IVB Atmospheric Light Scattering

August 6, 2012

1. Introduction

Atmospheric light scattering is an important natural phenomenon, which arises when light interacts with the particles distributed in the media. Rendering such effects can be exploited by many applications, such as computer games, to greatly improve scene realism. To accurately compute scattering contribution, a complex nested integral has to be solved for each screen pixel. Due to the complexity of the computations involved, achieving natural-looking atmospheric scattering effects at interactive frame rates is a challenging problem.

The technique demonstrated by this sample combines a number of recent approaches for rendering light scattering effect in participating media as well as several optimization techniques. It exploits epipolar sampling [ED10] to significantly reduce the number of samples for which computationally expensive ray marching is performed, while ray marching itself is accelerated with the 1D min/max mipmaps [CBDJ11]. This enables achieving high quality rendering at interactive frame rates on Intel processor graphics.

Since the technique implemented in this sample is primarily based on concepts presented in [ED10] and [CBDJ11], it is highly recommended to read these papers.

2. Light scattering basics

There are three main phenomena that affect the light as it goes through the media:

- **Absorption** is when electromagnetic energy is transformed to other forms of energy for example, heat
- **Emission** is radiating electromagnetic energy
- **Scattering** is changing the direction of light

Emission and absorption can be neglected for air, so we will consider effects of scattering only. Theoretically, a photon can reach the eye after a number of scattering events. Complete equation describing light transport in participating media which takes multiple scattering into account is very hard to solve. So in real time rendering single scattering model is usually used which assumes that the light is scattered only once in the media and no further scatterings are taken into account.

Scattering effects affect the scene objects in two ways which account for phenomenon called *aerial perspective* (fig.1). From one hand some portion of light L_O initially emitted from the object is out-scattered due to interactions with particles. From the other hand, some sun light is scattered towards the camera. Thus the final radiance measured at the camera is a sum of two contributions: attenuated object radiance and inscattering:

$$L = L_O \cdot F_{ex} + L_{Insctr} \quad (1)$$

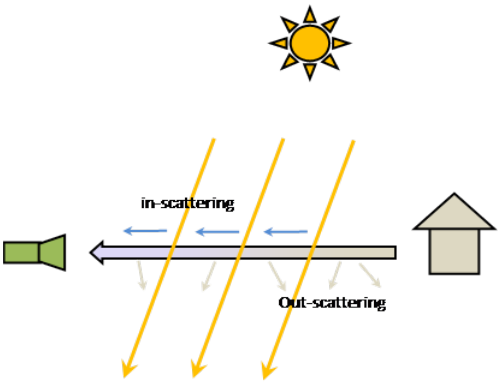

Fig.1: Scattering in participating media creates the effect of aerial perspective

Note that light intensity L is the function of 3 variables: the position in space $\mathbf{x}$, direction $\mathbf{v}$ and wavelength λ.

Light scattering in some point in space is described by two parameters: the scattering coefficient $\beta(\mathbf{x}, \lambda)$ which depends on position and wavelength and the phase function $p(\mathbf{v}, \mathbf{l})$ which takes view and light directions as the arguments. The scattering coefficient describes which portion of light is scattered per unit length in any direction. The phase function describes the angular distribution of the scattered light. Since each scattered photon must go somewhere, the phase function must be normalized such that $\int_{\Omega} p(\mathbf{v}, \mathbf{l}) d\omega = 1$ where integration is performed over the whole set of directions Ω.

Now let us consider the two parts of equation (1): the extinction F_{ex} and the inscattering $L_{Inscatr}$.

Extinction

The amount of light scattered per some differential ray section ds is proportional to the light intensity, the scattering coefficient and the section length:

$$dL_{Scatr}(\mathbf{x}, \mathbf{v}, \lambda) = \beta(\mathbf{x}, \lambda) \cdot L(\mathbf{x}, \mathbf{v}, \lambda) \cdot ds \quad (2)$$

Since scattered amount of light is removed from the initial intensity, light attenuation by out-scattering is given by the following differential equation:

$$dL(\mathbf{x}, \mathbf{v}, \lambda) = -\beta(\mathbf{x}, \lambda) \cdot L(\mathbf{x}, \mathbf{v}, \lambda) \cdot ds \quad (3)$$

Integrating this over the whole path from the camera position $\mathbf{C}$ to the object $\mathbf{O}$ will give us the following formula for the attenuated light reaching the camera:

$$L(\mathbf{C}, \mathbf{v}, \lambda) = L_0 \cdot e^{-\int_{\mathbf{C}}^{\mathbf{O}} \beta(\mathbf{x}, \lambda) \cdot ds} \quad (4)$$

where L_0 is the radiance at the end of the ray (emitted by the object). Since in general scattering coefficient $\beta(\mathbf{x}, \lambda)$ is not constant and varies along the ray, there is no closed form for integral (4).

Optical depth or *thickness* along the path from point **A** to **B** is given by the following equation:

$$T(\mathbf{A}, \mathbf{B}, \lambda) = \int_{\mathbf{A}}^{\mathbf{B}} \beta(\mathbf{x}, \lambda) \cdot ds \quad (5)$$

Extinction coefficient is related to optical depth as follows:

$$F_{ex}(\mathbf{A}, \mathbf{B}, \lambda) = \exp(-T(\mathbf{A}, \mathbf{B}, \lambda))$$

Inscattering

Inscattering is more complex to compute. Let us again consider some differential ray segment ds (fig. 2). The total amount of light scattered at this point is given by (2), but now we consider the sun light scattered towards the camera. It is proportional to the sun intensity at this point and also must be modulated by the phase function to get the fraction of light which is scattered in exactly the view ray. To account for shadowing we also need to introduce visibility term $V(\mathbf{x})$ which equals 1 if point $\mathbf{x}$ is not in shadow and 0 otherwise. Thus, differential amount of light scattered towards the camera is given by the following equation:

$$dL_{Inscatr}(\mathbf{x}, \mathbf{v}, \lambda) = E_{Sun}(\mathbf{x}, \lambda) \cdot \beta(\mathbf{x}, \lambda) \cdot V(\mathbf{x}) \cdot p(\mathbf{v}, \mathbf{l})$$

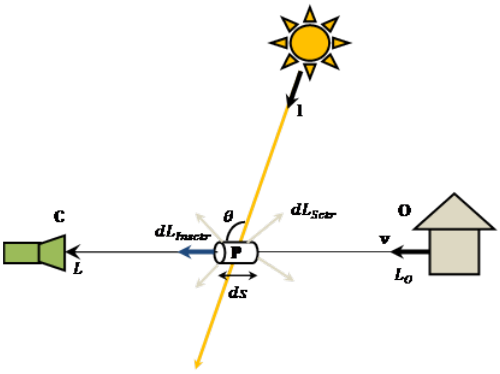

Fig.2: Inscattering contribution from differential ray segment

This inscattered light is attenuated before it reaches the camera. If we denote current position on the ray by $\mathbf{P} = \mathbf{C} + \mathbf{v} \cdot s$ then the total inscattering is given by integrating differential inscattering over the whole ray:

$$L_{Insctr}(\mathbf{C}, \mathbf{v}, \lambda) = \int_{\mathbf{C}}^{\mathbf{O}} dL_{Insctr}(\mathbf{x}, \mathbf{v}, \lambda) \cdot \exp\left(-T(\mathbf{P}, \mathbf{C}, \lambda)\right) ds$$

Or

$$L_{Insctr}(\mathbf{C}, \mathbf{v}, \lambda) = \int_{\mathbf{C}}^{\mathbf{O}} E_{Sun}(\mathbf{x}, \lambda) \cdot \beta(\mathbf{x}, \lambda) \cdot V(\mathbf{x}) \cdot p(\mathbf{v}, \mathbf{l}) \cdot \exp\left(-T(\mathbf{P}, \mathbf{C}, \lambda)\right) ds \quad (6)$$

Scattering properties of air

Air is usually modeled as a mix of two types of particles: molecules and aerosols. Rayleigh scattering theory describes scattering on molecules with diameter $r < 0.1\lambda$. The scattering probability depends only on the angle θ between the light direction and the scattering direction (fig. 2). Normalized phase function for Rayleigh particles is given by the following equation:

$$p_{Rlgh}(\theta) = \frac{3}{16\pi}(1 + \cos^2\theta) \quad (7)$$

Scattering on air molecules is wavelength-dependent and short wavelengths are scattered approximately 10 times as much as long wavelengths. This is why the sky is blue.

