STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVI**, Number 4, 2011

ACHIEVING REAL-TIME SOFT SHADOWS USING LAYERED VARIANCE SHADOW MAPS (LVSM) IN A REAL-TIME STRATEGY (RTS) GAME

ALEXANDRU MARINESCU

Abstract. While building a game engine in Microsoft XNA 4 that powered a RTS (real-time strategy) tower defense type game, we were faced with the issue of increasing the amount of visual feedback received by the player and adding value to the gameplay by creating a more immersive atmosphere. This is a common goal shared by all games, and with the recent advancements in graphics hardware (namely OpenGL, DirectX and the advent of programmable shaders) it has become a necessity. In this paper we will build upon the shadowing techniques known as VSM (variance shadow map) and LVSM (layered variance shadow map) and discuss some of the issues and optimizations we employed in order to add real-time soft shadowing capabilities to our game engine.

1. Introduction to existing shadowing techniques

Shadowing techniques can be divided in 2 major groups: shadow volumes and shadow mapping. Each has its own advantages and disadvantages and is best suited to a certain scenario. Shadow volumes rely on extruding actual polygons from the geometry of the rendered model, in the direction of the light.

We can see in Figure 1, taken from [6], the actual geometry that is added to the model in order to achieve shadowing. Shadow volumes have the advantage that they provide extremely accurate shadows, regardless of the distance between the shadow caster and receiver. On the other hand, they require a computational overhead on the CPU for the actual computation of the extra geometry and an added fill rate on the GPU for the actual rasterization,

Received by the editors: November 9, 2011.

2010 *Mathematics Subject Classification.* 68U05.

1998 *CR Categories and Descriptors.* A.1 General Literature [**INTRODUCTORY AND SURVEY**]; I.3.7 Computing Methodologies [**COMPUTER GRAPHICS**]: Three-Dimensional Graphics and Realism – *Color, shading, shadowing, and texture* .

Key words and phrases. shadow map, VSM, LVSM, render target, filtering, camera and light space, probability and statistics.

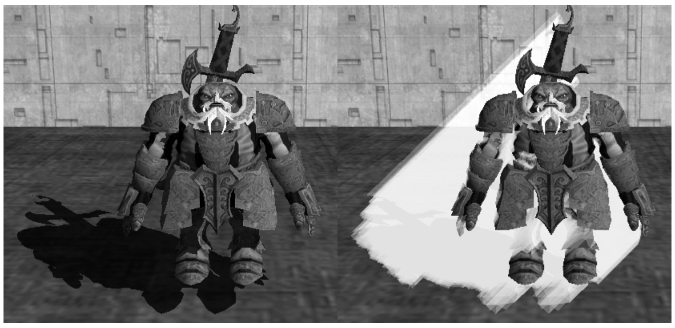

FIGURE 1. Shadow volumes and extruded shadow polygons.

depending on the volume density. In addition to this, shadow volumes are highly dependent on the complexity of the scene and naive implementations are sensitive to non-convex geometry while also suffering from self-shadowing artifacts (shadows cast by the geometry on itself).

Shadow maps utilize what is called “the light’s point of view”. The difference between camera and light view space is illustrated in Figure 2. The scene on which we perform shadowing is rendered twice. First, it is drawn as seen by the light (which means that we position our camera in the light’s position, facing in the direction of the light) and store the depth of each drawn object relative to the light in a render target (we will see that only this depth is needed).

As a side note, one should visualize this whole process in the following way: everything that comes out as output from the pixel shader (GPU program responsible with filling rasterized geometry with color information) is implicitly sent to the graphics card’s backbuffer and drawn on screen (the backbuffer can be thought of as a sort of standard graphics output). A render target is simply a region of graphics memory, and sometimes we wish to perform some post-processing before presenting the final scene to the user. This is why we hijack the backbuffer and instead draw to a render target. After we have drawn the distance from each object to the light, we obtain a render target which stores this information as a texture (depth map). Next we draw the geometry as seen by the actual game camera (with whom the user can interact) and take into account the depth map in order to compute the shadowing condition. In the vertex shader, each vertex of the geometry, besides being transformed into camera view space, is also transformed into light view space (so we know where to sample the depth map). The most basic condition for a visible pixel (as seen by the camera) to be in shadow is that its depth relative to the light is greater than the depth sampled from the depth map. Simply

