To appear in a IEEE TCVG sponsored conference proceedings

Fitted Virtual Shadow Maps

Markus Giegl* Michael Wimmer*

*Vienna University of Technology

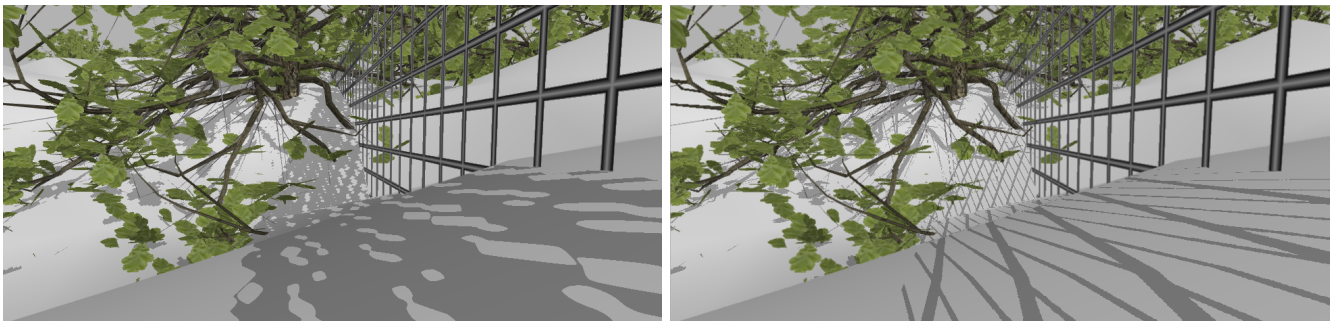


Figure 1: Left: Shadow map reparametrization techniques (lightspace perspective shadow maps is used here) alone cannot guarantee subpixel accuracy for neither all light directions nor large scenes, even with the largest shadow map supported by the GPU. Right: Fitted Virtual Shadow Maps allow the scene to be shadowed with subpixel accuracy.

ABSTRACT

Too little shadow map resolution and resulting undersampling artifacts, perspective and projection aliasing, have long been a fundamental problem of shadowing scenes with shadow mapping.

We present a new smart, real-time shadow mapping algorithm that virtually increases the resolution of the shadow map beyond the GPU hardware limit where needed. We first sample the scene from the eye-point on the GPU to get the needed shadow map resolution in different parts of the scene. We then process the resulting data on the CPU and finally arrive at a hierarchical grid structure, which we traverse in kd-tree fashion, shadowing the scene with shadow map tiles where needed.

Shadow quality can be traded for speed through an intuitive parameter, with a homogenous quality reduction in the whole scene, down to normal shadow mapping. This allows the algorithm to be used on a wide range of hardware.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: shadow, shadow map, large environments, realtime shadowing

1 INTRODUCTION

Shadow mapping is a very appealing approach to employ rasterization to solve the first hit visibility problem and use this result to calculate the direct light shadowing of a scene. This elegant approach has just one fundamental problem: The shadow map must contain enough information to allow the visibility queries to be answered with subpixel accuracy for a given frame buffer resolution, otherwise aliasing artifacts will be visible. If this information is not contained in the shadow map, then all any algorithm can do is try to mask these artifacts, e.g. by means of filtering.

Please see the introduction section in [8] for a definition of the two types of shadow map aliasing, projection and perspective aliasing.

Aila *et al* [1] and Johnson *et al* [10] elegantly bypass the aliasing problem altogether but depend on hardware extensions for realtime performance which are not currently available.

A straightforward way to increase the information contained in a uniform shadow map is to increase the resolution of the shadow map texture. This becomes impractical very fast, due to its quadratic increase in memory consumption (on current hardware the maximum supported texture size, typically 4096×4096 is the limiting factor, even before running out of memory).

This paper presents an algorithm which runs on current graphics hardware and increases the effective shadow map resolution available to shadow the scene, while avoiding the quadratic increase in memory consumption.

From a practical point of view a common complaint that e.g. game developers have with the popular reparametrization techniques is that there is a large quality difference between the best and the worst case (as also shown by Lloyd *et al* [12]). The presented algorithm addresses this criticism and allows for the same shadow quality in all cases, while being orthogonal to shadow map reparametrization techniques and shadow map focusing. It can therefore be combined with these techniques, and we have done so for Light Space Perspective Shadow Maps [19] together with shadow map focusing [3].

1.1 Abbreviations

The following is a list of abbreviations used in this paper:

FVSMs: Fitted Virtual Shadow Maps, LiSPSM: Lightspace Perspective Shadow Maps [19], SM: Shadow Map, SMing: Shadow Mapping, SM-Tile: Shadow Map Tile (see section 3), SMTMM: Shadow Map Tile Mapping Map (see section 4).

*{giegl|wimmer}@cg.tuwien.ac.at, 1040 Vienna, Austria

2 PREVIOUS WORK

The two most important categories of shadow algorithms are *shadow volumes* [5] and *shadow mapping* [18].

Most of the shadow map publications try to solve the problem of aliasing artifacts. Percentage closer filtering [15] alleviates reprojection problems by sampling the shadow map. In Variance Shadow Maps, Donelly *et al* [6] use the variance of the depth values to further improve the shadow map sampling results. A number of papers have tried to solve the *perspective aliasing* coming from the perspective view frustum projection. Originally pioneered by Stamminger and Drettakis [16], who try to remove perspective aliasing by subjecting the shadow map to the same perspective transform as the viewer, this idea was later refined by Martin and Tan [13] with Trapezoidal Shadow Maps, Wimmer *et al* [19] with Light Space Perspective Shadow Maps and Chong *et al* [4] with A Lixel for Every Pixel. However, all shadow map reparametrization methods deal only with perspective aliasing. They cannot increase the principal resolution of shadow maps, which would be necessary for example to improve projection aliasing, or in cases where the scene is simply too large for the SM resolution. Furthermore, they work well only for the case that light and view direction are orthogonal. If these directions are parallel, they have to revert to uniform shadow mapping because the shadow map parametrization runs across the whole screen, not from near to distant points. Recently Lloyd *et al* [12] have studied the use of more than one shadow map applied to the sides or slices of the view frustum together with reparametrization techniques intensively, with interesting results; all these approaches can however only deal with perspective aliasing, but can do nothing to alleviate projection aliasing. The work presented in this paper aims to increase the resolution of shadow maps regardless of the view frustum orientation or whether the artifacts come from perspective or projection aliasing.

Another approach to solve the aliasing problem are adaptive shadow maps [7] (see also section 4.8 below for a comparison with Fitted Virtual Shadow Maps), where shadow maps are stored in a hierarchical fashion in order to provide more resolution where it is required due to different aliasing artifacts. However, the approach requires multiple readbacks and does not map well to current graphics hardware. Lefohn [11] has proposed an extension that makes better use of the GPU, whereas Arvo [2] slices the light view to increase the resolution of the SM.

Second depth shadow mapping [17] can be used to reduce problems due to depth quantization and self occlusions. Brabec *et al* [3] improve uniform shadow map quality by focusing the shadow map to the intersection of the view frustum with the scene.

Recently Queried Virtual Shadow Maps [8] have used the occlusion query mechanism of GPUs to adaptively refine the shadow map in quadtree fashion based on counting the number of pixels that changed in the shadow during the last refinement step.

