# Rendering Vector Displacement Mapped Surfaces in a GPU Ray Tracer

Takahiro Harada

## 1.1 Introduction

Ray tracing is an elegant solution to render high quality images. By combining Monte Carlo integration with ray tracing, we can solve the rendering equation. However, a disadvantage of using ray tracing is its high computational cost which makes render time long. To improve the performance, GPUs have been used. However, GPU ray tracers typically do not have as many features as CPU ray tracers. Vector displacement mapping is one of the features, which we do not see much in GPU ray tracers. When vector

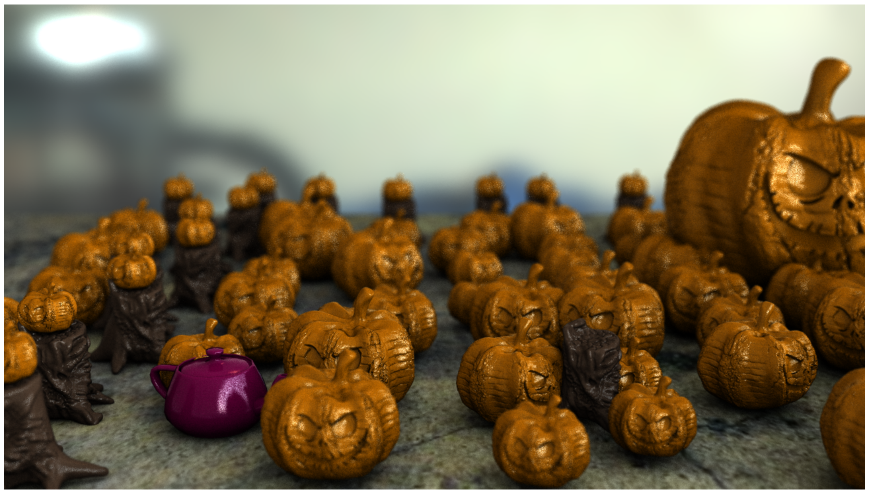


Figure 1.1: The "Party" scene with vector displacement mapped surfaces rendered using the proposed method. The rendering time is 77ms/frame on an AMD FirePro W9100 GPU. Instancing is not used to stress the rendering algorithm. If pre-tessellated, the geometry requires 52GB memory.

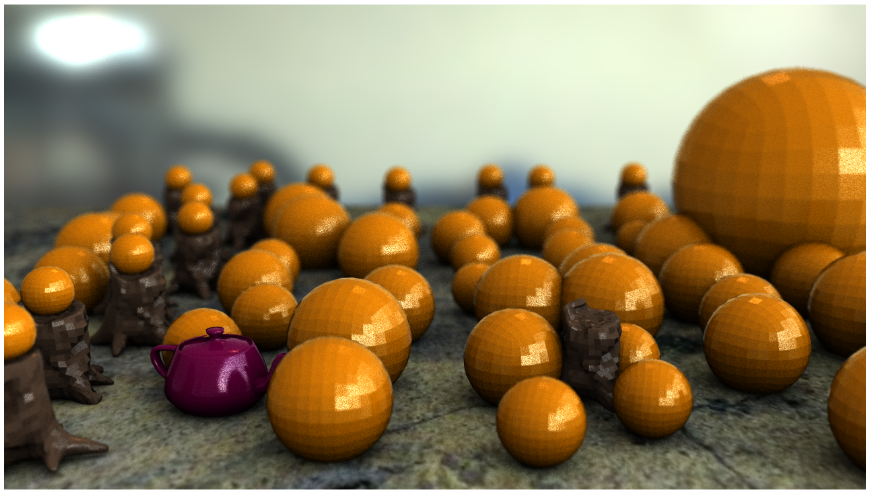


Figure 1.2: The base mesh used for the ”Party” scene.

displacement mapping is evaluated on the fly (i.e., without creating a large number of polygons in the preprocess and storing them in the memory), it allows us to render a highly geometric detailed scene from a simple mesh. Since geometric detail is an important factor for realism, vector displacement mapping is an important technique in ray tracing. In this chapter, we describe a method to render vector displacement mapped surfaces in a GPU ray tracer.

## 1.2 Displacement Mapping

Displacement mapping is a technique to add the geometric detail to a simple geometry. Although the goal is similar to normal mapping, it actually creates high resolution geometries as shown in Fig. 1.1 from low resolution mesh (Fig. 1.2) while normal mapping only changes the normal vector to add an illusion of having a geometric detail. There are two types of displacement mapping. The one we usually call displacement mapping uses textures storing scalar values, which are used as offsets for the displacement using the surface normal as the displacement direction. We call this approach scalar displacement mapping. The other is vector displacement mapping which uses a texture storing vector values, that are used as displacement vector of the surface. Because the displacement can be an arbitrary direction, it gives a lot of freedom for what we create from a simple geometry. For example, scalar displacement mapping cannot create

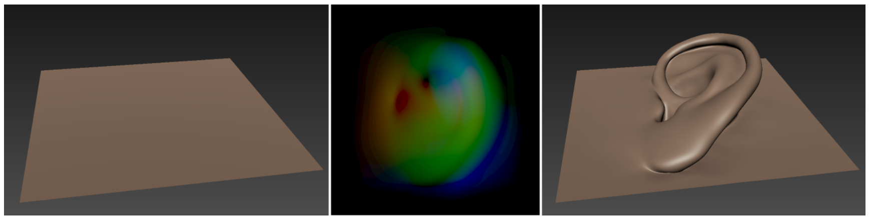


Figure 1.3: Illustration of vector displacement mapping. (a) Simple geometry (a quad). (b) A vector displacement map. (c) Surface after applying vector displacement.

an overhang as shown in Fig. 1.3, but vector displacement mapping can.

