Frequency-Based Texture Caching

Oscar Dadfar
Department of Computer Science
Carnegie Mellon University
odadfar@andrew.cmu.edu

Andrew Yang
Department of Computer Science
Carnegie Mellon University
atyang@andrew.cmu.edu

GitHub Repo: https://github.com/cardadfar/740-project

Abstract—**Most modern-day real-time rasterization pipelines are memory-bound due to high-latency texture lookups in the fragment shading stage of the rasterization pipeline. While there exists a dedicated multi-level hardware cache for textures, the overall latency for the rasterization is bottlenecked from cache misses that result in searching main memory for the few pixel values corresponding to a large texture image. Research has been done on being able to predict future memory accesses from past access patterns to help with cache hit rates, but during rasterization we know all the textures referenced by each primitive, and thus the total number of accesses per texture over the scene's rasterization lifetime. This data can be used to help predict how likely the same texture will be accessed in the near future, which is essential in determining whether to keep certain texture data in the cache. We propose a new cache eviction policy that combines a baseline LRU ranking with texture access frequency to compute an eviction score that evicts textures from the cache that have not been recently used and have a low probability of being accessed in the future. We evaluate our results for multiple cache configurations across different scenes with varying primitive and texture quantities and find that our algorithm does increasingly better than pure LRU for higher cache associativities.**

I. Introduction

Fast rasterization techniques are necessary to support the real-time rendering required by fast-paced video, AR, and game graphics. A major bottleneck in the rasterization pipeline is texture reads, comprising more than a third of overall pipeline latency. Texture reads primarily happen in the fragment shading stage where screen-space coordinates are converted to texture coordinates in order to sample texture colors and populate a pixel output buffer.

In fragment shading, more time is spent on reading texture values than on executing instructions. Dedicated texture cache hardware and texture compression have been proposed, but the biggest speedup to the pipeline comes from cache-eviction policies that maximize low-latency texture reads. Storing entire images in a cache is infeasible given they are orders of magnitude bigger than L1 and even L2 caches. Rather, we can partition textures into smaller blocks to make them cache-friendly and promote spatial locality.

Most caches use temporal-based eviction such as LRU or attempt to learn an optimal policy from cache access patterns. Knowing ahead of time when a program needs data significantly helps in deciding what to evict in case of a conflict miss. While general programs tend to be unaware of access patterns prior to run time, rasterization is a bit different. When loading a scene, rasterizers know the specific vertex coordinates, triangulated mesh, and texture references corresponding to each primitive. That is, we know on-demand how frequently each texture will be accessed based on how many primitives reference it.

We introduce a new eviction policy that scores cached texture blocks using a combination of texture access frequency (which we can pre-compute) and LRU to increase hit rates during rasterization. We retrieve a mapping of texture ID to reference frequency while parsing the initial scene file and can then use these values in the texture cache to assign eviction scores, finding that this new eviction policy performs better than pure LRU as we increase cache associativity.

II. Related Works

Texture caching has been an important field of study in the rasterization pipeline given its high temporal and energy consumption compared to other stages of the pipeline [1]. One of the first suggestions for dedicated texture hardware focused on the representations of textures in memory and the impact of spatial locality [2]. Many cache access patterns such as spatial locality have their own variant in the graphics domain, such as texture-blocking. In fragment shading, anti-aliasing techniques such as bilinear and anisotropic interpolation utilize a neighborhood of pixels averaged together to return a color value. It is very common for texturing to return 2×2 samples of adjacent pixels to average together to sample a color, so a simple row-major blocking strategy would fail in this case. Texture blocking solves this problem by blocking each group of $N \times N$ pixels into a cache line to improve spatial locality of the block-like access patterns of texturing which we use when building our own texture cache simulator [3].The same spatial locality can also be used for texture prefetching, where nearby textures can be sent to a FIFO queue to prefetch nearby pixels based on current pixel access patterns [4].

