Post-Tessellation Geometry Caches

Rahul Sathe, Tim Foley and Marco Salvi

Intel Corporation

Figure 1: *Left: A frame from Heaven 2.0 running in our simulator. Center: Redundant shading work (red) occurs along shared patch edges. Right: Savings from an eight-entry edge cache (green).*

Abstract

Current 3D rendering architectures support adaptive tessellation of patches, allowing for increased geometric detail. Patches are specified independently, giving the implementation freedom to exploit parallel execution. However, this independence leads to redundant shading computations at patch corners and along edges. In this paper, we present post-tessellation geometry caches, the edge cache and corner cache, that can reduce redundant shading along patch edges and corners, respectively. We demonstrate the two caches in a software-simulated D3D11 rendering pipeline, and show that for current tessellation workloads our approach saves up to 37% of post-tessellation vertex shading using caches with as few as 8 entries.

1. Introduction

To provide high performance, GPU architectures rely on processing multiple items (e.g., vertices, fragments, patches) in parallel. However, when there is *sharing* between items, independent processing may lead to duplication of work. In the case of tessellation, processing individual patches in isolation leads to redundant computation along shared boundaries. Taking inspiration from post-transformation vertex caches, we propose to reduce redundant work by introducing caches for post-tessellation vertices at patch corners and along edges, respectively. We present a modification to existing rendering APIs to facilitate use of these caches with only minimal changes to existing workloads. Figure 1 shows how even small 8-entry caches can save up to 37% of post-tessellation vertex processing.

2. Background

The left half of Figure 2 shows the tessellation related stages of the D3D11 pipeline. Tessellation is controlled by two programmable stages, the Hull Shader (HS) and Domain Shader (DS), and a fixed-function Tessellator (TS). While we use D3D11 terminology, similar functionality is also supported by OpenGL 4. The HS processes one patch at a time, taking as input a *control cage* assembled from vertices output by the Vertex Shader. Given a control cage, the HS computes patch *control points*, as well as desired tessellation rates. The TS uses these rates to generate domain locations in either a triangular or quadrilateral *domain*, along with the connectivity for tessellated geometry. The DS is then responsible for computing the properties of each tessellated vertex given the patch control points output by the HS, and a domain location output by the TS (e.g. barycentric coordinates (u,v,w)). D3D11 makes no assumption about the connectivity of, or relationship between the input control cage and output control points. Similarly, no *a priori* correspondence exists between the control points and the parameter domain. A benefit of this approach is that the fixed-function TS is independent of any particular patch representation. However, this complicates our work since there is no way to automatically determine which vertices in the control cage, if any, correspond to the patch corners.

DOI: 10.2312/egsh.20141014

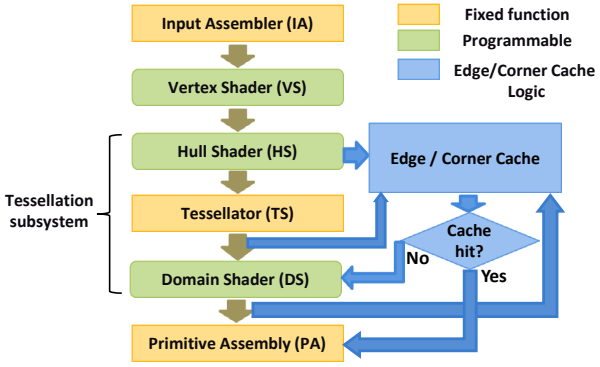


Figure 2: *D3D11 pipeline extended with the edge and corner cache.*

In order to facilitate parallel processing, each patch input to the HS is completely independent. While independent processing of patches enables parallelism, it can also lead to redundant computation. If two subsequent patches share a common edge, the tessellated vertices along this edge are evaluated twice. If several patches share a corner, the tessellated vertex at that corner may be shaded many times. If the duplicate evaluations do not produce bit-identical results, one can see cracks in the geometry.

Prior work provides several examples of how caching can reduce redundant shading work in a rendering pipeline. Our post-tessellation corner cache is similar to the transparent vertex cache of Hoppe [Hop99]. D3D11 and OpenGL 4 allow triangles within a patch to be generated in any order so long as the order is repeatable. Hardware vendors may take advantage of this fact by employing a small domain shader cache akin to vertex cache and control the triangle order within the patch to maximize the hits into this cache. We extend the idea of such a domain shader cache to reduce redundant domain shading across adjacent patches.