An excellent overview of shadow mapping and shadow algorithms in general can also be found in Möller and Haines' Real-Time Rendering book [14], as well as in [9].

3 VIRTUAL TILED SHADOW MAPPING

The following section explaining Virtual Tiled Shadow Mapping, on which Fitted Virtual Shadow Mapping is based, is reproduced from [8]. Virtual Tiled Shadow Mapping is a brute-force approach for increasing the resolution of the shadow map beyond the maximum texture size supported by the hardware. The basic algorithm works as follows:

1. Allocate the biggest shadow map texture supported by the GPU. For example 4096^2.
2. Partition the shadow map along the shadow map x- and y-axis into $n \times n$ (e.g. 16×16) equally-sized **SM-tiles** (each tile using the full shadow map texture resolution of e.g. 4096^2 texels, i.e. the effective resolution of the full shadow map in this example is $(16 * 4096)^2 = 65536^2$).

   For each tile

   (a) Render a shadow map into the shadow map texture (overwriting the shadow map for the previous tile).

   (b) Use it immediately to shadow (modulate) the part of the scene which is covered by the current shadow map tile.

There are two ways to implement the loop over the tiles: multi-pass shadowing and virtual deferred shadowing.

3.1 Multi-Pass Shadowing

One way to apply successive shadow map tiles to the scene is by multi-pass rendering. In the first pass, the scene is rendered normally (with full shading and depth-writes enabled), with the first shadow-map tile applied to it. For each subsequent shadow-map tile, the scene is rendered again, but only shadow mapping using the relevant tile is applied to the frame buffer. Pixels falling outside the shadow map tile are suppressed. Depth writes and shading are disabled and the depth comparison function is set to `EQUAL` in those passes (depending on driver support, it can make sense to substitute `LESSEQUAL` for `EQUAL`).

3.2 Deferred Shadowing

Multi-pass shadowing, although easy to implement, comes with a significant performance overhead of rendering the whole scene several times. To speed up the application of the shadow map tiles to the scene, we use a variation of deferred shading we call "deferred shadowing" where the shadowing is done using a linear depth buffer of the scene instead of re-rasterizing the scene geometry and the information needed to do the next shadowing pass, i.e., the next shadow map tile, is created on the fly between the passes. The scene is first rendered to a texture that stores eye-space depth, called the "**Eye-Space Depth Buffer**". Each subsequent tiled shadowing pass can then read this texture and calculate the world-space position of the visible surface at each pixel using the screen coordinates and the depth stored in the Eye-Space Depth Buffer. The world-space position is then shadowed using the shadow map tile as before. Note that storing the unmodified eye-space z-coordinate in the Eye-Space Depth Buffer guarantees that the shadow map lookup produces the same results as if the original scene objects were used for shadow mapping. This is important because any other method of obtaining the z-value, e.g., using window-space z-coordinates (which is highly non-linear) or a fixed-precision w-buffer (if it were still supported on current hardware) would inevitably lead to image artifacts. In detail this works as follows:

1. In a first pass, render the scene as described above, but into a 4 component 32bit floating point render target. In the pixel shader, store the unmodified eye-space z-coordinate into the α-component. This component forms the Eye-Space Depth Buffer (however for simplicity, we refer to the whole 4 component target as the Eye-Space Depth Buffer). The color of

each pixel in the object when lit by this light (ignoring shadowing) is written to the RGB channels.

2. For each shadow-map tile
   (a) Render a shadow map into the shadow map texture as with Multi-Pass Shadow Mapping.
   (b) Instead of rendering the geometry for the whole scene again, render a full-screen quad with the Eye-Space Depth Buffer bound as a texture.
   (c) In the pixel shader for each fragment, look up the eye-space depth of the fragment in the Eye-Space Depth Buffer's alpha-channel and unproject it into world space (see below). Using the unprojected fragment, calculate the shadowing term. Then modulate the already shaded RGB value from the Eye-Space Depth Buffer with the shadowing term.
   (d) The resulting shaded and possibly shadowed fragment is then written to the frame buffer.

The pixel shader operations in the individual passes are quite straightforward, with the exception of the unproject operation. Unlike a standard viewport unprojection, which transforms from window (x_w, y_w, z_w)-coordinates to eyespace (x_e, y_e, z_e)-coordinates, this operation has to deduce eye-space (x_e, y_e, z_e) from (x_w, y_w) (given as texture coordinates, i.e. running from 0 to 1) and z_e. This can be done using the following matrix transform:

$$\begin{pmatrix} x_e \\ y_e \\ z_e \end{pmatrix} = z_e \cdot \begin{pmatrix} \frac{1}{a_x} & 0 & -\frac{b_x}{a_x} \\ 0 & \frac{1}{a_y} & -\frac{b_y}{a_y} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 0 & 1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_w \\ y_w \\ 1 \end{pmatrix} \quad (1)$$

where the parameters a_x, a_y, b_x, b_y in the first matrix should be taken from the projection matrix P supplied to the graphics API:

$$P = \begin{pmatrix} a_x & 0 & b_x & 0 \\ 0 & a_y & b_y & 0 \\ 0 & 0 & \dots & \dots \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

4 Fitted Virtual Shadow Mapping

4.1 Fitted Virtual Shadow Maps: Smart Refinement Where Necessary

Like Queried Virtual Shadow Maps, Fitted Virtual Shadow Maps aim to refine the shadow map only where needed. The algorithm is also designed to be fast enough to do the full refinement each frame. Instead of counting the number of changed shadow pixels in the last refinement step, and use this metric to decide whether to further refine a SM-tile into 4 sub-tiles, FVSMs try to discern beforehand what SM-resolution is needed where in the scene.

The following gives an overview over the FVSM algorithm (please see the following sections for details):

1. Render the view-space linear depth information of the scene into the Eye-Space Depth Buffer, as above under Virtual Tiled Shadow Mapping.
2. Use the Eye-Space Depth Buffer bound to a fragment-shader to create what we call the "**Shadow Map Tile Mapping Map**" ("**SMTMM**"; see Figure 3). The SMTMM contains information for each pixel in the scene about 1) where the pixel will query the shadow map, when inquiring whether it lies in the shadow, and 2) what resolution the shadow map would require along each SM-axis at this position to supply subpixel accuracy when answering the shadow map query.
3. Transfer the SMTMM to CPU memory and process it to create the "**Shadow Map Tile Grid**". The Shadow Map Tile Grid contains information about what resolution each SM-tile of a virtual $n \times n$ Tiled SM would need along each SM-axis to supply subpixel accuracy when used to shadow the scene.
4. Construct the "**Shadow Map Tile Grid Pyramid**" above the Shadow Map Tile Grid, by pulling up the maximum needed SM-tile resolution along each axis.
5. Traverse the Shadow Map Tile Grid Pyramid recursively top down, building an implicit kd-tree of SM-tiles. When the resolution requirement of a such created SM-tile can be satisfied along both SM-axes with a SM-tile-texture with dimensions supported by the GPU, the corresponding SM-tile shadow map is created and immediately used to shadow its part of the scene as under Deferred Shadowing (section 3.2) above.

