ZP+: Correct *Z-pass* Stencil Shadows

Samuel Hornus[*] ARTIS / INRIA
Jared Hoberock[†] UIUC
Sylvain Lefebvre[*] GRAVIR / IMAG-INRIA
John Hart[†] UIUC

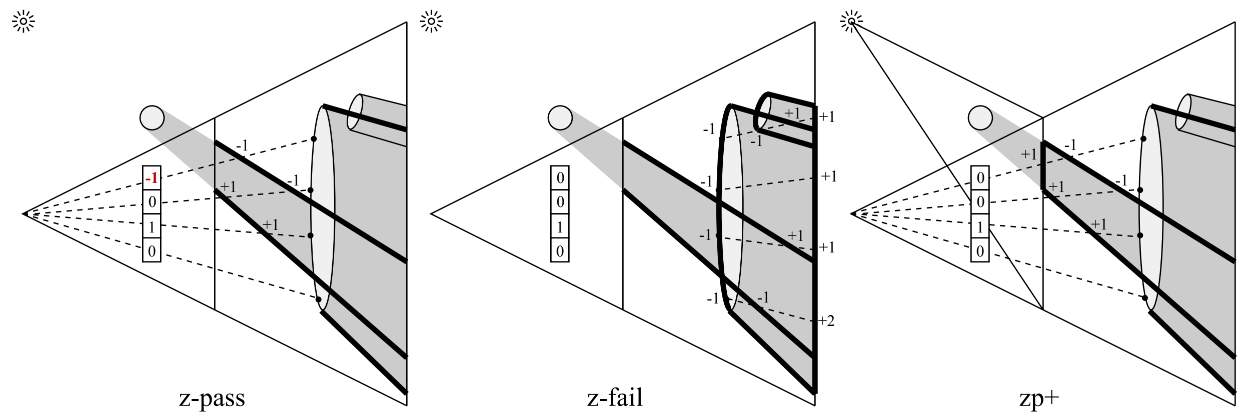

Figure 1: The *Z-pass* shadow volume method (left) pre-renders a scene into the z-buffer, and then renders shadow volumes (bold), incrementing (decrementing) the stencil buffer when front (back) facing fragments pass the z-test, but fails when shadow volume geometry is clipped by the near plane. The *Z-fail* method instead counts shadow volume fragments that fail the depth test, using far plane caps to robustly fix stencil sums, but requires all shadow volumes to be rendered including completely occluded shadow volumes. This paper's *ZP+* algorithm initializes the stencil buffer by first rendering from the light source to correctly set the stencil buffer to the proper values ordinarily clipped by *Z-pass*, and thus processes shadow volumes with the depth-culled speed of *Z-pass* but with the robustness of *Z-fail*.

Abstract

We present a novel algorithm for the rendering of hard shadows cast by a point light source. The well-known *Z-pass* method for rasterizing shadow volumes is not always correct. Our algorithm, which we call *ZP+*, elegantly corrects *Z-pass* defects. *ZP+* takes advantage of triangle strips and the fast culling capabilities of graphics hardware not available to conventional robust methods like *Z-fail*. While *Z-fail* can be up to 80% slower than *Z-pass*, our new method *ZP+* is typically less than 10% slower than *Z-pass*. Finally, we compare the three methods. When a scene is geometry-bound, *ZP+* is always faster than *Z-fail*. We also explain why, in some situations, *Z-pass* (hence *ZP+*) is surprisingly slower than *Z-fail* on more recent graphics hardware.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Real-Time Shadowing;

Keywords: stencil shadows, shadow volume, real-time rendering, graphics hardware

1 Introduction

Shadows are an important visual cue for the perception of depth in the 2-D depiction of 3-D scenes, and for some scenes, shadows often provide the only cue of the spatial relationship of disjoint objects. The quest to make videogames, virtual environments and other interactive graphics applications more photorealistic has included a recent surge in real-time shadow generation [Kilgard 2001; Everitt and Kilgard 2002; Aila and Akenine-Möller 2004; Chan and Durand 2004; Lloyd et al. 2004; Aldridge and Woods 2004; Hasenfratz et al. 2003].

The current state-of-the-art in real-time shadowing relies on the efficient processing of shadow volumes, and is used by the latest video game engines including the one used for iD Software's Doom 3. This paper describes a new algorithm called ZP+ that accelerates the shadow volume processing used in Doom 3 and other interactive 3-D games and applications.

Shadow volumes are object-space tesselations of the boundaries of the regions of space occluded from the light source [Crow 1977]. The alternative to shadow volumes is shadow mapping, which computes shadows by transforming a visible fragment's positions into the light's viewing coordinate system and compares it against a depth map precomputed in this coordinate system. Stenciled shadow volumes are popular and widely adopted because they provide crisp geometric shadow boundaries whereas shadow mapping suffers from the limited resolution of its image-based approach.

The *Z-pass* method is an early approach to implementing stenciled shadow volumes [Heidmann 1991] demonstrated here in Figure 1 (left). The *Z-pass* method first initializes the stencil buffer to zero and the z-buffer with the depth values of visible objects in the scene, and then rasterizes the sides of the shadow volumes. For each visible shadow volume fragment (rasterized pixel that passes the depth-test), the method increments the corresponding stencil buffer pixel if the fragment is front-facing and decrements if the fragment is back-facing. A version of the Jordan curve theorem leads to the conclusion that shadows occur at pixels with non-zero stencil buffer entries.

[*]FirstName.Name@inria.fr
[†]{hoberock,jch}@uiuc.edu

The *Z-pass* method fails when the shadow volume intersects the near clipping plane (the terminology we use is presented in Figure 2). In the example in Figure 1 (left), the topmost point is incorrectly classified as shadowed because the circle's lower shadow volume boundary was clipped by the near clipping plane. Methods that cap the clipped shadow volume with additional geometry [Diefenbach 1996; Batagelo and Junior 1999; Kilgard 2001] are computationally expensive and suffer from robustness problems that lead to glaring pixel-wide shadow cracks.

This near-plane clipping problem of the *Z-pass* method led others, including Carmack [2000], to devise the *Z-fail* method, which processes shadow volume fragments that fail (instead of pass) the depth test, as demonstrated in Figure 1 (middle). This approach moves the problem from the near clipping plane to the far clipping plane which can be handled robustly by extending it to homogeneous infinity [Everitt and Kilgard 2002]. This robustness comes at the cost of speed, as the *Z-fail* method must now, during shadow volume rasterization, also rasterize the front faces of the scene geometry at their original location, and the back faces of scene geometry projected onto the far plane. Furthermore, *Z-fail* cannot capitalize on early z-culling which *Z-pass* uses to avoid the processing of shadow volume fragments occluded from view by scene geometry.

