Universiteit Utrecht

ACCELERATION STRUCTURES

Bounding Volume Hierarchies

November 11, 2018

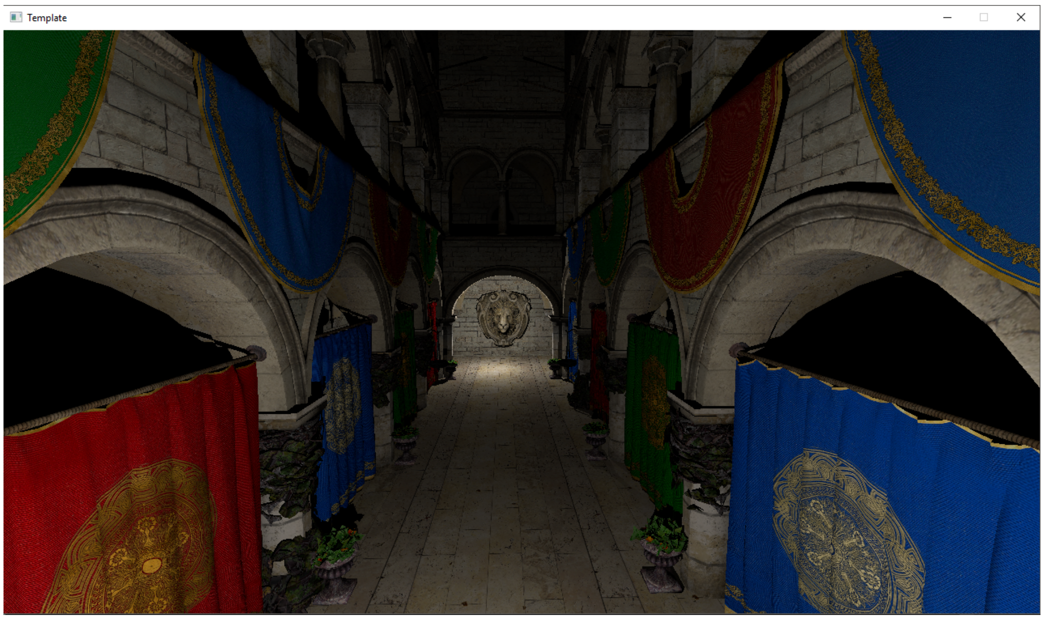
Student:
Mathijs Molenaar
5958970

Claudio Freda
6030769

Teacher:
Jacco Bikker

Course:
Advanced Graphics

Course code:
INFOMAGR



Abstract

We improve on the Whitted-style GPU raytracer built for the first assignment by introducing bounded volume hierarchies that empower it to draw a large amounts of triangles per frame.

1 Introduction

For the first assignment we wrote an OpenCL Whitted style ray tracer that supported triangles, spheres and planes. Even with the OpenCL backend, the performance was not very good. In this paper we present the performance improvements we made to our previous work. By adding various Bounding Volume Hierarchy (BVH) techniques we have been able to get a very large speed up.

A bounding volume hierarchy is a, in our case binary, tree of bounding volumes. As in most literature, we use Axis Aligned Bounding Boxes (AABB) as bounding volumes. AABBs provide a decent fit in most cases but their main advantage is that they are easy to operate and have low intersection cost.

To still be able to support a full range of animations we have implemented multiple BVH builders and various techniques revolving around BVHs.

2 Spatial Split Bounding Volume Hierarchy

For building sub-bvhs for large and complex static geometry, the standard binning approach is not enough. In this context we are targeting speed of traversal and disregarding build speed. For this type of BVH construction, NVIDIA's SBVH approach should give very good results.

The basic idea works like binning, but instead of neatly dividing each triangle in each bin based on its centroid, we insert a reference to the triangle in every intersecting bin and clip it to that bin. Standard binning (a.k.a. object binning in the paper) is still used. Then among all of the possible object splits and spatial splits, the one with the smallest cost is chosen.

This effectively causes some triangle duplications, so at the end of the operation we reconstruct the mesh's triangle buffer with several additional duplicated triangles. This is necessary because the BVH nodes access the triangle buffer sequentially and, the objective being speed of execution, it would not be helpful to try out other access patterns.

Additionally, at each split, instead of testing the axis on which the current node extends the most, we test every axis to ensure that the absolute best split is found.

