# White Paper

## Volume Light

January 2008
WP-03017-001_v01

# Document Change History

| Version | Date | Responsible | Reason for Change |
|---|---|---|---|
| | January 15, 2008 | EM, TS | Initial release |
| | | | |
| | | | |
| | | | |

Go to sdkfeedback@nvidia.com to provide feedback on Volume Light.

# Volume Light

## Abstract

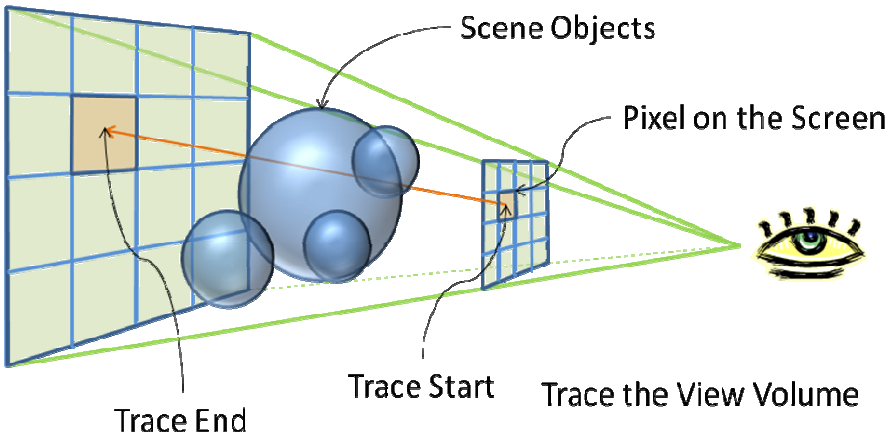
*Volume Light* technique can be considered a simple approximation of real world light scattering effect. Small particles, sprayed in the air, interact with light beams and produce different effects like rainbows or sun shafts. Volumetric effects, are traditionally considered to be computational heavy, but can be approximated in real-time. In most cases a convincing look can be achieved by a good looking effect that integrates seamlessly into the scene. Thus, volumetric light is going to be a visual effect, rather than a physically correct simulation.


**NVIDIA Corporation**
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

# How Volume Light Works

The ultimate goal of our technique is to produce the volumetric look of real light scattering.

We use a light space shadow map and depth buffer as inputs for a Volume Light technique. Starting from the near clip plane, we trace the whole scene and accumulate sampling values. For each sample we determine if it is lit by the source light or not (based on the shadow map values comparison). Note, that only the lit samples give a final impact to the resulting pixel color value.



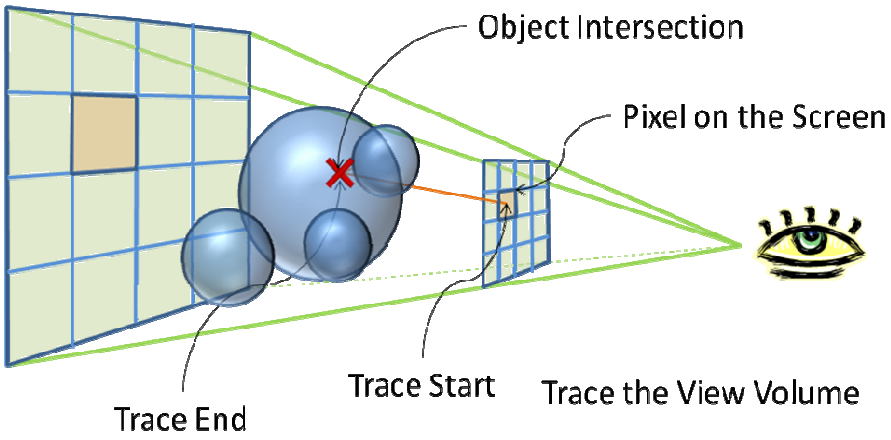

Actually, we need to perform tracing only up to the first intersection with a scene object. To find those, we use the scene depth buffer.