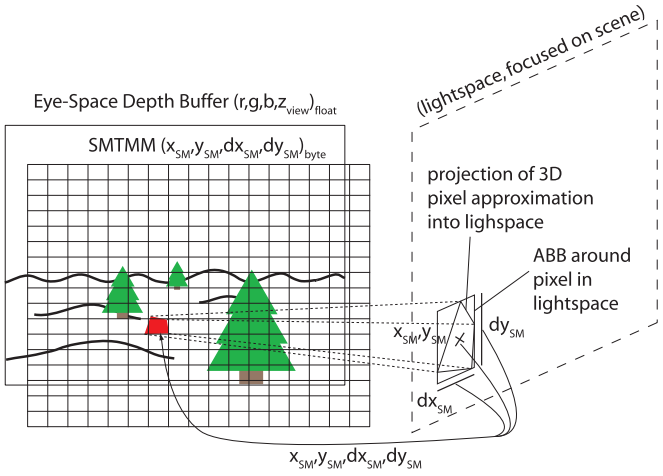


Figure 3: Shadow Map Tile Mapping Map (SMTMM) creation.

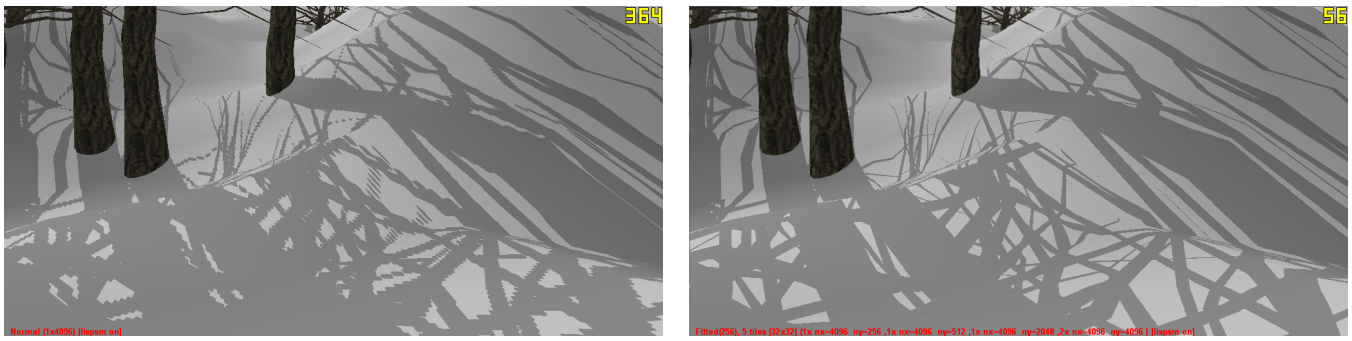
The following explains the steps in more detail and in section 4.10 introduces an important optimization to the basic algorithm:

4.2 Virtual Shadow Mapping Preparation: Eye-Space Depth Buffer

First, we render the view-space depth information of the scene into the Eye-Space Depth Buffer, as above under 3.2. For efficiency reasons we again use a $4 \times float$ RGBA-buffer and at the same time render into it the unshadowed RGB color of the scene, so we do not have to rerender the scene (Note: If an application is using depth-first rendering, i.e. starting with a Z-only pass, then a $1 \times float$ buffer should be used for the Eye-Space Depth Buffer for the Z-only pass, with a conventional $1/z$-depth buffer attached).

4.3 Shadow Map Tile Mapping Map Creation

The "**Shadow Map Tile Mapping Map**" ("**SMTMM**") is a $4 \times byte$ buffer. One can think of it as being laid on top of the frame



(a) 4096^2 normal SM

(b) FVSM with 32×32 max SM-tiles

Figure 2: Quality comparison at end of test path through winter forest (LiSPSM SM reparametrization active in all screenshots).

buffer, normally having less resolution than the frame buffer, and containing information about the shadow map resolution needs of the scene in the area that each SMTMM "pixel" covers. Figure 3 gives a graphical representation of the SMTMM.

The first two byte values in each SMTMM entry ("pixel") contain information about the position where the center of the frame buffer rectangle will query the shadow map; the last two byte entries represent the resolution needed along each SM-axis at the position in the shadow map. We use byte values for the entries, to keep the read back operation and the CPU processing in the next step fast; for the same reason, the SMTMM is normally chosen to have lower resolution than the frame buffer (see the results section for a practical range of values). Using byte values for the shadow map gives us information about the needed SM-resolution discretized to a 256×256 grid of SM-tiles; this is no restriction in practice, since one finds that for 4096^2 SM-tile-textures, a maximum refinement along each SM-axis of 16 to 32 (i.e. max 32×32 SM-tiles) gives subpixel accuracy even for large scenes.

The position in the shadow map is calculated in the pixel shader by transforming the screen-space coordinates (x_w, y_w) of the pixel (passed to the pixel shader as texture coordinates) and the eye-space z (=depth) entry z_e, read from the Eye-Space Depth Buffer, into eye-space (x_e, y_e, z_e) using the matrix given in section 3.2 (formula (1)); from there it is transformed into the light space of the shadow map. Since the coordinates will already be in the range $[0,1]$, simply outputting them to the $4 \times byte$ SMTMM surface will automatically lead to conversion into $[0,255]$ byte range by the graphics hardware.

The resolution requirement along each SM-axis is approximated as follows in the pixel shader: First we calculate the (x,y)-coordinates of the neighboring pixels in x- and y-direction in $[0,1]^2$ (i.e. the left/right and upper/lower neighbors position in texture coordinates) from the texture coordinate of the current pixel passed to the pixel-shader. We then use these texture coordinate to look up the corresponding view-space depth values in the Eye-Space Depth Buffer; from these, we calculate the smaller absolute Δz along the x- and y-axis, Δz_x and Δz_y. We then use these Δz values together with the x,y-coodinates of the neighboring pixels to construct an approximate rectangle representing the current pixel in space. Then we project this rectangle into SM-space, and calculate a SM-axis-aligned bounding box around it. The half length of each of this bounding box's extent, Δ_{sm_axis}, with $sm\_axis = \{0,1\}$, is then used as the base measure for the required SM-resolution along each SM-axis at this point.

To then quantize the needed SM-resolution along the SM-axis into a byte value, we use the following formula:
$-log_2(round(\Delta_{sm\_axis} + float2(0.5,0.5))/256$
(i.e. we output it as a logarithmic value normalized to the range $[0,1]$, which the the graphics hardware again automatically converts to byte range).

The full SMTMM creation pixel shader can be found in Appendix A.

4.4 Shadow Map Tile Grid Creation

To create the "**Shadow Map Tile Grid**" ("**SMTG**") we then read back the SMTMM to CPU memory. In practice it suffices for the SMTMM to have lower resolution than the frame buffer, e.g. 256×256, which makes the readback and CPU processing fast (note that the equality of the SMTMM dimension of 256 in this example and the number 256 of distinct values in the SMTMM entries is coincidental).

What we want is a $n \times n$ SM-tile-grid structure, with each grid cell containing the needed resolution along each SM-axis and the **screen space bounding rectangle** for each SM-tile, an axis aligned rectangle around the pixels on screen that are affected by the SM-tile. As in brute force $n \times n$ Virtual Tiled Shadow Mapping above, n is the maximum number of SM-slices along each SM-axis we would like to allow; a typical value for n would be 16 or 32, corresponding to 256 or 1024 SM tiles for Virtual Tiled Shadow Maps.

