CS248A Final Project
Improved BVH + Multi-Spectral Rendering

Boyu Zhang(bzhang99@stanford.edu) Mingfei Guo(mfguo@stanford.edu)

March 21, 2023

1 The Problem

BVH is a foundational tool in geometric computing, as many relevant algorithms depend on it, including but not limited to collision detection and simulation, ray tracing, and spectrum simulation. So the performance depends heavily on the efficiency of BVH. We try to produce a faster implementation of BVH to benefit many other geometric algorithms.

We build on previous work in BVH and explore which optimizations are useful for improving performance. We have produced a vectorized parallel CPU version of bounding box collision check with ISPC, as well as k-ary BVH tree construction.

In this project, we also want to extend our path tracing project to do the multi-spectral rendering. Our goal is to render prism-like effects. We created a new Spectrum class to support more color channels, assigned a new wavelength variable to the Ray class, and modified the glass BSDF sample function to support wavelength-dependent refraction.

2 Faster BVH

2.1 Approach

2.1.1 K-ary Tree Construction

The basic idea is we build a k-ary BVH tree instead of a binary BVH tree, which can be traversed in parallel using SIMD. We implement two different k-ary tree construction methods, one is simply flattening the binary tree using SAH more than once, and the other is k-means clustering.

2.1.1.1 Flattening Binary Trees Inspired by [0], we first try to use the same construction principle used for binary trees, which is min-max binning with 8 bins to approximate the surface area heuristic (SAH) in our case, to construct 4-ary and 8-ary tree. The same split strategy for the binary case are used. Instead of creating nodes after each split, we split again to produce 4 disjoint subsets (and again to produce 8 disjoint subsets). Thus, the tree contains 4 bounding boxes (or 8 bounding boxes) can be traversed in parallel using SIMD.

The code for flattening binary trees is in bvh.inl and bvh.h. We add new data structures Node4 and Node8 for 4-ary and 8-ary trees respectively. Also, we add new functions SAH4(), SAH8(), hit_queue4(), hit_queue8(), and bucket_split() to support 4-ary and 8-ary trees.

2.1.1.2 K-means We also use the k-means algorithm [0] to subdivide primitives into clusters recursively. K-means clustering is one of the simplest and most popular unsupervised machine learning algorithms, where we calculate the distance between each data point and a centroid to assign it to a cluster.

Initially, k cluster representatives are randomly chosen. Then we iterate though all primitives and assign them to the cluster representatives based on the distance. After forming clusters, we update the

cluster representatives with the mean of all primitives forming the cluster. We repeat this procedure until we reach the maximum number of iterations is reached. By repeating the k-means algorithm recursively on the clusters, we can successfully construct a k-ary tree.

We add a new class Cluster in bvh.h to represent a cluster of primitives, a new node KBVHNode to represent a node in the k-means tree, and a new class KMeans to implement the k-means algorithm. Also, in bvh.inl, we add new functions hit_kmeans(), allocateKaryTree() and constructKaryTree(). We add initialization code in SAH() (but it is commented out) to support k-means tree construction. By the way, we comment out code just to test other approaches to compare the result, the code is actually clean and valid. All the commented parts are for other implementations.

2.1.2 SIMD

We use Intel ISPC to implement easy and portable SIMD code, to parallel bounding box collision and the whole tree traverse process.

2.1.2.1 Parallel Bounding Box Collision We use ISPC to vectorize the bounding box collision check. The code is in ispc_bvh.ispc. We redefined a bunch of data structures in ispc_bvh.ispc to help with the bounding box collision check, including but not limited to Ray, BBox, and Trace. Before we call the ISPC function, we need to convert the data structure to the ISPC version. Then we call bbox_hit(), which launches k tasks to check the intersection of the ray with the k bounding boxes in parallel. And finally, we read the bool result of hit check, and decide if we need to visit this child node or not.

