Lighting Effects for Mobile Games

Jeppe Revall Frisvad[*] Niels Jørgen Christensen[†] Peter Falster[‡]

Informatics and Mathematical Modelling
Technical University of Denmark

Abstract

Adapting games to the miniature screens and limited processing power of mobile phones is quite a challenge. To accommodate these technical limitations, mobile games are often two-dimensional (2D), tile-based, and seen from above. Such games rarely present much creativity with respect to dynamic lighting. Textures are merely plastered onto the tiles. A recent game release has, however, shown that effects such as dynamic light sources can spice up a mobile game of this type quite a bit. In this paper we carry the idea of dynamic lighting effects for tile-based 2D games even further. In particular, we present simple and efficient techniques for shadows and fluctuating fog which can greatly improve the gamer's visual experience.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: lighting effects, mobile games, shadows, fog synthesis, real-time.

1 Introduction

Mobile games rarely feature dynamic lighting effects. An obvious reason is the limited processing power available. Another reason is the different gaming experience that one encounter when playing a game on the small screen of a mobile phone. Since the gaming experience is different, the gameplay designed for a mobile game should be adapted accordingly, see [Nok 2003].

On a small screen the player quickly looses track of game characters and objects. A solution is to make games in a style where everything is seen from above. This ensures the best possible overview. It, therefore, seems reasonable to assume that two-dimensional, tile-based games viewed from above always will be relevant on the miniature screens of mobile phones. For that reason, we propose some simple methods enabling dynamic lighting effects in games of this type.

Quite recently the mobile game Darkest Fear™ from Rovio Mobile was shipped as the first mobile game marketing itself on its ability to incorporate dynamic lighting effects in its gameplay, see [Rov 2005]. And it received an Airgamer Award from Germany's leading mobile game reviewer with the review punch line: "Innovative and exciting!" [Biedermann 2005]. This strongly indicates that lighting effects are worth the effort in this category of games.

[*]e-mail: jrf@imm.dtu.dk
[†]e-mail: njc@imm.dtu.dk
[‡]e-mail: pfa@imm.dtu.dk

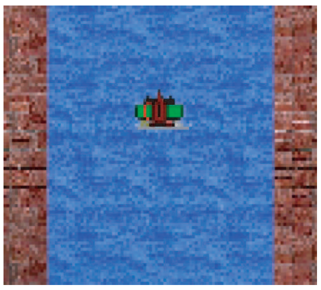


Figure 1: The gamelike scenario we will be working with.

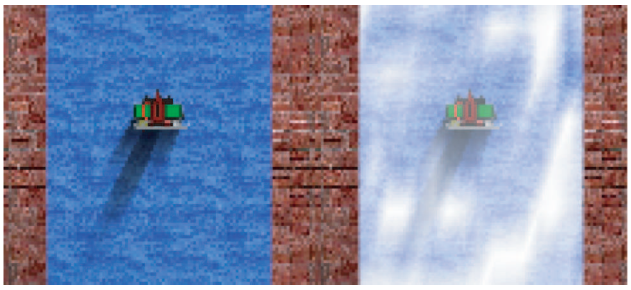


Figure 2: Examples of the lighting effects we will describe efficient methods for in this paper.

Darkest Fear has lighting effects such as movable light sources providing an attenuating illumination of the surroundings. Moreover several light sources are turned on and off dynamically. The shadows of the dynamic characters are merely blobs, which are a part of the sprites[1] associated with a character.

In this paper we suggest a simple technique to do dynamic shadows in tile-based games. Moreover we give a simple technique to create fluctuating fog in such games.

Figure 1 presents the central part of a sample frame from the gamelike scenario we will be working with in this paper. This tile-based demo scenario is compiled for a Windows PC platform, but could as well have been compiled for a mobile phone platform if we had had the necessary development tools at our disposal. The scenery is viewed from above, as we argued is often sensible in mobile games. In Figure 1 no lighting effects have been applied. Compare this example to the images in Figure 2, where the sample frame has been spiced up with the inexpensive lighting effects to be described in the following sections. Hopefully the reader agrees that such effects make the scene more interesting.

