

Performant GPU Pathtracing

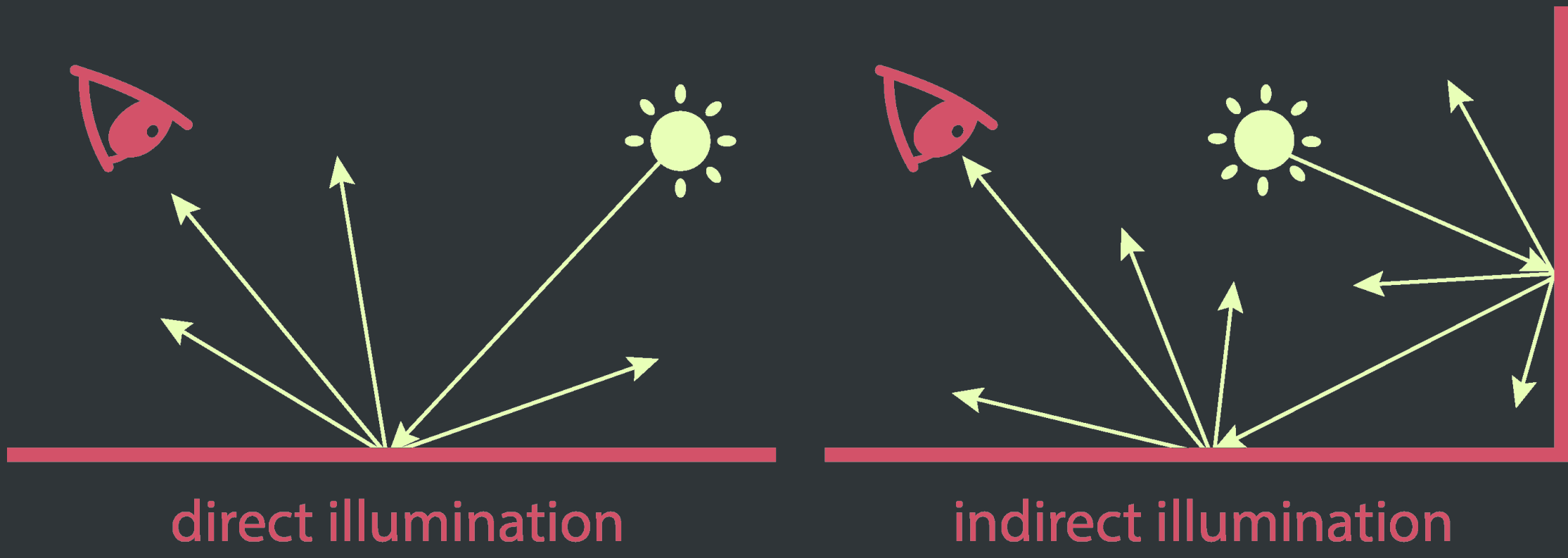
Introduction

Raytracing

Raytracing is a common method used in computer graphics to render three-dimensional images. Commercial renderers focus primarily on image fidelity and secondarily on performance (speed), but for professionals like filmmakers, architects, and game developers, the extended time involved in rendering a complex scene limits the number of iterations and refinements possible during the creative process. This project tests the effectiveness of developing a unidirectional pathtracer (a type of raytracer) that prioritizes performance over fidelity. The hypothesis is that with a focus on GPGPU (General-Purpose Graphical Processing Unit) computing and a novel implementation of algorithms, render times can be reduced without significantly impacting image quality. This could greatly improve iteration speed for large projects.

Pathtracing

Pathtracing uses Monte-Carlo integration (a method for approximating an integral) to simulate global illumination (indirect-illumination). It would take infinitely long to truly calculate the lighting for a scene, but by using Monte-Carlo integration, the global illumination for a point can be approximated to a high degree of accuracy in very little time. The pathtracer shoots out rays for each pixel on the screen. It then goes along each ray until it hits something in the scene. At the point of intersection, it accumulates colour and lighting data from the collider and shoots more (pseudo) random rays (samples of the Monte-Carlo integration) off into the scene again. It repeats this process until it either doesn't collide with anything or reaches the maximum number of bounces. The averaged result of these samples is the returned colour for the original pixel with approximated global illumination.



k-d Tree Traversal

k-d trees are the multidimensional analogue of Binary Search Trees, where a hyperplane is used to split elements of a node (this process happens recursively until a termination function is satisfied). In short, a k-d tree is a method of spatially splitting a scene into nodes. In a scene without spatial splitting, each ray would have to iterate through each object in the scene and separately do a collision test. This can be extremely costly in large scenes with millions of triangles (the base unit of geometry in a scene). In large scenes, each pixel will do hundreds of samples and each sample will have ten or so bounces, resulting in billions of intersection tests. With spatial splits, you can traverse through the tree and only do collision tests with triangles that are likely to intersect the ray. During runtime, if a ray is colliding with the scene, it can recurse down the tree with each node that it enters, removing an exponential amount of intersection tests. This dramatically improves render times, going from an $O(n^2)$ algorithm to $O(\log n)$. In reality, no scene can be perfectly divided so that there are no overlapping triangles, so issues arise with traversal where multiple nodes can contain triangles that could potentially intersect with the ray. Solutions are discussed below.

Procedure

Research was conducted and a Parallel GPU Pathtracer was developed to meet the criteria of the study. (See concatenated source code, Addendum A, display binder.) The final product was tested comparing its performance to that of commercial pathtracers. Using the same hardware, each test scene was rendered across three different pathtracers - Cycles CPU, Cycles GPU, and my own. The tests were repeated with a varying number of samples to compare performance with increasing fidelity. The results were recorded with their time and image. As computers are deterministic machines, multiple tests for each scenario were not necessary.

Algorithms & Implementation

Groundwork and Strategy

Work began by laying down the core structural features of the pathtracer: Scene Loading; Custom Scene Format; OBJ Loading; Scene Serialization; OSX and Windows Front ends; OpenCL workload distribution utilities; a server serving the UI (as a website); and a remote debugger via websocket server. Then to meet the criteria of the study, it was most valuable to focus on the acceleration structure of the pathtracer. The two algorithms chosen were: the $O(n \log^2 n)$ Surface Area Heuristic (SAH) k-d tree construction algorithm; and a persistent short stack k-d tree traversal algorithm.

