Project Report - Group 100

Bogdan Paramon
5741513

Yair Chizi
5736668

Harald Toth
5710812

BVH

Acceleration Data Structure – Generation and traversal

Implementation of the acceleration data structure is placed in the bvh.cpp file and is slightly extended to main and bvh.h. The bvh.cpp contains multiple helper methods for the two main methods intended to generate – buildRecursive() and traverse – intersectRayWithBVH() the model. Generation: buildRecursive()
I implemented this method by first considering two possible variations of the Axis Aligned Bounded Box, one that uses method computeSpanAABB() to compute the AABB of a span of primitives considering their vertices and a second one that uses the method computeCentroidSpanAABB(), which I defined myself such that it only considers the centroids of the triangles in the span to avoid edge cases while using Sah+Binning (such as primitives that are too elongated and have centroids grouped together leading to inefficient splitting in bins or even faulty behaviour). The next step is defining the base case, thus when the number of primitives gets to a value equal or lower than the specified LeafSize(4) defined in bvh.h a leaf is built with the current primitives using the function buildLeafData(). Additionaly, I have decided to directy code the logic for the buildNumLevels() and buildNumLeaves() in the buildRecursive() method as I believe that it is slightly more efficient than recursing twice again through the nodes. Continuing with the generating method, if the base case is not met, the recursive part of the method takes place, first identifying whether Sah+Binning is active or not. Since I am describing the basic feature, I will stick to the part of the method regarding the BVH implemented with a simple median split on the longest

axis of the AABB using the following methods computeAABBLongestAxis() and splitPrimitivesByMedian(). After the splitting position is returned, the primitives get split, indexes advance, the specific node is built using the function buildNodeData() and the recursion advances on the left partition and the right partition previously established. It is worth noting that the choice of building the number of levels and leaves came with an additional parameter in the definition of the method such that the current level can be tracked.

Traversal: intersectRayWithBVH()

I made use of a stack to implement this method, simulating a DFS. The method checks for nodes that are intersecting with the given ray, carefully reverting the ray to when checking if children intersect it using the method intersectRayWithShape(). When a leaf is found, the primitives stored in the leaf are iterated and upon intersection, tested with the help of intersectRayWithTriangle() method, the flag for being hit turns true and the hit info is updated by the method updateHitInfo(). Finally, the method considers whether an intersection has been found between the ray and the model and accordingly returns the value of the flag.

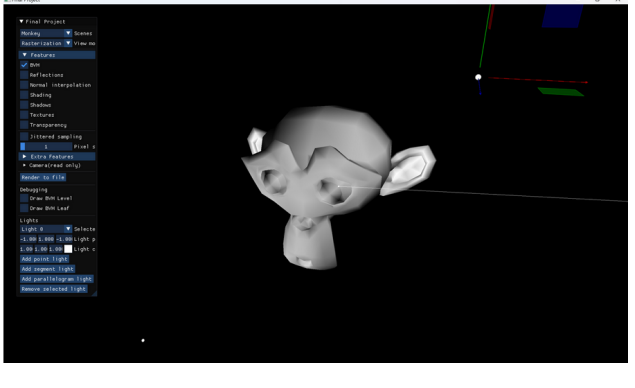


Figure 1: intersectRayWithBVH() correct behaviour can be observed

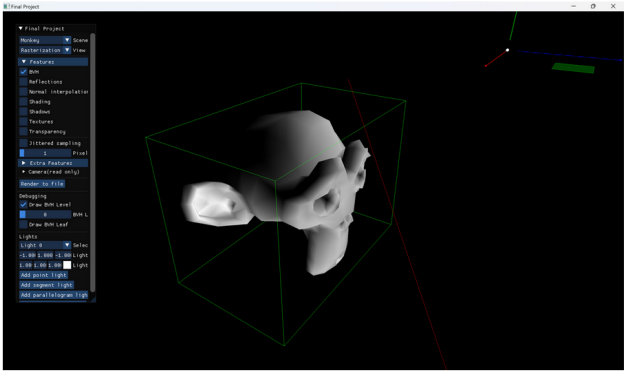


Figure 2: intersectRayWithBVH() correct behaviour when ray intersects AABB, but does not intersect primitives

