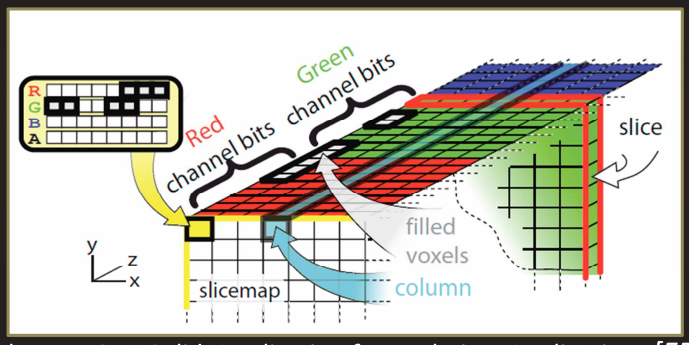
Now, the point to point form factor is not a good choice, because it becomes infinite as the distance to samples approach zero, and produces noise.

So we replace it with a different formulation based on the close formula for point to disc form factor. The result is the same, but variance is removed.

Visibility



- Fast scene voxelization from [ED08a]



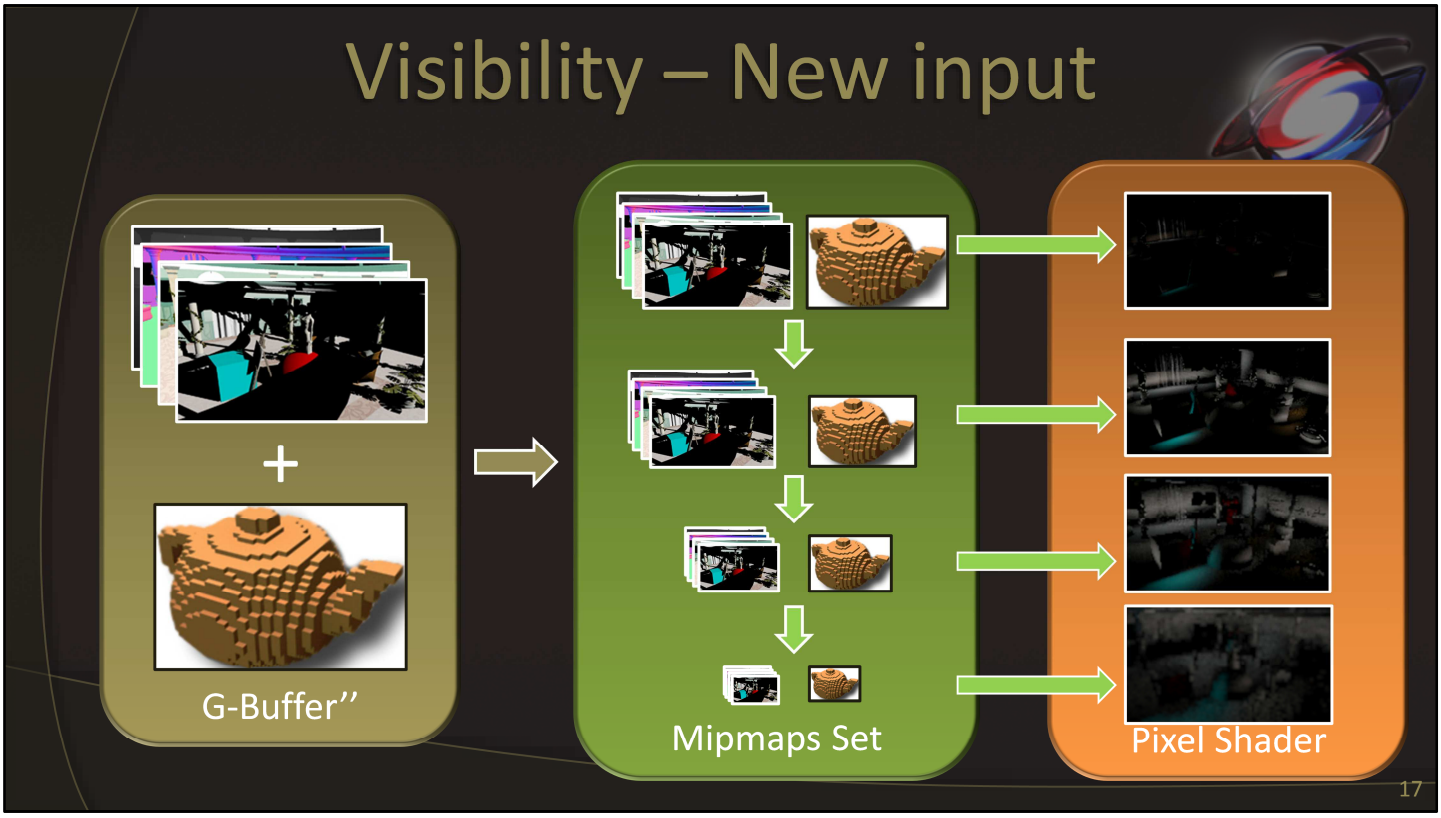
Single-Pass GPU Solid Voxelization for Real-Time Applications [ED08a]

