High Performance Graphics (2010)
M. Doggett, S. Laine, and W. Hunt (Editors)

Space-Time Hierarchical Occlusion Culling for Micropolygon Rendering with Motion Blur

Solomon Boulos[1] Edward Luong[1] Kayvon Fatahalian[1] Henry Moreton[2] Pat Hanrahan[1]

[1]Stanford University [2]NVIDIA Corporation

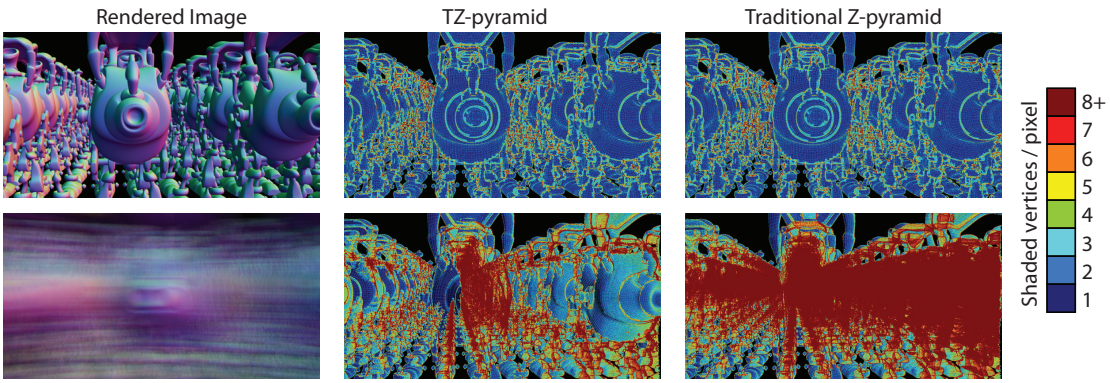


Figure 1: *A scene consisting of 15 rows of soldiers rendered at 1080p without motion blur (top) and with 1300 pixels of motion blur (bottom). The visualiations depict the number of shaded micropolygon vertices per pixel (shown at the beginning of the frame) when performing occlusion culling with the tz-pyramid (middle) and a traditional z-pyramid (right). Rendering with a traditional z-pyramid for occlusion culling shades 4.1x as many micropolygons than are visible. The tz-pyramid culls scene geometry more effectively, shading only 1.2x as many vertices than are visible, a 3.5x improvement.*

Abstract

Occlusion culling using a traditional hierarchical depth buffer, or z-pyramid, is less effective when rendering with motion blur. We present a new data structure, the tz-pyramid, that extends the traditional z-pyramid to represent scene depth values in time. This temporal information improves culling efficacy when rendering with motion blur. The tz-pyramid allows occlusion culling to adapt to the amount of scene motion, providing a balance of high efficacy with large motion and low cost in terms of depth comparisons when motion is small. Compared to a traditional z-pyramid, using the tz-pyramid for occlusion culling reduces the number of micropolygons shaded by up to 3.5x. In addition to better culling, the tz-pyramid reduces the number of depth comparisons by up to 1.4x.

1. Introduction

Motion blur is an important effect for cinematic-quality rendering and is present in almost all computer generated films. Lately, there has been increasing interest in adding motion blur to real-time rendering systems. While the problem of rasterizing moving objects has received some attention [CCC87, AMMH07, FLB*09], the related problem of occlusion culling for moving objects has not.

A simple way to cull moving objects with an existing static occlusion-culling algorithm is to test an object for occlusion using a conservative bound over the object's range of motion. Unfortunately, under large motion, this bound is overly conservative, diminishing the effectiveness of culling. Figure 1 shows a scene rendered with increasing amounts of motion. With a large amount of motion (Figure 1-bottom), the renderer shades 4.1x more micropolygon vertices than are ultimately visible. Work is wasted both to generate these vertices and process them in subsequent pipeline stages, such as shading and rasterization.

The goal of occlusion culling is to discard occluded portions of objects as early as possible in the rendering pipeline to improve overall performance. To do so, this paper extends the image-space z-pyramid of Greene et al. [GKM93] to support effective culling of motion blurred objects. In particular, we present a new data structure that aggregates image-space

depth values hierarchically in both space and time: the tz-pyramid. The tz-pyramid is much more effective than a traditional z-pyramid when rendering scenes with motion blur. For the scene in Figure 1-bottom, occlusion culling with the tz-pyramid reduces the number of shaded micropolygon vertices by more than 3.5x compared to culling with a traditional z-pyramid. While providing better culling, the tz-pyramid also reduces the number of depth comparisons by 1.4x.