SAH k-d Tree Construction

SAH k-d tree construction is one of the best-known ways to find the cost of a plane split. The SAH of a split is the potential cost of traversal and intersection for a given split. The SAH itself provides no way of finding the minimal costs. Luckily, the minima can only be found on minimum and maximum bounds of an object within the voxel being split. This is because the change in cost is linear between each edge bound of an object. It is then easy to sweep across all possible minima with only $O(n \log^2 n)$ complexity. While there are faster algorithms, the increased implementation time for them would not have been worth it for the minimal speed increases. Recently the practice of using a favouring function (λ) to increase the chance of a split where one of the sides has no objects has become quite common showing consistent speed increases. My modified implementations of these algorithms can be found below:

Favouring Function

$$\lambda(P_L, P_R, N_L, N_R) = \begin{cases} 80\% & (N_L \equiv 0 \vee N_R \equiv 0) \wedge (P_L \neq 1 \wedge P_R \neq 1) \\ 100\% & \text{otherwise} \end{cases}$$

Cost Function

$$C(P_L, P_R, N_L, N_R) = \lambda(P_L, P_R, N_L, N_R) (K_T + K_I(P_L N_L + P_R N_R))$$

Algorithm .1: Surface Area Heuristic.

```
1 (Cost_Side) SAH(p,V,N_L,N_R)
2 {
3   Voxel V_L,V_R;
4   voxel_split(p,V,&V_L,&V_R);
5   P_L = N_L/|N_L|;
6   P_R = N_R/|N_R|;
7   C_L = C(P_L,P_R,N_L+N_R,N_R);
8   C_R = C(P_L,P_R,N_L+N_R,N_L);
9   return min((C_L, LEFT), (C_R, RIGHT));
10 }
```

Algorithm .3: Classify.

```
1 (T_L,T_R) classify(tree,T,p,N_L,N_R)
2 {
3   T_L = null; T_R = null;
4   T_L = null; T_R = null;
5   T_L = null; T_R = null;
6   T_L = null; T_R = null;
7   for(i=0; i<T; i++)
8   {
9     if(isLeft) isRight = false;
10    for(j=0; j<3; j++)
11    {
12      t_j = t_j;
13    }
14    if(t_j < p_j)
15      isLeft = true;
16    if(t_j > p_j)
17      isRight = true;
18    if(isLeft & isRight) // planar
19    {
20      if(p_j == RIGHT)
21      {
22        T_L = null; T_R = null;
23      }
24      else
25      {
26        T_L = null; T_R = null;
27      }
28    }
29    if(isLeft)
30    {
31      T_L = null; T_R = null;
32    }
33    if(isRight)
34    {
35      T_R = null; T_L = null;
36    }
37    return (T_L,T_R);
38 }
```

Algorithm .4: GenerateTree.

```
1 Node gen_node(tree,V,T,depth)
2 {
3   Node n;
4   (N_L,N_R,p,side) = find_plane(tree,V,T);
5   n_p = p;
6   if(depth == tree_max_depth || p == K[T])
7   {
8     n_T = T;
9     return n;
10  }
11  }
12  }
13  }
14  }
15  }
16  }
17  }
18  }
19  }
20  }
```

Algorithm .2: Find Split Plane $O(n \log n)$.

```
1 (N_L,N_R,p,side) find_plane(tree,V,T)
2 {
3   best_cost = inf; best_p = null; side = null; E = null;
4   best_N_L = best_N_R = best_N_P = 0;
5   for(i=0; i<tree->k; i++)
6   {
7     j = 0;
8     E = null;
9     for(t=0; t<T; t++)
10    {
11      Voxel V;
12      B = voxel_gen_from(t,t);
13      B = voxel_clip(B,V);
14      if(voxel.is_planar(B, i))
15      {
16        E_L = { p, B_max, k, PLANAR };
17      }
18      else
19      {
20        E_L = { p, B_max, k, START };
21        E_R = { p, B_max, k, END };
22      }
23    }
24    sort(E); // sort the events by b
25    N_L = N_R = 0;
26    N_P = |E|;
27    for(i=0; i<|E|; i++)
28    {
29      p = E[i];
30      P_max = P_min = P_cost = 0;
31      while(i<|E| & E[i].p == p & E[i].type == END)
32      {
33        P_cost++; i++;
34      }
35      while(i<|E| & E[i].p == p & E[i].type == PLANAR)
36      {
37        P_cost++; i++;
38      }
39      while(i<|E| & E[i].p == p & E[i].type == START)
40      {
41        P_cost--; i++;
42      }
43      N_L = P_cost; N_R = P_cost; N_P = P_cost;
44      sub_data = SAH(k, p, V, N_L, N_R);
45      if(sub_data.cost < best_cost)
46      {
47        best_cost = sub_data.cost;
48        best_p = p;
49        best_side = sub_data.side;
50        best_N_L = N_L; best_N_R = N_R; best_N_P = N_P;
51      }
52      N_L += P_cost; N_R += P_cost; N_P = 0;
53    }
54  }
55  return (best_N_L,best_N_R,best_p,best_side);
56 }
```

