Enhancing the looks of Source™ Engine in **Military Conflict: Vietnam**

Dmitriy Andreev

Dmitriy Andreev

Feb. 2021, edit 3

Bachelor’s degree in Applied informatics and Information Security

Abstract

Military Conflict: Vietnam is a multiplayer first-person shooter game running on **Valve's Source™ 1 Engine**. Engine was chosen due to high amount of resources, simplicity of testing gameplay mechanics, building prototypes and because of large community around Source™ games. Still, engine was lacking many of modern visual techniques and effects and my task was to make the game look up-to-date without any fundamental changes to renderer and keep the frame rate as high as possible.

This paper will cover shaders written for *Shader Model 3.0* and running under DX9 API. The techniques covered will involve HDR Rendering, lighting and shadowing, procedural sky, my attempts to optimize and re-use the same effects in multiple scenes, and simple post-process effects to give little touches to final look of the game. Keep in mind that modern engines already have all those features and do them at way more advanced level, everything written here only relevant to Source™ or similar DX9 engines.

I would appreciate any feedback at *dmitrex@dmitrex.com*

Acknowledgements

I would like to mention that this work has been made possible thanks to many people – especially from *gamedev.net* community who have shared many useful articles and code samples.

I would also like to thank CEO of *DustFade Studio*, Kai Sidler, who has provided the opportunity to do this work in first place.

Big thanks to *Valve Software* for answering engine-related questions and even providing me code samples and fixes.

I would also like to thank my friends that have given corrections to the paper and provided me with useful feedback.

Contents

Glossary

MCV – short for Military Conflict: Vietnam

DX9 – short for DirectX™ 9

Shader Model - a set of profiles found in a generation of GPUs.

Frame-rate - how many frames (or images) per second are displayed on screen.

2D - short for 2-Dimensional.

3D - short for 3-Dimensional.

HDR – short for High Dynamic Range

LDR – short for Low Dynamic Range

Polygons - any 2D shape defined by three or more straight lines (triangle, square, pentagon, ...). Used extensively in computer-graphics to define and render shapes.

FP16 – floating point 16bit image format.

Render-pass - to create a fully rendered scene, the rendering is usually separated into multiple renderpasses, where each render-pass can perform separate tasks. For example, when creating shadows, there are usually at least two render-passes, one that from the light-source's perspective calculates and stores the depth into a depth-map rendertarget, and one that uses the depth-map to apply the shadows when rendering objects to the screen.

Render-target - a run-time image used to store rendered information. An example of a render-targets is the shadow-map, a render-target that usually contains the depth values for objects in relation of the light's direction.

Render-time - the amount of time that it takes to draw the image to the screen, measured in milliseconds for the entire thesis.

RGBA - red, green, blue and alpha. Frequently used to store pixel data for images, where different combinations of these colors can form any other color. However, in this thesis, these color channels are often used individually to store some attribute.

Screen-space - space used to specify where an object is in relation to the screen. Therefore, it can be used to specify where an object is on the screen.

Viewing-ray - the ray that can be seen as a combination of the cameras direction and the pixel that is currently being rendered.

Ray marching - is a ray intersection test in which you step along a until an intersection is found.

Volumetric clouds - a way of representing clouds in 3D. The part "volumetric" does not specify a specific method, just that the clouds have a volume with some density.

Procedural generation – is a method of creating data algorithmically as opposed to manually, typically through a combination of human-generated assets.

FBM - fractional Brownian motion, fractal noise, but with some sort of memory to the process of generation.

Draw Call - is the command that tells the GPU to render a certain set of vertices as triangles with a certain state.

1 Introduction

Military Conflict: Vietnam (or MCV) – is a *multiplayer* game. This is important clarification, because working processes are more difficult, algorithms and performance targets are quite different from similar *singleplayer* projects. During development of the game, we have determined main directions which we want to go with graphics improvements and performance goals. There are a lot of limitations and performance issues can be met under DX9 API, yet, plenty of work still can be done:

- One of most annoying problems is outdated implementation of HDR and bloom. Source™ supports textures which can emit light, but they don't *look* like they do it, in other words, do not glow – level designers have to draw custom "glow" textures for each prop/brush/particle which emits light and apply it on top of them. This usually cause problems like glow sprite's shape do not match light source's shape or sprite popping through walls.
- Sky. Skyboxes used in Source™ - is very old tech and have various drawbacks. One of them is corners – player can still see sky's corners even if they match perfectly. The other is that the sky is completely static – even if you decide to use scrolling cloud textures you would better make a sphere, representing your sky, and put it inside 3D skybox, otherwise previously mentioned problem will pop in.
- Lighting. Lighting in Source™ has improved a lot since initial Half-Life 2™ release in 2004, though, it was lacking dynamic effects, such as dynamic ambient occlusion and volumetric light. Adding these effects would give level designers more freedom in expressing their ideas, make lighting more detailed and overall look of the game fresher.
- Texture detail can be improved using parallax mapping. This is very old technique, but only a few implementations can live in real game, not only in dev levels. We have decided to look towards *Interval Mapping*, an expansion to *Relief Mapping*, as fastest and most detailed algorithm possible in DX9.
- Post-effects play big role in final look of the game as well. They can make final image look sharper, detailed, or add fancy screen-space effects like lens flares and contusion after explosion. Post-effects play their role in gameplay as well – various effects may appear on screen when player takes damage, low on health or inside smoke/gas cloud.

Apart from new effects themselves, there is performance target which must be met. Game supposed to work at *1920x1080@60fps* with maximum settings on 2016's middle-end hardware. Taking in account that we're working under DX9 API this makes things even harder and cuts on number of rendering techniques or shaders we can implement. Though, we have to find ways to optimize/combine several major techniques and make them work inside Source™ engine fast enough.

2 Floating Point HDR and Bloom

HDR Rendering is a feature of the Source™ Engine since *Half-Life 2™: Lost Coast* release (Accardo, 2005). However, *Valve* has preferred their own Integer HDR implementation over floating-point one as the fastest and most supported by all hardware at that time (McTaggart, 2006). Despite all the cons it had, the scene was still rendered in LDR and bloom existed only in bright parts of skybox or from *very* bright light sources (ex. Light blasting through the window in dark room). After a decade since then, software and hardware have improved a lot so I decided to give Floating Point HDR second chance.

2.1 Floating Point HDR

The principle behind Floating Point HDR remains unchanged since then. Moreover, big part of the code already existed in engine, so I mostly had to just enable it.

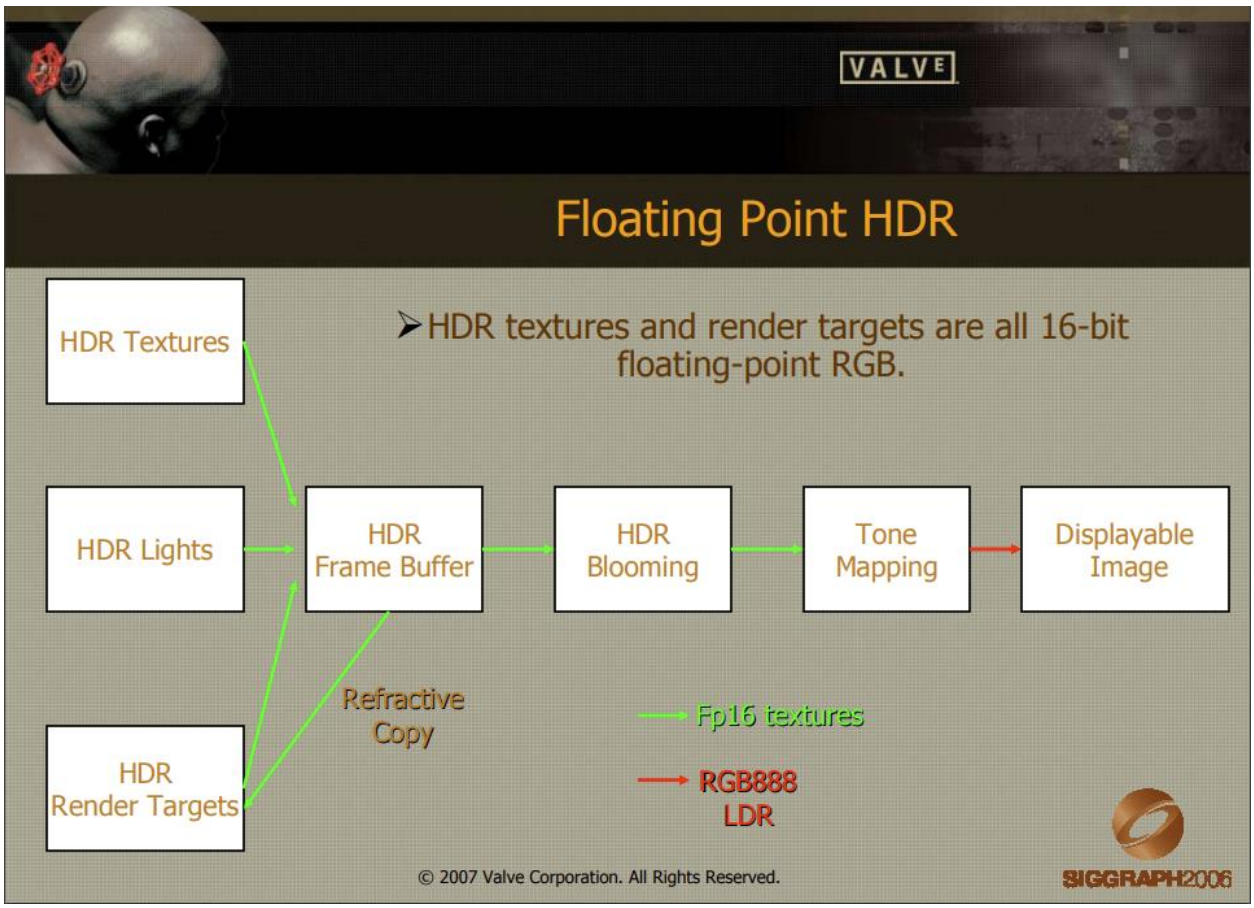


Figure 1: *HDR pipeline graph from "*HDR in Valve's Source Engine" (McTaggart, 2006)

Frame buffer and bloom buffers are now using FP16 format, image is tonemapped back to LDR using *Filmic Tonemapping.* MSAA quality got worse, which is DX9 API limitation, but it's not a problem for us since screen resolution of majority of users increased a lot over decade, plus, an additional *SMAA* pass is performed after.

The following pictures show difference between Valve's original HDR (figure 2) and Floating Point HDR in MCV (figure 3), which is based off Valve's leftover code. Note that we are not using HDR skyboxes anymore – our sky is completely procedural.

Figure 2*: This is how Valve's HDR looks in MCV – colors are vanished, bright areas are just pure white. Level designers have to apply custom-shaped glow sprites to all objects in scene which are supposed to "glow".*

Figure 3*: Floating Point HDR – More color variations, bloom is now taken in account and makes bright areas stand out and glow. As well other effects like Lens Flare can now utilize HDR bloom buffer to detect bright pixels and generate flares from them. MSAA quality difference is negligible.*

2.2 Five-Pass Bloom

Bloom pass has undergone changes too, because switching to FP16 HDR opened a couple of new opportunities. Bloom is a real-life lens artifact and used in games to indicate that an object is emitting light. However, the implementation of bloom may differ from engine to engine (or from game to game). In Source™ bloom is performed in LDR space and applied to every pixel which has luminosity value close to 1.0. Bloom appears if tonemapping, or, in other words, eye adaptation scale is higher than 1.0 and when tonemapping scale is near or lower than 1.0 – bloom disappears completely even from objects which are still should be glowing. Here is an example (figure 4):

Figure 4: *Lamp is emitting light, but is not "glowing" unless glow sprite is added*

This lamp is supposed to glow, but *Valve's* HDR still works in LDR [0..1] color space so there is no way to determine bright pixels. *Valve* "fakes" bloom on such objects by using high tonemapping (eye adaptation) scale, but since the room is illuminated enough – the eye adaptation scale is low and hence the lamp is not glowing as well. The only way to make this lamp constantly glow is to add a pre-drawn sprite with "glow" texture.

In MCV we prefer different option – bloom blends with HDR image "as-is" without any tonemapping scales and thresholds, the amount of blending is calculated from luminosity of pixel from original HDR image. There is also an artist-driven parameter which can modify blend weight and make bloom appear only at *very* bright pixels (or vice versa: make whole image glow – depends on artist's wish). This way glowing objects in scene will always remain glowing, no matter what current tonemapping scale is.

Figure 5: *This is the result of Floating Point HDR + New Bloom working together. Pixels with high luminosity values make bloom contribute to final image, while the rest of the scene remained as-is. In the end – lamp glows by itself, no need to use "glow" sprites anymore.*

Figure 6: *See how illuminated part of barrel glows on sun – you will never get this result by using sprites from 1998.*

The rest of the bloom is similar to the one from *Unreal Engine 4* (Epic Games, 2014). There are 5 passes at each of them scene is downscaled to 1/2 of image size from previous pass and blurred using wider gaussian kernels. So, 1st pass bloom has size 1/2 of user's screen resolution and very small blur kernel and then 5th pass has size 1/32 of screen resolution and very wide kernel.

While the result was very nice, it introduced a new problem - bloom from very bright sources was aliased (figure 7):

Figure 7: *Aliasing bloom artifact*

To fix this, I introduced another, 6th pass, where I combine all previous 5 passes, apply another small gaussian blur and bilinearly upsample the result back to full-size. Artifacts are not fully gone, but now never visible 99% of the time.

After main work with bloom was finished, I decided to add a little touch to the sun and wrote simple starburst shader (figure 8):

Figure 8: *Starburst. Original Shadertoy: https://www.shadertoy.com/view/tlfyRS*

Rays length depends on player's viewing angle and how clearly sun can be seen through the sky. Since our sky is procedural, I had to cache the clouds map and use it as a mask for starburst effect.

2.3 Filmic Tonemapping

At the end of HDR rendering pipeline tone mapping is usually performed to convert HDR image to resulting LDR image. This is needed, because majority of user's monitors do not support HDR displaying yet. There are many tonemapping algorithms that can be found around on the Internet (HABLE, 2010). I preferred the "Filmic" ones, especially *ACES*, for their "photorealistic" result.

In Source™ tonemapping (or eye adaptation) performs at the end of rendering of each object in scene, but since we've moved to Floating Point HDR and want to have HDR Bloom, we had to defer tonemapping process and do it as post-effect at the end of rendering pipeline after bloom pass. This also allowed us to add various effects to pre-tonemapped and post-tonemapped images. For example: if you implement any bloom-based *Lens Flare* effect to a game with old HDR pipeline then you will notice that brightness of flares will change along with tonemapping scale, which is incorrect and there is no optimal solution to this problem. Now, in MCV we have pre-tonemapping stage and post-tonemapping stage – we simply apply *Lens Flare* before tonemapping and effect stays the same no matter of what current tonemapping (or eye adaptation) scale is.