2. Background and Related Work

2.1. Occlusion Culling Using the Z-Pyramid

In z-buffer based renderers, an object is occluded if its depth is farther than the contents of the z-buffer at all visibility sample locations it overlaps. Instead of checking each value individually, occlusion tests are often accelerated by using a hierarchical z-buffer, or *z-pyramid* [SS89, GKM93]. A z-pyramid is a quad-tree where each node stores the maximum depth value (i.e. the farthest value) for a spatial region of the screen. The leaf level of the z-pyramid is the multi-sample z-buffer. The root level is a single value equal to the maximum depth in the multi-sample z-buffer. When the z-buffer is updated, the z-pyramid is also updated in a bottom-up fashion.

To determine if an object is occluded, the z-pyramid is tested recursively. For a given node in the z-pyramid, the object's minimum depth over that spatial region is compared against the value stored in that node. If the node's z-value is closer than the object's minimum depth, the object is occluded in that region of the screen. If the object is closer than the value in the z-pyramid, the object *may* be visible. In this case, the four children of the node are tested recursively. The object is occluded (and may be culled) only if all four children determine the object is occluded in their respective screen regions. If the test reaches the leaves of the z-pyramid, the recursion terminates and either returns true (the object is occluded) or false (the object may not be occluded).

For an occlusion test, the z-pyramid returns true only when a test against all overlapped leaves would return true as well. However, by pre-aggregating maximum depth information for large regions of the screen, the z-pyramid requires only a few depth comparisons (against values in higher tree levels) to quickly cull large objects. Therefore, the z-pyramid is an acceleration structure used to avoid checking object depth against all overlapped leaf cells. As an optimization to avoid visiting many tree levels, it is common to begin traversal of the z-pyramid at the level where the spatial extent of nodes is roughly the size of the object [GKM93].

To support occlusion culling, a rendering system must be able to compute the screen regions covered by an object, as well as the object's depth in these regions. For complicated objects, this information may be difficult to compute exactly without performing the expensive work occlusion culling seeks to avoid. Instead, conservative bounds are used for occlusion tests. When the conservative bounds are tight, occlusion culling is more effective.

Figure 2: *A simplified REYES pipeline. Base patches are provided as input to Split which recursively generates sub-patches. When a splitting threshold is reached, patches are sent to Dice producing an output grid of micropolygons. The grid is then shaded and rasterized.*

2.2. REYES Occlusion Culling

This paper focuses on occlusion culling in the context of a simplified REYES-style [CCC87] pipeline shown in Figure 2. This pipeline adaptively subdivides application-provided input primitives into many small sub-primitives (Split), then tessellates the sub-primitives into meshes of micropolygons called grids. Programmable displacement (Dice) and surface shading (Shade) occur at each grid vertex prior to stochastic rasterization (Rast). Our implementation uses a z-buffer to resolve visibility of rasterized fragments.

Occlusion culling may be used at three major places in our simplified REYES pipeline: during Split, just before Dice, and just before Shade (see Figure 2). Each pipeline location has a different cost and benefit associated with performing occlusion culling. Culling during the splitting process has the maximum benefit, but uses coarse bounds that incorporate displacement expansion and the control cage of Bézier patches. Culling prior to dicing (like [HMAM09]) has tighter bounds, but incurs the cost of splitting all primitives into diceable subpatches. Culling before shading has the tightest bounds (taken from the final geometric positions), but must generate the micropolygons for the entire scene.

Unfortunately, there is very little literature related to occlusion culling for REYES-based systems. The original REYES description did not explicitly include occlusion culling [CCC87]. More recent versions of Pixar's RenderMan [AG00] have some form of hierarchical occlusion culling, but it is unclear if it leverages temporal resolution to effectively cull moving objects or not.

The Gelato system [WGER05] rasterizes diced grids before shading to determine if they should be shaded. If the grid is visible (i.e. it is unoccluded for at least one time or lens sample), it is shaded and then rasterized again. This is an expensive, but effective way to reduce shading work. As mentioned previously, however, culling at this point does not avoid the cost of generating the micropolygons for the full scene. While not focused on micropolygon rendering, Akenine-Möller et al. [AMMH07] proposed, but did not implement, a simple occlusion-culling scheme that would maintain multiple z-pyramids, one for each time sample. We will later refer to this approach as tz-slice.

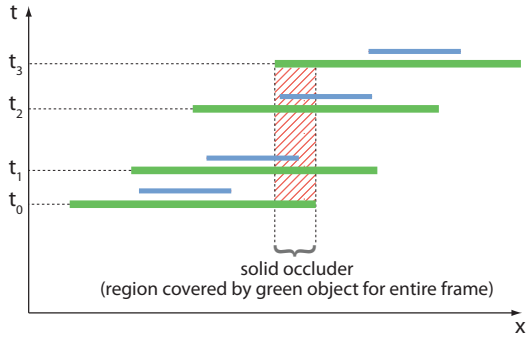


Figure 3: *A simple occlusion scenario with two objects both moving left to right during a frame. The occludee (blue) is behind the occluder (green) for each instant in time, but would not be culled by a traditional z-pyramid. A traditional z-pyramid only discerns occlusion when the occludee is always behind the region we refer to as the* solid occluder.