Persistent Threading

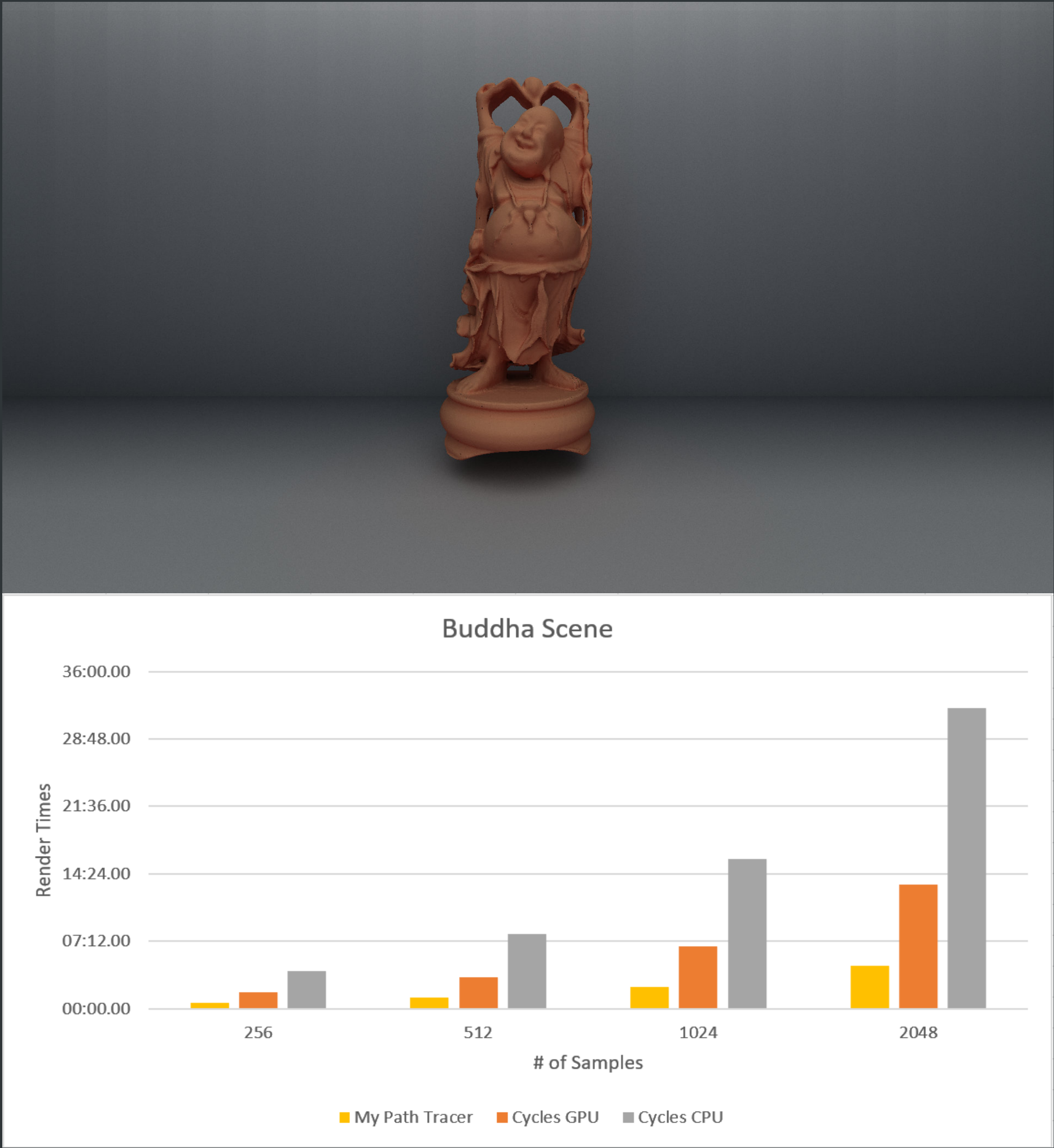
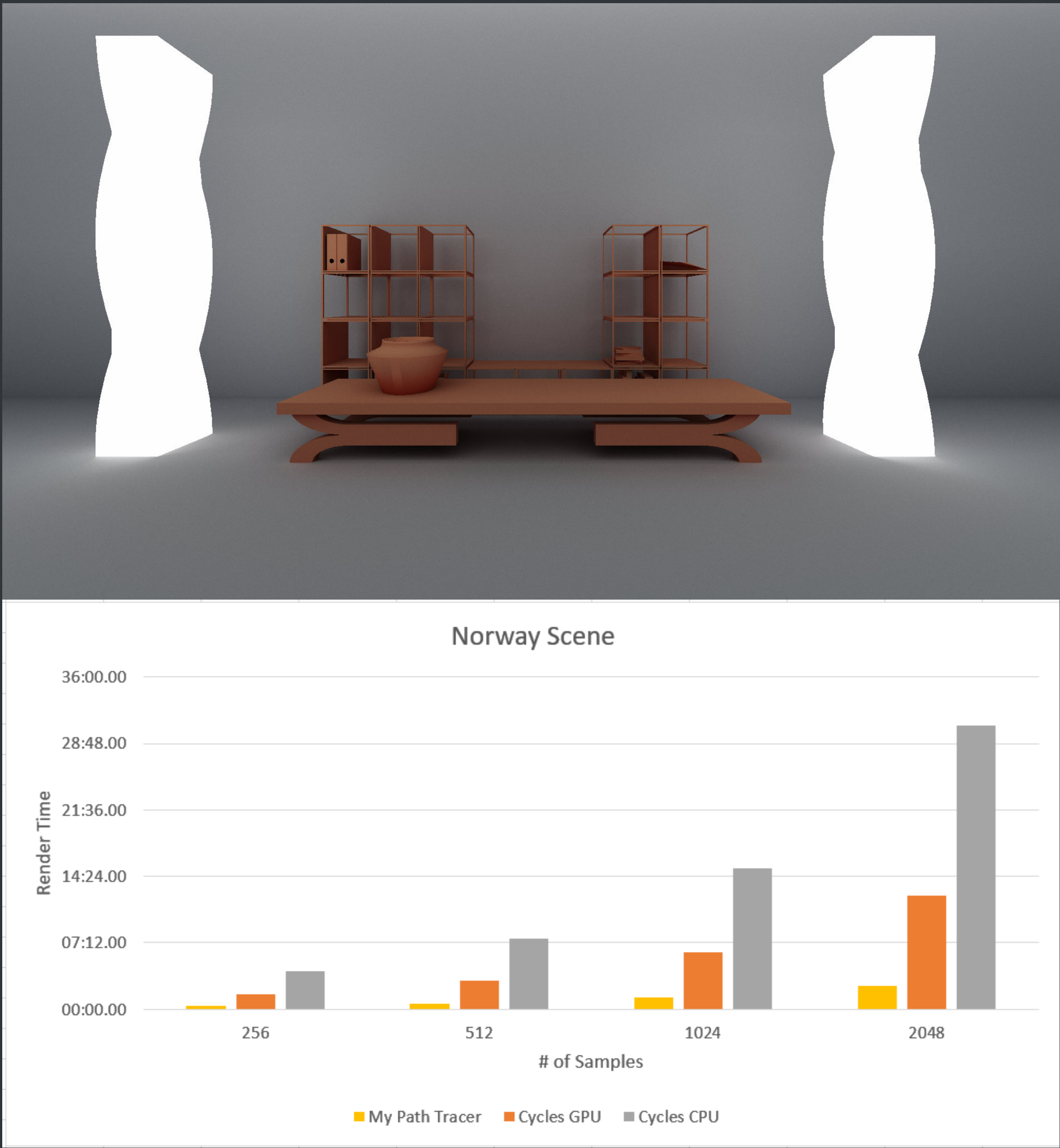
Persistent Threading utilizes the improved performance of warp synchronous execution (a warp is Nvidia's unit for a group of processors that run using SIMT, also known as a Wavefront on AMD hardware). It enforces that only a single task (thread) is assigned to each core of the GPU. This allows for improved SIMT performance between cores of a warp/wavefront as it enforces equal distribution of work tasks (especially when there is a non-trivial workload). It gets around the limited workloads of warp synchronous programming by implementing a global work queue that each thread pulls off of. Persistent threading also alleviates the unbalanced workload distribution that can happen with algorithms that don't fit the SIMT model well (such as tree traversal). The GPU scheduler can sometimes distribute work incorrectly where certain warps will be doing all of the heavy lifting for an operation, but because each core Only has one thread, persistent threading can bypass the schedulers as the workload must be evenly distributed. This can improve the performance of these algorithms as it takes advantage of the entire GPU.



