Performant GPU Pathtracing

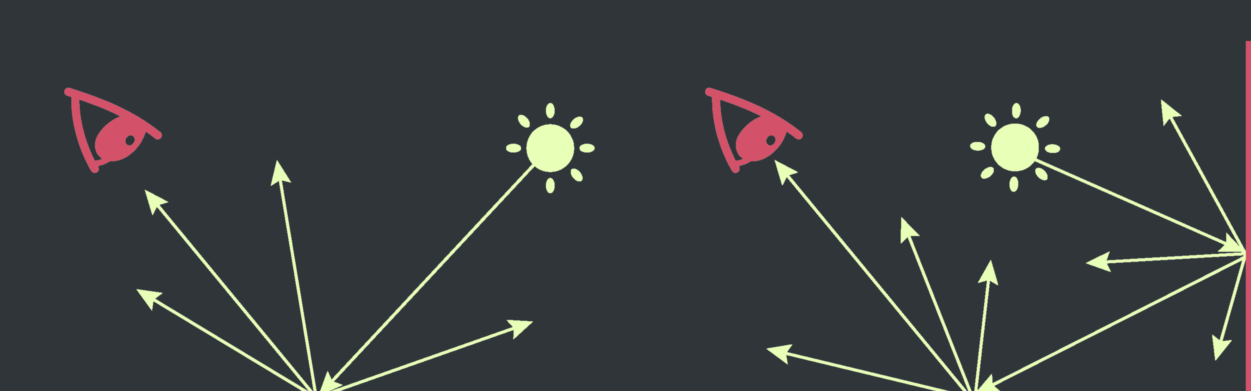
Introduction

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.



direct illumination | indirect 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.

```
(Cost,Side) SAH(p,V,N_L,N_R,N_P)
{
  Voxel V_L,V_R;
  voxel_split(p,V, &V_L, &V_R);
  P_L = SA(V_L)/SA(V);
  P_R = SA(V_R)/SA(V);
  C_L = C(P_L,P_R,N_L+N_P,N_R);
  C_R = C(P_L,P_R,N_L,N_R+N_P);
  return min((C_L, LEFT), (C_R, RIGHT));
}
```

Algorithm .4: GenerateTree.

```
Node gen_node(tree,V,T,depth)
{
  Node n;

  (N_L,N_R,N_P,p,side) = find_plane(tree,V,T);
  n_p = p;

  if(depth ≡ tree_max_depth ∨ p_cost > K_I|T|)
  {
    n_T = T;
    return n;
  }

  Voxel V_L,V_R;
  voxel_split(p,V, &V_L, &V_R);
  (T_R,T_L) = classify(tree,T,p,N_L,N_R,N_P);
  n_left = gen_node(tree,V_L,T_L,depth+1);
  n_right = gen_node(tree,V_R,T_R,depth+1);
  return n;
}
```

Algorithm .3: Classify.

```
(T_R,T_L) classify(tree,T,p,N_L,N_R,N_P)
{
  T_L = null; T_R = null;
  T_L = malloc(N_L+N_P);
  T_R = malloc(N_R+N_P);
  i_TL = i_TR = 0
  for(t < T)
  {
    is_left = is_right = false
    for(j = 0; j < 3; j++)
    {
      t_b = t_j,p_k;

      if(t_b < p_b)
        is_left = true;

      if(t_b > p_b)
        is_right = true;
    }

    if(¬is_left ∧ ¬is_right) // planar
    {
      if(p_Side ≡ RIGHT)
        T_R,i_TR++ = t;
      else
        T_L,i_TL++ = t;
    }
    if(is_left)
      T_L,i_TL++ = t;
    if(is_right)
      T_R,i_TR++ = t;
  }
  return (T_R,T_L);
}
```