This freedom in vector displacement mapping poses technical challenges when it is ray traced. Although we could use algorithms, such as the method proposed by [Smits et al. 00], for ray tracing a scalar displacement mapped surface by utilizing the constraint in the displacement direction, we cannot apply it for a vector displacement mapped surface as the assumption does not apply. In vector displacement mapping, there is no constraint in displacement direction. So when we check the intersection of a ray with a vector displacement patch (VD patch), we cannot avoid creating the detailed geometry by tessellating and displacing vertices and building a spatial acceleration structure for those.

## 1.3 Ray Tracing a Scene with Vector Displacement Maps

Ray tracing requires identifying a closest hit point for a ray with the scene which is accelerated by using a spatial acceleration structure. Bounding volume hierarchies (BVHs) as acceleration structure are often employed. When we implement a ray tracer only for simple primitives such as triangles and quads, we compute the intersection to a primitive once we encounter it during BVH traversal. However, an intersection to a VD patch is much more expensive to compute than an intersection test with these simple primitives, especially when direct ray tracing is used (i.e., a VD patch is tessellated and displaced on the fly). To amortize the cost of tessellation and displacement, we want to gather all the rays intersecting to the AABB of a VD patch and process them at once rather than subdividing and displacing a VD patch every time a ray hits to its AABB as studied by [Hanika et al. 10].

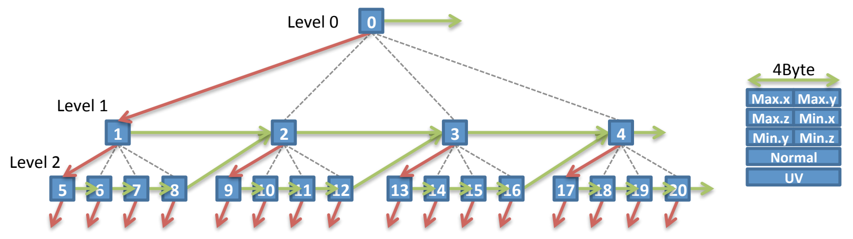


Figure 1.4: Quad BVH. Each node stores two links; one pointing to the first children (red), and one pointing to the skip node (green). To check if a node is a leaf of level$i$ the node index is compared to $(4^i - 1)/3$ e.g., leaf nodes of level 2 BVH are nodes whose index is greater than 5. Data layout in a node is shown on the left.

## 1.4 Ray Tracing a Vector Displacement Patch

This section focuses on the ray-VD patch intersection although using it in a ray tracer requires additional changes which are going to be discussed in Sec. 1.5. In this section, we first describe a single threaded implementation of the intersection of a ray with a VD patch to simplify the explanation. We then extend it for a parallel implementation using OpenCL.

### 1.4.1 Single Ray

To intersect a ray with a VD patch, we first need to build the detailed geometry of the patch by tessellating it to generate vertices which are then displaced by the value fetched from the vector displacement map. Although there are several ways to generate vertices, we simply generate them uniformly on the patch (i.e., all the generated vertices are on the plane of the patch) without geometry smoothing.

Data Structure We could find the closest intersection by testing primitives in the scene one-by-one but it is better to create a spatial acceleration structure to do this efficiently. As we build it on the fly, the build performance is as important as the intersection performance. Therefore, we employed a simple acceleration structure. A patch is split into four patches recursively to build a complete quad BVH. At the lowest level of the BVH, four vertex positions and texture coordinates are linearly interpolated from the values of the root patch. The displaced vertex position is then calculated by adding the displacement vector value which is fetched from a texture using the interpolated texture coordinate. Next, the AABBs enclosing these four vertices are then computed and they are used as the geometry at the leaves instead of using a quad because we subdivide the patch smaller than a pixel size. This allows us not to store geometries (e.g., vertices), but only store the BVH. Thus we can reduce the data size for a VD patch. A texture coordinate and normal vector are also computed and stored within a node. Once leaf nodes are computed, it ascends the tree level-by-level and builds the nodes of the inner level. It does this by computing the union of AABBs and averaging normal vectors and texture coordinates of the four child nodes. This process is repeated until it reaches the root node.