The random memory access ability of the CPU is well suited for this task; after having read back the SMTMM, we lock the surface and process each pixel entry: We use the stored information about the SM-tile position to access its corresponding SM-tile-grid cell, and update 1) its needed SM-resolution entries along each SM-axis (minimally by maximizing the existing value with the entries in the SMTMM; see below for details) and 2) its screen space bounding rectangle (through extending it to enclose the pixel position of the current pixel in the SMTMM).

Since the SMTMM generally is chosen to have a much larger resolution than the SM-tile-grid, e.g. 256 entries per axis compared to e.g. 32, data from several SMTMM entries will be accumulated in the same SM-tile-grid cell.

Minimally it would suffice to only record the maximum needed resolution along each SM-axis in each grid cell; however, to be able to allow for discarding very few pixels requiring a large resolution later on (which can come from e.g. a very small area on screen having an orientation which leads to large projection aliasing), we

actually count the number of pixels in each grid cell requiring a certain resolution. We use fixed size arrays at each SM-tile-grid-cell, to hold the pixel count statistics. To allow us to use fixed size arrays and keep them small, we count the number of pixels below a resolution η_0 and above threshold resolution η_1 in one array entry respectively, and the number of pixels needing a resolution in between each in their own entry; this is to keep cache locality high, since we are not interested in the detailed statistics of pixels with very small resolution requirements, because evidently they are easy to fulfill, and hypothetical pixels with extremely high resolution requirements, which do not occur in practice. See the results section for practical values for η_0 and η_1.

The following pseudocode illustrates the basic version of the Shadow Map Tile Grid Creation:

```
// smtg ... instance of the SMTG
// shift ... shift-converts from the SMTMM SM-coordinates
// entries to SMTG ones (e.g. [0,255] => [0,31])
const int shift = 256/smtg.n
// smtmm ... instance of the SMTMM
// smtmm.n ... extent of SMTMM along both axes
for ix_smtmm = 0 to smtmm.n - 1
 for iy_smtmm = 0 to smtmm.n - 1
  SMTMM_Cell c_smtmm = smtmm(ix_smtmm,iy_smtmm)
  SMTG_Cell c_smtg =
   smtg(smtmm.ix_sm >> shift,smtmm.iy_sm >> shift)
  // Update the screen-space, axis aligned bounding box
  // around the SM-tile
  c_smtg.abb_screen.ExpandToIncludePoint(
   ix_smtmm/smtmm.n,iy_smtmm/smtmm.n
  )
  // Update the maximum needed SM-resolution
  c_smtg.sm_res_x = MAX(c_smtg.sm_res_x,c_smtmm.sm_res_x)
  c_smtg.sm_res_y = MAX(c_smtg.sm_res_y,c_smtmm.sm_res_y)
```

4.5 Shadow Map Tile Grid Pyramid Creation

After we have filled the SM-tile-grid with data from the SMTMM, we proceed by building a pyramid ("**Shadow Map Tile Grid Pyramid**", "**SMTGP**") of SM-tile grids on top of it, where each successive grid has halved dimensions of its predecessor and the needed resolution along each SM-axis is the maximum of the corresponding 2×2 grid cells in the predecessor grid; i.e. we pull up the needed resolution along each SM-axis by replacing 2×2 cells with one cell in the next smaller grid, containing the maximum value of each of the 4 cells and the screen space bounding rectangle around all 4 bounding rectangles.

Note that the needed resolution along each SM-axis refers to a hypothetical, focused shadow map needed to shadow the whole scene with subpixel accuracy; replacing the $2 \times 2 = 4$ values with their maximum in the parent cell in the next higher Shadow Map Tile Grid Pyramid level is therefore not a heuristic, but a mathematically exact operation (The 1×1 top level grid then contains the resolution needs for this hypothetical single shadow map which would give subpixel accuracy for the current frame buffer dimensions; as can be seen in the Results section below, the resolution requirements become ≥ 131072 even for medium sized scenes, $16\times$ larger than the maximum texture dimension of 8192 currently supported in hardware; it would require more than 64 GB to store).

The Shadow Map Tile Grid Pyramid Creation in pseudocode:

```
// smtg ... instance of the initial SMTG
// smtg.n ... extent of SMTG along both axes
// i_pyramid ... SMTGP index
const int i_pyramid = log2(smtg.n)
while(i_pyramid > 0)
 // smtgp ... instance of the SMTGP
 smtgp(i_pyramid) = smtg
 for ix = 0 to smtgp(i_pyramid).n - 1
  for iy = 0 to smtgp(i_pyramid).n - 1
   SMTGP_Grid_Cell c_curr = smtgp(i)(ix,iy)
   SMTGP_Grid_Cell c_parent = smtgp(i-1)(ix >> 1,iy >> 1)
   // Update the screen-space, axis aligned bounding box
   // around the parent SM-tile
   c_parent.abb_screen.ExpandToIncludeABB(c_curr.abb_screen)
   // Update the maximum needed SM-resolution
   c_parent.sm_res_x = MAX(c_parent.sm_res_x,c_curr.sm_res_x)
   c_parent.sm_res_y = MAX(c_parent.sm_res_y,c_curr.sm_res_y)
   i_pyramid = i_pyramid >> 1
```

4.6 Shadow Map Tile Grid Pyramid Traversal

Finally we traverse the grid pyramid top down, building an implicit kd-tree as we recursively traverse it, as follows: If the resolution requirement of the SM-tile-grid cell along at least one axis cannot be satisfied with a SM-tile-texture with dimensions supported by the GPU (e.g. on current GPUs typically: required SM-dimension $>$ 4096), we split it symmetrically along one or both SM-axis into 2 or 4 subcells. We split into 2 subcells if only one axis has SM resolution requirements which cannot be fulfilled, otherwise we split into 4 subcells. Otherwise we use Deferred Shadowing (see section 3.2), and immediately create the SM-tile with the required resolution along each axis and shadow the "Shadow Result Texture" (see next paragraph) with it, using the Eye-Space Depth Buffer to get the depth values of the scene, as described in section 3.2.

The "**Shadow Result Texture**" is a $1 \times$ *byte* texture with the same dimensions as the frame buffer, into which we write only the results of the shadowing. This makes the application of the SM-tiles faster, since we write to a surface with only one byte entry per pixel; it also allows us to avoid any potential problems with slightly overlapping SM-tiles, since shadowing results of a tile which is applied later can simply overwrite previous results. (The Shadow Result Texture can also be used to apply postprocessing effects to the shadow, such as screen space blurring depending on distance to the shadow caster.)