This paper presents a new algorithm we call *ZP+* (for *Z-pass* plus) that provides the robust correctness of *Z-fail* at a speed more comparable to the *Z-pass* approach. The *ZP+* algorithm, demonstrated in Figure 1 (right), constructs in a separate pass a sheared frustum extending from the light source to the original view's near plane. Since the far clipping plane of this sheared light frustum aligns with the near plane of the view frustum, it contains only the scene geometry responsible for generating the shadow volume clipped by the original near plane. Rasterizing this scene geometry projects its fragments onto the original near clipping plane where it can be used to properly and robustly initialize the stencil buffer.

Section 2 compares the *ZP+* approach to its predecessors, establishing its novelty and utility. Section 3 describes the *ZP+* algorithm in more general configurations beyond the simple demonstration in Figure 1 (right), concluding with a proposal for a new OpenGL extension to aid its efficiency. Section 4 describes why *ZP+* is faster than *Z-fail*, whereas Section 5 analyzes the robustness of *ZP+*, focusing on the discrepancy between rasterization and clipping that can lead to cracks and recommendations on how to avoid them. We note one case of numerical inaccuracy that, while exceedingly rare, causes artifacts that may warrant a switch to *Z-fail*. Section 6 exhibits the quality and speed of its shadows with demonstrations and performance charts on a variety of recent graphics processors.

2 Previous Work

2.1 *Z-pass* Corrections

The introduction demonstrated that the original *Z-pass* algorithm [Heidmann 1991] leads to errors when the shadow volumes intersect the near rectangle (the near clipping plane of the view frustum). The problem is initializing the stencil buffer: some portions may be lit, and some may be shadowed. It is not as simple as knowing whether or not the viewer is shadowed; the eye's position may be lit even though the entire near plane is shadowed. Figure 3a illustrates such a configuration.

To compensate, some have proposed the generation of additional geometry at the near plane to *cap* the shadow volume. Batagelo et al. [1999] suggest computing the intersection of every polygon's shadow geometry with the near plane, creating additional geometry to cap the portions of the near plane in shadow. However, computing the intersection of shadow volumes with the near plane was expensive and prone to numerical errors that exhibit themselves as cracks in the shadow.

Diefenbach [1996] caps the near rectangle by rasterizing additional geometry enclosing the space between the light source and occluder's silhouette. Everitt and Kilgard [2002] note Diefenbach's approach is not robust in every configuration and provide three counter-examples where the cap would not be rendered due to combined clipping by both the near and far clip planes.

Kilgard's solution [Kilgard 2001] explicitly mirrors scene geometry close to the near plane. An additional pass renders a cap by projecting an occluder through the light source. This projection pushes occluding geometry up against a *near ledge,* a plane very close to the near plane, yet not so near as to clip this extra geometry. Since this projected geometry does not share vertices with shadow geometry, this approach too suffered from cracking problems.

None of these methods prove satisfying due to the complicated geometric computations required of the CPU. Moreover, critics note that any sort of method requiring computational geometry is inherently fragile, due to the imprecise nature of computer arithmetic. These problems tend to manifest themselves as unsightly cracking artifacts, which are unacceptable to the demanding expectations of real-time graphics enthusiasts. This suggests that any sort of feasible solution must rely on simple rasterization-based operations available to graphics hardware.

2.2 *Z-fail*

In light of the difficulties inherent in creating a truly robust *Z-pass* algorithm, Carmack [2000] discovered *Z-fail*. This algorithm, symmetric and equivalent to *Z-pass*, removes the problem at the near plane and pushes it to its far plane counterpart. The idea is that just as a pixel's shadow status may be determined by counting the number and orientation of shadow faces *in front* of it, one may compute the identical solution by inspecting the shadow faces *behind*. In this way, *Z-fail* sidesteps *Z-pass*'s near rectangle problem, since shadow geometry between the eye and a pixel are irrelevant.

Bilodeau and Songy [1999] developed earlier a similar idea now called *reversed Z-pass*. Instead of counting shadow volume fragments that fail the depth test, their method reversed the depth test and counted fragments that passed this reversed test. While they may appear similar, *Z-fail* outperforms reversed *Z-pass* on modern GPUs that terminate subsequent fragment processing operations once the z-test has failed.

Yet, we are left with the problem of shadow volume clipping resulting in incorrect stencil counts. As Figure 2 (right) illustrates, we can close a shadow volume by adding its *light cap* and *dark cap*. The light cap is simply composed of scene object polygons that face the light, whereas the dark cap is composed of the object's back-facing polygons (w.r.t. the light) whose vertices have been

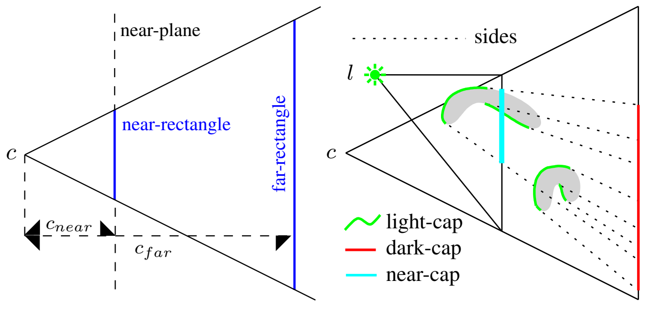

Figure 2: Some terminology. c is the camera position, l is a point light source. The picture on the right depicts the various shadow volume elements: the *light-cap* is the set of polygons facing the light. The *dark-cap* is the set of polygons not facing the light, projected onto the far-plane. The light- and dark-cap are used in the *Z-fail* method. The *near-cap* is the intersection of the camera's near-rectangle with the shadow volume. The near-cap is used in the new *ZP+* method. The *sides* are used in all three methods.

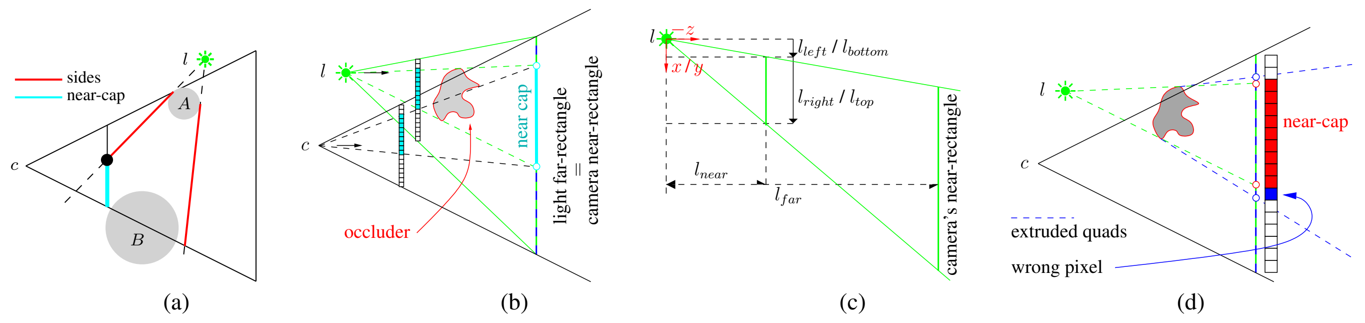

Figure 3: **(a)** In this situation, the *Z-pass* method fails because part of the shadow volume sides has been clipped away by the near-plane, resulting in object B being wrongly lit. Our method also rasterizes the near-cap (light-blue) which corrects *Z-pass* defects. **(b)** Rasterizing the near-cap. **(c)**. Setting up the *OpenGL* projection matrix using the `glFrustum` function. **(d)** Rasterizing the same edge by two different means yields wrong pixels.

projected to the end of the shadow volume. The problem now lies at the far plane. If the dark cap is clipped, inner portions of the shadow volume will be exposed, producing incorrect stencil values. Everitt and Kilgard [2002] solved this problem by extending the far clipping plane and the projection of the dark cap along the shadow volume to homogeneous infinity.