2.3. GPU Occlusion Culling

In modern GPUs, the traditional z-pyramid has been adapted for efficient, fixed-function occlusion culling [Mor00, AMHH08] (referred to as HiZ by ATI and ZCULL by NVIDIA [NVI08]). Occlusion culling in GPUs aims to reduce both rasterization cost and the depth buffer comparisons that occur as part of "early-z". Before quad-fragments are shaded, the covered depth values are compared against the multi-sample depth buffer. If all the covered depth values of a quad-fragment are farther than the values in the depth buffer, the quad-fragment is discarded removing unnecessary shading work. This test is referred to as early-z because it occurs prior to shading and frame-buffer blending.

The original z-pyramid focused only on determining "is occluded". To avoid performing depth comparisons when an object is definitely visible, an additional z_{min} pyramid may be maintained to determine "is not occluded" [SS89, Gre95, AMS03]. The fast z-clear mechanism in HyperZ [Mor00] also has this benefit for the first layer of polygons drawn (knowing that a depth tile is clear is equivalent to knowing that its closest z value is z_{near}).

High-performance implementations also compress the multi-sample depth buffer and z-pyramid to reduce the cost of reading the structures (see [HAM06] for both a survey of patents and a contributed compression scheme in this area). Removing several layers of the z-pyramid is also common practice (e.g. treating 2x2 pixel tiles as leaves, and only storing up to 8x8 pixel tiles as the coarsest level [Mor00]).

3. Occlusion Culling with Motion Blur

3.1. Problem Framing

Figure 3 illustrates a simple scenario involving a large occluder (in green) and a small occludee (in blue). The spatial extent of the two objects at four frame sample times is shown in the x-t space-time diagram. Both objects are moving from left to right. The blue object is fully occluded at all times and is not visible in the rendered image.

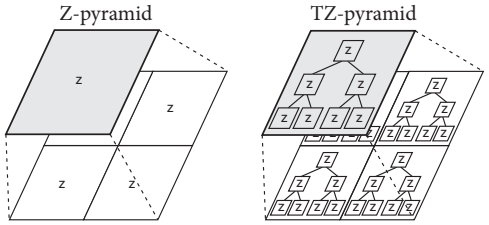


Figure 4: *A traditional z-pyramid (left) has a single z-max value per node while the tz-pyramid (right) has a tree of z-max values per node. Each value in the tz-pyramid represents the maximum z-value in a region of the depth buffer for a span of frame time.*

The green object covers a small region of the image for the entire frame. This highlighted region, which we refer to as the *solid occluder*, is the intersection of the green object's spatial extent at all times. The size of the solid occluder is the size of the object minus the amount of motion. When an object moves by an amount greater than its size in a single frame, no pixels are fully covered by that object (this makes motion blur similar to transparency, inspiring the term solid occluder). After the green object has been rendered, only non-leaf nodes of the z-pyramid that lie within the solid occluder contain z_{max} values closer than the far plane.

As stated earlier, a simple way to conservatively determine if the blue object is occluded is to bound its spatial extent and depth over the entire frame, and to test this bound against the z-pyramid. This test will classify the blue object as occluded only if its bound lies within the solid occluder for the entire frame. Given the motion of objects in Figure 3, this test will fail and the blue object will not be culled. Intuitively, this use of the z-pyramid for moving objects compares the union of an occludee's positions in a frame to the intersection of the occluder's positions. When objects are moving quickly, these loose bounds make culling much less effective.

Clearly, we would like to tighten bounds by performing occlusion checks over narrower windows of time. For example, the solid occluder of the green object does occlude a conservative bound of the blue object for the time interval $[t_0, t_1]$ (the blue object is still not occluded between $[t_2, t_3]$). The simplest way to introduce temporal information into occlusion culling is to have an array of N z-pyramids, where N is the number of unique times (for interleaved sampling) or time strata (for stratified sampling) used in visibility sampling. To perform an occlusion check, each z-pyramid is checked in sequence for occlusion using a tight bound for the occludee at each time (or time interval). We refer to this approach as "slicing" time or tz-slice [AMMH07]. Although it provides effective culling, tz-slice requires up to N times as many depth comparisons as the traditional z-pyramid.

When the occluder or the occludee is stationary, temporal resolution provides no occlusion-culling benefit. If the occluder is static, the solid occluder is equivalent to the occluder itself. Similarly, if the occludee is static, testing for occlusion at individual time samples is unnecessary as the object must be occluded for every time sample to be occluded. There is a need for occlusion-culling mechanisms to adapt to the motion of scene objects, only paying the cost of extra depth comparisons when high motion is present.

3.2. The tz-pyramid

We extend the traditional z-pyramid to store a hierarchy of z_{max} values at each node (Figure 4). Nodes in this hierarchy represent the maximum depth value for the corresponding region of space, for different spans of time. We refer to this data structure as a time-dependent z-pyramid or tz-pyramid.