- Trace rays to samples to resolve visibility

16

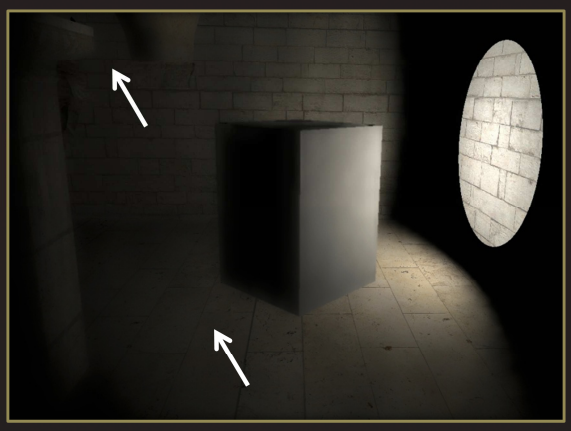
To take into account visibility, we perform a solid voxelization of the view frustrum. The idea is to render the scene into textures using the color channels as bit fields.

This produces a binary voxel grid that we could use to compute visibility.

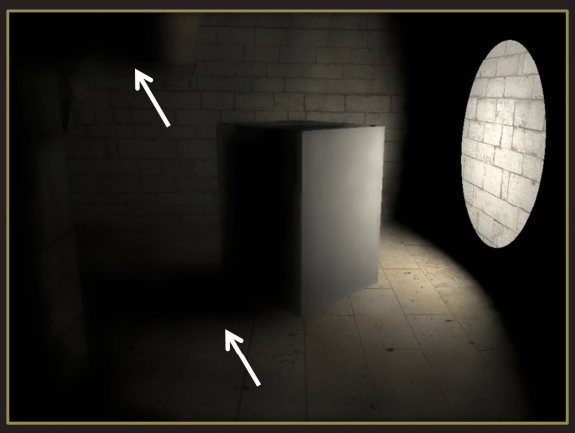


So we add new texture encoding our voxel grid, and also include it to the mipmap chain. This way visibility requires only local texture lookup as well.

Visibility - Results



Without visibility



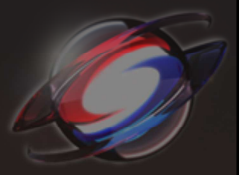
With visibility

18

Here is a simple example showing the quality gain of using visibility. Note that there is only one light source which illuminates the right wall, and everything else is indirect lighting.

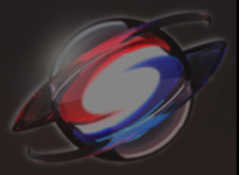
You can notice that we can get some indirect shadows at the left of the cube and also at the left of the arch.

Outline



I. Related Work
II. Screen Space sampling
III. Pipeline
IV. Results

Mipmapping (1/3)



- Interpolate normals / depth ?
- “Nearest”: inconsistent over time

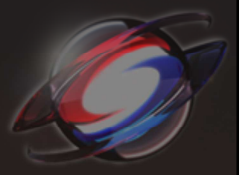
  Source of variance => need a specific scheme
- /!\ Voxel grid is binary

  => Bitwise operator

20

Obviously we can’t just interpolate geometric data, and nearest filtering leads to inconsistent result over frames

Mipmapping (2/3)