The scattering on aerosols is more complex and is described by Mie theory. In atmospheric scattering, Mie phase function for haze is commonly approximated with the Henyey-Greenstein phase function:

$$p_{Mie}(\theta) = \frac{1-g^2}{4\pi(1+g^2-2g\cos\theta)^{3/2}} \quad (8)$$

A detailed derivation of scattering coefficients for Rayleigh and Mie particles can be found in [PSS99], [NSTN93] and [HP02]. We will be using the following values:

$$\beta_{Rlgh}.\mathrm{rgb} = (5.8, 13.5, 33.1) \times 10^{-6}$$

$$\beta_{Mie}.\mathrm{rgb} = 2.0 \times 10^{-5}$$

Note that these values must be adjusted to match the scene scale before using.

Assumptions

Since integral (6) is very complex to compute, a number of simplifications are usually made. We will follow [HP02] and assume that that

1. Scattering coefficients do not depend on position in space (homogeneous media): $\beta(\mathbf{x}, \lambda) = \beta(\lambda)$

2. Sun intensity is constant: $E_{Sun}(\mathbf{x}, \lambda) = E_{Sun}(\lambda)$

These simplifications are reasonable for rendering the scattering effects at the ground. For more general solutions refer to [NSTN93], [BN08].

Under these assumptions equation (5) for optical depth simplifies to the following:

$$T(\mathbf{A}, \mathbf{B}, \lambda) = \int_{\mathbf{A}}^{\mathbf{B}} \beta(\mathbf{x}, \lambda) \cdot ds = \left(\beta_{Rlgh}(\lambda) + \beta_{Mie}(\lambda)\right) \int_{\mathbf{A}}^{\mathbf{B}} ds = \left(\beta_{Rlgh}(\lambda) + \beta_{Mie}(\lambda)\right) \|\mathbf{A} - \mathbf{B}\| \quad (9)$$

Where $\|\mathbf{A}-\mathbf{B}\|$ is the distance between points $\mathbf{A}$ and $\mathbf{B}$.

Now we can rewrite equation (6) for in-scattering:

$$L_{Insctr}(\mathbf{C},\mathbf{v},\lambda) = E_{Sun}(\lambda)\sum_{i=Rlgh,Mie}\beta_i(\lambda)\cdot p_i(\theta)\int_{\mathbf{C}}^{\mathbf{O}} V(\mathbf{x})\cdot\exp\left(-\left(\beta_{Rlgh}(\lambda)+\beta_{Mie}(\lambda)\right)s\right)ds \quad (10)$$

If we drop visibility term $V(\mathbf{x})$ for the moment, we will be able to solve integral (10) analytically:

$$\tilde{L}_{Insctr}(\mathbf{C},\mathbf{v},\lambda) = E_{Sun}(\lambda)\left(\beta_{Rlgh}(\lambda)\cdot p_{Rlgh}(\theta)+\beta_{Mie}(\lambda)\cdot p_{Mie}(\theta)\right)\frac{1-\exp\left(-\left(\beta_{Rlgh}(\lambda)+\beta_{Mie}(\lambda)\right)\|\mathbf{O}-\mathbf{C}\|\right)}{\beta_{Rlgh}(\lambda)+\beta_{Mie}(\lambda)} \quad (11)$$

If we introduce the following notations:

$$F(\lambda,\theta) = E_{Sun}(\lambda)\frac{\beta_{Rlgh}(\lambda)\cdot p_{Rlgh}(\theta)+\beta_{Mie}(\lambda)\cdot p_{Mie}(\theta)}{\beta_{Rlgh}(\lambda)+\beta_{Mie}(\lambda)} \quad (12)$$

$$I(\lambda,s) = -\exp\left(-\left(\beta_{Rlgh}(\lambda)+\beta_{Mie}(\lambda)\right)s\right) \quad (13)$$

then (11) can be rewritten as follows:

$$\tilde{L}_{Insctr}(\mathbf{C},\mathbf{v},\lambda) = F(\lambda,\theta)\cdot\left(1+I(\lambda,\|\mathbf{O}-\mathbf{C}\|)\right) \quad (14)$$

Since there is no in-scattering contribution in shadowed region, integral (10) can be re-written as a sum of contributions from lit sections only (fig. 3). If we subdivide the ray into the lit sections $[\mathbf{S}_i,\mathbf{E}_i]$, we will be able to rewrite (10) as follows:

$$L_{Insctr}(\mathbf{C},\mathbf{v},\lambda) = \sum_{i=0}^{N}\left(\tilde{L}_{Insctr}(\mathbf{E}_i,\mathbf{v},\lambda)-\tilde{L}_{Insctr}(\mathbf{S}_i,\mathbf{v},\lambda)\right) \quad (15)$$

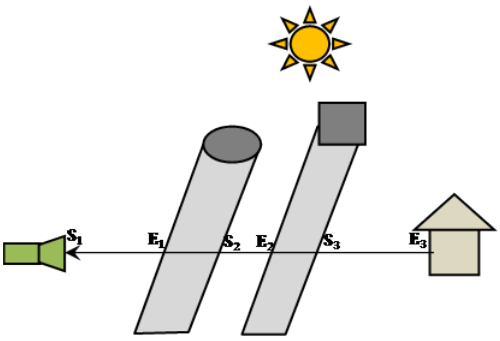

Fig.3: Partitioning view ray into lit sections

Using notations (12) and (13), equation (15) can be rewritten as follows:

$$L_{Insctr}(\mathbf{C},\mathbf{v},\lambda) = F(\lambda,\theta)\sum_{i=0}^{N}\left(I(\lambda,\|\mathbf{E}_i-\mathbf{C}\|)-I(\lambda,\|\mathbf{S}_i-\mathbf{C}\|)\right) \quad (16)$$

Using (16) we can now formulate our initial algorithm for calculating inscattering integral:

1. Subdivide the ray into *N* segments

2. Set up $L_{Insct}.\mathrm{rgb}=0$, $PrevI.\mathrm{rgb}=-1$ (this is $I(\lambda_{rgb},0)$)

3. For each segment *i* do the following:

a. Compute $CurrI.\mathrm{rgb}=I(\lambda_{rgb},d_i)$ where $d_i=\|\mathbf{E}_i-\mathbf{C}\|$ is the distance to the end of the current ray section

b. If current section is not in shadow, then $L_{Insct}.\mathrm{rgb}\mathrel{+}=CurrI.\mathrm{rgb}-PrevI.\mathrm{rgb}$

c. $PrevI.\mathrm{rgb}=CurrI.\mathrm{rgb}$

$L_{Insct}.\mathrm{rgb}\mathrel{*}=F(\lambda_{rgb},\theta)$

Algorithm 1: Computing in-scattering integral.

3. Epipolar sampling

To apply volumetric lighting effects, we need to cast a ray from the camera through each pixel and execute Algorithm 1, which is too computationally expensive. So it is necessary to find a way to reduce the number of computations involved. Light shafts seen on the screen has special structure: they all emanate from the position of the sun on the screen. Engelhardt and Dachsbacher [ED10] noticed that the inscattered light varies orthogonally to these rays, but mostly smoothly along them. To account for this property they proposed an efficient sampling scheme. Their idea is to locate ray marching samples sparsely along epipolar lines going from the sun position to the screen borders with additional samples placed at depth breaks (fig. 4). Since light intensity varies smoothly along the rays, the intensity can be linearly interpolated from sparsely placed ray marching samples.

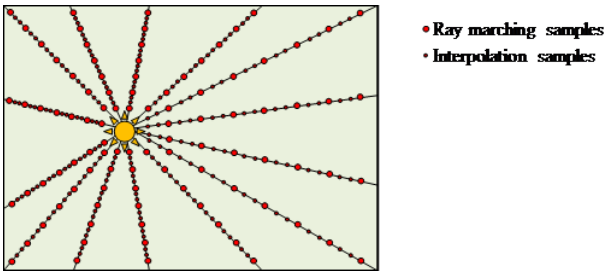

Fig.4: Epipolar sampling with ray marching and interpolation samples

The algorithm proposed by Engelhardt and Dachsbacher [ED10] consists of the following steps:

- Render the scene from the camera and from the light source
- Compute the inscattering contribution
  - Sparsely locate initial ray marching samples along epipolar lines
  - Place additional samples at depth discontinuities
  - Perform ray marching for the selected samples
  - Interpolate inscattering for remaining samples from ray marching samples
  - Compute scattering for each pixel using interpolation from nearby epipolar lines
- Attenuate background and combine with inscattering

In our implementation we follow basic ideas of Engelhardt and Dachsbacher with some improvements and modifications discussed in section 5.