Pseudocode for the traversal of the Shadow Map Tile Grid Pyramid:

```
// SMT ... SM-tile instance
// P ... SMTGP pos index + pyramid index
// smtq ... queue holding SMT
smtq.push(SMT(P(0,0),P(0,0)))
while(!smtq.empty())
 SMT smt = smtq.pop()
 int ip_x = smt.ip_x, int ip_y = smt.ip_y
 int sx = max(0,ip_y-ip_x), int sy = max(0,ip_x-ip_y)
 Rect rect(ix << sx,iy << sy, ((ix+1) << sx)-1,((iy+1) << sy)-1)
 // ex and ey are 0 for no further refinement, 1 otherwise
 int ex =
  Refine(smtgp(MAX(ip_x,ip_y)).MaxSmResInRect(rect).sm_res_x,
   ip_x,framebuffer.nx)
 int ey =
  Refine(smtgp(MAX(ip_x,ip_y)).MaxSmResInRect(rect).sm_res_y,
  ip_y,framebuffer.ny)
 if(ex > 0 || ey > 0) // refine this SM-tile further
  int ip_x_sub = smt.ix + ex, int ip_y_sub = smt.iy + ey;
  int ix_sub = smt.ix << ex, int iy_sub = smt.iy << ey;
  for diy=0 to ey
   for dix=0 to ex
    smtq.push(SMT(P(ix_sub+dix,ip_x_sub),P(iy_sub+diy,ip_y_sub)))
 else  // do not refine this SM-tile further
  ShadowShadowResultTextureWithSmTile(smt)
```

with

```
// sm ... SM-texture
Refine(sm_res_needed,i_refinement,framebuffer_nx_or_ny) {
 return sm_res_needed > i_refinement -
```

```
  log2(sm.n) - round(log2(framebuffer_nx_or_ny/smtmm.n)+0.5)
}
```

4.7 Apply Shadow to Scene

In a final step we modulate the scene RGB from the Eye-Space Depth Buffer with the Shadow Result Texture, and write the resulting shadowed scene into the frame buffer.

The whole algorithm not only leads to a greatly reduced number of SM-tiles that need to be created compared to the brute force approach, but also uses smaller and rectangular SM-tile textures for farther away SM-tiles. It therefore makes much higher quality shadow maps possible in realtime. Please see the Results section for a quantitative comparison.

4.8 Comparison with Adaptive Shadow Maps

One previously published shadow mapping technique that looks similar to our approach, is "Adaptive Shadow Maps" [7], and its GPU-based implementation, "Dynamic Adaptive Shadow Maps on Graphics Hardware" [11]. In contrast to Adaptive Shadow Maps our algorithm is built around performing the complete refinement procedure for each frame, making it well suited for dynamic scenes. Adaptive Shadow Maps on the other hand, have to cache the recently used shadow tiles for best performance - an approach which is not suited to dynamic scenes, because, whether the light direction changes or objects in the scene move, this invariably invalidates the cached shadow maps (see the large performance drop for this case in [11], even though the test scene consists only of a single tree on a small quad). This also means that we do not need video memory for cached tiles. Our algorithm has the added advantage that the shadow map can be "focused" [3] on the relevant part of the scene each frame.

4.9 Quality vs Performance Parameter

Fitted Virtual Shadow Maps allow for the introduction of a very intuitive quality vs performance parameter ξ: Subtracting an integer number ξ from the logarithmic resolution requirement value coming from the SMTMM, when deciding whether to further refine a SM-tile, allows us to intuitively influence the quality of the resulting shadow in the scene; the larger ξ, the less tiles will be created and the better the performance will be. This allows the algorithm to be tuned to a wide range of hardware. Note that the influence of the parameter is smooth, in the sense that it influences the shadow quality of the whole scene in the same way. If ξ is chosen to be large enough, so that only one SM-tile is created, then the shadow quality of Fitted Virtual Shadow Mapping is the same as normal shadow mapping.

4.10 Shadow Map Tile Texture Size Optimization

The basic FVSM algorithm refines the shadow map, until the required resolution of each SM-tile along each SM-axis is small enough that it can be satisfied with the maximum SM texture size which the GPU can handle. In practice, the resolution needs along both SM-tile axis do seldom require a quadratic SM-texture with maximum resolution. This can be understood from the fact, that there will always be perspective shortening in the scene, i.e. the pixels farther away from the eye-point will always require less SM-resolution (Note that due to the splitting of the SM into tiles this is always the case for farther away tiles, independent of the view direction relative to the light direction, contrary to SM reparametrization techniques, which can only profit from the perspective shortening for light directions which are not parallel or antiparallel to the view direction); Projection aliasing can of course counteract this, but even then, the projection aliasing does not normally influence both SM-axis at the same time. This leads to the optimization, that, instead of using a quadratic maximum sized texture for all SM-tiles, we create a rectangular SM according to the resolution requirements along each axis. There are 3 different ways to do this:

1. Render into a sub-rectangle of the same quadratic, maximum sized texture.
2. Render into a sub-rectangle of a series of quadratic power-of-two shadow map textures, where each texture in the series has halved dimensions relative to its predecessor.
3. Render into shadow map textures with the exact needed dimensions (= resolution) along each SM-axis.

The memory requirements of the 4 different approaches are as follows (with t_{max} being the maximum texture dimension, and w and h being the needed minimum width and height SM texture dimensions to satisfy the SM-tile resolution requirements):

SM dimension	mem usage	for $t_{max} = 4096$
${t_{max}}^2$	1	64 MB
$w \times h$ sub-rect of ${t_{max}}^2$	1	64 MB
$w \times h$ sub-rect of $max(w,h)^2$	$4/3$	85 MB
$w \times h$	$2 \cdot 4/3$	171 MB

This can be seen, by observing that the for optimization 2, the required SM textures have dimensions: $\{{t_{max}}^2, (\frac{1}{2}t_{max})^2, (\frac{1}{4}t_{max})^2, \ldots\}$ leading to the series (with ${t_{max}}^2$ pulled out) $\sum 1 + \frac{1}{4} + \frac{1}{16} + \ldots = \sum_{n=0} (\frac{1}{4})^n \leq \frac{1}{1-\frac{1}{4}} = \frac{4}{3}$.

For optimization 3, we arrive at the series (${t_{max}}^2$ pulled out again) $\sum (1 \cdot 1 + \frac{1}{2} \cdot 1 + 1 \cdot \frac{1}{2}) + (\frac{1}{2} \cdot \frac{1}{2} + (\frac{1}{2} \cdot \frac{1}{2}) \cdot \frac{1}{2} + \frac{1}{2} \cdot (\frac{1}{2} \cdot \frac{1}{2})) + \ldots = 2 \cdot \sum 1 + \frac{1}{4} + \frac{1}{16} + \ldots = 2 \cdot \frac{4}{3}$.

The effective memory consumption given for $t_{max} = 4096$ is for $1 \times float$ SM textures.

Evidently what we want is a combination of small GPU memory consumption together with good performance; please see the Results section below for a performance comparison of the different approaches and a resulting recommendation which optimization variation to use in practice.

4.11 Handling Semitransparent Objects

When using deferred shadowing semitransparent objects must be treated separately, due to the fact that only the depth entry of the foremost opaque pixel is stored in the Eye-Space Depth Buffer.

A trivial solution is evidently to render the semitransparent objects after having shadowed the scene without shadowing them; this will give them a (slight) glow effect, which might or might not be acceptable.

A second approach would be to again render the semitransparent objects after having shadowed the scene, shadowing them with with conventional shadow mapping (i.e. using a single shadow map). In practice, semitransparent objects are usually implemented to only

receive shadows, but not cast them, due to the increased complexity of the shadowing problem when semitransparent objects are involved. The fact that in this case no self shadowing can occur makes this approach practical. Also note that the shadow (and therefore any artifacts) will be less visible, the more transparent the object is.

Less practical but more accurate solutions would work along the line to render the semitransparent objects to a separate buffer, storing the color and linear depth of the foremost semitransparent object. One could then use this buffer later on to also shadow the foremost semitransparent object correctly. Note that in this case one also needs double the entries int the SMTMM.