NVIDIA GP100 Stream Multiprocessor

Performant GPU Pathtracing

Results



Short Stack k-d Tree Traversal With Peristent Threading

```
Algorithm 5: Persistent Short Stack K-D Tree Traversal.
1  (type,node,leaf) = update_state(tree,buffer)
2  type = tree.type;
3  leaf = node == null;
4  if (type == LEAF)
5  {
6    leaf = tree.leaf;
7  }
8  else if (type == NODE)
9  {
10   node = tree.node;
11   leaf = tree.leaf;
12 }
13 return (type,node,leaf);
14
15 (index,leaf) = traverse(ray,buffer, indices, vertices, tree,buffer);
16
17 blocksize = STREAM_PROCESSORS.PER_STREAM_MULTIPROCESSOR;
18 blocksize = SIMD_GROUPS.PER_STREAM_MULTIPROCESSOR;
19 x = SIMD_ID % blocksize; //id within the SIMD GROUP
20 y = SIMD_ID / blocksize; //id of the SIMD GROUP within the Stream Multiprocessor
21 //NOTE: shared memory is called local memory in OpenCL
22 shared volatile next_ray_array[blocksize]; //shared across all processors in the multiprocessor
23 shared volatile ray_count_array[blocksize];
24 //NOTE: In the implementation, the warp_counter is initialised on the cpu and copied.
25 global volatile warp_counter; //global memory is shared across the entire device.
26
27 next_ray_array = 0;
28 next_warp_counter = 0;
29 (node_ptr, num, num) stack[STACK_SIZE];
30
31 ray r;
32 hit = 0;
33
34 triangle_index = 0;
35 num = 0;
36 num_max = 0; num_min = m;
37 ktree.node = node;
38 ktree.leaf = leaf;
39 count_type = NODE;
40 pushdown = false;
41 ray_index = 0;
42
43 while (true)
44 {
45   //get this SIMD group's ray count
46   shared volatile int local_pool_ray_count = ray_count_array[x];
47   //get this SIMD group's next ray
48   shared volatile int local_pool_next_ray = next_ray_array[x];
49   //if (x == 0) local_pool_ray_count = 0;
50   //local_pool_next_ray = atomic_add(warp_counter, BATCH_SIZE); //retrieve and increment
51   local_pool_ray_count = BATCH_SIZE;
52 }
53 ray_index = local_pool_next_ray + x;
54 if (ray_index >= ray_count) break;
55
56 if (x == 0)
57 {
58   local_pool_next_ray = 32;
59   local_pool_ray_count = 32;
60 }
61
62 r = ray_buffer[ray_index];
63 (dist, count, num_max) = collide_voxel(SCENE_V, r);
64 if (dist < 0)
65 {
66   num_max = m;
67 }
```

```

71 stack.clear();
72 root = tree.root;
73 while (true)
74 {
75   if (stack.empty())
76   {
77     node = root;
78     current_type = NODE;
79     num = num_max;
80     num_min = num_max;
81     pushdown = true;
82   }
83   else if (type == LEAF)
84   {
85     (index,leaf) = stack.pop();
86     (type,node,leaf) = update_state(tree,buffer, index);
87     pushdown = false;
88   }
89   while (current_type != LEAF)
90   {
91     type = current_type;
92     leaf = current_type == LEAF;
93     if (leaf)
94     {
95       if (type == LEAF)
96       {
97         (index,leaf) = stack.pop();
98         (type,node,leaf) = update_state(tree,buffer, index);
99         pushdown = false;
100       }
101       else if (type == NODE)
102       {
103         (index,leaf) = stack.pop();
104         (type,node,leaf) = update_state(tree,buffer, index);
105         pushdown = false;
106       }
107       if (pushdown)
108       {
109         (index,leaf) = stack.pop();
110         (type,node,leaf) = update_state(tree,buffer, index);
111         pushdown = false;
112       }
113       if (pushdown)
114       {
115         (index,leaf) = stack.pop();
116         (type,node,leaf) = update_state(tree,buffer, index);
117         pushdown = false;
118       }
119       if (pushdown)
120       {
121         (index,leaf) = stack.pop();
122         (type,node,leaf) = update_state(tree,buffer, index);
123         pushdown = false;
124       }
125       if (pushdown)
126       {
127         (index,leaf) = stack.pop();
128         (type,node,leaf) = update_state(tree,buffer, index);
129         pushdown = false;
130       }
131       if (pushdown)
132       {
133         (index,leaf) = stack.pop();
134         (type,node,leaf) = update_state(tree,buffer, index);
135         pushdown = false;
136       }
137       if (pushdown)
138       {
139         (index,leaf) = stack.pop();
140         (type,node,leaf) = update_state(tree,buffer, index);
141         pushdown = false;
142       }
143       if (pushdown)
144       {
145         (index,leaf) = stack.pop();
146         (type,node,leaf) = update_state(tree,buffer, index);
147         pushdown = false;
148       }
149       if (pushdown)
150       {
151         (index,leaf) = stack.pop();
152         (type,node,leaf) = update_state(tree,buffer, index);
153         pushdown = false;
154       }
155       if (pushdown)
156       {
157         (index,leaf) = stack.pop();
158         (type,node,leaf) = update_state(tree,buffer, index);
159         pushdown = false;
160       }
161       if (pushdown)
162       {
163         (index,leaf) = stack.pop();
164         (type,node,leaf) = update_state(tree,buffer, index);
165         pushdown = false;
166       }
167       if (pushdown)
168       {
169         (index,leaf) = stack.pop();
170         (type,node,leaf) = update_state(tree,buffer, index);
171         pushdown = false;
172       }
173       if (pushdown)
174       {
175         (index,leaf) = stack.pop();
176         (type,node,leaf) = update_state(tree,buffer, index);
177         pushdown = false;
178       }
179       if (pushdown)
180       {
181         (index,leaf) = stack.pop();
182         (type,node,leaf) = update_state(tree,buffer, index);
183         pushdown = false;
184       }
185       if (pushdown)
186       {
187         (index,leaf) = stack.pop();
188         (type,node,leaf) = update_state(tree,buffer, index);
189         pushdown = false;
190       }
191       if (pushdown)
192       {
193         (index,leaf) = stack.pop();
194         (type,node,leaf) = update_state(tree,buffer, index);
195         pushdown = false;
196       }
197       if (pushdown)
198       {
199         (index,leaf) = stack.pop();
200         (type,node,leaf) = update_state(tree,buffer, index);
201         pushdown = false;
202       }
203       if (pushdown)
204       {
205         (index,leaf) = stack.pop();
206         (type,node,leaf) = update_state(tree,buffer, index);
207         pushdown = false;
208       }
209       if (pushdown)
210       {
211         (index,leaf) = stack.pop();