4. 1D min/max mipmap optimization

Epipolar sampling has one important property: all camera rays in an epipolar slice share the same plane. Intersection of this plane with the shadow map essentially forms a one-dimensional height map. Shadow test in Algorithm 1 is intrinsically a check if current position on the ray is under this height map or above it (fig. 5).

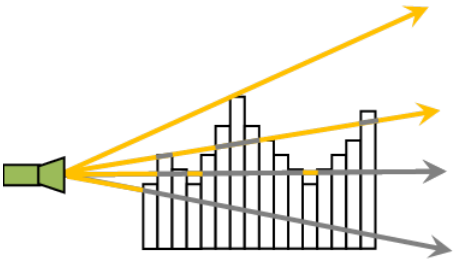

Fig.5: All camera rays in one epipolar slices are tested against the same 1D height map

The property of all camera rays from one epipolar slice using the same 1D height map was recognized by Chen et al [CBDJ11]. To accelerate ray marching they proposed constructing 1D min/max binary tree for each epipolar slice (fig. 6) and using this structure to identify long lit and shadowed regions on the ray.

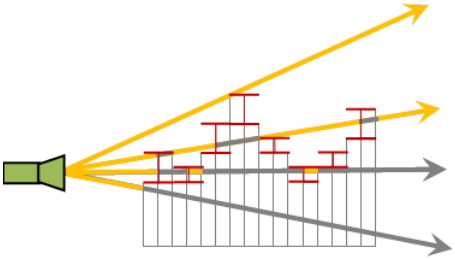

Fig.6: First level of min/max binary tree for the epipolar slice

Consider fig. 7. If the maximum of depths at the ends of the current ray section is less than the minimum depth stored in the min/max tree, then current ray section is completely lit and we can add inscattering contribution. This condition is true for section **AB**: $\max(d_A, d_B) < d_{min}$, which means that during the next four steps (because this is the second level of the tree) of the Algorithm 1, the current point will be in light. Thus instead of doing 4 iterations we can safely do just one without expensive shadow map fetches and obtain the same result.

From the other hand, if the minimum of depths at the ends is greater than the maximum value stored in the tree, current ray section is fully in shadow and we can skip it. This holds true for section **CD**: $\min(d_C, d_D) > d_{max}$. In this case we can safely advance by 4 steps along the ray without introducing any error.

It is also possible that neither condition is true which is the case for section **EF**. In this case it is necessary to go down to the next finer tree level and repeat the test. If we have reached the original shadow map at this point, we should perform the test using it.

Note that in practice we use complimentary depth buffering, so that all checks are inverted.

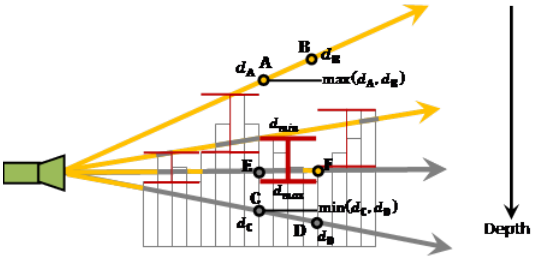

Fig.7: Using second level of min/max binary tree to determine shadowed/lit regions

The binary tree traversal algorithm presented by Chen et al [CBDJ11] is essentially an adaptation of ray/height map intersection method described in [TIS08] where the traversal is done without recursion. We adopt the same idea in our sample with some differences discussed below. Having 1D min/max binary tree for the slice, the Algorithm 1 can be improved as shown in Algorithm 2. Note that ray marching is done in shadow map space. For this, end position of the ray is projected onto the shadow map and sampling is done along the resulting projected line [GMF09].

1. Project start and end points of the view ray onto shadow map, compute UV_{Start}, UV_{End}, UV_{Step}

2. Set up $L_{Insct}.\text{rgb} = 0$, $PrevI.\text{rgb} = -1$

3. Set up $Level = 0$, $SampleInd = 0$

4. Set up $UV = UV_{Start}$

5. **until** ray is marched, do the following

5.1. **if** $(SampleInd \bmod 2^{Level+1}) == 0$, then $Level = Level + 1$

5.2. **while** $Level > 0$

5.2.1. Compute depths d_{Start} and d_{End} at the ends of the current ray section taking into account scale 2^{Level} of the current level step

5.2.2. Fetch min/max depth d_{min} and d_{max} for level $Level$ at position $SampleInd$

5.2.3. **if** $\min(d_{Start}, d_{End}) > d_{max}$, then $IsInShadow = true$, **break**

5.2.4. **else if** $\max(d_{Start}, d_{End}) < d_{min}$, then $IsInShadow = false$, **break**

5.2.5. **else** $Level = Level - 1$

5.2.6. **if** $Level == 0$, then compute $IsInShadow$ by sampling shadow map at location UV

5.3. Compute $CurrI.\text{rgb} = I(\lambda_{rgb}, d_i)$ where $d_i = \|\mathbf{E}_i - \boldsymbol{C}\|$ is the distance to the end of the current ray section taking into account scale 2^{Level} of the current level step

5.4. **if** $IsInShadow == false$, then $L_{Insct}.\text{rgb} += CurrI.\text{rgb} - PrevI.\text{rgb}$

5.5. $PrevI.\text{rgb} = CurrI.\text{rgb}$

5.6. $UV = UV + UV_{Step} \cdot 2^{Level}$

5.7. $SampleInd = SampleInd + 2^{Level}$

6. $L_{Insct}.\text{rgb} *= F(\lambda_{rgb}, \theta)$

Algorithm 2: Optimized version of the in-scattering integral calculation.

There are a number of important differences compared to Algorithm 1. Step length in Algorithm 1 is fixed, while in Algorithm 2 it is scaled by 2^{Level}, which is the key to performance improvement. As a result, Algorithm 1 always executes the same number of steps, while number of iterations Algorithm 2 performs depends on the complexity of the camera ray intersection with the shadow map. The most important differences are in steps 5.1 and 5.2 where min/max tree is accessed. Step 5.1 is responsible for going to the next coarser level of the tree at appropriate locations. Loop in the step 5.2 executes until it is detected that at some tree level current ray section is fully shadowed (step 5.2.3) or fully lit (step 5.2.4), or finest level is reached (step 5.2.6). Note that computational results of both algorithms are identical.

There are also a number of differences with original 1D min/max mipmap acceleration algorithm by Chen et al [CBDJ11]:

- [CBDJ11] exploits a sophisticated mathematical approach based on singular value decomposition to bring view ray-dependent terms out of the shadow test. They are also required to use a partial sum tree structure to compute scattering contributions on a ray section. In our method, we use an analytical approach that is both simpler and more accurate.
- In the original algorithm, it is necessary to execute brute force ray marching for the small area near the epipole, without using the 1D min/max mipmap, which can be expensive. In our algorithm, we do not have such problems and can process all the rays using accelerated traversal.
- [CBDJ11] also performs nonlinear transformation of the shadow map. For directional light sources, their shadow map stores the angle to the light direction. This involves additional computational overhead. In our approach we use just the depth seen from the light source. We also construct min/max mipmap directly from the original shadow map.

Notice that Engelhardt and Dachsbacher [ED10] tried using a 1-dimensional min-max depth mipmap in their method but for the purpose of searching depth discontinuities along epipolar lines, not for accelerating ray marching.

