Real-Time Photorealistic Terrain Lighting

Naty Hoffman
Westwood Studios
Email: naty@westwood.com

Kenny Mitchell
Westwood Studios
Email: kmitchell@westwood.com

Introduction

Many games have scenes set in outdoor environments. In these environments, the most visually dominant object is the terrain. There is much published work in the game development community on efficiently generating visible terrain geometry, but relatively little on lighting it. In this talk, we will explain the various factors contributing to outdoor lighting of terrain in the real world, and give two techniques for simulating these lighting factors in real time under changing light conditions.

Outdoor Lighting - Overview

We distinguish between *direct illumination* which arrives at the terrain directly from a light source, and *indirect illumination* which is reflected onto a given point on the terrain from other points. We concentrate on daylight, though a similar approach would work at night.

The most important source of direct illumination is the sun. The sun is very distant, thus it can be treated as a directional light source. However, it is not a point source – it has a small but nonzero angular diameter (about half a degree). This causes shadows cast by the sun to have soft edges. The intensity and color of light from the sun varies depending on the hour and season. Also, cloud cover may hide the sun from view.

The sky is another important source of direct illumination. Of course, this light ultimately originates from the sun and is scattered from air molecules, water droplets, and various airborne impurities, but we shall treat it as a second light source independent of the sun. The sky is not only an area light source but a very large one, covering an entire hemisphere. The color and intensity of light emitted from the sky varies over its area at any given time, as well as varying over time. The visual effect of skylight is strongest in shadows where it is not overwhelmed by sunlight (which usually has a higher intensity).

Indirect illumination has a more subtle effect. Its intensity tends to be less than that of the direct illumination, and it tends to strike the terrain surface at a high angle of incidence (angle between the light vector and the normal) so its total contribution to the radiance reflected from the terrain is relatively low. However, this subtle contribution is important for realism.

Note on Lighting Notation and Terminology

We assume Lambertian (diffuse) terrain. In this case, the outgoing radiance (Lo) from a terrain point p is given by:

$$\text{Eq. 1: } Lo(p) = \frac{C(p)}{\pi} \int_{\vec{v} \in H(p)} Li(p,\vec{v}) \vec{N}(p) \cdot \vec{v} d\Omega$$

Where $N(p)$ is the normal vector at p, $H(p)$ is the hemisphere of outgoing unit vectors v centered on $N(p)$, $C(p)$ is the diffuse color of terrain point p, $Li(p,v)$ is the incident radiance from direction v into point p, and $d\Omega$ is an infinitesimal solid angle.

The techniques discussed here are for use with a terrain rendering system where a high-resolution diffuse color texture, $C(p)$, is modulated with a low-resolution lightmap texture. In Eq. 1, $C(p)/\pi$ is the terrain BRDF and the integral is the irradiance. $1/\pi$ is a normalization factor, and for the purposes of this talk, we will put it into the lightmap texture and compute the *normalized irradiance* which is irradiance divided by π. We denote this quantity with a lowercase l to differentiate it from radiance (*L*) and unnormalized irradiance (*E*). This gives us:

$$\text{Eq. 2: } l(p) = \frac{1}{\pi} \int_{\vec{v} \in H(p)} Li(p,\vec{v}) \vec{N}(p) \cdot \vec{v} d\Omega$$

Note that these are radiometric equations which are continuous over the electromagnetic spectrum. We sample the spectrum at three discrete frequencies for R, G and B, therefore all illumination and color quantities we use are RGB quantities.

Direct Sunlight

We will treat the sun as a simple directional light source, except for the possibility of partial occlusion. Then the sun's direct contribution to the irradiance at each terrain point p ($l_{SunDirect}(p)$) is given by:

$$\text{Eq. 3: } l_{SunDirect}(p) = O_{Sun}(p) Li_{Sun} \vec{N}(p) \cdot \vec{v}_{Sun}$$

Where $O_{Sun}(p)$ is the sun's visibility/occlusion factor at p (1: completely visible, 0: completely occluded), and v_{Sun} is the outgoing sunlight direction vector.