To be more precise, I've preferred *ACES Filmic Tone Mapping* operator (Narkowicz, 2016). After multiple tests it was chosen for providing the most "natural" colors:

Figure 9: *Default look without Filmic Tonemapping*

Figure 10: *Filmic Tone mapping enabled – green is now greener, black is now darker, overall colors look more alive, removed "foggy" look of the scene.*

Of course, level designers should keep in mind that lighting must be reconfigured according to new algorithm, otherwise some scenes may look incorrect.

2.4 Fireflies

Due to image undersampling, the new artifact appears, called "Firflies" (Jimenez, 2014), (figure 11). To reduce it, we apply Karis's luma weighted average to first pass of bloom (while downsampling from full to half-res). It reduces high spikes of brightness for single pixels.

Figure 11: *Random firefly appeared out of nowhere*

3 SMAA

SMAA is a post-processing Subpixel Morphological Antialiasing algorithm. In MCV it's aimed to give additional aliasing to situations where hardware *MSAA* is not working. Source code and description can be found here: https://github.com/iryoku/smaa

3.1 SMAA in Source™

MSAA is good hardware anti-aliasing algorithm, but switching to HDR pipeline made an impact on its quality. Moreover, it doesn't work with translucent objects. and off-screen buffers in DX9, which is a problem especially since MCV is using many aspects from *Deferred Rendering* pipeline. To solve this, I've decided to try various post-processing anti-aliasing algorithms. *FXAA* was not the option since it makes the image very blurry. It can be negated by using Unsharp Masking, though, but still end result was not as good as with *SMAA*.

For MCV, I went with simplest version – SMAA 1x. T2x is possible to implement in DX9 as well, but it requires generating of per-object velocity buffer for each pixel on screen. This is not hard task, especially when you have a working *GBuffer*, but you need to cache all matrices for every bone of every model in scene from previous frame. This requires big amount of vertex shader registers: for a skinned object engine reserves *156* float4 registers, plus another *156* you would need as data from previous frame. Add to this lots of other data like lighting, flex weights, model/view projection matrices etc. and you will exceed vertex shader constant register limit very easily. With that in mind, I've decided to cut velocity buffer and use *SMAA 1x* only. But on the good side, SMAA predication and stencil optimizations are still there and the shader can be combined with hardware MSAA to achieve better results. For example, MSAA 4x + SMAA 1x could save a few milliseconds of rendering time (Table 1), but remain same or better AA quality.

Images below (figure 12 and figure 13) represent the difference when SMAA is applied and not, both have MSAA 8x enabled:

Figure 12: *MSAA 8x, SMAA disabled, jagged edges can be seen on palm leaves*

Figure 13: *MSAA 8x, SMAA 1x. Jagged edges are noticeably smoothed out*

While implementing this shader to Source™, I faced one interesting problem - SMAA uses two helper textures: area texture and search texture. While search texture is fine, the area texture has unusual for Source™ Engine size: *160x560px*. There is no way for engine to read such texture easily, so I had to convert it to *256x1024* by adding black areas around source image (figure 14).

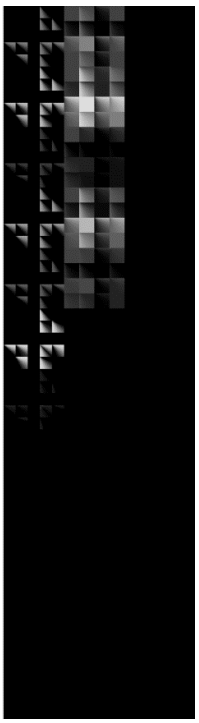


Figure 14: *SMAA area texture hack*

I also had to change the offset value inside "float2 SMAAAreaDiag(…)" from *0.5* to *0.3125* due to area texture size change. This shouldn't affect anti-aliasing since shader just searches *20x20px* areas no matter what size the area texture has itself.

Here are what image formats were used for various SMAA render targets:

AreaTex *- AL88*
SearchTex *- L8*
EdgeDetection *- BRGA8888*
BlendWeights *- BRGA8888*

Also note that shader compilators in earlier engine versions (before *L4D2*) will fail to compile SMAA – you have to use compilators from *Alien Swarm* codebase at least.

3.2 Unsharp Mask

SMAA blurs image as well, but not as much as FXAA. Still, I added an artist-driven unsharp mask shader to neutralize blurring + gain more sharp details even when SMAA is turned off. Unsharp mask works simply by blurring an image and then adding difference between blurred and original results with given weight back to original image.

```
float3 blur = tex2D( blurredFBSampler, texuv ).rgb;
color = float4( color.rgb + ( color.rgb - blur ) * g_Strength, color.a );
```

Listing 1: *Unsharp masking example*

Figure 15: *No unsharp masking*

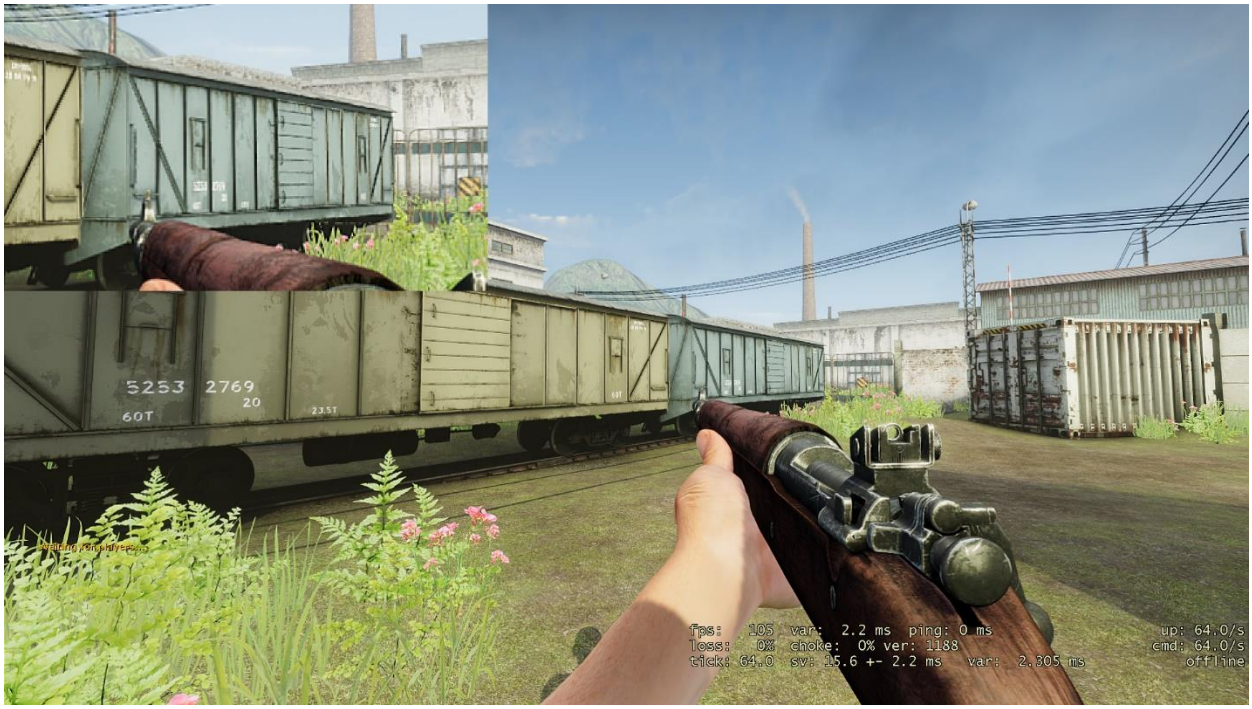


Figure 16: *Unsharp masking applied*

4 Procedural sky

Since Day One of MCV development there was a goal to make sky more alive. I went through many approaches: billboards, scrolling textures, sky domes and decided to go with the most challenging, but promising approach – **volumetric clouds**.

Procedural sky in MCV is a group of very complex **post-process** shaders consisting of multiple passes while some of them use Temporal Reprojection technique. At first, we calculate sky color from given sun position using Hosek-Wilkie model (Wilkie & Hosek, 2013), then low-res cloud buffer is generated from 3D textures, then final combine pass.

Figure 17: *Procedural sky in action*

Figure 18: *Procedural sky in action*

Figure 19: *Aggressive sky settings*

Live demonstration - *https://www.youtube.com/watch?v=Ot3oSOKH5qA*

4.1 Hosek-Wilkie Model

The first shader in group and the base for procedural sky is a sky color model which can generate day/night skies according to given sun position, turbidity and Earth's reflectance (albedo). Hosek & Wilkie's analytical model was chosen as most realistic and up-to-date by the time. Detailed info about their model can be found here:

https://cgg.mff.cuni.cz/projects/SkylightModelling/

Turbidity value (from 1 to 10) defines how much liquid particles are floating in the sky, 1 - represents completely clear arctic-like sky, 10 - rainy/stormy sky.

Albedo (0.0-1.0) is how much light reflects from Earth's surface. Can brighten or darken the sky.

But before performing any calculations, we need to setup a camera direction vector first, it's a bit tricky as we are working in post-process (listing 2).

```
// reconstruct using inverse projection multiplied by inversed rotation part of view matrix
float4 cameraRay = float4( i.vTexCoord * 2.0f - 1.0f, 1.0f, 1.0f );

cameraRay = mul( matInvProjViewrot, cameraRay );
o.vCamDir = cameraRay.xyz / cameraRay.w;

#if REFLECT_VIEW_RAY
	o.vCamDir.z = -o.vCamDir.z;
#endif
```

Listing 2: *View direction reconstruction in post-process vertex shader*

Don't forget to normalize it in pixel shader. "*REFLECT_VIEW_RAY*" block is needed if you want the sky to appear on reflective surfaces (ex. water).

And in shader DLL matrices are computed like this (listing 3):

```
VMatrix matView, matViewInverse, matProj, matProjInverse, matResult;
pShaderAPI->GetMatrix( MATERIAL_MATRIX_VIEW, matView.Base() );
pShaderAPI->GetMatrix( MATERIAL_MATRIX_PROJ, matProj.Base() );

// prevent image from stretching when camera moves too far from coord center
matView.m[3][0] = 0.0f;
matView.m[3][1] = 0.0f;
matView.m[3][2] = 0.0f;

MatrixInverseGeneral( matProj, matProjInverse );
MatrixInverseGeneral( matView, matViewInverse );
MatrixMultiply( matProjInverse, matViewInverse, matResult );
pShaderAPI->SetVertexShaderConstant( VERTEX_SHADER_SHADER_SPECIFIC_CONST_0,
matResult.Base(), 4 );
```

Listing 3: *Calculate and load inverse view-projection matrix to shader*

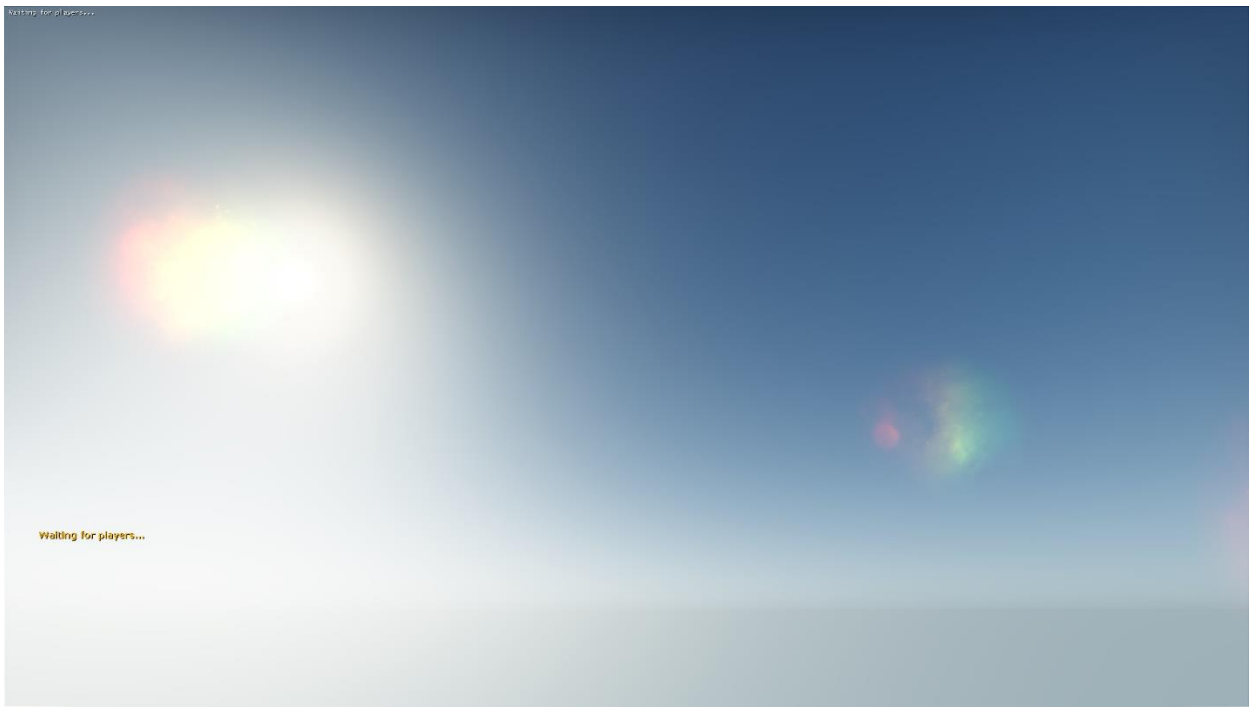


Figure 20: *Turbidity 3 example*

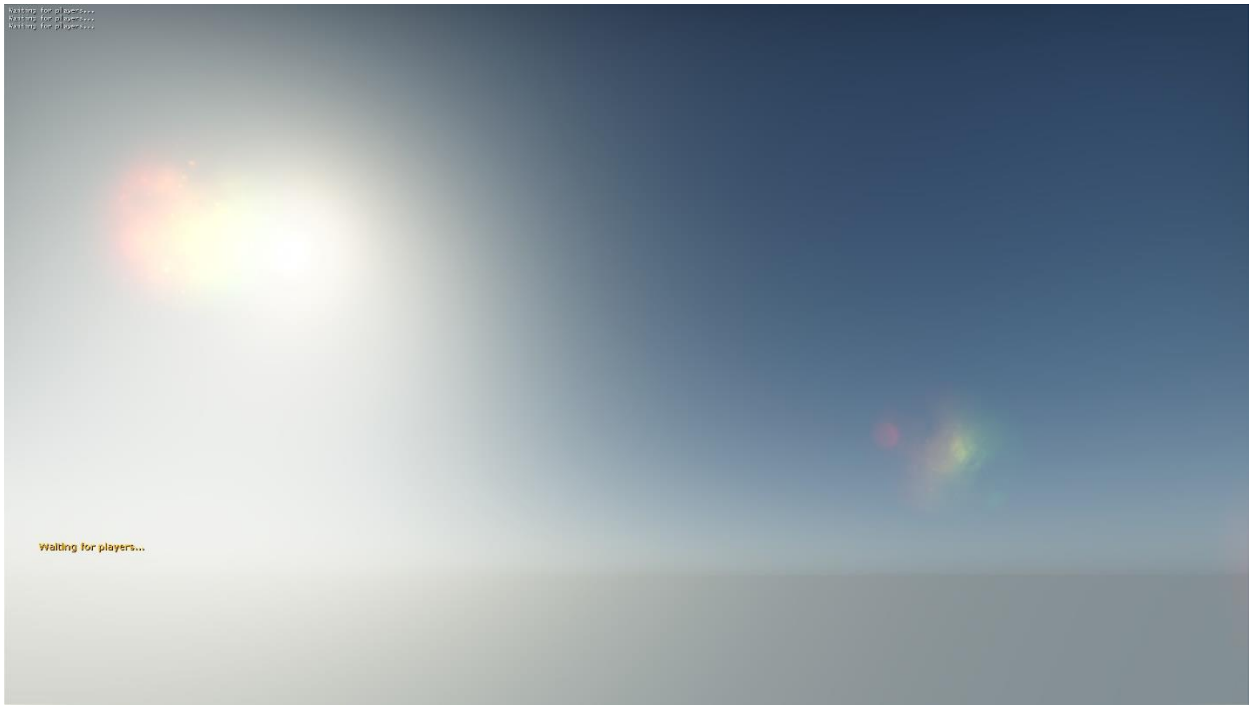


Figure 21: *Turbidity 5 example*

Figure 22: *Turbidity 10 example*

4.2 Sun

After skylight model is working, it's important to simulate a realistic looking sun. *DICE* has very interesting approach in simulating Sun and Moon (Hillaire, 2016), but I went with simpler approach in order to save on rendering time. Sun color is computed using Preetham's sun radiance table according to given turbidity and angle above (Preetham, Shirley, & Smits, 1999). The limb darkening code was taken from *DICE's* paper, Listing 7.