Drawing a fullscreen quad, we can reconstruct world space position for each fragment, using the following code:

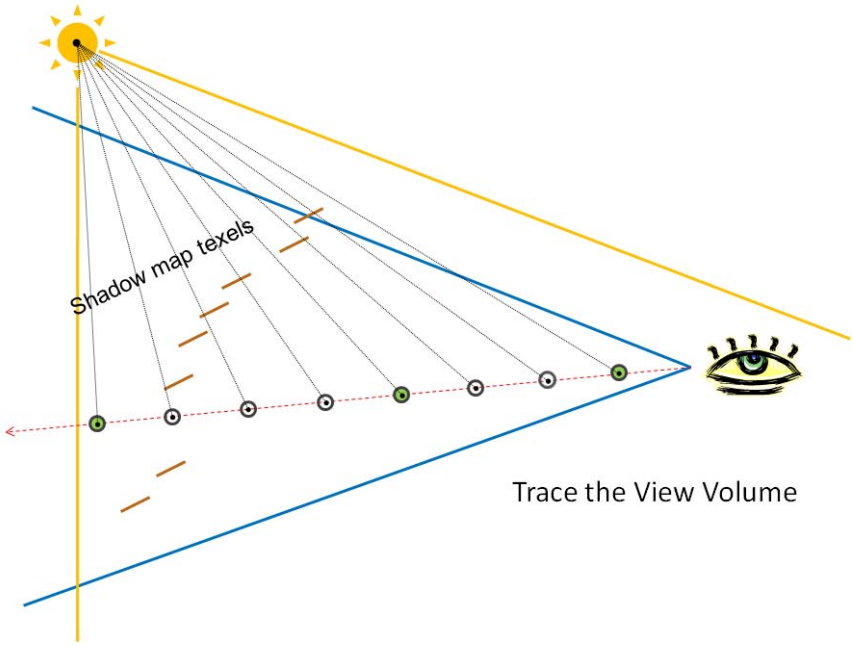
```
float sceneDepth = DepthBufferTexture.Sample( samplerPoint,
input.texCoord.xy );

float4 clipPos;
clipPos.x = 2.0 * input.texCoord.x - 1.0;
clipPos.y = -2.0 * input.texCoord.y + 1.0;
clipPos.z = sceneDepth;
clipPos.w = 1.0;

float4 positionWS = mul( clipPos, g_MWorldViewProjectionInv );
positionWS.w = 1.0 / positionWS.w;
positionWS.xyz *= positionWS.w;
```

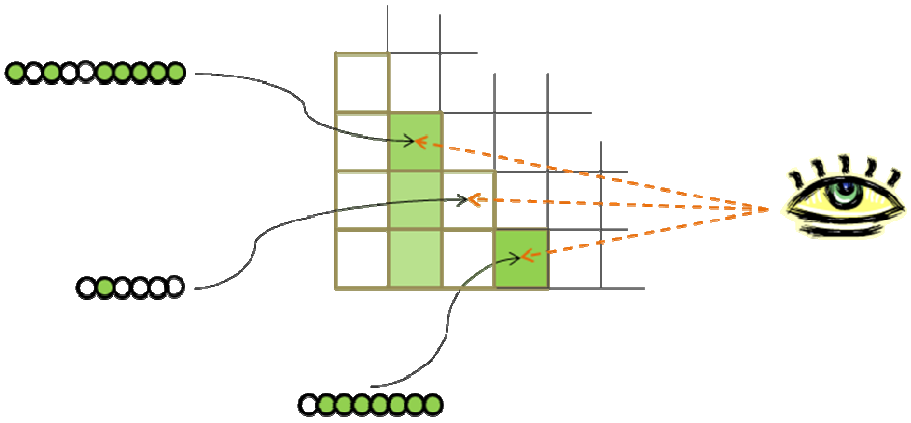



Tracing the scene, we accumulate lit samples in the buffer, based on the shadow map comparison




results.

According to the number of lit samples, we calculate final intensity of screen pixels. With this, we make the light shafts visible.



The basic scheme works, however we need to introduce some changes, to make it look better in terms of visual quality and performance.

