Ray Tracing and Rasterization: an Hybrid renderer

How ray tracing can leverage the actual pipeline to speed up rendering

FABRIZIO SANTORO, Cnam Enjmin, Angoulême

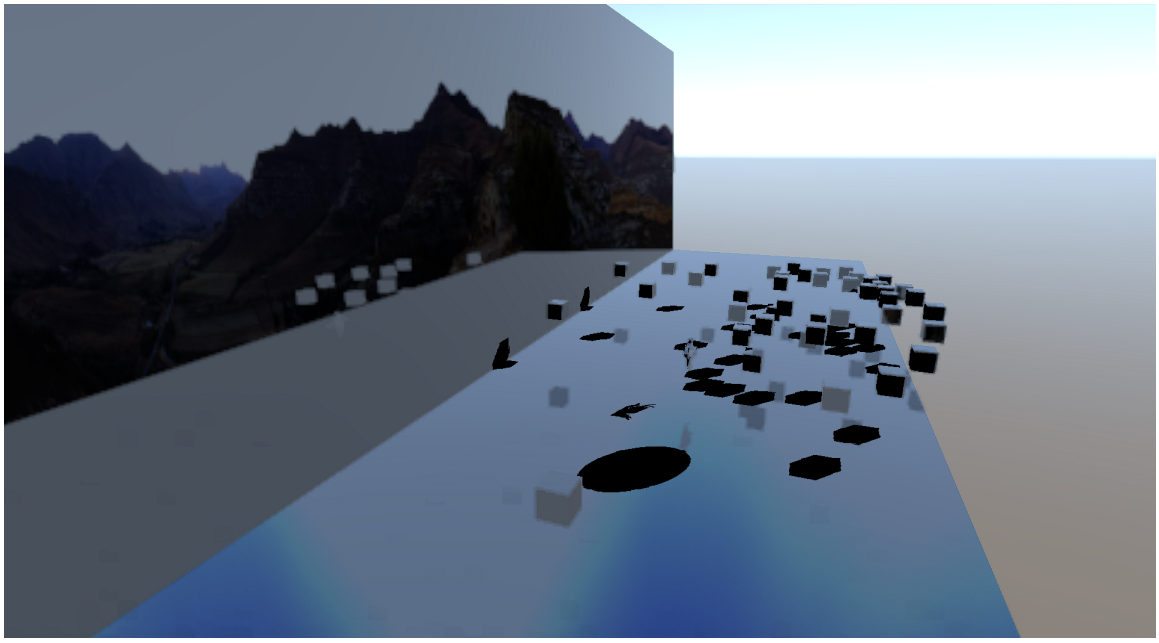

Fig. 1. Final render with the hybrid solution.

The graphic pipeline has been used for years, providing a stable and fast way of rendering pixels on the screen. Multiple techniques have been developed for it, enabling creativity to game creators and even film makers, as of today the majority of games uses this very pipeline. However, these techniques lack certain properties to produce ground thruth renderings, thus another technique should be used: ray tracing. Ray tracing is starting to reappear, thanks to dedicated hardware, re-starting the research in the field. However, the dedicated hardware is still expensive and so hybrid solutions are emerging from the field. In this paper I will attempt to explore an hybrid solution.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Ray tracing**.

Additional Key Words and Phrases: hybrid rendering, bvh, rasterization, deferred, ray tracing

Author's address: Fabrizio Santoro, Cnam Enjmin, Angoulême.

0730-0301/2020/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Fabrizio Santoro. 2020. Ray Tracing and Rasterization: an Hybrid renderer: How ray tracing can leverage the actual pipeline to speed up rendering. *ACM Trans. Graph.* 1, 1 (August 2020), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The intent of this work is to make myself familiar with the state-of-the-art ray traced rendering techniques, as they are becoming more and more a reality now that dedicated hardware is being made and sold to gamers. The scope is relatively small as it was my first time using compute shaders and a deferred pipeline, so the techniques are implemented in a simple manner.

The engine I used is the Unity engine, for no other reason that it is the one I'm most comfortable with and my custom engine still uses forward rendering, so I didn't wanted to spend time on implementing a deferred renderer and rather spend more time on discovering this hybrid approach.

