N-Buffers for efficient depth map query

Xavier Décoret

EUROGRAPHICS 2005 / M. Alexa and J. Marks
(Guest Editors)

Volume 24 (*2005*), *Number 3*

N-Buffers for efficient depth map query

Xavier Décoret

ARTIS GRAVIR/IMAG INRIA†

Abstract

We introduce the N-buffer as a tool for multiresolution depth map representation. This neighborhood buffer *encodes the value and position of local depth extrema at different scales in an image cube, in contrast to the image pyramid. The resulting increase in storage space is largely compensated by the following benefits: objects of any size can be culled in constant time against an occlusion map using four depth lookups; visibility-like queries can be performed in vertex and fragment programs; N-buffers can be computed very efficiently with graphics hardware. We present three applications of this datastructure, and in particular a novel approach for shadow volume acceleration.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

In the context of real-time rendering, it is crucial to avoid rendering objects that do not contribute to the final image. *Occlusion culling* refers to techniques that can quickly identify those objects that are not visible from the current viewpoint. Among the many methods for computing visibility [COCSD02, PT02] *occlusion maps* are simple yet effective. A set of potential occluders such as portals [LG95], precomputed virtual occluders [KCCO00] or simply a subset of the scene's objects are scan-converted to produce a depth value at every pixel. Objects are then scan converted and visibility is tested on pixels using a depth comparison. To avoid doing all tests at pixel level, Zhang et al. maintain multiscale versions of the Z-buffer for hierarchical scan conversion and depth comparisons [ZMHH97]. Similarly, Hey et al. render in front-to-back order and use a low resolution grid that maintains visibility for large areas and is updated in a lazy manner [HTP01]. An interesting approach was also proposed by Ho et al.[HW99] : summed area tables are used to quickly determine if the projection of an object is completely "covered" by the occluders. If yes, the depth test is performed only on those pixels of a row that has local maximum depth.

Occlusion maps are particularly easy to use because every current graphic card implements an efficient z-buffer. Efficiency comes from the rasterization power of specialized GPUs. Various optimizations such as Fast Z-clear, Z compression [Mor00] or recently proposed ones [AMN03] can further improve performance. Originally designed for hidden surface removal, Z-buffer has been recently exposed to programmers through occlusion queries [OQ03] which makes hardware-accelerated culling very efficient [HSLM02, BWPP04]. Because they are so efficiently implemented, Z-buffers have also found other applications [TPK01] such as the generation of shadow maps [Wil78], culling and clamping of shadow volumes [LWGM04] or determination of level of details [ASVNB00]. Depth maps are present in other areas of Computer Graphics. They are used to model height fields for terrain rendering, or model details with displacement mapping and relief texturing [OBM00, PNC05].

In this paper, we present a novel hierarchical representation of depth maps called N-buffers that allows querying of the depth within a region in a very simple manner. It is particularly well suited for an implementation of visibility-like queries within GPU programs. After presenting the principle in section 2, we will present three applications in section 3

† ARTIS is a research team in the GRAVIR/IMAG laboratory, a joint unit of CNRS, INRIA, INPG and UJF

that will be used to discuss the advantages and limitations of N-buffers in section 4.

2. Description of N-buffers

An N-buffer is a sequence of depth maps similar to an image pyramid [Wil83] except that all levels have the same resolution. Level 0 is a standard depth map. A pixel at level i stores the *maximum* depth of the pixels in a *neighborhood* of *size* i in level 0. Different definitions of neighborhood can be used. For the moment, we define the neighborhood of size i of pixel (x,y) to be the $2^i \times 2^i$ grid of pixels whose lower left corner is located at (x,y). Figure 1 shows an example of the four first levels of an N-buffer. Because all levels have the same resolution, an N-buffer can be built from a depth map of any size; for this definition of neighborhood, the number of levels is the *log* of the largest side of the initial depth map.

Construction. It is quite straightforward that level $i+1$ of an N-buffer can be built from level i using the formula:

$$l_{i+1}[x,y] \leftarrow \max(\max(l_i[x,y], l_i[x,y+2^i]), \max(l_i[x+2^i,y], l_i[x+2^i,y+2^i]))$$

Although a very efficient software implementation can be derived (the formula shows in particular that the level can be built in two passes: a horizontal and a vertical one), today's graphic hardware can perform this computation much faster, by evaluating the above formula in a fragment program.