3. Algorithm

Figure 2 shows how our caches fit into the D3D11 pipeline. Our system intercepts the corner identifiers generated by the HS as discussed in Section 3.1 and the domain locations output by the TS: either (u,v) or (u,v,w) tuples for quadrilateral or triangle patches, respectively. We identify domain locations that correspond to patch edges or corners by noting when u, v, or w are zero or one. We refer to points along the edges that are not corners as *edge points*. For these *edge points* and *corner points* we generate a unique edge/corner *identifier* and use this as a tag for checking the edge/corner cache, respectively. If we find a hit in the cache, then we can re-use an existing shaded vertex for this domain location, and we skip the DS stage. If we miss in the cache, we shade using the DS and update the cache. Cracks generated by incorrect shaders may disappear at the domain locations where there are cache hits. However due to the finite size

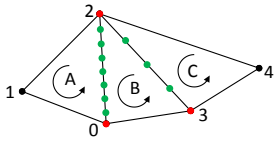


Figure 3: *Triangle patches A, B and C share both corners and edges. Each of the corners is labeled with its user-defined* corner identifier. *Shared corners are shown in red. Edge points along shared edges are shown in green.*

of these caches, there is no guarantee that adjacent patches will arrive close enough in time for the cache hits. Therefore users should not rely on the these caches to produce watertight surfaces that would otherwise show cracks.

3.1. The Corner Cache

We use distinct caches for patch corners and edges, because while an edge is typically shared by only two patches, a corner may be shared by any number of patches. Our corner cache is similar in spirit to the transparent vertex cache [Hop99]. The primary difference is that, as discussed in Section 2, there is no *a priori* relationship between the indices of vertices in a control cage, and the corners of a patch. This relationship gets defined by the programmer in the HS, and in general can vary from one patch to another.

Thus, to allow a programmer to identify shared corners in the most general case, we extend D3D11 so that the HS may output an arbitrary 32-bit *corner identifier* for each patch corner (e.g., three corners for a triangular domain). These identifiers are exposed to HLSL shaders through a semantic: `SV_CornerID[]`. Our system uses these user-provided identifiers directly as tags for the corner cache – that is, we trust the user to correctly identify shared corners. For the example shown in Figure 3, the corner cache stores entries for corners with identifiers 0, 1 and 2 after processing patch A. If we next process patch B, we get cache hits on the shared corners. As a special case, the largest possible identifier marks corners that should not be cached. This allows applications to opt out of corner caching. We use this as a default corner identifier if shaders do not assign one.

One desirable application of tessellation is rendering of

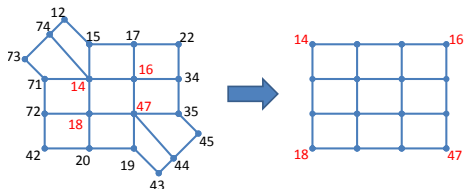


Figure 4: *Basis conversion for approximate subdivision surfaces. Corner identifiers for the patch can be derived from vertex identifiers for the base-mesh face.*

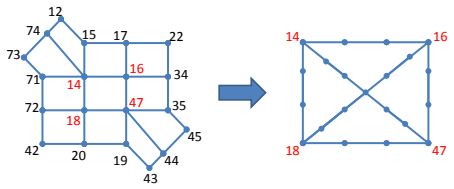


Figure 5: *Basis conversion where one input face yields four triangular patches. The user needs to generate a unique identifier for the corner where the four patches meet.*

subdivision surfaces: either through direct evaluation [Sta98] or through approximation with parametric surfaces [LS08, LSNCn09]. Figure 4 illustrates such an approach, in which the 1-ring neighborhood of a face in the base mesh is used to compute coefficients for a bicubic Bézier patch. In this case, a user can simply assign corner identifiers for the patch based on the system-generated vertex identifiers (HLSL semantic `SV_VertexID`) of the base-mesh face. This same approach is also sufficient for patch representations like PN triangles [VPBM01], that do not require access to a 1-ring neighborhood. Figure 5 shows a more complex scenario in which one base-mesh face is converted to four triangular patches, so that a new corner is introduced [MNP08]. It is the user's responsibility to generate unique corner identifiers for such new corners. In the general case, it is possible to precompute corner identifiers and store them in an auxiliary data structure.