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

```
(N_L,N_R,N_P,p,side) find_plane(tree,V,T)
{
  best_cost = ∞; best_p = null; side = null; E = null;
  best_N_L = best_N_R = best_N_P = 0;

  for(k = 0; k < tree->k; k++)
  {
    j = 0;
    E = malloc(2|T|);
    for(t in T)
    {
      Voxel B;
      B = voxel_gen_from_tri(t);
      B = voxel_clip(B, V);
      if(voxel_is_planar(B, k))
      {
        E_j++ = {t, B_min,k, k, PLANAR};
      }
      else
      {
        E_j++ = {t, B_min,k, k, START};
        E_j++ = {t, B_max,k, k, END};
      }
    }
    sort(E); // sort the events by b
    N_L = N_P = 0;
    N_P = |E|;
    for(i = 0; i < |E|)
    {
      p = E_i;
      P_START = P_END = P_PLANAR = 0;

      while(i < |E| ∧ E[i]_b ≡ p_b ∧ E_type ≡ END)
      {P_END++; i++;}

      while(i < |E| ∧ E[i]_b ≡ p_b ∧ E_type ≡ PLANAR)
      {P_PLANAR++; i++;}

      while(i < |E| ∧ E[i]_b ≡ p_b ∧ E_type ≡ START)
      {P_START++; i++;}

      N_P = P_PLANAR; N_R -= P_PLANAR; N_R -= P_END;

      sah_data = SAH(k, p_b, V, N_L, N_R, N_P);

      if(sah_data_Cost < best_cost)
      {
        best_cost = sah_data_Cost;
        best_p = p;
        best_side = sah_data_Side;
        best_N_L = N_L; best_N_R = N_R; best_N_P = N_P;
      }
      N_L += P_PLANAR; N_L += P_START; N_P = 0;
    }
  }
  return (best_N_L,best_N_R,best_N_P,best_p,best_side);
}
```