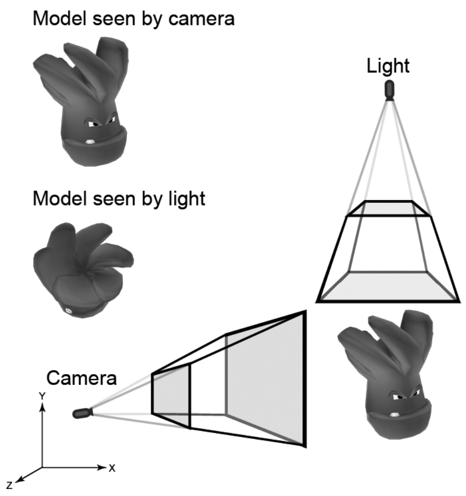

FIGURE 2. Camera view and light view frustums.

put, this states that, if the pixel seen by the camera is further away (from the light) than the corresponding pixel seen by the light, the pixel is in shadow; otherwise it is fully lit (since there is no occluding geometry).

Given the nature of GPUs, which natively operate with 32-bit floating point values, the above inequality $d_{camera} < d_{light}$ will inevitably yield floating point precision errors. These manifest visually as a flickering of the light/shadow boundary and grow in strength as the camera or light position move further apart. This can be solved by adjusting the camera and light view frustums such that the distance from the near to the far planes is as tight as possible. This will increase the actual precision of stored depth values; later on, we will utilize depth values scaled to the range [0..1] and this scaling will be done linearly using the formula:

$$d_{scaled} = \frac{d - nearPlane}{farPlane - nearPlane} \tag{1}$$

It works because, for example, a range near-far of 1-10000 provides us with 1/9999=0.0001 range per unit of distance, while a near-far plane distance of 1-100 provides 1/99=0.01 range per unit of distance, which is 100 times more accurate; also one should not forget that 32-bit floating point values are only accurate up to 10^{-6} which means that differences in depth finer than this minimum value will not be numerically recognized. Other methods for improving shadow quality include increasing the shadow map size, at the cost of increased memory storage and draw time. Nevertheless, there are situations when even

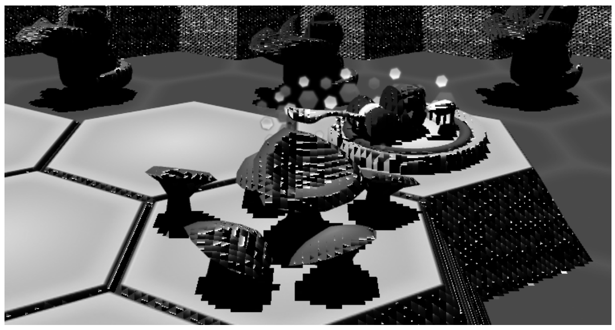

Figure 3. Regular shadow map, with point sampling.

a 4096*4096 render target is insufficient (which exceeds sizes supported by most graphics hardware). Still shadow maps are an extremely viable option, especially in our case where the scene is extremely complex and dynamic, due to their relative independence of scene complexity and because we can easily perform a blur post-process (using a Gaussian blur for example) to obtain soft shadows.

2. Variance Shadow Maps (VSMs)

Before delving further into VSM and LVSM shadowing techniques, we should discuss our available options regarding the depth map render target and their numerical implications. XNA provides us with 2 types of render targets: floating-point and pixel formats [5, 7]. Of specific interest to us are the Single (32-bit float format using 32 bits for the red channel), Vector2 (64-bit float format using 32 bits for the red/green channels), Vector4 (128-bit float format using 32 bits for each channel - red, green, blue and alpha), their half precision counterparts (HalfSingle, HalfVector2, HalfVector4) and the Rg32 (32-bit pixel format using 16 bits for the red/green channels) and Rgba64 (64-bit pixel format using 16 bits for each channel). Floating point render targets can store values in the same range as their underlying numeric type (float/half). These can store values in the depth map much more accurately, but most graphics hardware is limited to POINT (nearest neighbor) filtering when sampling from floating point surface formats, thus making shadows appear jagged. A more detailed discussion on texture filtering can be found at [3]. Figure 3 describes the severity of shadowing artifacts on a standard shadow map, using POINT filtering.

We could create our own filtering kernel and perform the actual texture filtering ourselves, but this is costly and we aim to keep any computational overheads to a minimum, due to the fact that a game engine has to perform many other functions, apart from shadowing.

On the other hand, pixel formats allow efficient filtering, mip-mapping, and anti-aliasing, but any value written by the pixel shader is clamped to the [0..1] interval. So, in order to keep depth values relevant, we need to map them using a depth metric to the range [0..1]. Paper [2] states that any strictly monotonically increasing function will do, but linear depth metrics are the most commonly used; in the actual implementation we have used the metric given by (1).