Finally, a completely different approach would be to not use Deferred Shadowing, but Multi Pass Shadowing (see section 3.1) to apply the SM-tiles to the scene, sped up by restricting the part of the scene that needs to be rerendered to the screen space extent (see 4.5 above) of each SM-tile. In this case the semitransparent objects can be rendered and shadowed (with the SM-tile) as if only a single shadow map was being used.

5 Results

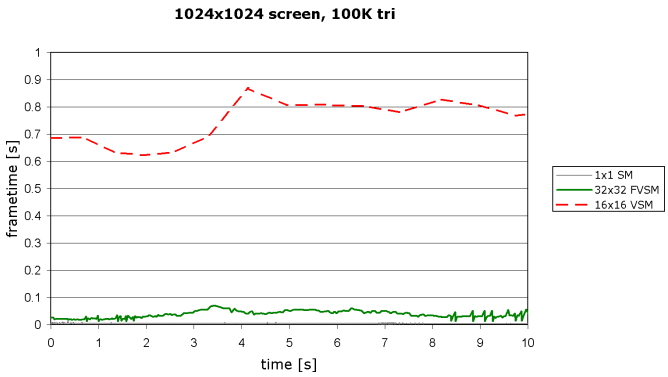


Figure 4: Performance comparison between normal, Virtual Tiled and Fitted Virtual SMing along path in forest test scene.

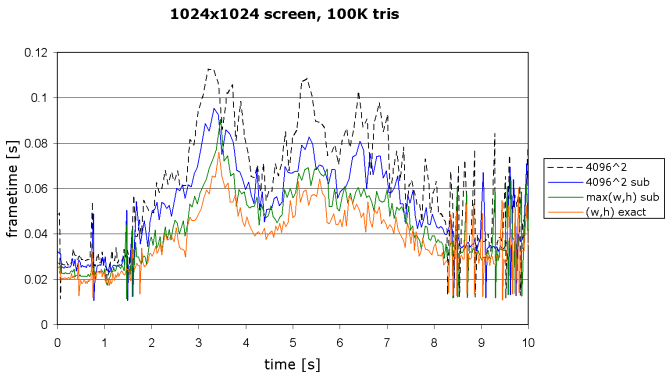


Figure 5: FVSM performance comparison between different SM-tile creation modes.

Unless otherwise noted, all results were created on a NVidia GeForce 8800GTS with 640 MB of RAM and a Pentium4 3.4 GHz (2 GB RAM).

The aim of our work is to research the applicability and improve the quality of dynamic shadow map shadows when applied to large

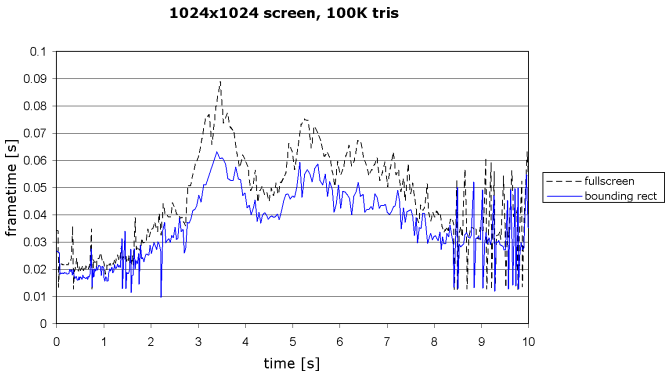


Figure 6: FVSM SM-tile application to shadow result texture: Comparison between using a fullscreen rectangle and using the SM-tile screen space bounding rectangle.

to medium sized scenes. In practice, the resolution of shadow maps currently supported in hardware suffices to create shadows for light sources with a small area of influence (spotlight located near the ground with a small spread angle, point light lighting a small room, etc), especially when combined with shadow map reparametrization. In addition, evidently the render costs of an algorithm such as FVSMs, although being much faster than brute force Virtual Tiled SMing, are currently too high to apply it to several light sources (see figure 4 for a frametime ccomparsion).

Our test scene of broad-leaf trees on a hilly terrain was chosen accordingly: It is a medium sized outdoor scene, which makes it of practical interest, since dynamically shadowing outdoor scenes in high quality continues to be a challenge. It will in many cases be lit by the sun, i.e. a single light source which can be modeled by a directional light or far away spotlight, and which can therefore be shadowed using a single shadow map (preferably of a very high resolution). It also lends itself to be efficiently modeled and rendered using alpha textured geometry, and can therefore not be shadowed e.g. by shadow volumes; in our case the fence and the leaves on the trees are alpha textured quads (this could be extended to e.g. using billboards or billboard clouds for farther away trees). Additionally, animating the direction of the sunlight turns it into a fully dynamic scene from the point of view of shadowing it (Note that even if the scene itself is completely static, any change in the light position, direction etc would result in invalidating any cached shadow map information, making it unsuited for algorithms that are built on caching shadow map information from previous frames to work efficiently).

Figure 1 shows a screenshot from our test scene; it compares the quality of normal shadow mapping (4096^2 shadow map) with Fitted Virtual Shadow Maps (lightspace perspective shadow maps (LiSPSM) active in both cases).

Figure 4 shows frame time curves from a path through a forest scene with 10^5 triangles rendered into a 1024×1024 frame buffer (you can download the videos of the flythrough along the path at http://www.cg.tuwien.ac.at/research/vr/fvsm). The uppermost curve depicts brute force Virtual Tiled SMing, using 16×16 4096^2 shadow maps; FVSM show Fitted Virtual SMs with adaptive SM-tile shadow map textures sizes, 32×32 maximum refinement. The 1×1 SM-curve finally gives the frame times for conventional SMing using a 4096^2 shadow map texture (leading to greatly reduced shadow quality). LiSPSM [19] was active for all renderings. Figure 2 shows screenshots at the end of the path through the forest scene (the full path can be viewed in the "Winter Forest Frametime Curve Path" folder in the accompanying media).

Figure 5 shows a performance comparison (along the same path in the scene) between different FVSM SM-tile creation modes (see section 4.10). The important thing one can see is, that this optimization smoothes the frametime peaks, i.e. it improves the worst case performance of the algorithm. One can also see that each of the SM-tile creation modes is faster than its predecessor. Since the memory consumption of the 3^{rd} scheme relative to the first two is $4/3$ (i.e. it consumes only one third more memory), in practice rendering to a sub-rectangle of a series of quadratic power-of-two shadow map textures is the best compromise between performance and memory consumption in most cases.

Figure 6 compares the performance of applying the SM-tiles to the shadow result texture using a fullscreen quad or the screen space bounding rectangle around each respective SM-tile (see section 4.5). Note that again the frametime peak at around $3.5s$ is smoothed, improving the worst case performance of the algorithm by around 25%.

For a 1024^2 frame buffer we found that a 256^2 SMTMM gave good results, while at the same time keeping the SMTMM creation- and processing overhead low.

For the detailed SM-tile-required-resolution-statistics parameters η_0 and η_1, we have found that $\eta_0 = 17$ and $\eta_1 = 22$ work well in practice.

Figure 7 shows the influence of the quality vs performance parameter ξ for $\xi = 0,\ldots,5$, where $\xi = 5$ gives equivalent shadow quality to normal shadow mapping. One can see that the shadow quality decreases homogenous in the whole scene.