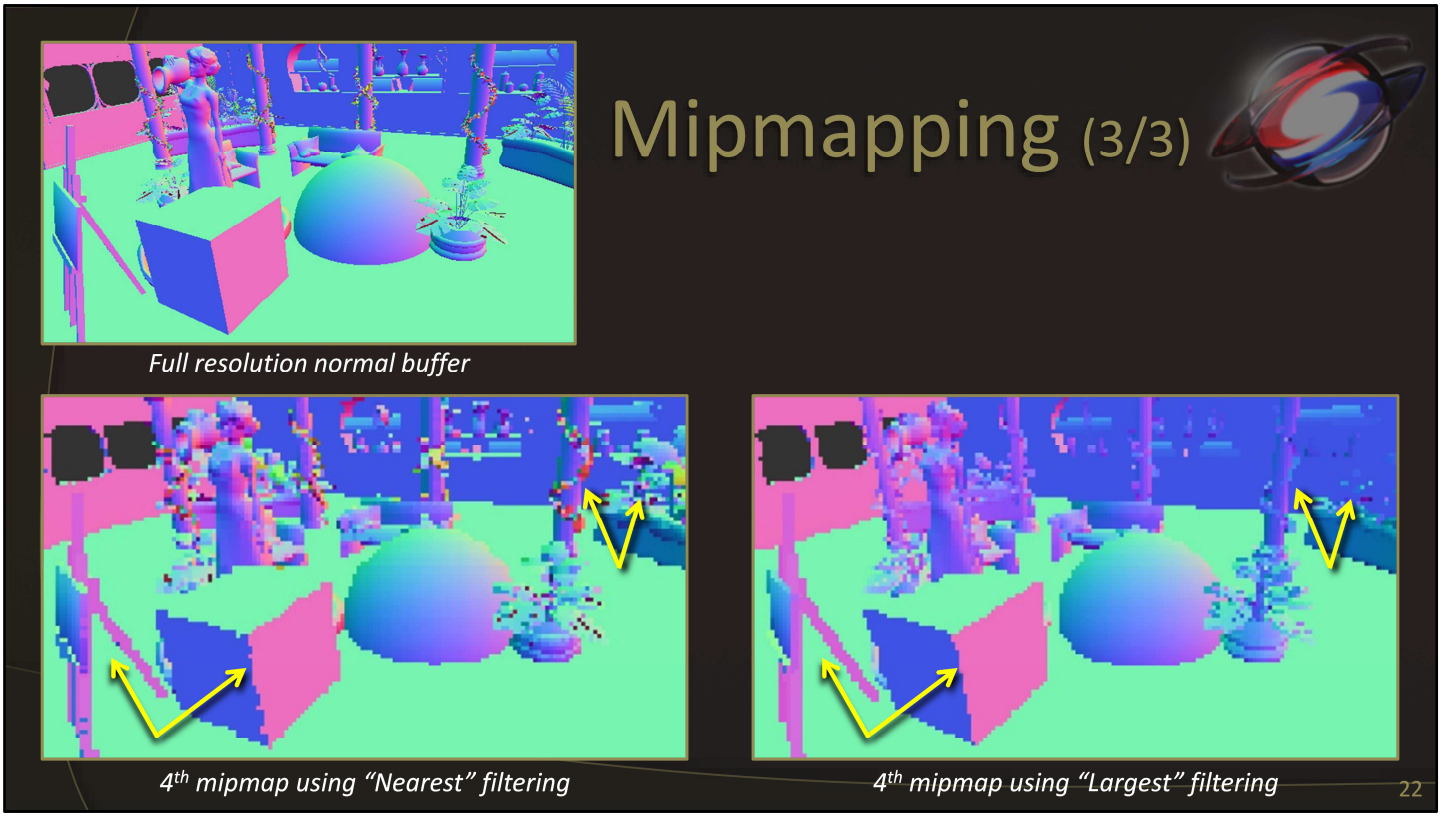


- Voting scheme to favor large objects
  - Comparing geometric similarity with neighbor pixels
  - ⇒ Discard small areas
  - ⇒ Consistent over frames

21

We then introduce a voting scheme which favor the largest objects.
For this, we compute a score for each sub-pixel, and we keep the one with the largest score.
This score represents the geometric similarity with neighbor pixels.
You can refer to the technical report for the exact formula.

It conservatively keeps large area across mipmap, and discard small inconsistent regions.
Moreover it also enables to filter normal maps attached to objects.

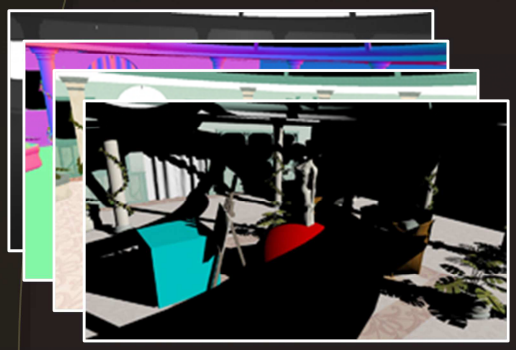


So you can see in the top-left corner the full resolution normal buffer of a scene.