This *Z-fail* method, while robust, incurs the added expense of rendering the light and dark cap geometry which must be processed separately since the dark cap is translated from its original position. This separation according to polygon orientation disables any advantage a meshed data structure (e.g. a triangle strip) might provide. Furthermore, since stencil counts are counted from the pixel's depth backward, shadow volumes that would normally be occluded by the existing depth buffer must be rendered, which may disable some acceleration that early z-culling might provide when rendering shadow volumes.

2.3 Shadow Mapping

Shadow maps were introduced by Williams [1978] as a simple, general, and efficient technique for generating shadows. They are now natively supported in graphics hardware. However, since it is an image-based technique, shadow mapping exhibits aliasing due to quantization and perspective projection. Recent work addresses these problems [Stamminger and Drettakis 2002; Wimmer et al. 2004; Martin and Tan 2004]. In particular, Chong and Gortler [2004] describe a technique for shadow maps with perfect sampling for planes of interest. Our contribution, *ZP+*, uses a similar approach in a different context. Our *ZP+* method effectively constructs a shadow map to initialize the stencil plane with a robust cap of the *Z-pass* shadow volumes. However, we do not store the nearest depth value, but instead increment the stencil buffer for each front-facing fragment. We still use shadow volumes to compute the precise *geometry* of shadow boundaries, and use a "shadow map" to repair the *topology* of clipped shadow volumes. As such, we do not suffer artifacts due to the finite image resolution of the shadow map. Figure 5 discusses the issue of rectifying the rules of polygon clipping with the stenciled rasterization of a shadow mask to construct a watertight seal that avoids shadow cracks.

2.4 Shadow Volumes Optimization

Recent research aims to enhance the shadow volumes algorithm. In *CC Shadow Volumes*, Lloyd *et al.* [2004] describe a method to reduce fill rate requirements by *culling* unnecessary occluders and *clamping* irrelevant shadow geometry. Aila and Akenine-Möller [2004] also reduce fill by using a hierarchical method that only performs shadow volume rasterization at the boundary of the shadow. Similarly, Chan and Durand [2004] reduce fill by using a shadow map to identify pixels that lie near shadow discontinuities. Shadow volumes are then rasterized only for these pixels to produce accurate shadow boundaries. Aldridge and Woods [2004] describe a robust method to produce correct shadow volumes for non-manifold geometry. Our contribution corrects the core of the shadow volumes algorithm, and is complementary to all these methods.

3 *ZP+*

As the previous section described in detail, the *Z-pass* approach processes shadow volumes efficiently but incorrectly when they intersect the near clipping plane, and methods designed to cap the clipped shadow volumes have proved less than robust. The *Z-fail* approach can be made robust but only after adding the expense of processing cap geometry and sacrificing the acceleration of early depth culling of occluded shadow volumes. This section describes the main contribution of this paper, the *ZP+* algorithm that corrects the near-clipping errors introducted by *Z-pass* at the expense of a single pass rasterization of the occluder into the stencil buffer. We restrict our description of *ZP+* to a single light-occluder pair interaction, as details on implementing a complete stencil shadowing system can be found elsewhere [McGuire et al. 2003].

The added cost of *Z-fail* motivates our search for a simpler method that produces equivalent results. Inspection of Figure 1 (right) reveals that *Z-pass*'s problem area is the pyramid-shaped volume between the light source and the camera's near plane. Any occluder lying within this region casts a shadow volume that intersects the near rectangle and causes *Z-pass* to malfunction.

The projection of the occluder from the light source onto the camera's near plane closes the gap created when the shadow volume is clipped. Previous methods which attempt to construct projected cap geometry [Diefenbach 1996; Batagelo and Junior 1999; Kilgard 2001] are computationally expensive and experience numerical problems when rendering because of the proximity of the cap geometry to the near clipping plane.

Whereas others project the light cap geometry onto the near plane to fix a clipped shadow volume, the *ZP+* approach rasterizes the light cap and uses the resulting fragments to initialize the stencil buffer, illustrated in Fig. 3b according to the following algorithm.

1. Position light frustum at light position.
2. If camera and light are on the same side of camera's near plane,
   (a) then orient light frustum parallel to camera frustum,
   (b) else orient light frustum antiparallel to camera frustum.
3. Fit the light frustum's far-rectangle to the camera's near-rectangle. See Figure 3c.
4. Rasterize front facing (w.r.t. the light frustum) scene geometry, accumulating fragment counts into the stencil buffer.
5. Execute standard z-pass shadow computation initialized with current stencil buffer.

A perspective view frustum is positioned at the light source and sheared so its far rectangle aligns with the camera's near rectangle. The rasterization of any light cap geometry that falls within this sheared light frustum generates the fragments corresponding to a cap of the portion of the shadow volumes clipped by the camera's near plane. We then use these fragments to initialize the stencil buffer to the values necessary to correct *Z-pass*.

This alignment provides an identification between every point in the camera's near-rectangle with a line segment connecting that point to the light. Thus, this skewed perspective allows us to initialize the stencil values for each one of these points simply by rendering light-front-facing polygons from the light's point of view.

Placing the light frustum at the light's position amounts to left-multipling a translation matrix to the camera's modelview matrix. If necessary, an antiparallel orientation results from a 180° rotation matrix about the frustum's y-axis. The light frustum is then sized using the values illustrated in Figure 3c.

When rasterizing the polygons inside the light frustum, we instruct the graphics library to increment stencil fragments and enable the culling of back-faces. Once the rasterization is complete, each stencil pixel contains precisely the number of entrances into shadow for the line segment starting at the light and ending at that pixel's 3-D position on the near-rectangle. We then proceed with the usual *Z-pass* method.

3.1 Light Frustum Setup

The only addition to the standard *Z-pass* method is the near-cap rasterization in the stencil buffer. We describe how to setup the OpenGL modelview and projection matrices for the near-cap rasterization.

The modelview matrix $\mathcal{M}_l$ is seen as a frame with origin at the light's position. This "light-frame" is setup so as to have the same vertical vector (Y) as the camera-frame, and a view vector (-Z) parallel to the camera view vector, and oriented towards the camera's near-plane. Thus, the transformation from the camera-modelview to the light-modelview involves a translation and possibly a rotation of 180 degrees around the up vector if the camera and the light lie on opposite sides of the camera's near-plane.