I began by reading a fair amount of papers ranging from rendering techniques to spatial partitioning, which took 1/3 of the time allocated for this paper, but enabled me to see where the research was and what is the current way of doing things. Thankfully, there are great literature on the subject, so I quickly came across Ray Tracing in One Weekend [Shirley 2020] which is a great tutorial on how to implement a basic path tracer. With this knowledge in mind,

I started to work in Unity on the hybrid approach I had discovered. My work is heavily inspired on a chapter of raytracing gems, where the SEED team at EA published their work on a hybrid renderer [Barré-Brisebois et al. 2019].

1.1 The rendering pipeline

When ray tracing-based techniques are taught, they are usually considering the first pass of rays, or primary rays, starting from the camera. Although this is a correct way to render accurately, it suffers from an heavy computation cost that doesn't add anything from the standard pipeline, as the first hit is usually utilized for colors, except of course translucent materials. This is where the hybrid approach shines, because instead of using ray tracing in all its passes and thus wasting computation time, it carefully chooses between the rasterized pipeline and the ray traced one depending on whether the pass will benefit from rays or not.

What this approach does to avoid the cost of primary rays is to sample the G-Buffer to obtain the necessary data for the next bounce, depth and roughness, thus computing the world position of the ray as well as its direction, the later will change depending on the pass. A simple optimization at this stage is to exclude rays where the depth buffer is 0, or 1 if inverted. This could coupled with a sorting rays pass to optimize further the rendering time, more on this later on.

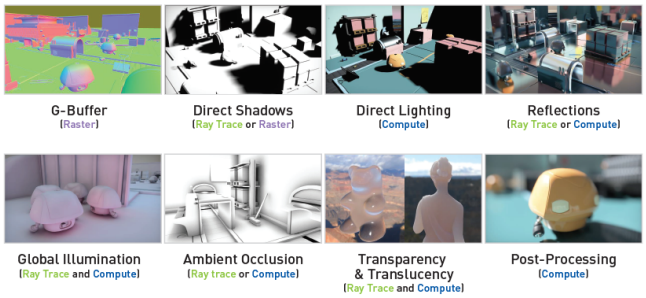

Fig. 2. Hybrid Pipeline, from PICA PICA, Ray Tracing Gems

2 RAY TRACED TECHNIQUES

In this hybrid pipeline we still heavily use the G-Buffer for the basics, but more complex effects require the ray traced approach as they benefit from the extra scene information that is not available in the G-Buffer. These techniques thus rely on the world position of the ray's origin, which can be computed as such:

```
float4 cs = float4(uvClip, depth, 1.0f);
float4 viewPos = mul(cameraInvProj, cs);

//perspective division
viewPos /= viewPos.w;

float3 worldPos = mul(cameraToWorld, viewPos);
```

Having now the origin of the ray, the direction will depend on the pass we are in.

2.1 Shadows

To simulate real time shadows, the standard pipeline, meaning the deferred one, relies on a depth buffer or in the case of Cascade Shadow Mapping, or csm, multiple buffers. Although this technique is used greatly, it requires two to four buffers to satisfy an enough precision or artifacts appear. This is where the ray tracing solution can solve this problem by not needing buffers but to actually shoot rays to see if a pixel is covered in penumbra or not, relying on the scene properties rather than a texture. The technique shines because it achieves a great result regardless of the scale of the object, whereas the csm would often need to lerp between cascades since the object would effectively overlap more than one.

Nonetheless, shadow acne do still occur with ray traced rendering, but not for the same reason as it is because the ray could miss a self-intersecting triangle and thus missing the intersection. To solve this issue, I used the same solution as the classic technique : bias. I didn't really look for an alternative solution, so this might not be the right way but the results were satisfying enough to not look further, here is my simple fix.

```
NormalData normalData;
DecodeFromNormalBuffer(positionSS, normalData);
float3 N = normalData.normalWS;

Ray r;
r.origin = worldPos + (N * 0.1f);//adding bias
r.direction = -directionalLight;
r.invDir = normalize(1 / -directionalLight);
```