Here is the algorithm described in OpenGL. Level 0 is obtained by copying the depth map into a texture, using `glCopyTexSubImage`. An offscreen buffer is then used. Its projection and modelview matrices are set to identity, its viewport matches the main window's one, the color mask is set to false, depth test is set to always pass. We render a square (from -1 to 1) with unit texture coordinates (from 0 to 1). Texture repeat mode is set to `GL_CLAMP_TO_BORDER` with border color set to 0. The fragment program of Figure 2 is activated. Finally, we copy the contents of the offscreen depth buffer into a texture and proceed to next level. This algorithm is a simple generalization of GPU based matrix reduction as described in [BP04].

Query. The N-buffer is designed to retrieve in a simple manner the maximum depth within an axis-aligned rectangular region of a depth map. For such a region R defined by (x,y) and $(x+\Delta_x, y+\Delta_y)$, we retrieve the appropriate level by finding the smallest i such that $2^i \geq \max(\Delta_x, \Delta_y)$ By construction, we then know that every pixel in R is at a depth less or equal than the depth stored in $l_i[x,y]$.

Of course, approximating R with a single power-of-2 sided square is often wasteful. It is possible to achieve a tighter approximation by covering R with four smaller power-of-2 sided squares, as shown in Figure 3. Maximum depth within R is then retrieved with four lookups in the appropriate level, at the coordinates of the lower left corners

```
void main(float4    t : TEXCOORD0,
          out float     z : DEPTH,
      uniform sampler2D prevLevel,
      uniform float3    l)
{
  // prevLevel is previous i-1 and has size WxH
  // l.xyz is 2^(i-1)/W , l.y is 2^(i-1)/H , 0
  float za = tex2D(prevLevel,t.xy);
  float zb = tex2D(prevLevel,t.xy+l.xz);
  float zc = tex2D(prevLevel,t.xy+l.xy);
  float zd = tex2D(prevLevel,t.xy+l.zy);
  // a,b,c,d get 0 for lookups "outside"  texture because
  // repeat mode is set to CLAMP_TO_BORDER with border=0
  z = max(max(za,zb),max(zc,zd));;
}
```

Figure 2: *Cg program to compute level i from level* $i-1$

of each square. Since the N-buffer stores maxima, the overlapping of squares is not an issue. The code for choosing the placement of the four boxes is straightforward. It is just a five-way branch that encodes the 5 cases in Figure 3. Note

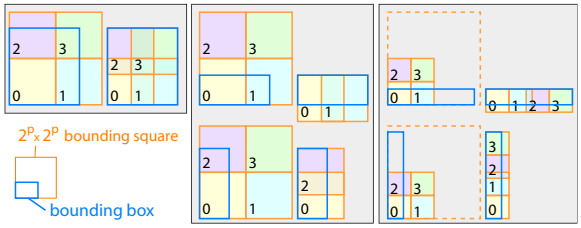


Figure 3: *The five ways to cover a 2D bounding box B with 4 power-of-two-sided squares. Each grey block shows the bounding box in thick blue line, the coverage by a single square and by 4 sub-squares. (left) B covers all quadrants of bounding power-of-two-sided square. (middle) B covers two quadrants. (right) B covers two quadrants on less that half their height (top) or half their width (bottom).*

that in the rightmost cases of Figure 3, the middle schemes could also have been applied, but they are not as tight. Note also that for the middle and right top schemes, we could have placed the sub-squares aligned with the top or with the bottom of the bounding box. We chose top because in many scenes, objects are lying on a surface (e.g. a terrain) so the visibility of an object does not change if the object is extended a little bit toward the bottom.