For better performance, the memory footprint for the BVH has to be reduced as much as possible. Thus an AABB is compressed by quantizing the maximum and minimum values into 2 byte integers ($\mathbf{max}_q, \mathbf{min}_q$) these as follows.

$$\begin{aligned}
\mathbf{max}_q &= 0xfff7 \times (\mathbf{max}_f - \mathbf{min}_{root})/\mathbf{extent}_{root} + 1 \quad (1.1)\\
\mathbf{min}_q &= 0xfff7 \times (\mathbf{min}_f - \mathbf{min}_{root})/\mathbf{extent}_{root} \quad (1.2)\\
\mathbf{extent}_{root} &= \mathbf{max}_{root} - \mathbf{min}_{root} \quad (1.3)
\end{aligned}$$

where $\mathbf{max}_f, \mathbf{min}_f$ are uncompressed maximum and minimum values of the AABB and $\mathbf{max}_{root}, \mathbf{min}_{root}$ are values of the root AABB. We considered compressing them into 1 byte integers but the accuracy was not high enough since the subdivision level can easily go higher than the resolution limit of 1 byte integers (i.e., 8 levels). We also quantized texture coordinates and the normal vectors into 4 bytes each. Therefore, the total memory footprint for a node is 20 Bytes (Fig. 1.4).

We separate the hierarchy of the BVH from the node data (i.e., a node does not store links to other nodes such as children). This is to keep the memory footprint for nodes small. We only store one hierarchy data structure for all VD patches because we always create a complete quad BVH so that the hierarchy structure is the same for all the BVHs we construct. Although we build a BVH at different depths (i.e., levels), we only compute and store the hierarchy structure for the maximum level we might build. As nodes are stored in breadth first order, leaf nodes can be identified easily by checking their index. Leaf nodes at $i$th level are nodes with indices larger than $(4^i - 1)/3$ as shown in Fig. 1.4.

We use stackless traversal for BVH traversal. Thus, a node in the hierarchy structure stores two indices of the first child and the skip node (Fig. 1.4). These two indices are packed and stored in a 4 bytes of data.

To summarize the data structure we have:

- DPrecomputed data for the hierarchy structure.
- BVH (array of nodes) built on the fly.

In Listing 1.1, they are denoted as *gNodes* and *gLinks*, respectively.

```
__global Node* gNodes;
__global u32* gLinks;
float f;
u32 n, uv;
int o = getOffset( lodRes );
while( nodeIdx != breakIdx )
{
  Aabb node = NodeGetAabb( gNodes[nodeIdx] ); // reconstruct AABB
  float frac = AabbIntersect( node, &from, &to, &invRay );
  bool isLeaf = nodeIdx >= o;
  if( frac < f )
  {
    if( isLeaf )
    {
      f = frac;
      n = gNodes[nodeIdx].m_n;
      uv = gNodes[nodeIdx].m_uv;
      nodeIdx = LinkGetSkip( gLinks[nodeIdx] );
    }
    else
      nodeIdx = LinkGetChild( gLinks[nodeIdx] );
  }
  else
    nodeIdx = LinkGetSkip( gLinks[nodeIdx] );
}
```

Listing 1.1: Bottom-level hierarchy traversal.

Traversal and Intersection The primary reason we employed a stackless traversal is to reduce the memory traffic and register pressure which affects the performance. Moreover, since the data for the state of the ray is the index of the current node, we could easily shuffle rays to improve the performance although we have not investigated this optimization yet.

As we have already built the BVH for the patch, the traversal is straightforward. Pseudo code is shown in Listing 1.1.

## 1.4.2 OpenCL Implementation

To fully utilize the GPU, we have to parallelize the algorithm described in the previous subsection. We implemented our algorithm using OpenCL and used AMD GPUs and thus follow the respective terminologies in the following explanation.

Before we start intersecting rays with VD patches, we gather all the rays hitting the AABB of any VD patches. When a ray hits multiple VD patches, we store multiple hits. These hits are sorted by a VD patch index. This results in a list of VD patches, each of which has a list of rays.