The lighting effects we present have, clearly, been done before in games for PCs and consoles. Here the environment is, however, most often three-dimensional which means that the employed shadow and fog algorithms are slightly different from the ones we describe. The types of 2D games which have a third dimension that is always projected on the same plane, give us an option for inex-

[1]A sprite is basically a rectangular pixel map.

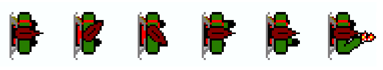


Figure 3: Six sprites for animation of the character in our gamelike environment. The character is a Robin Hood-like person wearing a hat and having a bow over his back pack. He is seen directly from above.

pensive calculation of 3D lighting effects. In the following we will describe how this is done.

2 Lighting Effects

Jim Blinn is one of the pioneers who introduced several lighting effects for three-dimensional environments in the seventies and eighties. Some of his ideas were planar projection shadows, see [Blinn 1988], and rendering of cloud cover, see [Blinn 1982]. Inspired by these ideas, we describe, in the following, how shadows and fog (or cloud cover) effects can be incorporated in two-dimensional games.

The general question to be answered in the following, is how to create a third dimension in an otherwise two-dimensional gaming environment. And, after this information has been provided, how can we construct lighting methods that are not processing intensive. To provide answers, we must take advantage of the two-dimensional nature of the environment.

2.1 Planar Projection Shadows

Consider the sprites for the character of our game, see Figure 3. The sprites have a resolution of 32×32. To find shadows cast by this character, we must provide some height information.

To keep memory costs low, we decided to make two height curves. One along each axis in the character sprites, see Figure 4. Let ℓ denote the number of pixels a height curve covers. In our case $\ell = 32$. A pair of curves could be constructed for each character sprite, but if the difference between the sprites is subtle, there is no need to have more than a single pair of curves for each character.

The curves are positioned above the character and the curve along the axis making the largest angle with the direction towards the light source, is used for calculation of the shadow. Figure 5 illustrates how the shadow outline is found and can be referred to throughout the remainder of this section.

Each light source in the game should have a 3D position:

$$L = (x_L, y_L, h_L)\,,$$

where h_L is the height of the light source above the xy-plane where everything is rendered.

Suppose we let $C = (x_C, y_C)$ denote the position where the center of the sprite will be drawn next. The 2D direction $\boldsymbol{d}_L$ towards a light source is then given as:

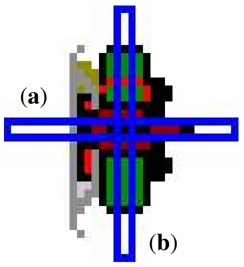
$$\boldsymbol{d}_L = (x_d, y_d) = (x_L, y_L) - C\,.$$

To decide which curve we want to use for the shadow (**a** or **b** in Fig. 4), we can calculate:

$$\cos\phi_1 = \frac{\boldsymbol{d}_L \cdot \boldsymbol{e}_x}{\|\boldsymbol{d}_L\|} = \frac{x_d}{\sqrt{x_d^2 + y_d^2}}\,, \tag{1}$$

where $\boldsymbol{e}_x = (1,0)$ is the direction of the x-axis.

When $\phi_1 \in [\frac{\pi}{4}, \frac{3\pi}{4}]$ or $\phi_1 \in [\frac{5\pi}{4}, \frac{7\pi}{4}]$ we want to use curve (**a**) otherwise curve (**b**). This corresponds to saying that when $\cos\phi_1 \in [-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}]$ we use curve (**a**) otherwise curve (**b**). Note that there



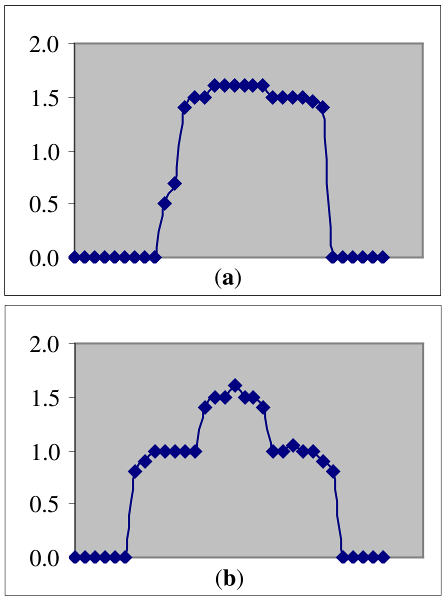


Figure 4: An illustration of the height curves for our character.