2.1 Clipping algorithm

The choice of clipping algorithm is important for the quality of the generated BVH. We don't actually need a full clipping algorithm, what we want is a bounding box for the triangle clipped inside one considered bin.

One first method would be to simply clip the triangle's bounds to the bin's bounds. This will generate a bound that in many cases is larger than

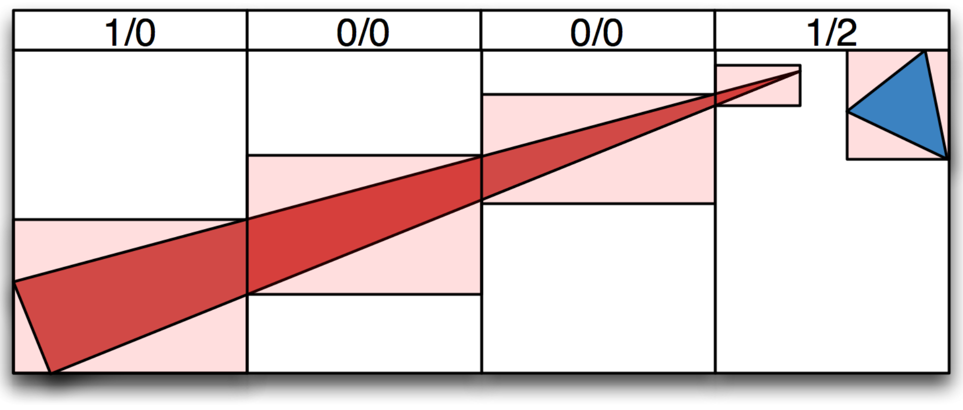


Figure 1: Triangle clipping. Image courtesy of NVidia.

the clipped triangle's actual bounds, but in our experience it still gives good results.

Obviously, to get the best results an actual clipping algorithm has to be used, and then a bounding box for the clipped polygon has to be found. We decided on the Sutherland-Hodgman plane/polygon clipping algorithm, which was slightly simplified due to us having to deal with axis-aligned planes. The triangle was clipped against all 6 planes of the bin's bounding box, and the resulting polygon was used to generate a bounding box.

The Sutherland-Hodgman gives the best results, but it is slower. For this reason we have provided `HI_QUALITY_CLIPPING` as a compile-time define that can be commented to disable it.

2.2 Node Unsplitting

We have implemented node unsplitting as detailed in the NVIDIA paper. When considering a split, each split the SAH heuristic is calculated for the case of the node being actually split, or the node being kept only in the left or right parts of the split. The alternative resulting in the lowest cost is chosen. This allows us to have a hybrid between an object and a spatial split where appropriate.

2.3 Restricting Aggressive Splitting

Splitting is costly memory-wise. If one were to care about memory consumption, there has to be a way to limit splitting only when actually necessary. A good criterion for whether we should consider a split is to consider the corresponding object split, take the surface area of the intersection of the left and right volumes, and divide it by the surface area of the current volume. If this is greater than a chosen value $\alpha \in [0, 1]$, then the spatial split is not considered, even if it would result in a lower SAH cost.

An α value of 0 corresponds to a full SBVH, while an alpha value of 1 corresponds to a standard BVH without spatial splitting.

2.4 Termination criterion

A termination criterion has to be chosen. The NVIDIA paper doesn't specify a single criterion, but suggests to use the SAH cost.

Simply interrupting the algorithm when a node has less than a certain number of nodes does not guarantee that the algorithm will end, because spatial splits make it so that triangles can be duplicated in both children.

A good way to limit the splitting is to consider the full SAH cost of splitting the node and compare it to the cost of a leaf node. This usually gives good results, but the intersection and traversal costs have to be chosen carefully.

To avoid BVH going too deep in case of a bad cost estimation, we limit the BVH depth to $log_2(numTriangles)$ where $numTriangles$ is the original triangle count. A BVH deeper than this is seldom useful.

2.4.1 SAH criterion detailed

Let us remind the SAH cost definition.

$$Cost_{NODE} = K_T + K_I \cdot \frac{SA(V_L) \cdot N_L + SA(V_R) \cdot N_R}{SA(V)}$$

$$Cost_{LEAF} = K_I \cdot N$$

Where SA is the surface area function, V is the current node volume, V_R and V_L are the child node volumes, likewise for N, N_L, and N_R, and K_I and K_T are respectively the cost of intersection of a primitive and the cost of traversal of a node.