We implemented a kernel doing both BVH build and its traversal. Work

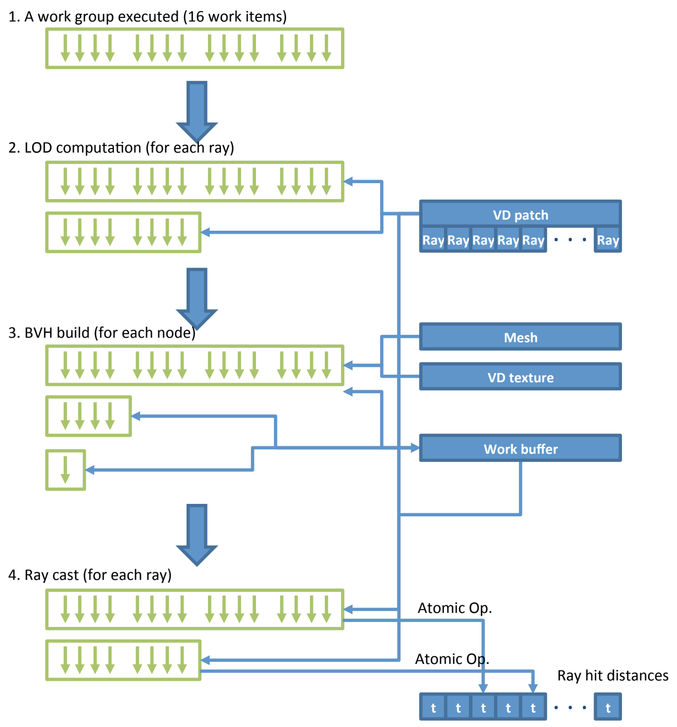

1. A work group executed (16 work items)
2. LOD computation (for each ray)
VD patch
Ray Ray Ray Ray Ray · · · Ray
3. BVH build (for each node)
Mesh
VD texture
Work buffer
4. Ray cast (for each ray)
Atomic Op.
Atomic Op.
Ray hit distances
t t t t t · · · t

Figure 1.5: Overview of the algorithm. In this illustration, the VD patch has 24 rays intersecting the root AABB, it builds a BVH with depth 3.

groups are launched with the number of work items optimal for the respective GPU architecture. We use AMD GPUs which are 64 wide SIMD so 64 work items are executed for a work group. A work group first fetches a VD patch from the list of unprocessed VD patches. This work group is responsible for the intersection of all rays hitting the AABBs of the root patch. First, we use work items executing in parallel for building the BVH. However, as we build a BVH for the patch which has to be stored somewhere, we need to allocate memory for that and therefore the question is where to allocate. The first candidate is in the local data share (LDS), but it is too small if we build a BVH with 6 levels ($64 \times 64$ leaf nodes), which requires 108KB (=5400 nodes x 20B). If we limit the number of levels to 5 ($32 \times 32$ leaf nodes), we only require 26KB. Although this is smaller than the maximum allocation size for the LDS (32KB) for an AMD FirePro W9100 GPU, we can only schedule 2 work groups per compute unit (A compute unit has 4 SIMD engines (SIMD)). Thus it cannot schedule enough work groups for a SIMD to hide latencies which results in poor performance. Instead of storing it in the LDS, we store it in the global memory whose access latency is higher than the LDS, but we do not have such a restriction in the size for the global memory. Since we do not use the LDS for the storage of the BVH data in this approach, the LDS usage is not the limiting factor for concurrent work group execution in a SIMD. The limiting factor is now the usage of vector general purpose registers (VGPRs). Our current implementation allows us to schedule 12 work groups in a compute unit (CU), which is 3 per SIMD as the kernel uses 72 VGPRs per SIMD lane.

Since we know the maximum number of work groups executed concurrently in a CU for this kernel, we can calculate the number of work groups executed in parallel on the GPU. We used an AMD FirePro W9100 GPU, which has 44 CUs. Thus, 528 work groups (44 CUs x 12 work groups) are launched for the kernel. A work group processes VD patches one after another, and executes until no VD patch is left unprocessed. As we know the number of work groups executed, we allocate memory for the BVH storage in global memory before execution and assign each chunk of memory for a work group as a work buffer. In all the test cases, we limit the maximum subdivision level to 5, and thus a 13 MB (= 26KB x 528) work buffer is allocated.

After work groups are launched and a VD patch is fetched, we first compute the required subdivision level for the patch by comparing the extent of the AABB of the root node to the area of a pixel at the distance from the camera. As we allow instancing for shapes with vector displacement maps (e.g., the same patch can be at multiple locations in the world), we need to compute the subdivision level for all the rays. Work items are used to process rays in parallel at this step. Once a subdivision level is computed for a ray, the maximum value is selected using an atomic operation to an LDS value.

Then, work items compute the node data, which is the AABB, texture coordinate, and normal vector of a leaf in parallel. If the number of leaf nodes is higher than the number of work items executed, a work item processes multiple nodes sequentially. Once the leaf level of the BVH is built, it ascends the hierarchy one step and computes nodes at the next level of the hierarchy. Work items are used to compute a node in parallel. Since we write node data to global memory at one level, and then read it at the next level, we need to guarantee that the write and read order is kept. This is enforced by placing a global memory barrier which guarantees the order in a work group only, thus it can be used for this purpose. This process is repeated until it reaches the root of the hierarchy. Pseudo code for the parallel BVH build is shown in Listing 1.2.

Once the hierarchy is built, we switch the work item usage from a work item for a node to a work item for a ray. A work item reads a ray from the list of rays hitting the AABB of the VD patch. A ray is then transformed to the object space of the model and traversed using the hierarchy information. If the current hit is closer than the last found hit, the hit distance, element index, normal vector, and texture coordinate at the hit point are updated. However, we cannot simply write this hit information because a ray can be processed by more than one work item in different work group. The current OpenCL programming model does not have a mechanism to have a critical section, which would be necessary for our case[1]. Instead, we used 64 bit atomic operations which are not optimal in terms of performance, but at least we can avoid the write hazard. When the element index, quantized normal vector, and quantized texture coordinate are all 32 bit data, the hit distance is converted into a 32 bit integer and appended at the top 32 bit to create 64 bit integers. By using an atomic min operation, we can store the closest hit information (Listing 1.1).

