Modern Texture Mapping in Computer Graphics

Wolfgang Wenigwieser*
TU Vienna

Figure 1: Rendering of virtual surface detail (left and center, [Policarpo et al. 2005]) and shadow mapping (right, from the pc game 'Crysis').

Abstract

The original texture mapping algorithm was introduced to add detail, color and surface texture to computer generated graphics. Because by itself it lacks any capability to simulate changes of lighting, shadowing and perspective perturbations caused by finer surface detail a number of improvements like bump mapping, parallax mapping and relief mapping have been introduced. Another common use of textures is shadow mapping. This paper not only describes those modern day additions to basic texture mapping but focuses on the latest refinements of those techniques which finally give them the quality and performance required for real world applications.

CR Categories: D.1.m [Software]: Programming Techniques—Miscellaneous;

Keywords: horizon mapping, parallax mapping, shadow mapping, perspective shadow maps, variance shadow maps

1 Introduction

Beginning with the introduction of texture mapping [Catmull 1974] there has been research focusing on how to simulate surface detail and geometry on a surface without having to actually model it with additional polygons. Bump mapping, introduced in 1978 changes the lighting of a surface on a per texel basis by reading the normals from a texture map instead of generating them from the actual geometry of the object. This alone generates a quite convincing impression of small bumps and wrinkles. But for rendering steeper and bigger structures like stones protruding from the wall of a medieval castle we need to do more than just change the lighting on a per pixel basis.

A human observes needs several visual clues before he is convinced that a surface is not just covered by a flat image. Apart from changes in lighting this includes the apparent shift of parts of a surface relative to each other caused by the motion of the observer. This is called the parallax effect and can be approximated or even exactly simulated by techniques like relief and parallax mapping. Another visual clue is shadowing. Not only does the side of a bump in wall facing away from a lightsource need to be shaded differently, this bump should also cast a shadow. Horizon mapping has been the first algorithm to simulate this combined with bump mapping.

But shadows are certainly not only cast by surface detail. The currently most widely used technique for casting shadows between surfaces is shadow mapping. This paper focuses on the latest variants and improvements of those techniques which in some cases have made them for the first time practical for real time rendering.

2 Bump Mapping

Bump mapping [Blinn 1978] is an approach to simulate rough and wrinkled surfaces. This is achieved by perturbing the surface normal on a per pixel basis. Hence only the shading is changed, the underlying geometry and the texels of a texture map are unaffected. The normals can either be calculated from a height map on the fly or simply read from a normal map already containing the precalculated normals.

Blinn used an approximation by adding the surface normal and the perturbed normal. The latter one he calculated using the partial derivatives of the surface in the u and v direction respectively.

[Peercy et al. 1997] developed an approach to bump mapping which is more friendly to graphics hardware and also the one used today. They construct an orthonormal frame on the surface which is used to rotate the light vector into the local frame of the bump map. This is done because we need to have the normals of the bump map and the light vector in the same reference frame to be able to do any kind of calculation involving both. While it would be possible to store the normals in world coordinates this would require to

*e-mail: wenigwieser@gmail.com

calculate a separate normal map for each object. The only benefit being that we could save the world to tangent space transform (tangent space being the local frame of the bump/normal map). As this transform simply isn't a performance issue on modern graphics hardware any more world space normal maps are no longer required. Now this tangent space frame of reference is constructed using N the unpertubated normal of the original surface, T the tangent vector and B the bi-normal computed as the cross product of N and T. This frame can be used as a rotation matrix. A rotation matrix is an orthonormal frame, which means each line and column in the matrix is a unit length vector and they are orthogonal to each other.

The transformation of the light vector L into tangent space only needs to be done at each vertex and is then interpolated between vertices to obtain the per pixel light vector L'. Usually L' is renormalized at each pixel because the interpolation may reduce the length of the vector from 1 to some lower value, an effect which is the strongest toward the center of the polygon resulting in dimming the final color values. The diffuse light intensity can then simply be computed as L'*N where N is the normal read from the bump map.

2.1 Adding Shadows to the Bumps - Horizon Mapping

The relatively simple approach behind bump mapping of changing the surface normals is enough to create a quite convincing simulation of detail where there actually only is a flat polygon. To make this effect even more convincing one could not only shade those bumps and wrinkles but let them cast shadows. When one remembers that the bumps of bump mapping are only virtual and not actually modeled in geometry it becomes clear that traditional techniques like shadow volumes or maps won't work.

[Max 1988] presented an approach where visibility information is precomputed for each point on a surface and then stored in a so called horizon map. This visibility information specifies the angle to the horizon seen from a specific point in a discrete number of directions. Because only a light source which is higher then the horizon can illuminate this point this information can be used to determine at run time if the current pixel is in shadow or not. This is done by testing if the angle of a vector pointing from the pixel to the light is higher then the angle to the horizon.

Just like with bump mapping we need to transform the light vector into the local frame of the horizon map. A vector in this frame has an angle with the normal Φ and an orientation in the tangent plane θ. [Max 1988] donates the resulting horizon map as $\Phi u, v, \theta$. Which means the map contains the azimuth angle Φ when a light would become visible at the discrete parameter values u and v in a set of directions θ. The parameters u and v are typically sampled at the same resolution as the bump map.

Usually to keep memory requirements down the angle to the horizon is stored for only eight directions: north, north east, east and so forth. At runtime the values can then be interpolated according to the exact direction the light source is at. In an efficient implementation those eight values are stored in two texture maps encoding the angles in the r, g, b, and alpha channels of the textures respectively (Figure 3).

The interpolated angle $\Phi(u_i, v_j, \theta)$ for the direction θ and the discrete coordinates u_i and v_j can be found by interpolating between the neighboring discrete directions as follows:

$$\Phi(u_i, v_j, \theta) = \sum_{k=1}^{M} B_k(\theta)\Phi(i, j, k) \quad (1)$$

Here M is our total number of discrete directions (for example 8). This means for each discrete direction we take the angle to the horizon at coordinates (i, j) and multiply it with the function $B_k(\theta)$, which basically gives us the interpolation weight for the k-th direction. It does that by evaluating to 1.0 for the corresponding direction and linearly falling off to 0.0 for the neighboring directions. So if north is the direction k=0 then $B_0(\theta)$ would return 1.0 for an angel θ pointing directly north and fall off towards zero for NE and NW. Thus by adding all M products of the interpolation weight B_k and the horizon angle towards the direction k we have linearly interpolated between the discrete directions.

In the implementation of [Sloan00] the values for $B_{0..8}$ are precalculated and stored in two texture maps. At runtime instead of calculating the angle θ they take the x and y component of the light vector and use those to look up the blending weights from the textures. Like with the 8 horizontal angles they store one value in each channel of what they call the basis textures (Figure 2). To sum a possible implementation of the algorithm up:

- for each vertex:
  - transform the light vector into tangent space
- for each pixel
  - read the 8 horizontal angles from the two horizon maps
  - read the 8 interpolation weights from the two basis maps using (x, y) from the current light vector as texture coordinates
  - multiply add those 8 angles and weights to get the final horizontal angle
  - subtract this from the angle between the light vector and the surface normal
  - if the result is > 0 render this pixel with ambient + diffuse light
  - if the result is < 0 only render this pixel with ambient light (= in shadow)

With modern graphics hardware it is possible to implement those steps in a single vertex and pixel shader. As an alternative [Sloan and Cohen 2000] describe an implementation which doesn't requires programmable hardware but needs several passes.

The drawbacks of horizon mapping are quite obvious: for every bump map two more maps (probably of the same resolution) are needed tripling the amount of texture memory normally required by bump mapping. Additional problems may arise if horizon mapping is applied to a curved surface. The curvature actually changes the horizon. Because the polygons making up the surface are somewhat approximating the curvature this may still work but artifacts can appear especially at the seams between the polygons. A solution for this would be to generate a horizon map per object instead of just per bump map (this object has to have an uniform u, v mapping). This may very well raise the memory consumption to a level which isn't acceptable any more for a real world application.