212         (type,node,leaf) = update_state(tree,buffer, index);
213         pushdown = false;
214       }
215       if (pushdown)
216       {
217         (index,leaf) = stack.pop();
218         (type,node,leaf) = update_state(tree,buffer, index);
219         pushdown = false;
220       }
221       if (pushdown)
222       {
223         (index,leaf) = stack.pop();
224         (type,node,leaf) = update_state(tree,buffer, index);
225         pushdown = false;
226       }
227       if (pushdown)
228       {
229         (index,leaf) = stack.pop();
230         (type,node,leaf) = update_state(tree,buffer, index);
231         pushdown = false;
232       }
233       if (pushdown)
234       {
235         (index,leaf) = stack.pop();
236         (type,node,leaf) = update_state(tree,buffer, index);
237         pushdown = false;
238       }
239       if (pushdown)
240       {
241         (index,leaf) = stack.pop();
242         (type,node,leaf) = update_state(tree,buffer, index);
243         pushdown = false;
244       }
245       if (pushdown)
246       {
247         (index,leaf) = stack.pop();
248         (type,node,leaf) = update_state(tree,buffer, index);
249         pushdown = false;
250       }
251       if (pushdown)
252       {
253         (index,leaf) = stack.pop();
254         (type,node,leaf) = update_state(tree,buffer, index);
255         pushdown = false;
256       }
257       if (pushdown)
258       {
259         (index,leaf) = stack.pop();
260         (type,node,leaf) = update_state(tree,buffer, index);
261         pushdown = false;
262       }
263       if (pushdown)
264       {
265         (index,leaf) = stack.pop();
266         (type,node,leaf) = update_state(tree,buffer, index);
267         pushdown = false;
268       }
269       if (pushdown)
270       {
271         (index,leaf) = stack.pop();
272         (type,node,leaf) = update_state(tree,buffer, index);
273         pushdown = false;
274       }
275       if (pushdown)
276       {
277         (index,leaf) = stack.pop();
278         (type,node,leaf) = update_state(tree,buffer, index);
279         pushdown = false;
280       }
281       if (pushdown)
282       {
283         (index,leaf) = stack.pop();
284         (type,node,leaf) = update_state(tree,buffer, index);
285         pushdown = false;
286       }
287       if (pushdown)
288       {
289         (index,leaf) = stack.pop();
290         (type,node,leaf) = update_state(tree,buffer, index);
291         pushdown = false;
292       }
293       if (pushdown)
294       {
295         (index,leaf) = stack.pop();
296         (type,node,leaf) = update_state(tree,buffer, index);
297         pushdown = false;
298       }
299       if (pushdown)
300       {
301         (index,leaf) = stack.pop();
302         (type,node,leaf) = update_state(tree,buffer, index);
303         pushdown = false;
304       }
305       if (pushdown)
306       {
307         (index,leaf) = stack.pop();
308         (type,node,leaf) = update_state(tree,buffer, index);
309         pushdown = false;
310       }
311       if (pushdown)
312       {
313         (index,leaf) = stack.pop();
314         (type,node,leaf) = update_state(tree,buffer, index);
315         pushdown = false;
316       }
317       if (pushdown)
318       {
319         (index,leaf) = stack.pop();
320         (type,node,leaf) = update_state(tree,buffer, index);
321         pushdown = false;
322       }
323       if (pushdown)
324       {
325         (index,leaf) = stack.pop();
326         (type,node,leaf) = update_state(tree,buffer, index);
327         pushdown = false;
328       }
329       if (pushdown)
330       {
331         (index,leaf) = stack.pop();
332         (type,node,leaf) = update_state(tree,buffer, index);
333         pushdown = false;
334       }
335       if (pushdown)
336       {
337         (index,leaf) = stack.pop();
338         (type,node,leaf) = update_state(tree,buffer, index);
339         pushdown = false;
340       }
341       if (pushdown)
342       {
343         (index,leaf) = stack.pop();
344         (type,node,leaf) = update_state(tree,buffer, index);
345         pushdown = false;
346       }
347       if (pushdown)
348       {
349         (index,leaf) = stack.pop();
350         (type,node,leaf) = update_state(tree,buffer, index);
351         pushdown = false;
352       }
353       if (pushdown)
354       {
355         (index,leaf) = stack.pop();
356         (type,node,leaf) = update_state(tree,buffer, index);
357         pushdown = false;
358       }
359       if (pushdown)
360       {
361         (index,leaf) = stack.pop();
362         (type,node,leaf) = update_state(tree,buffer, index);
363         pushdown = false;
364       }
365       if (pushdown)
366       {
367         (index,leaf) = stack.pop();
368         (type,node,leaf) = update_state(tree,buffer, index);
369         pushdown = false;
370       }
371       if (pushdown)
372       {
373         (index,leaf) = stack.pop();
374         (type,node,leaf) = update_state(tree,buffer, index);
375         pushdown = false;
376       }
377       if (pushdown)
378       {
379         (index,leaf) = stack.pop();
380         (type,node,leaf) = update_state(tree,buffer, index);
381         pushdown = false;
382       }
383       if (pushdown)
384       {
385         (index,leaf) = stack.pop();
386         (type,node,leaf) = update_state(tree,buffer, index);
387         pushdown = false;
388       }
389       if (pushdown)
390       {
391         (index,leaf) = stack.pop();
392         (type,node,leaf) = update_state(tree,buffer, index);
393         pushdown = false;
394       }
395       if (pushdown)
396       {
397         (index,leaf) = stack.pop();
398         (type,node,leaf) = update_state(tree,buffer, index);
399         pushdown = false;
400       }
401       if (pushdown)
402       {
403         (index,leaf) = stack.pop();
404         (type,node,leaf) = update_state(tree,buffer, index);
405         pushdown = false;
406       }
407       if (pushdown)
408       {
409         (index,leaf) = stack.pop();
410         (type,node,leaf) = update_state(tree,buffer, index);
411         pushdown = false;
412       }
413       if (pushdown)
414       {
415         (index,leaf) = stack.pop();
416         (type,node,leaf) = update_state(tree,buffer, index);