It is possible to quantify the improvement of this enhanced coverage. We measured the number of extra pixels covered using one square and using four squares for all possible rectangle sizes and positions. The results are shown on Figure 4. In particular, when $\lfloor \log(\Delta_x) \rfloor = \lfloor \log(\Delta_y) \rfloor$, the extra cover-

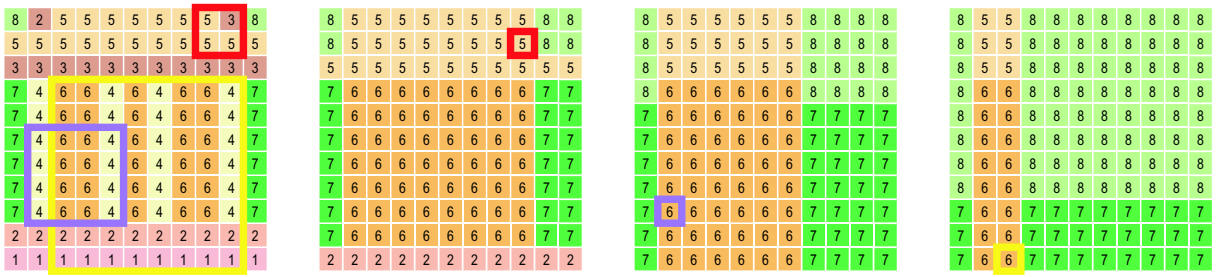


Figure 1: *(a) Level 0 of an N-buffer is a depth map. (b) A pixel (boxed in red) in level 1 stores the maximum depth of the* 2×2 *pixels north and east of it (shown on left image). (c) At level 2 it stores the maximum depth of* 4×4 *pixels (boxed in purple). (d) At level 3 it stores the maximum depth of* 8×8 *pixels (boxed in yellow).*

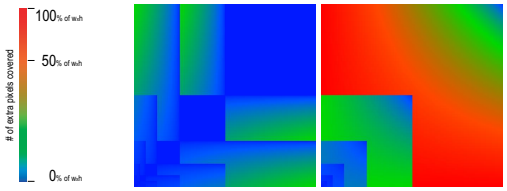


Figure 4: *A point* (i, j) *in image represents a box of dimensions* $i \times j$ *in an image of size* $w \times h$*. We cover this box with four (left) squares or one (right) square. We measure the number of pixels in squares that are not in the box and indicate it using false colors.*

age is null because the box can be covered exactly with four squares (leftmost case on Figure 3).

3. Applications

N-buffers are useful for many algorithms. In this section, we present three example applications of N-buffers.

3.1. Occlusion culling

A straightforward application is culling objects against an occlusion map. To render a view of a scene, we first select candidate occluders. We chose the heuristic that selects those objects that were visible at previous frame [BWPP04]. We render these occluders in the observer's view and build an N-buffer with the content of the depth buffer as described in section 2. We then test every object, including the selected occluders, against the N-buffer in the following way.

For every object E, we find an axis-aligned bounding box in image space of E by projecting its bounding volume. That is, we find pixels (x,y) and $(x+\Delta_x, y+\Delta_y)$ and depth z such that the rasterization of E would produce fragments (x',y',z') such that $x' \in [x, x+\Delta_x], y' \in [y, y+\Delta_y]$ and $z' \geq z$. Using the N-buffer as previously described, we query the maximum depth z_O within the box in the occlusion map. If $z_O < z$ we can safely conclude that E is not visible. Finally, we render the objects determined to be visible, unless they were already rendered as occluders.

The benefit of N-buffers here is that it drastically reduces the number of depth comparisons required by a classical occlusion-query based or a hierarchical occlusion maps approach. Indeed, whatever the size and location of the projection of a tested occludee, the visibility query requires exactly 4 depth comparisons. In counterpart, the computation of the N-buffer levels has a fixed cost that depends only on the viewport's resolution and is independent of the scene complexity, notably the number of tested occludees. These facts will be discussed in detail in section 4.

Finally, N-buffers performs more conservative culling because we test axis-aligned regions where occlusion queries exactly test the projection of occludees bounding boxes. Moreover, we compare the z min of the bounding boxes with the z max of a region so a group of occluders will hide only objects that are behind their *back ghost polygon*, as defined by Heo [HKW00], that is, the intersection of their joined shadow frusta and the closest plane parallel to the image plane that is entirely behind the occluders. For example, a sphere inside a slightly larger sphere would not be found as hidden. Up to that limit, N-buffers perform implicit occluder fusion in image space, so that a dense forest of trees with very small leaves would hide an object in the distance.