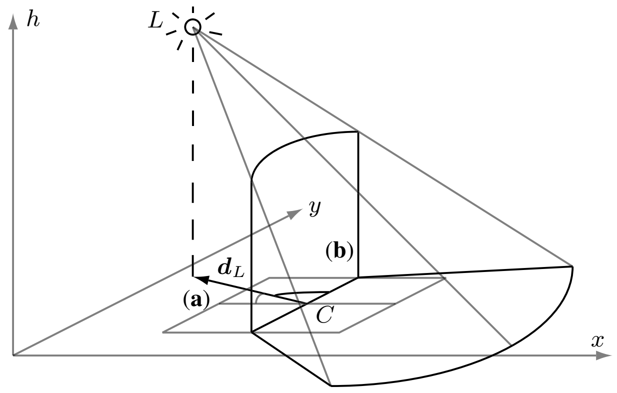


Figure 5: A sketch illustrating how the shadow outline is found. Here curve (**b**) is the chosen curve, since it has the largest angle with the direction towards the light source.

is no need to find the angle ϕ_1 itself as long as the sprites are not being rotated.

It is, however, desirable to rotate the sprites, since then we only need to store them facing a single direction. In our game, an angle $\phi_0 \in [0, 2\pi]$ specifies how much the character sprite has been rotated around its center[2]. This complicates the matter a bit. To find out which curve to use in this case, we calculate

$$\phi = \phi_0 - \operatorname{sign}(y_{\boldsymbol{d}}) \cos^{-1}\left(\frac{x_{\boldsymbol{d}}}{\sqrt{x_{\boldsymbol{d}}^2 + y_{\boldsymbol{d}}^2}}\right) , \qquad (2)$$

and again, if $\phi \in [\frac{\pi}{4}, \frac{3\pi}{4}]$ or $\phi \in [\frac{5\pi}{4}, \frac{7\pi}{4}]$, we use curve (**a**) otherwise curve (**b**).

Note, in (2), that $\cos\phi_1$ does not supply us with sufficient information to determine ϕ_1. We need to consider the sign of $y_{\boldsymbol{d}}$ to get $\phi_1 \in [0, 2\pi]$. It is also important to realize that ϕ_1 should be subtracted from ϕ_0. This follows since ϕ is the angle between the local x-axis of the sprite and the direction towards the light source. The local x-axis has been rotated to have the angle ϕ_0 with the global x-axis and ϕ_1 is the angle between the global x-axis and the direction towards the light source, hence, $\phi = \phi_0 - \phi_1$.

Depending on whether the character sprites are being rotated in the game or not, either (2) or (1) determines which curve to be used for the projection of the shadow. Each curve should have a direction vector and an origin. The direction of curve (**a**) is $\boldsymbol{d}_{\mathbf{a}} = (w_T/\ell, 0)$ and the direction of curve (**b**) is $\boldsymbol{d}_{\mathbf{b}} = (0, h_T/\ell)$, where w_T and h_T are, respectively, the width and height of a tile in world coordinate units[3]. The needed origin, either $P_{\mathbf{a}}$ or $P_{\mathbf{b}}$, is found according to the current position of the sprite.

Let $\boldsymbol{d}$ and P denote the direction and origin, respectively, of the chosen curve. If the sprite has been rotated, $\boldsymbol{d}$ and P should be rotated accordingly. Each height value on the chosen curve now has a corresponding position:

$$(x_{0,i}, y_{0,i}) = P + i\,\boldsymbol{d} \quad , \quad i = 0, \dots, \ell - 1 \,.$$

Combining the $(x_{0,i}, y_{0,i})$-position with the height value $h_{0,i}$ stored for each position on the curve (see again Fig. 4) gives 3D points $P_{0,i} \in \mathbb{R}^3$, $i = 0, \dots, \ell - 1$, along the curve. When those have been established, the direction $\boldsymbol{d}_i$ from the light source to P_i is found as

$$\boldsymbol{d}_i = P_{0,i} - L$$

and intersection with the $\boldsymbol{xy}$-plane can be calculated. This is a simple calculation, since we are in possession of the parametric equation of the line with origin at the position of the light source, L, and direction $\boldsymbol{d}_i$ through the point on the curve $P_{0,i}$:

$$(x_i, y_i, h_i) = L + t_i\,\boldsymbol{d}_i \,. \qquad (3)$$

When $h_i = 0$, the line intersects the xy-plane. Hence, we quickly discover that

$$0 = h_L + t_i(h_{0,i} - h_L) \quad \Leftrightarrow \quad t_i = \frac{h_L}{h_L - h_{0,i}} \qquad (4)$$

finds the value of t_i which, inserted in (3), gives the projection of the ith point on the height curve to the xy-plane in the direction away from the light source. In other words, the points

$$(x_i, y_i) = (x_L, y_L) + \frac{h_L}{h_L - h_{0,i}}(x_{0,i} - x_L, y_{0,i} - y_L) \qquad (5)$$

and $(x_{0,i}, y_{0,i})$ with $i = 0, \dots, \ell - 1$, defines the outline of the shadow in the xy-plane (see again Fig. 5).

[2]In fact we use integers and degrees (from $0°$ to $359°$) for the angles not radians, but that is not essential for the description of the concept.

[3]It is sensible and, of course, convenient to let $w_T = h_T = 1$.

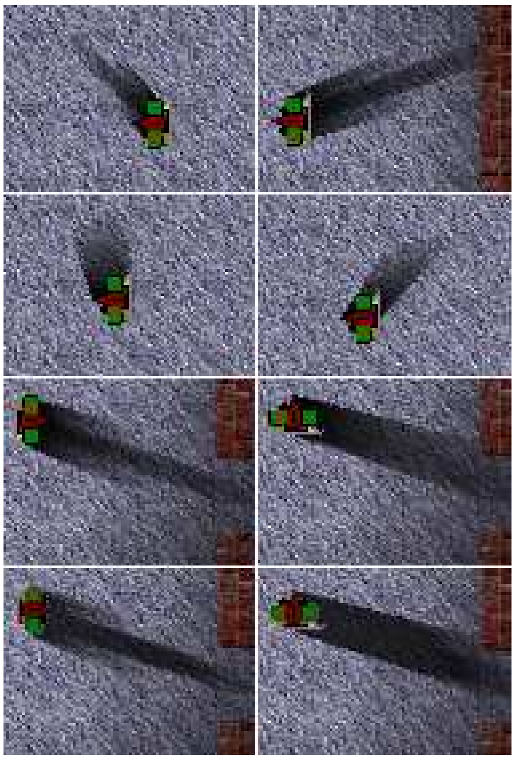


Figure 6: Shadows resulting when the described technique is employed in our case study. In the first three rows the shadow has been rendered with alpha values interpolated over a single triangle strip. For comparison the last row has been rendered using two triangle strips: One representing the umbra region and one representing the penumbra region. The first with constant alpha values, the second with interpolated alpha values. While the difference is subtle, the advantage of the second approach is that we can control the size of the penumbra region.

Having the shadow outline, the shadow can be visualized in many different ways. With a slight abuse of notation let $P_{0,i} = (x_{0,i}, y_{0,i})$, and let $P_i = (x_i, y_i)$ both with $i = 0, \dots, \ell - 1$. We simply draw the shadow as a triangle strip using the points $P_{0,0}, P_0, P_{0,1}, P_1, \dots, P_{0,\ell-1}, P_{\ell-1}$. To give a touch of softness to the shadows, we make the points P_i quite transparent ($\alpha = 0.1$) while the points $P_{0,i}$ are left opaque ($\alpha = 1$). Standard linear interpolation of the alpha values between the vertices of each triangle (Gouraud shading, see [Gouraud 1971]) is performed[4] when the triangle strip is rendered. Examples from our case study of the resulting shadows are given in Figure 6.

Linear interpolation of the alpha values is physically incorrect, but seems visually more pleasing than a hard shadow. If distinguishable umbra and penumbra regions are desired, it is necessary to render the shadow as two triangle strips. First we project the height curve to a plane which is moved a constant c closer to the light source. The constant determines the size of the penumbra regions and the shadow outline found at the plane moved closer to the light source will describe the umbra region (the idea to move the intersection plane closer to the light source was proposed by Gootch et al. [1999] for rendering of planar soft shadows in 3D environments). The projected points $P_{u,i} = (x_{u,i}, y_{u,i})$, $i = 0, \dots, \ell - 1$ are found using (3) with $t_{u,i} = t_i - c$ when $t_i - c \geq 1$ otherwise

[4]In OpenGL and OpenGL ES, Gouraud shading happens automatically when the smooth shading model has been chosen.