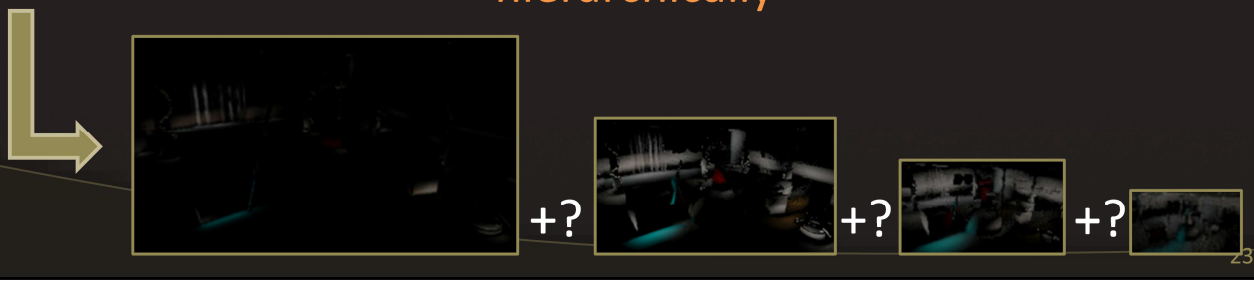
Here is a scaled view of the fourth mipmap.

As you can see our solution discards small details and faces across mipmap, like leaves on the pillar, or plants on the right. But it also enable to keep global shape of objects in a better way, like you see on the cube or the easel on the left.

Hierarchical upsampling (1/2)



- Different resolution to combine
  - No overlapping => ADD
- Coarse and noisy
  - Use cross bilateral filter hierarchically



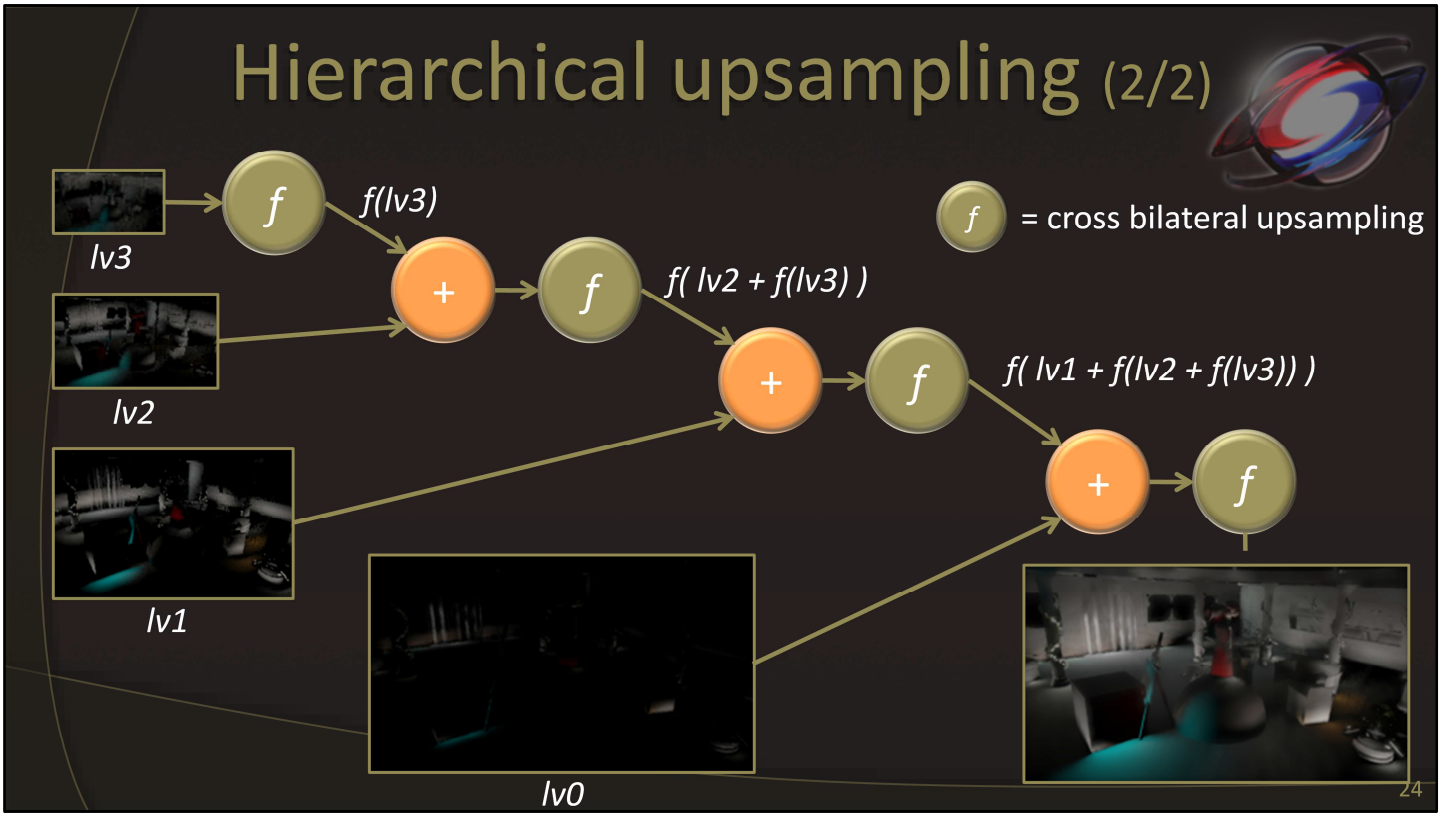
23

We compute indirect lighting at different resolution and now we have to blend them all.