Direct Skylight

The sky is a hemispherical diffuse light source, where the emitted color and intensity varies over the hemisphere. To calculate the skylight for a given terrain point p, we must integrate incoming light over all directions in the set $D(p)$ (the subset of $H(p)$ which is not occluded by other terrain points). Then the sky's direct contribution to the irradiance at each terrain point p ($l_{SkyDirect}(p)$) is given by:

$$\text{Eq. 4: } l_{SkyDirect}(p) = \frac{1}{\pi} \int_{\vec{v} \in D(p)} Li_{Sky}(\vec{v}) \vec{N}(p) \cdot \vec{v} d\Omega$$

Indirect Illumination

Indirect illumination of the terrain is the result of interreflections of light between different terrain points. To fully calculate these, a global illumination algorithm such as radiosity is needed. Such algorithms cannot be executed in real-time. However, it may be possible to approximate indirect light using a much faster algorithm.

Technique A: Real-Time Analytical Computation of Outdoor Lighting

This technique makes use of several simplifying assumptions, approximations and precomputed data to calculate a complete outdoor lighting solution in real-time.

We separate the outdoor lighting into two separate problems: sunlight alone and skylight alone, solve each, and combine the two solutions to get the total lighting solution. This is used to update a lightmap texture every frame. This texture is relatively low-resolution, and it is updated in a multiresolution fashion (the update resolution is lower farther away from the camera) so the amount of texture uploaded every frame is not excessive.

This technique works with changing lighting conditions, and the amount of precomputation required is relatively small (taking about 6 seconds on a PII-450 for a 512x512 lightmap).

Sunlight Alone:

We assume that the sun travels in a zenithal arc (a circular arc which passes directly overhead at its highest point). For each texel in the lightmap, we precompute and store horizon angles in the same plane as the sun's arc, thus creating a *horizon map* (first introduced by Max in [1]). The horizon angles (see Fig. 1) are scaled so that a quarter-circle fills the [0, 1] range, and stored as 16-bit fixed-point numbers.

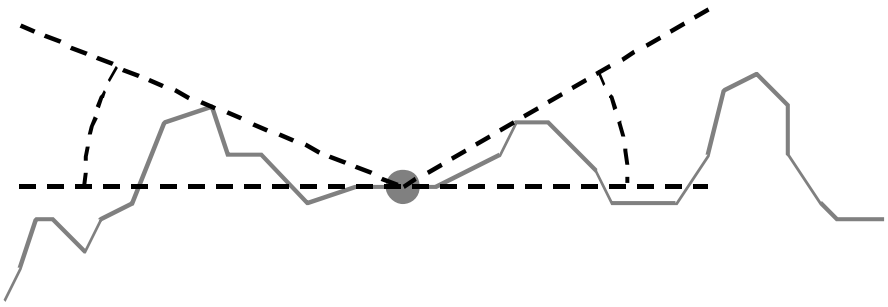

Fig. 1: Horizon angles

Note on horizon angles: all horizon angles for a point p must be clamped to $H(p)$, the hemisphere of outgoing directions centered on the surface normal. The reason is that the radiosity equation is only linear within $H(p)$– outside it a light's contribution is 0. This "effective horizon" is illustrated in Fig. 2.