$t_{u,i} = 1$. Now the points $P_{0,0}, P_{u,0}, \ldots, P_{0,\ell-1}, P_{u,\ell-1}$ can be used to draw the triangle strip for the umbra region with a constant alpha value, eg. $\alpha = 0.75$.

The second strip is drawn using the points $P_{u,0}, P_0, \ldots, P_{u,\ell-1}, P_{\ell-1}$, where the points P_i, are the ones found in (5). The points $P_{u,i}$ should be rendered with the same alpha value as before, but the points P_i should be rendered with $\alpha = 0$. A few examples of the shadows resulting from this two strip method are given in Figure 6 for comparison with the method using one strip.

2.2 Fluctuating Fog

The second lighting effect that we would like to describe a simple rendering technique for, is fog and similar semi-transparent phenomena. One way to do fog is simply to have a texture with varying shades and transparencies (alpha values). The texture is then spread over the tiles that should be foggy. This is, however, expensive with respect to memory storage and the result is a static fog which does not give much feeling of realism.

Instead we suggest that a height field composed of a 16×16, 10×10, or an even smaller grid of height values should be sufficient to generate an interesting fluctuating fog.

Suppose the fog is to be spread across a rectangular area in the game specified by three 2D points: Q_0, Q_{top}, and Q_{right}. Let ℓ_x and ℓ_y denote the resolution of the height field grid. Then the vectors used to step through the height field are

$$\boldsymbol{v}_x = \frac{Q_{\text{right}} - Q_0}{\ell_x} \quad \text{and} \quad \boldsymbol{v}_y = \frac{Q_{\text{top}} - Q_0}{\ell_y} .$$

The 2D position of each vertex in the height field is then found as

$$(x_{ij}, y_{ij}) = Q_0 + i\,\boldsymbol{v}_y + j\,\boldsymbol{v}_x ,$$

where $i = 0, \ldots, \ell_y - 1$ and $j = 0 \ldots, \ell_x - 1$. This is combined with the values h_{ij} available in the height field to obtain the 3D positions $Q_{ij} = (x_{ij}, y_{ij}, h_{ij})$.

To have our fog change appearance depending on the viewpoint of the player (that is, to create fluctuations), we need normal information as well as positions. For each vertex in the height field we can calculate two basis vectors $\boldsymbol{b}_{1,ij}$ and $\boldsymbol{b}_{2,ij}$ using one neighboring point in the x direction and one neighboring point in the y direction. Normalized cross products of the two basis vectors at each vertex then provides the normals:

$$\boldsymbol{n}_{ij} = \frac{\boldsymbol{b}_{1,ij} \times \boldsymbol{b}_{2,ij}}{\|\boldsymbol{b}_{1,ij} \times \boldsymbol{b}_{2,ij}\|} .$$

Depending on the processing power available versus the memory available, normals or positions could be stored in memory or recalculated as one thinks fit. If the fog is supposed to move around in the game, the positions must, of course, be recalculated, but the normals could still be kept static. Normals should be recalculated if the height values change.

To render the fog, the position of the eye V must be available. In our game, the eye is positioned directly above the controlled character which is always centered. Centering the character controlled by the player is one of the good advises in [Nok 2003]. This gives the player a better chance to follow the game on a small screen.

The angle θ_{ij} between the direction towards the viewpoint, V, and the normal, $\boldsymbol{n}_{ij}$, at each vertex is now used to attenuate the light transmitted directly through the fog. The formula calculating attenuation of directly transmitted light is well known, see eg. [Chandrasekhar 1960]. Assuming that the fog medium has a constant extinction coefficient σ_t throughout, and that the fog goes all the way to the xy-plane, the attenuation computation is given as

$$\alpha_{ij} = e^{-\tau_{ij}/|\cos\theta_{ij}|} , \tag{6}$$

where $\tau_{ij} = \sigma_t h_{ij}$ is the optical depth of the fog below the vertex. We do not have to worry too much about the exact meaning of the extinction coefficient in this context. It suffices to think of σ_t merely as a scale: When it increases the fog becomes more dense. To calculate $\cos\theta_{ij}$ we have

$$\cos\theta_{ij} = \frac{V - Q_{ij}}{\|V - Q_{ij}\|} \cdot \boldsymbol{n}_{ij} . \tag{7}$$

Left to find is a shade for the fog at each vertex. In our opinion the height values scaled such that they live in $[0, 1]$ gives acceptable grey shades for a fog in dark surroundings.