5. The algorithm

Our algorithm combines epipolar sampling by Engelhardt and Dachsbacher [ED10] with 1D min/max mipmap by Chen et al. [CBDJ11]. It goes through the following steps:

1. Render the scene from the camera and from the light source
2. Reconstruct linear camera space Z from the depth buffer
3. Build 1D min/max mipmap
4. Render coordinate texture
   - Samples are placed along the lines connecting the light source projected position with a user-defined number of points equidistantly placed along the border of the screen
   - If the light source is outside the screen, the line is intersected with the screen border and samples are placed between the intersection points
   - Samples that fall outside the screen are excluded from further processing
5. Detect depth discontinuities and refine initially placed sample locations
   - Initial samples are placed to capture light variation in the absence of occludes

    - Additional samples are placed at locations where camera space z difference between neighboring pixels exhibit a large threshold

6. Perform ray marching for selected samples
7. Interpolate inscattering radiance in epipolar coordinates for the rest of the samples from ray marching samples
8. Transform inscattering from epipolar coordinates to downscaled rectangular buffer
    - Determine the two closest epipolar lines
    - Project the sample onto the lines and perform bilateral bilinear filtering taking into account the z difference
    - Mark pixels for which there is no appropriate source samples
9. Correct inscattering for these samples, which could not be correctly interpolated from epipolar coordinates by performing ray marching
10. Upscale the inscattering image to the original resolution and combine it with the attenuated background
    - Perform bilateral filtering
    - Mark pixels for which there is no appropriate source samples
11. Correct inscattering

While the most important concepts of the original epipolar sampling algorithm [ED10] are preserved, there are a number of improvements:

- We use the additional down-sampling step to provide additional control over speed/quality tradeoff. Usually, downscaling by a factor of 2x does not significantly degrade visual quality, at the same time making the rays look smoother and improving performance
- Additional inscattering correction steps are introduced:
    - In the original approach, the authors used bilateral filtering along and between epipolar lines when performing transformation from epipolar geometry to the original rectangular coordinates. Up to five samples along each epipolar line were used. If there is no appropriate sample to filter from, the authors proposed going to the next outer epipolar line. We found this approach still produces rendering artifacts and can't be efficiently implemented on the GPU because it requires branching.
    - We implemented another method: we mark samples that cannot be correctly interpolated with stencil and perform an additional ray marching pass for these pixels using fewer steps. Note that this correction is done when both epipolar inscattering is transformed into rectangular geometry and when upscaling is performed.

o We implemented another method: we mark samples which cannot be correctly interpolated with stencil and perform additional ray marching pass for these pixels using lower number of steps. Note that this correction is done when both epipolar inscattering is transformed into rectangular geometry and when upscaling is performed

6. Implementation details

The algorithm implementation follows steps described in section 5. There are a number of textures which are used to store intermediate data required during the processing, which are summarized in table 1.

Texture name	Format	Dimension	Description
Coordinate texture	RG_FLOAT32	$N_{Samples} \times N_{Slices}$	Stores screen coordinates for each epipolar sample
Epipolar depth-stencil	D24_S8	$N_{Samples} \times N_{Slices}$	Depth-stencil buffer used to select samples for processing
Camera space Z	R_FLOAT32	$N_{Samples} \times N_{Slices}$	Stores camera space Z for each epipolar sample
Interpolation source	RG_UINT16	$N_{Samples} \times N_{Slices}$	Stores indices of the two interpolation sources for the current epipolar sample
Initial inscattering	RGBA_FLOAT16	$N_{Samples} \times N_{Slices}$	Stores initial inscattering computed during ray marching step
	RGBA_FLOAT16		

Texture name	Format	Dimension	Description
Interpolated inscattering		$N_{Samples} \times N_{Slices}$	Stores inscattering interpolated from ray-marching samples
Downscaled inscattering	RGBA_FLOAT16	$W/F \times H/F$	Stores downscaled inscattering in rectangular coordinates. *W* and *H* are width and height of the screen and *F* is the downscaling factor.
Downscaled screen depth-stencil	D24_S8	$W/F \times H/F$	Used to mark samples in downscaled inscattering texture that require correction
Slice UV direction	RG_FLOAT32	$N_{Slices} \times_1$	Auxiliary 1D texture storing direction of each epipolar slice in shadow map UV space
1D min/max shadow map × 2	RG_FLOAT16	$S \times N_{Slices}$	Texture storing 1D min/max shadow map. *S* is the shadow map resolution

Table 1: Textures used by the algorithm implementation

The algorithm workflow is summarized in fig. 8 and 9, while the rest of this section provides various details.

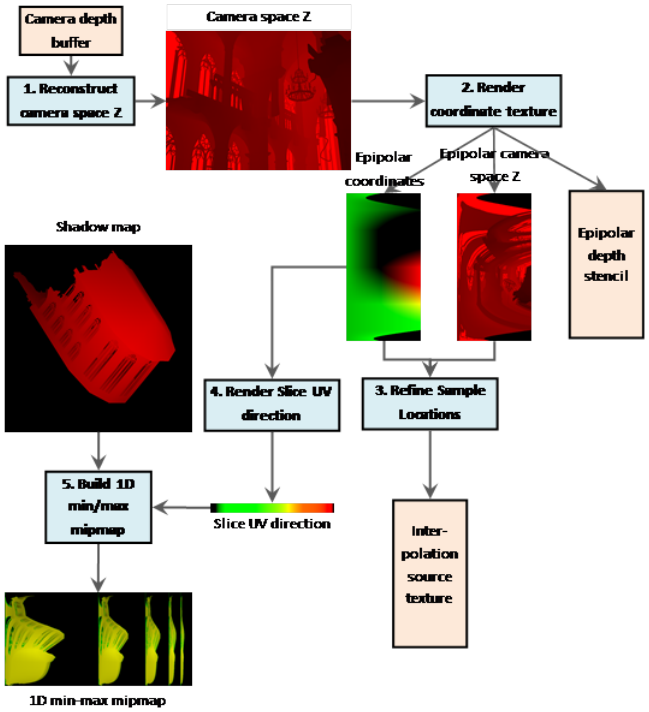

Fig.8: Preliminary steps of the algorithm

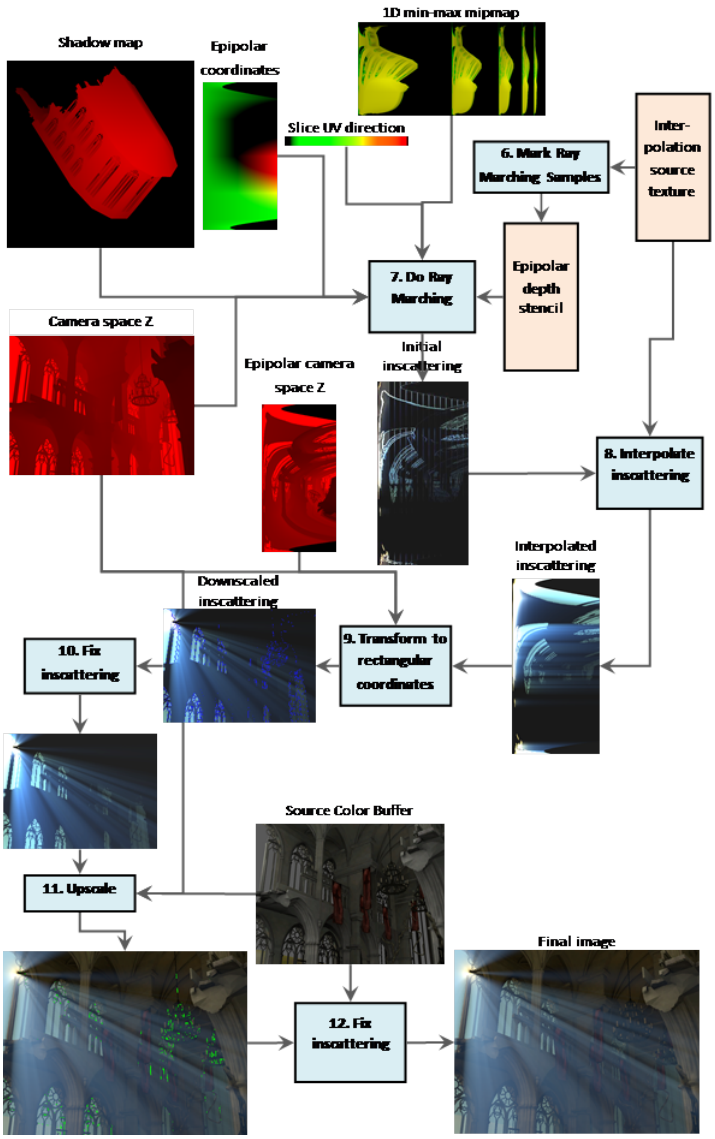

Fig.9: Final steps of the algorithm

Note that inscattering correction at steps 10 and 12 also uses camera space z, shadow map and slice UV direction textures, which is not shown in fig. 9 for clarity.

The remaining part of this section details all the stages of the algorithm.

6.1. Rendering coordinates texture

Coordinate texture generation is done by rendering screen-size quad with the texture set up to the pipeline as a render target. Pixel shader `GenerateCoordinateTexturePS()` performs all the required processing. Depth stencil state is configured to increment stencil value, thus all valid samples will be marked by 1 in the stencil, while all invalid will keep initial 0 value. Thus samples that fall behind the screen will be skipped from all further processing.