2.1.2.2 Parallel Tree Traverse We also try to parallel the whole tree traverse process, including the closest hit point queries instead of just the bounding box collision check. This approach requires more work in ISPC because except for bounding boxes, we also need other data structures, such as the triangle list, to be vectorized. Thus, we add more Vec3 functions like determinant() Vec3_cross(), and Vec3_dot() and new structures Triangle and Tri_Mesh_Vert for functions like triangle_hit() to support the vectorized triangle intersection check. To make the whole traverse process parallel, every time we visit the child node, we launch a new task to traverse the child node. This part of the code is in ispc_bvh.ispc (commented out). However, since Cardinal3D is highly encapsulated, we have some problems directly accessing the Triangle data inside the BVH class, thus this approach can only be done outside of the BVH class.

2.2 Result

Our results are shown in the table below.

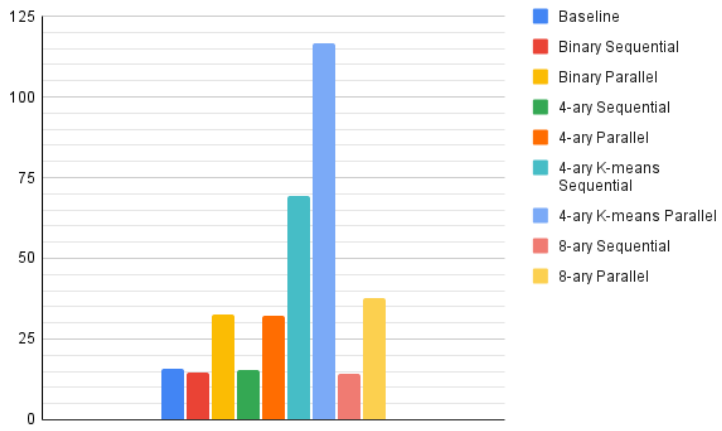

Figure 1: Rendering time (s) of CBox with different approaches.

Unfortunately, the more we try to implement, the worse the performance. Intuitively, we think it is because the number of meshes of CBox is too small so the ISPC loading and converting overhead is too large

3 Multi-Spectral Rendering

3.1 New Spectrum Class

To have a better understanding of how to create the new Spectrum class. I reviewed section 5.1, Spectral Representation, from the book "Physically Based Rendering."[0].

We can see the rainbow effect because light at different wavelengths has different refractive indices when they travel through the glass. Although all the components in a white light enter the glass have the same incoming direction, because they have different refractive indexes, their outgoing direction are all different. This is how a prism can disperse light. Instead of storing RGB three values as a vector like in the old spectrum class, the new spectrum class needs to support more color channels. In our case, we have an array of 60 elements. Each element corresponds to the light intensity at a particular wavelength. The wavelength starts from 400 nm and ends at 700 nm with an interval of 5 nm.

We added "sampleAtLambda()" and "addValueAtLambda()" functions to the new spectrum class so that we can sample or add the light intensity at a particular wavelength. We store our light spectrum discretely, but the wavelength is a random float number between 400 and 700nm, so we had to do linear interpolation in both functions.

We added "Convert2RGB()" and "Convert2Spectrum()" functions to the class so that we can convert spectrum to RGB or convert RGB back to spectrum. To convert a spectrum to RGB, we must first convert the spectrum to XYZ. XYZ value can be computed by integrating the spectrum with the spectral matching curves. These matching curves can be found online. Converting XYZ to RGB is just a change of basis. Converting RGB to spectrum is not easy because there are an infinite number of the spectrum that can match the RGB color. We chose to implement a method that Smits proposed in his paper "An RGB-to-spectrum conversion for reflectances"[0]. We get all the base color spectrum samples value from the book's GitHub (https://github.com/mmp/pbrt-v3/blob/master/).

3.2 Adapt the new Spectrum class in the Pathtracer

The throughput variable in the original Ray class was a Spectrum object. Now each Ray has its own wavelength, so we added wavelength lambda to Ray's member variable and changed the throughput to just a float number (light intensity at a particular wavelength is just a float number).