Large texture blocks can also be compressed down into smaller blocks when being read into the cache [5]. Direct3D supports multiple variants of block compression that reduce the memory footprint per block by reducing precision or interpolating via nearby colors [6]. Such compression strategies were motivated by the encoder-decoder dynamic commonly used in video-compression algorithms such as H.264 [7].

With the emergence of multicore architectures, rasterization was quick to adopt parallel texture caching with a shared L2

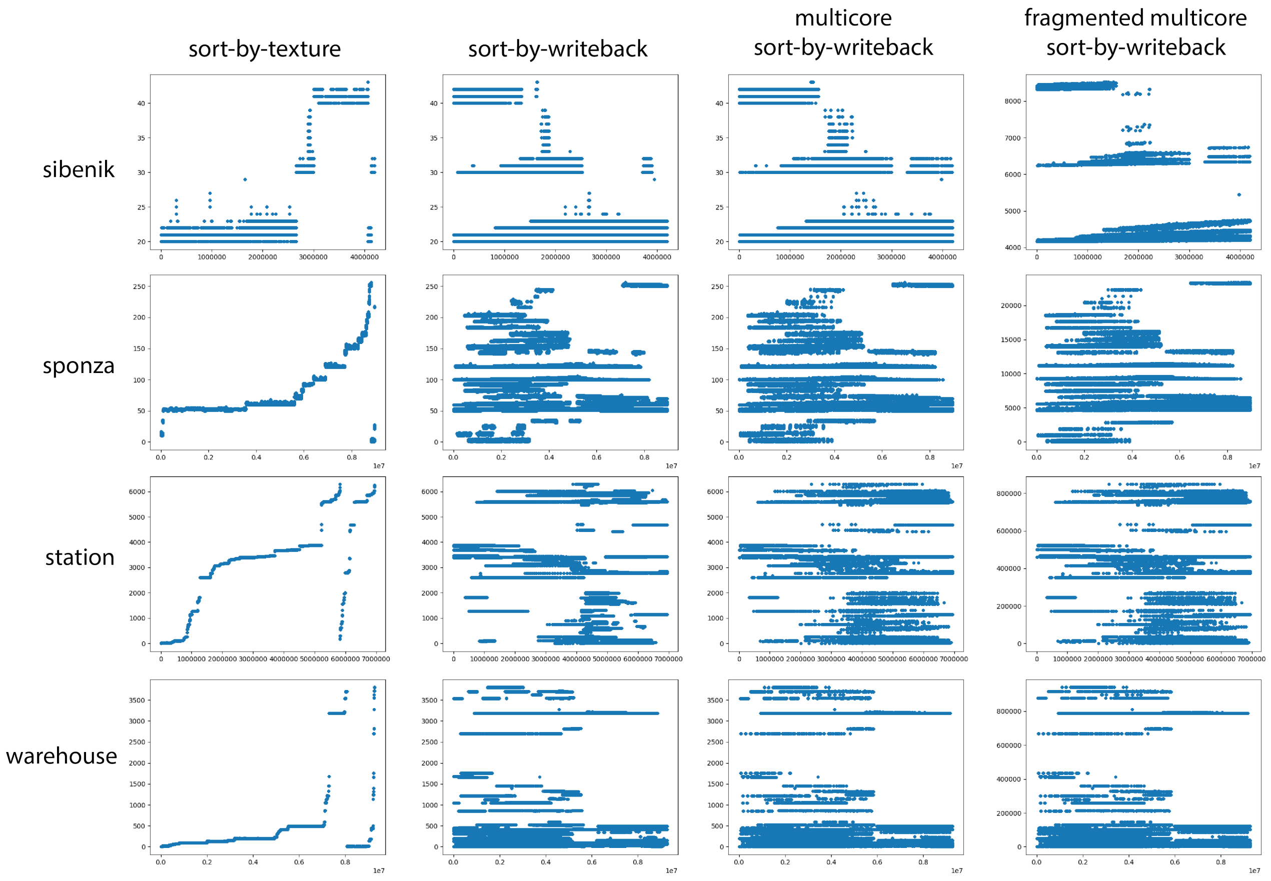

Fig. 1. Texture access plots for each scene. x-axis is the timestep in the program the texture is accessed and y-access is the textureID accessed. Long lines in the plot demonstrate the lifetime a texture is accessed for, and short lines show the brief period a texture is used. sort-by-texture orders all texture accesses by ID, sort-by-writeback sorts all texture accesses by the screenspace writeback coordinate, multicore sort-by-writeback is the same as sort-by-writeback but in the 8 thread (shared cache) case, and fragmented multicore sort-by-writeback fragments textures into more fine, grain textures.