3.2. Particle culling

Modelling with particles is widely used for effects like fire, explosions, etc. To enhance visual appearance, particles are usually rendered as screen-space axis aligned quads with an appropriate texture. To release the CPU from the burden of computing these quads, the particles can be rendered with the `ARB_point_sprite`. The CPU sends only 3 coordinates per particle to the GPU which computes and rasterizes the quad. These coordinates can even be stored on the GPU (e.g using vertex buffer objects) and animated within a vertex shader.

Using point sprites however, increases the fill rate and one might want to perform culling on particles, especially for a large number of particles where most are occluded by other parts of the scene. Unfortunately, culling point sprites is tricky. If one just tests the visibility of the particle's point, then popping will be observed near the silhouettes of occluders, since part of the particle's sprite is visible before its center comes into view (and conversely). On the other hand, testing the visibility of the sprite requires computing its extents which ruins the benefits and simplicity of use of the point sprite extension.

The benefit of N-buffers for this application is that because the number of depth comparisons is fixed, it can be done very easily in a vertex shader that supports texture lookups (which is the case of latest generation of cards). In comparison, using hierarchical occlusion maps [ZMHH97, GKM93] is much more complex, because a point sprite may straddle boundaries of the image pyramid. The code would involve tests with a varying number of depth comparisons.

Moreover, since point sprites are always screen space aligned squares, we can cover them with exactly 4 power-of-2 sided squares whose positions can be pre-computed for given point sprites sizes. If this size is itself a power of 2, we can even use the direct covering approach which reduces to a single depth comparison. Finally, since point sprites are typically not wider than 32x32, only the first few levels of the N-buffer need to be computed.

3.3. Shadow volume clamping

Knowing the maximum depth within an area of an occlusion map can be used to perform shadow volume culling and clamping, as coined by Loyd [LWGM04]. Indeed, the shadow volume of a shadow caster does not need to extend to infinity; it can be clamped as long as it contains the shadowed regions. We propose the following algorithm. The scene is viewed by an observer, O, and lit by a spot light, L. The part of the scene visible from O is called the *visible surface*. We project the visible surface twice (details further in the section) from light view, once with depth test `GL_LEQUAL` and once with `GL_GEQUAL`. We read back the two depth buffers. They form what we call the *min/max litmaps*. A lexel in the min (resp. max) litmap thus records for the corresponding light ray the distance to the closest (resp. furthest) point *on the visible surface*. We then project each shadow caster in the light view, with depth buffer write disabled, and track the minimum (resp. maximum) of the minimum (resp. maximum) distances of the covered lexels in the appropriate litmap. These two values define the *extents* of the shadow volume. Figure 5 shows some configurations for the extents of a shadow caster.

Generation of litmaps. The litmaps can be obtained very easily by dualizing the standard shadow map approach. We render in light view the scene as lit by a virtual light placed at the observer's position. But instead of setting "shadowed" fragments' color to black, we discard them by setting their depth outside range $[0,1]$. The algorithm outline is the following. We read back the depth buffer from the camera's view in virtual shadow map `obsDepthMap`. We also record the combined projection and modelview matrices of the observer `obsMtx`. Then we render the scene in light's view. For each incoming fragment, we compute the distance to observer and compare it with the corresponding value in `obsDepthMap` using projective texture mapping with matrix `obsMtx`. If it is further, the fragment is discarded by setting its depth to 2.0. Figure 6 illustrates this process.