We assume that epipolar slice coordinate ranges from 0 to 1. Screen borders are traversed in a counter clockwise order starting from the left top corner (fig. 10): values from 0 to 0.25 define locations on the left border, values in the ranges from 0.25 to 0.25, from 0.5 to 0.75 and from 0.75 to 1.0, define locations on bottom, right and top borders correspondingly. Values 0, 0.25, 0.5, 0.75, 1 define locations in exactly the screen corners.

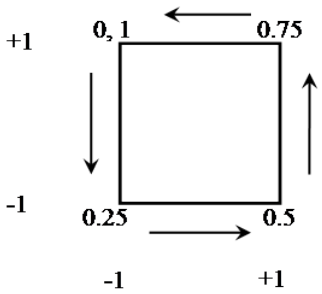

Fig.10: Screen borders traversal order

The stage consists of the following steps:

- Computing epipolar line exit point
- Computing epipolar line entry point given its exit point
- Rescaling epipolar line length to provide even texel to screen pixel correspondence
- Computing camera space z for the location

Computing epipolar line exit point is relatively simple as it lies on one of the four screen boundaries. The following code effectively computes this location using arithmetic instructions only:

```
1 uint uiBoundary = clamp(floor( fEpipolarSlice * 4 ), 0, 3); float fPosOnBoundary
-fBoundaryCoord); float4 f4BoundaryYCoord = float4(-fBoundaryCoord, -1, fBoundar
(f4BoundaryXCoord, b4BoundaryFlags), dot(f4BoundaryYCoord, b4BoundaryFlags));
```

The next step is a bit more complex: we need to compute epipolar line entry point given its exit point and position of the light source on the screen. This is accomplished by the `GetEpipolarLineEntryPoint()` function. There are two possible cases: light is located inside the screen and outside it (fig. 11). The first case is simple: entry point is simply the position of the light on the screen. In the second case we have to find intersection of the epipolar line with the appropriate boundary.

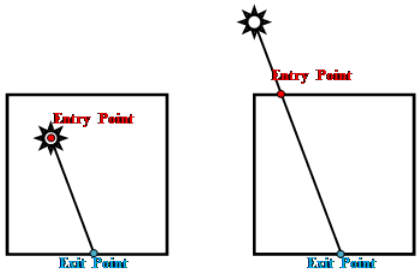

Fig.11: Entry and exit points of an epipolar line when light source is on the screen and outside it

Our task is to find first intersection of the ray connecting light projected position and the exit point before the exit point (fig. 12).

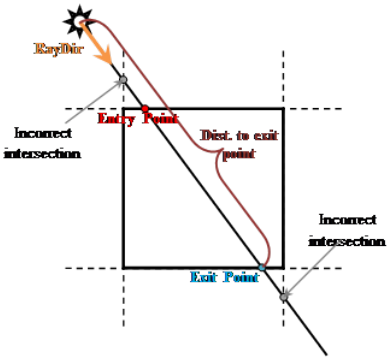

Fig.12: Computing entry and exit points of an epipolar line when light source is outside the screen

For this, we compute signed distances to left, bottom, right and top boundaries along the ray and find the maximum distance which is less then distance to the exit point (examine fig. 12). We also take care of near horizontal and near vertical ray orientations when distances to left and right or top and bottom boundaries cannot be properly computed. We use special flag vector to skip computations with incorrect boundaries. The following code snippet accomplishes this task using mathematical instructions only and avoiding branches:

```
// Compute direction from the light source to the ray exit point: float2 f2RayDi
the ray from the light position to all four boundaries bool4 b4IsCorrectIntersec
b4IsCorrectIntersectionFlag); // Addition of !b4IsCorrectIntersectionFlag is rec
boundary // This means that we need to find maximum intersection distance which
these boundaries will result in skipping them: b4IsCorrectIntersectionFlag = b4I
b4IsCorrectIntersectionFlag * float4(-FLT_MAX, -FLT_MAX, -FLT_MAX, -FLT_MAX); fI
fFirstIntersecDist = max(fFirstIntersecDist, f4DistToBoundaries.z); fFirstInters
fFirstIntersecDist;
```

Note that if light source is located outside the screen, there could be several cases when the whole slice is not visible (fig. 13). For such cases coordinates of the entry point will also be outside the screen.

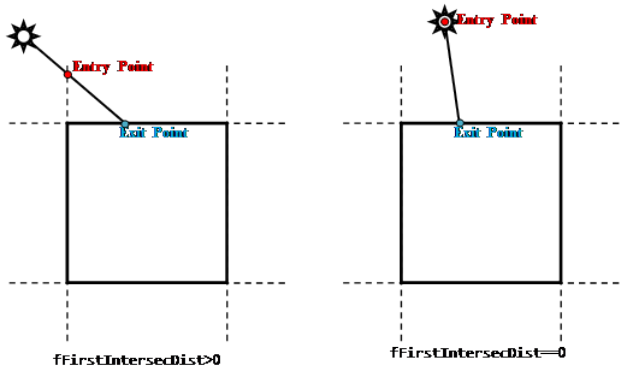

Fig.13: Entry point of a completely invisible epipolar line

Such pixels are easily detected and discarded so that they will be skipped from further processing.

```
if( any(abs(f2EntryPoint) > 1+1e-4) ) discard;
```

If light source is located close to screen boundary, the screen length of epipolar lines could vary significantly. This will result in using too dense sampling for short lines and doing redundant calculations and also could cause aliasing artifacts. To solve this issue, we rescale the epipolar lines by advancing exit point (fig. 14). We strive to provide 1:1 correspondence between samples on the epipolar line and screen pixels. The following code updates epipolar line exit point:

```
float fEpipolarSliceScreenLen = length( (f2ExitPoint - f2EntryPoint) * g_PPAttri
```

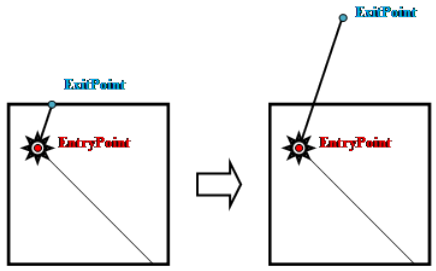

Fig.14: Advancing epipolar line exit point to provide even sample to screen pixel correspondence

This step not only reduces the amount of computations necessary, but also results in a more natural circular-shaped distributions of samples against rectangular-shaped distribution in original algorithm.

Finally, we compute interpolated location of the sample between entry and exit points and discard these samples that fall outside the screen:

```
f2XY = lerp(f2EntryPoint, f2ExitPoint, fSamplePosOnEpipolarLine); if( any(abs(f2
```

The shader also outputs camera space Z coordinate for the epipolar sample which is used by subsequent steps of the algorithm.

Note that coordinate texture as well as camera space z texture are initially cleared with incorrect coordinates which are outside the allowable ranges. This is important since these values will help skip such pixels from further processing.

6.2. Refining sample locations

The next step of the algorithm is refining initially placed ray marching sample locations by finding depth discontinuities. This stage generates interpolation source texture which for each sample contains indices of two samples from the same slice, from which current sample will be interpolated. Basically, the algorithm performing search for the interpolation samples (indexed by `left` and `right`) in the depth array `depth1d` of size `N`, for the current texel at location `x` as presented in [ED10] is the following:

```
left = right = x; while ( left > 0 ) { if(abs( depth1d[left-1], depth1d[left] )
```

Algorithm 3: Searching for depth discontinuities.

If there is no depth discontinuities on the ray section, the interpolation sources are the end point of this section. If depth discontinuity is detected, the sample will be interpolated from sample placed directly before the break.

We tried several strategies while implementing search for depth discontinuities. Our first approach was straightforward implementation of the algorithm 2 in pixel shader as suggested by [ED10]. We found out that this implementation is too slow, especially when large initial sampling step (32 and greater) is used.

We implemented the optimized search algorithm using compute shader. The performance of the implementation is up to 6x higher than the performance of the original pixel-shader based approach and almost independent of the initial sampling step.

The compute shader organizes the work it performs into groups of threads. Number of threads in each group must be at least 32 and not less than the initial sample step due to the reasons which will be clear a bit later. Each group of threads processes one or several sections in one epipolar slice. If initial sample step S_i is less than the total number of threads N_t in a group, then the group processes N_t/S_i ray sections. Otherwise ($N_t=S_i$) each group processes one section. Location of one sample group within interpolation source texture is illustrated in fig. 15.