Figure 23: *Sunset*

Figure 24: *Sunset with different turbidity value*

4.3 Night

There wasn't any goal of making real-time day-night cycle, but night maps were still in plans, so the next step was to create the night sky.

Night sky doesn't use any complex lighting model, it's just a spherical texture mapped on a screen-space quad and a bluish light added near horizon.

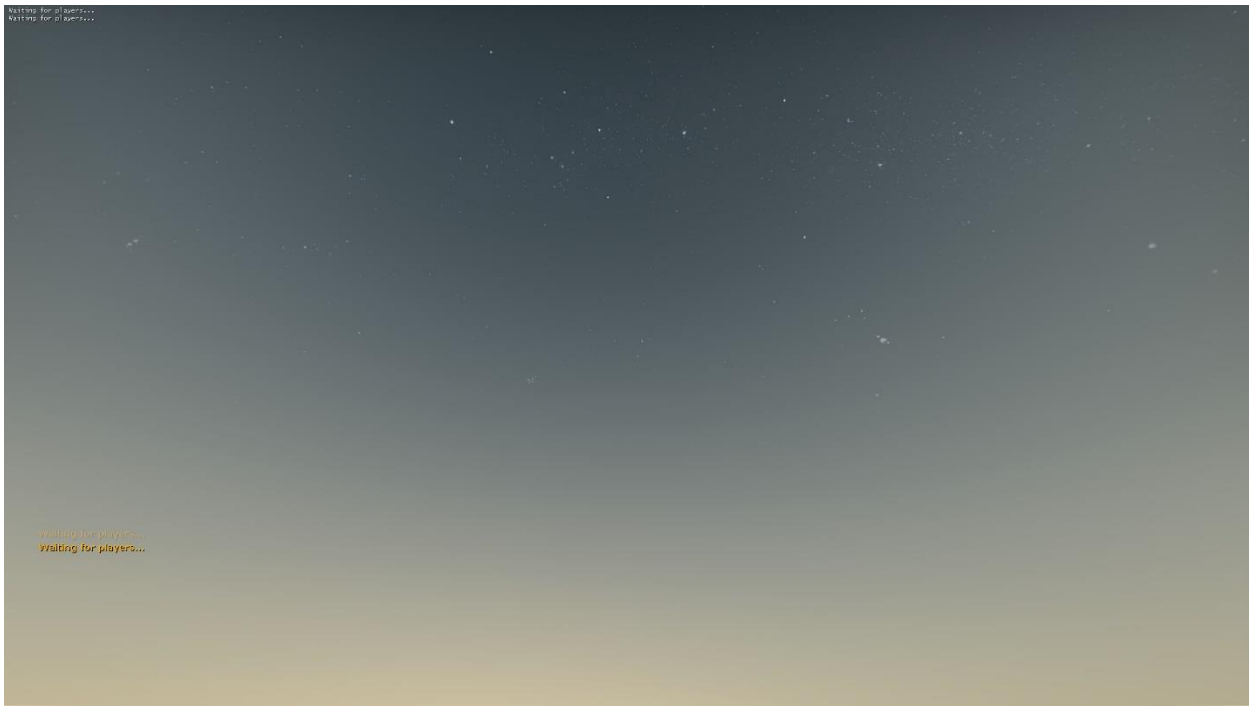


Figure 25: *Night sky at various sun positions*

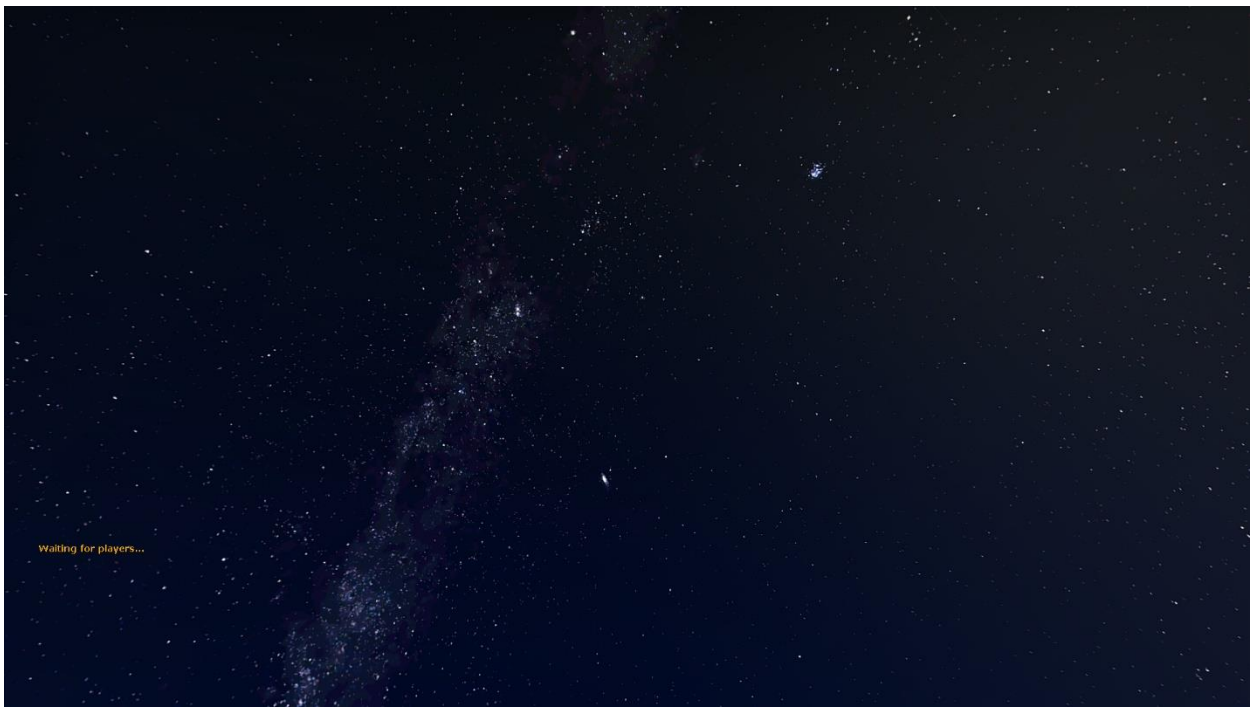


Figure 26: *Night sky at various sun positions*

Spherical texture can be mapped on screen space quad very easily (listing 4).

```
float2 uv = float2( atan2( viewDirection.x, -viewDirection.y ) / ( PI * 2.0f ) + 0.5f, acos( viewDirection.z ) / PI );
```

Listing 4: *Spherical texture mapping*

You can use this trick for daylight sky too and replace old skyboxes with simple post-process shader which will render spherical textures for you.

4.4 Volumetric Clouds

Second shader in a group of procedural sky shaders is volumetric clouds. This shader can generate stratus, cumulus and stratocumulus clouds. Clouds are ray-marched and heavily based off *Horizon Zero Dawn*'s system and their code samples (Schneider & Vos, 2015) and GPU Gems 3 book. They are generated from 3 types of noises: 128x128x128 base 3D perlin-worley noise for main cloud shape (figure 27), 32x32x32 detail worley noise (figure 28) and 128x128 2D curl noise (figure 29) to add effect of motion. Weather map (figure 30) is used to determine coverage, type (stratus-stratocumulus-cumulus) and precipitation.

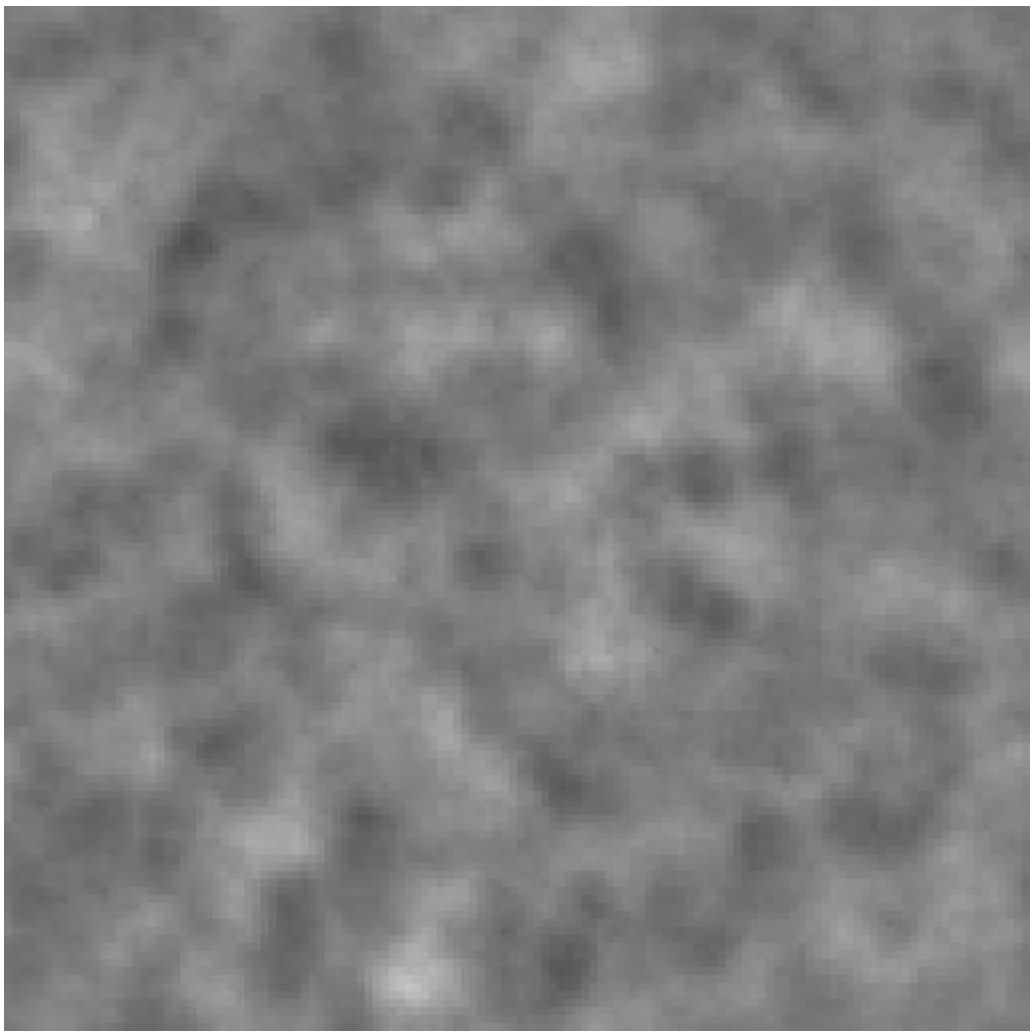


Figure 27: *Example of 128x128x128 perlin-worley noise*

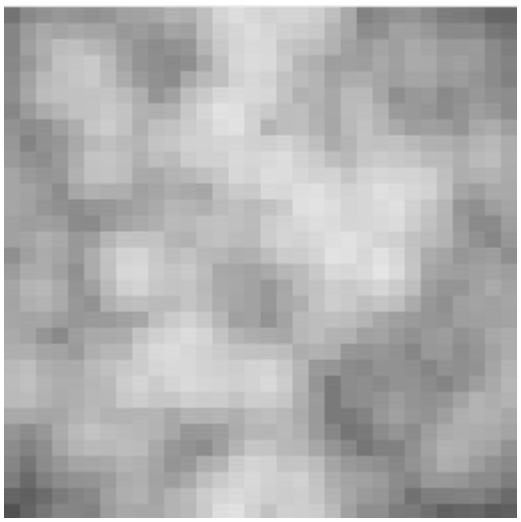


Figure 28: *Example of 32x32x32 high-frequency perlin-worley noise*

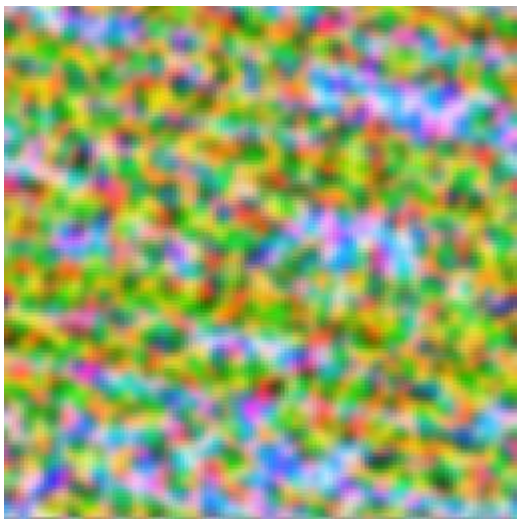


Figure 29: *Curl, which supposed to give clouds effect of motion*

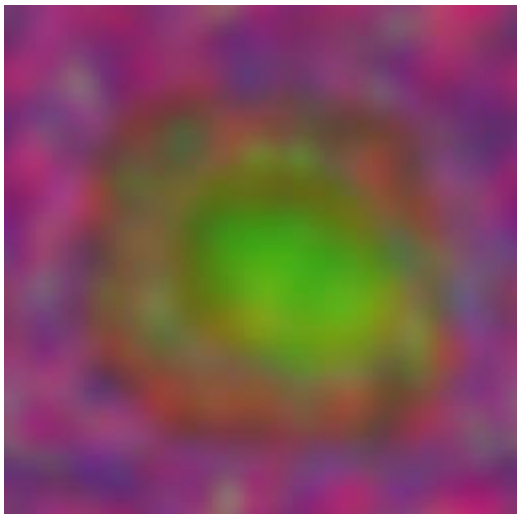


Figure 30: *Weather map*

Code samples can be found in GPU Gems 3 book and on *gamedev.net* forums. One of the greatest threads for this subject:

https://www.gamedev.net/forums/topic/680832-horizonzero-dawn-cloud-system/

Noise generation program can be found here by *Sébastien Hillaire* and *M-A Loyer*, who made it according to information from GPU Gems slides - https://github.com/sebh/TileableVolumeNoise.