texture cache [8]. Developing texture caching techniques for parallel systems is increasingly difficult since multiple requests for different texture accesses are interleaved, harming spatial locality. Texture requests can be queued and serviced periodically for nearby or overlapping requests using a requests queue in order to reduce the amount of memory reads [4]. We look into how our proposed eviction algorithm performs for shared/parallel caching domain.

III. Approach

We integrate texture-access frequency into our cache to design a cache-eviction policy to help better predict how often a texture block will be re-referenced in the future. We break our pipeline into two sections, rasterization and texture cache simulator, and we describe how the two components interface with one another. We evaluate our algorithm on several scenes with varying primitive and texture counts seen in Fig. 2.

A. Rasterization Pipeline

Our rasterization pipeline follows the standard graphics-library pipeline [9] built from a tiled-based renderer [10], thought we will focus on the high-latency fragment shading stage. Upon loading a scene, we obtain a vector of every primitive (triangle) as well as a reference to what texture the primitive is to be shaded with. We can perform our pre-computation step by iterating over all primitives and counting the references to each texture. While this may add some additional overhead to the rasterization setup stage, we can delegate this process to the application that created the scene, saving the texture reference counts within the scene file. The resulting texture frequency metadata is stored as a map from texture ID to frequency count and sent over to the texture cache simulator with other initialization parameters.

Most rasterization pipelines require support for mipmap level rendering [11]. These mipmap textures store downsampled representations of these texture as their own texture, which helps counter aliasing artifacts for zoomed out/high-frequency scenes by sampling textures from higher-level mipmaps. To alleviate complications for various level references of the same texture, our program makes the distinction of referencing each mipmap level as its own independent texture. We otherwise leave the texture IDs unchanged.

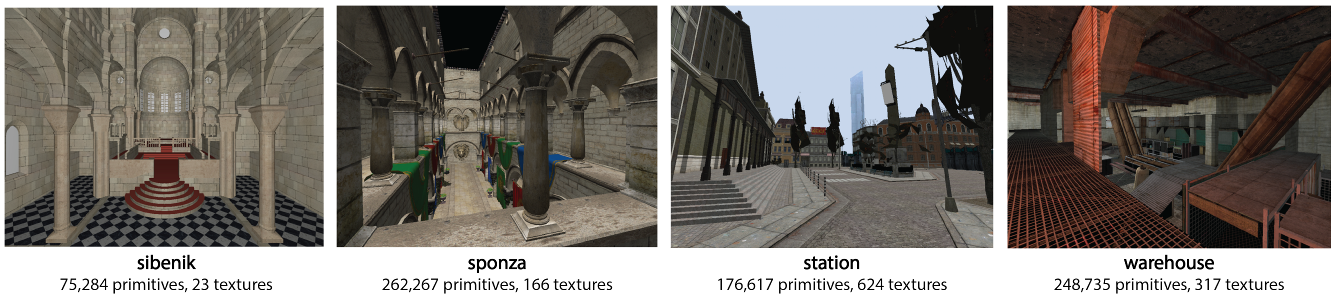
sibenik
75,284 primitives, 23 textures
sponza
262,267 primitives, 166 textures
station
176,617 primitives, 624 textures
warehouse
248,735 primitives, 317 textures

Fig. 2. Scenes rasterized with different primitives/texture counts.

1) Trace Generation: We generate texture access traces for our cache simulator via print statements. Our fragment shader iterates through each primitive and processes fragments in 2x2 pixel batches using SIMD intrinsics. For each fragment, we compute the texture lookup location and retrieve the corresponding texture pixels. Upon lookup, we print the texture ID, texture width/height, and texture (x,y) coordinates for our trace.