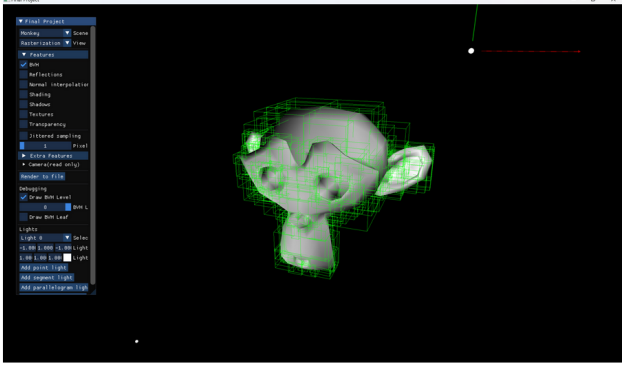


Figure 3: Debug method showing the leaves on level 8

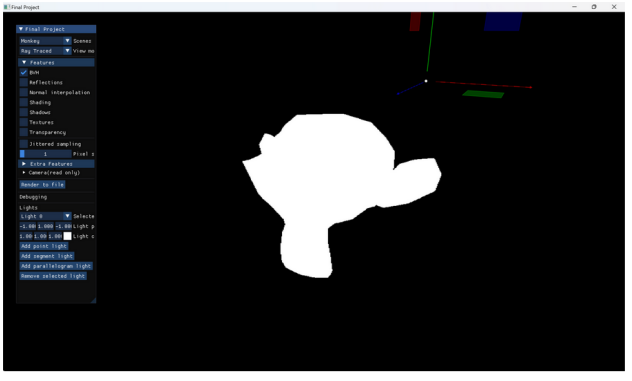


Figure 4: buildRecursive() correct behaviour when using ray tracing

Performance Tests – BVH + SAH

	Cornell Box	Monkey	Dragon
Num triangles	32	968	87k
Time to create (debug)	0.817667ms	35.953ms	1307 ms
Time to render (release)	526 ms	873 ms	19979 ms
BVH levels	4	9	16
BVH leaves	8	256	32768
Sah+binning creation (debug)	8.81592ms	310.589ms	39784.1ms
Sah+binning render (release)	444 ms	668 ms	1045 ms

Figure 5: Table BVH + SAH

SAH+binning as splitting criterion for the BVH

SAH+binning is an alternative to the less efficient method of splitting the primitives by the median. In my implementation, I begin with sorting the primitives along the longest axis of the AABB of centroids provided as parameter than I split the primitives in bins using an efficient way found in a paper published (Wald, 2007). After that, I iterate over the possible partitions (one less than the number of bins) and map the bins to a left and a right partition, to then calculate the AABB of each one, which is needed in order to obtain the total are of the left and right AABB. The areas along with the number of primitives on each side are used to calculate the cost specific to every partition which are constantly compared against a minimal cost. Every time a smaller cost is calculated, the best partition is remembered and in the end the best partition position is returned by offering the number of

primitives in the left partition. The method splitPrimitivesBySAHBin() was implemented in extra.cpp.