We would also like to address the implementation problem of visible SM-tile boundaries in the resulting shadow due to precision issues (note that there is no other source for visible boundaries, since the SM-tiles are generated by the algorithm as to provide subpixel accuracy): This is easy to fix, by letting the SM-tiles overlap slightly; note that due to the fact that the shadowing results are accumulated in the shadow result texture first, before being combined with the scene color, overwriting previous shadow values from other SM-tiles can be done without any problem.

The absence or presence of undersampling artifacts when using shadow mapping with focusing can be best seen in motion, so please take a look at the accompanying videos and screenshots found at http://www.cg.tuwien.ac.at/research/vr/fvsm.

6 Conclusion

We have introduced Fitted Virtual Shadow Maps, a new smart shadow map algorithm which allows for the efficient shadowing of large scenes without undersampling artifacts, while at the same time making use of previous shadow map improvements, such as shadow map focusing and shadow map reparametrization techniques. Virtual Shadow Mapping allows the algorithm to bypass the memory cost and texture size limits of current GPUs. Instead of brute force Virtual Tiled Shadow Mapping, Fitted Virtual Shadow Maps employ a combination of GPU and CPU processing to create a map, which contains information about what resolution would be required where in a shadow map. This leads to an performance increase of at least an order of magnitude over the brute force approach, while still greatly reducing or even removing perspective *and* projection aliasing. The algorithm can be tuned with several parameters according to the quality requirements of the scene; most importantly the shadow quality can be uniformly reduced down to normal shadow mapping by use of a very intuitive quality-vs-speed parameter.

References

[1] Timo Aila and Samuli Laine. Alias-free shadow maps. In *Proc. Eurographics Symposium on Rendering 2004*, pages 161–166, 2004.

[2] Jukka Arvo. Tiled shadow maps. In *Proc. Computer Graphics International 2004*, pages 240–247, 2004.

[3] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. Practical shadow mapping. *Journal of Graphics Tools*, 7(4):9–18, 2002.

[4] H. Chong and S. J. Gortler. A lixel for every pixel. In *Proceedings of Eurographics Symposium on Rendering 2004*, 2004.

[5] Franklin C. Crow. Shadow algorithms for computer graphics. *Computer Graphics (Proc. ACM SIGGRAPH 77)*, 11(2):242–248, 1977.

[6] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165, New York, NY, USA, 2006. ACM Press.

[7] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive shadow maps. In *Proc. ACM SIGGRAPH 2001*, pages 387–390, 2001.

[8] Markus Giegl and Michael Wimmer. Queried virtual shadow maps. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. ACM Press, 2007.

[9] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. In *Eurographics State-of-the-Art Reports*, 2003.

[10] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph.*, 24(4):1462–1482, 2005.

[11] Aaron Lefohn, Shubhabrata Sengupta, Joe M. Kniss, Robert Strzodka, and John D. Owens. Dynamic adaptive shadow maps on graphics hardware. In *ACM SIGGRAPH 2005 Sketches*, 2005.

[12] Brandon Lloyd, David Tuft, Sung-eui Yoon, and Dinesh Manocha. Warping and partitioning for low error shadow maps. In *Proceedings of the Eurographics Symposium on Rendering 2006*, pages 215–226. Eurographics Association, 2006.

[13] T. Martin and T.-S. Tan. Anti-aliasing and continuity with trapezoidal shadow maps. In *Proc. Eurographics Symposium on Rendering 2004*, pages 153–160, 2004.

[14] Tomas Möller and Eric Haines. *Real-Time Rendering, Second Edition*. A. K. Peters Limited, 2002. ISBN 1568811829.

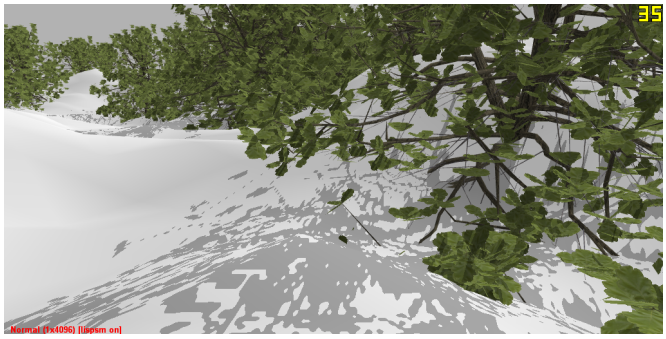
[15] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics (Proc. ACM SIGGRAPH 87)*, 21(4):283–291, 1987.

[16] Marc Stamminger and George Drettakis. Perspective shadow maps. *ACM Transactions on Graphics (Proc. ACM SIGGRAPH 2002)*, 21(3):557–562, 2002.

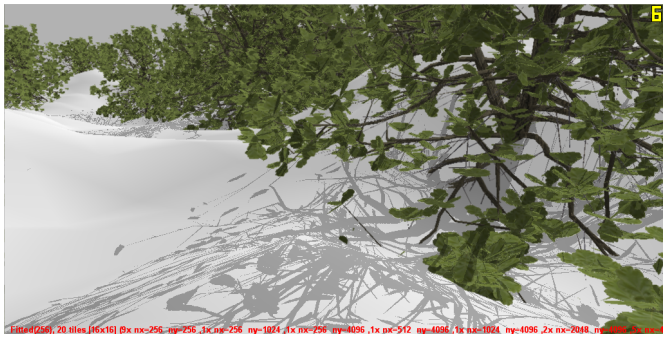
[17] Yulan Wang and Steven Molnar. Second-depth shadow mapping. Technical Report TR94-019, University of North Carolina at Chapel Hill, 1994.

[18] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics (Proc. ACM SIGGRAPH 78)*, 12(3):270–274, 1978.

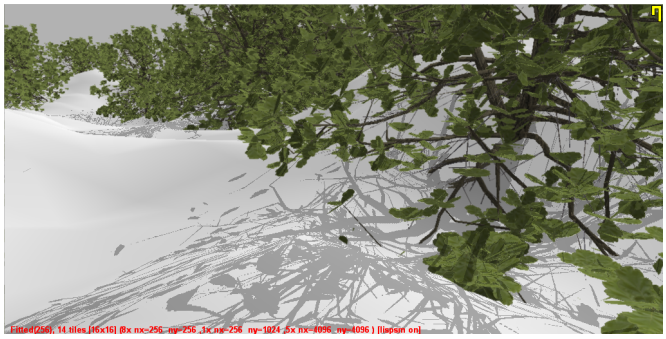
[19] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In *Proc. Eurographics Symposium on Rendering 2004*, pages 143–151, 2004.