My fork where I made this program compatible with Source's *vtex.exe* can be found here - https://github.com/DmitRex/TileableVolumeNoise

This program and the shader support packed noise, which saves on shader instructions (listing 5).

```
#if PACKED_NOISE
	float base_cloud = tex3Dlod( CloudNoiseTex, float4( coord3D.xy, height_fraction, lod ) ).r;
#else
	float4 low_freq_noise = tex3Dlod( CloudNoiseTex, float4( coord3D.xy, height_fraction, lod ) );

	float low_freq_fbm = ( low_freq_noise.g * 0.625f ) + ( low_freq_noise.b * 0.25f ) + ( low_freq_noise.a * 0.125f );
	float base_cloud = Remap( low_freq_noise.r, -( 1.0f - low_freq_fbm ), 1.0f, 0.0f, 1.0f );
#endif
```

Listing 5: *Example usage of packed and non-packed low-frequency noises*

And the same for high-frequency noise (listing 6).

```
#if PACKED_NOISE
	float high_freq_FBM = tex3Dlod( CloudHighFrqNoiseTex, float4( coord3D.xy * g_NoiseScale3D.w, height_fraction, lod ) ).r;
#else
	float3 high_frequency_noises = tex3Dlod( CloudHighFrqNoiseTex, float4( coord3D.xy * g_NoiseScale3D.w, height_fraction, lod ) ).rgb;

	float high_freq_FBM = ( high_frequency_noises.r * 0.625f ) + ( high_frequency_noises.g * 0.25f ) + ( high_frequency_noises.b * 0.125f );
#endif
```

Listing 6: *Example usage of packed and non-packed high-frequency noises*

4.5 Cloud Lighting

Clouds shader does 256 main raymarching steps and for each step it does 5 additional lighting steps. Lighting is described in HZD's paper as well. In EA's paper (Hillaire, Physically Based Sky, Atmosphere and Cloud Rendering in Frostbite, 2016) you can find more info about usage of *beer's law* and *Henyey-Greenstein*'s functions.

Figure 31: *Sky with beer-powder function used only*

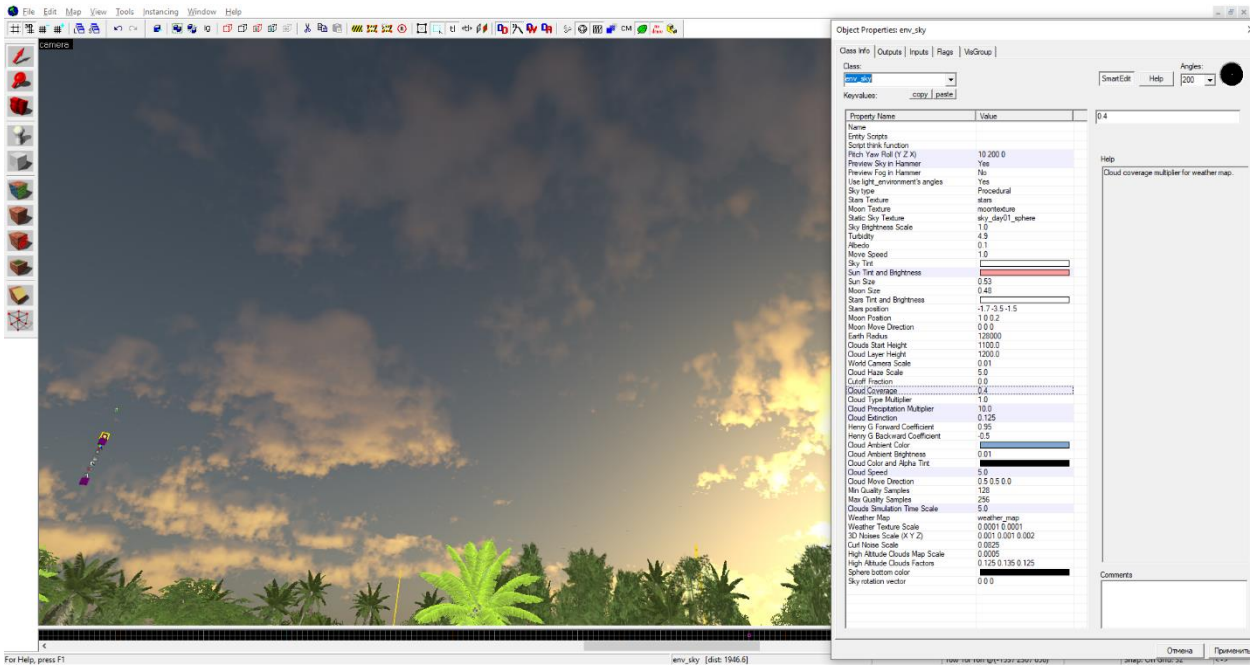


Figure 32: *Sunset clouds with both Beer's Law and Henyey-Greenstein's functions applied*

Of course, optimizations are taken in account, lighting passes are skipped when no cloud was found or cloud is fully opaque already, etc.

After the lighting is done, clouds are tonemapped using *Filmic Tonemapping* operator to prevent bloom buffer from flooding with very bright pixels.

4.6 Optimization

Cloud shader does 256 * 5 raymarching steps in worst possible case. Though, it's quite rare situation, this is still heavy workload for a GPU. Not taking in account shader algorithm's optimizations, there are two main ways to optimize it: lower the resolution and use temporal reprojection (Schneider & Vos, 2015).

Temporal reprojection is a technique which allows to render an effect during N frames. Clouds in MCV are rendered during 16 frames and combined with the result of previous frame. It means that during each frame only 1/16th part of screen is rendered, while the rest is re-used from previous frame or from some failsafe very-low res buffer. Failsafe buffer is mostly needed when player's view direction changes suddenly (ex. player turns back fast).

First of all, I lowered resolution of clouds to 1/4th of user's screen size, still framerate barely went above 30FPS, so the next step was adding temporal reprojection. With it, each frame we render only 1/16th of this buffer. Let's say, user has resolution 1920x1080, then clouds buffer gets size 480x270 and then only 32x16 part of this buffer is rendered per-frame. This brings maximum possible performance while having ~ok~ quality. Still, artists must use as low marching steps as possible to improve performance even more. As a result, all these optimizations together did enormous performance boost and made it possible to work in a multiplayer shooter in first place (Table 1).

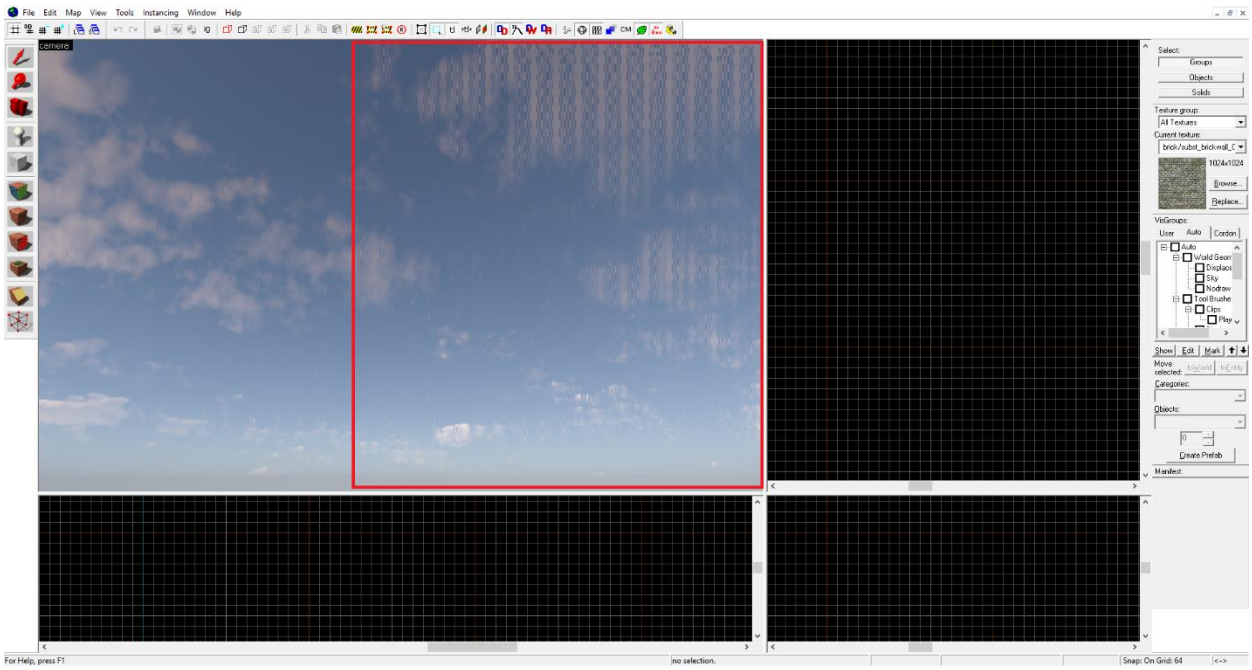


Figure 33: *Reprojection process can be very clearly visible in Hammer Editor.*

As you can see on picture above (figure 33), the reprojection process is very obvious, but it can be hidden using cross-pattern pixel selection (Häggström, 2018), for example, I prefer *Bayer Pixel Writing* - https://www.shadertoy.com/view/4tVcRm

4.7 Combining results and High-Altitude Clouds

After sky and clouds finished rendering it's time for last combine pass. This pass didn't exist in original implementation, it appeared as a consequence of cloud shader's over-complexity – DX9 compiler simply could not compile so big and complex shader.

Figure 34: *High-altitude clouds*

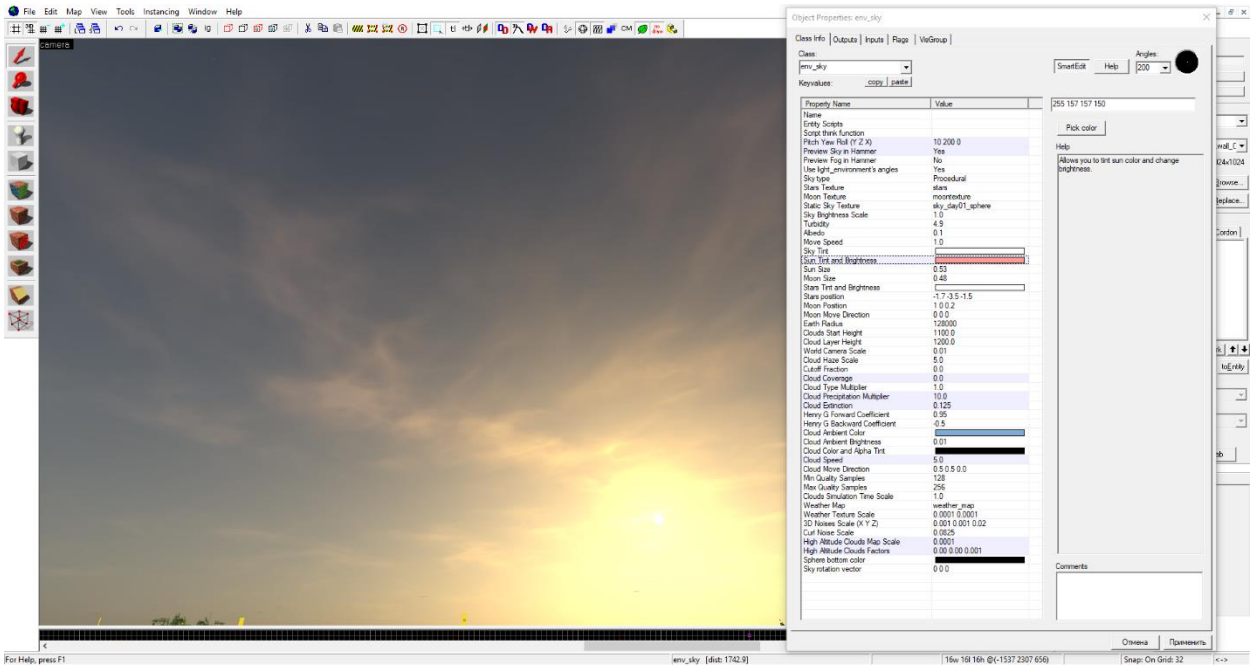


Figure 35: *High-altitude clouds in Hammer Editor*

High-altitude clouds (figure 34) is an additional simple layer of clouds which is added during combine pass and moving with different direction and at different height. These clouds are sharing the same lighting model as volumetric ones.

There are could be additional effects like *God Rays*, but they were cut for performance reasons.

Figure 36: *Final result with skylight model and both types of clouds combined*

4.8 Sky Preview in Hammer Editor

In the end, procedural sky was looking very good, but the shader and controlling entity had more than 20 various settings, so I was concerned about adding a GUI to control it in real-time. But instead of creating my own UI, I decided to go with Hammer Editor (figure 32, figure 35). Hammer support was very easy to add, since procedural sky is just a group of post-process shaders, so I just drawn them before rendering actual world. To make sky easier to configure, I also enabled fog rendering inside Hammer. The only drawback here is that you have to disable skybox visgroup to configure the sky and then enable it back before compiling.

5 Depth-based Effects and Sub-views Reprojection

Despite that several major features were already added to renderer, there was a room for improvement in depth-based effects and lighting. All of them are well-explained in books and papers, but in our case, there is much more work behind them – we have to render effects not only in main view, but in some sub-views as well.

5.1 Soft Particles

Soft particles is a very old engine feature, still it has never been used in games as it requires high precision depth buffer which Source™ always had problems with. In MCV we have high-precision 24-bit hardware depth buffer with 8 additional bits reserved for stencil. Connecting this to leftover soft particles code already provided desired result (figure 37).

Figure 37: *Soft particles*

5.2 SSAO and view model depth hacks

SSAO has been popular way to add self-shadowing to objects for a decade already. In Source™ it's already performed gently: *vrad.exe* does additional self-shadowing steps and models can use special pre-drawn SSAO masks. Nonetheless, we were looking for a real-time approach to save on texture memory, map compile time and improve on shadowing quality.