417         pushdown = false;
418       }
419       if (pushdown)
420       {
421         (index,leaf) = stack.pop();
422         (type,node,leaf) = update_state(tree,buffer, index);
423         pushdown = false;
424       }
425       if (pushdown)
426       {
427         (index,leaf) = stack.pop();
428         (type,node,leaf) = update_state(tree,buffer, index);
429         pushdown = false;
430       }
431       if (pushdown)
432       {
433         (index,leaf) = stack.pop();
434         (type,node,leaf) = update_state(tree,buffer, index);
435         pushdown = false;
436       }
437       if (pushdown)
438       {
439         (index,leaf) = stack.pop();
440         (type,node,leaf) = update_state(tree,buffer, index);
441         pushdown = false;
442       }
443       if (pushdown)
444       {
445         (index,leaf) = stack.pop();
446         (type,node,leaf) = update_state(tree,buffer, index);
447         pushdown = false;
448       }
449       if (pushdown)
450       {
451         (index,leaf) = stack.pop();
452         (type,node,leaf) = update_state(tree,buffer, index);
453         pushdown = false;
454       }
455       if (pushdown)
456       {
457         (index,leaf) = stack.pop();
458         (type,node,leaf) = update_state(tree,buffer, index);
459         pushdown = false;
460       }
461       if (pushdown)
462       {
463         (index,leaf) = stack.pop();
464         (type,node,leaf) = update_state(tree,buffer, index);
465         pushdown = false;
466       }
467       if (pushdown)
468       {
469         (index,leaf) = stack.pop();
470         (type,node,leaf) = update_state(tree,buffer, index);
471         pushdown = false;
472       }
473       if (pushdown)
474       {
475         (index,leaf) = stack.pop();
476         (type,node,leaf) = update_state(tree,buffer, index);
477         pushdown = false;
478       }
479       if (pushdown)
480       {
481         (index,leaf) = stack.pop();
482         (type,node,leaf) = update_state(tree,buffer, index);
483         pushdown = false;
484       }
485       if (pushdown)
486       {
487         (index,leaf) = stack.pop();
488         (type,node,leaf) = update_state(tree,buffer, index);
489         pushdown = false;
490       }
491       if (pushdown)
492       {
493         (index,leaf) = stack.pop();
494         (type,node,leaf) = update_state(tree,buffer, index);
495         pushdown = false;
496       }
497       if (pushdown)
498       {
499         (index,leaf) = stack.pop();
500         (type,node,leaf) = update_state(tree,buffer, index);
501         pushdown = false;
502       }
503       if (pushdown)
504       {
505         (index,leaf) = stack.pop();
506         (type,node,leaf) = update_state(tree,buffer, index);
507         pushdown = false;
508       }
509       if (pushdown)
510       {
511         (index,leaf) = stack.pop();
512         (type,node,leaf) = update_state(tree,buffer, index);
513         pushdown = false;
514       }
515       if (pushdown)
516       {
517         (index,leaf) = stack.pop();
518         (type,node,leaf) = update_state(tree,buffer, index);
519         pushdown = false;
520       }
521       if (pushdown)
522       {
523         (index,leaf) = stack.pop();
524         (type,node,leaf) = update_state(tree,buffer, index);
525         pushdown = false;
526       }
527       if (pushdown)
528       {
529         (index,leaf) = stack.pop();
530         (type,node,leaf) = update_state(tree,buffer, index);
531         pushdown = false;
532       }
533       if (pushdown)
534       {
535         (index,leaf) = stack.pop();
536         (type,node,leaf) = update_state(tree,buffer, index);
537         pushdown = false;
538       }
539       if (pushdown)
540       {
541         (index,leaf) = stack.pop();
542         (type,node,leaf) = update_state(tree,buffer, index);
543         pushdown = false;
544       }
545       if (pushdown)
546       {
547         (index,leaf) = stack.pop();
548         (type,node,leaf) = update_state(tree,buffer, index);
549         pushdown = false;
550       }
551       if (pushdown)
552       {
553         (index,leaf) = stack.pop();
554         (type,node,leaf) = update_state(tree,buffer, index);
555         pushdown = false;
556       }
557       if (pushdown)
558       {
559         (index,leaf) = stack.pop();
560         (type,node,leaf) = update_state(tree,buffer, index);
561         pushdown = false;
562       }
563       if (pushdown)
564       {
565         (index,leaf) = stack.pop();
566         (type,node,leaf) = update_state(tree,buffer, index);
567         pushdown = false;
568       }
569       if (pushdown)
570       {
571         (index,leaf) = stack.pop();
572         (type,node,leaf) = update_state(tree,buffer, index);
573         pushdown = false;
574       }
575       if (pushdown)
576       {
577         (index,leaf) = stack.pop();
578         (type,node,leaf) = update_state(tree,buffer, index);
579         pushdown = false;
580       }
581       if (pushdown)
582       {
583         (index,leaf) = stack.pop();
584         (type,node,leaf) = update_state(tree,buffer, index);
585         pushdown = false;
586       }
587       if (pushdown)
588       {
589         (index,leaf) = stack.pop();
590         (type,node,leaf) = update_state(tree,buffer, index);
591         pushdown = false;
592       }
593       if (pushdown)
594       {
595         (index,leaf) = stack.pop();
596         (type,node,leaf) = update_state(tree,buffer, index);
597         pushdown = false;
598       }
599       if (pushdown)
600       {
601         (index,leaf) = stack.pop();
602         (type,node,leaf) = update_state(tree,buffer, index);
603         pushdown = false;
604       }
605       if (pushdown)
606       {
607         (index,leaf) = stack.pop();
608         (type,node,leaf) = update_state(tree,buffer, index);
609         pushdown = false;
610       }
611       if (pushdown)
612       {
613         (index,leaf) = stack.pop();
614         (type,node,leaf) = update_state(tree,buffer, index);
615         pushdown = false;
616       }
617       if (pushdown)
618       {
619         (index,leaf) = stack.pop();
620         (type,node,leaf) = update_state(tree,buffer, index);
621         pushdown = false;
622       }