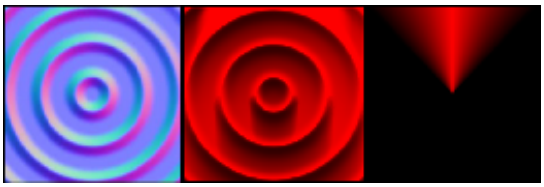

Figure 2: A bump map, the north horizon map and the north basis texture as proposed by [Sloan and Cohen 2000].

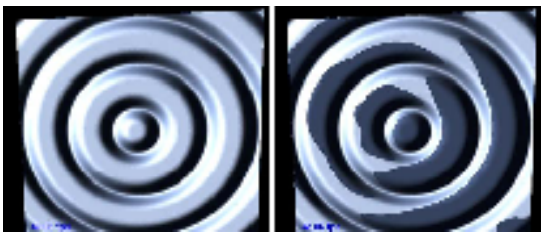

Figure 3: The results of horizon mapping, left: bump mapped only, right: with self shadowing [Sloan and Cohen 2000].

3 Parallax Mapping

The last two techniques discussed can simulate shading and shadowing of 'virtual' detail on a flat polygon in a convincing way. But that's only the case as long as the detail isn't raising too far from the plane of the polygon. Figure 4 shows what happens if you paint a pyramid and a sphere into a bump map and view this rendered from a steep angle (top left and top right).

Figure 4: A pyramid and a sphere painted onto a surface using a bump map, top left: viewed from top this somewhat works, top right: from the side the flat nature of the texture is unveiled, bottom: how the image should look, rendered using actual geometry.

The result isn't very convincing (or not convincing at all) because there is one very important effect still missing, motion parallax. What the parallax effect refers to is: take two points in space, viewed from a certain direction and then move the observer. What happens is that those two points apparently shift relative to each other and relative to a background. So when viewing the sphere and pyramid from a steeper angle then from the top what should happen is shown in the bottom picture of Figure 4, part of the image should shift due to the change in perspective. The good news is we can actually recreate this effect in a shader.

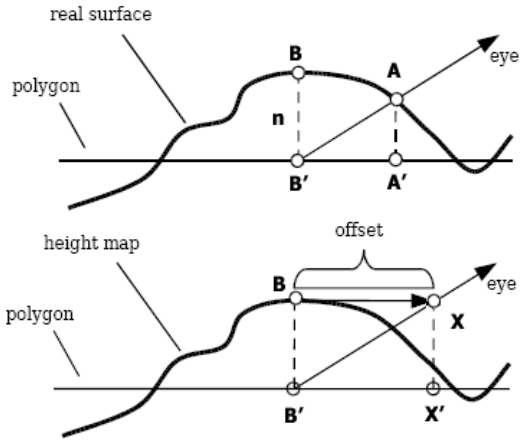

Figure 5: The approximation error, resulting in the points X and X' when we actually looking for A and A' [Welsh 2004].

The first people who proposed a real time texture technique for geometry rendering based on the motion parallax effect were [Kaneko et al. 2001]. They name image based rendering as an inspiration. In this field rendering is based on a set of two dimensional images instead of three dimensional geometry. In IBR it is common to distort the images so other viewpoints then in the original photographs can be rendered (for example generating a seamless panoramic 360 degree view from a set of still images). Kaneko et al. now distort a texture at rendering time according to perspective. Actually they change the texture coordinates on a per pixel basis instead of distorting the texture image itself.

The basic principle is this: when one compares the top right and the bottom picture in Figure 4 it becomes clear that the parts of the texture which are elevated from the surface of the plane move farther away from the viewer (like the top of the sphere). Parts which are lower then the plane would move closer to the viewer. It is important to note that Kaneko et al. suggest that their parallax mapping approach is only used for surfaces relatively small in the rendered image or for billboard-type objects because it has some inaccuracies and therefore has problems with very steep detail structures or viewing angles.

The heart of their algorithm is how they calculate the shifted texture coordinates. In Figure 5 one can see the surface normal n and the surface point to eye vector e. The curve represents the virtual detail on the flat polygons surface we want to render, it is commonly defined by the values in a grayscale height map. In the diagram the eye vector e hits the polygon at point B'. That's also where the current texel would be read from. But what we actually want is to see point A because that's what the eye vector would intersect. Now to see point A all we have to do is change the texture coordinates from B' to A'.

We call the depth of our virtual surface depth(u, v), this is the height of the curve at coordinates u, v. The angel between e and the surface normal n we call θ. Now we have a right angled triangle where the angle at B' and the length of the side B' to B are already given. The side B' to A' which we are looking for can be calculated by:

$$tan\theta x depth(B') \quad (2)$$

This is the offset we have to add to the current texture coordinate u. But in this calculation we have made one important approximation. We don't have the height at A' yet because that's the point we are looking for, so instead we used the height at B'. In other words instead of using the triangle B', A', A as would be correct we had to use the triangle B', B, X'. Else we wouldn't have enough sides or angles to calculate B' to A'. This results in only a small error as long as the difference between the two heights is relatively small. The lower diagram in Figure 5 shows how the approximation error grows as the as the viewing angle becomes steeper. Here we once again have the point A' we are looking for but using θ and depth(B') we arrive at point X.

This is the one approximation this algorithm has to make. This is also the reason why it works best with relatively smooth surfaces were the height values don't change too quickly.

3.1 Parallax Mapping with Offset Limiting

There's one further problem with this original approach described in the last chapter. With the viewing angle becoming steeper tan(θ) becomes infinitely large and so does the texture offset, the result being that the texture deforms into a unrecognizable mess of pixels. So what we want to do is cap our expression tan θ x depth(B') at some value. The following tweak has been introduced by [Welsh 2004]. Instead of tan θ x depth(B') we can also write:

$$\frac{e.xy}{e.z} x depth(B') \quad (3)$$

where e is the viewing angle. Welsh now conveniently selects depth(B') as the length limit, this can be done by simply dropping the division by e.z from the equation, resulting in:

$$e.xy x depth(B') \quad (4)$$

The length of the offset is now limited by the height (or depth, however one views it) at point B. While this not only limits but also changes the values of the expression one should remember that parallax mapping itself is a approximation so this shouldn't be a problem. We have not only solved the problem of (e.xy / e.z) becoming too large but also simplified the code in the pixel shader by dropping a costly division. With this small change parallax mapping is finally a technique useable in practice.

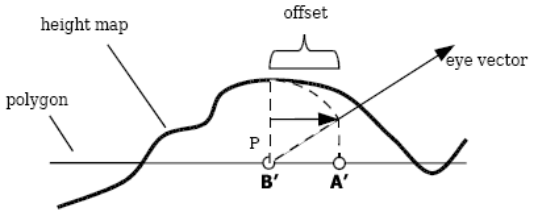

Figure 6: Limiting the offset by the height at B' [Welsh 2004].

Figure 7: The mythical stone wall rendered with the performance wise very cheap technique of parallax mapping, the perceived depth of the stone wall is greatly enhanced [Welsh 2004].

3.2 Improving on Parallax Mapping

While parallax mapping for its simplicity provides surprisingly good results every approximation technique has its limits. So what if your application needs high frequency height fields or surfaces viewed extremely close from steep angles? Moreover there are two important visual cues the technique in the last chapter still cannot provide, self occlusion and rendering a correct silhouette (one which includes the surface detail). Self occlusion means that parts of a surface which rise from it should occlude parts lying behind them. This is shown in the bottom picture of Figure 4. The backside of the sphere is not only distorted due to perspective but not rendered at all, something which basic parallax mapping can't do. How much changing the silhouette to reflect the surface detail can add to realism is shown in Figure 8.