2.2 Reflections

One of the rendering features that really pushes the photo-realistic feel of a game is reflective surfaces, it allows the player to immerse itself in the virtual world. However, the state-of-the-art techniques currently used are screen-spaced based, meaning that they rely on what is actually rendered on screen to compute the reflections. This problem is obvious when the camera doesn't face the entire reflected object, thus not giving enough data to the SSR pass yielding a bad reflection. In the ray traced counterpart we can clearly see all the

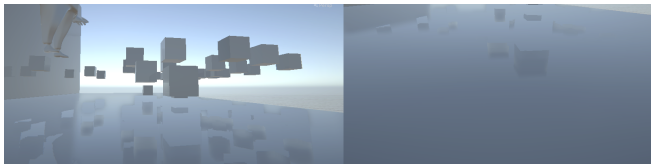

Fig. 3. Normal view of the objects(Left) cropped view of the objects (Right), Unity

scene details, even if they are not directly drawn on screen. However, such rendering is not cheap this is why it is usually done at 1/4 pixel per rays and then blurred, with some temporal accumulation. In my implementation, the temporal aspect is there but not the blur neither is a good stochastic importance sampler, I hope that i could in the near future implement the Halton sequence to have a better distributed random sampling.

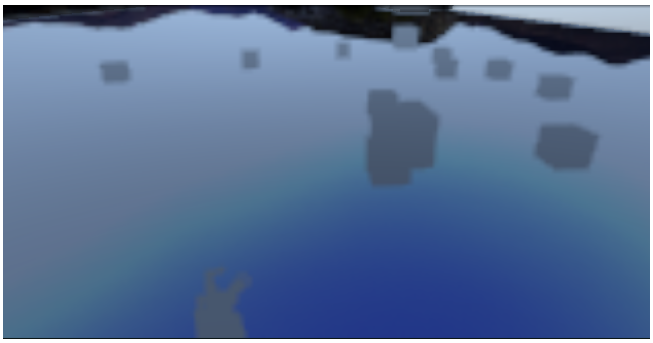

Fig. 4. Ray Traced reflections of the inaccurate view, Unity

3 ACCELERATION STRUCTURE: BVH

Although this hybrid approach enables a more accurate rendering, it also increases the processing power needed to achieve that, this is in part due to the fact that when a ray needs to know whether it hits something it has to search for a lot of triangles. Fortunately, a lot of research has been done in this field and two different approach have emerged, one that subdivides the scene spatially and the other subdivide the set of objects. The first approach can yield better results than the second one but takes longer to construct, which is why the second one is preferred, although there are ways to dynamically update the data structure avoiding a full reconstruction.

In my implementation, heavily inspired by the pbr book [Pharr et al. 2018], I implemented the second approach, particularly a Boundary Volume Hierarchy [KAYT. L. 1986] or bvh, the authors describe a data structure that subdivides the object in AABB in a tree-like manner, which is fast for querying objects thanks to its tree structure and it is also fast to compute the AABB/ray intersection, with the slab method for example.

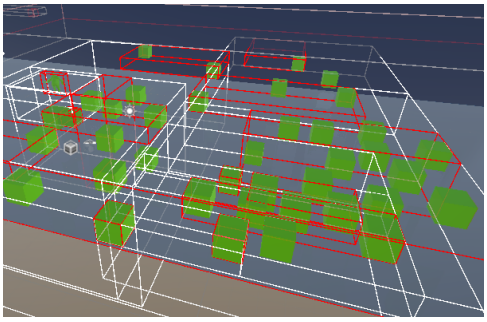

Fig. 5. BVH: leafs in red, node in white

3.1 Implementation

The implementation was done in three steps:

- Create all the AABBs for the meshes
- Build the BVH from them
- Flatten the tree to be used in the GPU

The first step was made quite easy as the Unity game engine provides AABB for all meshes, called bounds. The second step is where the actual tree is built. In my implementation, I use a classic BVH which is built on the CPU and then sent to the GPU, but it is important to note that there are other types of tree derived from this that can be done entirely on the GPU, such as the HLBVH [Pantaleoni and Luebke 2010].