Our sampling scheme ensures that samples do not overlap from one level to another, meaning that the blending operator would be a simple addition.

But just adding is not sufficient: coarse resolution will produce blocky results when upscaled, and some noise appears due to random sampling.

As our input contains geometric data, we can easily use a cross bilateral upsampling filter to smooth levels while preserving edges. But we will do it in a hierarchical way.



We start to apply cross bilateral upsampling to the coarsest resolution.

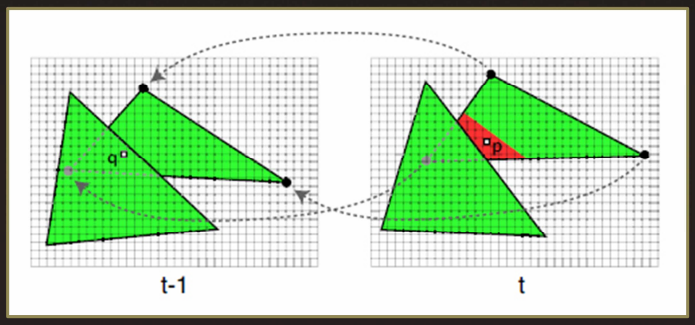
We can then add it to level 2, and reapply filtering to the output.

And next steps are just repeating this pattern until we reach the final resolution.

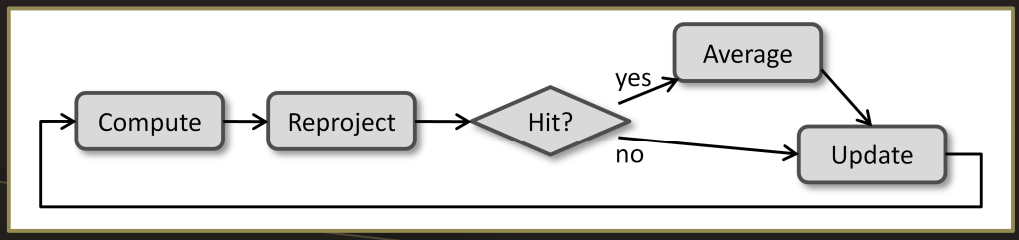
At the end of the process, the coarsest level has been filtered several times at no extra cost. Additionally edges are preserved at each uspscaling step.

So far this removes the noise and blocky patterns, and leads to smooth results

Temporal coherence



Reverse Reprojection Caching [Nehab et al 07]