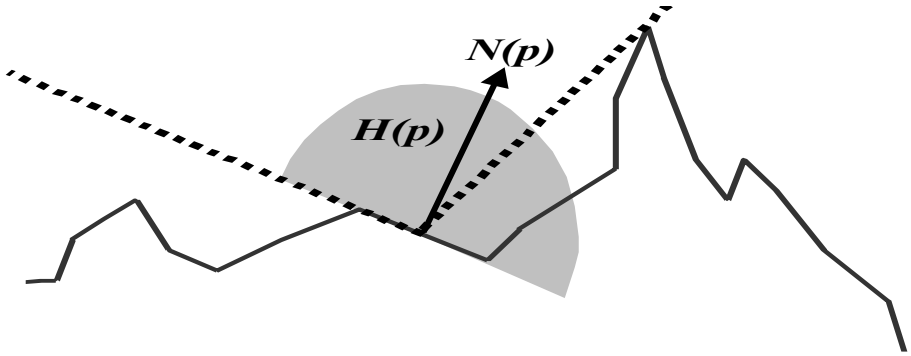

Fig. 2: Effective horizon

In another preprocessing step, the surface normal is calculated for every texel in the lightmap, and then quantized to a table of 256 normals. The index into this table is stored for every lightmap texel.

Then at runtime, as the sun's position and color changes, we calculate maximum and minimum angles each frame:

$$\text{Eq. 5: } \theta_{Min} = \theta_{Center} - \frac{1}{2}\alpha$$

$$\text{Eq. 6: } \theta_{Max} = \theta_{Center} + \frac{1}{2}\alpha$$

Where θ_{Center} is the elevation angle of the center of the sun's disk, and α is the angular diameter of the sun (about ½ degree, though to achieve pleasing soft-shadow effects, a larger quantity can be used – soft shadow edges also have a useful antialiasing effect on the low-resolution lightmap).

Each frame we also calculate the RGB illumination for each normal in the normal table (by taking the dot product between the sun vector and the normal, and multiplying by the current sun color). This is stored in a precalculated lighting table.

Then we loop over the lightmap texels which need to be updated, calculating the sunlight's contribution to each one by looking up the precalculated lighting using the normal index, and multiplying it by the visibility/occlusion factor. This factor is calculated by comparing the sun's minimum and maximum angles with the appropriate horizon angle. If the horizon angle is smaller than the sun's minimum angle, the factor is 1. If it is larger than the sun's maximum angle, then the factor is 0. If it is between the min and max angle, the factor is calculated by:

$$\text{Eq. 7: } O_{Sun} = \frac{\theta_{Max} - \theta_{Horizon}}{\theta_{Max} - \theta_{Min}}$$

Skylight Alone:

We divide the sky into a small number of patches and assume a constant color for each patch (in practice, effective results can be achieved with just two patches, or even one patch in some cases). We precalculate for each lightmap texel and each sky patch the contribution of that sky patch to that texel, assuming that the patch's color is (1,1,1). During runtime, we loop over all lightmap texels, multiply the current patch colors by the precalculated contribution factors, add the resulting skylight to the sunlight (calculated at the same time as described in the previous section), and write the result to the lightmap.

The precalculation uses Eq. 4 for direct skylight. This equation integrates incoming radiance over the visible sky area. However, no integration is performed over those directions occluded by terrain. This means that Eq. 4 alone will produce overly dark results – lighting each terrain point as if the terrain surrounding the point was a perfect matte black which reflects no light. In theory, the solution to this problem would involve computing the actual reflected light from the surrounding terrain using some type of iterative global illumination solution. If long precomputation times are not a problem, then this is a perfectly valid approach. However, our game required short precomputation times, which ruled this out.

Fortunately, Stewart and Langer [2] have found a good cheap approximation for global illumination under diffuse lighting conditions (this approximation, and its use for terrain rendering is also discussed by Stewart in [3]). The basic idea is that under conditions of diffuse lighting, each point tends to "see" points which have similar lighting to itself – the terrain points visible from a point in a dark valley tend to also be in a dark valley, points visible from a bright mountain peak tend to also be bright mountain peaks, etc. Therefore, when lighting a terrain point, we can assume that all other visible terrain points have the same radiance as the point we are currently lighting. With some algebra, this yields a nice closed-form expression taking interreflections as well as direct skylight into account. This approximation has been tested and the resulting errors have been measured and shown to be quite small [3]. The original papers assumed a monochromatic terrain of constant albedo, but the same principle generalizes to multicolored terrain: terrain points will tend to see other terrain points of the same terrain type (forest, snow, etc.) so they would have the same color. Small-scale texture details do not matter since our lighting solution is of much lower resolution than the terrain textures.