When building the tree, an efficient way of subdividing the objects is to choose an axis by which we will divide them. The best axis to choose is the one with largest length, by unifying all the AABBs we can find the coordinate we are looking for. All the objects now lie on the chosen axis, to continue we need to know if we create a leaf or subdivide the tree and if so where do we split the region. To solve this problem the SAH has been used in my implementation, although it is not the only one available it is the one that produces the best answer.

3.2 SAH

The Surface Area Heuristic is a technique that computes heuristically the cost between splitting the group of objects and creating a leaf, where the cost of creating the leaf equals:

$$\sum_{i=1}^{N} t_{isect}(i)$$

Where N is the number of primitives and $t_{isect}(i)$ represents the cost of intersection ray-object of the ith object. The cost of splitting is derived by this heuristic:

$$c(A,B) = t_{trav} + p_A \sum_{i=1}^{N_A} t_{isect}(a_i) + p_B \sum_{i=1}^{N_B} t_{isect}(b_i)$$

Where t_{trav} is the time taken to compute which child is traversed by the ray, pA and pB are the probabilities of the ray passing through the child, Na and Nb represent the objects falling inside both the regions. In my implementation I assume that the cost of $t_{isect}(i)$ is constant for all objects.

The probabilities Pa and Pb are computed as such:

$$p(A|B) = \frac{s_A}{s_B}$$

From geometric probability we can show that if A a surface area is hit by a ray, B and C which are subdivision of A have s_B/s_A and s_C/s_A probability to be hit as well, respectively. If the SAH suggests

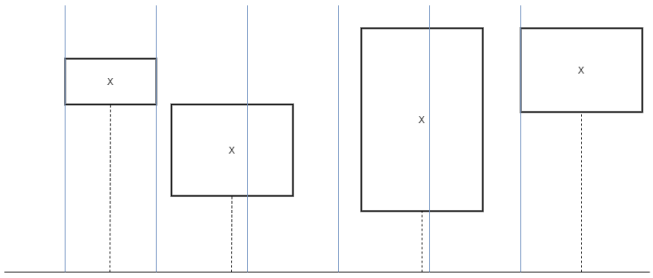

Fig. 6. Dotted lines are the projected centroids (the X), the blue line are the splitting planes assigned to each bucket

to split then, the next step is to decide where to on the axis. For that

we can utilize buckets that reduce the number of comparisons by projecting the centroids of the objects to the chosen axis and see which one of the splitting plane is the cheapest. When the splitting occurs, the algorithm rearranges the objects, this step is useful for the flattening that happens later on.

```
//We compute the buckets count and bounds
for (int i = start; i < end; i++)
{
    int bucket = (int) (nBuckets *
    GetRelativePosition(centroidBounds,
            primitivesInfo[i].bounds.center)[axis]);
    //the last bucket is useless for subdividing
    if (bucket == nBuckets) bucket = nBuckets - 1;
    buckets[bucket].count++;
    buckets[bucket]
        .bounds.Encapsulate(primitivesInfo[i].bounds);
}
//computing the cost of splitting
for (int i = 0; i < nBuckets - 1; i++)
{
    Bounds b0 = buckets[0].bounds;
    int count0 = buckets[0].count;

    Bounds b1 = buckets[i+1].bounds;
    int count1 = buckets[i+1].count;

    //left side of the division
    for (int j = 1; j <= i; j++)
    {
        b0.Encapsulate(buckets[j].bounds);
        count0 += buckets[j].count;
    }

    for (int j = i+2; j < nBuckets; j++)
    {
        b1.Encapsulate(buckets[j].bounds);
        count1 += buckets[j].count;
    }
    //the constant here is the cost of choosing the child
    //relative to the cost of ray/object intersection
    costs[i] = 0.5f
             + (count0 * GetSurfaceArea(b0) + count1 *
     GetSurfaceArea(b1))
             / GetSurfaceArea(bounds);
}
```

Knowing which split plane to choose now is just a matter of getting the minimum value of the array costs.