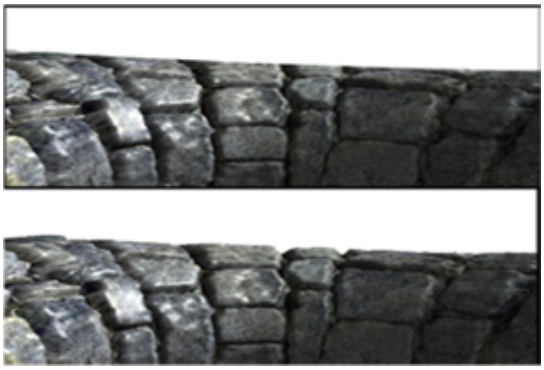

Figure 8: Rendering a correct silhouette adds quite a lot to realism [Policarpo et al. 2005].

To arrive at a technique which can finally do all those things, provide the illusion of depth, self occlusion, self-shadowing and ideally draw a correct silhouette a flurry of techniques and variations have been proposed. One of the earlier ones was relief texture mapping [Oliveira et al. 2000], which uses software-based image-warping, which means every texture has to be pre-warped in software using a raycasting based algorithm before it can be rendered. This approach shows correct depth with self occlusion but was only demonstrated in a software implementation. It is a two step approach, its biggest drawback probably is that it has not been extended beyond flat surfaces (to arbitrary polygonal surfaces) and it's requirement of pre-

warping textures. While it served as an inspiration for other techniques it's not really used today in its original form.

[Policarpo et al. 2005] build upon relief texture mapping to come up with something they simply call relief mapping. It is capable of rendering correct self-occlusions, shadows, interpenetrations, and per-pixel lighting. The basic idea behind it is, to find out which point of the height map containing the surface detail would be visible at the current pixel being rendered, instead of using some huge lookup table or prewarped data, just calculate it directly in real time. This is done by tracing a ray along the view vector starting from the plane of the polygon until it intersects with the height field. While this may sound prohibitively expensive at first Policarp et al. demonstrated that their algorithm runs at real time framerates. As a downside their implementation produces some flattening, requires several very costly dependent texture reads and is by itself not capable of rendering silhouettes.

Other approaches are based on ray tracing [Patterson et al. 1991; Pharr and Hanrahan 1996; Heidrich and Seidel 1998; Smits et al. 2000]. They are computationally very expensive and therefore not suitable for real-time applications. The same goes for 3D inverse image warping [Schaufler and Priglinger 1999]. It was also suggested to render surface detail by stacking layers of 2D textures onto a polygon. This 3D texture mapping [Meyer and Neyret 1998; Kautz and Seidel 2001] requires huge amounts of memory and has a severe performance hit it may also introduce objectionable artifacts depending on the viewing direction.

There are also several techniques based on precalculating extensive datasets from a heightmap which end up storing some kind of lookup table or function into a 3D texture requiring huge amount of memory even for a single moderately big texture. While rendering performance and quality based on this precalcualted data is often quite good in an ideal world what we want is a technique which doesn't require more then the original heightmap itself, just like the relief mapping by Policarpo et al.

3.3 Relief Mapping

In real time graphics the most successful algorithms are often those which do not require any kind of precalculated data. From the approaches listed in the previous paragraphs this would be relief mapping by Policarpo et al. Another one we haven't mentioned yet is described in [Tatarchuk 2006] which the author calls parallax occlusion mapping. Both are actually quite similar in principle but have some important differences in their implementations.

They are both based on the idea to render the unevenness stored in the height map threw per pixel ray tracing at run time. Basically for every pixel rendered you cast a ray along the view vector from the camera through the current pixel and intersect it with the height map positioned below the surface of the polygon. This is shown in Figure 9.

The difficulty with this approach lies in the fact that we can only sample the height field along the ray at a finite number of intervals. So you will run into all the problems associated with sampling a dataset at a lower resolution then the original data, like aliasing. To improve the quality of the sampling process and still keep the number of samples at a minimum Policarpo et al. use a binary search. In detail what they do is first find the view vector from the current point on the polygon to the camera. The current pixel is the point A and the intersection of the view vector with the bottom of the height map is B. Their depth values are 0.0 at the surface of the polygon, the maximum depth value is 1.0. Now the starting points for the binary search are A and B. The first sample is taken at the midpoint, in

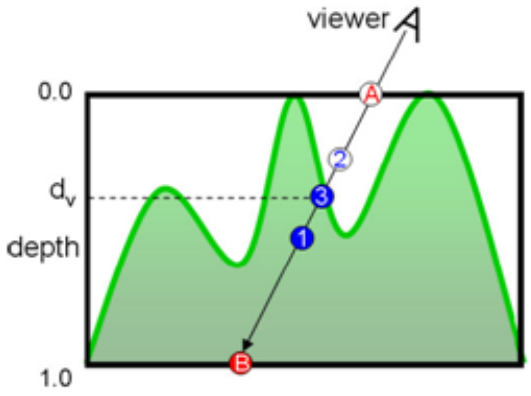

Figure 9: Using a binary search to intersect a ray with a height-field [Policarpo et al. 2005].

Figure 9 this is the circle marked "1". If the stored depth is smaller then the computed one, this point lies inside the height-field. The search always continues with one endpoint inside and another one outside the height-field. So the next sample is taken at "2", which lies outside the surface. Policarpo et al. have found that eight steps are enough to produce very satisfactory results. This number equals $2^8 = 256$ equally spaced subdivisions of the full depth range of the height-field.

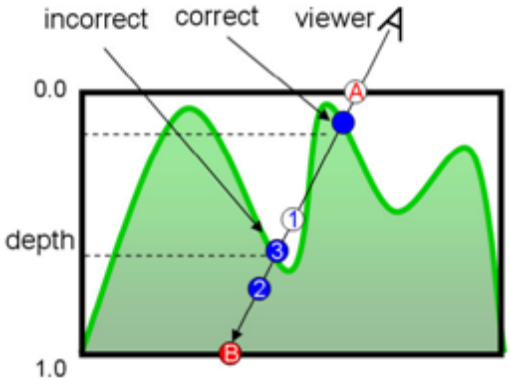

Figure 10: Why a binary search may terminate at the wrong point of intersection [Policarpo et al. 2005].

Performing a binary search yields good results with a relatively low number of iterations, but one point to consider is that it's not very friendly to the architecture of today's graphics hardware. The position of the next sample is calculated based on the depth from the last sample. This is a so called dependent texture read, it cannot be processed by the hardware in parallel with the next few texture reads and therefore creates quite a performance hit. While this may change or be less severe on future generations of hardware there's also one conceptual problem with performing only a binary search. If the view ray intersect the height-field in more than one point, the binary search will terminate randomly at one of the intersections, not necessarily at the first one. This is illustrated in Figure 12. To overcome this problem Policarpo et al. start with a linear search. They step along the interval AB until the first point inside the surface is found. On graphics hardware capable of dynamic flow control (shader model 3.0 or higher) the number of steps taken is a function of the angle between the view vector and the normal of the polygon. As this angle becomes greater the length of AB increases so one has to divide the line into more intervals. Policarpo

et al. cap the number of steps at 32. When the first point inside the surface is found the binary search starts with the last point outside the surface and the current one. As this interval is way smaller then the full length of AB a smaller number of binary subdivisions is sufficient.