Expressed mathematically, this approximation gives the following equation:

Eq. 6: $$l_{Sky}(p) = \frac{1}{\pi} \int_{\vec{v} \in D(p)} Li_{Sky}(\vec{v}) \vec{N}(p) \cdot \vec{v} d\Omega + \frac{1}{\pi} \int_{\vec{v} \in H(p) \setminus D(p)} Lo_{Sky}(p) \vec{N}(p) \cdot \vec{v} d\Omega$$

(Note: the following derivation is from Stewart and Langer [2] – see that reference for more detail). We will now assume a single constant sky color (one patch – later we will show how to handle multiple patches), making Li_{Sky} a constant. We also substitute $C(p)l_{Sky}(p)$ for $Lo_{Sky}(p)$ (using the average texture color of the surface region covered by the lightmap texel for $C(p)$). Then some simple algebra gives us:

$$\text{Eq. 7: } l_{Sky}(p) = \frac{Li_{Sky}\frac{1}{\pi}\int_{\vec{v}\in D(p)}\vec{N}(p)\cdot\vec{v}\,d\Omega}{1-C(p)\left(1-\frac{1}{\pi}\int_{\vec{v}\in D(p)}\vec{N}(p)\cdot\vec{v}\,d\Omega\right)}$$

We need to compute a factor (we will call it f_{Sky}) which we will multiply by Li_{Sky} to get the final result. We can further simplify the expression by substituting $K(p)$ for the repeated portions:

$$\text{Eq. 8: } f_{Sky}(p) = \frac{K(p)}{1-C(p)\left(1-K(p)\right)}$$

$$\text{Eq. 9: } K(p) = \frac{1}{\pi}\int_{\vec{v}\in D(p)}\vec{N}(p)\cdot\vec{v}\,d\Omega$$

Now we need to calculate $K(p)$. First, we need to express the visible sky area $D(p)$. A horizon map [1] is ideally suited to this purpose as well. Instead of just two directions, now we compute the clamped horizon angles in the eight cardinal directions (N, NW, W, SW, S, SE, E, NE). (two of these will also be used for sunlight as described above). These horizon angles are relatively cheap to calculate since we only need to scan along the rows, columns and diagonals of the heightfield. The two directions which are also used for sunlight need to be fairly accurate, thus requiring scans over long distances – for better performance this direction should correspond to the rows of the heightfield data. The other six directions can be generated with relatively short scans.

For expressing $D(p)$, we divide the horizon into eight sectors numbered 0 to 7, so that sector *i* spans the azimuth angle range $[(\pi/4)i, (\pi/4)(i+1)]$. We will use the appropriate horizon angle (measured from the vertical this time) as the elevation angle φ_i for each sector. Based on these, Stewart and Langer[2] give us an expression for $K(p)$:

$$\text{Eq. 10: } K(p) = \frac{1}{2\pi}\vec{N}(p)\cdot\sum_{i=0}^{7}\left(\left(\varphi_i-\frac{\sin 2\varphi_i}{2}\right)\Delta\sin_i \quad \left(\varphi_i-\frac{\sin 2\varphi_i}{2}\right)\Delta\cos_i \quad \frac{\pi}{4}\sin^2\varphi_i\right)$$

$$\text{Eq.11: } \Delta\sin_i = \sin\left(\frac{\pi}{4}(i+1)\right)-\sin\left(\frac{\pi}{4}i\right)$$

$$\text{Eq.12: } \Delta\cos_i = \cos\left(\frac{\pi}{4}i\right)-\cos\left(\frac{\pi}{4}(i+1)\right)$$

Note: the three expressions inside the sum in Eq. 10 are the components of a 3-vector. The sum will produce a vector, then the dot-product between this vector and $N(p)$ is calculated and the result multiplied by a constant. Note also that $\Delta\sin_i$ and $\Delta\cos_i$ are constants and can be computed once and reused.