623       if (pushdown)
624       {
625         (index,leaf) = stack.pop();
626         (type,node,leaf) = update_state(tree,buffer, index);
627         pushdown = false;
628       }
629       if (pushdown)
630       {
631         (index,leaf) = stack.pop();
632         (type,node,leaf) = update_state(tree,buffer, index);
633         pushdown = false;
634       }
635       if (pushdown)
636       {
637         (index,leaf) = stack.pop();
638         (type,node,leaf) = update_state(tree,buffer, index);
639         pushdown = false;
640       }
641       if (pushdown)
642       {
643         (index,leaf) = stack.pop();
644         (type,node,leaf) = update_state(tree,buffer, index);
645         pushdown = false;
646       }
647       if (pushdown)
648       {
649         (index,leaf) = stack.pop();
650         (type,node,leaf) = update_state(tree,buffer, index);
651         pushdown = false;
652       }
653       if (pushdown)
654       {
655         (index,leaf) = stack.pop();
656         (type,node,leaf) = update_state(tree,buffer, index);
657         pushdown = false;
658       }
659       if (pushdown)
660       {
661         (index,leaf) = stack.pop();
662         (type,node,leaf) = update_state(tree,buffer, index);
663         pushdown = false;
664       }
665       if (pushdown)
666       {
667         (index,leaf) = stack.pop();
668         (type,node,leaf) = update_state(tree,buffer, index);
669         pushdown = false;
670       }
671       if (pushdown)
672       {
673         (index,leaf) = stack.pop();
674         (type,node,leaf) = update_state(tree,buffer, index);
675         pushdown = false;
676       }
677       if (pushdown)
678       {
679         (index,leaf) = stack.pop();
680         (type,node,leaf) = update_state(tree,buffer, index);
681         pushdown = false;
682       }
683       if (pushdown)
684       {
685         (index,leaf) = stack.pop();
686         (type,node,leaf) = update_state(tree,buffer, index);
687         pushdown = false;
688       }
689       if (pushdown)
690       {
691         (index,leaf) = stack.pop();
692         (type,node,leaf) = update_state(tree,buffer, index);
693         pushdown = false;
694       }
695       if (pushdown)
696       {
697         (index,leaf) = stack.pop();
698         (type,node,leaf) = update_state(tree,buffer, index);
699         pushdown = false;
700       }
701       if (pushdown)
702       {
703         (index,leaf) = stack.pop();
704         (type,node,leaf) = update_state(tree,buffer, index);
705         pushdown = false;
706       }
707       if (pushdown)
708       {
709         (index,leaf) = stack.pop();
710         (type,node,leaf) = update_state(tree,buffer, index);
711         pushdown = false;
712       }
713       if (pushdown)
714       {
715         (index,leaf) = stack.pop();
716         (type,node,leaf) = update_state(tree,buffer, index);
717         pushdown = false;
718       }
719       if (pushdown)
720       {
721         (index,leaf) = stack.pop();
722         (type,node,leaf) = update_state(tree,buffer, index);
723         pushdown = false;
724       }
725       if (pushdown)
726       {
727         (index,leaf) = stack.pop();
728         (type,node,leaf) = update_state(tree,buffer, index);
729         pushdown = false;
730       }
731       if (pushdown)
732       {
733         (index,leaf) = stack.pop();
734         (type,node,leaf) = update_state(tree,buffer, index);
735         pushdown = false;
736       }
737       if (pushdown)
738       {
739         (index,leaf) = stack.pop();
740         (type,node,leaf) = update_state(tree,buffer, index);
741         pushdown = false;
742       }
743       if (pushdown)
744       {
745         (index,leaf) = stack.pop();
746         (type,node,leaf) = update_state(tree,buffer, index);
747         pushdown = false;
748       }
749       if (pushdown)
750       {
751         (index,leaf) = stack.pop();
752         (type,node,leaf) = update_state(tree,buffer, index);
753         pushdown = false;
754       }
755       if (pushdown)
756       {
757         (index,leaf) = stack.pop();
758         (type,node,leaf) = update_state(tree,buffer, index);
759         pushdown = false;
760       }
761       if (pushdown)
762       {
763         (index,leaf) = stack.pop();
764         (type,node,leaf) = update_state(tree,buffer, index);
765         pushdown = false;
766       }
767       if (pushdown)
768       {
769         (index,leaf) = stack.pop();
770         (type,node,leaf) = update_state(tree,buffer, index);
771         pushdown = false;
772       }
773       if (pushdown)
774       {
775         (index,leaf) = stack.pop();
776         (type,node,leaf) = update_state(tree,buffer, index);
777         pushdown = false;
778       }
779       if (pushdown)
780       {
781         (index,leaf) = stack.pop();
782         (type,node,leaf) = update_state(tree,buffer, index);
783         pushdown = false;
784       }
785       if (pushdown)
786       {
787         (index,leaf) = stack.pop();
788         (type,node,leaf) = update_state(tree,buffer, index);
789         pushdown = false;
790       }
791       if (pushdown)
792       {
793         (index,leaf) = stack.pop();
794         (type,node,leaf) = update_state(tree,buffer, index);
795         pushdown = false;
796       }
797       if (pushdown)
798       {
799         (index,leaf) = stack.pop();
800         (type,node,leaf) = update_state(tree,buffer, index);
801         pushdown = false;
802       }
803       if (pushdown)
804       {
805         (index,leaf) = stack.pop();
806         (type,node,leaf) = update_state(tree,buffer, index);
807         pushdown = false;
808       }
809       if (pushdown)
810       {
811         (index,leaf) = stack.pop();
812         (type,node,leaf) = update_state(tree,buffer, index);
813         pushdown = false;
814       }
815       if (pushdown)
816       {
817         (index,leaf) = stack.pop();
818         (type,node,leaf) = update_state(tree,buffer, index);
819         pushdown = false;
820       }
821       if (pushdown)
822       {
823         (index,leaf) = stack.pop();
824         (type,node,leaf) = update_state(tree,buffer, index);
825         pushdown = false;
826       }
827       if (pushdown)
828       {