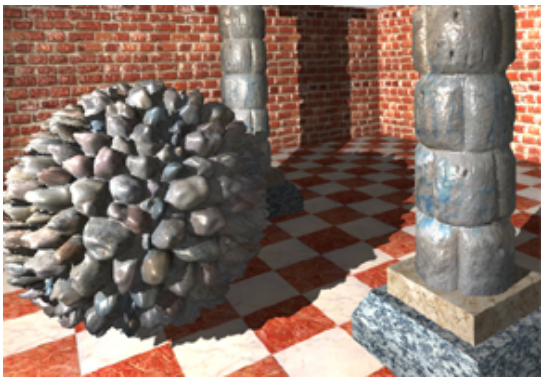

Figure 11: Relief mapping achieves a quality which clearly surpasses the one of parallax mapping [Policarpo et al. 2005].

Because with this approach we essentially have a small ray tracer another standard problem of CG is quite easy to solve, calculating self shadowing. We can do this by casting a ray from the intersection point obtained by the binary search towards the light source and test whether it intersects the height-field. Note that this time determining the exact point of intersection is not required.

3.4 Parallax Occlusion Mapping

Like mentioned before the binary search used in relief mapping creates a quite severe performance hit. Still at grazing angles it would produce aliasing artifacts, to get rid of them Policarpo et al. use a depth bias, but this solution flattens the surface detail at steep viewing angles. Also with a very high-frequency depth map the hard shadows calculated by a single raycast may show aliasing artifacts. In [Tatarchuk 2006] the author presents a technique which seems to solve all those issues. While he calls it differently it is mainly a variation and tweak of the original relief mapping approach.

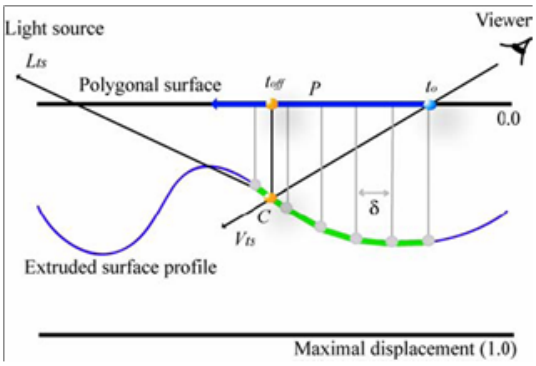

Figure 12: First a linear search at fixed intervals is done, when a hit is found the exact hit location is approximated [Tatarchuk 2006].

The starting point for calculating the displaced texture coordinates is also a linear search. The interval size along the view ray is again a function of the angle between the view ray and the polygons normal. Tatarchuk uses a minimum and maximum bound for the number of intervals δ. The main difference in his approach is, once the first sample lies inside the surface instead of starting a binary search he approximates the curvature of the surface with one quite inexpensive calculation. He takes the current height sampled and the previous one and connects them with a line segment, if the intervals are small enough this will resemble the actual surface curvature quite closely. He then intersects this line segment with the view ray. This can be seen in Figure 12. Comparison shots in [Tatarchuk 2006] show that this results in the same quality as a binary search with applied depth bias, with the significant advantage that no flattening occurs.

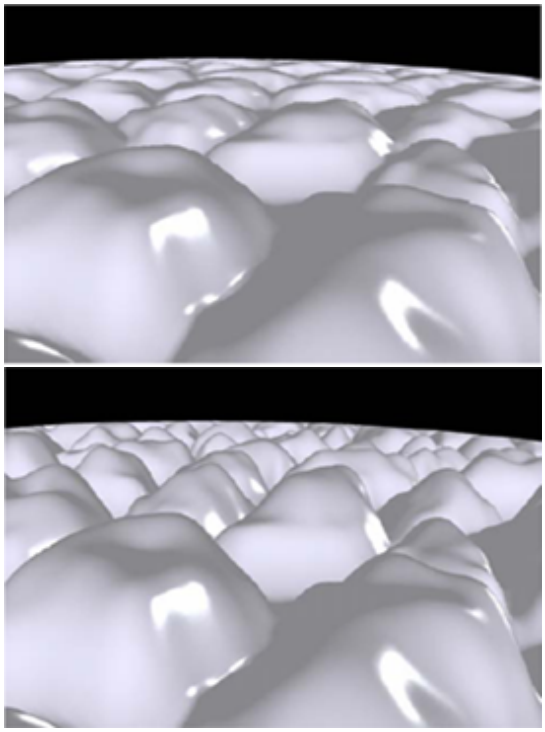

Figure 13: Top: relief mapping with binary search and depth bias shows some flattening; bottom: parallax occlusion mapping with linear search only, the same quality is achieved without any flattening [Tatarchuk 2006].

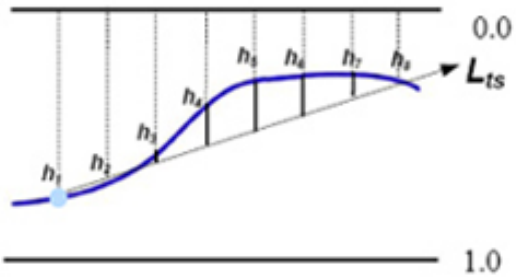

Figure 14: Sampling the height-field profile along the light ray direction [Tatarchuk 2006].

Another optimization is applied to shadows rendering. Only doing a intersection-no-intersection test would yield hard shadows. Instead Tatarchuk samples eight heights along the light vector, as illustrated in Figure 15. He then weights those samples according to their distance from h1 (the point of intersection). After subtracting h1

from the other samples the highest sample is used to calculate the amount of shadowing. Because the result is a value between 0.0 and 1.0 this technique renders smooth soft shadows. Performance can be increased by only doing this calculations for pixels facing the light source.

Figure 15: Top: the hard shadows produced by relief mapping, bottom: soft shadows in parallax occlusion mapping. The self occlusion in both images would not be achievable with parallax mapping. [Tatarchuk 2006].

In comparison we can see that the last techniques described here all try to solve the same problem. Parallax mapping (with offset limiting) offers great performance which is almost identical with standard bump mapping. It can produce suprisingly good results as can be seen in Figure 7. Still it may suffer from severe artifacts with highly detailed height maps and when used with certain viewing angles. Moreover it simply can't do self occlusion and doesn't provide an easy way for self shadowing, both techniques can make a severe difference in quality. Relief mapping and parallax occlusion mapping on the other hand offer all those things and achieve an extremely high level of quality but have far greater performance requirements.

3.5 Rendering Silhouettes

Like mentioned before as nice as those techniques based on per pixel ray tracing look, for a human observer there's still something missing. Figure 8 shows how much realism can be gained by correctly deforming the silhouette of an object.

In [Oliveira et al. 2006] the authors calculate a quadric surface for each vertex. They fit the quadric $z = ax^2 + by^2$ to each vertex, the details how the coefficients a, b are obtained can be found in the paper. Now when rendering at fragment level this quadric is used to check if the current viewing ray misses the surface or not.

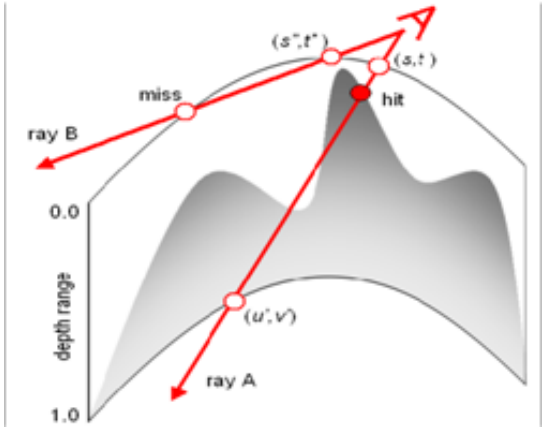

Figure 16: Rays returning to depth 0.0 while sampled are not rendered to create a correct silhouette [Oliveira et al. 2006].