The fog is now ready to be rendered on top of the tiles it was spread across. To do the blending of the fog with the scenery below it, we call the grey shade, found for each fragment as an interpolation of the scaled height values, the *source* and denote it L_{src}. The *destinations* are the existing colors L_{dst} of the fragments. The blending should be done such that

$$L_{\text{blend}} = L_{\text{src}} + \alpha_{\text{src}} L_{\text{dst}} .$$

Some resulting images from our case study are presented in Figure 7. If we want a white fog, corresponding to a cloud cover lit from above by the sun, we do not use grey shades, but set all color values (except the alpha value, of course) to 1 and use the following function for blending:

$$L_{\text{blend}} = (1 - \alpha_{\text{src}}) L_{\text{src}} + \alpha_{\text{src}} L_{\text{dst}} .$$

An example of the result is shown in Figure 2.

3 Dealing with Mobile Phone Limitations

Battery life is a main concern on all mobile devices. Floating point processors consume more power than integer processors, therefore most mobile devices (even high-end devices) have no floating point processor [Astle and Durnil 2004, Chap. 6]. Instead floating point calculations are simulated in software and that is too slow for game programming. How can we implement the lighting effects described in the previous section with no floating point operations? The answer is to employ fixed point arithmetics. Michael Street [2004] gives an excellent primer to fixed point arithmetics describing efficient implementations of all the basic math operations including square roots, vector normalization, and trigonometric functions (code is included).

The trigonometric functions are not really used in the calculations for our lighting effects, since cosine of an angle is found using the dot product between two vectors as shown in (1) and (7). If a game uses rotation of sprites as we do in our demo, it is necessary to evaluate (2) where an arcus cosine ($\cos^{-1}$) is employed. Moreover evaluation of the exponential function (e) is needed for calculation of fog transparency in (6). We recommend that look-up tables are used for fixed point evaluation of these two functions.

While it is straight forward to construct a look-up table for arcus cosine, since $\cos^{-1}(x)$ only takes values $x \in [-1, 1]$ as argument, it is not immediately obvious how to construct a look-up table for the exponential function. In our implementation we chose a look-up table with twenty entries for evaluation of e^{-x} when $x \in [0, 1)$, we have ten entries for $x \in [1, 2)$, and five entries for $x \in [2, 3)$. If $x \in [3, 10)$, the function is approximated by a straight line:

$$f(x) = \frac{e^{-3}}{7}(10 - x) .$$

Finally, when $x \in [10, +\infty)$, the return value is set to zero. We find no visual difference between floating point evaluation of the two discussed functions and use of table look-ups.