After trying many approaches, we've ended up using *SAO*, or *Scalable Ambient Obscurance* - https://research.nvidia.com/sites/default/files/pubs/2012-06_Scalable-Ambient-Obscurance/McGuire12SAO.pdf

Figure 38: *Scene without SSAO*

Figure 39: *SSAO Enabled*

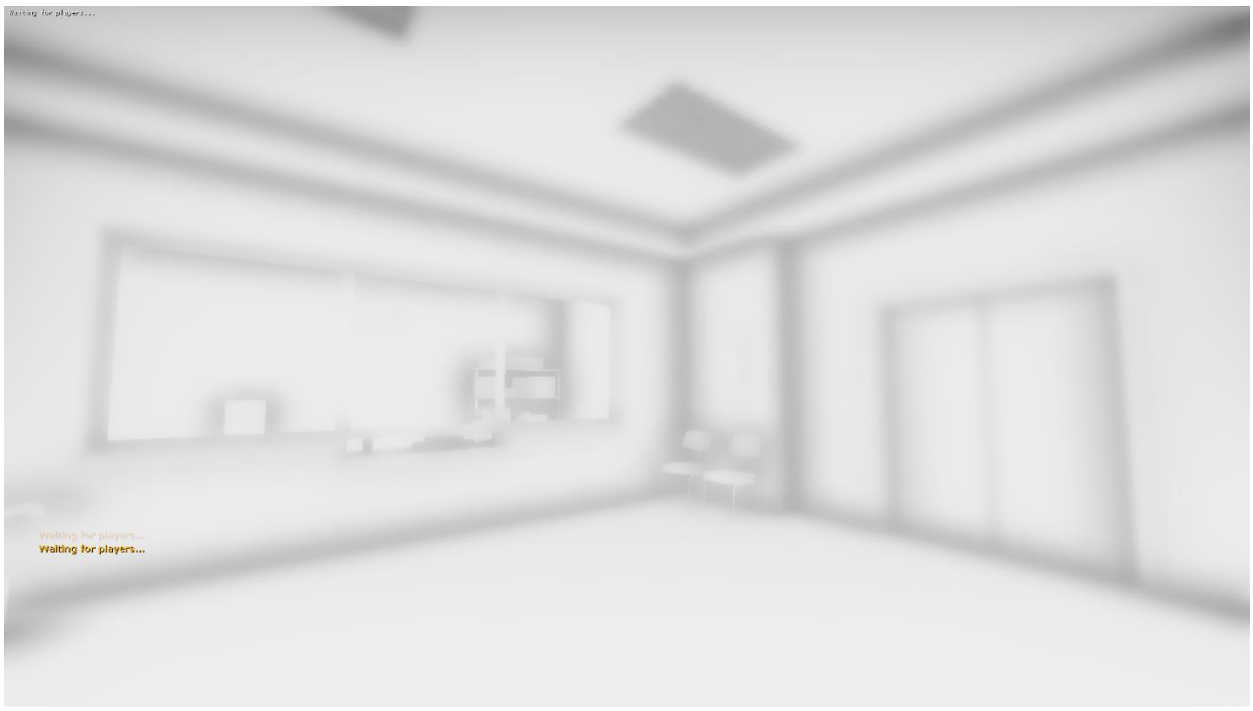


Figure 40: *SSAO Buffer*

To improve performance, we downsample original depth buffer using checkerboard pattern and use it in the shader instead. Downsampling process explained very deeply here: https://eleni.mutantstargoat.com/hikiko/depth-aware-upsampling-2/

SSAO effect in general was pretty straightforward to add and there are a lot of open-sourced examples on the Internet, but using hardware depth buffer had one serious drawback – **viewmodels rendering**. View models is a special group of objects which are rendered on top of final image with custom **depth** and FOV values. Figure 41 clearly show the issue:

Figure 41: *Viewmodel is enormous and pokes through floor and walls*

If you try to render viewmodels with the same FOV and **depth** settings as for main scene, then you will get the following:

Figure 42: *Viewmodel is small now, wrongly placed in relation to camera and still pokes through walls*

Technically, this is not an issue, this how viewmodels *should* work and it may cause major headache especially for people who are working with depth-based effects for the first time. People tend to solve this problem differently: from not drawing SSAO on viewmodels at all, to completely ditching viewmodels or render separate depth buffers specially for them. However, there is another option I preferred most: **viewmodel depth hack**.

First of all, we decided to render our own software colored depth buffer for all future depth effects. This makes a performance impact (we will try to optimize this and get of it later, in section 5.5 of this paper), but for now we can control what we want to render to it and how we want to do it. Plus, we can form a *GBuffer* later for any custom lighting effects. Now, since we have **two** depth buffers (software and hardware), we can render viewmodel for *hardware* depth regularly so it will look on screen as expected, but when we are going to render to *software* depth we can fake depth values in pixel shader (i.e use depth settings from main scene) – then in hardware depth buffer the viewmodel will appear as it should without poking through anything, but **SSAO will be generated from depth values we fed to pixel shader**. This will result in both correct view and correct SSAO at the same time, the only drawback here is that you will still notice the extra shadow around gun if you get too close to the wall, as gun would still have poked through it.

Below is an example of how this hack is implemented. First of all, we compute required matrices on client using data from main scene (listing 7).

```
// save off view-projection matrix for rendering correct Z-values for SSAO depth buffer
viewModelSetup.zNear = mainView.zNear;
viewModelSetup.zFar = mainView.zFar;

VMatrix vSavedDepthProjMat, vSavedDepthViewMat, vSavedDepthViewProjMatrix;
viewModelSetup.ComputeViewMatrices( &vSavedDepthViewMat, &vSavedDepthProjMat,
&vSavedDepthViewProjMatrix );

pRenderContext->SetCustomViewProjectionMatrixForColorDepthPass( vSavedDepthViewProjMatrix );

// now, modify znear, zfar for correct rendering of viewmodels to hardware depth buffer
// doing it this way allows us to render viewmodels on top of the world correclty using original matrices while
having REAL z-values in SSAO color buffer,
// otherwise viewmodels will poke through walls in SSAO buffer or have incorrect depth
viewModelSetup.zNear = mainView.zNearViewmodel;
viewModelSetup.zFar = mainView.zFarViewmodel;
render->Push3DView( pRenderContext, viewModelSetup, 0, NULL, GetFrustum() );
…
```

Listing 7: *Computation of custom view-projection matrix for viewmodel depth hack*

Then in pixel shader we simply do it like this way (listing 8):

```
#if ( SSAO_VIEWMODEL_VIEWPROJ_HACK == 1 )
	float4 vHackProjPos = mul( float4( i.vWorldPos, 1.0f ), g_ViewProj );
	vDepth.x = vHackProjPos.z / vHackProjPos.w;
#else
	vDepth.x = i.vWorldPos_projPosZ.z / i.vWorldPos_projPosZ.w;
#endif
```

Listing 8: *Computation of depth value from custom view-projection matrix*

Aside from that, there is one more important note to mention: we generate SSAO map as post-effect, but **apply** it during the rendering of each object in each object shader. This way it helps a lot with translucent objects, like when shadows can appear on top of fog, or translucent windows. It also prevents alpha blending issues and even performance drops from happening.

5.3 Temporal SSAO

Temporal SSAO is an expansion to original effect which aimed to reduce flickering of distant pixels and improve performance. The principle behind it is that we generate shadows from very low number of samples, but combine it with the result from previous frame. It will introduce small ghosting around moving objects, but on the good side, since we blend two frames, end result will appear less flickery when camera is moving.

Figure 43: *SSAO Normal Mode*

Figure 44: *SSAO Temporal Mode*

As you can see on figure 44, shadows appeared way blurrier – it's a result of using very low SSAO quality settings (which improves performance), but if you wait a few frames - multiple images will blend together and end result will look close to figure 43. You can find more details about this feature here: https://bartwronski.com/2014/04/27/temporal-supersampling-pt-2-ssao-demonstration/

5.4 RT Cameras and sub-views reprojection

Problem with viewmodels is not the only problem I met. When working on weapons, especially on sniper rifles, we've decided to replace sniper scope overlays with a real-time solution (figure 45).

Figure 45: *Realtime scope example*

We preferred to go with RT cameras. RT (render target) camera is a technique used to render scene from custom point of view onto given surface. The main goal here is to make it as cheap as possible without noticeable quality drop.

First optimization is that we don't render RT scope when weapon is not scoped. Second one is to use as low resolution as possible and do not redraw any effects for scope view. Though that could be enough for a cheap camera, there was noticeable quality difference: object receiving SSAO shadow now appears unlit and soft particles are rendering without "softness". Rendering separate depth and SSAO maps for scope cameras sounds like an overkill even on paper, especially for a DX9 engine. But our implementation has one major advantage: scope camera positioned and angled the same as main camera, only field of view is smaller. Knowing that, we may attempt to **re-project** SSAO and depth maps from main scene to scope camera view.

At first, we cache the colored depth map from main scene before viewmodels are rendered to it. The cached version will be used as a replacement for original depth when player will look through his scope camera. Then, we save view and projection matrices from main scene and use them in re-projection process. In shader we branch the code for main and sub- views and do reprojection (listing 9).

```
if( g_bSSAO )
{
	[branch]
	if( g_bSSAOSubViews )
	{
		float4 vSSAOProjPos = mul( float4( worldPos, 1.0f ), g_MainViewProjMat );
		diffuseLighting = BlendDiffuseLightWithSSAOProj( diffuseLighting, vSSAOProjPos,
cScreenSize, g_SSAOParams );
	}
	else
	{
		diffuseLighting = BlendDiffuseLightWithSSAO( diffuseLighting, ComputeScreenPos(
  i.vScreenPos ), g_SSAOParams );
	}
}
```

Listing 9: *Applying SSAO for sub- and main views*

Where "*BlendDiffuseLightWithSSAO"* stands for (listing 10):

```
float3 BlendDiffuseLightWithSSAOProj( in float3 diffuseLighting, in float4 vSSAOProjPos /*for sub views (ex.
sniper rt scope*/, in float4 vViewportMad, in float2 SSAOParams )
{
	float2 uv = ( vSSAOProjPos.xy / vSSAOProjPos.w ) * vViewportMad.xy + vViewportMad.zw;

	return BlendDiffuseLightWithSSAO( diffuseLighting, uv, SSAOParams );
}
```

Listing 10: *Calculating the re-projected texture coordinates for sub-views*

And while rendering actual depth map for main scene, the scope part of viewmodel is skipped: we simply draw our previously cached pre-viewmodel depth buffer instead in scope shader (listing 11, figure 46).

```
#if( SCOPE_DEPTH_HACK )
{
	vDepth = tex2D( BackbufferDepthSampler, ComputeScreenPos( i.vScreenPos ) );
}
#endif
```

Listing 11: *Trick to return pre-viewmodel depth values for scope shader*

Figure 46: *SSAO depth map with "skipped" scope part of rifle*

As a result, we get the following (figure 47):

Figure 47: *SSAO map re-projected from main scene to scope camera*

Now, SSAO is visible through sniper scope at low cost without extra render passes (Table 1). The only drawback is reloading animations – when gun is moving it can obscure part of SSAO map and cause artifacts, but this issue is never noticeable in motion.

The same trick is done with soft particles. Math there is more complex, as particles have multiple render modes and alignments (listing 12):

```
// for sub-views (rt scopes) use viewprojection matrix from main camera so particle can be projected correctly
#if ( SUB_VIEW_DEPTHBLEND )
	#if ( ( ORIENTATION == 1 ) || ( ORIENTATION == 3 ) || ( ORIENTATION == 4 ) )
		projPos = mul( float4( worldPos, 1.0f ), g_MainSceneViewProj );
	#elif( ORIENTATION == 2 )
		projPos = mul( float4( wpos, 1.0f ), cModelViewProjMainScene );
	#else // ORIENTATION == 0
		// Screen-aligned case
		float3 viewPos;
		viewPos = mul4x3( v.vPos, cModelViewMainScene );

		float3 disp = float3( -x1, y1, 0 );

		float tmpx = disp.x * sc_yaw.x + disp.z * sc_yaw.y;
		disp.z = disp.z * sc_yaw.x - disp.x * sc_yaw.y;
		disp.x = tmpx;

		viewPos.xyz += disp * rad;

		projPos = mul( float4( viewPos, 1.0f ), cProjMainScene );
	#endif
#endif
```

Listing 12: *Calculating re-projected from main scene position for Depth Blend in sub-views*

And the result (figure 48):

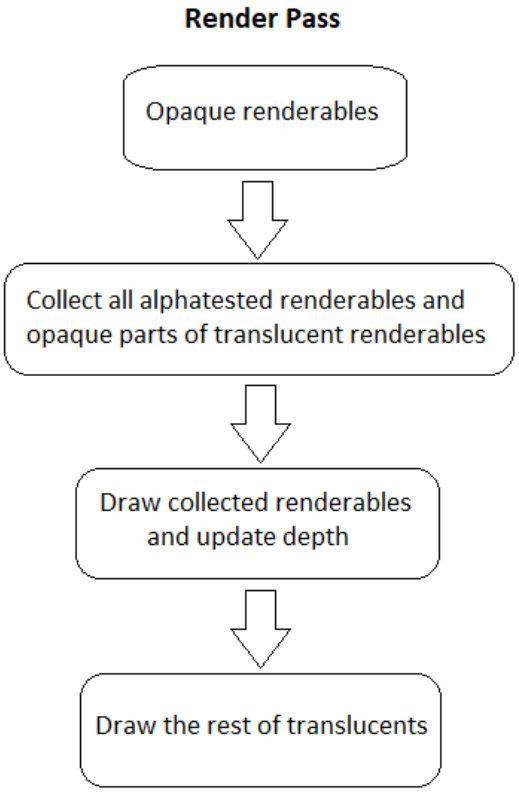
Figure 48: *Soft particle appears correctly in Sniper RT camera without any extra depth buffers and render passes*

5.5 Depth buffer optimization tricks

Previously, we mentioned that we have two depth buffers: software 32-bit floating point one and 24-bit hardware one. This allowed us to control what exactly we want to render to it, solve viewmodel-related issues and, potentially, form a *GBuffer* for future Deferred Lighting implementation. Unfortunately, performance with separate depth buffer was still not good enough for us, so we had to ditch Deferred Lighting idea and try to remove software depth pass as well, without breaking previously implemented features.

I come up with an idea that we can copy hardware depth result to our software buffer and simply render viewmodels on top of it using previously mentioned hacks. This way we only render viewmodels twice for our depth buffer, not the whole scene anymore. We still have to make an extra copy of it before rendering viewmodels, so previously mentioned sub-views tricks can work. And this worked, though, not as good as it's supposed to.

First of all, alpha-tested renderables were missing from depth, and hence SSAO buffers. This is because game updates hardware depth buffer *before* rendering translucent objects. Moving update code *after* rendering translucents caused all of them to appear in depth buffer and screw it up. So, what we did to solve this problem is that we manually iterate through all translucent objects and collect alphatested ones in special container, then we render them, then we update hardware depth and, at last, render the rest of translucents (graph 1).



Graph 1. *Sorting translucents pipeline to get correct depth buffer*

Worth to mention as well that we have changed our software depth render target's depth mode to **MATERIAL_RT_DEPTH_SEPARATE** instead of **MATERIAL_RT_DEPTH_SHARED** since operating depth values while having it in shared mode may corrupt original hardware depth for some GPUs.

The next problem is that SSAO now is 1 frame late and cause noticeable trailing when player moves camera aggressively. Previously, we rendered depth and SSAO buffers *before* rendering main scene, but now we have to render SSAO *after* we finished rendering alpha-tested renderables, hence we have to move SSAO to post-process again. This was an easy task to do, though it may bring back artifacts with alpha blending or seeing SSAO through water surface, but currently we are fine with that. We had a new issue with fog and had to manually blend SSAO with it in SSAO shader, because fog is still applied per each object.