3.3 Flattening the tree

We now have a bvh tree built and ready to be used, but we first have to linearize it as the GPU doesn't support pointer that well and cache misses would be avoided. Flattening a tree is not a complex operation but making sure that it is well suited for the GPU require some work. First, the actual flattening is done by recursively visiting the tree depth-first and setting the children node id in the flat-node. The order is crucial, as we can skip the left-most children as it is directly after the parent thus not needing its location in the structure. The whole tree is stored in an array that is sent to the GPU, the structure that represent a node is defined as such :

```
struct LBVH
{
 float3 minBox;
 float3 maxBox;
 int offset; //could be second child offset or object
     index
 int primAndAxis; // LMB -> node  RMB -> axis
};
```

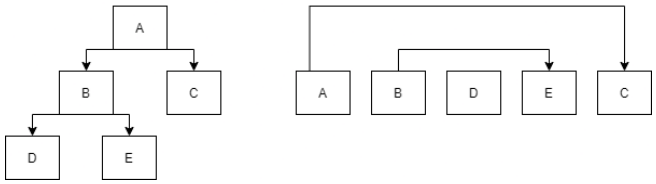

Fig. 7. From a tree with pointer to a pointerless compact tree

The first two members are for the AABB, the offset is used either as an offset or a index object, to know which one the last member primAndAxis is masked, currently the 32bits are split between the axis used in the splitting and the number of objects in the AABB. This might be optimized further, as the axis is only 2 bits so a better splitting could be 30 bits for the number of objects and 2 bits for the axis, but as most of my test scenes don't have much objects in them I didn't find it useful, but it is important to keep in mind.

As for the size in memory of the struct, it fits in a SIMD unit as it is exactly the same size as a float4x4, or an homogeneous matrix, that are commonly used in shaders. Traversing the tree is then done as follows, reduced for clarity :

```
int stackNodes[32];
//This is useful to know which branch to choose
bool isDirNeg[] = {r.invDir.x < 0, r.invDir.y < 0,
r.invDir.z < 0};
int toVisitOffset = 0, currentNodeIndex = 0;
LBVH node = bvhTree[currentNodeIndex];
while (true)
{
node = bvhTree[currentNodeIndex];

if (rayBoxIntersection(node.minBox, node.maxBox, r))
{
    int primitives = node.primAndAxis >> 16;
    if ( primitives > 0) //it's a leaf
    {
        // Test the ray/object intersections
        [...]
         if (toVisitOffset == 0) return false;
         currentNodeIndex = stackNodes[--toVisitOffset];
     }
     else
     {
       // We choose the correct child
       // the other one goes into the stack
       if (isDirNeg[node.primAndAxis & 3])
       {
           //left hand child
           stackNodes[toVisitOffset++] = currentNodeIndex
     + 1;
           currentNodeIndex = node.offset;
       }
       else
       {
           stackNodes[toVisitOffset++] = node.offset;
           currentNodeIndex = currentNodeIndex + 1;
       }
     }
 }
 else
 {
    // we check for the node in the stack
```

```
    // if none we stop
     if (toVisitOffset == 0) return false;
         currentNodeIndex = stackNodes[--toVisitOffset];
 }
}
```

A stack is used instead of a recursive call which saves a bit of computation time and memory but introduces a problem for scenes with a lot of objects, as the maximum depth of the tree should not be above the stack allocated size. In this case some solutions are available, the easiest is to regroup more objects in a leaf which will reduce the depth of the tree but increase the number of ray/object intersection tests. Another possible solution would be to stop when the stack is full and trying all nodes that are in, with the hope that one would be a leaf.

4 PERFORMANCE

To understand why a piece of code is fast or not, we need to know how the architecture running it works, which in our case is the GPU. We could translate most of the paradigms from a multi-threaded CPU program to a compute shader, but the massively parallel architecture of the GPU makes it difficult. As the GPU has a large amount of core, divided in groups which share L2 Cache, global memory and other types of memory chips thus highlighting the need of independent workload to make sure that the thread are the most independent from each other. Another one of the major differences caused by the groups of thread is that, the group is finished only when all thread in it are, so if a thread takes a computation heavy path on a branch the other threads have to wait causing idle time and reducing the occupancy of the GPU.