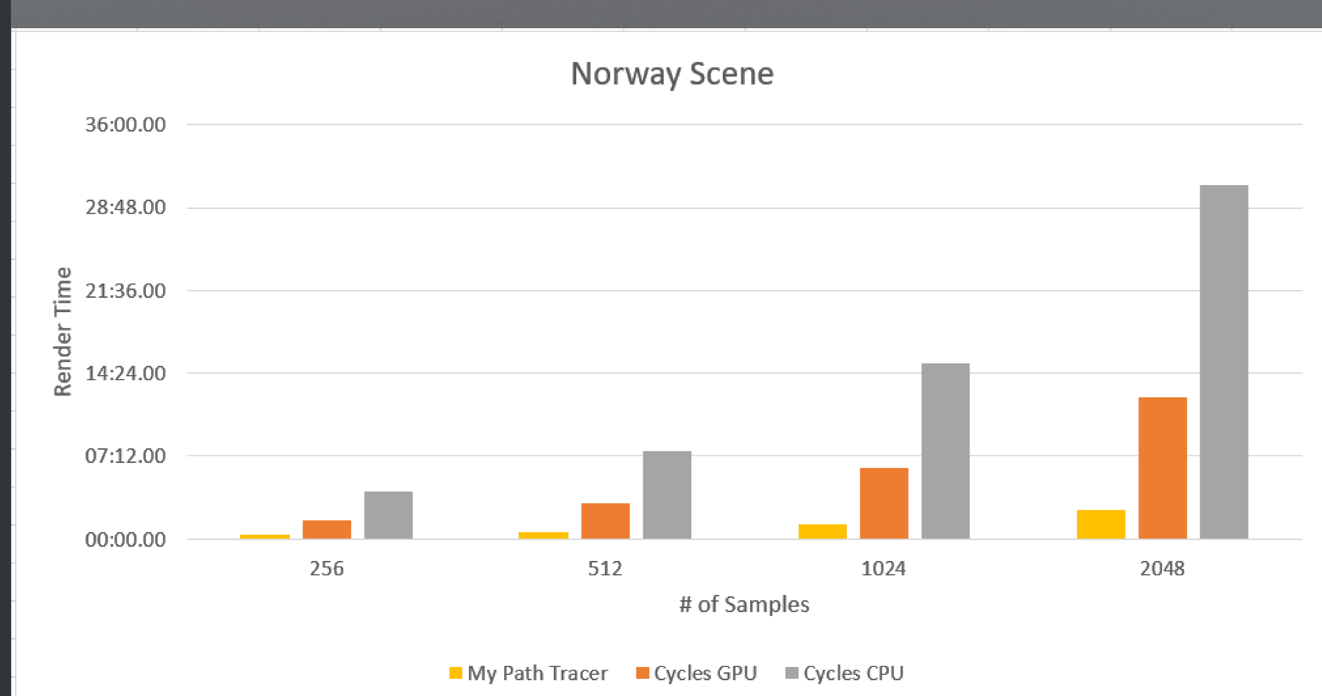
Persistent Threading

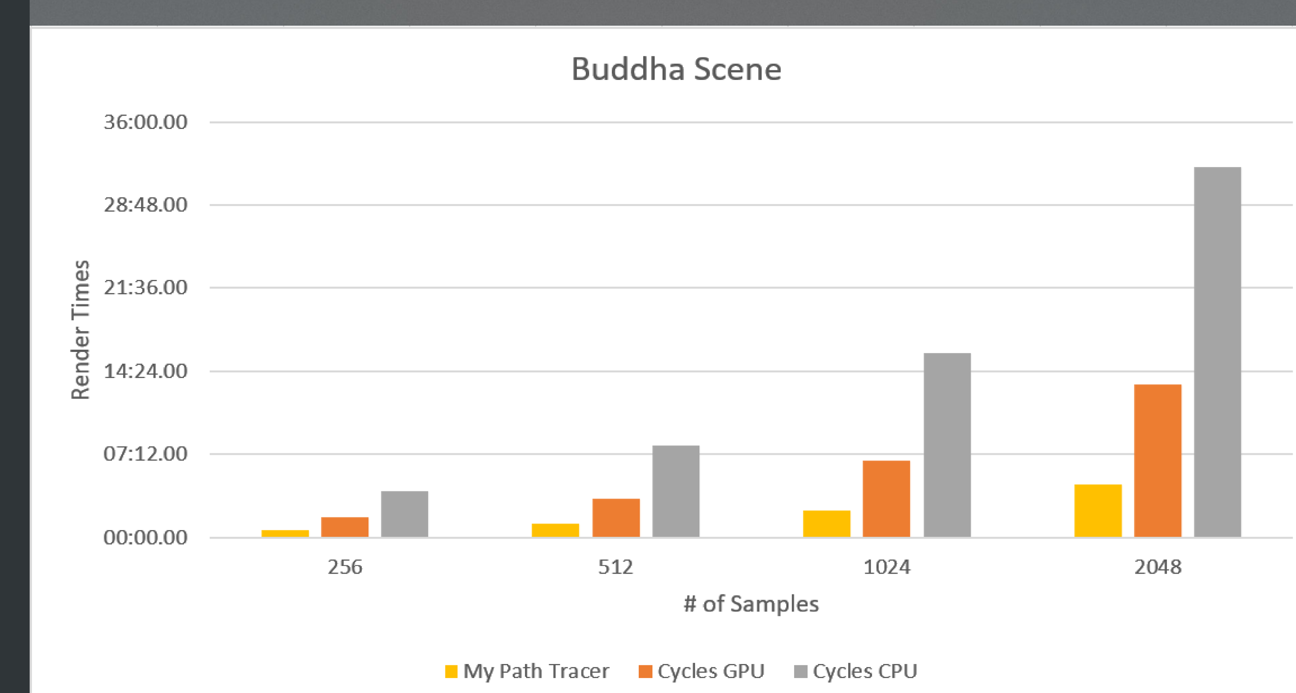
Persistent Threading utilizes the improved performance of warp synchronous execution (a warp is Nvidia's unit for a group of processors that run using SIMT, also known as a Wavefront on AMD hardware). It enforces that only a single task (thread) is assigned to each core of the GPU. This allows for improved 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 liftingfor an operation, but because each core Only has one thread, persistent threadingcan 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