Different sampling methods can be used to resolve aliasing artifacts from incorrect or low precision sampling, such as jagged lines along high-frequency edges. Higher-order sampling methods yield clearer results but require more texture lookups from various texture levels. We use anistropic texture sampling which requires 2 quad texture lookups, where a quad texture is a 2x2 contiguous region of pixel memory, thus leading to 8 memory references across 2 textures. Using anistropic sampling makes out trace file twice as large, but provides us with more spatial locality since we will be referencing the same two levels of a texture for each primitive.

2) Texture Access Patterns: A common rasterization technique is sorting primitives by texture and rasterizing all primitives of the same texture first so that the cache will continue to contain data from the same texture until all primitives that need that texture have finished rasterizing, and we move onto primitives of the next texture. This harms write locality since we have to jump around the output buffer when rasterizing primitives sorted on texture ID. Instead, we can improve write locality by sorting primitives based on screen-space (left-to-right, top-to-bottom). Reordering textures based on this premise harms texture read locality since we now need to support multiple textures rasterizing at or near the same time. We expect worse LRU miss rates from this approach due to higher thrashing rates from multiple textures as seen by the sort-by-writeback access patterns in Fig 1., which motivates us to integrate a frequency-based policy for eviction to only keep highly-referenced textures in the cache when excessive thrashing occurs.

We run our traces for a sequential implementation and for a multithreaded implementation using 8 threads. Our multithreaded code rasterizes in a dynamically-scheduled interleaved fashion, where a chunk of rows is partitioned to each thread, and a thread picks up additional blocks of rows once they finish their current work. The multithreaded implementation, as we expect, produces additional thrashing due to each thread working on their own set of texture rasterizaions as seen in multicore sort-by-writeback in Fig 1. This beckons the need for a policy better than LRU in shared cache space where thrashing is more likely to be found.

Another approach to trace generation was where we did an additional pre-processing step to fragment textures into smaller textures before computing texture access frequency. This approach involved first computing the frequency of accesses for each texture, and if a texture had a high amount of accesses, it would be partitioned into 4 (2x2) or 16 (4x4) equally-sized smaller textures, where the texture frequencies would be recomputed. This helped give us a better sense in the traces if there was a specific region in each texture that would be accessed more than others, and lead to a sparser trace file as a result. For example, if we had a texture that had a high frequency of access, but it was the case that only the bottom right of the texture was accessed heavily, then we should not rely on caching texture accesses from the top right if they are infrequent. Fragmenting the texture would help identify circumstances such as this and assign low frequency accesses to ares of the texture that are not accessed frequently, thus allowing them to evict them earlier on. The access patterns of fragmented multicore sort-by-writeback is shown in Fig 1 where the pattern looks generally the same, but references are spread out over more textures, giving a finer-grain texture access frequency.

B. Simulated Texture Cache

Our texture cache is set-associative with n sets and m ways per set, where n and m are configurable parameters. Integral to our design are texture blocks, translating texture blocks to cache tags, computing set indices from cache tags, and our weighted LRU-frequency eviction policy.

1) Texture Blocks: Texture blocks are rectangular partitions of w pixels by h pixels in a texture. The first (0-indexed) block begins at the top left of the texture, i.e. pixel $(0, 0)$, and blocks are indexed in row-major order. Figure 4 shows an example of blocks in a texture with block indices. Note that textures with sizes that are not even multiples of the block size will have empty regions in the right-most and/or bottom-most blocks.