This hit or miss test is done by calculating the distance of the viewing ray to the surface. In Figure 16 there are two red viewing rays shown. The surface of the polygonal mesh is the upper curve. From this curve the depth increases until it reaches 1.0 at the bottom of the height field. Now if one calculates the distance from the viewing ray to the upper curve at multiple samples a hit is found if the ray reaches a distance of 1.0. If the ray returns to a depth of 0.0 we have a miss and this pixel is not rendered. The distance is calulcated using the formula:

$$PQ_{distance} = P_z - (a(P_x)^2 + b(P_y)^2) \tag{5}$$

This is the z distance of the current point P along the viewing ray from the quadric surface $ax^2 + by^2$. It should be noted that this distance is only mathematically correct for surfaces with negative curvature as in Figure 16. For surfaces with positive curvature the z distance is only an approximation. However the authors have found that while there are differences between the exact and the approximated solution, the approximation only looks slightly different and most importantly it doesn't cause any artifacts or swimming and simply doesn't look wrong.

4 Shadow Mapping

This chapter describes what is probably the most widely used technique for generating shadows in computer graphics, shadow mapping. First the basic algorithm [Williams 1978] is described, followed by a discussion why its original implementation in a lot of cases doesn't provide acceptable quality. The remainder and biggest part of this chapter will then present several improvements and variations on shadow mapping proposed in recent years.

The basic idea behind shadow mapping is to test if the pixel currently being rendered is visible by the light source. If it is not it clearly has to be in shadow. This is done by first rendering the scene from the viewpoint of the light source, In this step only the depth values need to be rendered and stored in a depth map. Then when rendering the scene from the observers viewing position each pixel is transformed into light space, it's projected onto the shadow

map. Now we can compare its depth in the shadow map to its distance from the light. If it is farther from the light the pixel is in shadow.

As with most techniques based on a discrete buffer, shadow maps suffer from aliasing. The aliasing takes two forms, perspective and projective. The former one being the most common. It is caused by the projection of the shadow map which is done from the lights point of view. This means that most texels in the shadow map cover the region close to the light and in world space they get increasingly big the farther away they are from the light. Now when we render the scene from the observers viewpoint we would need the highest resolution near the observer and not near the light. This means that close to the camera the shadows are often made out of clearly visible big blocks of darker pixels (Figure 17).

Figure 17: Shadow maps suffer from perspective aliasing.

Projective aliasing occurs on surfaces which are parallel or almost parallel to light rays. In this case the shadow map texels become extremely stretched over a big area. This leads to severe artifacts if the viewing direction is close to being perpendicular to the light rays because we practically see the stretched shadow texels from the 'top'.

A commonly used technique to reduce all kinds of aliasing problems with shadow maps is percentage closer filtering (PCF). The idea is to filter the binary values of 'in shadow' or 'not in shadow' of the pixels surrounding the current one. This is done by calculating a percentage value of how many of those pixels are in shadow, giving more weight to the closer pixels. This value is then used to blend between 'completely lit' and 'in shadow'. A very nice side effect of this approach is that because of the filtering it generates soft shadows.

While this helps with aliasing it doesn't solve the problem itself. To finally improve shadow mapping to a point where it is an artifact free technique with constant good quality results a number of approaches have been proposed, of those three of the most promising are described in more detail in the following subchapters.

4.1 Perspective Shadow Maps

We have seen that the main reason for aliasing is that the resolution of the shadow map is not distributed in an optimal way. We would need the highest resolution near the camera and decreasing resolution with the distance to the camera growing. This lead to the idea described in [Stamminger and Drettakis 2002]. What the authors propose is to change the distribution of shadow map pixels by applying a perspective transform before rendering the shadow map. To be more specific they apply the same perspective transform which is used for the current camera image. This works because what the perspective transform does is enlarging objects near the camera and decreasing the size of objects farther away.

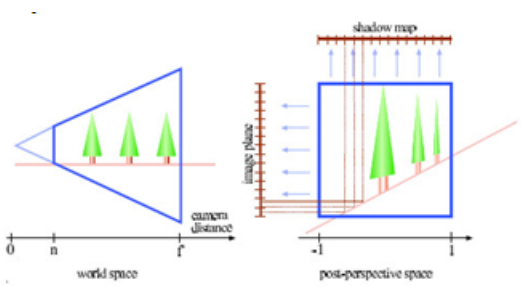

Figure 18: Perspective shadow mapping applies the perspective projection of the eye-view transform before rendering the shadow map [Stamminger and Drettakis 2002].

As shown in Figure 18 this increases the part of the shadow map covering the region near the camera and therefore increases the resolution in this area. To optimize the resolution of the shadows it's also important to focus the shadow map on the objects casting shadows which are actually within the viewing frustum. Figure 19 shows how this can be done. The first step is to construct a convex hull M enclosing all rays from points within the view frustum to the light source. M is guaranteed to include all objects casting shadows visible to the observer, but it probably includes more then is actually needed. As there can't be any shadows outside the lights frustum L we can intersect L with M. Finally we intersect the result with the bounding box of the whole scene S, generating the final volume N.

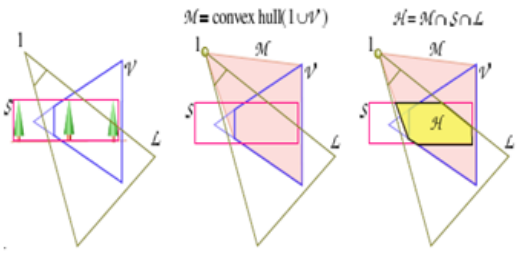

Figure 19: Focusing the shadow map on the relevant area N [Stamminger and Drettakis 2002].

Now while the basic idea behind this approach is relatively simple, applying the perspective projection in practice leads to a number of problems and special cases. Most of them are related to directional and point light sources. The perspective projection may actually change the type of a light from a directional to a point light or vice-versa. Figure 18 and 19 show the possible cases for what is originally a directional light source. In Figure 20 one can see on the left that a directional light shining from the front becomes a point light shining from the same direction, whereas a directional light from behind becomes a 'inverted' point light. This means that all light rays converge towards a single point. If the world space light is parallel to the viewing direction it is mapped to a point light just in front of the viewer. The last case is shown in Figure 18. A directional light parallel to the image plane is the only case where

no change happens. While the world space directional lights are located at infinity the post-perspective space point lights are located at a finite position, the so called infinity plane at z = (f + n) / (f - n).

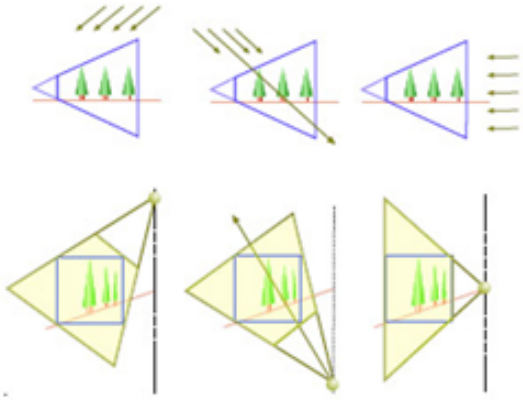

Figure 20: One set of the special cases perspective shadow mapping creates: three cases where directional lights become point lights in post-perspective space [Stamminger and Drettakis 2002].

Similar transformations happen for point lights. Point lights on a plane which is perpendicular to the view direction and goes through the view point become directional. In all other cases they remain point lights but may become inverted.

Another problem arises when shadow casting geometry which has to be included in the shadow map is behind the camera plane. This geometry would be projected to beyond the infinity plane. To solve this Stamminger uses a virtual camera shift which moves the camera backwards so that all geometry casting shadows into the viewing frustum is in front of the camera. This is only done for rendering the shadow map, not for rendering the image.