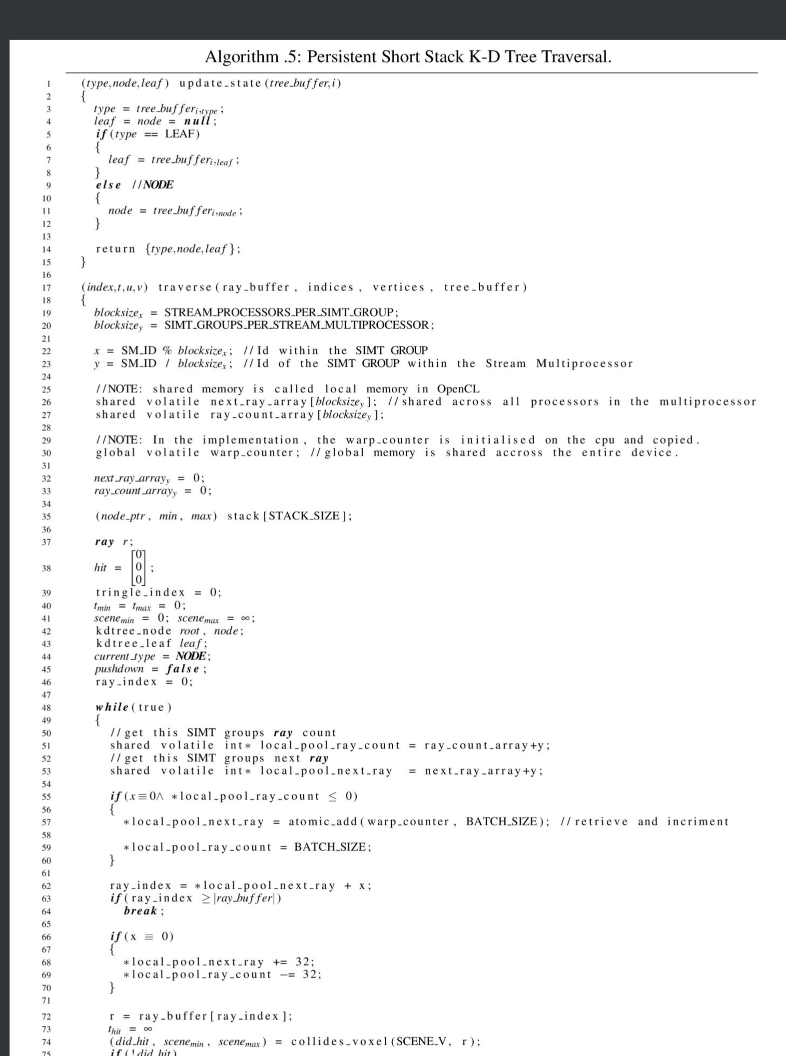
Performant GPU Pathtracing

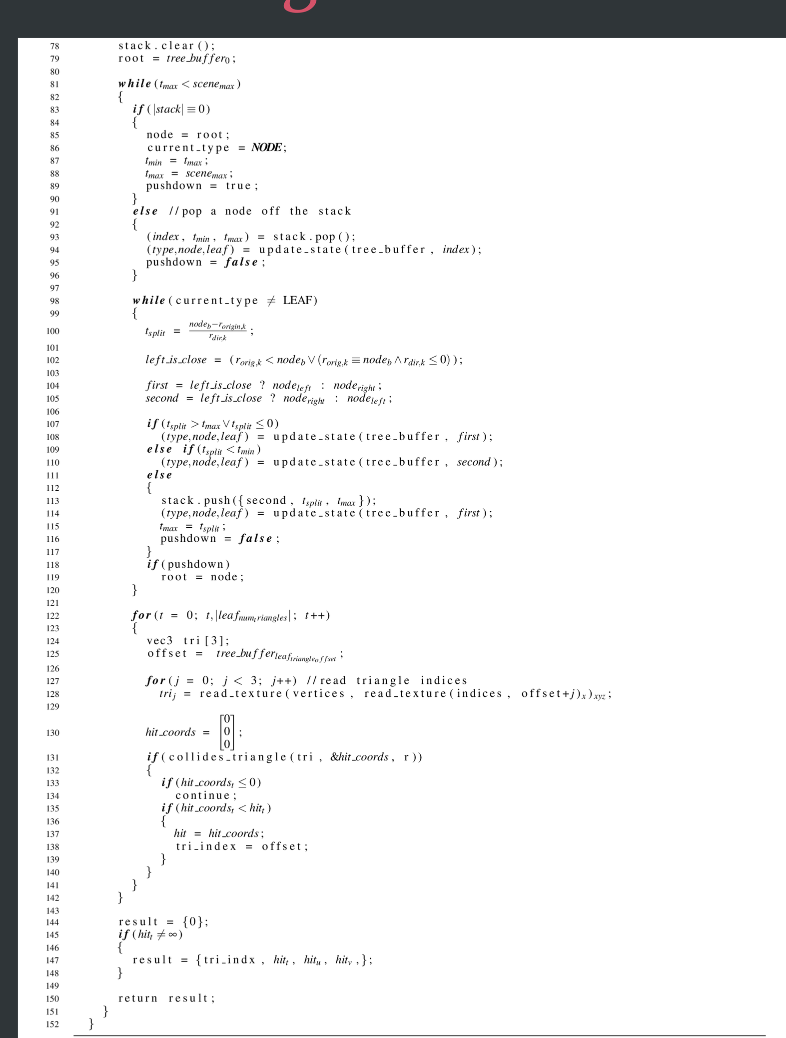
Results





Short Stack k-d Tree Traversal With Peristent Threading





Discussion

The results of the tests were significantly better than expected. I was anticipating that my pathtracer would be only marginally faster, as these pathtracers are industry standard and use the most up-to-date technology. One of the main reasons why the results were so surprising was that I was using a fairly outdated acceleration structure. Most of the industry has moved towards BVHs (Bounding Volume Hierarchies) as they offer better construction and traversal times than k-d trees.

The CPU pathtracer performed the worst by a fairly large margin. My pathtracer was roughly twelve times faster in the Norway scene than Cycles CPU. Results primarily come down to the fact that the test GPU was faster than the test CPU (this is very common in computers). This validates the focus on parallelism. One reason for the speed difference could, potentially, be the complexity of the shaders that Cycles uses. They use PBR-shader pipelines, which are significantly more complicated than my simple diffuse/emission shader set-up. That being said, shading itself is not a performance-draining task and wouldn't account for the extent of the performance gap. More research will have to be conducted to determine why there is such a marked difference.

Future Directions

Sampling Algorithm: Implementing a better sampling algorithm, such as Correlated Multi Jittered Sampling or Sobol Sampling, would dramatically improve image results with minimal performance hits. Time constraints were the limiting factor for the implementation of these algorithms. While they would have made it look better, they wouldn't have made the algorithm itself any faster and considering the focus of render speed, other subjects were prioritized. That said, it would be a quick improvement that would greatly improve appearance with an overall cleaner (less noisy) result.