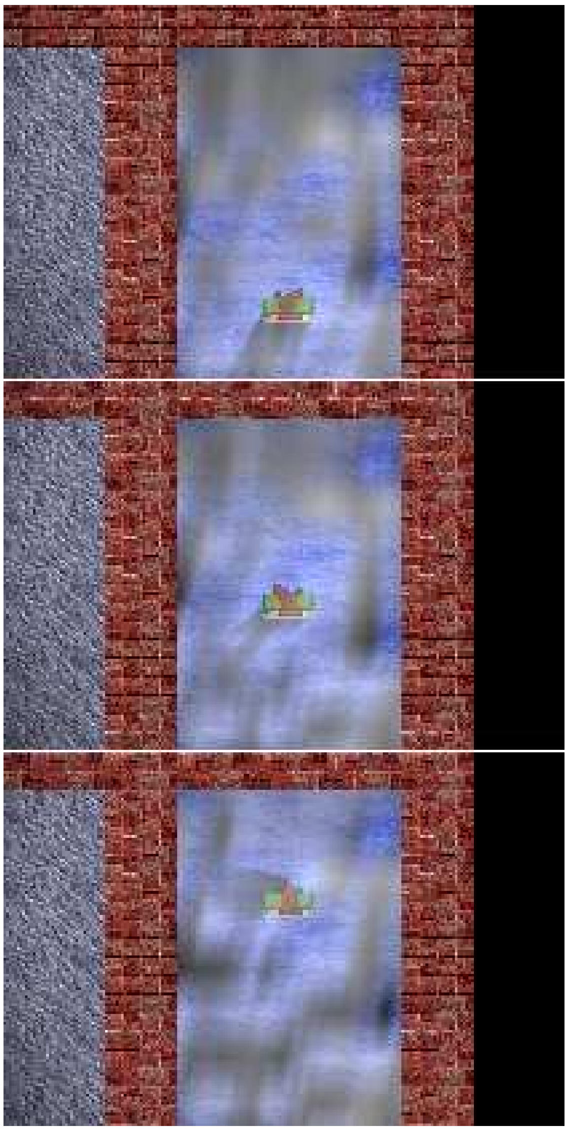


Figure 7: A few screen shots to illustrate that the fog changes appearance as the eye point moves around above it. The eye point is always placed directly above the character.

To make sure that our demo can potentially run on a mobile device, we have made an implementation using the PowerVR MBX OpenGL ES 1.1 SDK for desktop PCs[5]. PowerVR even supply a look-up table for fixed point evaluation of arcus cosine as a part of their OpenGL ES Tools library.

Both the shadow and fog described in the previous section can and should be drawn using triangle strips. This is efficient and suitable for both OpenGL ES and the J2ME 3D graphics API.

4 Results and Discussion

While there, as indicated in the previous section, are many implementation constraints when games are written for mobile phones, see also [Coulton et al. 2005], processing power is rarely an issue. The reason is that smaller screen size means "fewer pixels to push with each buffer flip" [Nok 2003]. Nokia also argues that the 104 MHz speed of the average mobile phone should be sufficient to support real-time rendered 3D graphics to a limited extent. This makes us confident that at least the shadows we have described are very well suited for mobile games. The fog may be slightly over the top. On the other hand, we only need to recalculate the terms in the part of the fog which is visible.

[5]The SDK is available at http://www.pvrdev.com/Pub/MBX/OGLES/.

	no shadow	**one strip**	**two strips**
w/o. fog	50.5	47.8	47.5
w. fog	30.1	29.0	28.9

Table 1: Frame rates (frames/sec) measured on an old laptop with or without fog and with no shadow, shadow drawn using a single triangle strip, or shadow drawn using two triangle strips. Please note the differences between the frame rates rather than the frame rates themselves.

To give a feeling of the performance hit entailed by the described methods, we have simulated our case study on a 400 MHz Pentium3 laptop. The simulations were run in a 250×250 resolution and with a fog grid of 16×16 vertices. What is important is not the frame rates themselves, but rather the difference between them. The program could run much faster in a lower screen resolution and with textures of a lower resolution, but then it would be difficult to tell the difference in performance between eg. shadow and no shadow since frame rates become quite unstable when they are high. The frame rates from our experiments on the laptop are given in Table 1.

The performance hit of the fog is seemingly a little high. The calculations described in the previous section are, however, not the expensive part of the fog rendering. They reduce the frame rate only by a frame or two. The expensive part is the Gouraud interpolation of colors and alpha values across the triangles that are drawn. Hence the larger the number of pixels which are covered by fog, the larger the performance hit. The white fog shown in Figure 2 is, in fact, less expensive to render than the fog with grey shades, since in the second case, the color values have to be interpolated as well as the alpha values. With the emergence of mobile GPUs (see some of the possibilities they entail in [Macedonia 2004]) which all have a hardware implementation of the Gouraud shading, the fog synthesis we describe should become very feasible for mobile games.

Considering the images in Figure 6 more closely, we will discover that the described shadows have some limitations. While the shadow outline is projected correctly onto the xy-plane, the method does not account for eg. the gap between the legs of the character. The gap is not captured by the shadow outline we can construct using a single height curve. To render such details, additional height curves (below the existing ones) would have to be incorporated.