With standard shadow maps, a texture unit (texel) of the depth map can only store the depth of a single point, so the basic idea behind VSMs is to store a statistical distribution of depths at each texel. In [2], this is done by storing the first and second moments μ_1, μ_2: the mean depth and the mean squared depth. Thus, we can approximate the average of two distributions by averaging the moments (which is done inherently by the filtering hardware of the graphics card). When drawing the scene from the camera's point of view, we will use the sampled moments to compute an upper bound on the probability of a pixel being lit. So, instead of dealing with a simple boolean situation as with simple shadow maps, now we consider that all pixels have a certain probability of being lit/shadowed. Chebyshev's inequality is applied on the mean μ and variance σ^2 derived from sampled moments (denoted M_1, M_2) to provide this upper bound (which is shown in [2] to be exact for the single planar occluder-receiver case).

$$mean = \mu = E(x) = M_1 \tag{2}$$

$$variance = \sigma^2 = E(x^2) - E(x)^2 = M_2 - M_1^2 \tag{3}$$

Let d be the depth of the current fragment seen by the camera, relative to the light, after it has been scaled using the chosen depth metric, then the probability P_{lit} of the fragment being lit is given by:

$$P_{lit} = \{1,\ if\ d \leq \mu\ and\ \frac{\sigma^2}{\sigma^2 + (d-\mu)^2}\ otherwise\} \tag{4}$$

It is worth mentioning that VSMs require double the amount of storage needed for simple shadow mapping techniques because we also store the depth squared in a separate color channel. Still an implementation at this stage suffers from shadowing artifacts (Figure 4), but there is room for improvement.

Chapter 8 of [8] discusses various techniques for improving shadow quality, some of which we have implemented and will be mentioned here. Numeric inaccuracy is still a potential issue; this problem can be dealt almost entirely

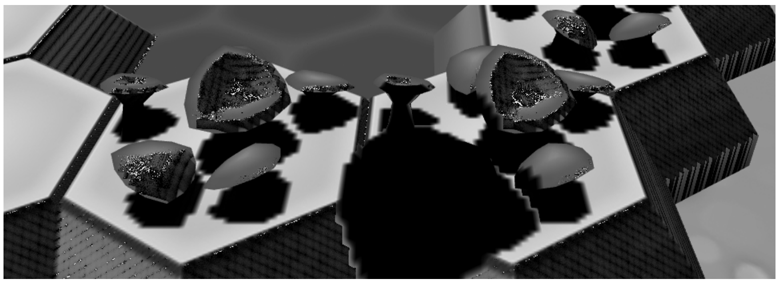

FIGURE 4. Simple VSM implementation.

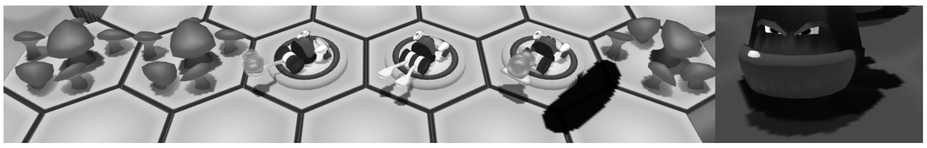

FIGURE 5. VSM with minimum variance clamped to a constant value.

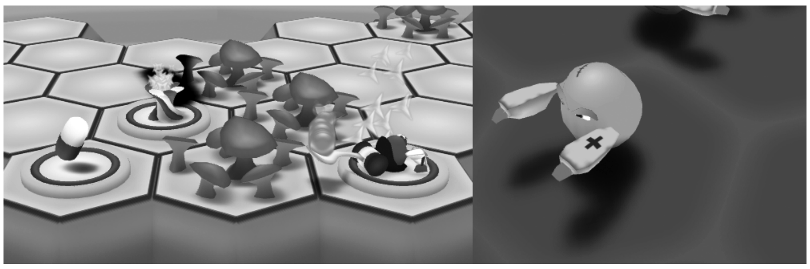

FIGURE 6. Additionally - bilinear filtering, MSAA and Gaussian blur.

if we clamp the minimum variance to a small value before computing P_{lit}; this minimum variance is a constant value, and once tweaked properly can yield significant improvement (Figure 5).

Blurring the shadow map may help hide shadowing artifacts; usually simply enabling mipmapping, trilinear/anisotropic filtering and multisample antialiasing while rendering the shadow map may greatly increase shadow quality at almost no extra cost (Figure 6).