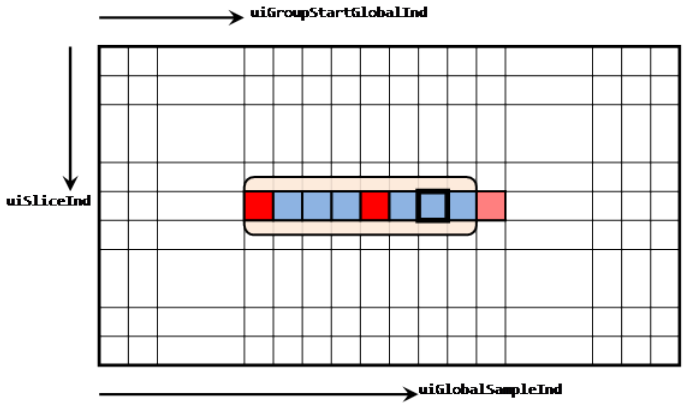

Fig.15: Location of the thread group within coordinate texture

Location of individual sample processed by one thread in the group is shown in fig. 16.

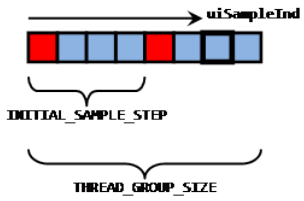

Fig.16: Location of individual sample processed by one thread in the group

The number of threads in the group must be large enough to fully utilize the SIMD units, so we set it to be at least 128. Our implementation exploits group shared memory exposed by the compute shader to efficiently implement discontinuities search. The idea is first to check if there is a depth break located next to each sample in the group and store this information in a compact shared array. Since 1 bit is sufficient to indicate depth break, we can pack 32 flags into one UINT variable. We use the following shared array:

```
1 static const uint g_uiNumPackedFlags = THREAD_GROUP_SIZE/32; groupshared uint g_
```

It is now clear why the group size must be at least 32. The array is then used to efficiently search for depth discontinuities.

At the very beginning the shader checks if the sample it processes is correct. Recall that coordinate texture is initially cleared with invalid coordinates, which indicate samples outside the screen:

```
1 bool bIsValidThread = all( abs(f2SampleLocationPS) < 1+1e-4 );
```

The shader then loads camera space z for current sample location and for its neighboring sample from the camera z coordinate texture generated at the previous stage, computes the difference and compares it with the threshold and sets up appropriate flag. Notice using `InterlockedOr()` function.

```
1 // Load camera space Z for this sample and for its right neighbor (remeber to us
  g_tex2DEpipolarCamSpaceZ.Load( uint3(uiGlobalSampleInd+1, uiSliceInd, 0) ); // C
  Or: InterlockedOr( g_uiPackedCamSpaceDiffFlags[uiSampleInd/32], bFlag << (uiSamp
```

An important aspect here is that camera space z texture contains large negative values for all samples which fall behind the screen. As a result, difference between the last sample on the screen and the first sample behind the screen will always exceed the threshold, and depth discontinuity will be detected.

After all the flags are set, all the threads must be synchronized which is done by the call to `GroupMemoryBarrierWithGroupSync()` function.

On the next step, the shader computes initial indices `uiInitialSample0Ind` and `uiInitialSample1Ind` of the source ray marching samples at the ends of the current ray section (fig. 17). Initial sample step is here adjusted to provide higher sampling density near the epipole to account for high variation of scattering intensity.

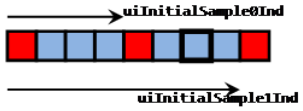

Fig.17: Initially placed ray marching samples at the ends of the ray section

If the sample index equals one of initial sample indices, the sample is ray marching and is interpolated from itself. Otherwise it is necessary to search for the two interpolation sources. But before performing the search itself, it is easy to check if there is at least one depth break on the current ray segment:

```
1 bool bNoDepthBreaks = true; #if INITIAL_SAMPLE_STEP < 32 { // Check if all uiIni
  uint uiFlagPack = uiPackedCamSpaceDiffFlags[iFlagPackOrder]; uint uiAllFlagsMask
  g_uiNumPackedFlags; ++i) if( uiPackedCamSpaceDiffFlags[i] != 0xFFFFFFFFU ) // It
```

If there are no depth breaks, no further processing is required. If it was found, that there is at least one depth break on current ray section, we perform search for two interpolation sources. Due to the efficient packing scheme, the search can be efficiently implemented using bit manipulation and two intrinsic functions: `firstbitlow()` and `firstbithigh()` which return bit position of the first non-zero bit starting from the lowest order bit and the highest order bit correspondingly. If initial sample step is not greater than 32, this implementation avoids using loops at all. If initial sample step is greater than 32, the number of iterations the loop is executed is at most $S_i/32$.

6.3. Building min/max mipmap

At this stage, a min/max mipmap is constructed. For this, a 1D auxiliary texture is first rendered which contains UV direction of the slice in the shadow map. The shader loads coordinates of the first sample in the slice (note that sample with the index 0 is the location of light source). It then reprojects this location from camera projection space to light space and computes direction from the light space position to the resulting location. Note that direction is normalized in a way that the it always covers one shadow map texel. Note also that complimentary depth buffering is used.

```
1 float2 RenderSliceUVDirInShadowMapTexturePS(SScreenSizeQuadVSOutput In) : SV_T
  f2FirstSampleInSliceLocationPS = g_tex2DCoordinates.Load( uint3(1, uiSliceInd,
```

```
f2FirstSampleInSliceUVInShadowMap = ProjToUV( f4FirstSampleInSlicePosInLightProj
g_LightAttribs.f4CameraUVAndDepthInShadowMap.xy; f2SliceDir /= max(abs(f2SliceDi
```

1D min/max mipmap is $S \times N_{Slices}$ in size where S is the resolution of original shadow map. Each row of the min/max mipmap contains 1D min/max binary tree for appropriate epipolar slice. The layout of the texture is shown in the figure below:

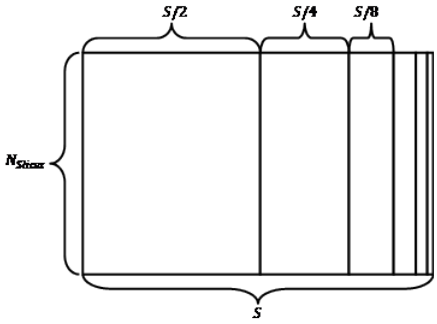

Fig.18: Layout of min/max mipmap texture

Note that the 1D min/max mipmap does not contain original shadow map, so its first level is two times down sampled and has $S/2$ samples. Note also that f2SliceDir is computed in a way that at most S steps are required to cross the shadow map texture.

Creating min/max shadow map is performed using well-known flip/flop approach when two textures are alternately used as a source and destination. Initial shadow map is used to initialize first level of the min/max binary trees. 1D min/max tree construction starts from the projected light source position. Note that `Gather()` instruction is used at this stage to load four source values which will be used to perform bilinear interpolation. In practice we do not construct full binary tree because it is very unlikely that coarse levels could be reached. Besides, rendering to low-resolution textures is inefficient on modern GPUs.

```
1  float2 InitializeMinMaxShadowMapPS(SScreenSizeQuadVSOutput In) : SV_Target { uir
   sample position on the ray float2 f2CurrUV = g_LightAttribs.f4CameraUVAndDepthIn
   immediate neighbor // along the epipolar slice float4 f4Depths = g_tex2DLightSpa
   f4NeighbDepths = g_tex2DLightSpaceDepthMap.Gather(samLinearBorder0, f2CurrUV); f
   max(f4Depths, f4NeighbDepths); f4MaxDepth.xy = max(f4MaxDepth.xy, f4MaxDepth.zw)
```

After that next levels of the binary trees are constructed by loading two min/max values from the next finer level and computing min/max value.

6.4. Ray marching

After the sampling refinement stage all ray marching samples are marked as being interpolated from themselves. Before performing ray marching, a screen-size quad is rendered with simple pixel shader discarding all pixels, which are not interpolated from themselves:

```
1  uint2 ui2InterpolationSources = g_tex2DInterpolationSource.Load( uint3(In.m_f4Po
```

Depth stencil state is configured to increment stencil value. As a result, all ray marching samples will be marked by the 2 in stencil.

After that another screen-size quad is rendered with depth stencil configured to pass only these pixels whose stencil values equals 2 and discards all other pixels. Pixel shader performs ray marching on the selected locations as described in Algorithm 2 presented in section 4.