At the end of the day, after all major issues were solved, we have ditched extra depth buffer render pass, but kept all goodies we have got with it. SSAO and depth effects were still working, sniper RT scope camera reprojections and viewmodel hacks were working too, but we saved a lot on not doing extra draw calls. Currently, if player have RT scopes disabled, game is not doing **any** extra draw calls apart from viewmodels, which is very good trade-off in terms of performance. This allowed us to get **7-10%** performance boost depending on hardware.

6 Volumetric Lighting

One of my latest major efforts was volumetric lighting. Main goal was to add rays going from the sun which will not disappear if you do not look at them, like in early games, for example, *Crysis 1*. But thanks to experience with *Procedural Sky*, we can utilize some of those technologies to make something more complex than just god rays.

6.1 Sun Shafts

The only way for us to make god rays stay when sun is not in player's point of view is to use CSM's shadowmap. We reconstruct world space position of pixels using depth map, compute viewing direction and start ray marching from camera's position. Then simply if we didn't hit CSM shadow – we add light, if we did – do not. Lighting model here is similar as in *Procedural Sky*, end result is tonemapped as well.

Figure 49: *Sun rays*

In order to improve performance, we downsample our depth buffer to ¼ of screen size and render rays to a ¼ buffer too. In the end we upscale image back to full size using bilaterial upsampling.

6.2 Height Fog

Since we know world space position of each pixel on screen, we can determine what pixels may have and what pixels may not have volumetric effects. Let's say that pixels 256 units above the ground should *not* have fog and we will get result as on figure 50:

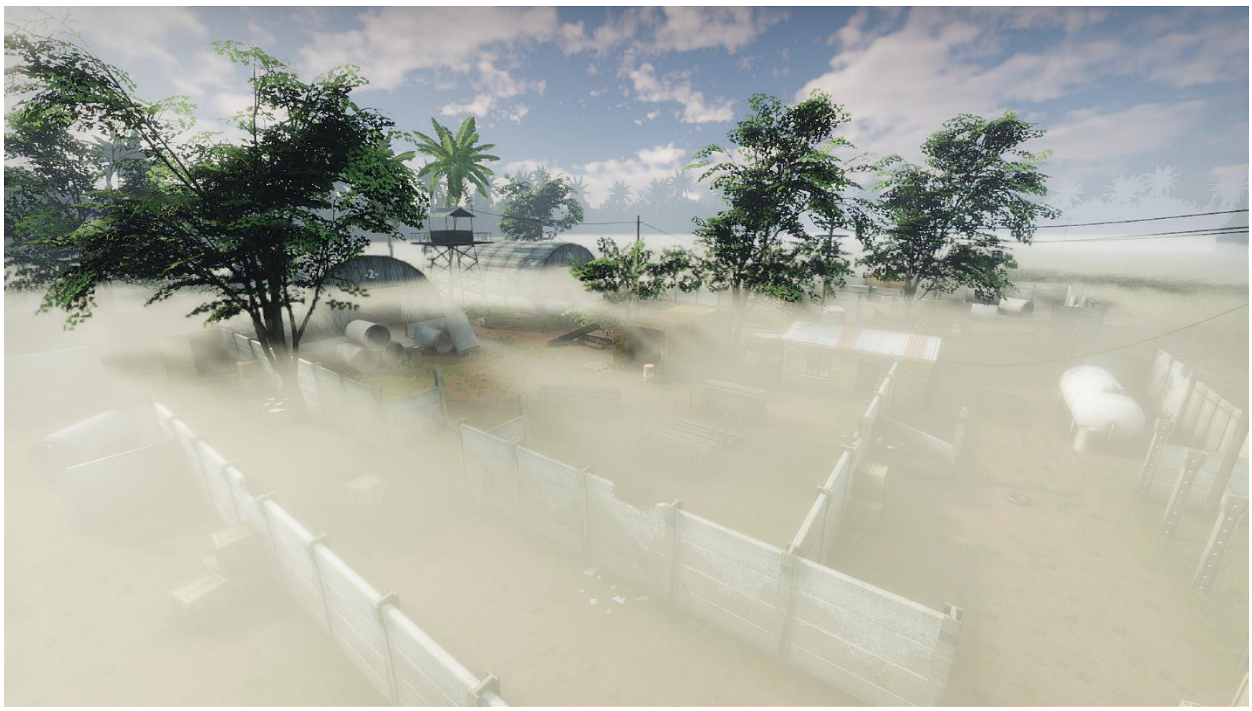


Figure 50: *Height fog*

6.3 Future Work

For now, the shader is only used for sun light, but it's also possible to implement support for point and spot lights. The drawback here will be a shadowmap: we don't have enough free resources to spend on generating shadowmaps for each point/spot light, so the effect will look more like dynamic glow sprite, rather than proper volumetric light. Still, this is something that could save us a couple of *Draw Calls*, since "spotlight"-sprites will not be rendered anymore.

7 Parallax Mapping

Parallax mapping always been a dream for many Source™ developers, but only few of them made it working in engine and even less of them used it right.

While looking through various implementations of this shader I saw many various drawbacks and artifacts, most of them were due to the fact that people have used the simplest and popular approaches. I've decided to go further and started looking for an advanced approach and, luckily, found one – *Interval Mapping* (Risser, Shah, & Pattanaik, 2007).

7.1 Interval Mapping

Interval mapping is the one of most advanced DX9 techniques to create parallax effect. It's the same as *Relief Mapping*, but binary search was replaced with interval one, which increases effect detail and performance.

Figure 51: *Example of sand depth illusion created by Interval Mapping shader*

Interval mapping allows us to achieve better visual results with the same amount of iterations as with relief mapping (figure 52), or equal results using lesser amount of them which helps a lot with performance in our already shader-heavy project.

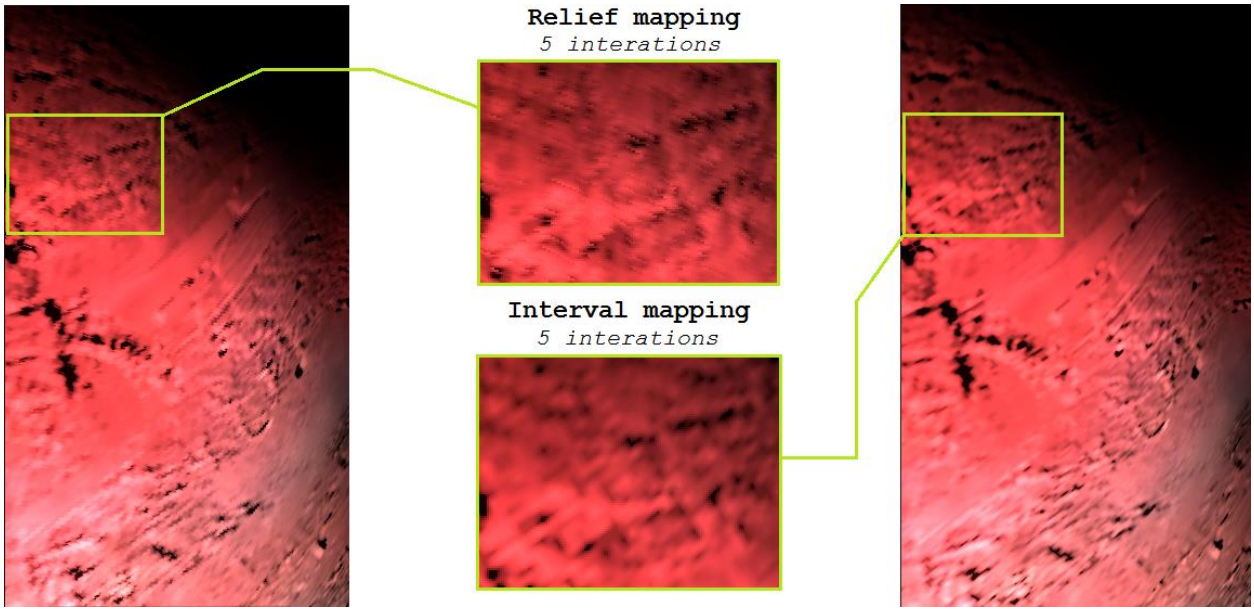


Figure 52: *Comparison between interval mapping (right) and relief mapping (left) with five iterations taken in interval and binary search steps respectively.*

But it doesn't really matter what implementation do you prefer when it comes to actual using the shader in maps. Almost every implementation will look good while the shader is applied to plain surface and looked from near-perpendicular angle. This is not common scene in games, many surfaces are not plain, non-perpendicular, can be looked at from variety of angles, etc. Let's apply parallax on a parallelepiped and see what might happen with it.

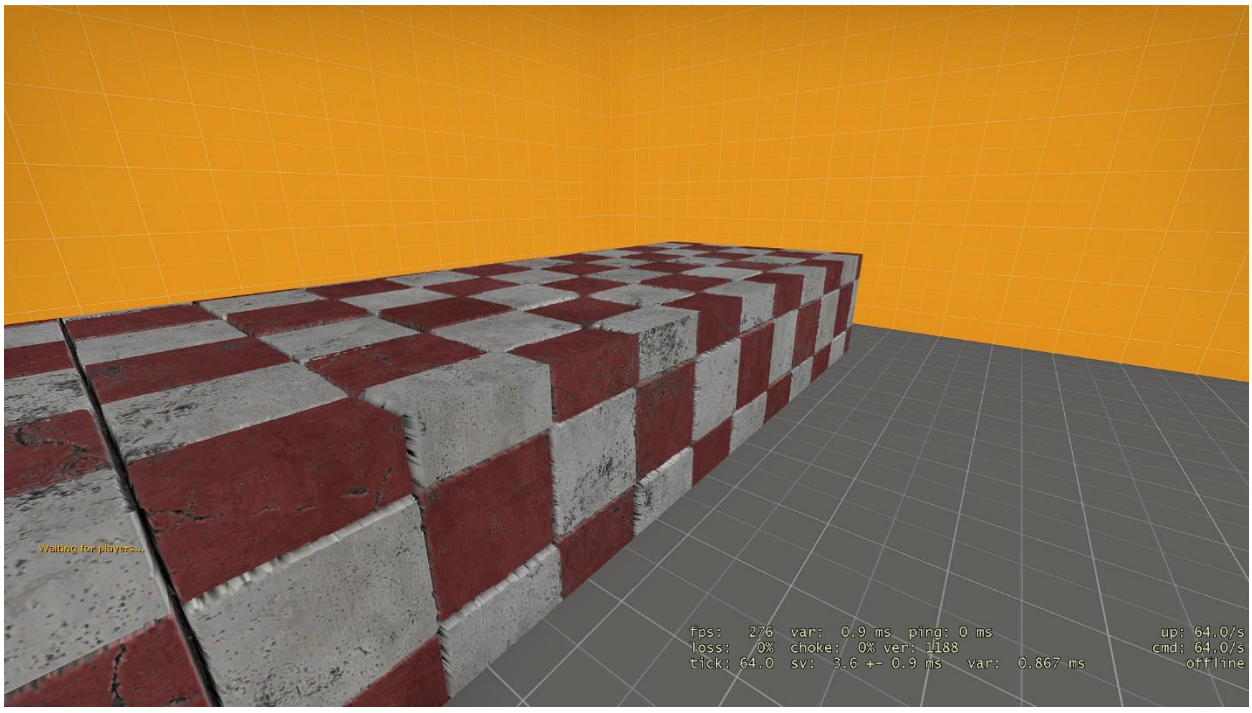


Figure 53: *Simple parallelepiped*

Let's take a look at this simple parallelepiped (figure 53). From current perspective it looks as expected, but if we actually step on it, we will see the following (figure 54):

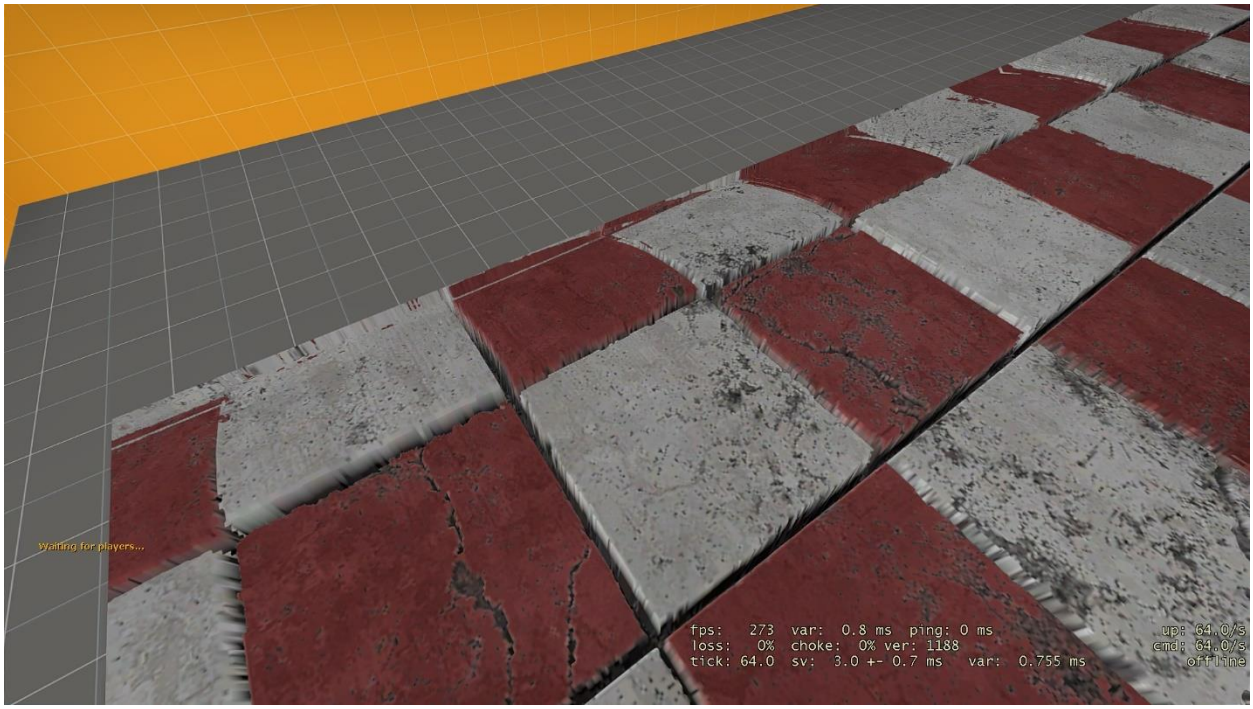


Figure 54: *Parallax distortion*

Distortions will appear at the edges of brush – this is not shader's fault, it's a result of vertex normals blending.