3.2. The Edge Cache

Given user-generated corner identifiers, we can automatically generate suitable identifiers for edges; intuitively, an edge can be identified by the two corners it connects. In practice, we must account for the fact that different patches may orient a shared edge differently, or may assign it different tessellation rates. We thus generate *canonical* edge identifiers in the following form:

```
1  struct EdgeID {
2    unsigned smallerCornerID;
3    unsigned largerCornerID;
4    float tessRate;
5    bool orientation;
6  }; // Size = 97 bits
7  // Corners c1 & c2 and tessFactor t
8  EdgeID GenEdgeId(c1, c2, t) {
9    a = min(c1, c2);
10   b = max(c1, c2);
11   o = (a != c1);
12   if (a or b is kInvalidCornerID)
13     return kInvalidEdgeID; // Do not cache
14   else
15     return EdgeID(a, b, t, o);
16   }
17 }
```

If either corner of an edge is marked as uncacheable, we do not cache the edge. We use this generated edge identifier (except the orientation bit) as a tag to query into the edge cache. On a cache miss, we evaluate the DS for all edge points, and populate an edge cache entry with their vertex attributes (positions, normals, etc.) and the orientation bit for that edge. On a cache hit, we retrieve vertex attributes, making sure that orientation bit is opposite to what is stored in the cache.

4. Implementation

We have implemented our edge and corner caches in a D3D11 software simulation framework (as depicted in Figure 2). In order to allow experimentation with unmodified D3D11 applications, we specify the mapping from control-cage vertices to patch corners on a per-application basis. Our work is primarily characterized by two design choices: the replacement policy for corner and edge entries, and the organization of entries in the edge cache. Now we describe our implementation choices, along with some alternatives.

4.1. Replacement Policy

We start with a FIFO replacement policy similar to Hoppe [Hop99]: when we need a new entry and the cache is full, we evict the least recently *written* one. We extend this policy with one optimization: on a cache hit, we may speculatively evict entries that are unlikely to be used again. A given edge is typically shared by at most two patches (for a manifold mesh). Therefore a hit in the edge cache signals an edge that is likely to not be used again. We take advantage of this observation by speculatively evicting an edge cache entry on a hit, making an additional entry available for other edges. We have found that this approach saves additional 1 to 1.5% of DS invocations. A similar, *erase after N-hits* policy could be applied to the corner cache too, if we count the number of cache hits for a given entry *(N)*. However, such an approach could lead to redundant shading for extraordinary vertices, so we do not employ it in our implementation.

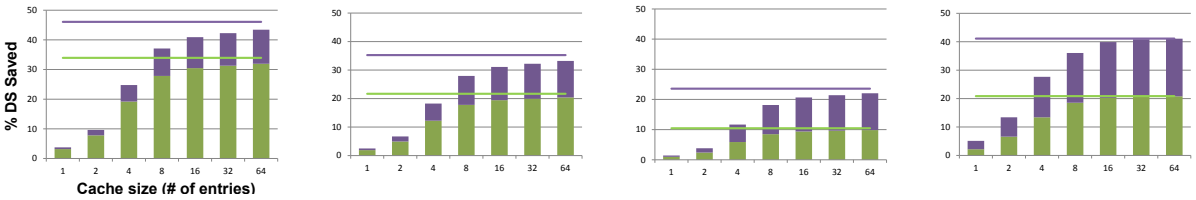
4.2. Edge Cache Organization

Our decision to cache entire edges leads to a complication; edges with different tessellation rates will yield different numbers of edge points. We considered a few different strategies to deal with this situation:

Simple We size each edge cache entry for the worst case. When tessellation rates are low, we end up wasting space.

Threshold We limit each cache entry to store only the first *N* vertices along an edge, and re-shade the remaining vertices even on a cache hit. This strategy is motivated by the observation that the relative benefits of an edge cache decrease at higher tessellation rates, because the number of edge points increases linearly, while the number of interior points increases quadratically.