Implementing colored light shafts is straightforward. For this we only need stained glass color texture as seen from the light source and on Each iteration of Algorithm 2 fetch color from this texture along with the shadow depth. This color is then used to modulate the sun color. Note that in this case we cannot skip long lit sections of the ray, which makes 1D min/max optimization less efficient.

6.5. Interpolating in-scattering

The next stage is interpolation of in-scattered radiance for the rest of the samples from ray marching ones. For this, indices of interpolation sources are loaded from the interpolation source texture and appropriate sample are loaded from initial in-scattering texture. This work is done by the following shader.

```
1  float3 InterpolateIrradiancePS(SScreenSizeQuadVSOutput In) : SV_Target { uint
   uiSliceInd, 0) ); float fInterpolationPos = float(uiSampleInd - ui2Interpolati
   uiSliceInd, 0) ); float3 f3Src1 = g_tex2DInitialInscatteredRadiance.Load( uint3(
```

6.6. Transforming epipolar in-scattering to rectangular coordinates

At this stage we have to transform inscattering image from epipolar coordinates back to the rectangular space. For this purpose, we first need to determine the two closest epipolar lines, project the pixel onto these and perform bilateral interpolation. Rendering is done with depth stencil state configured to increment stencil value. Note that different depth stencil buffer is used at this stage. The following steps are performed:

· The epipolar line which contains current pixel is identified

· Current sample is projected onto the line

· Bilateral interpolation is performed using interpolated in-scattering texture and epipolar camera space z

At the first step, we have to determine which epipolar line current pixel lies on. What we need to do is to connect light source projected position with one of four boundaries, and determine the slice entry and exit points. This is not as simple as it seems from the first glance because the light source could be outside the screen. Thus the obvious solution of determining the closest intersection with the boundary along the ray connecting light source and current pixel will not work if the light is not on the screen. A universal solution we implemented is based on determining to which of 4 sectors the pixel belongs to (fig. 19). If we know this, we will be immediately able to connect the light source with the appropriate border and get the exit point. After that we can compute entry point using `GetEpipolarLineEntryPoint()` function describe in section 6.1.

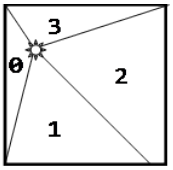

Fig.19: Four sectors of the screen formed by connecting light source with screen corners

To determine the sector, we first determine which of four half spaces formed by connecting each of the four screen corners with the light source the current pixel belongs to (fig 20).

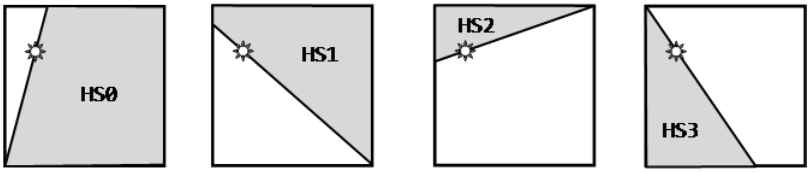

Fig.20: Four half-spaces formed by connecting each of the four screen corners with the light source

It is then easy to see that the following relations are always true for any pixel *P*:

- $P \in Sector0 \leftrightarrow P \notin HS0 \wedge P \in HS3$
- $P \in Sector1 \leftrightarrow P \notin HS1 \wedge P \in HS_0$
- $P \in Sector2 \leftrightarrow P \notin HS2 \wedge P \in HS1$
- $P \in Sector3 \leftrightarrow P \notin HS3 \wedge P \in HS2$

The following code snippet efficiently computes sector mask vector, which has 1 in the component corresponding to the required sector:

```
1  float2 f2RayDir = normalize( In.m_f2PosPS - g_LightAttribs.f4LightScreenPos.xy )
   f4HalfSpaceEquationTerms.zzww; bool4 b4SectorFlags = b4HalfSpaceFlags.wxyz && !b
```

Having this mask we can compute epipolar line exit point by computing distances to all four boundaries and then selecting the required distance using b4SectorFlags. After that and entry point can be computed with `GetEpipolarLineEntryPoint()`:

```
1  float4 f4DistToBoundaries = ( float4(-1,-1, 1,1) - g_LightAttribs.f4LightScreenP
   f4DistToBoundaries ); // Compute exit point on the boundary: float2 f2ExitPoint
```

Note that the method above works for any light source location either when it is outside and inside the screen.

Epipolar slice number which corresponds to ordering presented in section 6.1 can be computed as follows:

```
float4 f4EpipolarSlice = float4(0, 0.25, 0.5, 0.75) + (0.5 + float4(-0.5, +0.5,
```

Sample location on epipolar slice is computed by simply projecting the pixel onto the line connecting entry and exit points. This gives us coordinate f2ScatteredColorUV in the interpolated inscattering texture to filter from.

To perform bilateral interpolation we need to compute bilinear weights and locations of the source samples. The following code snippet computes bilinear filter weights as well as texture coordinates `f2ScatteredColorIJ` of the center of the left bottom source texel.

```
float2 f2ScatteredColorUVScaled = f2ScatteredColorUV.xy * f2ScatteredColorTexDir
- f2ScatteredColorIJ; // Get texture coordinates of the left bottom source texel
```

Camera space z values for 4 source samples can be obtained using `Gather()` intrinsic function. An important aspect here is offsetting coordinates to be at the same distance from all four source texels to eliminate rounding artifacts:

```
float4 f4SrcLocationsCamSpaceZ = g_tex2DEpipolarCamSpaceZ.Gather(samLinearClamp,
```

After that bilateral weights `f4BilateralWeights` are computed. To perform inscattering texture filtering using two fetches instead of four, we use the following trick: we can obtain weighted sum of two samples using hardware supported bilinear filtering. For this we only need to compute appropriate offset:

```
float fRow0UOffset = f4BilateralWeights.z / max(f4BilateralWeights.z + f4Bilater
f2LeftBottomSrcTexelUV + float2(fRow0UOffset, 0), 0, int2(0,0)); float fRow1UOff
f4BilateralWeights.y) * tex2DSrcTexture.SampleLevel(Sampler, f2LeftBottomSrcTexe
```

Note that presented implementation does not perform branching (except for discarding invalid pixel) and performs bilateral filtering of inscattering texture using just one gather and two bilinear fetches.

If total bilateral weight is close to zero, there are no appropriate samples which can be used to calculate inscattering for this sample. In this case we discard the sample, keeping 0 in the stencil.

6.7. Correcting in-scattering

For each pixel, which cannot be correctly interpolated from inscattering texture, we perform additional ray marching pass. Since these pixels are marked in stencil with 0, all we need to identify these pixels is to configure depth stencil state to pass only pixels whose stencil value equals 0 and discards all other pixels. Note that at this stage we cannot use 1D min/max mipmap, because it will require constructing this for each sample. We also use lower number of samples when performing this step with no visual impact.

6.8. Up-scaling in-scattering to original resolution

The final step of the algorithm is up-scaling the downscaled inscattering texture to original resolution and applying it to the attenuated background. This step is very similar to transforming the inscattering image from epipolar coordinates to rectangular with the main difference being is that original depth buffer is used to load camera space z and downscaled inscattering texture is used to load inscattering values. In the same manner pixels which cannot be correctly interpolated are marked in the stencil at this stage and finally corrected in the later pass. Note that we also apply phase function here, because it exhibits high variation near the epipole.

7. Sample structure

The sample project consists of the following files:

- LightScattering.h, LightScattering.cpp – Responsible for application startup/shutdown, rendering the scene and shadow map, handling user input
- LightSctrPostProcess.h, LightSctrPostProcess.cpp – Responsible for performing all the post processing steps to create light scattering effects
- RenderTechnique.h, RenderTechnique.cpp – Provide auxiliary functionality for creating effect technique
- Common.fxh – Contains common shader definitions
- LightScattering.fx – Contains all the shaders performing post processing steps
- RefineSampleLocations.fx – Contains compute shader performing sample refinement
- Structures.fxh – Contains definitions of structures used by the shaders

8. How to integrate the technique into your engine

Since the technique is completely post processing, integrating it into an existing engine is relatively simple. The technique is fully implemented in LightSctrPostProcess.h, LightSctrPostProcess.cpp and shader files Common.fxh, LightScattering.fx, RefineSampleLocations.fx and Structures.fxh. To compile the files you need DirectX SDK only.