The leaf nodes of the time pyramid store z_{max} values for narrow spans of time (e.g. single time samples or individual time strata). Inner hierarchy nodes correspond to larger time spans. The root of the time pyramid stores the region's maximum depth value for the entire frame and is equivalent to the single z_{max} value stored in a traditional z-pyramid node.

The tz-pyramid's time hierarchy allows occlusion culling to adapt to scene motion characteristics. When motion is small (recall there is little benefit to high time resolution) occlusion culling is performed using a few depth comparisons against trees nodes near the root level. When motion is large, finer time resolution is gained by performing depth comparisons against a larger number of nodes at a lower time level.

Our tz-pyramid implementation stores the temporal hierarchy in each spatial node as a complete binary tree. The total size of the tz-pyramid (including the multi-sample depth buffer) is 8/3 the size of the underlying multi-sample depth buffer (the spatial quad-tree contributes a factor of 4/3, the temporal binary tree contributes another factor of 2). For a 2 megapixel framebuffer with 16 samples per pixel, the tz-pyramid requires 342 MB of storage (214 MB for the internal nodes and 128 MB for the mutli-sample depth buffer). Implementations may reduce this storage cost by sacrificing spatial or temporal resolution, at the cost of culling efficacy.

A tz-pyramid is a generalization of both the traditional z-pyramid and the tz-slice. If the temporal hierarchy in the tz-pyramid is simply a single node, the tz-pyramid is equivalent to a traditional z-pyramid (each node represents a region of space for all values of time). Instead, if non-leaf nodes of the temporal hierarchy are removed the tz-pyramid is equivalent to the tz-slice.

3.2.1. Updating the tz-pyramid

Like a traditional z-pyramid, updating the tz-pyramid occurs whenever a new depth value is written into the z-buffer. With the INTERLEAVEUVT stochastic rasterization algorithm [FLB*09], each visibility sample has an associated tuple index i (for the INTERVAL algorithm, samples have associated intervals of time instead of indices). Given a new depth value for a particular visibility sample, the pyramid is updated bottom-up in the same fashion as the traditional z-pyramid. The only difference is that instead of updating a single depth value in each spatial node, a tree of depth values for the time samples is updated. Pseudocode for the updating procedure is shown in Algorithm 1.

Algorithm 1 UpdateTZPyramid(x, y, t, z)

```
spatialLevel = SpatialLeafLevel
currentTile = SpatialTile(x, y)
UpdateTimePyramid(currentTile, t, z)
while spatialLevel != RootLevel do
  parentTile = GetParent(currentTile)
  parentZ = GetPyramidValue(parentTile, t)
  zMax = z_near
  for all children of parentTile do
    zMax = max(zMax, GetTimeValue(child, t))
  end for
  if IsCloser(zMax, parentZ) then
    UpdateTimePyramid(parentTile, t, zMax)
    spatialLevel = parentLevel
    currentTile = parentTile
  else
    return
  end if
end while
```

The number of depth updates to the tz-pyramid (both multi-sample depth and inner node values) is bounded above by 8/3 the number of writes to the underlying multi-sample depth buffer. For reasonably depth sorted primitives, the number of updates does not increase with depth complexity (i.e. only the final visible geometry writes depth). Importantly, the algorithmic cost to updated z-pyramids is dominated by computing maximum values over the multi-sample depth buffer, not writing new values to the tree. The cost to compute the maximum value is identical for all configurations of the tz-pyramid, time-dependent or not, as the multi-sample depth buffer is the same in all cases.

3.2.2. Occlusion Culling with the tz-pyramid

Testing objects for occlusion with the tz-pyramid is also similar to the traditional z-pyramid. Given a particular temporal and spatial level, an object's minimum depth is compared to the values stored in the tz-pyramid. If the object cannot definitively be occluded for a given space-time tile, the subtiles are checked recursively.

To decide where to begin traversal, both a spatial and temporal level must be chosen. For the spatial level, we use a similar heuristic as Greene et al. [GKM93] for non-moving primitives: given an object size, determine the level at which

Algorithm 2 IsOccluded(spaceLevel, timeLevel)

```
for all timeRanges in timeLevel do
  for all spatialTiles covered by bounds[timeRanges] do
    tileZ = GetPyramidValue(spatialTile, timeRange)
    if IsCloser(zMin[timeRange], tileZ) then
      if spaceLevel == SpatialLeafLevel and
        timeLevel == TimeLeafLevel then
        return false
      else
        newSpace = NextLevel(spaceLevel)
        newTime += TimeStep(newSpace, timeRange)
        if !IsOccluded(newSpace, newTime) then
          return false
        end if
      end if
    end if
  end for
end for
return true
```

a single tile is large enough to cover the object. Instead of using the bounds over the full range of motion, we use the size of the object at the start of the frame. This avoids choosing a coarser spatial level simply because the object is moving.