The projection matrix $\mathcal{P}_l$ is an off-centered perspective transformation. It can be setup using a call to `glFrustum`. To minimize numerical inaccuracies, we set it up directly as:

$$\mathcal{P}_l = \begin{pmatrix} \frac{2\alpha l_{far}}{c_{width}} & 0 & \frac{-2\Delta_x}{c_{width}} & 0 \\ 0 & \frac{2l_{far}}{c_{height}} & \frac{-2\Delta_y}{c_{height}} & 0 \\ 0 & 0 & \frac{l_{near}+l_{far}}{l_{near}-l_{far}} & \frac{2l_{near}l_{far}}{l_{near}-l_{far}} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

where $\alpha = 1$ if the light and the camera are on the same side of the camera's near-plane (otherwise $\alpha = -1$), c_{width} (*resp.* c_{height}) is the horizontal (*resp.* vertical) dimension of the camera's near rectangle, and $\Delta = l - c$, in camera space, i.e., the position of the light l in camera space. The matrix above is derived directly from the matrix given in the OpenGL reference book and Figure 3d.

We now send the occluder geometry to the graphics pipeline to effectively rasterize the near-cap. Since we want to count, for each pixel of the stencil buffer, the number of entrances into the shadow volume, we set the OpenGL states so as to (**1**) cull back faces (*resp.* front faces) if $\alpha = 1$ (*resp.* $\alpha = -1$). (**2**) increment the stencil buffer value for each fragment. (**3**) disable writing to the color and depth buffers. (**4**) disable depth testing.

The rest of the procedure follows exactly the *Z-pass* method.

3.2 Setting the Light's Near Plane

When rasterizing the near-cap, only the part of the occluder inside the light-frustum gets rasterized onto the stencil buffer. Let

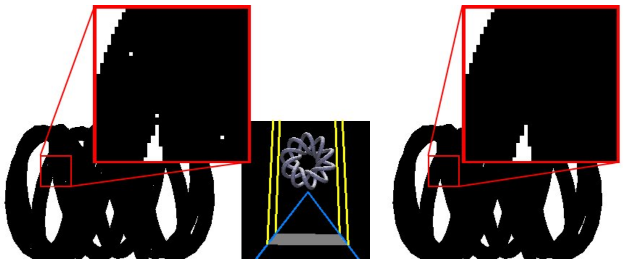

Figure 4: *Left.* Pixel artifacts: the closeup reveals 3 lit pixels that should be dark. On the full image, one can see 8 inverted pixels. *Middle.* View from above. The white rectangle is the camera near-rectangle, yellow is the light-frustum, blue is the camera frustum. The shadow receiver is a plane in front of the camera (not visible). *Right.* Artifact-free shadow.

us call the *light-pyramid* the pyramid formed by the light and its near-rectangle. If a part of the occluder lies in the light-pyramid, then it will not be rasterized onto the stencil buffer because it will be clipped by the light's near plane. Therefore, it is crucial that the light-pyramid be empty. If the light lies outside the occluder (which is assumed to be the case), it is always possible to find a light's near-plane distance l_{near} for the light-frustum so that the light-pyramid is empty. However l_{near} should be as large as possible so as to avoid precision loss. We give a simple procedure to calculate l_{near}.

If we know d, the distance from the light to the occluder, and d_{max}, the farthest distance from the light to its far-rectangle (the camera's near-rectangle), then $l_{near} = l_{far}\ d/d_{max}$.

3.3 Proposal for an OpenGL Extension

Recent NVIDIA hardware offers an OpenGL extension called NV_depth_clamp that – simply speaking – disables clipping due to the near and far planes and instead clamps the depth of the fragments to the [near, far] range. This extension would be useful to make sure we do not clip the part of the occluder in the light-pyramid. However, activating this extension would also disable clipping by the light's far-plane, which we clearly do not want. We believe that a slight modification to this extension to allow *clamping* on one side (e.g., the near-plane) and *clipping* on the other side (e.g., the far-plane) would preclude the computation of l_{near}, and ease the implementation of *ZP+*.

4 Discussion

Why is *ZP+* more efficient than *Z-fail*? First, we eliminate the need for the dark cap at infinity. *Z-fail* requires the occluder to be rendered twice, to increment and decrement the stencil for both the light- and dark-caps. *ZP+* does not need to rasterize the dark cap, since a *Z-pass*-style stencil test ensures the far plane never clips relevant shadow geometry.

Second, *Z-fail* may not take advantage of "batched geometry" such as triangle strips, because they prevent the classification of a vertex as belonging to the light-cap or the dark-cap. *ZP+* needs no such classification, and may enjoy the performance advantages of batched geometry.

Third, the proportion of capping geometry that is culled by *Z-pass* is much higher, since the size of the light frustum is much smaller than the camera frustum used by *Z-fail*. Since *Z-fail* renders capping geometry in the camera's frame, cap faces which do not project on the near plane must be culled manually. Conversely, since *ZP+* renders the cap in the light's frame, it may take advantage of the fast automatic culling capabilities of graphics hardware.

And fourth, from a fragment processing point of view, we believe that *Z-fail* may not take advantage of the depth-test optimizations featured in graphics hardware (early depth culling), if the depth-test is tuned to quickly *reject* fragments that *fail* the depth-test, instead of quickly letting them through. In that regard, *ZP+*, as *Z-pass*, are "depth-test friendly". We validate these predictions in Section 6.

ZP+ is compatible with all previous work on optimizing stencil shadow volume algorithms: hybrid shadows [Chan and Durand 2004], hierarchical shadow volumes [Aila and Akenine-Möller 2004], CC-shadow volumes [Lloyd et al. 2004], and other various optimizations [McGuire et al. 2003].

5 Avoiding Cracks

The implementation described above leads to visual artifacts as illustrated in Figure 4: when the camera's near-rectangle lies both in light and shadow, one can observe a few (less than 10 in practice) sparse pixels that are lit while they should be dark or vice versa. These artifacts may or may not be a concern depending on the application: we happily tested *ZP+* for a few hours before discovering the problem. A closer observation reveals that these wrong pixels lie on the boundary of the near-cap, along the edges where the sides and the near-cap meet on the near-plane (see the dark spot in Figure 3a). The screen-space coordinates of these edges are computed in two different ways: when rendering the near-cap, they are the projection of some silhouette edges with the special modelview $\mathcal{M}_l$ and projection $\mathcal{P}_l$ matrices. When rendering the sides, they are the coordinates of some clipped sides. Because of non exact arithmetic precision, the results of the two transformations differ slightly and the rasterizations of these edges are no longer pixel-wise identical. Figure 3d illustrates this discrepancy.