We can already see that $SA(V_L) \cdot N_L + SA(V_R) \cdot N_R$ is already the simplified SAH statistic that we use to compare splits. We can thus derive a condition for the simplified SAH which is true when the cost of splitting the node is higher than the cost of the node itself.

$$K_I \cdot N > K_T + K_I \cdot \frac{SAH_{simp}}{SA(V)} \implies SAH_{simp} < (N - \frac{K_T}{K_I}) \cdot SA(V)$$

We simply calculate this $(N - \frac{K_T}{K_I}) \cdot SA(V)$ value before each partition. Any split whose cost function is greater is automatically discarded.

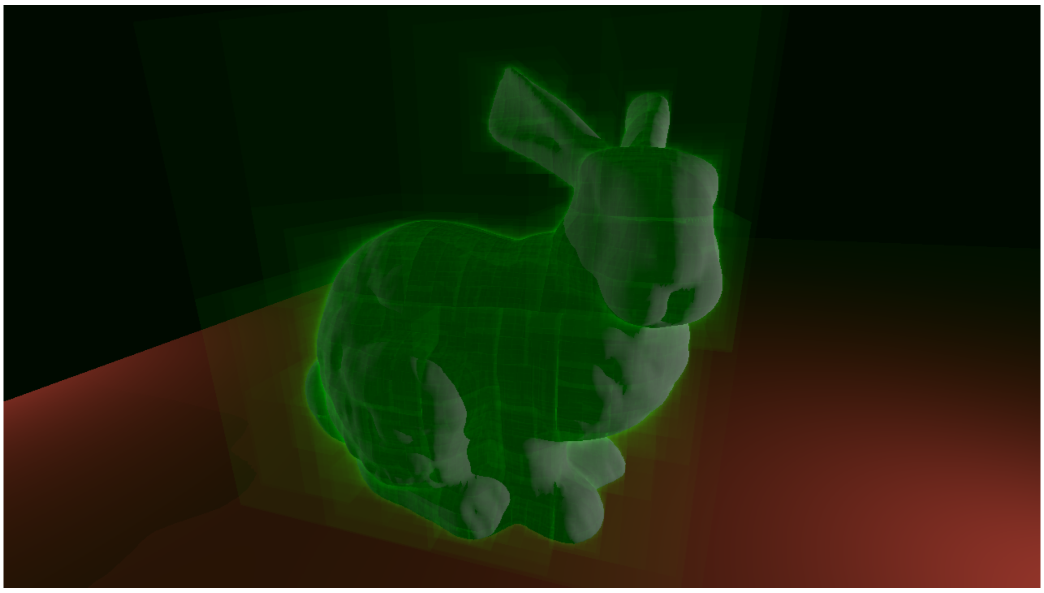


Figure 2: The Stanford bunny model with a full-featured SBVH and node visualisation turned on.

2.5 Disk Cache

The SBVH algorithm is very computationally expensive. This is acceptable because it is not run while ray tracing but only at start up. To reduce the start up time we have added a “disk cache“. After the BVH is generated, we store it and the triangle list into a file. When the program runs the next time, it will look for the file and read the BVH and triangles from there. This removes the cost of calculating the BVH again and is very important for keeping up our development efficiency.

3 Animation

Our ray tracer supports a whole range of animations. Rigid body transformations are provided by a top level bounding volume hierarchy. Deformations can be represented using either a fast, but less accurate, BVH construction using binning. Meshes that do not change their triangle order (and count) may also use refitting. To test our animations, we use a sequence of mesh files. However, our system is extensible to other types of mesh generation.

3.1 Top Level Bounding Volume Hierarchy

To support rigid body animations, we have extended our scene class to represent a tree. A scene graph node is returned when adding a mesh to the scene. These can also be used as arguments of this function to create the tree. A top level BVH is created from the meshes in the scene graph. A top level BVH is a BVH that contains other bounding volume hierarchies at the leaf nodes.

Every frame, the ray tracer class (CPU) iteratively traverses the scene graph and constructs the top level leaf nodes. To construct the top level BVH itself we use the heapless implementation of "Fast agglomerative clustering for rendering"[6].