As one can see perspective shadow mapping suffers from a number of special cases which make an implementation quite involved. Moreover it gives the best results in only two specific cases. A directional light parallel to the image plane as shown in Figure 18 or a point light which lies on the camera plane. The latter case also being the optimal case for standard shadow mapping (like a miners lamp positioned just above the camera). In the other cases when the post-perspective light is a point light the quality greatly depends of the depth range of the light. If it is positioned on the infinity plane opposite the camera as in the right case in Figure 20 the results are the same as for standard uniform shadow maps. As this is the worst case for standard shadow maps, we can conclude that the best and worst cases for uniform and for perspective shadow mapping fall together and there's no improvement in neither of them.

We have seen that the perspective aliasing in shadow mapping can be reduced if not completely overcome by a reparameterization of the shadow map. The route taken in [Stamminger and Drettakis 2002] to use the perspective projection of the observer sounds logical but leads to a number of problems and it's improvement in quality is quite inconsistent. Luckily this is not the only reparameterization one can choose, an insight which lead to several variations of perspective shadow mapping being proposed one of those is presented in the next chapter.

4.2 Light Space Perspective Shadow Maps

The work by [Stamminger and Drettakis 2002] has introduced the idea that we can better distribute the resolution of a shadow map according to the perspectively transformed view of the observer. In an ideal world what we want is for every pixel in the shadow map, regardless of how far away it is from the observer, to have the same size in the final projected image. Perspective shadow mapping tries to minimize the magnification of shadow pixels near the camera. But as the distance to the observers camera grows it loses resolution even faster then uniform shadow mapping. Obviously what we need is something in the middle.

As mentioned before an important insight is that we are not confined to using the cameras projective transform as a reparameterization. We can rather choose any other perspective projection. Our goal is to change the distribution of shadow map pixels, so why not use a projection which effects mainly the shadow map plane while leaving the other axes relatively unchanged.

This is the idea presented in [Wimmer et al. 2004]. The authors suggest using a perspective transform based on the axes of light space, they therefore call this technique light space perspective shadow mapping (LiSPSM). This approach avoids the problem of geometry behind the camera being projected behind the infinity plane and probably most importantly it treats all lights as parallel light sources, instead of a number of special cases of light type conversions. The algorithm behind LiSPSM works as follows. First we focus the shadow map on the area containing all objects casting shadows into the view frame as has been explained in more detail in the last chapter. This is the volume B in Figure 21. Next we enclose B by a perspective frustum P which view vector is parallel to the shadow plane. The distance between the perspective reference point p and the near plane of the frustum P is the free parameter n. It determines the strength of the perspective warping effect. The projection P is then applied during rendering of the shadow map and for calculating the projected shadow map coordinates during scene rendering. For point lights we have to apply the perspective transform associated with the light before the frustum P is calculated. The combined transform doesn't have any singularities because the objects in the volume B are all in front of the light. Later the point light can then be treated as a directional light.

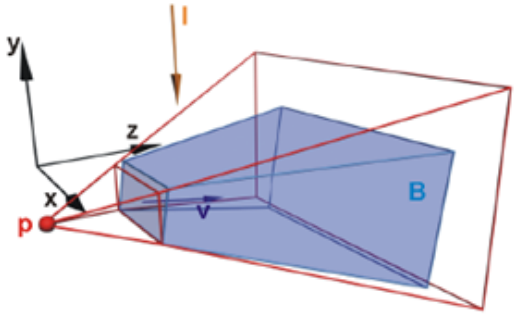

Figure 21: The perspective frustum used for LiSPSM [Wimmer et al. 2004].

In more detail the perspective transform P is constructed by first defining the y-axis as pointing in the same direction as the light vector I. The z-axis is perpendicular to the light vector and lies in the plane defined by the vectors I and v (the observers view vector). This means that the frustum P points in the observers view direction only rotated so that it's 'parallel' to the shadow map plane. The axis x is chosen so that it forms an orthogonal coordinate system with the other two axes. The y-coordinate of p is calculated as the middle between the minimum and maximum y-coordinates of the body B. The x-coordinate is taken from the transformed viewpoints x-coordinate. Finally the near and far plane of p are moved as close to the volume B as possible.

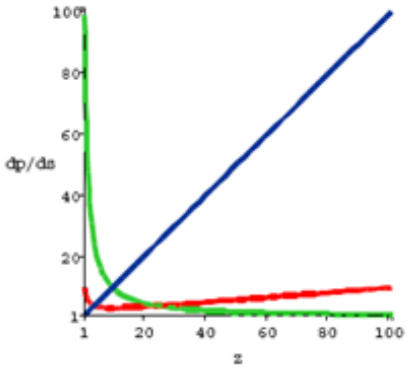

Figure 22: Comparison of the perspective aliasing error for different shadow mapping techniques [Wimmer et al. 2004].

To decide whether LiSPSM does indeed feature a better parameterization then perspective shadow mapping the authors conducted an extensive analysis of perspective aliasing errors. They compare uniform, perspective and light space perspective shadow mapping in an ideal and in a worst case scenario. Their probably most important find can be seen in Figure 22. It plots the perspective aliasing error for uniform shadow mapping (green), perspective (blue) and light space perspective (red). Ideally you want the factor dp/ds to be as close to one as possible over the whole z-range. All other values mean some amount of aliasing. As can be seen uniform mapping produces quite a big error near the camera which grows again with greater z values. Surprisingly perspective shadow mapping shows a really big error for everything which is not close to the camera, it trades resolution in a drastic way. LiSPSM shows a slightly bigger error close to the observer but it also doesn't grow nearly as much in the distance.

As a conclusion we can say that LiSPSM offers several improvements over perspective shadow maps. It eliminates practically all special cases and singularities, it distributes the perspective aliasing error more evenly and it should therefore be easier to implement.

4.3 Parallel Split Plane Shadow Maps

Another result of [Wimmer et al. 2004] we haven't mentioned yet is that the theoretical optimal parameterization for a shadow map is logarithmic. This would result in a constant factor dp/ds over the whole depth range. Now to render such a parameterization we would need a logarithmic projection, sadly this is not supported by any current graphics hardware. Even worse there's another problem with PSMs and LISPSMs and similar techniques which try to solve aliasing by redistributing the resolution of a single shadow map. Their parameterizations have to be adjusted every frame so in actual implementations they may suffer from f 'swimming' at the shadow borders. Moreover they all have some worst cases where their quality is not better then uniform shadow mapping.

The difficulties with warping a single shadow map and hardware lacking support for logarithmic projections led researchers to the idea of splitting the view frustum up into several shadow maps [Zhang et al. 2006]. The reparameterization can then be done by shifting the planes where the frustum is split. This avoids all problems with projection singularities or swimming artifacts. It should however be noted that this is a very fillrate intensive technique and therefore best suited for scenes with a single light source like large outdoor areas, which on the other hand is a scenario where most other shadow mapping techniques fail. A short overview of the algorithm looks as follows:

- divide the view frustum into multiple layers, the separations between them are the name giving parallel split planes
- split the light frustum into smaller ones, each one covering a single split layer of the view frustum and all objects casting shadows into it
- render a texture for each split layer
- render the scene with shadows, this may have to be done in multiple passes depending on the graphics hardware

Selecting the right split plane distances in the first step is central for the final shadow quality. The authors have analyzed three possible configurations. They show that dividing the view frame into splits which have the same size in world space provides the theoretically worst aliasing distribution. A logarithmic split scheme on the other hand would be optimal in theory. The problem is that this assumes that the shadow map is perfectly mapped to the view frame split and no resolution is wasted on objects which do not cast visible shadows. As this is not achievable in most situations a logarithmic scheme would produce split parts near the viewer which are too small and result in oversampling, and undersampling in split parts further away. They therefore choose what they call a practical split scheme defined as the average of a logarithmic and uniform scheme.

Figure 23: Three different depth splitting schemes: uniform, logarithmic and practical [Zhang et al. 2006].