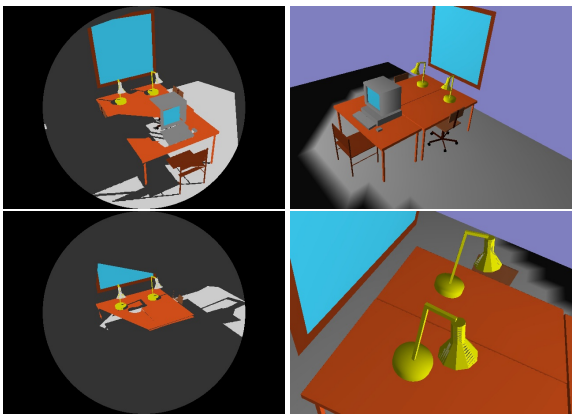
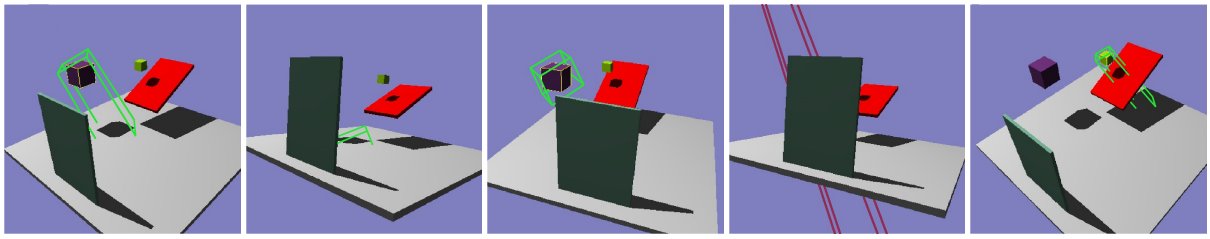


Figure 6: *Two examples of litmaps. (right) observer view of the scene (left) the visible surface reprojected in light view forms the litmap. For clarity, the color buffer is shown but only the depth buffer is actually relevant.*

Finding extents. This is where N-buffers comes into play. Once the litmaps are rendered, we construct the N-buffer levels as described in section 2. For the min part of the litmap, we change the program of Figure 2 so that it computes the minimum depth within neighborhoods. Then, for each shadow caster K, we project its vertices (or the vertices of a bounding volume if the shadow caster is too complex) in light's view. We compute a 2D axis aligned bounding box of these projections and retrieve the minimum and maximum depth z_{min} and z_{max} within that box as described in section 3.

Shadowing. If $z_{max} < z_{min}$ we know that K does not cast a shadow visible by the observer and we proceed to the next caster. Otherwise, we build the two planes perpendicular to the light direction and at distances corresponding to z_{min} and z_{max} . We finally run through the silhouette edges and build shadow volume by projecting each edge on the two planes. If the edge is further to the light than the min plane (case (a),(c) and (e) of Figure 5), we do not project it on the min plane. For the shadow volume to be correct, we must cap it by also projecting the shadow caster on the two planes.



(a) (b) (c) (d) (e)

Figure 5: *Shadow volume extents for a selected shadow caster (highlighted in yellow). (a) Both the object and its shadow are visible, the shadow volume encompasses both. (b,c) Either the object or its shadow are hidden, the volume encloses only what's visible. (d) Neither the object nor its shadow are visible, the shadow volume can be dropped, indicated by red lines. (e) Our approach cannot detect intermediate occluders and extends shadow volumes to the furthest visible surface points.*

4. Results and discussion

We implemented the three applications on a Pentium 2.4Ghz with an NVidia GeForce FX 6800 using OpenGL and Cg 1.3. This section presents statistics along with analysis and discussion.

4.1. N-buffer generation

The computation cost of the N-buffer depends only on the resolution of the the initial depth map (level 0). As opposed to image pyramids, this resolution does not need to be a power of 2. We wrote a test application that renders a rotating cube in the z-buffer, builds all levels of the N-buffer and records the computation time. As shown in Table 1, it

resolution	256^2	512^2	640×480	1024^2
nb levels	8	9	10	10
time (ms)	2	8	10	31

Table 1: *N-buffer computation time at various resolutions*

is compatible with real-time applications. It shows that the hardware already has the ability to perform the N-buffers construction efficiently. Our claim is that dedicated hardware could do even better. It would also adress some technical problems faced when using the GPU for something it is not exactly meant for. For example, our implementation uses a texture per level of the N-buffer which requires texture binding switches. To adress this, we also implemented a version with a single texture and levels packed horizontally. Unfortunately, it currently does not work for width greater than 256 pixels as it seems very wide textures are not supported. Indeed you can allocate a 2048×2048 texture but not a 8192×512 one although they amount to the same number of pixels.