Finally, we can attenuate the light bleeding phenomenon (visible in Figure 4), which is usually seen when the soft edge of a shadow is visible both on the first receiver (as it should be) and, to a lesser extent, on a second receiver (which should be fully occluded by the first). This can be done by modifying

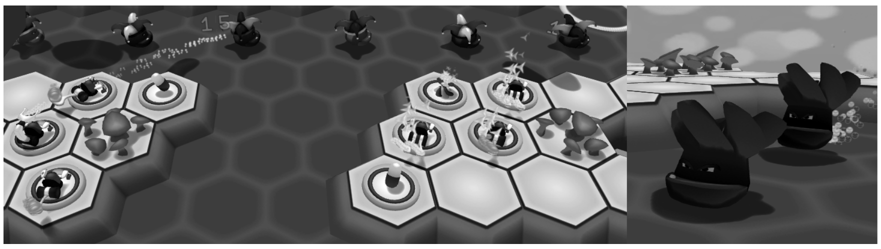

FIGURE 7. Additionally - light bleeding reduction is applied.

P_{lit} before applying lighting calculations, for example, any values below some minimum intensity are mapped to 0, while the remaining values are rescaled to [0..1] (Figure 7).

3. Layered Variance Shadow Maps (LVSMs)

Attempts to further build upon VSM and reduce the light bleeding artifacts are taken in [4], where LVSMs are first introduced. Instead of storing a single depth map with its associated depth metric (1), we partition the light's view space into multiple layers, each with its own depth metric and corresponding depth map. Thus, for each layer L_i covering a depth interval $(nearPlane_i, farPlane_i)$ and an unscaled light distance d, we define the depth metric:

$$d_{scaled} = \frac{d - nearPlane_i}{farPlane_i - nearPlane_i} \tag{5}$$

As stated above, this also has the effect of increasing shadow precision when compared to standard VSM because the near-far planes are now closer together. Furthermore, as shown in [4], when shading a surface it is enough to sample a single layer to determine the shadowing condition: the layer L_j that contains the receiver surface we are shading (at depth d).

We must choose our strategy for splitting the depth range carefully. If we want the best numeric precision overall, a simple uniform split is ideal. If we want to minimize shadow bleeding, we should analyze the distribution of shadow casters and receivers in our scene and choose depth intervals accordingly. Nevertheless, this improvement in quality comes at the cost of additional storage per depth layer.

4. Tradeoffs

As with all graphics applications, a trade-off must be reached between speed and quality and various techniques must be combined in order to attain

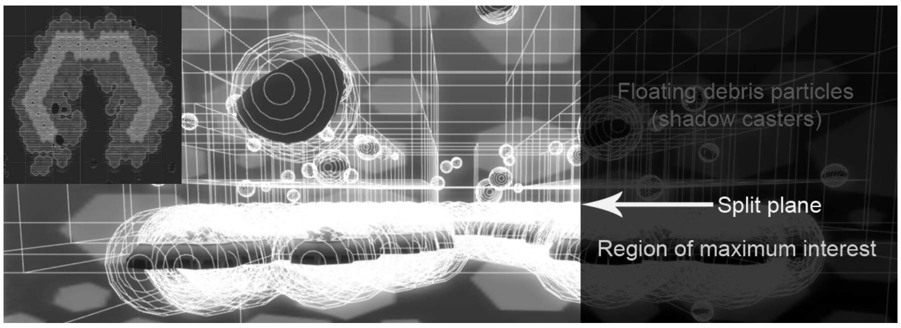

FIGURE 8. OcTree, bounding spheres for entities, and the VSM (from light's perspective).

a pleasing effect. It is a widely accepted fact that there does not exist a general method that works for all situations. A broader range of shadowing algorithms and the environments they are best suited to are outlined in [1].

Consider the following scenario: we need to apply real-time soft shadows to a large scale environment that is specific to a real-time strategy game. The gamer is expected to amass a significant number of units and buildings, which are also coupled to a complex animation system. The terrain (map) is illuminated by a single directional light (no actual position, just light coming from a general direction); this should enable all geometry to cast dynamic soft shadows and augment the gameplay feel. The camera allows for zooming in close to units/buildings and zooming out facilitating strategic thinking; shadow quality should not suffer greatly in neither of these situations. Furthermore, as seen in Figure 8, the shadowing system is expected to take into account for example a particle system that floats above the map and casts dynamic shadows on the terrain.