Given a chosen spatial level, the temporal level is chosen to be the coarsest level for which the object does not *appear* to be moving at the given spatial scale. More precisely, the temporal level is computed by finding the first temporal sample which results in a different spatial region than the object at the beginning of the frame (the level is then the base-2 logarithm of the temporal sample index). This results in choosing levels near the root of the temporal hierarchy when the object is not moving much, but using finer levels if necessary.

Our algorithm always traverses to a finer level in space but only changes the temporal level when necessary. As with the spatial dimension of a traditional z-pyramid, it only makes sense to "refine" the temporal dimension and traverse to the finer levels of the tree. Deciding when to refine in time is based on applying the same condition used to initialize traversal: if at the new spatial resolution the object would appear to be moving, the temporal resolution is increased. For simplicity, our algorithm only refines time by one level instead of allowing an arbitrary jump.

Pseudocode for testing an object for occlusion is shown in Algorithm 2. For compactness, we've omitted passing in the object's screen-space bounds (*bounds*) and corresponding z_{min} values (*zMin*). These variables are arrays that hold values for the all ranges of time used in the tz-pyramid (including a single sample). If the traversal has reached either the spatial or temporal leaf level, that dimension can no longer be refined (i.e. NextLevel returns spaceLevel or TimeStep returns 0). Once the spatial and temporal leaf has been reached a definitive "not occluded" can be returned.

Like the traditional z-pyramid, the temporal hierarchy of the tz-pyramid reduces the number of depth comparisons compared to testing individual temporal samples. Both the tz-pyramid and the tz-slice return the same answer (occluded or not) for a particular object. Because the tz-pyramid allows occlusion tests to adapt to the amount of motion present, the tz-pyramid will usually discover an object is occluded with fewer depth comparisons than the tz-slice. However, if an object is not occluded, the tz-pyramid may perform more comparisons traversing the temporal hierarchy.

4. Evaluation

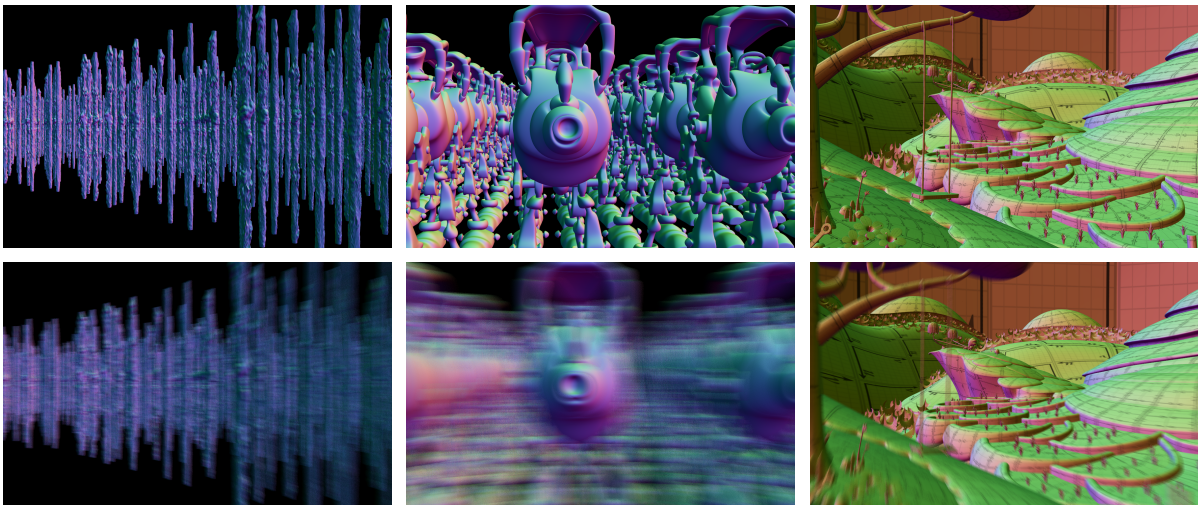
We implemented our algorithms as part of a software micropolygon rendering pipeline. To generate half-pixel area micropolygons, we use the DiagSplit [FFB*09] tessellation algorithm. For stochastic rasterization, we use the INTERLEAVEUVT algorithm [FLB*09] with 16 samples per pixel and a 2x2 pixel interleaving tile (64 interleaved time samples). Our set of test scenes (see Figure 5) include varying amounts of motion blur, occlusion granularity, and depth complexity. All scenes are rendered with camera path animations at 1080p.

Instead of reporting architecturally-dependent metrics (e.g. wall-clock time), we measure algorithmic quantities: the number of generated positions (diced points), shaded vertices (shaded points), and depth comparisons (equivalently reads) for both the inner nodes of the tz-pyramid and the underlying multi-sample depth buffer. As mentioned previously, the cost of updating the data structures is similar for all schemes and independent of depth complexity, so we do not include it here.