In some applications, it is desirable to eliminate these artifacts. We present a possible way to do so, which we have successfully implemented. The central idea to correct these artifacts is twofold. First, we leave the near-cap rasterization as is. Second, we manually clip shadow volume quads on the camera's near-plane. If a quad vertex A is found to be clipped, we replace its screen position by the one it would have as transformed with matrices $\mathcal{M}_l$ and $\mathcal{P}_l$, and clamp its depth value to lie on the near-plane. We call this replacement vertex *A's near-cap counterpart*. We implement this in a vertex program.

Let us examine an infinite quad Q intersecting the near-plane (Figure 5). Four cases can be distinguished:

(A): The near-plane does not intersect Q. There is nothing to do.

(B): The near-plane intersects both I_1 and I_2 (green line in Figure 5). The two clipped vertices are replaced with their near-cap counterpart. Thus, the quad's rasterization will perfectly match the corresponding part of the near-cap boundary.

(C): The near-plane intersects one infinite edge I_1 or I_2 and one finite edge E_m or E_∞ (cyan and blue lines in Figure 5).

(D): The near plane intersects E_m and E_∞ (red line in Figure 5).

In case **(C)**, depending on the orientation of the near-plane, one or three vertices will be clipped. As for case (B) we replace these by their near-cap counterpart. In doing so, alas, we create a triangular "hole" in the quad: Let us examine the case of the cyan near-plane in Figure 5. Assume that it is oriented so that vertex V_1 is clipped and replaced by vertex V_1^* while the three other vertices stay in place. Then the modified quad misses the dark-gray triangle in the figure. We fill the hole by rasterizing a new triangle $V_1V_2V_1^*$. The implementation details below explain how we efficiently create these corrective triangles using degenerate quads.

Case **(D)** is more involved. This geometric situation appears when the light-frustum is extremely skewed and flat, because the light is both far from the viewpoint and close to the near-plane. Figure 6 illustrates case **(D)** artifact. In this case, we cannot simply use the vertices' near-cap counterparts. Furthermore, case **(D)** conservatively detects the situation where the matrices $\mathcal{M}_l$ and $\mathcal{P}_l$ behave

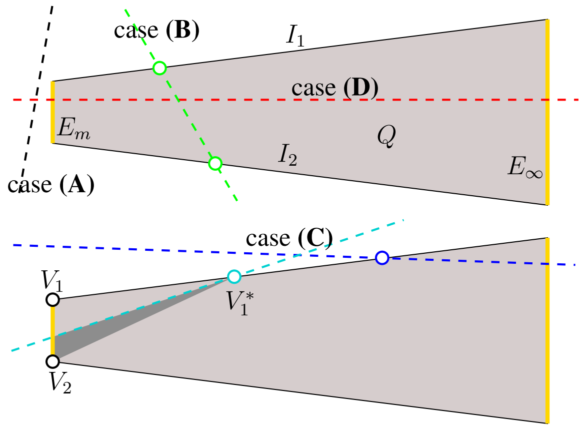

Figure 5: The quad Q has two infinite edges, I_1 and I_2, and two finite edges. One finite edge of Q, E_m, is also an edge of the mesh, while the other, E_∞, is finite in the angular domain, and lives at infinity. The dashed lines stand for the possible near-plane positions, up to symmetry.

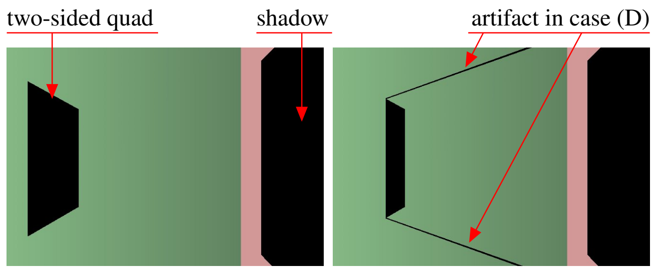

Figure 6: *Left.* The occluder is a two-sided quad. The light is on the left. *Right.* We push the near-plane forward to make the light as close as possible to the near-plane. Artifacts appear because numerical innacuracies make the near-cap wrongly translated to the right. Note that in general, case **(D)** does not yield artifacts of this size; the right image features the worst artifacts we obtained.

very poorly numerically. For these reasons, if case (D) is detected for some occluder, we temporarily switch to the *Z-fail* method, but for this light/occluder pair only.

The cases are explicitly examined during the search for silhouette edges. Our experiments show that, fortunately, case (D) is extremely rare. In a general, free walkthrough in our test scenes, case (D) almost never happened. When modelling specific camera paths to reveal case (D), the transition from *ZP+* to *Z-fail* and vice versa is smooth, with no artifacts. We dicuss the switch to *Z-fail* in more detail at the end of this section. Case (C) could be the cause of some artifacts, because we cannot perfectly correct the quad at the intersection between the near-plane and $E_{m|\infty}$. However, in our experiments, we do not notice artifacts due to case (C). Case (B) is the most common (about on par with case (A)), and we have seen that its correction with the near-cap counterparts is exact. To sum up, we have no more wrong pixels.

5.1 Vertex Shader Details

Figure 7 presents the vertex program for rendering the sides. The x, y, z input coordinates contain the position of the corresponding silhouette vertex. The w input coordinate is $0, 1, 2$ or 4. 0 and 1 carry their standard "homogeneous" meaning: 1 leaves the vertex in place. 0 indicates that the vertex must be extruded infinitely away from the light (lines 6, 7). If the vertex happens to be clipped by the near-plane (line 12), then the vertex $(x, y, z, 1)$ (lines 13, 14) is transformed as in the near-cap rasterization (lines 15, 16) and is projected on the near-plane (line 17). A vertex V with $w = 0$ or 1

```
 0 uniform mat4 lightMVMatrix;
 1 uniform mat4 lightPMatrix;
 2 uniform vec4 lightPosition;

 3 void main(void)
   {
 4     vec4 eyePos = gl_Vertex;
       // SHOULD WE EXTRUDE TO INFINITY ?
 5     if( gl_Vertex.w == 0.0 || gl_Vertex.w == 4.0)
       {
 6         eyePos = eyePos - lightPosition;
 7         eyePos.w = 0.0;
       }
       // SHOULD WE STICK THE VERTEX TO ITS ORIGINAL POS ?
 8     if( gl_Vertex.w == 2.0 )
       {
 9         eyePos.w = 1.0;
       }

       // TRANSFORM IN CAMERA FRUSTUM
10     eyePos = gl_ModelViewProjectionMatrix * eyePos;

       // IS IT MOVABLE ...
11     if( ( gl_Vertex.w < 1.5 ) &&
        // ... AND CLIPPED BY THE NEAR PLANE ?
           ( ( eyePos.w <= 0.0 ) || ( eyePos.z < - eyePos.w ) ) )
12     {   // YES: RE-TRANSFORM THE VERTEX IN LIGHT FRUSTUM
13         eyePos = gl_Vertex;
14         eyePos.w = 1.0;
15         eyePos = lightMVMatrix * eyePos;
16         eyePos = lightPMatrix * eyePos;
17         eyePos.z = -eyePos.w; // PUSH VERTEX ON NEAR-PLANE
       }
       // OK, WE ARE DONE
18     gl_Position = eyePos;
   };
```

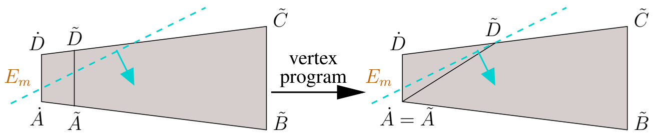

Figure 7: The vertex program for rendering the sides.

is called a *floating* vertex and is noted $\tilde{V}$.