4.2. Occlusion culling

We implemented occlusion culling with N-buffers in a walkthrough and compared it to an implementation of the same algorithm but using hardware occlusion queries to test the visibility of bounding boxes. We found that currently occlusion queries are generally faster. The first reason is that computing the projection of bounding boxes in image space and covering them with tiles adds some extra load on the CPU. The second reason is that querying the N-buffers is currently not optimized. Since the levels are stored in textures on the GPU, we cannot easily query them. Here is what we do. When processing the bounding boxes, we queue the 4 pixels that must be queried in list L_i where i is the level at which they must be queried. Then, when we construct level i of the N-buffer, we query every pixel $(x,y) \in L_i$ with `glReadPixels(x,y,1,1)`. Those pixel readbacks are quite inefficient, and even if they are not numerous (4 times the number of tested occludees), they incur a penalty that defavors N-buffers. In comparison, occlusion queries are very efficiently implemented by the hardware and can be issued concurrently to the rendering. Wimmer et al. [BWPP04] recently showed how to efficiently interleave occlusion queries with rendering, in a hierarchical manner to achieve high performance. In their approach, N-buffer would not be suited since the occlusion map evolves during rendering which would require costly level updates.

Therefore, without hardware support, N-buffer based occlusion culling can not compete with occlusion queries. However, if added to the hardware together with API support for querying the levels, they could bring significant performance increases. In order to evaluate the expected gains, we conducted the following experiment. During a walkthrough in a complex city model, we measured the number of depth tests performed when using occlusion queries (viewport of 512×512, back face culling enabled) and when using N-buffers. We made the measurement with and without hierarchical culling based on a hierarchy of bounding volumes. Results are reported in Table 2. The first observation is that N-buffer performs much fewer depth tests, about 3 orders

	OQ w/ hier.	OQ	NB w/ hier.	NB
# tests	1 100 000	200 000	350	500

Table 2: *Average number of depth comparisons with occlusion query (OQ) and with N-buffers (NB) with and without bounding volumes hierarchy.*

of magnitude. This ratio increases with the number of tested occludees. With Occlusion Query, the cost (number of depth comparisons) of tests is output sensitive: it depends on the screen projection of the tested objects. Conversely, with N-buffer, the cost is fixed and is equal to 4 times the number of tested object (we used frustum culling [AM00] so this number is not constant during the walkthrough). Of course, we must account for the number of fragments rasterized for the computations of the levels of the N-buffer. So we think of N-buffers as "factorizing" depth comparisons into a fixed-cost part independent of the number of tested objects, and a very-low-cost part repeated for each tested object. Consequently, benefit can be expected only if there is a large number of tested objects.

The second observation is surprising: with occlusion queries, hierarchical visibility culling incurs quite many more depth comparisons. The reason is that for visible nodes, several bounding boxes are rasterized, some of them (the higher ones in the hierarchy) covering a large region on screen. In consequence, it can be more efficient to directly test all leaf nodes. Of course it is highly scene/hierarchy/viewpoint dependent and is hard to predict/optimize. The key point is that the cost of testing a node can be higher than the cost of testing all its children. Conversely, N-buffers have the nice property that testing a node is always cheaper than testing its children because the cost of a test is constant. Therefore, unless everything is visible, we will observe gains when using a hierarchy.

In the light of these results, we think that N-buffers might be of interest if incorporated in graphic cards. It would require some extra memory to store the levels and a specialized component to perform their updates after a change in the z-buffer (note that when a pixel is changed, only the pixels on its left and below must be updated in the N-buffer; note also that low resolution N-buffers could also be used modulo some simple adaptations). Before rasterizing a primitive, the GPU could then do 4 depth comparisons in the appropriate level to check if the primitive is not completely hidden, thus saving rasterization and potentially many depth comparisons. This might prove beneficial when many objects with a large screen projection are actually hidden. Moreover, it would be better for parallelism since every query is done in a constant number of operations.

4.3. Particles

Particle culling is another example of the potential performance gain of hardware N-buffers could achieve. For N particles typically rendered as 16×16 sprites, the number of depth tests is $256N$ if rasterization is used. With N-buffers it is simply N; that is 256 times less. Moreover, as discussed earlier, N-buffer is the only way to do culling within the vertex program and without breaking the point-sprite extension. To show the feasibility, we implemented point-sprite culling. We first render the scene, build the N-buffer and render the particles, enabling the point sprite extension and using a vertex shader for culling sprites. However, our implementation suffers from current performance limitation of the Linux drivers for texture lookups in vertex programs so the frame rate is 3Hz for 60^2 particles with culling while we get 100Hz without culling.