829         (index,leaf) = stack.pop();
830         (type,node,leaf) = update_state(tree,buffer, index);
831         pushdown = false;
832       }
833       if (pushdown)
834       {
835         (index,leaf) = stack.pop();
836         (type,node,leaf) = update_state(tree,buffer, index);
837         pushdown = false;
838       }
839       if (pushdown)
840       {
841         (index,leaf) = stack.pop();
842         (type,node,leaf) = update_state(tree,buffer, index);
843         pushdown = false;
844       }
845       if (pushdown)
846       {
847         (index,leaf) = stack.pop();
848         (type,node,leaf) = update_state(tree,buffer, index);
849         pushdown = false;
850       }
851       if (pushdown)
852       {
853         (index,leaf) = stack.pop();
854         (type,node,leaf) = update_state(tree,buffer, index);
855         pushdown = false;
856       }
857       if (pushdown)
858       {
859         (index,leaf) = stack.pop();
860         (type,node,leaf) = update_state(tree,buffer, index);
861         pushdown = false;
862       }
863       if (pushdown)
864       {
865         (index,leaf) = stack.pop();
866         (type,node,leaf) = update_state(tree,buffer, index);
867         pushdown = false;
868       }
869       if (pushdown)
870       {
871         (index,leaf) = stack.pop();
872         (type,node,leaf) = update_state(tree,buffer, index);
873         pushdown = false;
874       }
875       if (pushdown)
876       {
877         (index,leaf) = stack.pop();
878         (type,node,leaf) = update_state(tree,buffer, index);
879         pushdown = false;
880       }
881       if (pushdown)
882       {
883         (index,leaf) = stack.pop();
884         (type,node,leaf) = update_state(tree,buffer, index);
885         pushdown = false;
886       }
887       if (pushdown)
888       {
889         (index,leaf) = stack.pop();
890         (type,node,leaf) = update_state(tree,buffer, index);
891         pushdown = false;
892       }
893       if (pushdown)
894       {
895         (index,leaf) = stack.pop();
896         (type,node,leaf) = update_state(tree,buffer, index);
897         pushdown = false;
898       }
899       if (pushdown)
900       {
901         (index,leaf) = stack.pop();
902         (type,node,leaf) = update_state(tree,buffer, index);
903         pushdown = false;
904       }
905       if (pushdown)
906       {
907         (index,leaf) = stack.pop();
908         (type,node,leaf) = update_state(tree,buffer, index);
909         pushdown = false;
910       }
911       if (pushdown)
912       {
913         (index,leaf) = stack.pop();
914         (type,node,leaf) = update_state(tree,buffer, index);
915         pushdown = false;
916       }
917       if (pushdown)
918       {
919         (index,leaf) = stack.pop();
920         (type,node,leaf) = update_state(tree,buffer, index);
921         pushdown = false;
922       }
923       if (pushdown)
924       {
925         (index,leaf) = stack.pop();
926         (type,node,leaf) = update_state(tree,buffer, index);
927         pushdown = false;
928       }
929       if (pushdown)
930       {
931         (index,leaf) = stack.pop();
932         (type,node,leaf) = update_state(tree,buffer, index);
933         pushdown = false;
934       }
935       if (pushdown)
936       {
937         (index,leaf) = stack.pop();
938         (type,node,leaf) = update_state(tree,buffer, index);
939         pushdown = false;
940       }
941       if (pushdown)
942       {
943         (index,leaf) = stack.pop();
944         (type,node,leaf) = update_state(tree,buffer, index);
945         pushdown = false;
946       }
947       if (pushdown)
948       {
949         (index,leaf) = stack.pop();
950         (type,node,leaf) = update_state(tree,buffer, index);
951         pushdown = false;
952       }
953       if (pushdown)
954       {
955         (index,leaf) = stack.pop();
956         (type,node,leaf) = update_state(tree,buffer, index);
957         pushdown = false;
958       }
959       if (pushdown)
960       {
961         (index,leaf) = stack.pop();
962         (type,node,leaf) = update_state(tree,buffer, index);
963         pushdown = false;
964       }
965       if (pushdown)
966       {
967         (index,leaf) = stack.pop();
968         (type,node,leaf) = update_state(tree,buffer, index);
969         pushdown = false;
970       }
971       if (pushdown)
972       {
973         (index,leaf) = stack.pop();
974         (type,node,leaf) = update_state(tree,buffer, index);
975         pushdown = false;
976       }
977       if (pushdown)
978       {
979         (index,leaf) = stack.pop();
980         (type,node,leaf) = update_state(tree,buffer, index);
981         pushdown = false;
982       }
983       if (pushdown)
984       {
985         (index,leaf) = stack.pop();
986         (type,node,leaf) = update_state(tree,buffer, index);
987         pushdown = false;
988       }
989       if (pushdown)
990       {
991         (index,leaf) = stack.pop();
992         (type,node,leaf) = update_state(tree,buffer, index);
993         pushdown = false;
994       }
995       if (pushdown)
996       {
997         (index,leaf) = stack.pop();
998         (type,node,leaf) = update_state(tree,buffer, index);
999         pushdown = false;
1000      }
```

Bibliography

Alla, T., & Laine, S. (2009). Understanding the Efficiency of Ray Traversal on GPUs. Proceedings of the Siggraph/Eurographics Conference on High Performance Graphics 2009, 145-149. ACM. doi:10.1145/1592220.1592222

Castaño, I. (2011). Irradiance Caching – Part 1. Retrieved from <http://the-witness.net/news/2011/07/irradiance-caching-part-1/>

Castaño, I. (2014). Irradiance Caching – Continued. Retrieved from <http://www.ludion.com/castano/blog/2014/07/irradiance-caching-continued/>

Chung, S., Choi, M., Youn, D., & Kim, S. (2018). Comparison of BVH and KD-tree for the GPGPU acceleration on real mobile devices. Retrieved from https://www.researchgate.net/publication/330219705_Comparison_of_BVH_and_KD-tree_for_the_GPGPU_acceleration_on_real_mobile_devices

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. Cambridge, MA: MIT Press.

Curless, Brian (2017). Anti-aliasing and Monte Carlo Path Tracing. CSE 557 Course Lecture Notes. University of Washington. Retrieved from