In the next step just as with PSMs we have to focus the shadow maps on the relevant areas. Here we have another advantage of PSPSMs. Normally we would have to approximate the focus area for the whole view frustum at once, and this being an approximation would include quite big areas where there actually may not be a single object casting a shadow. Now that we have divided the view frustum into several splits we can do the focusing separately and in the end for many cases the union of the light frustums should be smaller than with one big approximation. This can be achieved by instead of using each view frustum split V_i as a whole we construct a bounding box axis aligned in light space around all objects in V_i called B_i. Those are the areas each light frustum has to focus on.

Splitting the view and the light frustum we have assumed that the light source is directional. To handle point lights just as with LiSPSM we apply the light's projective transform to the scene before rendering the shadow map and when looking up the current pixel in the shadow map during normal scene rendering.

For each split part V_i we have to render a shadow map T_i. In an implementation it works well to create each shadow map with the same resolution. This has the advantage that a feature of modern graphics hardware called multiple render targets (MRT) can be used. Instead of rendering each map in a separate pass it makes rendering them all in a single pass possible.

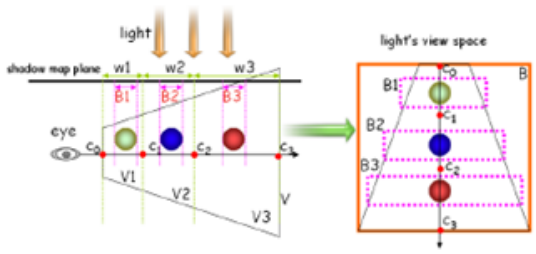

Figure 24: For the split sections view frustums axis aligned bounding boxes are created in light's space [Zhang et al. 2006].

After the shadow maps have been generated we have to use them for shadow generation during rendering of the scene. As with uniform shadow mapping each pixel has to be transformed into light space with the difference that we first have to decide into which light frustum it falls, which then determines which shadow map T_i will be used for this pixel. Depending on the capabilities of the graphics hardware this may mean that we have to render the whole scene in multiple passes, one for each view frustum split. Better performance can be achieved if the hardware's shader capabilities allow it to dynamically sample only the one shadow map required for the current pixel. For hardware not supporting dynamic branching it may be possible to sample all shadow maps for every fragment and then only use the results from the appropriate one. It should be noted that with the usually done filtering of the shadows, sampling all maps may lead to unacceptable performance. The projected depth of each split plane during rendering can be calculated by plugging the view space depth into the projection formula of the used graphics API.

Performance results obtained by the authors show that with this technique highly interactive frame rates can be achieved. They recommend using three split parts for a good compromise between performance and quality. Their technique has for example been recently adopted quite successfully by several pc game developers for generating outdoor shadows.

4.4 Variance Shadow Maps

All those techniques we have mentioned until now can do a lot to reduce aliasing artifacts caused by perspective projection. Still it remains that shadow maps by default generate hard shadows with jagged edges. The premiere approach to combat this has already been mentioned shortly, percentage closer filtering (PCF).

The key insight here is that it wouldn't help to filter the depth values in the shadow map. If you have a single occluder casting a shadow on a planar surface and when rendering the scene use bilinear filtering on the shadow map, this would simply increase the area shadowed, it would still result in hard edged shadows. So what PCF does is it filters the results of the depth comparisons instead of the depth values themselfes. For example for a 3x3 PCF filter kernel you would do nine depth comparisons, then multiply each result by a weight and then add them all up to get a percentage how much the current pixel is in shadow.

The biggest problem with this approach is that to get good results one has to use a lot of samples per pixels. The best quality can be achieved by bilinear filtering the current PCF result (Figure 26, the third image from the left). As this again multiplies the number of samples required and adds the computational burden of bilinear filtering this is not really an option for most real time applications.

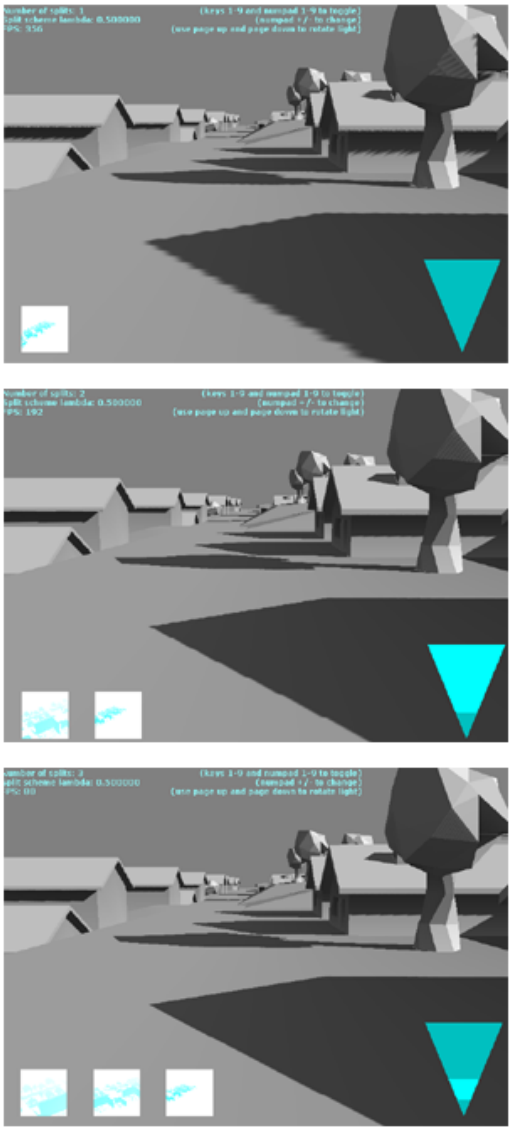

Figure 25: Comparison of an increasing number of depth splits, using 1,2 and 3 shadow maps. At the bottom of the images the shadow maps and the depth division scheme of the view frustum can be seen.

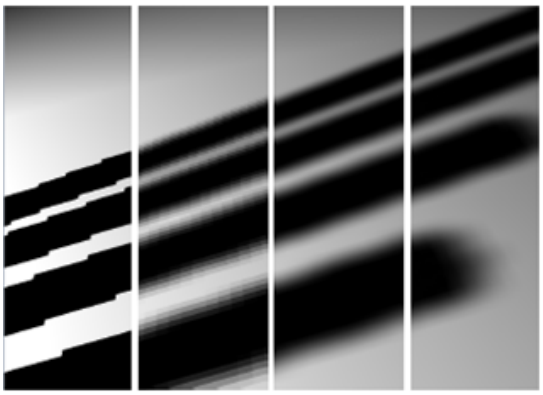

Figure 26: Comparison of different shadow mapping filtering methods, from left to right: no filtering, 5x5 PCF, 5x5 bilinear PCF and variance shadow maps with 5x5 separable gaussian blur [Donnelly and Lauritzen 2006].

What we would need is a technique which can make use of the mipmap, anisotropic and bilinear filter methods modern graphics hardware already provides. That's where the method proposed in [Donnelly and Lauritzen 2006] comes into play. What they do is instead of storing the depth of a single point at each pixel they store a distribution of depths at each texel. To keep the data required to a minimum they only store an approximation of this distribution, the first and second moments. Those are the mean and the mean squared depth. What this distribution can be used for, is finding the answer to one simple question: How many of the texels surrounding the current shadow map texel have a equal or greater depth then the pixel we are currently drawing. Those texels are lit. So we can use the result to do percentage closer filtering. The important difference with variance shadow maps is we only have to look up the single current shadow map texel and can use the stored mean and variance to do the filtering. With standard PCF we would have to look up several of the surround texels as well. Basically with the addition of storing the variance we are including knowledge about the surrounding texels.