If $w = 2$ or 4, the vertex is kept in place ($w = 2$) or extruded ($w = 4$) but is **not** modified if it happens to be clipped (line 11). A vertex with V $w = 2$ or 4 is called a *nailed* vertex and is noted $\dot{V}$. Nailed vertices are used to create the corrective quads, as we explain now.

In case (C), one need to add a corrective triangle to fill the "hole" left by the modified quad. However we use a corrective *quad* instead. The reason is twofold. First, this permits to handle symmetric cases automatically (e.g., when the near-plane intersects $[I_1$ and $E_m]$ or $[I_2$ and $E_m]$, the same corrective quad is used). Second, and perhaps most importantly, we can append these corrective quads to the vertex array containing the sides of the shadow volume, thus avoiding the burden of having to treat them in a special vertex array.

Let AD be a silhouette edge yielding a quad Q in case (C). AD is oriented so that the normal of triangle $ADlight$ points away from the shadow volume.

Case (C1) corresponds to the cyan dashed line (the near-plane intersects E_m). The corrective quad for case (C1) is $\{(A, 2); (A, 1); (D, 1); (D, 2)\}$.

Case (C2) corresponds to the blue dashed line (the near-plane intersects E_∞). The corrective quad for case (C2) is $\{(A, 0); (A, 4); (D, 4); (D, 0)\}$.

In all cases, two vertices of the corrective quad collapse and the corrective quad becomes a corrective triangle. The bottom of Figure 7 explains the case (C1): The edge E_m of quad Q is duplicated so as to create a second (corrective) quad, two of whose endpoints are nailed.

5.2 Switching to *Z-fail*

In the broader context of a full stencil shadowing system, that is, with many lights and many occluders, switching from *ZP+* to *Z-fail* for a single light/occluder pair is not possible. Indeed, *ZP+* initially renders the scene into the depth buffer, with respect to finitely remote near- and far-planes. After this is done, rasterizing a shadow volume as in the *Z-fail* method would produce fragment depths incompatible with the values stored in the depth-buffer. Therefore, one has two choices for the "switched" method: if the NV_depth_clamp is available, the *Z-fail* method can be used without pushing the far-plane to infinity. Otherwise, one can use the *ZF+* approach that we will describe in Section 7. This is guaranteed to give a correct result, because the switch is made upon encounter of case (D), i.e., when the light is close to the near-plane. Unless a silhouette edge is large enough to pass through both the near- and far-planes, this implies that case (D) will not be triggered for that light/occluder pair with the *ZF+* approach.

Note that the "switched" method should be used only for those light/occluder pairs for which a case (D) has been detected. For efficiency, one could treat these light/occluder pairs together.

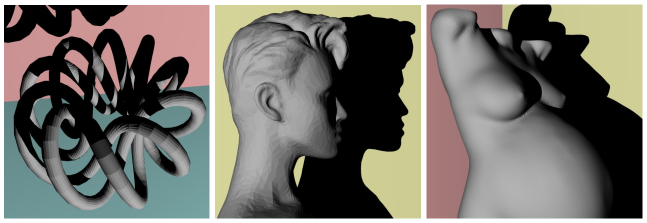

Figure 8: The occluders used in our tests. The *twisted torus* (12000 triangles) is interesting because of its highly complex shadow volume. The *head* (20222 triangles) and the *pregnant woman* (83666 triangles) are used to test the increase in geometric data.

6 Performance

We implemented the three methods: *Z-pass*, *Z-fail* and *ZP+*. *Z-pass* does not produce correct shadows in general, but is used as reference for performance comparison. To make their comparison as fair as possible, we have heavily optimized each method:

- **Common optimization to all three methods:** The shadow volume sides' coordinates are stored in a dynamic vertex buffer (we used the ARB_vertex_buffer_object extension). The second rasterization of the sides (for decrementing in the stencil buffer) can be executed in a single OpenGL call.

- ***Z-fail*:** A version of the occluder's mesh is stored as independent triangles in a vertex array, wherein each entry has the vertex coordinates and the triangle's normal. A vertex program computes whether or not to project the vertices of this mesh to infinity, based on its facingness with respect to the light. This rasterizes the dark- and light-caps. This mesh is rasterized twice: once to increment the stencil buffer and once to decrement it.

- ***ZP+*:** A version of the occluder's mesh is stored as a triangle strip in a vertex array, wherein each entry only contains the vertex coordinates. This mesh is rendered only once to rasterize the near-cap. Sides are rasterized together with additional corrective quads, using a special vertex program described in Section 5.

Our test scenes consist of a central object (the occluder, see Figure 8), a surrounding box, and a point light source. The rendering of a frame includes the brute-force computation of silhouette edges, the preliminary pass to fill the depth-buffer, the shadow volume rasterization, and the lighting pass to shade lit pixels.

Comparing the three methods is no trivial matter. The rendering times are dependent on the scene in both its complexity and spatial configuration. They also largely depend on the pixel coverage of