We begin our evaluation by determining where occlusion culling should be used in our micropolygon rendering pipeline. Next, we evaluate the temporal and spatial resolution tradeoff when using the tz-pyramid in terms of the three metrics listed above (diced points, shaded points, and depth reads). Finally, we show the tz-pyramid can drastically reduce the number of depth comparisons relative to tz-slice without impacting culling efficacy.

4.1. Pipeline Placement

In Figure 6 we evaluate four different options for pipeline placement using the tz-pyramid: before shading, before dicing, both before dicing and shading, and culling at every stage in the pipeline. Culling at every stage in the pipeline is the obvious choice to remove the downstream work, but may increase the number of depth comparisons performed. The data in Figure 6 demonstrates that with the exception of only culling prior to shading, culling at every stage performs *fewer* total depth comparisons than culling at later stages only. This is because culling a large object will perform fewer depth comparisons than culling the equivalent



	Average Motion	Occlusion Granularity	Average Depth Complexity
STICKS	Moderate (40 pixels)	Fine	4
ARMY	High (350 pixels)	Coarse	11
ZINKIA	Low (10 pixels)	Varying	6

Figure 5: *Frames from* STICKS, ARMY, *and* ZINKIA *with and without motion blur. The scenes have a range of motion amounts, occlusion granularity, and depth complexity.*

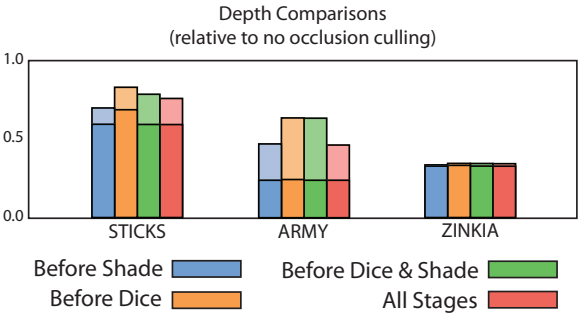


Figure 6: *Culling at every stage in the pipeline is the best choice in our tests. Each bar shows the number of multi-sample (dark) and pyramid inner node (light) depth reads relative to not culling (lower bars are better). Culling at all stages has low cost while culling the most downstream work.*

set of diced patches. While only culling prior to shading performs slightly fewer depth comparisons (due to the reduction in coarse comparisons), postponing culling until after dicing would be prohibitively expensive in practice (e.g. doing so results in roughly 3.5x more diced points in ZINKIA).

4.2. Impact of Temporal and Spatial Resolution

In Table 1 we compare 16 different spatial and temporal configurations for the tz-pyramid for the three different metrics. The spatial leaf sizes range from 2x2 pixels to 16x16 pixels (decreasing in area by 4x at each step). The temporal resolution ranges from 64 slices in time (full temporal resolution) down to a single value. Traditional z-pyramids are represented in the rightmost column for each scene. A step up or left increases the leaf storage by 4x, while steps down or right decrease storage by 4x. Stepping along the up-right diagonal roughly maintains the data structure footprint.

Increasing temporal resolution reduces the amount of dicing and shading work performed. At the same time, increasing temporal resolution often incurs a relatively small increase in the number of depth comparisons. The maximum benefit occurs for scenes with higher average amounts of motion (STICKS and ARMY). For STICKS, at 2x2 pixel leaves with full temporal resolution the tz-pyramid dices 1.8x fewer points than the traditional z-pyramid at the same spatial resolution. For ARMY, the relative benefit increases to 3.2x, while ZINKIA sees only a slight benefit from improved culling in the foreground. For higher amount of motions (as in Figure 1), the benefit of using the tz-pyramid increases.

If full spatial and temporal resolution would not be possible (e.g. because storage is at a premium), Table 1 suggests it is often preferable to sacrifice spatial resolution rather than temporal resolution for scenes with high motion. Intuitively, sacrificing temporal resolution inflates spatial bounds resulting in reduced resolution in both space and time. For ARMY, the 16x16 pixel spatial resolution with full temporal resolution results in fewer diced points, shaded points, and total depth comparisons than the similar footprint 2x2 pixel z-pyramid with no temporal resolution. On the other hand, scenes with low average motion (ZINKIA) or fine-grained occlusion (STICKS) prefer a more balanced tradeoff between spatial resolution (tied to the size of the occluders) and temporal resolution (tied to the amount of motion).