4.4. Shadow volume clamping

We implemented shadow volume clamping and tested it on two scenes: a model of a city and a terrain with towers and birds flying above. Figure 7 shows some frames. We measured along recorded path the number of fragments rasterized with or without shadow clamping and the time required to render shadow volumes. In the city scene, the average fill rate without clamping is 20M pixels and decreases to 512 000 pixels with clamping (window size is 640×480) that is only 2.5%. The same ratio is observed for the terrain scene. For other simpler scenes, we observed an average ratio of 20%.

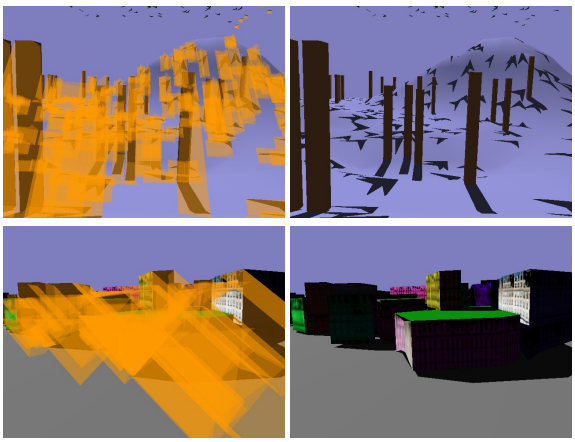


Figure 7: *Shadow volume clamping on towers and city scenes. (left) clamped shadow volumes. (right) resulting shadows.*

In the city scene, the benefits of culling and clamping are due to the fact that the shadows of many shadow casters are not visible so shadow casters are discarded. In the terrain scene, the benefit comes from the birds that cast shadows on

the ground but are not visible from the observer's position. In that case, the shadow volume is tight around the shadow on the ground instead of extending through the whole screen.

Our approach does shadow volume culling and clamping in a unified way. However, the clamping is less efficient than in [LWGM04] as we construct only a single slice. On the other hand, the slice is found in a direct and simple way, where their approach requires either CPU-based interval arithmetic or multiple occlusion queries. We also believe that the litmap construction is an interesting contribution for retrieving Potential Shadow Receivers. Still, we did not try to optimize our implementation to compete with CC shadow volumes in term of rendering speed. Rather, our purpose was to show that the N-buffer representation can potentially save fill rate, either directly (occlusion culling, particle culling) or indirectly (shadow volume clamping).

5. Conclusion and Future work

We presented a novel representation for depth maps. It allows query for the maximum (or minimum) depth within a rectangular region in constant time, no matter the rectangle's position and size. We showed that it can drastically reduce the number of depth tests compared to rasterization based approaches. The constant time property is particularly interesting since it makes visibility-like queries available in vertex or fragment programs. Indeed, querying depth extrema over a region could be done by rasterizing the region and using some to-be-proposed extension similar to the histogram one. However, such functionalities could not be called during the rendering of other primitives, that is within a vertex/fragment shader.

Our belief is that with the trend of porting many algorithms to the GPU (e.g. shadow quad generation [BS03], ray-tracing), being able to represent depth maps in an access-efficient manner will open new possibilities. In the future, we plan to investigate such applications as heightfield ray-tracing [PNC05].

On a more theoretical note, N-Buffer is a representation that allows both multi-scale and localized analysis of a discrete signal, in our case a depth map. For that reason, it is related to the theory of Wavelet Zoom [Mal01] and we plan to express it in this formalism. We also want to study other "neighborhood bases", and in particular to consider not only squares but rectangles, similar to the idea of RIP-maps mentionned in [KLK+00].