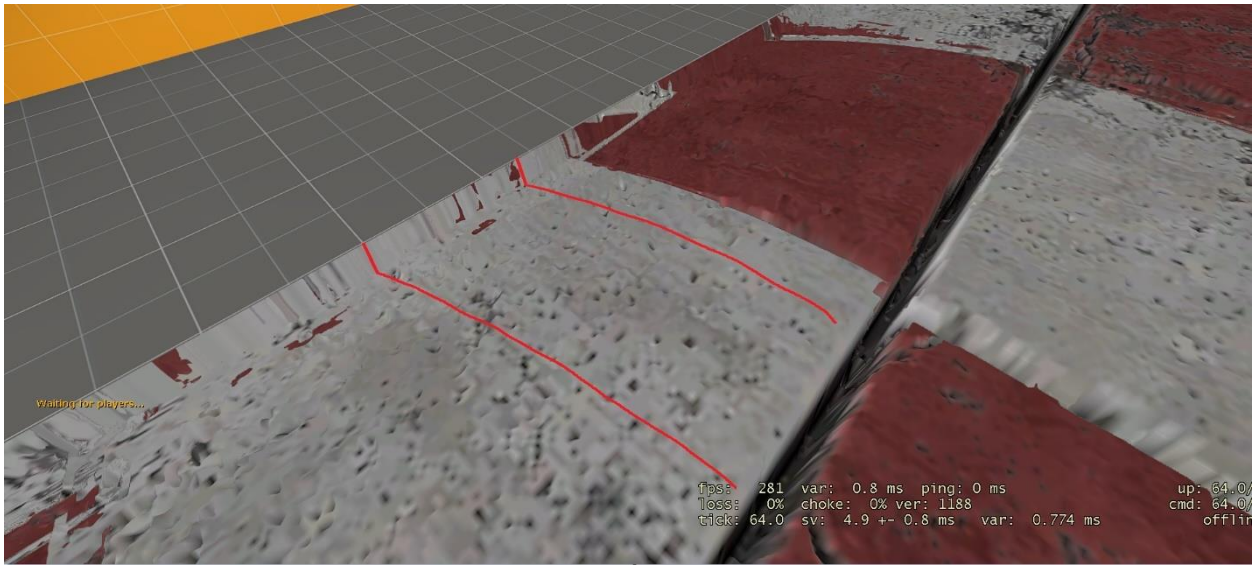


Figure 55: *Parallax distortion closeup. Shader attempts to make the corner round*

By default, vertex normals are always blended according to DirectX specifications, but additionally map compilers can combine meshes which share common edge so their vertex normals are blended together too.

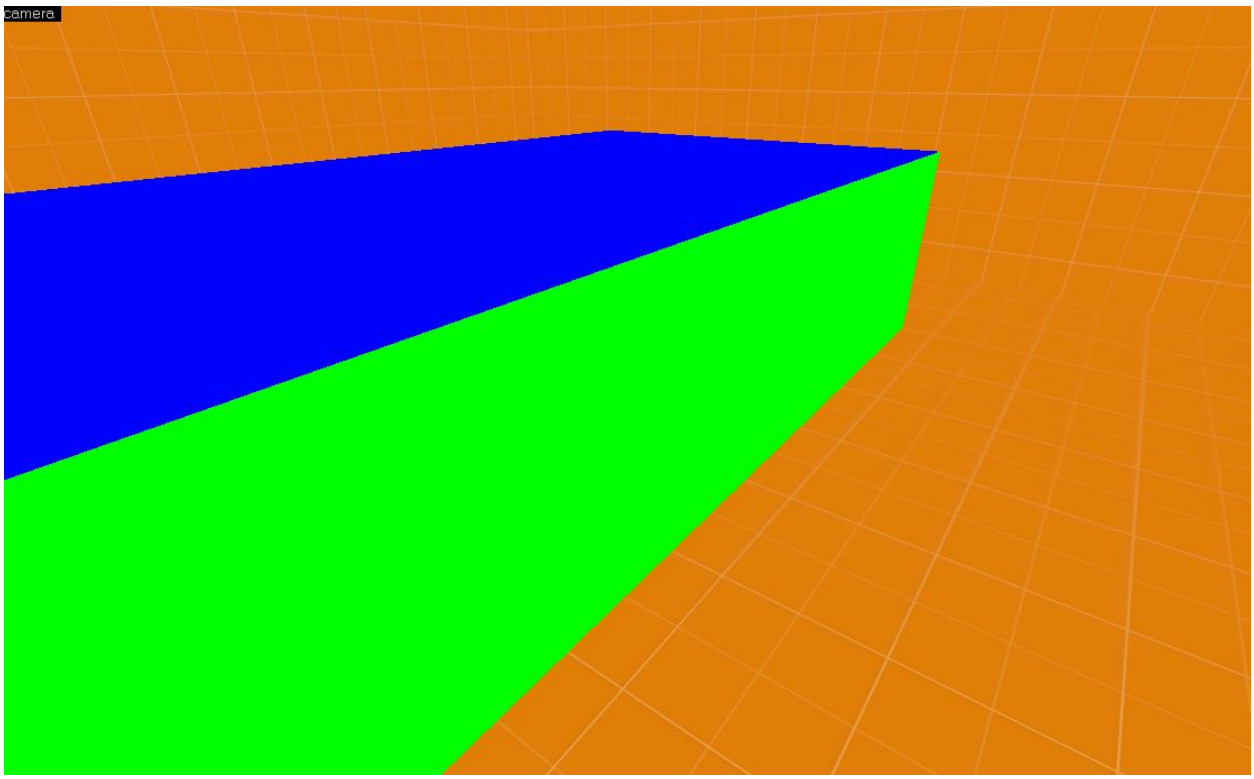


Figure 56: *Hammer Editor shows unmodified vertex normals*

In Hammer Editor this process is not happening, so we can see the unmodified vertex normals, which exactly what parallax shader requires for correct result. But if we compile the map, we will see the following (figure 57):

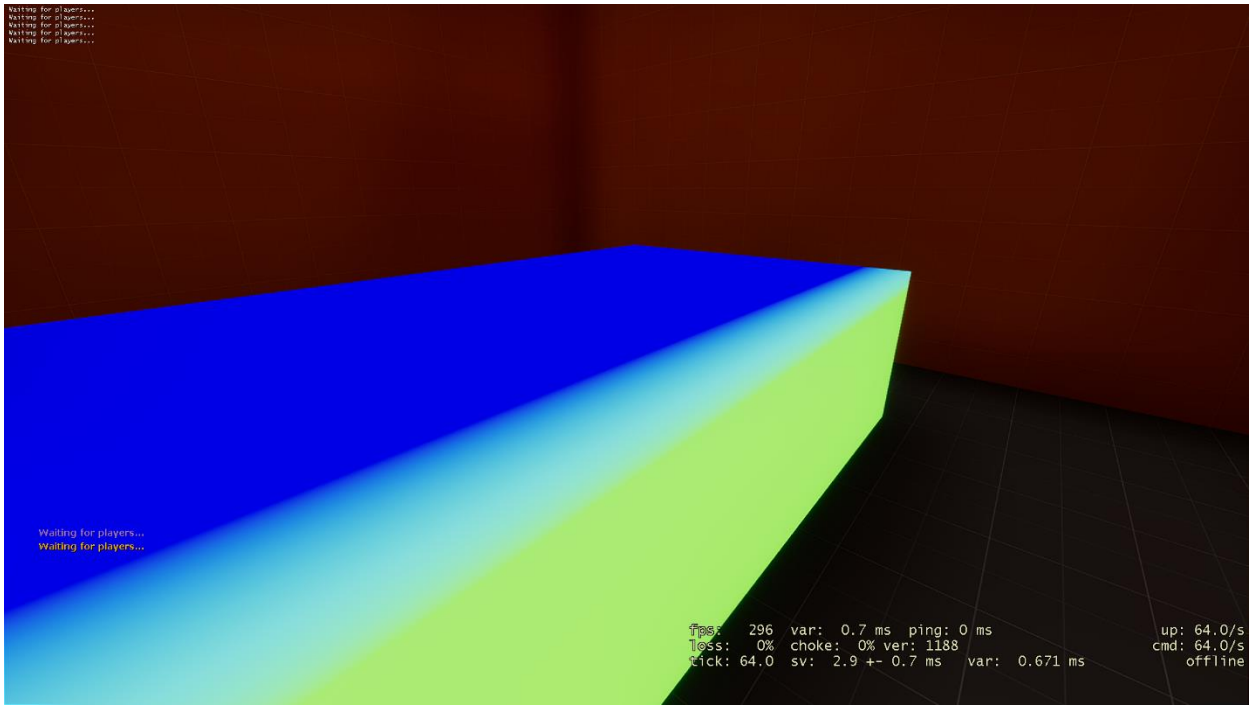


Figure 57: *In-game view of blended vertex normals*

In-game we see that normals are now blended (figure 57) and this is the cause of distortions. Here are a couple of ways how to solve it: remove smoothing via Smoothing Group Window in Hammer Editor, disable smoothing at all during compile process if surface is using parallax shader, or pass unmodified normals as additional array of data to shader. We preferred 2nd option and trying to avoid complex surfaces when using parallax shader in general.

7.2 Future work

Despite the fact that this shader became very complex with support of 4-way blending, CSM and other engine features, there are still rooms for improvement. First of all is silhouettes – if you look sideways at the object, you may notice that shader never takes object's edges in account. It just continues creating illusion of depth infinitely. Second problem is lightmapping – for bumpmapped surfaces engine computes 3 lightmaps at given 3 directions and then combines them into one (McTaggart, Half-Life® 2 / Valve Source™ Shading, 2004). Given that in mind we need to calculate parallax UV coordinates for lightmaps as well, but this is an overkill for performance and dozens of extra shader instructions. As better solution we may possibly calculate parallax'ed lightmap coordinates during map compile process, but we will have to keep both default and parallax'ed lightmap coordinates in memory in case user wants to turn Parallax Mapping off.

8 Additional Effects

8.1 Barrel Distortion

Barrel distortion is a fisheye-type post-process shader which aimed to reduce image stretching at high FOV values (Carpentier, 2015).

Figure 58: *Example scene with FOV 150 and barrel distortion applied to neutralize FOV artifacts.*

Figure 59: *Original unmodified image*

Shader can also help at regular FOV values (ex. 75-90) (figure 60).

Figure 60: *Regular scene with FOV 90*

Figure 61: *Same scene but with Barrel Distortion applied*

Notice how objects near screen edges are now less distorted (figure 61). This can help people with motion sickness play MCV for longer time.

You cannot render 360-degree panoramas, though, due to shader's post-process nature. At larger FOVs image will just break into pixels.

8.2 Bokeh Depth of Field

Bokeh DOF is another post-process shader, which works by generating CoC (circle of confusion) map for near and far planes, similar to DOOM 2016 (Courrèges, 2016). Activated mostly when players are using iron sights, or at scripted events.

Figure 62: *Pentagon-shape blur*

Figure 63: *DoF when looking through the scope*

8.3 Motion Blur

Motion Blur in MCV is a relatively simple technique. Originally, this post-process shader supposed to utilize *Velocity Buffer* for every pixel in scene, but unfortunately, after we got rid of GBuffer-pass it became highly unlikely to make possible.

Still, we generate screen-*Velocity Buffer* and do camera motion blur. Even with this limitation, image is blurred in the direction of moving and objects at the edges of the screen are blurred more than ones in the center.

Figure 64: *Motion blur in action*

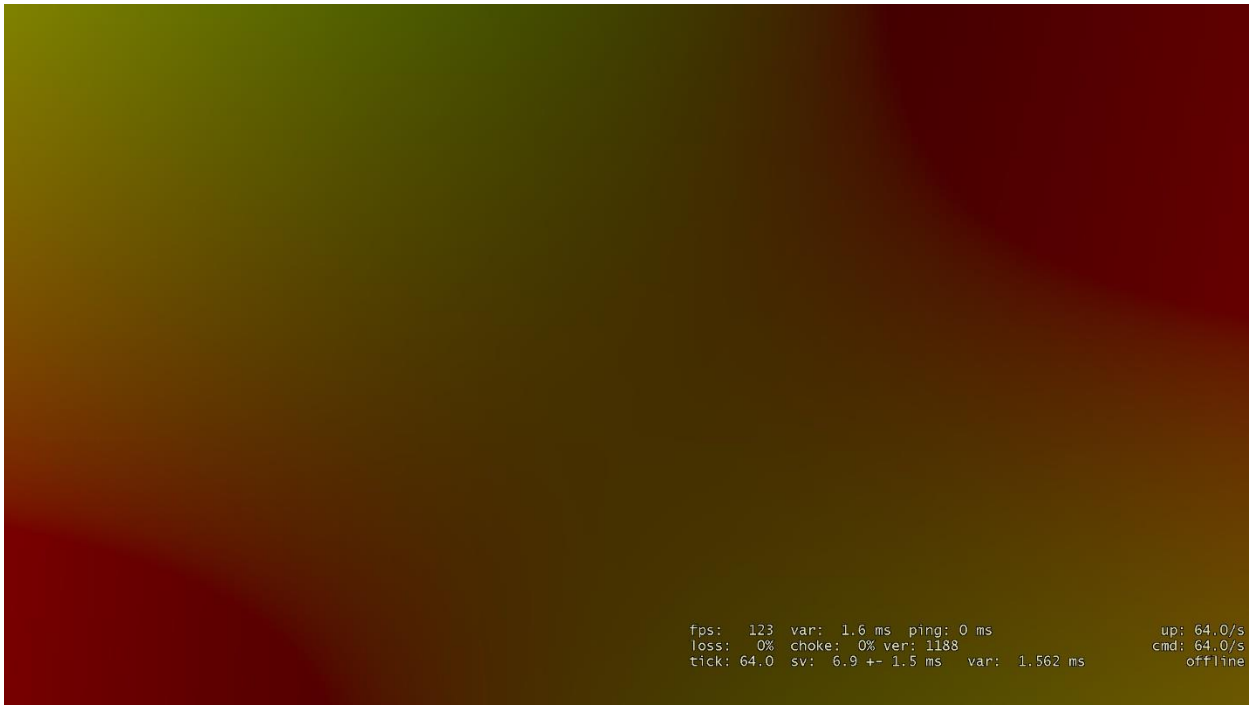


Figure 65: *Screen Velocity Buffer*

8.4 Subsurface Scattering

Subsurface Scattering is very complex technique which, when done properly, requires rendering additional buffers with object's back sides in order to provide natural simulation of light passing through an object.

Unfortunately, in MCV (and in DX9 in general) this technique is very simplified due to API limitations and performance issues. Currently, it's used only on Sniper Scopes, windows and other objects where drawing back sides is not required: We ended up using implementation as mentioned in this post: https://www.alanzucconi.com/2017/08/30/fast-subsurface-scattering-1/

Figure 66: *Sun light passing through sniper scope*

9 Conclusion and Future Work

Putting all this work together was challenging task, because engine was not supposed to handle so many different effects and something was constantly breaking or had compatibility issues with old engine's code. Though, after all serious issues were fixed, final look of the game has changed dramatically. It doesn't look like any other Source™ game anymore and does run fast enough. Procedural clouds make the sky dynamic and alive, volumetric lighting adds fine detail and creates unique atmosphere on map, lighting effects and reprojection tricks create an impression of a modern looking game.

Figure 67: *Final look of the game with SSAO, Volumetric Lighting, SMAA and other effects working together*

There are a couple of issues left here and there, which still need to be fixed, but in general we can make final conclusions now.