The first optimization step can be rendering the whole volume light texture at a coarser resolution. The volume light shafts do not have a lot of internal contrast by their nature, due to multiple scattering events. Thus, practically, we can downsample the buffer and still achieve good quality results. Practically, the buffer can be downsampled by a factor of four for each side. Using higher values can result in too big loss in details and visible aliasing.

# Scene Integration

When we want to combine a downsampled variant of volume light texture with the main scene, we face the aliasing problem. The edges in the coarse texture do not match those in the original scene, thus we need some preprocessing here.


*Original upsampled Volume Light texture*

The areas with the highest intensity variations basically appear on the scene edges, where we have depth discontinuities. We can use this fact and try to make those look better.

We use a Sobel filter for edge detection, applied to the original depth buffer, to get as much detail as possible. We blur the upsampled volume light texture image along the edges to reduce aliasing.

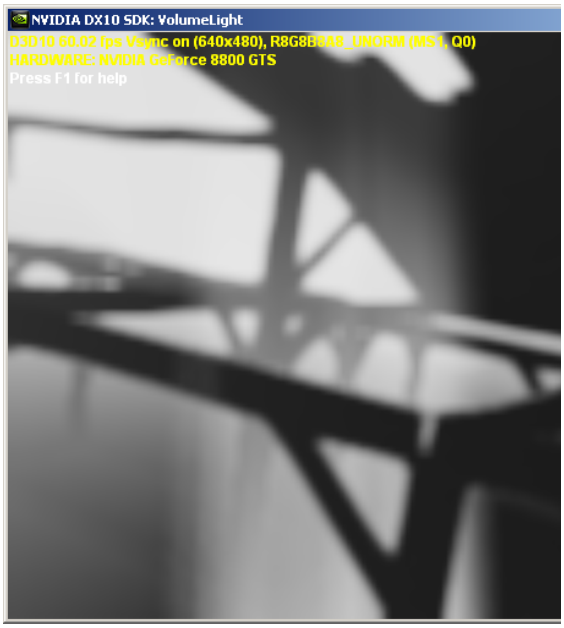
| | | | | |
|---|---|---|---|---|
| 1 | -2 | 0 | 2 | 1 |
| -2 | -3 | 0 | 3 | 2 |
| -3 | -5 | 0 | 5 | 3 |
| -2 | -3 | 0 | 3 | 2 |
| 1 | -2 | 0 | 2 | 1 |

Gx

Gy

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 2 | 1 |
| 2 | 3 | 5 | 3 | 2 |
| 0 | 0 | 0 | 0 | 0 |
| -2 | -3 | -5 | -3 | -2 |
| -1 | -2 | -3 | -2 | -1 |

*5x5 filter kernels. Edge direction is defined as float2( Gx, Gy ).*

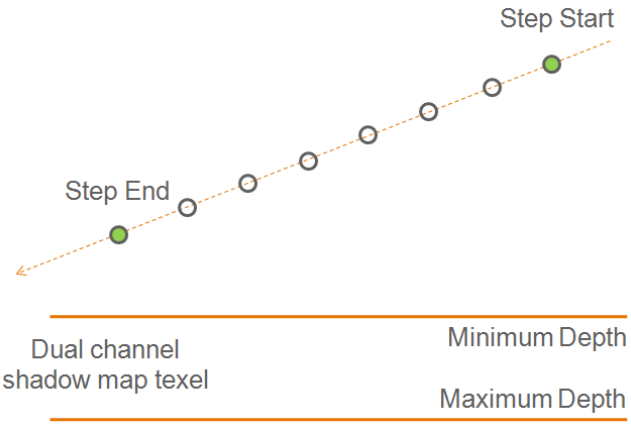



*Upsampled texture with edge blurring applied*

# Optimized Tracing

In order to make scene tracing faster, we use variable sampling step sizes, based on the shadow map hierarchy. For the initial map we build a two component MIP-level chain. For each quad of pixels we calculate a minimum and maximum depth values and store those in a separate texture. Using a coarse MIP-level, we can try to predict if all samples referencing the same texel are lit or shadowed.