Since each vector being summed depends only on φ_i, we can precalculate this vector for each sector and for 256 values of φ_i resulting in a 256x8 table. Note that better accuracy is achieved

if we use the tangent of the angle for the table lookup (the tangent of the angle is calculated easily when computing horizon angles).

After K(p) is computed, we calculate the skylight factor (an RGB value) and store it for each lightmap texel. This is multiplied with the current sky color at runtime to get the skylight contribution. If we want more than one sky patch, we can assign different sectors to different patches to get up to eight sky patches (though less would probably suffice). In this case, a skylight factor needs to be computed and stored for each patch. Elevation angles can also be used to separate patches if wished – for example each sector could be separated into two regions (0° to 45° and 45° to 90°). In the multiple-patch case, the patch color needs to be multiplied by each patch's factor and the results added up to get the total skylight contribution.

Previous Work and Future Directions

Currently, this technique calculates and uploads lightmaps each frame. It is tempting to use hardware to generate the lightmaps instead, using texture blending techniques (such as D3D8 pixel shaders) and multiple passes. The skylight contribution is fairly simple – each patch's factors can be stored as an RGB texture, and we can just render each texture (modulated with the current patch color) additively to add them up. The sunlight contribution is a little more complicated. DP3 blending can do the diffuse lighting, and if we drop soft shadows then a simple alpha test could handle the shadowing. These possibilities will be further discussed in the talk, time permitting.

Weather can be somewhat simulated by tweaking the sun and sky colors – for example a heavily overcast day will have the sun set to zero and a uniform gray sky color. Local cloud shadows can also be simulated by having an additional texture, projected from above, which modulates the sunlight contribution.

Comparisons to previous work – this basic concept (horizon maps) derives from Max [1]. The details owe much to Stewart and Langer's work [2, 3], though there are some minor differences in implementation. For example, we have more restrictions on our lighting environment and use a simpler technique to compute horizon maps. Heidrich, Daubert, Kautz, and Seidel [4] have an approach for hardware shadowing and indirect lighting of bump maps which works well with terrain. In handles a variety of BRDFs and is more flexible in terms of light direction. However, it is fairly expensive, more general than we need (given the restrictions on outdoor lighting in our game), and does not handle sky lighting. It is a very powerful technique nonetheless, and we plan to investigate the possibility of combining ideas from both approaches. This could potentially yield more realism (for example, it might be possible to take account of the indirect contribution of sunlight) or allow us to shadow local light sources. Sloan and Cohen [5] have a bump map shadowing method using horizon maps which also supports lights in arbitrary directions – it is more general (and expensive) than simple horizon map shadowing. We can probably use ideas from [5] in creating a hardware implementation of our technique. Sloan and Cohen's work might also suggest ways to shadow local lights.