9.1 End Result and Performance

After all major features were described, we can combine them and their corresponding performance into single table and see if we have met our goal. Let's do not forget that most of the players will prefer having some effects off while others will be turned on, so in fact game will run faster than in our tests. Nonetheless, there can be various hardware-dependent issues which will be discovered later. For our performance tests, we have built a system with a NVidia GeForce 1070 video card running at 1920x1080 resolution and here is what we've got:

Effect name	Performance
MSAA Off, SMAA 1x	0.917ms
MSAA 4x, SMAA Off	0.780ms
MSAA 4x + SMAA 1x	0.950ms
MSAA 8x, SMAA Off	0.982ms
MSAA 8x + SMAA 1x	1.250ms
Procedural Sky (without reflections)	0.418ms - 1.578ms
Procedural Sky (with reflections)	1.733ms – 2.760ms
SSAO Normal	1.316ms
SSAO Temporal	0.933ms
SSAO Normal + Sub-views reprojection	1.333ms
SSAO Temporal + Sub-views reprojection	0.953ms
Volumetric Lighting	0.917ms
Parallax Mapping	0.120ms
Post Effects	0.880ms

***Table 1**: Performance of main shaders*

So, returning back to *sub-views reprojection* feature, we can see that SSAO gets only ~0.02ms slower when RT camera is active which is very good result. If we had SSAO re-rendered for sub-views we would have wasted additional millisecond on it. As for procedural sky, you can see that speed is heavily dependent on complexity of sky scenery. When there are a lot of cloud types and height variations on screen the shader may slow down to *3ms*, but we are trying to keep such sceneries for more open, fewer detailed maps to compensate performance loss. SMAA itself takes about a millisecond to render, but when combined with hardware MSAA you can see that 4x MSAA + SMAA 1x can save you about *0.03ms* of rendering time. Not much, though, but it depends on scene's complexity as well and still way faster than 8x MSAA + 1x SMAA while visual difference between them is negligible. Parallax mapping turned out to be the fastest shader here, but it's mostly due to clever usage of it on maps and setting up shader parameters by artists and level designers. Volumetric Lighting is one of few shaders, which had no major issues with optimizing and doesn't require any special settings per-map, it takes almost *1ms* to render mostly because of CSM shadowmap texture reads, but in general it works fast enough.

HDR rendering pipeline is not mentioned in the table, but it takes *~0.5m*s of rendering time. Resulting performance of all custom effects is *~7m*s in total – this is not really fast, all these shaders can work faster on newer DirectX APIs or Vulkan, but for now we are trapped with DX9 API and have to live with it for some while. Nonetheless, **we have met our goal** – FPS never goes below 60 even in not fully optimized maps on NVidia GeForce 1070 at 1920x1080 with all settings maxed out. Game was also tested on various AMD hardware and confirmed working well there too.

In the end, I can say that I was happy working on all these shaders and brainstorming optimization tricks, it gave me the most valuable experience. Now it's time to move forward and work with newer APIs on way more advanced techniques.

9.2 Future Work

Originally, there were more tech we wanted to have in MCV than actually made it to our game. One of them was **NVidia PCSS.** This technique aimed to create realistic looking shadows by using various blur kernels depending on how far shadow caster and receiver are from each other (Myers, Fernando, & Bavoil, 2008). The best way to test it was to import original *Half-Life 2™* map, because it has many scenes where different shadow angles and atmosphere settings can be seen. On example below (figure 68) there is scene with tall trees casting shadows on the ground:

***Figure 68**: Original imported Half-Life 2™ scene with CSM shadows*

***Figure 69**: Original imported Half-Life 2™ scene with CSM shadows and PCSS on*

As you can see on figure 69, PCSS shader makes shadows from tall trees very blurry and gets them very close to pure lightmapped variant, but dynamic. This is actually looking good and this is how it's *supposed* to look in real life, but, unfortunately, there are many artifacts and conflicts with how our scenes are built. For example, shadows are now missing many details, like ropes or subtle shadows from smaller objects. Moreover, blur from distant shadow tend to leak over nearby shadows and blur them even if they shouldn't be blurred. This is because single shadow map cannot store values from multiple blockers, hence only farthest blocker will be taken in account. Here appears a performance impact, caused by requirement of using very wide Poisson disks, which are not hardware-accelerated, obviously. While giving results close to real-life effects, we still have to search for another soft shadow algorithm.

Physically-based rendering. PBR was our dream for a long time already and couple of years back I've decided to give it a go. Our PBR implementation has Image-Based Lighting (IBL) support and using Metal/Roughness approach.

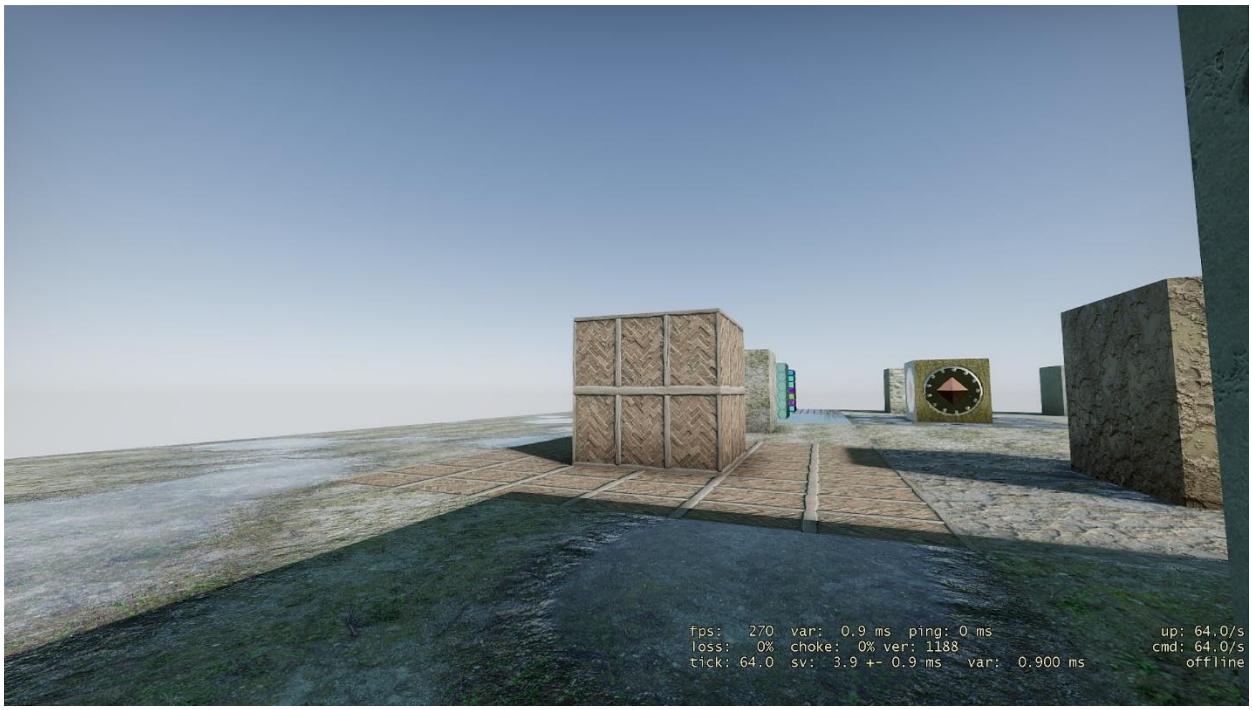


***Figure 70**: PBR lighting scene test*

After first test version was made, we have immediately noticed how easy, fast the shader is and how good it looks. Such effects as *Phong Lighting* or flashlight, which are separate techniques added (sometimes with hacks) on top of vanilla shaders and have various issues with them, are now part of our PBR shader's **nature**. Now there is no need to setup dozens of shader parameters for each material and flashlight now is just another light source with the same rights as any other light source in the map. Moreover, PBR shader works slightly faster than vanilla *LightmappedGeneric* or *Phong* shaders. Everything looked very promising, but we've met two major issues: first one is that 80% of our content was already made having vanilla shaders in mind and recreating it would require additional time and work while visual difference won't be ground breaking. Second issue is our IBL implementation. To make our PBR shader look as realistic as possible we have to put thousands of pre-filtered light probes in our maps.

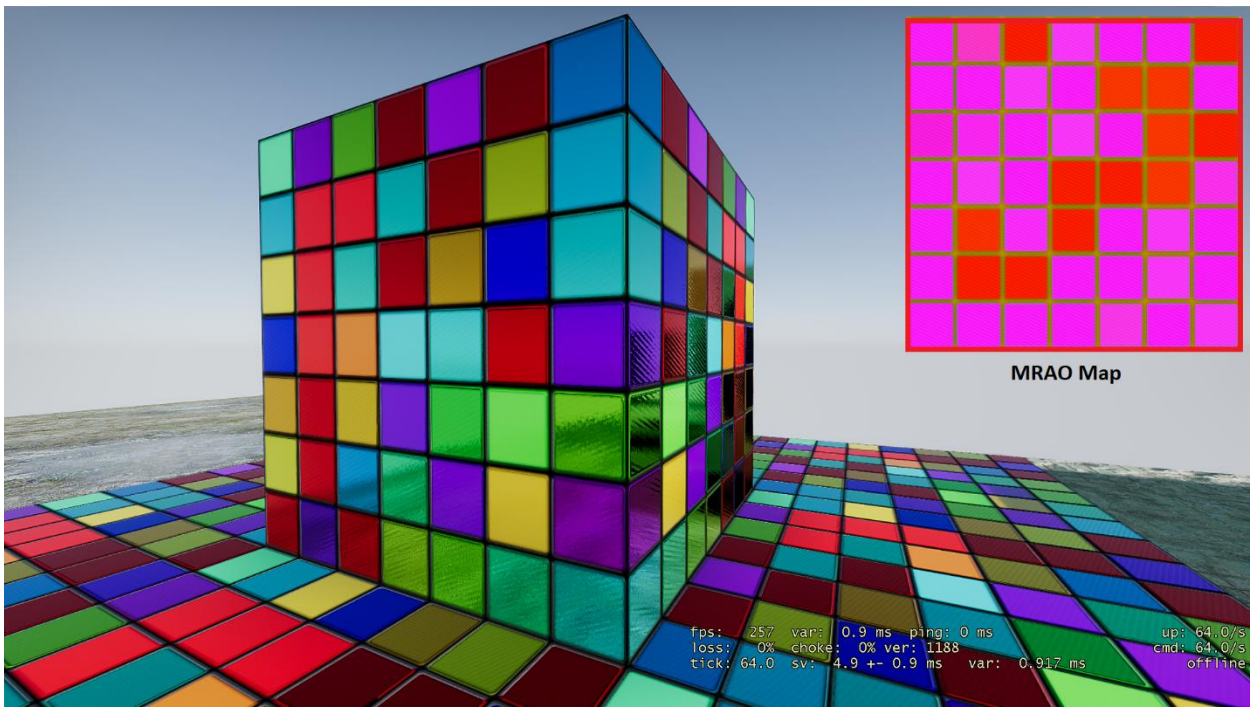


***Figure 71**: PBR Testing material*

While generating a screen-space grid of light probes is possible, DX9 API is not capable of filtering them. We can perform filtering on CPU, but it will obviously slow down map compile time and since our future plans about PBR are yet to be determined, we have postponed any work on this shader for now. All in all, our PBR implementation has potential and we are definitely going back to it in the future.

***Figure 72**: PBR test lighting model*

***Figure 73**: PBR test lighting model*

Apart from these great major features, we have mostly to polish our existing shaders and make sure they work on as wide variety hardware as possible. There is not much of what else we can implement under existing API, moreover, adding new effects without any particular reason would hurt performance and game's look. In the future we are going to switch our API and go with full Physically-Based Rendering.

References

Sal 'Sluggo' Accardo, Half-Life 2: The Lost Coast (2005) - http://pc.gamespy.com/pc/half-life-2-lost-coast/662640p1.html

Gary McTaggart, HDR in Valve's Source Engine (SIGGRAPH 2006) - https://steamcdn-a.akamaihd.net/apps/valve/2006/SIGGRAPH06_Course_HDRInValvesSourceEngine_Slides.pdf

Epic Games, Unreal Engine 4 Documentation – Bloom (2014) https://docs.unrealengine.com/en-US/RenderingAndGraphics/PostProcessEffects/Bloom/index.html

John Hable, Filmic Tonemapping Operators (2010) - http://filmicworlds.com/blog/filmic-tonemapping-operators/

Krzysztof Narkowicz, ACES Filmic Tone Mapping Curve (2016) – https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/

Jorge Jimenez, NEXT GENERATION POST PROCESSING IN CALL OF DUTY: ADVANCED WARFARE (2014) - http://advances.realtimerendering.com/s2014/#_NEXT_GENERATION_POST

Alexander Wilkie, Lukas Hosek, Predicting Sky Dome Appearance on Earth-like Extrasolar Worlds, (2013) – https://cgg.mff.cuni.cz/projects/SkylightModelling/sccg_2013_alien_sun_preprint.pdf

Sébastien Hillaire, Physically Based Sky, Atmosphere and Cloud Rendering in Frostbite, DICE, (2016) – https://media.contentapi.ea.com/content/dam/eacom/frostbite/files/s2016-pbs-frostbite-sky-clouds-new.pdf

A. J. Preetham, Peter Shirley, Brian Smits, A Practical Analytic Model for Daylight, (1999) - https://www.graphicon.ru/oldgr/library/siggraph/99/papers/preetham/preetham.pdf

Andrew Schneider, Nathan Vos, THE REAL-TIME VOLUMETRIC CLOUDSCAPES OF HORIZON ZERO DAWN (Siggraph, 2015) – https://www.guerrilla-games.com/read/the-real-time-volumetric-cloudscapes-of-horizon-zero-dawn

Physically Based Sky, Atmosphere and Cloud Rendering in Frostbite, (SIGGRAPH 2016) - https://blog.selfshadow.com/publications/s2016-shading-course/hillaire/s2016_pbs_frostbite_sky_clouds.pptx

Fredrik HÄGGSTRÖM, Real-time rendering of volumetric clouds (2018) - http://www.diva-portal.org/smash/get/diva2:1223894/FULLTEXT01.pdf

Eric Risser, Musawir Shah, Sumanta Pattanaik, Interval Mapping (2007) - https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.5935&rep=rep1&type=pdf

Gary McTaggart, Half-Life® 2 / Valve Source™ Shading (2004) - http://www.decew.net/OSS/References/D3DTutorial10_Half-Life2_Shading.pdf

Giliam de Carpentier, Reducing stretch in high-FOV games using barrel distortion (2015) - https://www.decarpentier.nl/lens-distortion

Adrian Courrèges, DOOM (2016) - Graphics Study (2016) - https://www.adriancourreges.com/blog/2016/09/09/doom-2016-graphics-study/

Kevin Myers, Randima (Randy) Fernando, Louis Bavoil, Integrating Realistic Soft Shadows into Your Game Engine (2008) - http://developer.download.nvidia.com/whitepapers/2008/PCSS_Integration.pdf