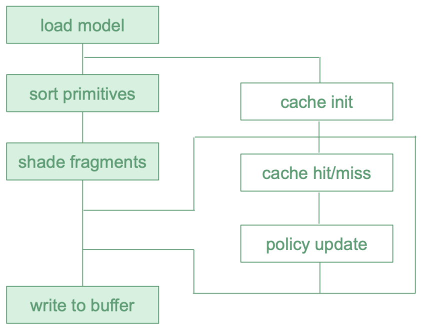

Fig. 3. Green cells refer to the rasterization pipeline, white cells refer to the cache pipeline, with wires illustrating how they interface with each other. After precomputing the texture frequency metadata, we initiate the cache with this metadata along with other cache specs. When our fragment shader requests data from the cache, we return the data requested and update the scores of each cache block based on the LRU ranking and texture frequency of the block, evicting any block with the lowest score on a cache miss.

2) Tag Translation: The cache translates texture block accesses into cache tags. It keeps a running sum (initialized to 0) of the total number of blocks from each unique texture. Every time the cache encounters a new texture, it first adds the previous sum to the texture's block indices to obtain a cache tag for each block, then it adds the texture's block count to the sum. This linear index translation ensures that no two blocks receive the same cache tag.

3) Hashed Set Index: The index of the set in which a tag lies is computed as follows. First, we hash the tag (an unsigned 32-bit integer) using the following function [12]:

Listing 1
INTEGER HASH FUNCTION

```
unsigned int hash(unsigned int x) {
    x = ((x >> 16) ^ x) * 0x45d9f3b;
    x = ((x >> 16) ^ x) * 0x45d9f3b;
    x = (x >> 16) ^ x;
    return x;
}
```

According to Mueller, "each input bit affects each output bit with about 50% probability" [12]. Finally, we take the hashed value modulo n to get the set index of the tag. The intent of the hash is to keep the probability of collision low and relatively uniform between any pair of unique tags.

4) Remaining Frequency: For each block, we keep an estimate of how many times it will still be accessed in the future, which we call *remaining frequency*. The cache takes as input total access counts for each texture, and to initialize the remaining frequency counts, we heuristically divide the total access counts evenly among each block. Each time a block is accessed, we decrease its remaining frequency by one.

5) Eviction Policy: We assign each block in a set a score based on LRU rank r and remaining frequency f. LRU rank ranges from 0 for least-recently used to $m-1$ for most-recently used. We define the score as follows, with adjustable weight factor w:

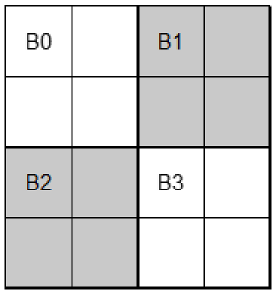

Fig. 4. 2×2 blocks in a 4×4 texture

$$\mathsf{score}(r, f) = r + w \cdot \ln(\max(f, 1))$$

When a new block is accessed, we compute its score using $r = m$ and compare it against the other blocks in the appropriate set. If the new block scores higher than the lowest-scoring block in the set, then it replaces that block. Setting $w = 0$ gives a pure LRU policy, and increasing w gives remaining frequency more influence. We use the natural logarithm because raw access counts could be extremely large.

With higher weights, our eviction policy is more inclined to keep texture blocks that will be used more in the future and evict texture blocks that will be used less. We hope this feature will synergize with LRU to reduce miss rates

IV. RESULTS

To keep the space of possible configurations within a reasonable size, we conducted two experiments in sequence: one to choose set associativity and another to compare our weighted eviction policy against texture fragmentation.

Our first experiment investigated the effect of associativity on miss rates for a pure LRU eviction policy. We fixed $(w, h) = (32, 32)$, the product $n \cdot m = 256$, and therefore the total cache size. We chose the associativity $m \in \{64, 32, 16, 8, 4\}$, declining the set size apropriately as we increased the associativity. For each configuration and scene, we ran the sort-by-writeback cases of *single core*, *multicore*, and *fragmented multicore* through our cache simulator. Table I shows our results for this experiment.

Our second experiment took the two best configurations from the first experiment (associativity $m \in \{64, 32\}$) and investigated the effects of our weighted eviction policy and texture fragmentation on miss rates. We chose the weight factor $w \in [0...6]$, and we ran the same scenes and cases through our simulator. Table II and Table III show our results for this experiment. Weight $w = 0$ is identical to the pure LRU policy, and increasing the weight increases the significance of texture frequency in the score. We outline the lowest hit rate in each column for Table II and Table III and find that our novel policy is slightly able to beat the pure LRU policy for certain weights.