Technique B: Video Based Illumination Maps

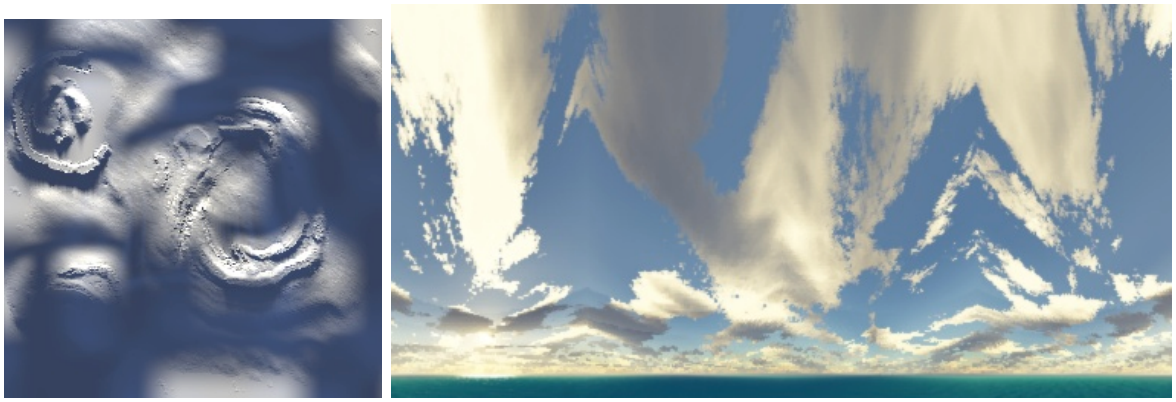

Fig. 3: A single frame of terrain lightmap and sky dome textures from a video based illumination map (VBIM) sequence.

Introduction

As an alternative method for terrain lighting, an image-based solution is distinct from the traditional analytical solution in that lighting in the environment can be derived from real photographs. This can lead to faster real-time rendering performance and potentially the highest levels of photorealism. In this section, we provide explanations and examples of using images to illuminate outdoor environments, including the fundamental concepts of using sampled light in scenes and the practical use of light maps and environment maps in real-time rendering.

With increasing support in graphics hardware for compressed textures and video texture playback, animation of image-based lighting methods can be achieved. We detail and demonstrate the use of graphics hardware for the efficient playback of video-based illumination maps. In particular, we show how the image processing unit (IPU) of Sony's Playstation 2 is employed for this purpose.

The capture of lengthy animation sequences on location from multiple camera view points can be impractical and expensive. However, advanced photorealistic rendering software can produce equally satisfying results without leaving the comfort of the desktop. We highlight the benefits and potential pitfalls of automating the use of such software for the production of video-based illumination maps.

Terrain Illumination Maps

Fig. 3 shows a single frame from a video sequence applied as a single lightmap texture pass to the terrain. In this color image the effects of self-shadowing, cloud shadows, sunlight, skylight, atmospheric blue effects, atmospheric scattering, and haze can be seen. All the calculations for these effects are pre-computed in the game’s asset creation process from raw

height-field data using a high quality terrain rendering application, Terragen[1]. In addition to streaming lightmaps, a matching sky dome video texture is streamed.

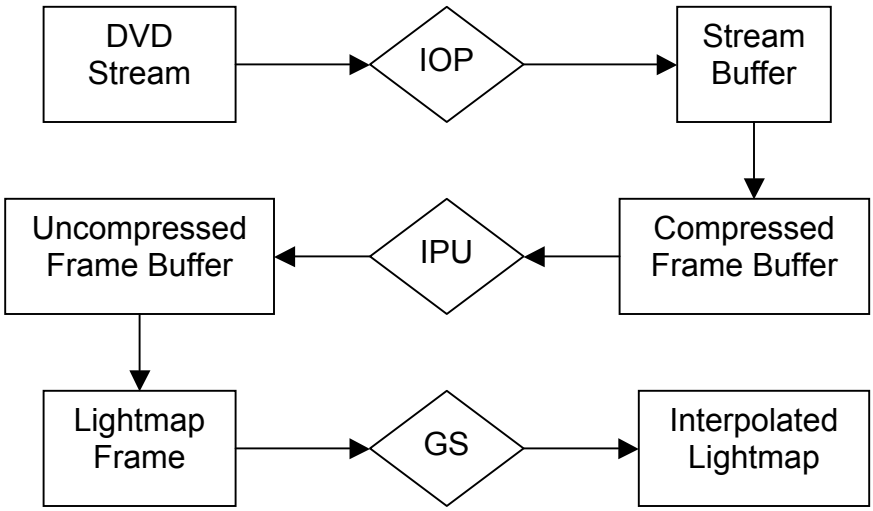

Fig. 4: Streaming Video Data Flow.