Pseudo code for the entire kernel is shown in Alg. 1.

```
while Unprocessed VD patch do
  {Max LOD level computation}
  for rays in parallel do
    $level \leftarrow computeLODLevel(ray_i)$
    $maxLevel \leftarrow max(level)$
  end for
  {Build BVH}
  for leaves in parallel do
    $computeLeafNode(leaf_i)$
  end for
  for $lv = maxLevel - 1$, $lv > 0$ do
    for nodes at level $lv$ in parallel do
      $computeNode(node_i)$
    end for
  end for
  {BVH traversal and Ray VD patch intersection}
  for rays in parallel do
    $level \leftarrow computeLODLevel(ray_i)$
    $hit \leftarrow rayCast(level)$
    $storeHit(ray_i, hit)$
  end for
end while
```

**Algorithm 1:** Bottom-level hierarchy build and traversal kernel

[1]Note that barrier (CLK_GLOBAL_MEM_FENCE) only guarantee synchronization of global memory access from a work group but not for different work groups.

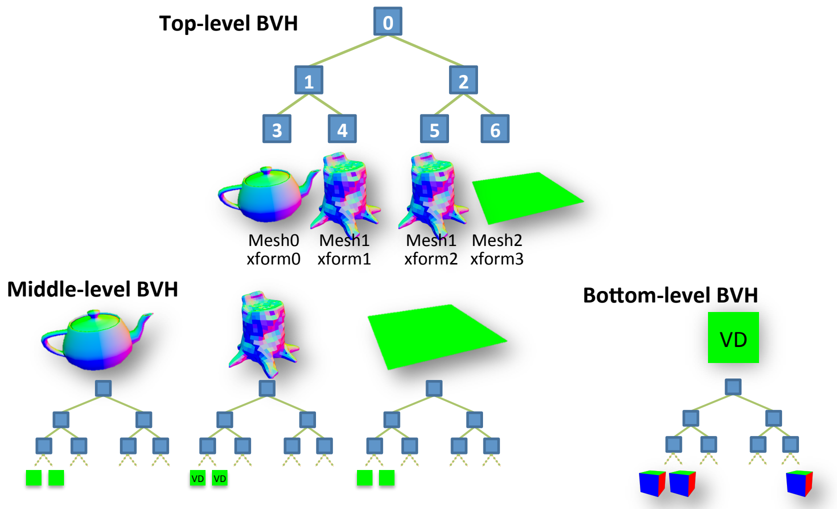


Figure 1.6: Three-level hierarchy. A leaf of the top-level BVH stores an object, which is a middle-level BVH and transform. A leaf of the middle-level BVH stores primitives such as a triangle, a quad, and a VD patch. There is a bottom-level BVH which is built on the fly during the rendering for a leaf storing a VD patch.

## 1.5 Integration into an OpenCL Ray Tracer

Although ray tracing one mesh with a vector displacement map is simple, we want to use several meshes with vector vector displacement maps, together with other triangle meshes as shown in Fig. 1.1. This section describes how the ray tracing of a VD patch is integrated into our OpenCL ray tracer.

### 1.5.1 Scene Description

We could store all the primitives in the scene in a single spatial acceleration structure. However, this does not allow us using techniques such as instancing, which is a powerful method to increase the scene complexity with small overhead. Therefore, we put meshes in the scene and build an acceleration structure storing meshes at leafs. A mesh is a triangle mesh, a quad mesh (some of which might be VD patches), or an instance of one of those with a world transformation. We then build another hierarchy for each mesh in which primitives (e.g., triangles, quads) are stored at leaf

Table 1.1: Memory usage for geometry and acceleration structure.

| Scene | Pre tessellation | Direct ray tracing |
|---|---|---|
| "Party" | 52GB | 16MB |
| "Bark" | 1.7GB | 0.47MB |
| "Barks" | 12GB | 3.3MB |
| "Pumpkin" | 380MB | 0.12MB |

nodes. If a primitive is a VD patch, we build another hierarchy in a patch as we discussed in Sec. 1.4. Therefore, we have a three-level hierarchy. The top and middle stores meshes and primitives and the bottom exists only for a VD patch, which is generated on the fly.

### 1.5.2 Preparation

Before rendering starts, we build top and middle-level BVHs which require the computation of AABBs for primitives. In case for VD patches, the computation of an accurate AABB is expensive as it requires tessellation and displacement. Instead, we compute the maximum displacement amount from a displacement texture and expand the AABB of a quad using the value. Although this results in a loose-fitted AABB, which makes ray tracing less efficient than when tight AABBs are computed, it makes the preparation time short.