Another problem is that the shadows are *always* projected onto the xy-plane. Suppose there is a wall next to the character, then the shadow will end up on top of the wall. This is definitely not desirable. A simple way to work around the problem is to draw the walls (and rooms on the opposite side of the wall) after the shadow has been rendered. This trick does not really fix the problem, the shadow may still appear on the opposite site of a closet or some other object close to the character. But then, on the other hand, the shadow caused by the object close to the character would usually cast a shadow itself on top of the shadow from the character. Hence, the problem is not fatal.

Throughout the main part of this paper we have referred to tile-based games viewed from above as the type of games where the described methods can be employed. There is, however, nothing to prevent us from using the same methods in games which are not tile-based or games which are not viewed directly from above. As long as the objects are all rendered as sprites moving on the same plane, the described shadow and fog techniques are applicable. It so happens that tile-based games usually belong to the category of games for which our techniques are useful. Therefore we describe our methods as rendering methods suitable for tile-based games.

5 Conclusion

An efficient method for rendering of projection shadows in tile-based 2D games has been presented in this paper. Shadows are found through construction of two height curves above each sprite describing a dynamic object or character in a game. The calculations needed for our shadows are sufficiently simple to allow for implementation on mobile phones. The shadows are not exactly physically correct, but they are a great improvement as compared to no shadows or simple blobs beneath the characters.

Additionally we describe a method for creation of fluctuating fog. The fog is described by a height field placed above a plane and fluctuates as the eye point moves around in the scene. The changing transparency of the fog is calculated according to physical considerations in the theory of radiative transfer. The shade of the fog is, however, simplified to limit the performance hit of the fog rendering. The OpenGL ES software simulation of Gouraud interpolation across the triangles visualizing the fog has shown to be the limiting factor with respect to processing power. With the emergence of mobile GPUs, this problem will disappear. In our opinion, the fog resulting from the rendering method we describe, is quite convincing. In particular we find that the fluctuations are an important part of the conviction. If the fog is rendered as a static semi-transparent layer, the gamer will hardly get any feeling of visual realism.

In light of recent development in and attention to the mobile gaming market[6], we believe that the time is right for incorporation of more lighting effects in mobile games. Hopefully this paper will help the gaming companies getting started on projection shadows and fluctuating fog.

Acknowledgement

Thanks to Rasmus Revall Frisvad for letting us use the character sprites which were created approximately ten years ago for a garage game project.

References

ASTLE, D., AND DURNIL, D. 2004. *OpenGL® ES Game Development*. Thomson Course Technology PTR, Boston, MA.

BIEDERMANN, S., 2005. Darkest fear: Bringen sie licht ins dunkel! Airgamer, July. http://www.airgamer.de/.

BLINN, J. F. 1982. Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics (SIGGRAPH '82 Proceedings) 16*, 3 (July), 21–29.

BLINN, J. F. 1988. Me and my (fake) shadow. *IEEE Computer Graphics and Applications 8*, 1 (January), 82–86.

CHANDRASEKHAR, S. 1960. *Radiative Transfer*. Dover Publications, Inc., New York. Unabridged and slightly revised edition of the work first published in 1950.

COULTON, P., RASHID, O., EDWARDS, R., AND THOMPSON, R. 2005. Creating entertainment applications for cellular phones. *ACM Computers in Entertainment 3*, 3 (July).

[6]The September 2005 issue of Game Developer Magazine was devoted entirely to the subject of mobile gaming.

GOOTCH, B., SLOAN, P.-P. J., GOOTCH, A., SHIRLEY, P., AND RIESENFELD, R. 1999. Interactive technical illustration. In *Proc. of the 1999 Symposium on Interactive 3D Graphics*, 31–38.

GOURAUD, H. 1971. Computer display of curved surfaces. Tech. Rep. UTEC-CSc-71-113, Department of Computer Science, University of Utah, June. Also in *IEEE Transactions on Computers*, vol. C-20, pp. 623–629, June 1971.

MACEDONIA, M. 2004. Small is beautiful. *Computer 37*, 12, 122–129.

NOKIA CORPORATION. 2003. *Designing Single-Player Mobile Games*, September. Version 1.01.

ROVIO MOBILE. 2005. *Darkest Fear: Fact Sheet*. http://www.rovio.com/.

STREET, M. 2004. A fixed point math primer. In Dave Astle and Dave Durnil: *OpenGL® ES Game Development*. Thomson Course Technology PTR, Boston, MA, ch. 4, 67–88.