Textures: Textures are a major feature that, given limited time, could not be implemented. It isn't necessarily difficult to implement, but again, the focus was more on improving performance than appearance. Implementing textures would greatly improve image quality with fairly minimal effort. With textures, it would then be trivial to implement greatly improved shaders, leveraging normal maps, parallax occlusion maps, and specularity/roughness maps.

Physically Based Rendering: Farther off in the distance, implementing a proper Physically Based Rendering (PBR) pipeline would be ideal as there would be greatly improved graphical fidelity. PBR leverages physically accurate lighting models as opposed to Phong lighting, which is a very high-level abstraction of how a surface interacts with light. It results in more realistic images.

Conclusion

The hypothesis proved true. Removing extraneous features and designing a pathtacer for GPGPU computing can boost render times. The program, while limited feature-wise, has use even in its current state. Where someone wants a quick high detail pre-visualization of a more complicated render, this pathtracer would render a much more accurate image than a cheap rasterized Phong imitation. With minimal work on this pathtracer, it could be possible to generate better images that would be comparable to even the best of commercial pathtracers.

Bibliography

Aila, 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: 10.1145/1572769.1572792

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.ludicon.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 courses.cs.washington.edu/courses/cse557/17au/assets/lectures/aa-and-mcpt-4pp.pdf

Driscoll, R. (2009). Better Sampling. Retrieved from http://www.rorydriscoll.com/2009/01/07/better-sampling/

Frolov, V., Vostryakov, K., Kharlamov, A., & Galaktionov, V. (2013). Implementing irradiance cache in a GPU realistic renderer. Transactions on Computational Science XIX, 17-32. Springer. doi: 10.1007/978-3-642-39759-2_2.