TABLE I
32 × 32 BLOCK SIZE; PURE LRU

		sibenik			**sponza**			**station**			**warehouse**		
		Single	**Multi**	**Frag.**	**Single**	**Multi**	**Frag.**	**Single**	**Multi**	**Frag.**	**Single**	**Multi**	**Frag.**
Sets	***Ways***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***
4	64	0.0786	0.4226	0.3168	1.1584	1.2237	1.2241	0.4033	0.9779	1.2423	1.0023	1.0709	1.1013
8	32	0.0789	0.4243	0.3246	1.1648	1.2306	1.2250	0.4184	0.9806	1.2439	0.9955	1.0712	1.1032
16	16	0.0804	0.4443	0.3496	1.1730	1.2433	1.2341	0.4480	0.9832	1.2474	0.9994	1.0797	1.1086
32	8	0.0956	0.5079	0.3930	1.1952	1.2677	1.2526	0.5120	1.1007	1.2574	1.0077	1.1072	1.1274
64	4	0.1712	0.6341	0.5069	1.2141	1.3906	1.3917	0.5875	1.1066	1.3224	1.0282	1.2563	1.2381

TABLE II
4 SETS, 64 WAYS, 32 × 32 BLOCK SIZE; WEIGHTED LRU + FREQUENCY

	sibenik			**sponza**			**station**			**warehouse**		
Freq.	**Single**	**Multi**	**Frag.**	**Single**	**Multi**	**Frag.**	**Single**	**Multi**	**Frag.**	**Single**	**Multi**	**Frag.**
Weight	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***
0.0	0.0786	0.4226	$\boxed{0.3168}$	1.1584	1.2237	$\boxed{1.2241}$	0.4033	0.9779	$\boxed{1.2423}$	1.0023	1.0709	$\boxed{1.1013}$
1.0	0.0771	0.4149	0.4303	1.1198	1.1946	1.2603	$\boxed{0.3874}$	0.9643	1.2695	0.9829	1.0607	1.1434
2.0	0.0756	0.4110	0.6427	$\boxed{1.1073}$	$\boxed{1.1814}$	1.3346	0.3946	0.9533	1.3103	0.9642	1.0516	1.2355
3.0	0.0744	0.4078	0.7642	1.1116	1.1843	1.4450	0.4234	$\boxed{0.9495}$	1.3496	0.9562	$\boxed{1.0483}$	1.3734
4.0	0.0740	0.4083	0.7766	1.1231	1.2171	1.6471	0.4823	0.9558	1.3975	0.9483	1.0586	1.7855
5.0	0.0739	$\boxed{0.4068}$	0.7768	1.1264	1.3425	2.2094	0.5399	0.9805	1.4600	$\boxed{0.9419}$	1.0993	4.2838
6.0	$\boxed{0.0735}$	0.4142	0.7768	1.1278	1.8737	7.8442	0.5701	1.0845	1.5597	0.9556	1.3400	46.309