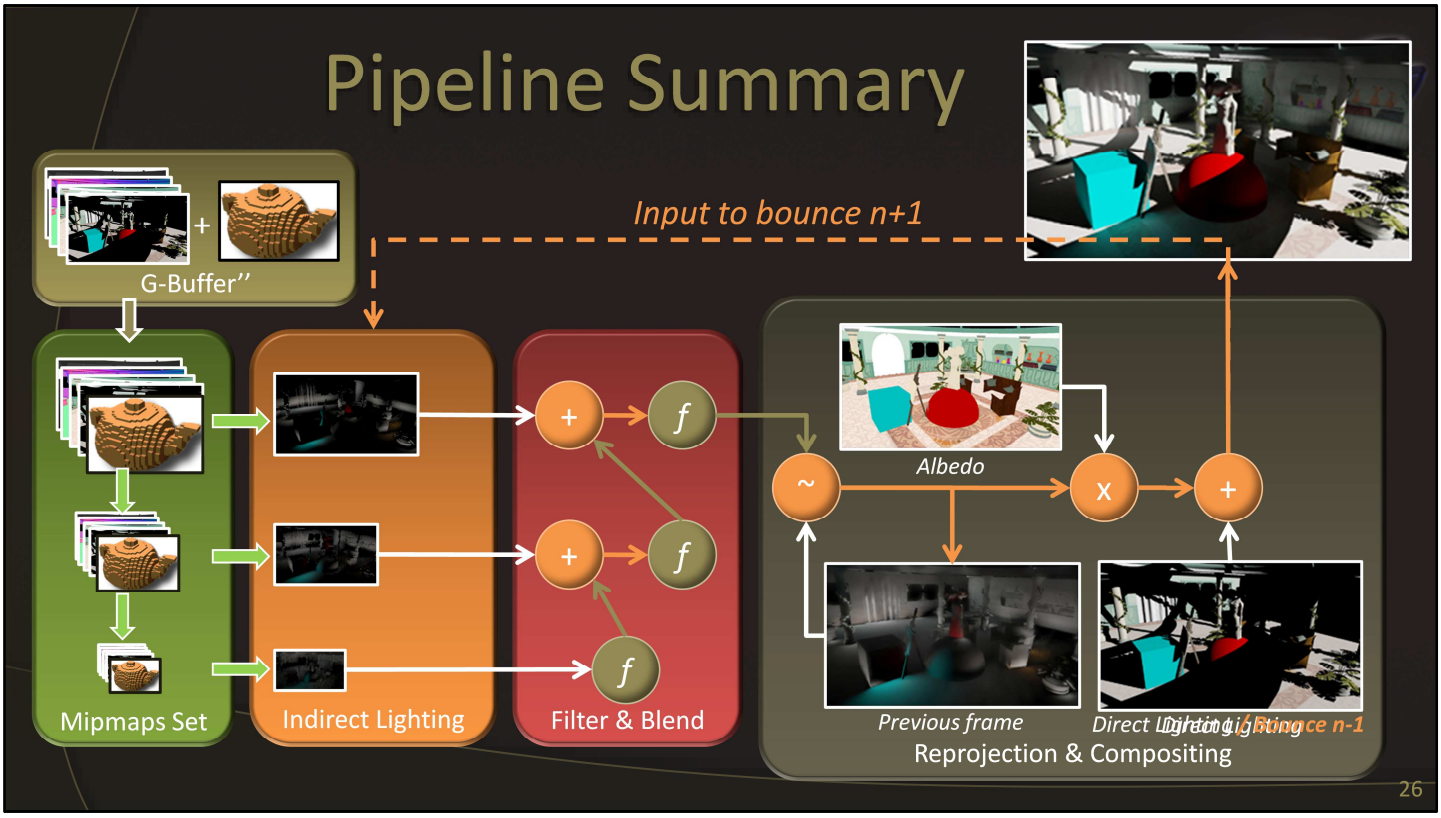


25

We also have to increase the temporal coherence. For this we use a reprojection caching method, like described by Nehab and colleagues.

The idea is to project each pixel back into the previous frame, and see if we can re-use its value.
If so, we can blend new value with the previous one, and update the history.

So we get a very good temporal coherence, for a very low and constant cost. It removes almost all flickering, and greatly smooth results over time.



As input, we have our extended G-Buffer with direct lighting and voxelization.
From this we compute mipmaps
And for each level we compute indirect lighting at different resolution

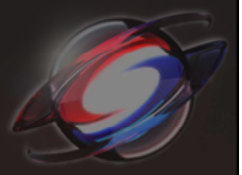
Next we combine all together.

In a following step, we compute the reprojection for temporal coherence. The result is used for final compositing: we multiply it with albedo, and compose it with direct lighting.

Thus we get our final image.

What is also really interesting, is that adding more than one bounce is straightforward. All we have to do is to use the output of our pipeline, as an input of the computation shader, and restart the same process.