If several samples are referencing the same texel, we can calculate the maximum light space depth of those and compare to the minimum value stored in two-component texture. If the reference value is smaller than those in the texture, all samples are lit. Thus, we can trivially check several samples at one. The same logic can be used to determine if all the samples are in shadow.




*If we perform a coarse step and both start and end points are lower than a minimum depth value, than all intermediate fine sample values are lit as well.*

We can determine if we can do a coarse step, using the following simple code

```
stepEndZ = stepStartZ + deltaZ * longStepScale;

float comparison = max(stepStartZ, stepEndZ );
float isLight = comparison < sampleDepthMin;
comparison = min(stepStartZ, stepEndZ );
float isShadow = comparison > sampleDepthMax;


float isLongStep = isLight + isShadow;
```

# Visual Quality

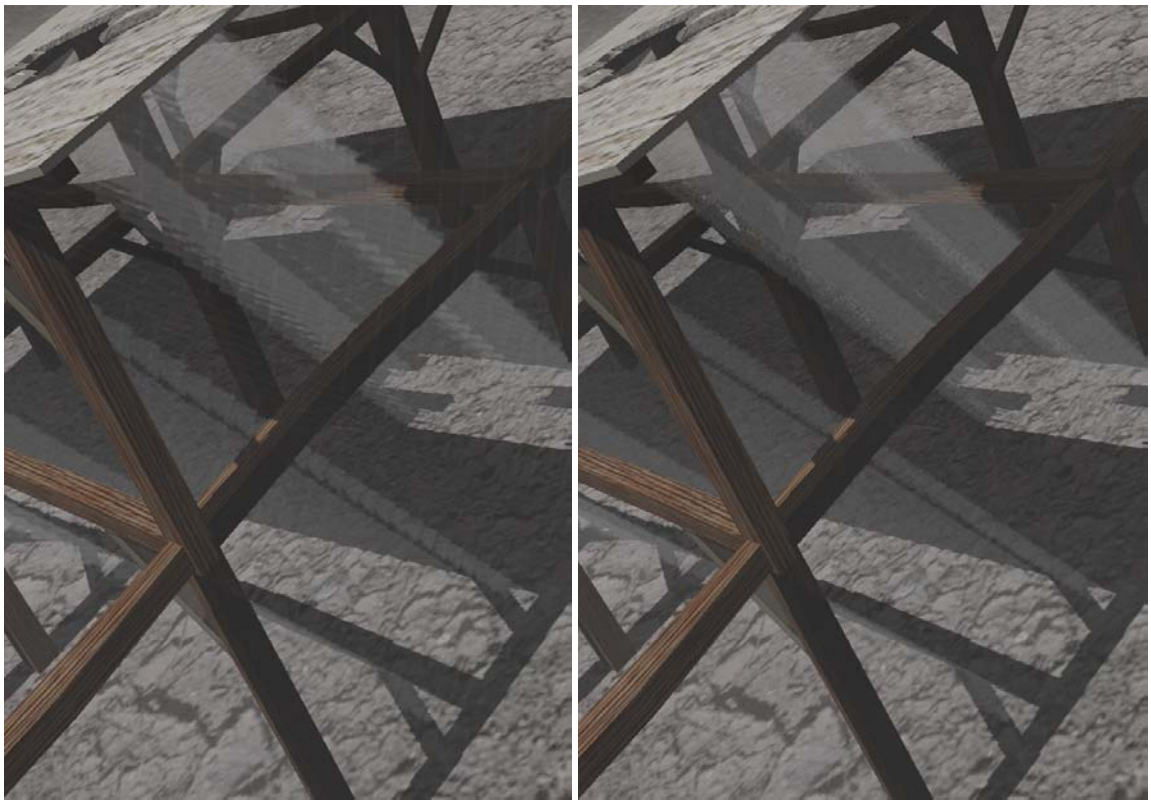
The current approach assumes, that we have uniform light scattering inside the volume. But this is not true in real life. The particles distribution is not uniform, plus multiple scattering decrease the light intensity along the light distribution vector.