TABLE III
8 SETS, 32 WAYS, 32 × 32 BLOCK SIZE; WEIGHTED LRU + FREQUENCY

	sibenik			**sponza**			**station**			**warehouse**		
Freq.	**Single**	**Multi**	**Frag.**	**Single**	**Multi**	**Frag.**	**Single**	**Multi**	**Frag.**	**Single**	**Multi**	**Frag.**
Weight	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***	***Miss%***
0.0	0.0789	0.4243	$\boxed{0.3246}$	1.1648	1.2305	$\boxed{1.2250}$	0.4183	0.9805	$\boxed{1.2439}$	0.9954	1.0712	$\boxed{1.1032}$
1.0	0.0767	0.4144	0.6080	1.1140	$\boxed{1.1942}$	1.3218	$\boxed{0.3973}$	0.9587	1.3082	0.9659	$\boxed{1.0564}$	1.2207
2.0	0.0748	$\boxed{0.4098}$	0.7844	$\boxed{1.1065}$	1.2267	1.6174	0.4546	$\boxed{0.9586}$	1.3942	$\boxed{0.9427}$	1.0609	1.7686
3.0	$\boxed{0.0748}$	0.4111	0.7848	1.1166	1.7361	7.1641	0.5384	1.0634	1.7267	0.9564	1.2406	35.143
4.0	0.2647	0.6436	0.7918	11.067	18.532	78.493	0.8040	5.4041	4.4791	3.7970	4.5836	68.462
5.0	0.2731	0.6647	0.7918	23.637	30.769	85.922	6.9107	15.874	8.0594	4.2904	4.9034	75.428
6.0	0.2812	0.6768	0.7918	26.315	32.146	87.746	9.6894	17.570	11.349	4.8209	5.4590	78.586

V. DISCUSSION

A. *Locality*

Our cache exploits spatial locality through the use of texture blocks. It also benefits from temporal locality in two ways. First, because the renderer writes pixels in row-major order but our cache creates two-dimensional texture blocks, blocks will be reused between adjacent rows. Second, textures of a single type that appear frequently in the scene tend to appear close to each other. With our best configurations, the "easiest" pure LRU cases produced miss rates as low as 0.0786%, and the "hardest" pure LRU cases produced miss rates of at most 1.1013%. Our LRU + frequency policy also benefits from the same low miss rates, demonstrating that most of the performance gain was already present from locality.

B. *Multithreading*

Multithreading interleaves texture accesses between cores, which hurts spatial locality and thrashes our cache. Compared to the sequential, *single core* cases, the corresponding *multicore* cases always performed worse in Tables I, II and III. However, the difference in performance was much greater for the *sibenik* and *station* scenes than it was for the *sponza* and *warehouse* scenes. We predict this may be because *sibenik* and *station* have the least amount of primitives, so there isn't as much texture data reuse as there is with high-primitive scenes that make a lot of references to textures.

C. *Associativity*

The highest associativity ($m = 64$) gives the lowest miss rate in all except one case (*warehouse*, *single core*) for pure LRU. We also see that $m = 64$ gives a lower miss rate than $m = 32$ for our weighted eviction policy, suggesting higher associativity is also better when we consider remaining frequency. As we demonstrated, increasing set associativity increases the influence of policies which allow the cache to make better eviction choices.

D. Fragmentation

Texture fragmentation has a mixed effect on miss rates for pure LRU, as *multicore* performs better than *fragmented multicore* on each scene except *sibenik*. Fragmentation by the renderer may have some anti-synergy with our cache-side texture blocks, but perhaps with fewer textures (such as for *sibenik*), it does provide some benefit.

E. Frequency

For the *single core* and *multicore* cases, our weighted eviction policy with $0 < w \leq 4$ produced slightly lower miss rates than pure LRU ($w = 0$). Looking at just the *multicore* cases, the best-performing weights were 3.0 for *sibenik*, 2.0 for *sponza*, 3.0 for *station*, and 3.0 for *warehouse*. In these cases, it seems giving some preference towards texture blocks that will be accessed more often creates a better eviction policy. Since these scenes contain textures that are accessed over the scenes lifetime (shown in Fig 1 access plots), we prioritize saving these high-access textures in the cache since it is very probably based on their access frequency that they will be re-accessed in the future. However, giving frequency too much weight eventually makes the policy worse than pure LRU, as we do not provide flexibility for other textures with less frequency accesses even if they are to be used in the near future. Thus, it is important to balance the concept of locality which LRU exploits, with access frequency, which our frequency policy uses.

Combining our frequency-based policy with fragmentation fails for all weights, with pure LRU working the best in the fragmentation case. This may be because our frequency-based policy more greedily throws out other textures when there are more textures to keep track of. With fragmentation, we broke textures into smaller pieces and generated more access-frequency metadata for these smaller texture fragments. Yet as we see in Table II and III, our miss rates for higher frequency weights reach miss rates of up to 46.309% and 78.586%, showing that the same fragments are being held onto even though they are not needed simply because they have a high access frequency in the future.