References

[AM00] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools: JGT*, 5(1):9–22, 2000. 6

[AMN03] Timo Aila, Ville Miettinen, and Petri Nordlund. Delay streams for graphics hardware. *ACM Transactions on Graphics*, 22(3):792–800, 2003. 1

[ASVNB00] Carlos Andújar, Carlos Saona-Vázquez, Isabel Navazo, and Pere Brunet. Integrating occlusion culling with levels of detail through hardly-visible sets. In *Comp. Graphics Forum (Proc. of Eurographics)*, volume 19(3), pages 499–506, 2000. 1

[BP04] Ian Buck and Tim Purcell. *GPU Gems*, chapter Ch. 37: A toolkit for Computations on GPUs, page 626. Addison-Wesley, 2004. 2

[BS03] Stefan Brabec and Hans-Peter Seidel. Shadow volumes on programmable graphics hardware. In *Computer Graphics Forum*, volume 3-22 of *EUROGRAPHICS Conference Proceedings*. Blackwell Publishers, 2003. 7

[BWPP04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum (Proceedings of Eurographics '04)*, (3), 2004. 1, 3, 5

[COCSD02] Daniel Cohen-Or, Y. Chrysanthou, Claudio Silva, and Fredo Durand. A survey of visibility for walkthrough applications. *IEEE Trans. on Visualization and Comp. Graphics*, 2002. 1

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238. ACM Press, 1993. 4

[HKW00] JunHyeok Heo, Jaeho Kim, and KwangYun Wohn. Conservative visibility preprocessing for walkthroughs of complex urban scenes. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 115–128. ACM Press, 2000. 3

[HSLM02] Karl Hillesland, Brian Salomon, Anselmo Lastra, and Dinesh Manocha. Fast and simple occlusion culling based on hardware depth querie. Tr02-039, UNC-CH, 2002. 1

[HTP01] Heinrich Hey, Robert F. Tobler, and Werner Purgathofer. Real-Time occlusion culling with a lazy occlusion grid. In *Rendering techniques (Proceedings of Eurographics Workshop on Rendering '01*, pages 217–222, 2001. 1

[HW99] Poon Chun Ho and Wenping Wang. Occlusion culling using minimum occluder set and opacity map. In *IV '99: Proceedings of the 1999*

International Conference on Information Visualisation, page 292. IEEE Computer Society, 1999. 1

[KCCO00] Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. Virtual occluders: An efficient intermediate PVS representation. In *Proc. of Eurographic Rendering Workshop*, 2000. 1

[KLK+00] Allison W. Klein, Wilmot Li, Michael M. Kazhdan, Wagner T. Corrêa, Adam Finkelstein, and Thomas A. Funkhouser. Non-photorealistic virtual environments. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 527–534. ACM Press/Addison-Wesley Publishing Co., 2000. 7

[LG95] David Luebke and Chris Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *Proc. of the 1995 Symposium on Interactive 3D Graphics*, pages 105–106, 1995. 1

[LWGM04] Brandon Lloyd, Jeremy Wendt, Naga Govindaraju, and Dinesh Manocha. Cc shadow volumes. In *Proceedings of the Eurographics Symposium on Rendering*. Eurographics, 2004. 1, 4, 7

[Mal01] Stéphane Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, 2001. 7

[Mor00] Steve Morein. Ati radeon—hyperz technology. In *SIGGRAPH/Eurographics Graphics Hardware Workshop 2000, Hot3D session*, 2000. 1

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 359–368. ACM Press/Addison-Wesley Publishing Co., 2000. 1

[OQ03] `ARB_occlusion_query` opengl extension specification. `http://oss.sgi.com`, 2003. 1

[PNC05] F. Policarpo, M. Oliveira Neto, and J. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of ACM Siggraph Interactive 3D Graphics and Games*, 2005. 1, 7

[PT02] Ioannis Pantazopoulos and Spyros Tzafestas. Occlusion culling algorithms: A comprehensive survey. *Journal of Intelligent and Robotic Systems*, 35(2):123–156, 2002. 1

[TPK01] Theoharis Theoharis, Georgios Papaioannou, and Evaggelia-Aggeliki Karabassi. The magic of the Z-buffer: A survey. In V. Skala, editor, *WSCG 2001 Conference Proceedings*, pages 379–386, 2001. 1

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274. ACM Press, 1978. 1

[Wil83] Lance Williams. Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11. ACM Press, 1983. 2

[ZMHH97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88. ACM Press/Addison-Wesley Publishing Co., 1997. 1, 4