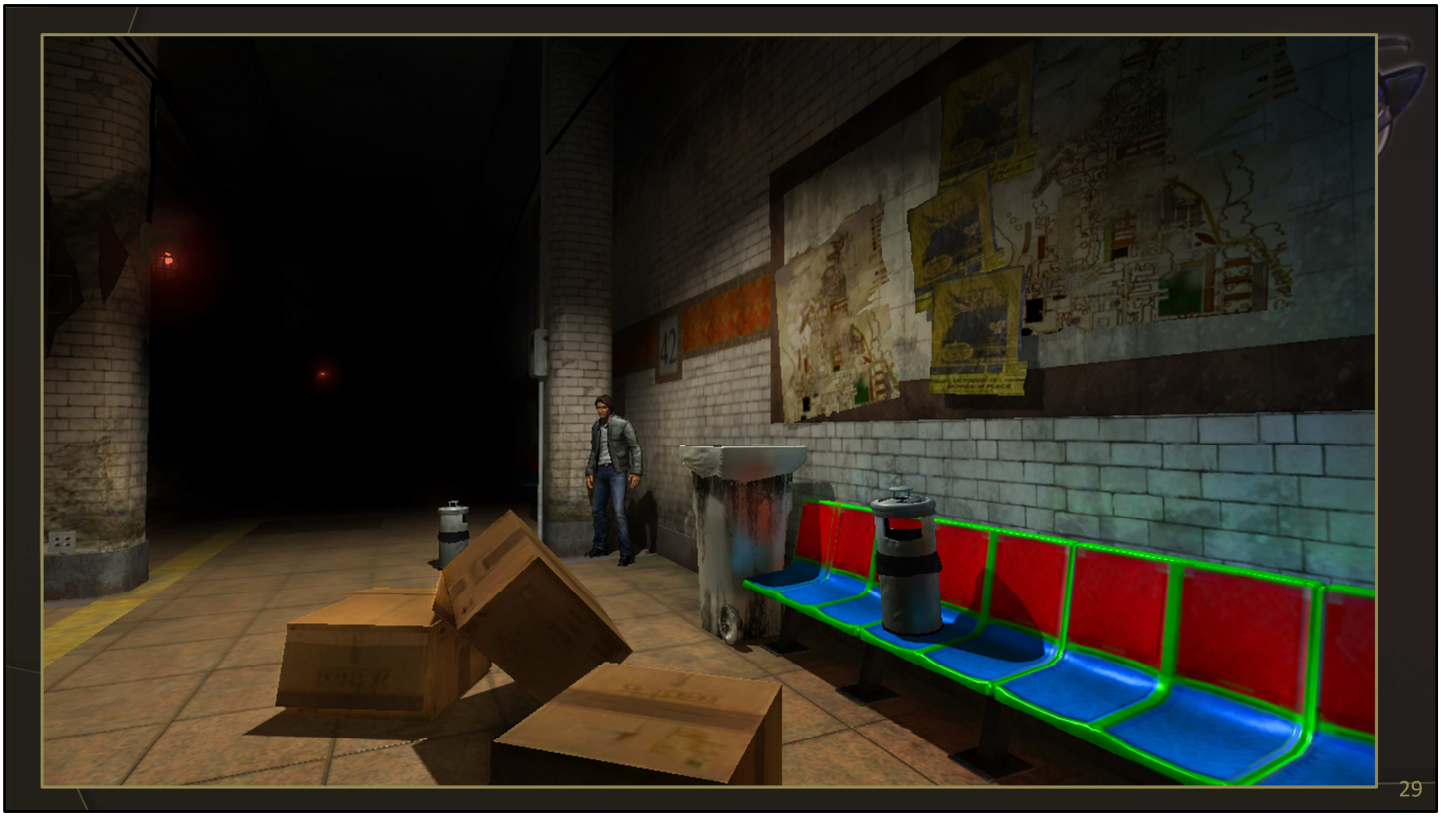
Outline



I. Related Work
II. Screen Space sampling
III. Pipeline

IV. Results

Here is a view of Crytek Sponza with only direct lighting and then we add indirect lighting.
So you can notice that we can get local color bleeding, like red from curtains on pillars.
But also some long range indirect lighting that will reveal the lion in the back



We integrate our method into the last Alone In The Dark engine to prove it can be used in a real game.
Note that in this scene, the trash bins are using glossy material.

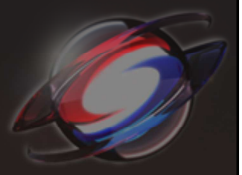
Real Time Video

GeForce GTX 260 @ 1280x720 – Using 256 samples

30

And now a short video shot in real time within the game.

Performances



- Performance / Quality trade off ☺
  - Quality: starting level & Number of samples
  - Range: sampling radius vs. number of levels

31

We have several ways to tweak the performance-quality trade off, at the depend of local indirect lighting accuracy.

For instance we can choose at which level we'd like to start computations. In our examples, we usually start from half or quarter resolution.

Real time !

	Starting res. : 1/4 Levels: 4 Samples: 64		Starting res. : 1/4 Levels: 4 Samples: 256		Starting res. : 1/2 Levels: 4 Samples: 64		Starting res. : 1/2 Levels: 5 Samples: 64	
Visibility	Without	With	Without	With	Without	With	Without	With
Voxelization (ms)		4.5		4.5		4.5		4.5
Indirect lighting (ms)	2.3	50.2	8.9	188	7.8	125	7.85	129
Upsampling & Compositing (ms)	2.3	2.3	2.3	2.3	2.7	2.7	2.7	2.7
Total (ms)	**4.6**	57	**11.2**	195	**10.5**	132	**10.55**	136

GeForce GTX 260 @ 1280x720

32

This table shows some computation times, in milliseconds, for different quality configuration. We choose these parameters to keep real-time frame rates, and you can see that total time is about 10ms without visibility.

Visibility 😐

	Starting res. : 1/4 Levels: 4 Samples: 64		Starting res. : 1/4 Levels: 4 Samples: 256		Starting res. : 1/2 Levels: 4 Samples: 64		Starting res. : 1/2 Levels: 5 Samples: 64	
Visibility	Without	With	Without	With	Without	With	Without	With
Voxelization (ms)		4.5		4.5		4.5		4.5
Indirect lighting (ms)	2.3	50.2	8.9	188	7.8	125	7.85	129
Upsampling & Compositing (ms)	2.3	2.3	2.3	2.3	2.7	2.7	2.7	2.7
Total (ms)	**4.6**	57	**11.2**	195	**10.5**	132	**10.55**	136

GeForce GTX 260 @ 1280x720

33

But unfortunately, visibility adds too many extra texture lookup per pixel and is then not usable in games.