This is why knowing how the GPU handles fetching data and branches is crucial to obtain high performance. However there are some gotchas to be aware of, for example branches are really your enemy in loops especially because the thread will start the comparing instructions but it usually takes more than cycle so instead of idling it starts both branching, also because of its SIMD nature. The SIMD nature also mean that when a thread is fetching data, the fetching instruction is executed but the actual retrieving of the data takes some cycles depending on where the data is, so instead of waiting the thread will execute another subgroup code thus hiding the stalls, but this requires that if N subgroups are executed, then the number of registers available by subgroups are divided by N.

After analyzing the pipeline it is clear that we must avoid branches in crucial parts of our code. A simple example would be

```
float3 color = (float3)0;
if(depth > 0)
    color = float3(1, 0, 0);
else
    color = float3(0, 1, 0);
TO
int depthGreater = depth > 0;
color = float3(1, 0, 0) * depthGreater
      + float3(0, 1, 0) * (1 - depthGreater);
```

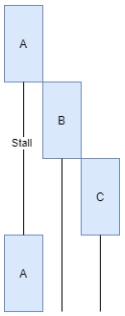

Fig. 8. Compute Unit stalling and executing other subgroups to then resume, requires storing all the subgroups data in registers

Which eliminates the branch and yield the same result, of course this is a simple example but is a technique that could save up some cycles.

4.1 Working with the cache

One of the most expensive operation is fetching data, its cost depends highly on where the data is stored and how fast can it be transferred. To counterpart this cost when loading memory, for example a float4, not only the data asked is fetched but a chunk after it as well, filling the cache line in our case. When the data we need is all in the cache line the operations are nearly free as the data is already loaded, this is why random access is not cheap, because when the data is not loaded in the line, the GPU flushes the line a loads the next chunk.

To optimize compute shaders code it is thus necessary to keep in mind the locality of the data and branches. We will analyze the reflection pass and see how cache misses influence the performance. The scenari tested are

- Same pixel offset set for all the threads(1)
- Different pixel offset for each threads(2)
- Mesh vertices retrieved with the ebo(1|2)
- Mesh vertices with no ebo(1|2)

To really show how the locality of the data can affect cache miss I have chosen to include test cases that share or not the same origin of the ray which is reflected in the cache miss.

Metrics	L1 hit	L1 miss	L2 hit	L2 miss
offset, no Ebo	4,873,530	291,719	115,552	217,475
offset, Ebo	6,251,450	258,988	91,876	214,712
no offset, no Ebo	5,374,096	290,892	134,387	194,732
no offset, Ebo	5,979,267	241,950	87,422	192,207

Table 1. Metrics reported by PIX on the Unity build

We can deduce some interesting findings thanks to the report, we see for example that the offset has a drastic change in the L1 cache usage and also that not using Ebos increases the L2 usage. The usage of different noise value per ray introduces less memory usage because more ray are culled, because their pixel from the depth

buffer is 0, or 1 if inverted, causing it to not yield useful information. The changes introduced by not using Ebo and thus having the buffer of vertices filled with all the triangles list, was a good idea at first because it reduced indirection so it should be beneficial. However, the indirection actually used locality thus hiding the cost of the indirect pointer, because the Ebo is sorted, when the vertices data were fetched they usually were used in the loop as the next vertices are close in the Ebo buffer.

This highlights the fact that locality is very important for performance and that the algorithm produces a lot of cache misses. Since locality matters, there are some techniques that help keeping the locality high, for example [Bavoil 2020] suggests reordering the threads to launch rays that a near each other to increases locality as the rays have a high probability to follow the same path. Another approach to keep locality between threads would be to sort the rays, [Garanzha1 and Loop 2010] presents a pipeline where rays are sorted in a way that threads only do work they need to, saving time and increasing locality, by implementing kernels that are very specific to one pass and avoiding branching as much as possible.