We also see in Table II and III that the optimal set of frequency weights decrease as we decrease our associative from $m = 64$ to $m = 32$. We extrapolate that running our policy on lower-associativity caches may show that LRU is stronger, and for higher associativities our frequency polocy is stronger. We predict this is because with lower associativity, evictions are going to happen more often, so we do not want to hold onto data we may use in the long-term from our frequency-based policy when all we care about is prioritizing what gets accessed in the near future, which is governed by locality (LRU does this really well). As we increase associativity, we have more room to predict for blocks that can be in the father future, and so having a frequency-based policy like ours that maintains more long-term data ends up performing better than LRU for higher associativities by merging the locality benefits of LRU with the future-access predictions of texture access frequency.

VI. Future Work

The current texture metadata we pass to the cache on startup only supplies the texture access frequency per texture at a primitive level. In reality, each primitive will access its texture several times, and the number of texture accesses will vary between primitives. Specifically, there is a correlation between primitive area and the number of texture reads for that primitive. Future work could explore an eviction policy that does not rely on the number of primitives that access a texture, but rather the summed area of the primitives that access it. It may be the case that a texture with a high frequency access count may only be so for a large amount of tiny primitives with a few texture accesses each, but a texture with low frequency access count may be for a few very large primitives with thousands of texture accesses each, in which case we would want to prioritize data from the few large primitives due to spatial locality. Our test scenes had roughly the same primitive sizes, but for scenes with high primitive size variance, this method of using texture frequency based on the sum of primitive area would be a more reliable metric of texture access approximation. The only downside is that this would add extra precomputation and operations per primitive, but this precomputation could be done on demand as we add primitives into our scene, making the cost constant amortized.

VII. Conclusion

We present a novel cache eviction algorithm for the texture cache during rasterization that takes into account the frequency of texture accesses in order to help predict what textures in the cache will be needed in the future, and thus kept in the cache. Our policy mixes LRU rank with frequency ranking that scales by the log of texture frequency in order to compute a score that can be used for eviction, where we evict the lowest score. We compare this score-based policy to pure LRU for associativities between $[4 - 64]$ and find that we are able to improve using our novel frequency-based policy for larger associativites. Future work could look at incorporating primitve area for a more accurate estimate of the number of texture lookups per texture during the rasterization pipeline. Nevertheless, our results show that with a little extra preprocessing, we can lower high-latency miss rates in the texture cache during the fragment shading state of the pipeline.

References

[1] Jeff Pool, ChungAnselmo Lastra, and Montek Singh. A per-unit breakdown of the energy consumption in a graphics processing unit. 2010.

[2] Z. S. Hakura and Anoop Gupta. The design and analysis of a cache architecture for texture mapping. *ISCA*, 1997.

[3] Jhe-Yu Liou and Chung-Ho Chen. Re-visit blocking texture cache design for modern gpu. pages 288–289, 2014.

[4] H. Igehy et al. Prefetching in a texture cache architecture. *IEEE International Conference on Computer Design*, 1998.

[5] T. Olson et al. Adaptive scalable texture compression. *High Performance Graphics*, 2012.

[6] MSDN Developer Reference. Block compression (direct3d 10). 2003.

[7] Gary J. Sullivan Thomas Wiegand. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 2003.

[8] H. Igehy et al. Parallel texture caching. *IEEE International Conference on Computer Design*, 2001.

[9] Khrono Wiki. Rendering pipeline overview.

[10] Kayvon Fatahalian. Implementing a parallel sort-middle tiled renderer. *Carnegie Mellon University*, 2014.

[11] Lance Williams. Pyramidal parametrics. *ACM*, 1983.

[12] Thomas Mueller (https://stackoverflow.com/users/382763/thomas mueller). What integer hash function are good that accepts an integer hash key? Stack Overflow. URL:https://stackoverflow.com/a/12996028 (version: 2018-08-02).