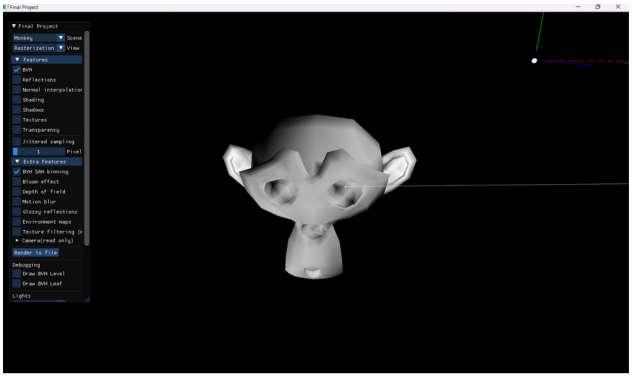


Figure 6: intersectRayWithBVH() correctly functions with SAH+binning

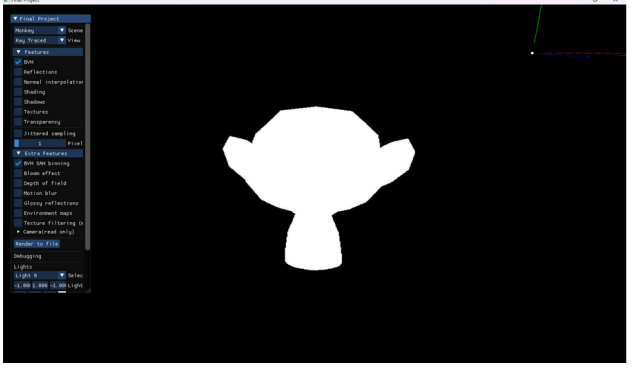


Figure 7: buildRecursive() correctly functions with SAH+binning

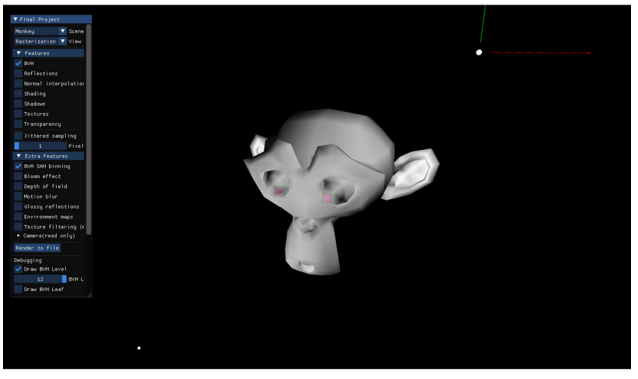


Figure 8: Draw level debug method shows how much more accurate the method is than splitting by median

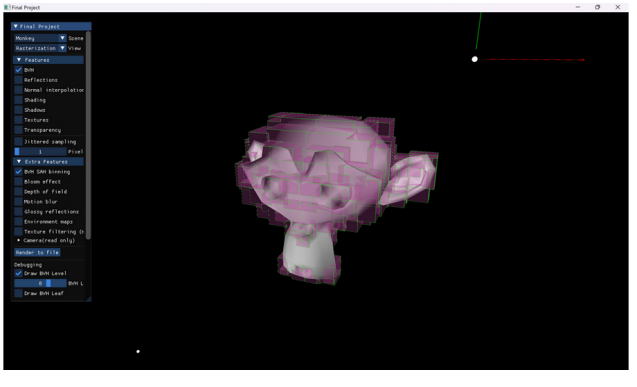


Figure 9: By the following debug method we can observe where areas overlap since the opacity of the color changes, offering an insight on the difference in performance compared to the basic splitting.

Light transparency sampling

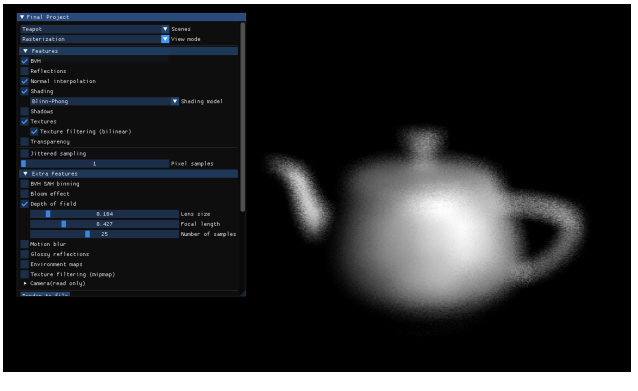
The algorithm suffers from two main problems. Firstly, in a scenario where an opaque object directly obstructs, no light is computed even if reflective surfaces around produce incident secondary rays. Secondly, only the obstruction of thin boundary surfaces contributes to the computation, with no consideration to obstructing object's internal medium.