### 1.5.3 Hierarchy Traversal

We fused the traversal of top and middle-level hierarchy into a traversal kernel. When a ray reaches a leaf of the top-level hierarchy, the ray is transformed into object space and starts traversing the middle-level hierarchy. Upon exiting the middle-level hierarchy, the ray is transformed back to world space. Once a ray hits a leaf node of the middle-level hierarchy, it computes a hit the primitive stored at the leaf node immediately if the primitive is a triangle or a quad. As discussed in Sec. 1.4, we do not compute the intersection of a ray with the VD patch on a visit to a leaf node. Instead, a primitive index and ray index is stored in a buffer for further processing (i.e., precisely, we also store the mesh index which is necessary to get its transform). An atomic operation is used to allocate space for a pair in the buffer. After the top and middle-level hierarchy traversal, the computed hits are only those computed with triangles and quads. Thus we need to determine if there are closer intersections with VD patches.

The primitive index and ray index are stored in random order. As we

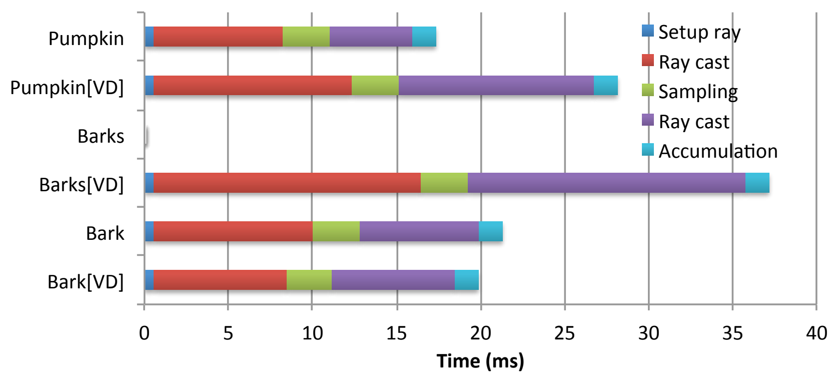


Figure 1.7: Breakdown of computational time for a frame. There are two graphs for each scene. One is with pre-tessellation and the other (VD) is with the proposed method. Barks cannot render without using instancing with VD patches.

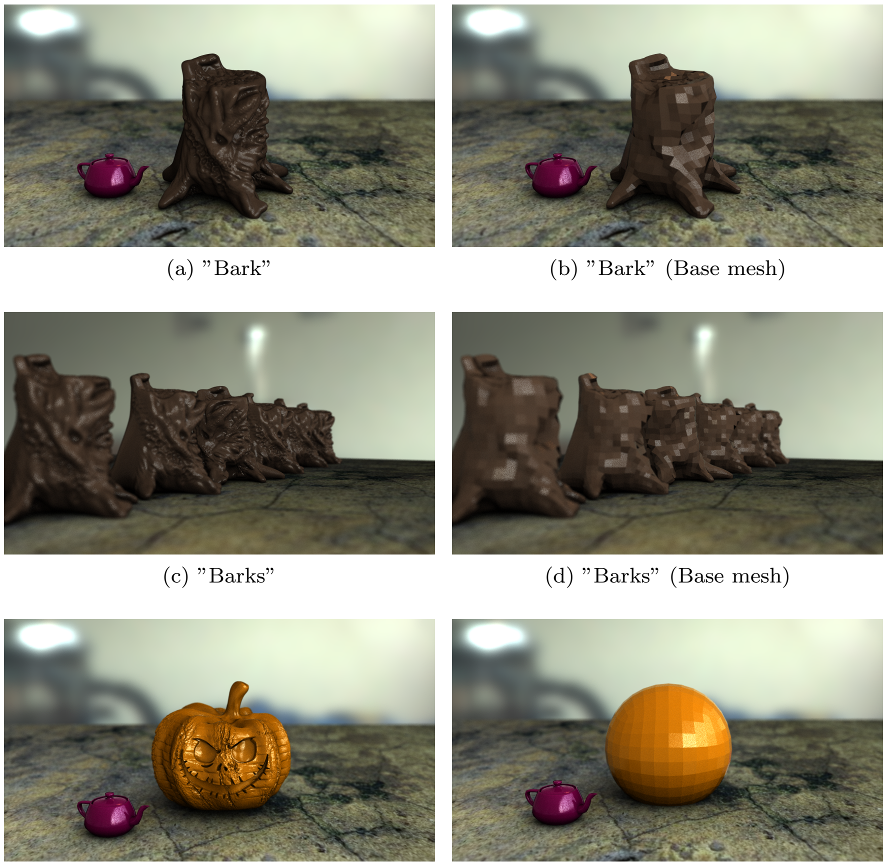
process patch-by-patch, these values are sorted by primitive index using a radix sort [Harada and Howes 11], and the start and end indices of pairs for a primitive are computed. The buffer storing the start indices is used as a job queue.

We then execute a kernel described in Sec. 1.4 which computes the intersection with VD patches. The minimum number of work groups filling the GPU is executed and each work group fetches an unprocessed VD patch from the queue, and then processes one after another.

## 1.6 Results and Discussion

We created models with vector displacement maps in Mudbox for evaluating the method. Base meshes and vector displacement maps are exported in object space. We created four test scenes with these models and models without vector displacement maps (Figs. 1.1 and 1.8). To stress the renderer, we intentionally did not use instancing for these tests, although we could use it to improve the performance for a scene in which a same geometry has been placed several times. We used an AMD FirePro W9100 GPU for all the tests.