Listing 1: Heapless Fast Agglomerative Clustering

```
list = getLeafs(sceneGraph);

nodeA = getAny(list);
nodeB = findBestMatch(list, nodeA)
while (list.size() >= 2)
{
    nodeC = findBestMatch(list, nodeB);
    if (nodeA == nodeC)
    {
        list.remove(nodeA);
        list.remove(nodeB);

        nodeA = mergeNodes(nodeA, nodeB);
        bvh.add(nodeA);
        B = findBestMatch(list, nodeA);
    } else {
        nodeA = nodeB;
        nodeB = nodeC;
    }
}
```

The algorithm stores a list of BVH nodes that need to be processed. Initially, this is the list of all the leaf node. The algorithm iteratively picks a node from the list and tries to find another node for which the combined AABB has the smallest surface. If the first node is also the best match for the found node, they are removed from the list and replaced by a new node: their parent. If their is no optimal match found, the second node will be used as the new main node.

3.2 Binned Construction

To support any kind of deformation, we need a fast BVH construction algorithm. It needs to be fast enough so that the GPU traversal is not bottlenecked by the BVH construction. Therefor we implemented binned BVH construction as described in the paper presented by Ingo Wald [5].

The algorithm tries to build a BVH that minimizes the Surface Area Heuristic (SAH). SAH is a function that tries to approximate the cost of traversing the BVH. Since a full evaluation is very expensive ($O(N^3)$), we use lazy evaluation. The full definition is given below in equation 1. Here, SA(V) is the surface area of V, and K_T and K_I are the cost for a traversal step and a triangle intersection, respectively.

$$Cost(V \longrightarrow \{L, R\}) = K_T + K_I(\frac{SA(V_L)}{SA(V)} N_L + \frac{SA(V_R)}{SA(V)} N_R) \qquad (1)$$

We start with a bounding box that encapsulates the whole mesh. The bounding box is divided into evenly spaced bins in the direction of the largest axis. Triangles are binned based on their bounding box's centre. We then consider each split and calculate the bounding box of the triangles to the left and right of that split. We calculate the approximate SAH as follows: $Cost = A_L * N_L + A_R * N_R$ (where A is the bounding box and N is number of triangles). Finally, we perform the actual split on the one that has the lowest SAH.

3.3 Refitting

Although binned construction is fast, it can result in some very bad bounding volume hierarchies. In our helicopter scene ($\sim$ 36.5k triangles) for example, the binned BVH is 10 to 20 times slower than a BVH constructed with the SBVH builder. For some animated meshes we can use an alternative method to rebuilding the BVH every frame.

Refitting relies on a high quality BVH builder to generate the BVH for the first frame of animation. But when we switch to a next frame we do not rebuild the BVH. Instead, we walk through the tree from the leafs up and adjust the nodes bounding boxes to fit the new triangles positions. Accordingly, this method only works when the triangle count does not change. To actually get good performance out of refitting, the mesh should not change too much. If vertices move too much, the resulting BVH will be correct, but very inefficient.

4 Traversal

On the GPU, the top level BVH is traversed in a least distance order. Both the top- and sub level BVHs are traversed using a stack based approach. To combine both BVHs traversels, we use a double while loop as used in the paper "Understanding the Efficiency of Ray Traversal on GPUs"[1]. The outer loop continues until we have processed the whole scene (or on the first triangle hit for shadow rays). Inside the loop we first traverse the top level BVH until we find a leaf node. When we do, we transform the ray based on the node's inverse transformation (which is calculated during BVH construction).

Afterwards, we use nested while loops to traverse the sub BVH and test the triangles at leaf nodes. Both the sub BVH and top level BVH are traversed in a distance based order and BVH nodes that are further away than the currently closest hit triangle are discarded.

Listing 2: GPU Ray Traversal Pseudo Code

```
float closestT = INFINITY;// (or line length)
while (true)
{
  SubBvhNode subNode;
  Ray transformedRay;

  while (!topLvlStack.empty())
  {
    TopBvhNode topNode = topLvlStack.pop();
    if (!intersect(ray, topNode, closestT))
      continue;

    if (topNode.isLeaf())
    {
      transformedRay = transform(topNode, ray);
      subNode = topNode.subBvh;
    } else {
      topLvlStack.push(closestChild);
      topLvlStack.push(furthestChild);
    }
  }

  if (subNode is not initialized)
    break;

  while (true)
  {
    if (node.isLeaf())
    {
      intersectTriangles();

      if  (!subLvlStack.empty())
        subNode = subLvlStack.pop();
      else
        break;
    } else {
      intersectChildNodes();
      if (both childs visible)
      {
        subLvlStack.push(furthestChild);
        subNode = closestChild;
      } else if (any child visible)
      {
        subNode = visibleChild;
      } else {
        if  (!subLvlStack.empty())
          subNode = subLvlStack.pop();
        else
          break;
      }
    }
  }
}
```