Video Streaming Hardware Implementation on Playstation 2

To animate terrain lighting a number of challenges must be overcome. This section describes the data flow of a single frame of the animation from a compressed video stream stored on DVD to the final rendered lightmap.

Concurrently with the application, compressed frames are streamed into the stream buffer in RAM, using the IO processor (IOP), in much the same way as sound or DVD video is transferred. Once a frame has been fully loaded into the stream buffer it is copied into the compressed frame buffer and loading of the next frame begins immediately. Compression is necessary to reduce the data rate to within the required limits of DVD media.

The first decompression stage occurs in the image processing unit (IPU), where it processes each frame to produce an uncompressed frame in RAM suitable for upload to VRAM.

Blending the new image with the previous image provides the second stage of decompression. This frame interpolation method occurs through the use of the graphics synthesizer (GS) and is performed in place by using a frame buffer motion blur technique [6] to reduce VRAM use.

[1] Terragen is free for non-commercial use and may be downloaded from http://www.planetside.co.uk/terragen. Commercial use is permitted for registered users.

This second decompression stage is important for a number reasons,

- It acts as a form of temporal anti-aliasing between compressed frames, which reduces the number of full frames required for smooth animation.
- The interpolation of frames avoids sudden changes when the looping video jumps to the beginning of the sequence.
- If DVD streaming is held up for any reason, the frame interpolation process will continue unhindered. When the next frame is finally loaded the same interpolation process will produce smooth ‘catch up’ frames and resume the video sequence as normal.

Illumination Map Generation vs. Real Image Capture

Ideally, VBIMs would be captured from real video camera footage. One idea for capturing the light field of a real terrain could be to place light sensors at regular interval in a grid at ground level. Another could be to extract this information from geo stationary satellite image data. Realistically, the logistical problems of setting up these situations make the real image capture method entirely impracticable for game production. Although, sky dome/box capture is less problematic it is just too time consuming to wait for the perfect sunset or the perfect storm. An artist generated illumination map and sky box could produce the same results in minutes given sufficiently sophisticated rendering tools.

Future Directions

With the increased realism of terrain lighting, objects that are placed in this environment must also match the local lighting conditions. A more sophisticated and efficient method for resolving this needs to be investigated.

For frame buffer blending and compression efficiency reasons, the lightmap is generated without reference to terrain colors. Terrain lightmaps, which take into account color radiosity calculations, may provide a further level of realism.

On PC, some 3D graphics boards are emerging with hardware video decompression suitable for applying to textures. This may open the way for this technique in future PC games.

Controlled changes in weather conditions could be simulated by branching to alternative video streams. The frame interpolation method would permit smooth transitions between these states.

References

[1] N.L. Max. *Horizon Mapping: Shadows for Bump-Mapped Surfaces.* The Visual Computer, 4(2):109-117, July 1988.

[2] A.J. Stewart., M.S. Langer. *Towards Accurate Recovery of Shape from Shading under Diffuse Lighting.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 19(9):1020-1025, Sept. 1997

[3] A.J. Stewart. *Fast horizon computation at all points of a terrain with visibility and shading applications*. IEEE Transactions on Visualization and Computer Graphics, 4(1):82--93, March 1998

[4] W. Heidrich, K. Daubert, J. Kautz, and H.-P. Seidel. *Illuminating micro geometry based on precomputed visibility*. Computer Graphics(Proceedings of SIGGRAPH 2000):455-464, July 2000

[5] P.-P. Sloan, M.F. Cohen. *Interactive Horizon Mapping.* Rendering Techniques 2000(Proceedings of Eurographics Rendering Workshop 2000): 281-298, June 2000

[6] K. Mitchell. *Real-Time Full Scene Anti-Aliasing for PCs and Consoles*. Proceedings of Game Developer Conference, March 2001.