		STICKS				ARMY				ZINKIA			
spatial \ temporal		64	16	4	1	64	16	4	1	64	16	4	1
diced points	2x2	.48	.53	.69	.85	.23	.32	.48	.73	.30	.30	.31	.34
	4x4	.54	.58	.70	.86	.25	.33	.49	.73	.31	.31	.32	.35
	8x8	.63	.66	.74	.87	.29	.36	.50	.74	.33	.33	.34	.36
	16x16	.74	.76	.80	.89	.35	.40	.52	.75	.36	.36	.36	.38
shaded points	2x2	.40	.44	.61	.83	.28	.35	.49	.73	.36	.37	.38	.39
	4x4	.45	.49	.63	.84	.30	.36	.50	.73	.37	.38	.38	.40
	8x8	.54	.57	.68	.86	.33	.39	.51	.73	.39	.39	.40	.41
	16x16	.69	.71	.77	.88	.38	.42	.52	.74	.41	.42	.42	.43
zreads	2x2	.96	.78	.77	.96	.64	.48	.58	.91	.34	.33	.33	.34
	4x4	.80	.73	.77	.95	.63	.48	.58	.90	.33	.33	.33	.34
	8x8	.76	.73	.78	.94	.62	.49	.58	.89	.34	.34	.34	.35
	16x16	.82	.81	.83	.94	.61	.51	.59	.89	.36	.36	.36	.37

Table 1: *Number of diced points, shaded points and depth comparisons relative to not culling, for our three scenes at different spatial and temporal leaf sizes (in pixels and time slices, respectively). Increasing temporal resolution reduces the number of diced points by up to 3.2x and shaded points by up to 2.6x when there is high motion (*ARMY*). The relative increase in total depth comparisons is minimal due to the higher culling efficacy.*

4.3. Reducing Depth Comparisons

In Figure 7 we compare the number of coarse depth comparisons for the two algorithms on the ARMY scene with varying amounts of motion blur. Recall that the only performance characteristic that differs between the the tz-pyramid and tz-slice is the number of depth comparisons (i.e. both algorithms result in the same number of diced and shaded points). For low amounts of motion blur, the tz-pyramid results in a large reduction in coarse depth comparisons (approximately 18x). As the amount of motion increases, the tz-pyramid performs relatively more comparisons. Once the amount of motion passes 1100 pixels in a single frame (65% of the screen), the tz-pyramid performs more depth comparisons than tz-slice. This amount of motion is extreme and would be uncommon in practice.

For this configuration of 16 samples per pixel and 2x2 pixel interleaving pattern, the tz-pyramid can perform 64x fewer depth comparisons to determine an object is occluded. However, if an object is not occluded, the tz-pyramid must reach the leaves of the tree to make this decision. Traversing the 6 temporal levels ($\log_2(64)$) could produce a 6x increase in depth comparisons performed. The same is true for the traditional z-pyramid: occluded objects will often stop higher than the leaves, but unoccluded objects now require more depth comparisons. For scenes with sufficiently high depth complexity (and consequently more occlusion), the benefit of the tz-pyramid outweighs the potential increase in depth comparisons for visible objects.

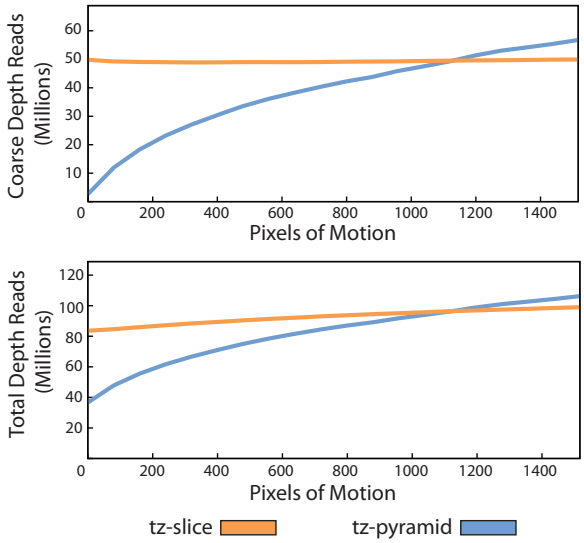


Figure 7: *The tz-pyramid reduces the number of coarse z-reads by 18x for* ARMY *with low amounts of motion. As the amount of motion increases, the relative benefit of the pyramid decreases. Once the motion becomes extreme, most of the scene becomes partially occluded, and the tz-pyramid performs more work.*

5. Discussion

The tz-pyramid extends the traditional z-pyramid by incorporating time resulting in effective culling when rendering scenes with motion blur. While we have focused on motion blur, the tz-pyramid would also work without modification for defocus blur as well if the INTERLEAVEUVT or INTERVAL stochastic rasterization algorithms are used [FLB*09] as both algorithms statically correlate the time and lens positions. However, unlike the sorted list of time values, the correspondence between time and lens positions is intentionally scrambled for higher quality. Taking two (u,v,t) tuples that are close together in time results in two lens positions that are far apart and vice versa. This suggests that while we

can expect a hierarchy in either time or lens position to have benefit, we likely cannot achieve both.