4.2 Sorting the rays

In this section I will describe how I implemented a naive way of sorting the rays and what results came up. First, the pipeline used in this experiment is : We first sort the rays by appending the rays

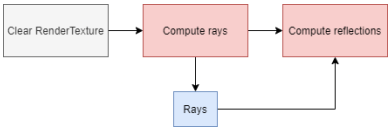

Fig. 9. The rays are appended by the first dispatch and consumed by the second one, only the necessary rays are included

that can be used to compute reflections, this is done by using an AppendBuffer, the shader decides whether or not to include the ray depending on the depth at the pixel coordinate, if it's 0 or 1 if it's inverted the ray will not be included. The next step is to compute the reflections by traversing the BVH and intersecting the triangles by consuming the rays in the buffer. Because we are using a Buffer instead of calculating the ray, we need to compute where in the texture the ray's result needs to go, for that I choose to project the ray origin back to clip space by doing the inverse of what was done to project it in the first place.

```
float4 pos = mul(worldToCamera, float4(r.a, 1.0f));
pos = mul(invProj, pos);
pos /= pos.w;

pos = (pos + 1.0f) / 2.0f;

texOut[pos.xy * texSize] = float4(hit.color, 1.0f);
```

While it seemed like a good idea, this naive approach turned out to be worse than the classic one I used just before. However, the results are interesting, first we can see that the actual traversing and intersecting is fast 7.90ms reported by PIX and 24.63ms for ray sorting, which is slower but shows that if the sorting algorithm is optimized so will be the rendering as it is the bottleneck now. This attempt also highlighted the decrease in cache miss and usage.

Metrics	L1 hit	L1 miss	L2 hit	L2 miss
Sorting rays	7,737,713	183,616	104,618	103,728
reflections	62,224,457	616,827	340,515	359,491

The spike in cache usage in the reflections pass is due to the fact that one ray is computed per thread, which results in (240, 135, 1) threads used which is more than the number of threads available in most graphic cards, my Vega 56 has 56 compute unit with 16 SIMD unit which is not enough. The GPU thus needs to schedule a greater number of threads that are available which increases the memory required to dispatch the that many threads. Although the high number of required threads is putting more pressure on the GPU, the most expensive operation remains the sorting which has lower cache usage but higher latency.

5 CONCLUSIONS AND FUTURE WORK

This work was primarily to make myself familiar with the state-of-the-art techniques in ray tracing, with the dedicated hardware being too expensive still I chose to only use non-dedicated hardware as most of the techniques could be translated to the dedicated one. With a careful implementation and more work, an optimal rendering could be achieved as shown in [Barré-Brisebois et al. 2019], but my limited understanding and time made me choose only the simpler techniques. A good base to extend upon has been done during the experimentation, with some important techniques lacking such as importance sampling or all randomness related techniques. However, the goal to make myself familiar is fulfilled as I now understand the principles of ray tracing and the challenges that go along with it.

When I was looking for partitioning data structures I wondered if a mix between octrees and BVHs existed, and for the 2020 SIGGRAPH edition there is this paper [D. and et al 2020] that presents a mix of the two, which shows great result but was too complicated for me to implement. Another field I'd like to explore further is compute shader optimizations, which was new to me but did make me discover how a GPU really work. My work can be found at https://github.com/Fabpk90/PathRasterizer

REFERENCES

Colin Barré-Brisebois, Henrik Halé, Graham Wihlidal, Andrew Lauritzen, Jasper Bekkers, Tomasz Stachowiak, and Johan Andersson. 2019. *Hybrid Rendering for Real-Time Ray Tracing.* Apress.

Louis Bavoil. 2020. Optimizing Compute Shaders for L2 Locality using Thread-Group ID Swizzling. https://developer.nvidia.com/blog/optimizing-compute-shaders-for-l2-locality-using-thread-group-id-swizzling/.

Ströter D. and Mueller-Roemer J.S.and Stork A. et al. 2020. OLBVH: octree linear bounding volume hierarchy for volumetric meshes. (2020). https://doi.org/10.1007/s00371-020-01886-6

Kirill Garanzha1 and Charles Loop. 2010. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. (2010).

KAJIYAJ. T. KAYT. L. 1986. Ray tracing complex scenes.

Jacopo Pantaleoni and David Luebke. 2010. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry.

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2018. *Physically Based Rendering: From Theory To Implementation.*

Peter Shirley. 2020. Ray Tracing in One Weekend. https://raytracing.github.io/.