The biggest advantage of using vector displacement maps is its small memory footprint, as it creates highly detailed geometry on the fly rather than preparing a high-resolution mesh. The memory usages with the



(a) "Bark" (b) "Bark" (Base mesh)

(c) "Barks" (d) "Barks" (Base mesh)

(e) "Pumpkin" (f) "Pumpkin" (Base mesh)

Figure 1.8: Some of our test scenes with and without vector displacement maping.

proposed method and with pre-tessellation are shown in Table 1.1. The "Party" scene requires most memory and does not fit into any existing GPU's memory with pre-tessellation. Even if we could store such a large scene in memory, it takes time to start the rendering because of the preprocess for rendering, such as IO and spatial acceleration structure build. This prevents a fast iteration of modeling and rendering. On the other hand, those overheads are low when direct ray tracing of vector displacement maps is used. The difference is noticeable, even for the "Pumpkin" scene.

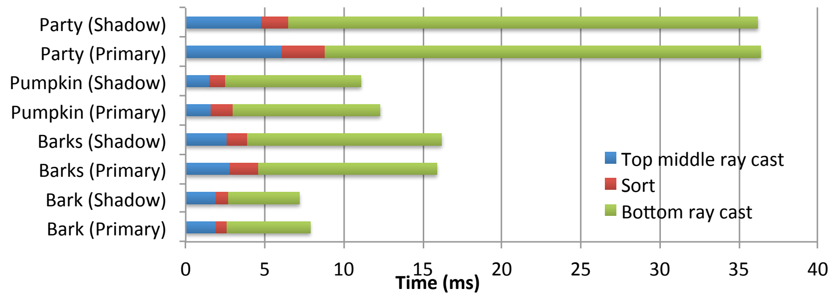


Figure 1.9: Time for top, middle ray cast, sort, and bottom ray cast.

The advantage of the memory footprint is obvious, but the question is "What is the cost at runtime, (i.e., the impact for the rendering speed)?". Despite its complexity in the ray casting algorithm, direct ray tracing of vector displacement maps was faster for most of the experiments. We rendered direct illumination of the scene under an environment light (i.e., 1 primary ray cast and 1 shadow ray cast) and measured the breakdown of the rendering time which is shown in Fig. 1.7 [2]. Pre-tessellation is faster only for the "Pumpkin" scene whose geometric complexity is the lowest among all tests. Pre-tessellation is slower for the "Bark" scene and it fails to render the other two larger scenes. This is interesting as direct ray tracing is doing more work than pre-tessellation. This performance came from less divergent computation of direct ray tracing (i.e.,the top, middle-level hierarchy is relatively shallow, and we batch the rays intersecting with a VD patch).

To understand the ray casting performance for direct ray tracing better, we analyzed the breakdown of each ray cast operation for the scenes (Fig. 1.9). These timings include kernel launch overhead which is substantial especially for sorting which requires launching many kernels. Computation time for sorting is roughly proportional to the number of hit pairs although it includes the overhead. Most of the time is spent on bottom-level BVH build and ray casting for VD patches. The time does not change much when we compare primary and shadow ray casts for the "Barks" scene, although the number of shadow rays is smaller than the number of primary rays. This indicates the weakness of the method, which is that the bottom-level BVH construction cost can be amortized when there are a large number of rays intersecting with a VD patch, but it cannot be amortized if this number

[2]The renderer is a progressive path tracer thus all screenshots are taken after it casts some samples per pixel.

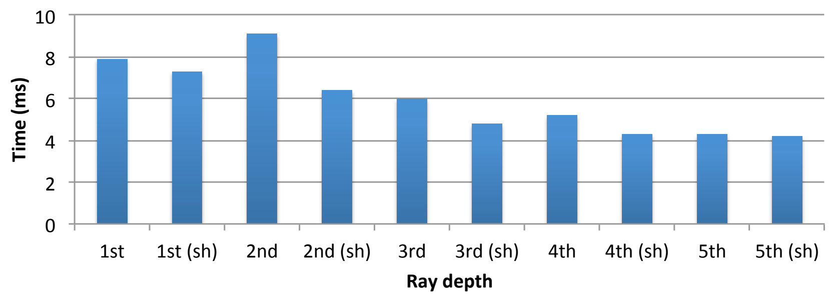


Figure 1.10: Ray casting time for each ray depth in indirect illumination computation. (sh) are ray casts for shadow rays.