All the work is done by `PerformPostProcessing()` method of `CLightSctrPostProcess` class, which takes two structs as arguments:

```
1 void PerformPostProcessing(SFrameAttribs &FrameAttribs, SPostProcessingAttribs &
```

The first structure defines attributes for the frame to be processed and has the following declaration:

```
1 struct SFrameAttribs { ID3D11Device *pd3dDevice; ID3D11DeviceContext *pd3dDevice
  ID3D11ShaderResourceView *ptex2DShadowMapSRV; ID3D11ShaderResourceView *ptex2DS
```

The structure has two nested structures which define light source parameters (such as light direction and color, light source position on the screen etc.). The second structure defines camera attributes (position, world, view, projection matrices). Destination render target to which post-processed scene is written is specified by the `pDstRTV` member of the structure.

The second argument `PPAttribs` of the `PerformPostProcessing()` defines the method parameters such as number of epipolar slices, number of samples, initial sample step etc.

9. Performance

Performance results are given for rendering the scene shown in the fig. 21

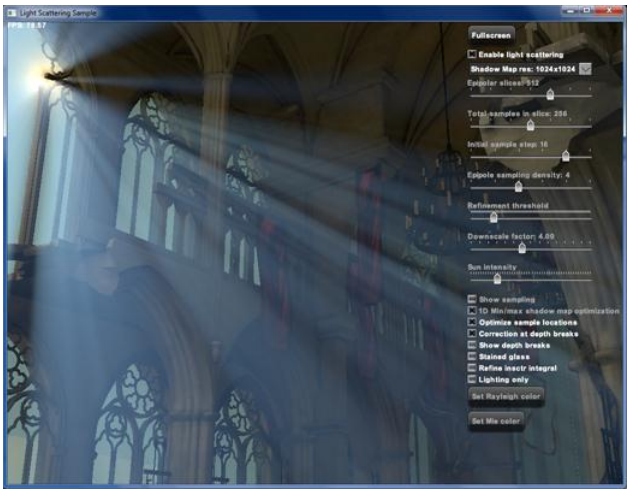

Fig.21: The scene used for performance test

The effect configuration is:

- Number of epipolar slices: 512
- Total number of slices: 256
- Initial sample step: 16
- Downscale factor: 4
- Shadow map resolution: 1024 x 1024
- Screen resolution 1024x768

Hardware configuration: 3rd Gen Intel® Core™ processor (code-named Ivy Bridge) with Intel® HD Graphics, 4 GB RAM.

Total time required to render the scene is 18.7 ms, of which 9.906 ms (52.9%) is spent on post processing.

The timings for the individual steps of the algorithm are as follows:

- Reconstructing camera space z coordinate: 0.390 ms (3.94 %)
- Rendering coordinates texture: 0.595 ms (6.01 %)

- Refining sample locations: 0.557 ms (5.62 %)
- Constructing min/max shadow map: **0.736 ms (7.43 %)**
- Ray marching: **2.638 ms (26.63 %)**
- Interpolation: 0.141 ms (1.42 %)
- Transformation to rectangular coords: 0.306 ms (3.09 %)
- Fixing depth discontinuities: 1.015 ms (10.25 %)
- Upscaling: 2.498 ms (25.22 %)
- Fixing depth discontinuities: 1.030 ms (10.40 %)
- Total: 9.906 ms (100 %)

Without 1D min/max mipmap optimization, ray marching step alone takes 5.651 ms which is **1.67x** slower than 2.638+0.736=3.374 ms for optimized version. Note that performance improvement is much higher for higher shadow map resolution, or higher number of epipolar slices/number of samples in slice.

Note that according to timings presented in [ED10] for ATI Radeon HD4850 GPU, epipolar line generation took 1.2 ms while discontinuity search took 6.0 ms which is 5 times slower. In our implementation discontinuity search takes even less time than coordinate texture generation thanks to optimization with compute shader.

Note also that Chen et al. report 55 fps for rendering the test scene on high-end NVidia GTX480 GPU [CBDJ11]. With similar quality settings (4096x4096 shadow map, 1024 epipolar slices with 1024 samples, 1x downscale factor, colored light shafts, 1280x960 screen resolution) for the scene of similar complexity (fig. 22) we were able to get more than 100 fps on the same graphics card and similar CPU.

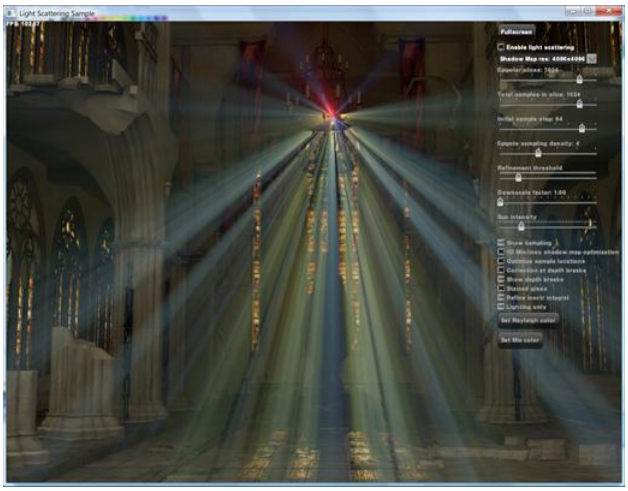

Fig.22: Colored light shafts in high-quality settings

10. Conclusion

The volumetric lighting method presented in the sample efficiently combines epipolar sampling with 1D min/max mipmap construction and a number of optimization tricks to achieve high rendering speed and visual quality on Intel integrated graphics. Being fully post-processing technique, it can be easily integrated into the game engines or other applications.

11. References

[PSS99] Preetham A.J., Shirley P., Smits B.E. (1999) A practical analytic model for daylight. In: Computer Graphics Proceedings, Annual Conference Series (Proc.SIGGRAPH '99), pp 91–100.

[NSTN93] Nishita T., Sirai T., Tadamura K., Nakamae E.: Display of the Earth taking into account atmospheric scattering. In SIGGRAPH 93 (1993), ACM, pp. 175–182.

[HP02] Hoffman N., Preetham A. J.: Rendering outdoor light scattering in real time. Proceedings of Game Developer Conference (2002).

[ED10] Engelhardt, T., and Dachsbacher, C. 2010. Epipolar sampling for shadows and crepuscular rays in participating media with single scattering. In Proc. 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, ACM, 119–125.

[BN08] Eric Bruneton and Fabrice Neyret.: Precomputed atmospheric scattering. Comput. Graph. Forum. Proceedings of the 19th Eurographics symposium on Rendering 2008 27:4 (2008), 1079-1086. Special issue.

[CBDJ11] Chen, J., Baran, I., Durand, F., and Jarosz, W. 2011. Real-time volumetric shadows using 1d min-max mipmaps. In Proceedings of the Symposium on Interactive 3D Graphics and Games, 39–46.

[GMF09] Gautrom, P., Marvie, J.-E., and Francois, G., 2009. Volumetric shadow mapping. ACM SIGGRAPH 2009 Sketches.

[TIS08] Tevs, A., Ihrke, I., and Seidel, H.-P. 2008. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In Symposium on Interactive 3D Graphics and Games (i3D'08), 183–190.

Categories: Game Development (/en-us/search/site/field_topic/game_development-20863/language/en) , Graphics (/en-us/search/site/field_topic/graphics-20864/language/en) , Microsoft Windows* 8 Desktop (/en-us/search/site/field_topic/microsoft_windows_8_desktop-36923/language/en) , Game Development (/en-us/search/site/field_platform/game_development-78391/language/en) , Developers (/en-us/search/site/field_audience/developers-17152/language/en) , Microsoft Windows* 8.x (/en-us/search/site/field_operating_system/microsoft_windows_8x-20786/language/en) , Advanced (/en-us/search/site/field_skill_level/advanced-20809/language/en)

Tags: ivy bridge (/en-us/search/site/field_tags/ivy_bridge-18023/language/en)

Comments (1)

Top

Anonymous said on Fri, 08/10/2012 - 03:36

It would be just fair to mention in this technical article that it describes what has been published in a paper at ACM SIGGRAPH I3D 2010 called "Epipolar Sampling for Shadows and Crepuscular Rays in Participating Media with Single Scattering": http://graphics.cs.williams.edu/i3d10/papers.htm

Add a Comment

Top

(For technical discussions visit our developer forums. For site or software product issues contact support.)

Please sign in to add a comment. Not a member? Join today ›