Gupta, K., Stuart, J. A., & Owens, J.D. (2012) A study of persistent threads style GPU programming for GPGPU workloads. 2012 Innovative Parallel Computing (InPar), 1-14. IEEE. doi: 10.1109/InPar.2012.6339596.
Harris, M. (2007). Optimizing cuda. SC07: High Performance Computing With CUDA. NVIDIA. Retrieved from http://gpgpu.org/static/sc2007/SC07_CUDA_5_Optimization_Harris.pdf.

Horn, D.R., Sugerman, J., Houston, M., & Hanrahan, P. (2007). Interactive k-D Tree GPU Raytracing. Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games. 167-174. ACM. doi: 10.1145/1230100.1230129

Kernighan, B., & Ritchie, D. M. (2017). The C Programming Language. Upper Saddle River, NJ: Prentice Hall.

Lapere, S. (2016, September 20). Optimised BVH building, faster traversal and intersection kernels and HDR environment lighting. GPU Path Tracing Tutorial, 4. Retrieved from http://raytracey.blogspot.com/2016/09/gpu-path-tracing-tutorial-4-optimised.html.

Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). NVIDIA Tesla: A unified Graphics and Computing Architecture. IEEE Micro (2008), 28 (2), 39-55. doi: 10.1109/MM.2008.31.

MacDonald, J.D, & Booth, K.S. (1990). Heuristics for ray tracing using space subdivision. The Visual Computer (1990), 6 (3), 153-166. Springer-Verlag. https://doi.org/10.1007/BF01911006.

Mikkelsen, P. T. (2009). Triers CUDA Ray Tracing Tutorial. Retrieved from https://viscomp.alexandra.dk/?p=278

Nvidia Corporation (2016). NVIDIA Tesla P100 Whitepaper. Retrieved from https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

Pharr, M., & Fernando, R. (Eds.). (2005). GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Boston, MA: Addison-Wesley Professional.

Pharr, M., Jakob, W., & Humphreys, G. (2017). Physically Based Rendering: From Theory to Implementation. Cambridge, MA: Elsevier.

Popov, S., Günther, J., Seidel, H. P., & Slusallek, P. (2007, September). Stackless kd-tree traversal for high performance GPU ray tracing, Computer Graphics Forum, 26 (3), 415-424. The Eurographics Association and Blackwell Publishing. doi: 10.1111/j.1467-8659.2007.01064.x .

Prunier, J. C., et. al. (2016). Scratchapixel. Retrieved from https://www.scratchapixel.com/

Scarpino, M. (2012). OpenCL in Action: How to Accelerate Graphics and Computations. Shelter Island, NY: Manning Publications.

Spencer, S, et al. (1993). Examining Radiosity. 1993 ACM SIGGRAPH Educator's Slide Set. Retrieved from https://education.siggraph.org/archive/slide-sets#

Vivo, P. G., & Lowe, J. (2018, October 24). The Book of Shaders. Retrieved from https://thebookofshaders.com.

Wald, I., & Havran, V. (2006) On building fast kd-Trees for Ray Tracing, and on doing that in O (N log N). 2006 IEEE Symposium on Interactive Ray Tracing, 61-69. IEEE. doi: 10.1109/RT.2006.280216

Ward, G. (2007). Irradiance caching algorithm, SIGGRAPH '07 Special Interest Group on Computer Graphics and Interactive Techniques Conference, Article 3. New York, NY: ACM. doi: 10.1145/1281500.1281619

Ward, G. (2007). Implementation of irradiance caching in radiance. SIGGRAPH '07 Special Interest Group on Computer Graphics and Interactive Techniques Conference, Article 4. ACM. doi: 10.1145/1281500.1281620

Zlatuška, M., & Havran, V. (2010) "Ray tracing on a GPU with CUDA–comparative study of three algorithms." (2010). Retrieved from https://www.researchgate.net/publication/22974703_Ray_Tracing_on_a_GPU_with_CUDA-comparative_study_of_three_algorithms.