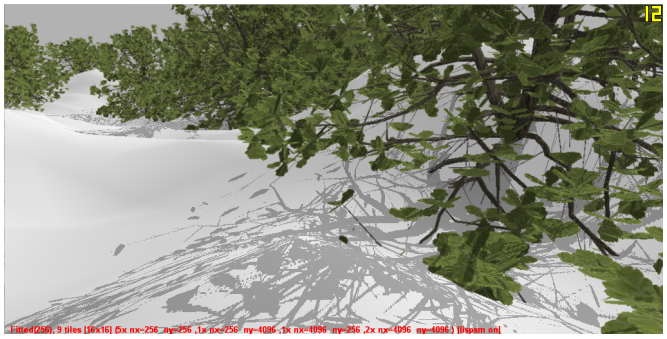
(a) 4096^2 normal SM



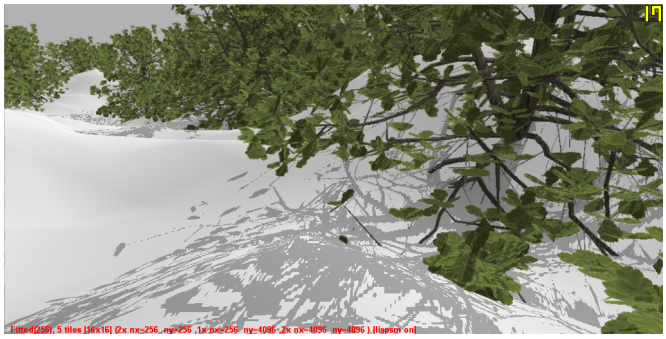
(b) FVSM, $\xi = 0$



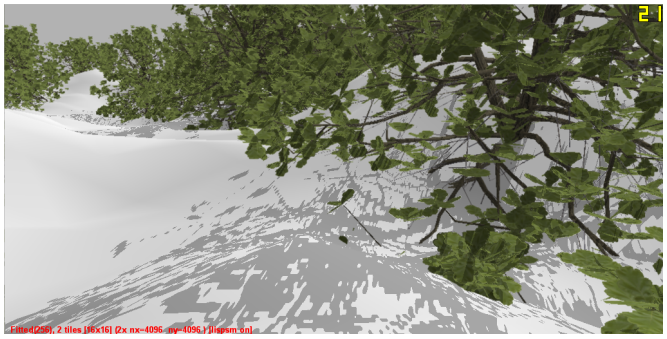
(c) FVSM, $\xi = 1$



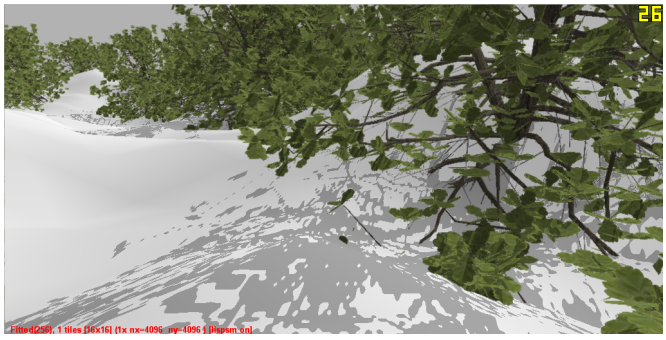
(d) FVSM, $\xi = 2$



(e) FVSM, $\xi = 3$



(f) FVSM, $\xi = 4$



(g) FVSM, $\xi = 5$

Figure 7: Influence of quality vs performance parameter ξ; $\xi = 5$ gives 1 SM-tile and is therefore equivalent to normal SMing. (LiSPSM SM reparametrization was active in all cases). Performance in frames per second can be observed in the upper right corner, respectively.

A Appendix: SMTMM Creation Pixel Shader

```
Ps_OUT PsCreateShadowMapTileMapping(Ps_IN IN)
{
    Ps_OUT OUT;

    // texture coordinate of center of current pixel
    float2 tc_pixel_center = IN.v2_tc.xy;

    // texture coordinates of right,left,upper and lower neighbor of current pixel, clamped to rendertarget extent
    float2 tc_pixel_neighbor_right =
        float2(clamp( tc_pixel_center.x + rendertarget_nr_pixel_inv.x, 0, 1), tc_pixel_center.y );
    float2 tc_pixel_neighbor_left =
        float2(clamp( tc_pixel_center.x - rendertarget_nr_pixel_inv.x, 0, 1), tc_pixel_center.y );
    float2 tc_pixel_neighbor_upper =
        float2(tc_pixel_center.x, clamp( tc_pixel_center.y + rendertarget_nr_pixel_inv.y, 0, 1));
    float2 tc_pixel_neighbor_lower =
        float2(tc_pixel_center.x, clamp( tc_pixel_center.y - rendertarget_nr_pixel_inv.y, 0, 1));

    // read viewspace z for current pixel from Eye-Space Depth Buffer
    float z_view_center = tex2D(tex_shadow_depth_buffer, tc_pixel_center).w;

    float2 v2_dz_view_use = float2(
            sm_tile_mapping_pick_smaller_dz(z_view_center,tc_pixel_neighbor_left,tc_pixel_neighbor_right),
            sm_tile_mapping_pick_smaller_dz(z_view_center,tc_pixel_neighbor_lower,tc_pixel_neighbor_upper)
        );

    // pos of left neighbor of current pixel in the shadowmap
    // ScreenspaceToShadowmapCoordinatesAndLightspaceDepth uses the matrix from section "Deferred Shadowing" to transform to eye-space.
    float3 pos_sm_left = ScreenspaceToShadowmapCoordinatesAndLightspaceDepth(
            tc_pixel_neighbor_left.x, tc_pixel_center.y, z_view_center - v2_dz_view_use.x
        );

    // pos of right neighbor of current pixel in the shadowmap
    float3 pos_sm_right = ScreenspaceToShadowmapCoordinatesAndLightspaceDepth(
            tc_pixel_neighbor_right.x, tc_pixel_center.y, z_view_center + v2_dz_view_use.x
        );

    // pos of lower neighbor of current pixel in the shadowmap
    float3 pos_sm_lower = ScreenspaceToShadowmapCoordinatesAndLightspaceDepth(
            tc_pixel_center.x, tc_pixel_neighbor_lower.y, z_view_center - v2_dz_view_use.y
        );

    // pos of upper neighbor of current pixel in the shadowmap
    float3 pos_sm_upper = ScreenspaceToShadowmapCoordinatesAndLightspaceDepth(
            tc_pixel_center.x, tc_pixel_neighbor_upper.y, z_view_center + v2_dz_view_use.y
        );

    float2 pos_sm_max = Max(pos_sm_left.xy,pos_sm_right.xy,pos_sm_lower.xy,pos_sm_upper.xy);
    float2 pos_sm_min = Min(pos_sm_left.xy,pos_sm_right.xy,pos_sm_lower.xy,pos_sm_upper.xy);

    // Approximate extent of the current pixel projected onto the shadowmap
    float2 dxy_pixel_on_shadowmap = 0.5 * (pos_sm_max - pos_sm_min);

    // Measure of resolution needed to shadow this pixel with subpixel accuracy
    float2 pixel_shadowmap_resolution_measure;
    // use pixel_shadowmap_resolution_measure = -log2(round(dxy_pixel_on_shadowmap + float2(0.5,0.5))/256
    frexp( sqrt(2.0) * dxy_pixel_on_shadowmap, pixel_shadowmap_resolution_measure);
    //pixel_shadowmap_resolution_measure *= (1.0/256.0);
    // [0,255] => [0,1] (for output to 8-bit surface)
    pixel_shadowmap_resolution_measure = ldexp(-pixel_shadowmap_resolution_measure, -8);

    // Postion of pixel center in the shadowmap
    float3 pos_shadowmap =
        ScreenspaceToShadowmapCoordinatesAndLightspaceDepth(
            tc_pixel_center.x, tc_pixel_center.y, z_view_center
        );

    // Output SM-tile position and resolution measure along SM x- and y-direction.
    OUT.color = float4(pos_shadowmap.x, pos_shadowmap.y, pixel_shadowmap_resolution_measure.x, pixel_shadowmap_resolution_measure.y);

    return OUT;
}
```