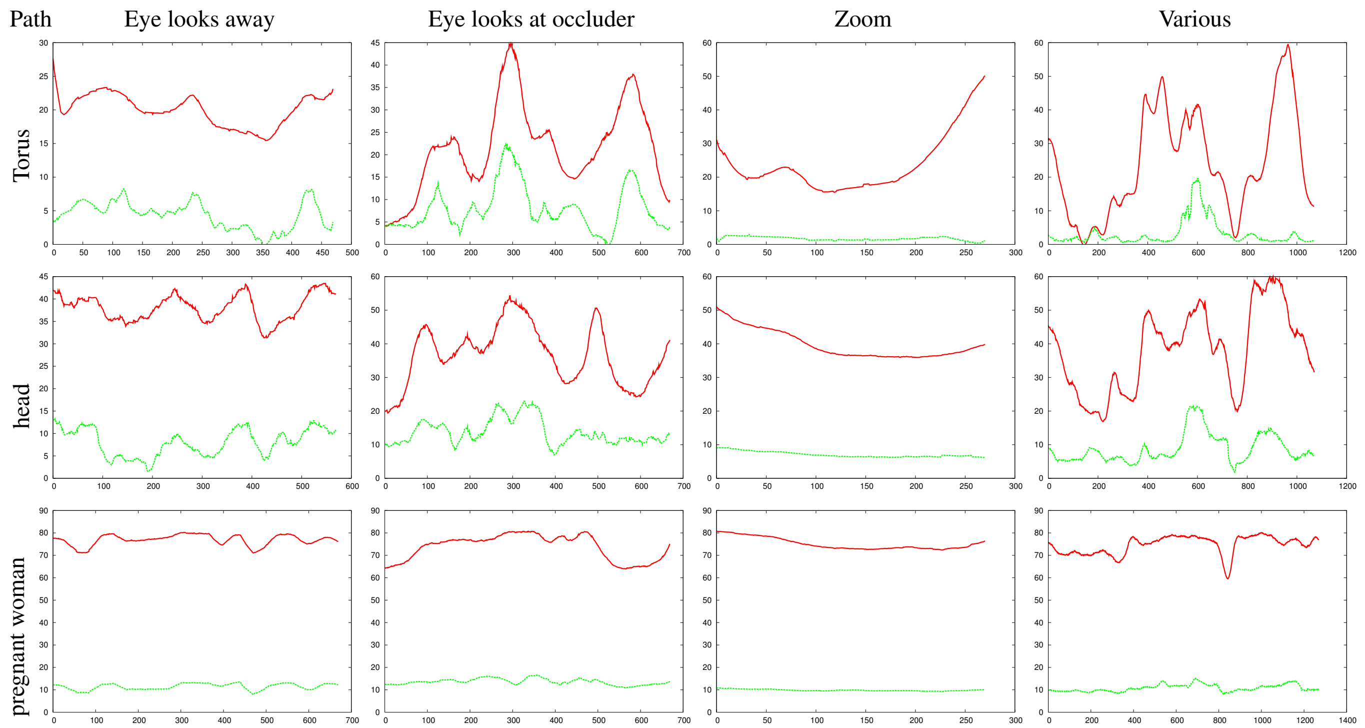

Figure 9: Relative timings – the lower the better. Each plot depicts the varying rendering time of *ZP+* (green) and *Z-fail* (red) relative to *Z-pass* (in %). The X axis is the frame number. For example, a value of 30 at frame 100 means that around frame 100, the average rendering time for each frame is 30% slower than *Z-pass*. Each line plots the results for one occluder. Each column plots results for one specific camera path: the paths used the first two columns leave the camera entirely *inside* the shadow, hence *Z-pass* gives a wrong shadow and can not be used in an application. The *Zoom* path is a forward zoom where the occluder covers only about 100 pixels at the first frame and occupies the whole screen at the last frame; the shadow begin cast toward the left of the screen. The *Various* path makes the camera fly around the scene in a random fashion.

the shadow-volume sides. Furthermore, the set of stencil pixels modified during the shadow-volume rasterization is very different between *Z-pass* and *ZP+* (the near-cap is added), and between *ZP+* and *Z-fail*. In particular, the set of stencil pixels modified in *ZP+* and the set of pixels modified in *Z-fail* form a partition of the set of pixels touched when rasterizing the capped shadow-volume. The respective size of these sets is not balanced in general.

The performance measurements below include the correction of the artifacts, which we found incur no significant overhead.

6.1 GeForce4 Performance

We first tested *ZP+* on a desktop PC (2.4GHz Xeon) with a GeForce4 Ti 4800. Results for these tests are presented in Figures 9 and 10.
• As expected, *ZP+* is slower than *Z-pass*.
• *ZP+* is faster than *Z-fail*.
• *ZP+* performance is closer to *Z-pass* than to *Z-fail*. This gets more salient as the geometric complexity of the occluder increases.

To accommodate for the high variability of the rendering time, we designed several specific camera paths, each of which highlights a different aspect of the methods' behavior. Figure 9 shows the behavior of *ZP+* and *Z-fail* relative to *Z-pass*. The first two columns of this figure indicate paths that place the camera entirely in shadow, in which case the *Z-pass* method always computes a wrong shadow and necessitates an alternative method, either *Z-fail* or *ZP+*. The "*Eye looks away*" camera path places the camera facing a wall, where most depth tests fail. This test favors *ZP+*. The "*Eye looks at occluder*" path places the camera viewing the occluder, where most depth tests pass. This favors *Z-fail*, which is illustrated by the curves being closer. In both tests however, using *ZP+* is profitable.

Another observation we can make from Figure 9 is that the relative timings for *Z-fail* increase with the geometric complexity of the occluder, while the relative timings of *ZP+* stabilize around 10%. The "flattening" of the curves observed with the *pregnant woman* occluder is due to the graphics pipeline becoming geometry bound. Compared to *Z-fail*, *ZP+* performs especially well with highly detailed 3D models, as shown in Figure 10 (right).

Section 5 explains why, in rare cases, one cannot use *ZP+* and must switch to *Z-fail* (or the modified *Z-fail* presented earlier). In all the tests of Figure 9, the switch to *Z-fail* was never triggered. We setup a special camera path to reveal this switch. The path makes the camera move slowly when the near-plane pass through both the light and the occluder. Relative timings are depicted on Figure 10 (left). As expected, the switch is triggered very seldom.

6.2 GeForce FX Performance

We also tested *ZP+* on a desktop PC (3.0GHz Xeon) with a GeForce FX 5900 Ultra. Results for these tests are presented in Figure 11. In particular, we enabled the use of the EXT_stencil_two_side extension (not available on GeForce4) so that rendering the sides and the caps is done only once in *Z-fail*, and rendering the sides is done only once in *ZP+*

The vertex and fragment processing of the GeForce FX is faster than the GeForce4. Thus, the plots in Figure 11 discriminate *Z-fail* and *ZP+* in an interesting way, which was not apparent with the GeForce4: When the occluder has small geometric complexity, the performance of *Z-pass* versus *Z-fail* depends highly on the geometric configuration of the scene (see the plots on the left side of the figure). Although the number of processed vertices is much larger in *Z-fail*, *Z-pass* (and thus, *ZP+*) happens to be slower in some geometric configurations of the scene. We may deduce that when the occluder is simple, the vertex processing time is negligible. Furthermore, the performance of *Z-pass* / *ZP+* varies with the number of

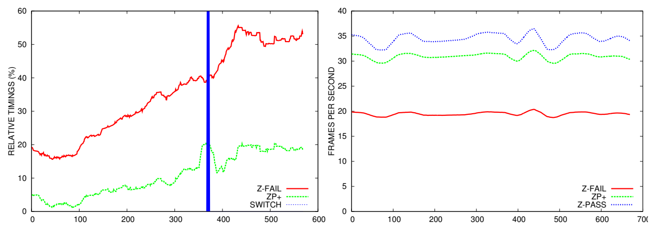

Figure 10: *Left.* The blue bar indicates frames where the use of *Z-fail* was detected as needed. One clearly observe the increased rendering times for *ZP+* around these frames. *Right.* Frame rates for the *Eye looks away* path and the *pregnant woman* occluder. *Z-pass* tops at 35fps but give wrong shadows all along this path. *ZP+* gives correct shadows and yet can render 50% more frames per second than *Z-fail*.

fragments that pass the depth test, in the opposite way that *Z-fail*'s does.