Now when the camera generates the ray, apart from calculating the ray's direction, we also need to generate a random float between 400nm and 700nm as the ray's wavelength.

The trace_ray function now returns a float instead of a Spectrum object. In the trace_ray function, attenuation and emissive variables from BSDF, env_light's sample and lights' radiance are still the original Spectrum objects, so We use Convert2Spectrum() to convert them to the new spectrum type and sample them at the wavelength of the ray. All other rays generated in the trace_ray function have the same wavelength as the camera ray.

In the end, in the HDR_Image class, when generating the final image, We use New2OldSpectrum() function to convert the new spectrum back to RGB value so that the image can be generated.

3.3 Wavelength-Dependent Refraction

We add the wavelength lambda as an extra parameter in the sample function of BSDF. By searching online, We know we need to use the Sellmeier equation to calculate the refraction index for a ray with a particular wavelength. However, just dividing the wavelength by 400 as the refraction index produces a nice prism effect, so We just keep the conversion simple.

3.4 Result

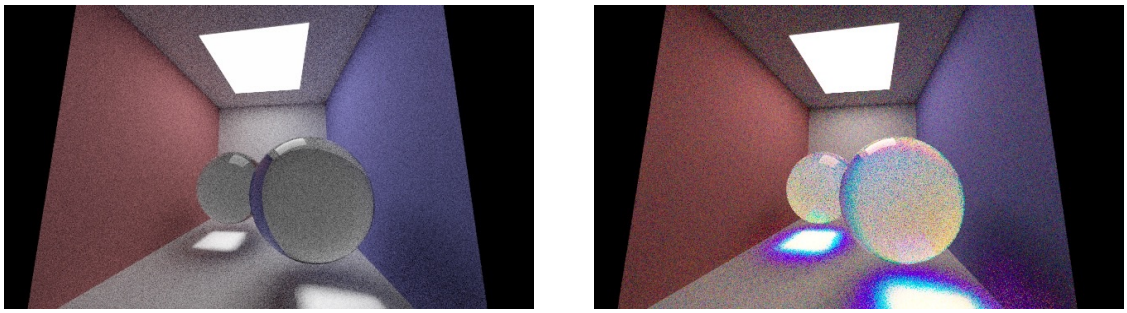

Figure 2: CBox with and without Multi-Spectral Rendering

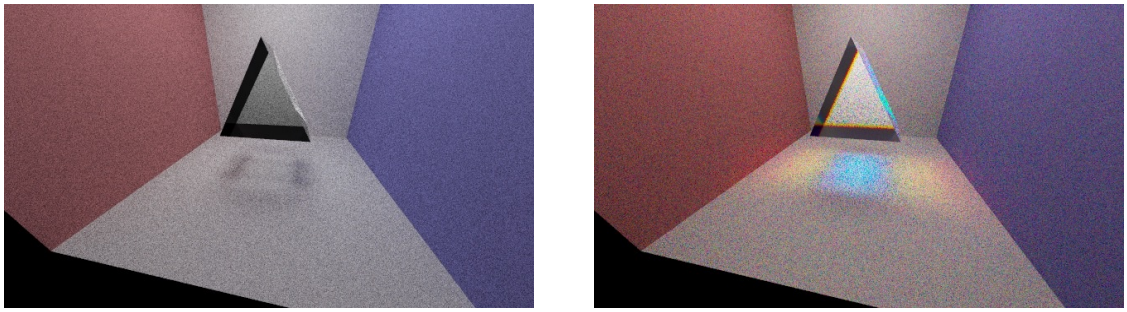

Figure 3: Prism with and without Multi-Spectral Rendering

4 Future Todos

5 Citation

[0] Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays, H. Dammertz and J. Hanika and A. Keller

[1] Parallel BVH Construction using k-means Clustering, Daniel Meister, Jir Bittner

[2] Physically Based Rendering: From Theory to Implementation, Pharr, Matt and Jakob, Wenzel and Humphreys, Greg

[3] An RGB-to-spectrum conversion for reflectances, Smits, B. 1999.