Depth of field

Description: The method iterates through all the pixels on the screen and calculates the position of the focal point relative to each pixel. It then generates several rays that start from a random location inside the pixel and end at the focal point. It calculates the color that results from the rays and averages their sum, which will be the final color of the pixel.

Visual debug: There are three sliders in the depth of field section of the GUI: focal length, lens size, and number of samples. Focal length adjusts the distance from the camera to the focal point. Lens size modifies the maximum offset from the center of each pixel to the starting position of each ray. The number of samples specifies how many rays should be generated.

Location: The method that implements depth of field is called renderImageWithDepthOfField and is located in the extra.cpp file. The only other changes related to this feature were made in main.cpp and common.h for the sliders in the GUI.



Environment maps

The setup for this method consists of adding 6 pictures to the data directory, which form a 360 degree view of the mapped environment. To access them I have created an array in the scene.h file, which I later used in my implementation of sampleEnvironmentMap(), defined in extra.cpp. The method defines a cube with coordinates from -1 to 1 on which we aim to propagate the chosen environment. We do that by shooting a ray from the origin and getting the intersection with the cube to then map the coordinates of the

intersection point to the texture coordinates u, v of the specific image that we are on (which depends on the face the intersection point is on). Once I have all the information I map the coordinates and access the proper picture to get the final result and I return the obtained vector. I used as a source of knowledge the Wikipedia article on Cube Mapping (Wikipedia, 2023)+.

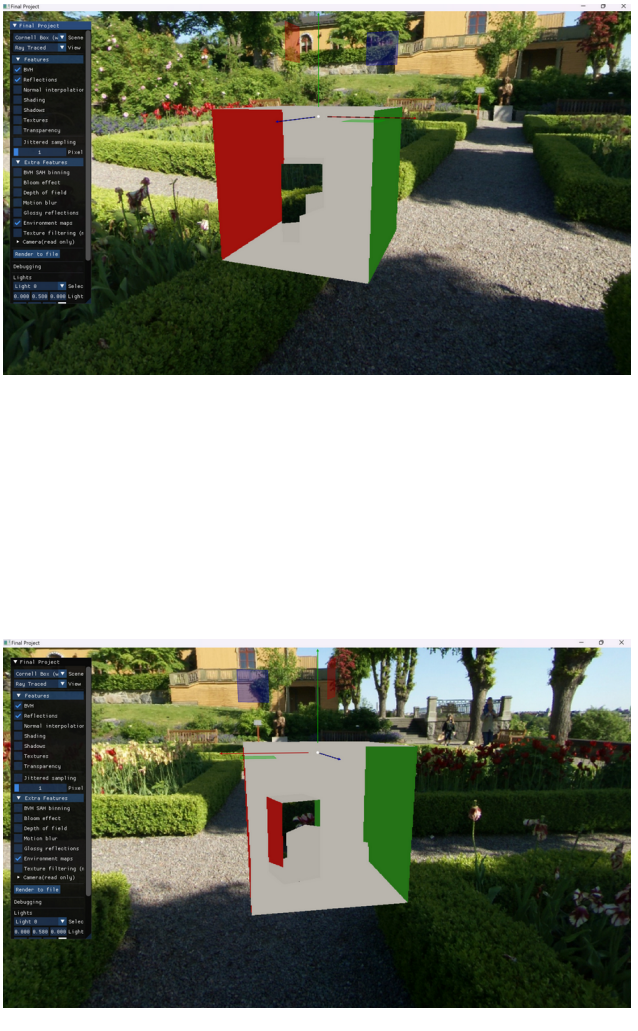


Figure 10: The above two pictures prove the correctness of the environmental mapping as the mirror reflects proper different positions of the picture opposite.

Figure 11: Corners where images meet are not visible.

References

Wald, I. (2007). *On fast construction of sah-based bounding volume hierarchies*. https://www.sci.utah.edu/~wald/Publications/2007/ParallelBVHBuild/fastbuild.pdf

Wikipedia. (2023). *Cube mapping*. https://en.wikipedia.org/wiki/Cube_mapping