What happens if we filter such a shadow map is that the first and second mean get averaged. Mathematically this is an approximation of the average of the distributions. This average can then be used to compute an upper bound for how much of the distribution is equally or more distant then the surface we are currently drawing. Because everything that is more distant is lit this tells us how much light is reaching the surface.

Note that when we generate a variance shadow map we only store the depth and the squared depth of the current texel just as with standard shadow maps. One could say we are storing distributions with a single value. Only after filtering the shadow map it contains averaged distributions over several depth values. Without filtering we would get the same results as with standard shadow maps.

The great thing about being allowed to filter the depth values is that we can now use all the filter options for the shadow map the hardware provides like mipmapping and anisotropic filtering. Even more important we can pre-filter the shadow map for example with a 5x5 gaussian blur. The results which can be obtained by that are shown the righter most image in Figure 26. Note that to get a comparable result, with bilinear PCF we would have to sacrifice way more precious processing power.

So we can use a variance shadow map not only to query whether a pixel is in shadow or not but also if it is on the edge of the shadow how much light is reaching it. The mathematical reasoning why that works goes like this: In the shadow map we are storing the first and the second moments M_1 and M_2. The first moment is the average depth, the second moment is the average of the squared depths. M_1 can be used to do the standard depth comparison as with normal shadow mapping. Let's call the depth of the current pixel t. If $t < M_1$ the pixel is lit and we can draw it with full lighting. Otherwise it is shadowed and we now use M_1 and M_2 to calculate how many of the surrounding pixel may still be lit arriving at a value with which we can scale the light intensity. Basically what we need to know is: in the distribution of depth values surrounding the current shadow map texel how many of them have an equal or greater depth then t (and are therefore unshadowed). First we calculate the mean μ and the variance σ of the distribution given by M_1 and M_2:

$$\mu = M_1 \tag{6}$$

$$\sigma = M_2 - M_1^2 \tag{7}$$

The variance can be seen as a value representing the width of a distribution. It tells us how far away the highest and lowest values in the distribution are from the mean. With variance shadow maps it tells us how far the greatest and smallest depth values are from the average depth value over a certain filter region. Now with this two values we can use a theorem from statistics called Chebychev's inequality to calculate an upper bound on how many of the texels depth values are ¿= t:

$$P(x >= t) < \frac{\sigma}{\sigma + (t-\mu)^2} \tag{8}$$

The probability P(x ¿= t) for a texel's depth x to be ¿= t is exactly the value we need to do percentage closer filtering. Note that in the middle of the shadow this probability would be close to 0 because none of the surrounding texels are lit. Only on the edge of the shadow this value will start to grow. When looking at the inequality it becomes clear that this happens because the average depth μ stored at this area in the shadow map is getting closer and closer to the current pixels' depth t, until they are the same in the lit areas and $(t-\mu)^2$ becomes zero. So the whole inequality moves toward 1 (σ/σ), which means that the probability for a pixel being lit is 100

Now like already stated the inequality only gives an upper bound there's no guarantee that this is the exact value P(x ¿= t). In [Donnelly and Lauritzen 2006] however it is shown that for the simple case of a single occluder casting a shadow on a planar surface the inequality actually becomes an equality and therefore gives the exact value. While not every case is that simple in practice this approximation looks good enough. A more detailed breakdown of the steps required for rendering variance shadow maps is:

- render the scene from the lights' point of view into a texture outputting the depth and squared depth ideally to a multi component floating point texture, scale those values so they are in the range of [-1,1] to avoid overflowing the numeric boundaries of the floating point numbers and to get the highest precision
- prefilter the variance shadow map using a two pass gaussian blur
- generate mipmaps for the shadow map
- Render the scene from the observer's point of view. For each pixel compare the current depth t with μ from the shadow map. If it is smaller then μ the pixel is fully lit. Else calculate

Figure 27: PSPSM are used in combination with variance shadow maps in the PC game 'Crysis' for rendering completely dynamic high quality soft shadows.

the variance σ and use Chebychev's inequality to calculate the factor with which to scale the light intensity.

The great thing about variance shadow maps is that they can be used in combination with all the reparametrization techniques we have mentioned before. They do the same thing PCF does but offer a better quality performance ratio. The result achieved by a recent PC game using variance shadow maps in combination with PSPSM can be seen in Figure 27.

5 Conclusion

This paper covered two very important areas in real time computer graphics, rendering of surface detail without additional geometry and shadow rendering. In both fields the most recent developments have finally gotten them to a stage where they can be regarded as highly robust completely automatic algorithms with a performance suitable for real time rendering.

Shadow mapping is a technique developed in the seventies but has always suffered from severe aliasing artifacts. In offline rendering those could be overcome by using extremely high shadow map resolutions and by adjusting the parameters for shadow rendering by hand frame by frame. For real time graphics both of those methods are not an option. We have seen that by dividing the view frustum into multiple depth layers as done in parallel split plane shadow mapping and filtering the results with variance shadow mapping results of a very high quality with high frame rates can finally be achieved. Those can be improved even further by using one of the described reparameterization schemes like PSM or LiSPSM.

References

BLINN, J. F. 1978. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 286–292.

CATMULL, E. E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis.

DONNELLY, W., AND LAURITZEN, A. 2006. Variance shadow maps. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, ACM Press, New York, NY, USA, 161–165.

HEIDRICH, W., AND SEIDEL, H.-P. 1998. Ray-tracing procedural displacement shaders. In *Graphics Interface*, 8–16.

KANEKO, T., TAKAHEI, T., INAMI, M., AND KAWAKAMI, N. 2001. Detailed shape representation with parallax mapping. *;Int Conf Artif Real Telexistence 11*, 2, 205–208.

KAUTZ, J., AND SEIDEL, H.-P. 2001. Hardware accelerated displacement mapping for image based rendering. In *Proceedings of Graphics Interface 2001*, B. Watson and J. W. Buchanan, Eds., 61–70.

MAX, N. L. 1988. Horizon mapping: shadows for bump-mapped surfaces. In *The Visual Computer*, Springer Berlin, Heidelberg, 109–117.

MEYER, A., AND NEYRET, F. Interactive volumetric textures. 157–168.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000. Relief texture mapping. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 359–368.

PEERCY, M., AIREY, J., AND CABRAL, B. 1997. Efficient bump mapping hardware. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 303–306.

PHARR, M., AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop 1996*, Springer Wien, New York City, NY, X. Pueyo and P. Schröder, Eds., 31–40.

POLICARPO, F., OLIVEIRA, M. M., AND COMBA, J. L. D. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM Press, New York, NY, USA, 935–935.

SCHAUFLER, G., AND PRIGLINGER, M. 1999. Efficient displacement mapping by image warping. In *10th Eurographics Workshop on Rendering*, Springer Verlag, New York City, NY, X. Pueyo and P. Schröder, Eds.

SLOAN, P.-P., AND COHEN, M. F., 2000. Interactive horizon mapping. Eurographics Rendering Workshop 2000.

SMITS, B., SHIRLEY, P., AND STARK, M., 2000. Direct ray tracing of displacement mapped triangles.

STAMMINGER, M., AND DRETTAKIS, G. 2002. Perspective shadow maps. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 557–562.

TATARCHUK, N. 2006. Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, ACM Press, New York, NY, USA, 81–112.

WELSH, T., 2004. Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces. Infiscape Corporation.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference*

on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 270–274.

WIMMER, M., SCHERZER, D., AND PURGATHOFER, W. 2004. Light space perspective shadow maps. In *Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering)*, Eurographics Association, A. Keller and H. W. Jensen, Eds., Eurographics, 143–151.

ZHANG, F., SUN, H., XU, L., AND LUN, L. K. 2006. Parallel-split shadow maps for large-scale virtual environments. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, ACM Press, New York, NY, USA, 311–318.