Trade off

	Starting res. : 1/4 Levels: 4 Samples: 64		Starting res. : 1/4 Levels: 4 Samples: 256		Starting res. : 1/2 Levels: 4 Samples: 64		Starting res. : 1/2 Levels: 5 Samples: 64	
Visibility	Without	With	Without	With	Without	With	Without	With
Voxelization (ms)		4.5		4.5		4.5		4.5
Indirect lighting (ms)	2.3	50.2	8.9	188	7.8	125	7.85	129
Upsampling & Compositing (ms)	2.3	2.3	2.3	2.3	2.7	2.7	2.7	2.7
Total (ms)	**4.6**	57	**11.2**	195	**10.5**	132	**10.55**	136

GeForce GTX 260 @ 1280x720

34

We also observe that, for same computation time, it is better to start from a low resolution with a lot of samples, than high resolution with few samples.
So for all our examples and the video, we choose this configuration.

Trade off

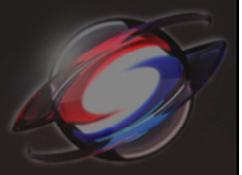
	Starting res. : 1/4 Levels: 4 Samples: 64		Starting res. : 1/4 Levels: 4 Samples: 256		Starting res. : 1/2 Levels: 4 Samples: 64		Starting res. : 1/2 Levels: 5 Samples: 64	
Visibility	Without	With	Without	With	Without	With	Without	With
Voxelization (ms)		4.5		4.5		4.5		4.5
Indirect lighting (ms)	2.3	50.2	8.9	188	7.8	125	7.85	129
Upsampling & Compositing (ms)	2.3	2.3	2.3	2.3	2.7	2.7	2.7	2.7
Total (ms)	**4.6**	57	**11.2**	195	**10.5**	132	**10.55**	136

GeForce GTX 260 @ 1280x720

35

It is also interesting to notice that, adding level to the pipeline, to increase the range, has a very low overhead compared to increasing the sampling radius.

Discussion



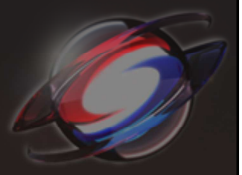
- Memory cost
  - But works fine on PC
- Indirect Visibility
  - Computationally expensive
  - Not visually relevant - [*Perceptual Influence of Approximate Visibility in Indirect Illumination* - Yu *et al* 2009]

36

One of the main drawback may be the memory cost. We have to store many render targets in video memory. And this could be a problem on video game platform, but it is still works fine on current PC hardware.

The visibility computation is too costly to allow real time performance. We could improve it by using a more approximate solution, but actually it doesn't worth it. The quality gain is not sufficient to justify such a cost. This has been already discussed in recent research.

Conclusion



- Cool results at constant cost
- Real time !

- Use frequency information

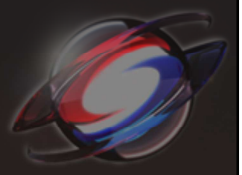
37

We present a new algorithm to compute approximate indirect illumination in real time. We take advantage of mipmaps, to compute indirect lighting between any objects of the scene efficiently.

We also include other techniques to reduce variance and noise. So at the end, we get very cool results, at a constant cost, since we're completely independent of geometry and lighting complexity.

For future improvement, as indirect lighting is low frequency, we think about using screen space frequency information, to automatically adapt starting level and number of samples, for optimal quality and performance.

References



- Technical report

  http://artis.imag.fr/Membres/Cyril.Soler/SSIL/ssil.techreport.2010.pdf

- Contact:

  olivier.hoel@inrialpes.fr

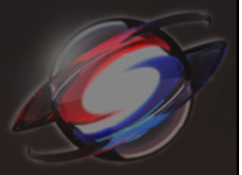
  csoler@imag.fr

  frochet@edengames.com

38

For more details and other references, you can have a look to our technical report, or contact us directly.

Thanks !



- Technical report

  http://artis.imag.fr/Membres/Cyril.Soler/SSIL/ssil.techreport.2010.pdf
- Contact:

  olivier.hoel@inrialpes.fr / csoler@imag.fr / frochet@edengames.com
- Acknowledgments

  K. Subr & N. Holzschuch (INRIA) for nice reviews

  Engineers and artists from Eden Games for useful advices