Our traversal initialization and walking strategy is just one heuristic of many possible. It is certainly not optimal, since it ignores information about the occluder. In particular, we do not take advantage of static occluders. Despite this, our algorithm works well in practice: for determining that an object is occluded, the tz-pyramid performs substantially fewer depth comparisons than the tz-slice. For objects that are not occluded, however, the extra time levels in the tz-pyramid may be a hindrance (similar to the traditional z-pyramid). Using the tz-pyramid for z_{min} culling would be straightforward, but was not the focus of our work.

Typical high-performance implementations use compressed representations for depth values to reduce bandwidth (including lossy compression of the z-pyramid to reduce storage). We have not analyzed this, but believe it would be an interesting topic for future work. In that same vein, the tz-pyramid itself could be transposed: instead of storing a hierarchy in time within each spatial cell, a time pyramid could have traditional z-pyramids at each node. The architecture-independent metrics we've presented are invariant under this change; however, it is likely that architectural tradeoffs would prefer one representation over the other.

While this work has been presented in the context of REYES-style micropolygon rendering pipeline, the tz-pyramid also applies to other systems that support motion blurred visibility [AMMH07, RKLC*10]. In those systems, both tessellation and rasterization work could be culled by using the tz-pyramid. Assuming that a fragment shading based system would still rely on an early-z mechanism to avoid shading work, the tz-pyramid should also reduce the number of depth comparisons performed as well.

Ultimately, we would like to see the tz-pyramid implemented as part of a future graphics pipeline. Motion blur is an important visual effect, but poor occlusion culling would deter its use. While this paper was algorithmic in focus, we believe variations of our description could result in interesting work to determine the right balance of complexity and effectiveness for a real-time implementation.

Acknowledgments

Support for this research was provided by the Stanford Pervasive Parallelism Laboratory funded by Oracle, NVIDIA, IBM, NEC, AMD, and Intel, the NSF Graduate Research Fellowship Program, and an Intel Larrabee Research Grant. We would also like to thank Kurt Akeley, Margarita Bratkova, James Hegarty, Mike Houston, Bill Mark, and Jonathan Ragan-Kelley for their valuable input. Zinkia scene © Zinkia Entertainment, S.A.

References

[AG00] APODACA A. A., GRITZ L.: *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000.

[AMHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-time rendering*. AK Peters Ltd, 2008.

[AMMH07] AKENINE-MÖLLER T., MUNKBERG J., HASSELGREN J.: Stochastic rasterization using time-continuous triangles. In *Graphics Hardware 2007* (2007), pp. 7–16.

[AMS03] AKENINE-MÖLLER T., STRÖM J.: Graphics for the masses: a hardware rasterization architecture for mobile phones. In *ACM SIGGRAPH 2003 Papers* (2003), ACM, p. 808.

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The reyes image rendering architecture. *Computer Graphics (Proceedings of SIGGRAPH '87) 21*, 4 (1987), 95–102.

[FFB*09] FISHER M., FATAHALIAN K., BOULOS S., AKELEY K., MARK W. R., HANRAHAN P.: DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. In *ACM Transactions on Graphics* (New York, NY, USA, 2009), ACM, pp. 1–10.

[FLB*09] FATAHALIAN K., LUONG E., BOULOS S., AKELEY K., MARK W. R., HANRAHAN P.: Data-parallel rasterization of micropolygons with defocus and motion blur. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 59–68.

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical z-buffer visibility. In *Proceedings of SIGGRAPH 1993* (New York, NY, USA, 1993), ACM Press / ACM SIGGRAPH, pp. 231–238.

[Gre95] GREENE N.: *Hierarchical rendering of complex environments*. PhD thesis, University of California at Santa Cruz, 1995.

[HAM06] HASSELGREN J., AKENINE-MÖLLER T.: Efficient depth buffer compression. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2006), ACM, p. 110.

[HMAM09] HASSELGREN J., MUNKBERG J., AKENINE-MÖLLER T.: Automatic pre-tessellation culling. *ACM Transactions on Graphics (TOG) 28*, 2 (2009), 19.

[Mor00] MOREIN S.: ATI Radeon HyperZ Technology. In *Graphics Hardware 2000* (2000).

[NVI08] NVIDIA: GPU Programming Guide, 2008. http://developer.nvidia.com/object/gpu_programming_guide.html.

[RKLC*10] RAGAN-KELLEY J., LEHTINEN J., CHEN J., DOGGETT M., DURAND F.: Decoupled sampling for real-time graphics pipelines. *MIT Computer Science and Artificial Intelligence Laboratory Technical Report Series, MIT-CSAIL-TR-2010-015* (2010).

[SS89] SALESIN D., STOLFI J.: The ZZ-buffer: A simple and efficient rendering algorithm with reliable antialiasing. In *Proceedings of the PIXIM 89* (1989), pp. 451–466.

[WGER05] WEXLER D., GRITZ L., ENDERTON E., RICE J.: GPU-accelerated high-quality hidden surface removal. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), ACM, pp. 7–14.