The fact that a directional light does not have an actual position in the 3D world is an issue which prevents us from constructing its proper View/Projection matrices. We have overcome this by constructing a bounding box around the geometry of the terrain. We compute the light's world position *LightPos* by backing out in the reverse direction of the light from the centroid of this bounding box as far as it is needed (to encompass the floating shadow casters).

$$LightPos = \frac{1}{8}\sum_{i=1}^{8} BoundingBox.Corner_i - LightDir * Distance \tag{6}$$

For the light projection matrix, we build an orthographic projection (all projection lines are orthogonal to the projection plane), which allows to change the pixel depth calculation from “distance to light position” to “distance to light plane”. It may not seem important at first, but this distance can be computed in the vertex shader and afterwards interpolated before passing to the pixel shader, where we output the variance depth map. This is a major gain in speed, because the distance to light position could only be calculated correctly on the pixel shader who is executed once for each fragment, whereas the vertex shader is executed once per-vertex. Development can be further taken in the direction of constructing light view/projection matrices that optimally adapt to the current view frustum of the player camera. Given the fact that all the units and buildings requiring detailed shadowing would be located within a small depth range, and the debris particles (which are also shadow casters) would wander within a much larger distance interval, we implemented LVSMs with 2 layers, one concentrated on the ground region and one that would deal with the floating shadow casters.

5. Conclusions

Finally we would like to summarize the whole shadowing algorithm steps and optimizations which we think can be applied not only to our example, but to any RTS type game:

- Construct optimal light View/Projection matrices only once (when loading map for example);
- Manage geometry through a data structure that enables efficient culling (we use an OcTree for spatial partitioning);
- Render layered variance depth maps into a 1024*1024 Rgba64 render target, with multisampling enabled, considering distance to light plane; we use red/green channels for the first layer and blue/alpha for the second;
- Blur depth map using a two pass Gaussian blur (separated into 2 blur filters, one horizontal and one vertical); we use a blur amount of 1.25;
- Change to camera view/projection and draw scene, enabling bilinear filtering for the shadow map; we clamp the minimum variance to a value of 0.01 and perform shadow bleeding reduction using a power function: $P_{lit} = P_{lit}^{13}$; we also offset P_{lit} with a value of 0.65 to lighten the shadows;
- Applying a post-processing bloom effect yields even nicer results since it also helps hide any remaining artifacts and makes the scene much more organic (Figure 9).

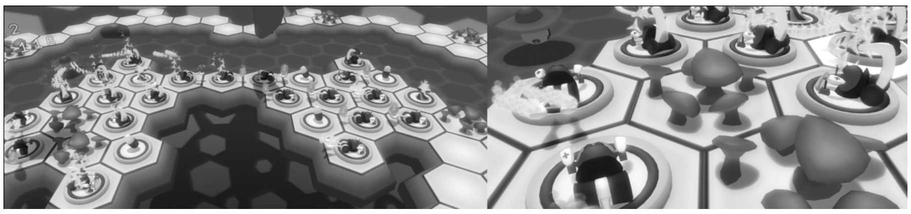

FIGURE 9. Final result, after applying a bloom post-process.

6. ACKNOWLEDGEMENTS

The author would like to thank Assoc. Prof. Ph.D. Simona Motogna[1] for her invaluable role as mentor for the Imagine Cup 2011 Game Design competition and Lect. Ph.D. Rares Boian[2] for tutoring his graduation thesis. Finally, the author would like to thank his team mates: Bogdan Tanasoiu[3] and Catalin Pintea[4] for creating such a wonderful competition atmosphere.

REFERENCES

[1] L. Bavoil, Advanced Soft Shadow Mapping Techniques, Game Developer Conference, 2008.
[2] W. Donnelly, A. Lauritzen, Variance Shadow Maps, Proceedings of the 2006 Symposium On Interactive 3D Graphics and Games, 2006, p. 161-165.
[3] S. Hargreaves, Shawn Hargreaves Blog - Game programming with the XNA Framework, http://blogs.msdn.com/b/shawnhar/
[4] A. Lauritzen, M. McCool, Layered Variance Shadow Maps, Proceedings of graphics interface 2008, 2008, p. 139-146.
[5] Microsoft, App Hub - XNA main developer portal for Windows Phone & Xbox 360, http://create.msdn.com/en-US/
[6] Microsoft, DirectX SDK June 2010.
[7] Microsoft, MSDN Library - Documentation for developers using Microsoft technologies and tools, http://msdn.microsoft.com/en-us/library
[8] H. Nguyen, GPU Gems 3, Addison Wesley Professional, 2007.

DEPARTMENT OF COMPUTER SCIENCE, BABES-BOLYAI UNIVERSITY, KOGALNICEANU 1, 400084 CLUJ-NAPOCA, ROMANIA
E-mail address: mams0507@scs.ubbcluj.ro

[1]motogna@cs.ubbcluj.ro
[2]rares@cs.ubbcluj.ro
[3]tbei0012@scs.ubbcluj.ro
[4]ppir0696@scs.ubbcluj.ro