Because GPUs dont like large branches, the trace function only supports rays. To get the desired functionality, it instead supports an extra parameter that indicates the maximum distance that should be considered. Internally, this is implemented as setting the closest intersected triangle distance to that value. At the end of the traversal we check whether this value has changed (so whether we hit a triangle or not). This work around makes use of the code that we were already using and is thus zero overhead. It also cleans up the code a lot because we do not need separate intersection functions anymore.

The NVIDIA paper also suggest that using persistent threads can provide a significant speed up. The authors claim that the thread creation cost is relatively high and contributes significantly to the performance. In their implementation, they show a performance increase of up to 60% by moving to persistent threads. We have tried to implement persistent threads in OpenCL but they are not a good fit.

The authors of the paper assume that a GPU executes 32 work items in parallel. This assumption may be correct for NVIDIA hardware, but AMD uses wavefronts of 64 threads. Furthermore, they make the assumption that in a 2D workgroup of (32, y), a warp will always execute work items with the same y coordinate. On our testing hardware (AMD RX480) this does not seem to work and we had to add memory barriers to fix it. With a lot of manual tweaking of the input parameters, we ended up with a 2% performance improvement.

Because we want our code to be portable, we have decided not to pursue this path any further. When we run an empty kernel, we get about 1200fps at 720p. This includes the OpenCL/OpenGL interop and the therefor required synchronization. It seems like the kernel overhead of OpenCL on AMD is much lower than NVIDIA's Cuda, and does not require any further optimization.

5 Parallel

With this assignment we have mainly focused on supporting a large range of animations and writing good and modern C++ code. This does not mean that our application is totally unoptimized. Where possibly, std move is used and data is pre-calculated.

Instead of spending our time on CPU code optimization, we have tried to write an efficient GPU traversal. One of our goals was to support animations with as little performance overhead. We try to accomplish this by letting the CPU work in parallel with the GPU. While the GPU is rendering the current frame, the CPU is already working on the next frame. To make this work, we use double buffering on all the dynamic GPU buffers. Furthermore, all modern graphics card support Direct Memory Access so that the GPU can work while data is being copied. To make use of this hardware, we create a second OpenCL work queue that is used for copying the dynamic data. The two queues are synchronized using OpenCL events, so that no CPU intervention is required.

6 Performance/Results

Static BVH building can be quite expensive and may take minutes to complete for large and/or complicated models. Preferably this would happen only once is a preprocessing step. For this reason we have not focused on the performance of the BVH builders in terms of construction time. In this section we will only test the average traversal time of some complex and large models. For all performance tests we used OpenCLs profiling commands to measure the execution time of the ray tracing kernel. The results shown are the running average of 200 frames.

We wanted the models that we tested to be widely available. We therefor chose to use two models from The Stanford 3D Scanning Repository[4]. The third model is a reworked version of the Crytek Sponza scene by Frank Meinl[2]. We initialize experimented with the original model but it had lacking textures so we use the updated one. For the deformation test, we had problems finding animated models that are animated as a sequence of meshes. We ended up with a 3D model of an UH-60 Black Hawk with animated rotors, which was modeled by A. Shamloo[3].

6.1 Static meshes

To compare the performance of the static builders we have used SBVH and the binned builder which always splits along the longest axis. Even though the SBVH has way longer construction times, it does not always deliver better results. The SBVH algorithm has multiple configuration parameters for which the optimal value may be different for every model. For our experiments we have used 8 bins, alpha=0.01, intersection cost=1.0 and traversal cost=1.5. We wanted to use a higher bin count for SBVH but this was not possible due to the extended construction times. For the binned builder we used 16 bins as suggested by the authors[5].

The most remarkable result was how the basic binning performed over 1000 times slower than the alternatives in the Sponza scene. The biggest problem in this scene is that it requires evaluation of all 3 axis, not just the longest one. To prove this we tested the scene with an extended version of the binned BVH builder that considers all 3 axis. This method got us within a couple percent of SBVH. We had high hopes for the SBVH builder considering it is an extension to the binned builder. Unfortunately it performed worse for the Stanford bunny and only slightly better for the Stanford dragon. We have experimented with different configuration values but it seems like splitting just does not help performance in both cases. One reason could be that the scanned Stanford models are made of very small, regular triangles that do not benefit that much from splitting.