In order to make the effect look better, we can try to increase the light intensity, where the effect is most visible and decrease those for open spaces to avoid a global foggy look.

We search for discontinuities in light-space shadow map buffer and try to locate holes in it. To detect the hole, we use a number of passes. First, we use several draw calls for minimum depth values propagation, from the boundaries, thus if the hole is relatively small it gets finally filled. The second step uses several passes to propagate maximum depth values from the new boundaries. If the hole was not "filled" completely, it almost returns to initial state. Looking at newly occluded areas, we treat those as holes in the original shadow map and increase the intensity for those.

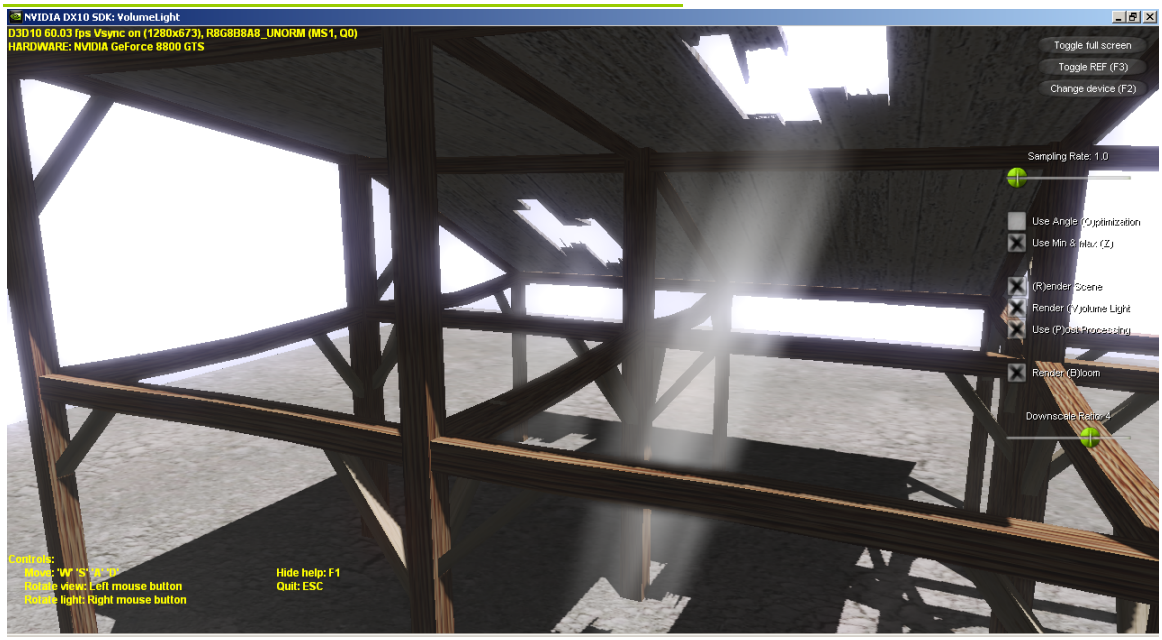
# Banding Artifacts

Regular sampling can produce visual banding artifacts. To avoid those, we use jittering, slightly shifting initial sampling positions in eye-space depth. The actual amount of those can be precomputed and stored in the noise texture.



*Regular sampling(left) and the same scene with a jittering applied(right).*

Note, that only the initial position is being shifted, thus, there are no performance penalties for all other sampling points being used.


NVIDIA DX10 SDK: VolumeLight
D3D10 60.03 fps Vsync on (1280x673), R8G8B8A8_UNORM (MS1, Q0)
HARDWARE: NVIDIA GeForce 8800 GTS
Toggle full screen
Toggle REF (F3)
Change device (F2)
Sampling Rate: 1.0
Use Angle (O)ptimization
Use Min & Max (Z)
(R)ender Scene
Render (V)olume Light
Use (P)ost Processing
Render (B)loom
Downscale Ratio: 4
Controls:
Hide help: F1
Quit: ESC
Rotate view: Left mouse button
Rotate light: Right mouse button