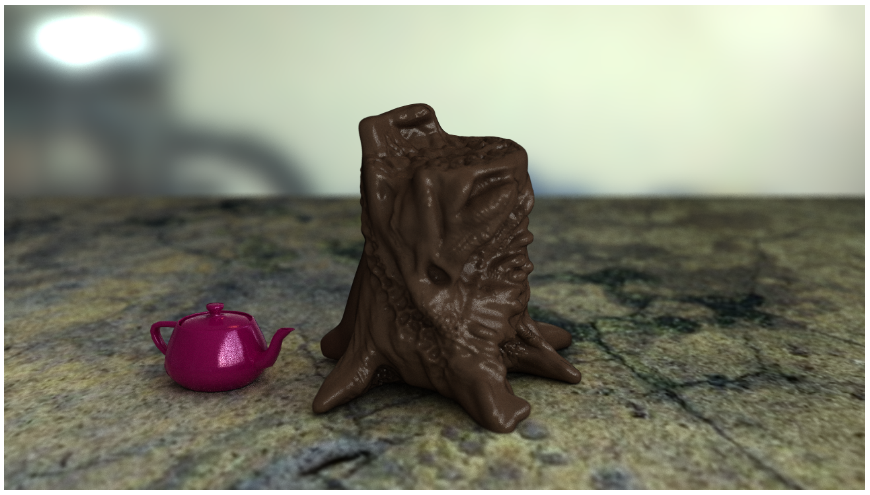


Figure 1.11: The ”bark” scene rendered with 5 bounce indirect illumination.

is too low. This is why the ray casting for shadow rays in the ”Pumpkin” scene is so slow compared to the time with pre-tessellation. The situation gets worse as the ray depth increases. We rendered indirect illumination with 5 ray bounces (depths) for the ”Bark” scene and measured the ray casting time (Fig. 1.10). Although the number of active rays decreases as it goes deeper, the ray casting time did not decrease much. This can be improved by caching the generated bottom-level BVH, which is disposed and computed again for each ray casting operation. This is an opportunity for future research.

## 1.7 Conclusion

In this chapter, we have presented a method to ray trace vector displacement mapped surfaces on the GPU. Our experiments show that direct ray tracing requires a small memory footprint only, and ray tracing performance is competitive or faster than ray tracing with pre-tessellation. The advantage gets stronger as there are more VD patches in the scene.

From the breakdown of the rendering time, we think that optimizing the BVH build for the scene and ray casting for simple geometries such as triangles and quads are not as important as optimizing the bottom-level hierarchy build and ray casting because the complexity of the bottom-level hierarchy easily becomes higher than the complexity of the top and middle-level hierarchy once we start adding vector displacement to the scene.

## Bibliography

[Hanika et al. 10] Johannes Hanika, Alexander Keller, and Hendrik P. A. Lensch. “Two-level Ray Tracing with Reordering for Highly Complex Scenes.” In *Proceedings of Graphics Interface, GI ’10*, pp. 145–152, 2010.

[Harada and Howes 11] T. Harada and L. Howes. “Introduction to GPU Radix Sort.” In *Heterogeneous Computing with OpenCL*, 2011.

[Smits et al. 00] Brian E. Smits, Peter Shirley, and Michael M. Stark. “Direct Ray Tracing of Displacement Mapped Triangles.” In *Proceedings of the Eurographics Workshop on Rendering Techniques*, pp. 307–318, 2000.

```
int localIdx = GET_LOCAL_IDX;
int lIdx = localIdx%8;// Assuming 64 work items in a work group
int lIdy = localIdx/8;
// Compute leaf nodes
for(int jj=lIdy*nn; jj<(lIdy+1)*nn; jj++)
for(int ii=lIdx*nn; ii<(lIdx+1)*nn; ii++)
{
  Aabb aabb;
  for(int j=0; j<2; j++) for(int i=0; i<2; i++)
  {
    float2 w = make_float2( (ii+i)/(float)nSplit, (jj+j)/(float)nSplit );
    float2 uv = interpolateUv( uv0, uv1, uv2, uv3, w );
    float4 v = interpolateVertex( v0, v1, v2, v3, w );
    v += texture_fetch( gVDispMap[faceIdx], uv );// Apply displacement
    AabbIncludePoint( &aabb, v );
  }
  int o = getOffset( tessLevel );
  __global GridCell* dst = &myCells[o + ii + jj*nSplit];
  dst->m_aabb = quantizeAabb( aabb );
  dst->m_n = compressF4( computeNormal(ii,jj) );
  dst->m_uv = compress( computeUv(ii,jj) );
}
GLOBAL_BARRIER;
// Computes internal nodes level by level
for(int level = tessLevel-1; level>=0; level--)
{
  int nc = (1<<level);
  int nf = (1<<(level+1));
  int oc = getOffset( level );
  int of = getOffset( level+1 );
  while( localIdx < nc*nc )
  {
    int ii = localIdx%nc;
    int jj = localIdx/nc;

    GridCell g = myCells[ of + (2*ii)+(2*jj)*nf ];
    GridCell g1 = myCells[ of + (2*ii+1)+(2*jj+1)*nf ];
    GridCell g2 = myCells[ of + (2*ii+1)+(2*jj)*nf ];
    GridCell g3 = myCells[ of + (2*ii)+(2*jj+1)*nf ];
    myCells[ oc + ii + jj*nc ] = merge( g, g1, g2, g3 );
    localIdx += WG_SIZE*WG_SIZE;
  }
  GLOBAL_BARRIER;
}
```

Listing 1.2: BVH build starting with the leaf level build and then the upper level build.