Table 1: Kernel execution time static builders (1280x720)

SBVH	Binned	3 Axis Binned	SBVH
Stanford Bunny	4.31ms	4.17ms	4.07ms
Reworked Crytek Sponza	12.01ms	14543.66ms	12.53ms
Stanford Dragon	7.30ms	7.67ms	7.27ms

6.2 Deformable meshes

As stated in the second paragraph of this section, we had problems finding suitable 3D models to test the animation on. We ended up with an animated helicopter model of which the rotors rotate. We have measured the ray tracer execution time of 5 frames of animation with both rebuilding and refitting. The rebuilding uses the binned builder that splits along the longest axis only, as described in “On fast construction of SAH-based bounding volume hierarchies”[5]. Refitting requires the triangle count to stay the same. The SBVH builder may add new triangles however, resulting in the triangle count of the reference frame changing. If we can make the assumption that the triangle list does not change for any frame of animation, we can reuse the triangle array of the reference frame. Although this holds true for most models, 3 axis binned BVH building can also be used in both cases.

This is another model where the longest axis splitting does not seem to work out very well. Although it is fast enough to rebuild in real time, the created BVH is of very low quality. Like with the Sponza scene, splitting along all 3 axis fixes the issue completely. Considering how bad basic binning performs in this scene, refitting seems to be the better option. With refitting, the first frame of animation is the unaltered high quality BVH. Any following frame is refitted based on the BVH of the first frame. The results show that refitting can create an acceptable BVH at a very low cost. Rebuilding

Table 2: Kernel execution time deformable animation

Model (Frame)	Binned	Refitting(3 Axis Binned)	Refitting(SBVH)
heli (0)	283.5ms	4.12ms	4.71ms
heli (1)	26.7ms	5.06ms	5.84ms
heli (2)	329.1ms	4.25ms	4.95ms
heli (3)	48.1ms	5.30ms	6.25ms
heli (4)	381.6ms	4.44ms	5.26ms

the BVH may be a better solution for models that deform more from there original state. For those cases a more optimized implementation of the 3 axis binning could be considered.

7 Conclusion

Building a BVH generally increases the performance of a scene, as expected. Dividing the scene between a top BVH for scene nodes and sub-BVHs for single meshes allows us to have rigid-body animation of static meshes without having to recalculate their sub-BVHs.

The SBVH approach greatly improves the performance of static, architectural scenes such as Sponza, that have very elongated triangles and barely run under the standard binned approach. It still provides a very small gain on meshes made of very small and similar triangles such as the Stanford dragon. The Stanford bunny, instead, performs slightly but not significantly worse. SBVH takes quite a long time to build, so it is only suitable for static scenes.

The fast BVH approach is exceptional for non-static animated objects. It provides the ability to rebuild BVHs in a very small amount of time, being able to keep up with the frame-by-frame rendering of the object. It is not as optimal as the SBVH for traversal speed, but makes up for it in execution speed.

We have also implemented refitting for models that don't change very much. This only updates the node boundaries, without rebuilding the hierarchy.

In addition, building the BVH hierarchy forced to make much needed change to our framework architecture, which is now better organised.

In general we can conclude that the usage of BVHs in raytracers is well-justified, and we managed to have a pretty good implementation including traversal code in OpenCL.

References

[1] Timo Aila and Samuli Laine. “Understanding the Efficiency of Ray Traversal on GPUs”. In: *Proc. High-Performance Graphics 2009*. 2009.

[2] Morgan McGuire. *Computer Graphics Archive*. Aug. 2011. URL: `http://graphics.cs.williams.edu/data`.

[3] Abdy Shamloo. *asART*. URL: `http://www.as-art.ch/downloads/helicopter-uh60-obj-sequence/`.

[4] *The Stanford 3D Scanning Repository*. Stanford University. URL: `https://graphics.stanford.edu/data/3Dscanrep/`.

[5] Ingo Wald. “On fast construction of SAH-based bounding volume hierarchies”. In: *2007 IEEE Symposium on Interactive Ray Tracing*. IEEE. 2007, pp. 33–40.

[6] Bruce Walter et al. “Fast agglomerative clustering for rendering”. In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE. 2008, pp. 81–86.