Indirect If the vertices shaded by the DS are stored in an on-chip buffer, then we can simply store pointers rather



(a) Heaven (moderate) (b) Heaven (normal) (c) Heaven (extreme) (d) Dirt 2

Figure 6: *DS invocations saved due to corner cache (green), and corner and edge cache combined (purple). Savings are expressed as a percentage of all (including internal) DS invocations. Bars correspond to corner/edge caches with a fixed number of entries; the solid lines show results for infinitely-sized caches. An 8- or 16-entry cache achieves close to peak savings.*

than full vertices in edge cache entries. This can be combined with other strategies to reduce the size of cache entries, at the expense of more complex bookkeeping logic.

Flat We append the edge cache tag with the parameter value of a domain point and store only that domain point in that entry. This will cause no wasted space at the expense of higher cache bandwidth and a larger tag.

Our implementation combines the *Threshold* and *Indirect* strategies to minimize the edge cache size. We will discuss the impact of different threshold cutoffs in the next section.

5. Results

We tested our corner and edge caches using two different workloads. We measured the results across multiple batches of 10 frames, separated by 100 frame intervals, in order to get a good sampling of the workload. The Heaven 2.0 demo, seen in Figure 1, uses tessellation for terrain and architectural models. This demo supports three different levels of tessellation: moderate, normal, and extreme. The game Dirt2 uses tessellation for characters and the borders of race tracks. Figure 6 shows that we save between 18% and 37% of DS invocations with corner and edge caches of only 8 entries each. Large caches can achieve higher savings, but the relative benefit falls off quickly; for the Heaven workload, an 8-entry cache achieves 77% of the savings possible with infinitely large caches. If we assume 16-bit pointers are sufficient to point to shaded domain points cached into an on chip memory, 8-entry *Indirect* corner cache requires 8*(16+32) bits plus room for 8 domain shaded points. An 8-entry *Indirect* edge cache with a threshold of 8 would require 8*(8*16 + 97) bits and room for 64 shaded domain points. Overall, both caches combined need 273 bytes and room for 72 shaded domain points.

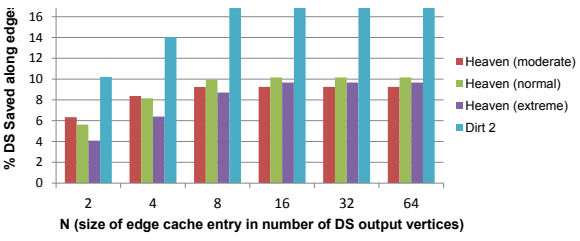


Figure 7: *DS invocation saved by 8-entry edge cache, as a function of threshold. We cache at most N vertices per edge entry, and reshade the rest. No additional benefit for $N > 16$.*

The Heaven benchmark illustrates how increased tessellation levels reduce the effectiveness of the edge and corner cache. In addition, as tessellation rates increase, the savings coming from the edge cache increase relative to those from the corner cache. This effect is independent of cache size, and results from the fact that the number of corner points is constant, the number of edge points increases linearly with tessellation rate. Figure 7 shows the effect of thresholds on the efficiency of an 8-entry edge cache (results are similar for caches with any number of entries). Savings increase as we store more vertices per edge, but plateau at 16 vertices per entry because none of our workloads produce more than 14 edge points.

We have demonstrated an approach for reducing post-tessellation geometry processing with minimal additional on chip storage.

References

[Hop99] HOPPE H.: Optimization of Mesh Locality for Transparent Vertex Caching. In *Proceedings of SIGGRAPH 1999* (1999), pp. 269–276. 2, 3

[LS08] LOOP C., SCHAEFER S.: Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics 27* (March 2008), 8:1–8:11. 3

[LSNCn09] LOOP C., SCHAEFER S., NI T., CASTAÑO I.: Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. *ACM Transactions on Graphics 28*, 5 (Dec. 2009), 151:1–151:9. 3

[MNP08] MYLES A., NI T., PETERS J.: Fast parallel construction of smooth surfaces from meshes with tri/quad/pent facets. In *Symposium on Geometry Processing* (2008), pp. 1365–1372. 3

[Sta98] STAM J.: Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of SIGGRAPH 1998* (1998), pp. 395–404. 3

[VPBM01] VLACHOS A., PETERS J., BOYD C., MITCHELL J. L.: Curved PN triangles. In *Symposium on Interactive 3D graphics* (2001), pp. 159–166. 3