As occluder complexity rises, *ZP+* always performs better because the main bottleneck comes from the vertex processing stage, and *ZP+* takes advantage of triangle strips (see the plots on the right side of Figures 10 and 11). The plots in Figure 11 clearly show that *ZP+* incurs only a marginal overhead to *Z-pass*. As a result, we now have two methods to robustly rasterize a shadow volume. Our observations show that for an occluder with small geometric complexity, the cost of shadow volume rasterization depends heavily on the proportion of fragments that pass (*Z-pass* and *ZP+*) or fail (*Z-fail*) the depth test: *ZP+* may be slower than *Z-fail* but this is inherently due to *Z-pass* being slower than *Z-fail* in that case. This is a surprising observation which opens new tracks of research: It does not seem trivial to know beforehand what proportion of fragments will pass or fail the depth test for a given shadow volume and scene.

We also performed tests with an ATI Radeon 9700. Our observations were inline with those from the NVIDIA experiments.

7 Conclusions

We have presented a simple method called *ZP+*, to correct the defects of the *Z-pass* method for the rendering of hard shadows. *ZP+* only involves the rasterization of the light-occluding mesh with special transformation matrices. While *ZP+* is mathematically exact, the discrete nature of computer calculus yields some minor artifacts. We have described both the reasons of these artifacts and an efficient way to remove them. We have seen that *ZP+* is generally faster than *Z-fail* and behaves particularly well with large meshes. We also have described a simple modification to the NV_depth_clamp OpenGL extension that would simplify some aspects in the implementation of *ZP+*.

The observations made when testing with a GeForce FX yields the following question: can we efficiently decide whether to use *ZP+* or *Z-fail* if its occluder is simple enough that processing shadow volume fragments outweighs the processing of its vertices?

We believe it is possible to express $\mathcal{P}_l$ in a different way, so as to avoid numerical inaccuracies which, for now, force us to switch to *Z-fail* in rare cases.

Finally, we can observe that *ZP+* can also be adapted to a *Z-fail*-like method (that we call *ZF+*), by capping the far-plane instead of the near-plane (i.e., rasterizing a *far-cap*). We would like to examine *ZF+* and *ZP+* in more detail in the search for a heuristic that would tell when to use one or the other.

Acknowledgments We are grateful to Mark Kilgard and Eric Chan who took the time to discuss *ZP+* with us. Thanks to Sylvain Paris and Xavier Decoret for their comments on early drafts. This research funded in part by the NSF ITR CCR-0219594 and a CNRS-UIUC partnership.

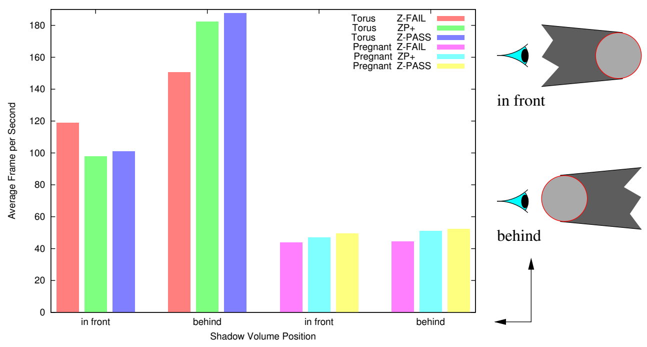

Figure 11: Average framerate on a GeForce FX 5900 Ultra, using two-sided stencil. The camera makes a forward zoom towards the occluder while the shadow volume admits 2 positions relative to the camera.

References

AILA, T., AND AKENINE-MÖLLER, T. 2004. A hierarchical shadow volume algorithm. In *Proceedings of Graphics Hardware 2004*, Eurographics Association, Eurographics, 15–23.

ALDRIDGE, G., AND WOODS, E. 2004. Robust, geometry-independent shadow volumes. In *Proc. 2nd Int. Conf. on Comp. graphics and Interactive Techniques in Austalasia and Southeast Asia (Graphite)*, ACM Press, vol. 2, ACM, 250–253.

BATAGELO, H. C., AND JUNIOR, I. C. 1999. Real-time shadow generation using bsp trees and stencil buffers. In *Proc. SIBGRAPI 99*, 93–102.

BILODEAU, B., AND SONGY, M. 1999. Real time shadows. In *Creative Labs Sponsored Game Developer Conference*, Creative Labs Inc.

CARMACK, J. 2000. E-mail to private list. Published on the NVIDIA website.

CHAN, E., AND DURAND, F. 2004. An efficient hybrid shadow rendering algorithm. In *Proc. Eurographics Symposium on Rendering*, Eurographics Association, Eurographics, 185–195.

CHONG, H., AND GORTLER, S. J. 2004. A lixel for every pixel. In *Proc. Eurographics Symposium on Rendering*, Eurographics Association, Eurographics.

CROW, F. C. 1977. Shadow algorithms for computer graphics. In *Proc. SIGGRAPH*, ACM Press, ACM, 242–248.

DIEFENBACH, P. J. 1996. *Multi-pass Pipeline Rendering: Interaction and Realism through Hardware Provisions*. PhD thesis, University of Pennsylvania.

EVERITT, C., AND KILGARD, M. J. 2002. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. Tech. rep., NVIDIA.

HASENFRATZ, J.-M., LAPIERRE, M., HOLZSCHUCH, N., AND SILLION, F. 2003. A survey of real-time soft shadows algorithms. *Computer Graphics Forum 22*, 4 (December), 753–774. State-of-the-Art Reviews.

HEIDMANN, T. 1991. Real shadows, real time. In *IRIS Universe*, vol. 18, Silicon Graphics, Inc, 23–31.

KILGARD, M. J. 2001. Robust stencil shadow volumes. In *CEDEC Presentation, Tokyo*.

LLOYD, B., WENDT, J., GOVINDARAJU, N., AND MANOCHA, D. 2004. CC shadow volumes. In *Proc. Eurographics Symposium on Rendering*, Eurographics Association, Eurographics.

MARTIN, T., AND TAN, T.-S. 2004. Anti-aliasing and continuity with trapezoidal shadow maps. In *Eurographics Symposium on Rendering*, Eurographics Association, Eurographics.

MCGUIRE, M., HUGHES, J. F., EGAN, K. T., KILGARD, M. J., AND EVERITT, C. 2003. Fast, practical and robust shadows. Tech. rep., NVIDIA.

STAMMINGER, M., AND DRETTAKIS, G. 2002. Perspective shadow maps. In *Proc. SIGGRAPH*, ACM Press, ACM, 557–562.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. In *Proc. SIGGRAPH*, ACM Press, ACM, 270–274.

WIMMER, M., SCHERZER, D., AND PURGATHOFER, W. 2004. Light space perspective shadow maps. In *Proc. Eurographics Symposium on Rendering*, Eurographics Association, Eurographics.