Image Synthesis Project
Paper: High Resolution Sparse Voxel DAGs [1]

Alexandre Pérot

I. Introduction

Ray tracing is shown a growing interest for real time rendering, as a way to truthfully simulate secondary illumination. However, for meshes with high level of detail (and augmented with displacement maps), the high triangle-count slows down the execution since it affects for example both the primary ray and secondary ray computations. Therefore, Sparse Voxel Octrees (SVO) are a promising field of research as a secondary scene representation. Indeed, it can provide a representation of the scene with varying level of details (lowering the level of detail when evaluating secondary ray does not affect the result much) which can be passed through with a time complexity that only depends on the level of detail chosen. However, high resolution SVO are heavy objects, hence the paper [1] presents a strategy to compress SVO without hindering the query time complexity.

II. Summary of the paper

A. Data format

First of all, the SVO is stored in the following format. Each node is composed of a *childmask* (an 8-bit bitmask such that the bit i represents whether or not the child voxel of index i is empty or not) followed by 32-bits pointers to the non-empty children (in order). Leaves of the SVO only contain a *childmask*. Though the childmask is only 8-bit, it is stored in a 32-bits unit next to the following pointers. In addition to that, they chose to represent the 2 lower levels (4^3 resolution voxels) as `64-bits integers`. In summary, the SVO is stored as a vector of 32-bits values (the nodes) and a vector of 64-bits values (the leaves).

B. Computation of the SVO

In the paper, the SVO is computed using triangle-cube intersection for the first levels, and then using depth peeling (rasterization based process, which becomes faster than triangle-cube intersection when the number of triangles becomes small compared to the number of cubes). The authors point out that this is not the most efficient SVO construction algorithm.

C. Sparse Voxel Directed Acyclic Graph

The authors of the paper identified that the SVO contained many identical subtrees. Therefore, their contribution is to propose an algorithm to remove these repetitions from the data structure, by transforming the SVO into a Sparse Voxel Directed Acyclic Graph (SVDAG). To do so, we identify identical subtrees in the SVO with a bottom-up approach and merge them. Concretely, they sort the nodes of a given level using lexicographic order which then allows them to quickly identify the identical nodes, merge them and update the pointers of the higher level. If the SVO is too big to be stored before being transformed into a SVDAG, one can start transforming a partial SVO into a SVDAG (for example the eight subtrees starting at depth 1 can be computed and transformed one-by-one).

D. Rendering

The paper does not propose any novel method to ray trace using the SVDAG. SVDAG can only be used for shadows and Ambient Occlusion (AO) because it does not store shading parameters. Since the goal of the paper is to evaluate the SVDAG, the authors still chose to compute the hit point of the primary and secondary rays with the SVDAG and then use a parallel SVO containing shading parameters to compute the color response.

To compute shadows and AO, since the SVDAG is queried exactly like a SVO, they refered to [2] for the computation of shadow rays (i.e function that output whether or not the ray was blocked before a maximal distance) and [3] for an algorithm to evaluate AO using only shadow rays.

III. Implementation

A. Construction of the data structure

1) Data format: Because C++ pointers are not necessarily 32-bits and to ensure the data is stored contiguously, I used 32-bits unsigned int as pointers that store the index of the child node in its container (a vector). I used one vector of `32-bits unsigned integer` per level. Therefore, *childmasks* are represented as `32-bits unsigned integer` too but only the last 8-bits are significant and the rest is always null.

2) SVO: I only used triangle-AABB intersection to construct the SVO, with the formula presented in [4] which uses a trick (translating everything on the refential centered on the AABB) to prevent redundant computations (only one vertice of the AABB instead of 8 on the candidate separating axes).

Because a node needs to store a pointer to the position of its children when created, and because the size of a node is not constant (it can range between 4 and 36 bytes), we can not create a node before all its children are created. Therefore, I used a recursive function to construct the SVO bottom-up. When this function is called on a voxel, with a list of triangles to test, it does the following:

- create a list of the triangles that intersect the voxel,

- if it is not empty, call the function on each of the 8 children voxels by giving them the list of triangles as argument and create a node with the results of the recursive call (the position of the children's nodes, if not empty)
- returns whether or not the voxel is empty, and the position of the the node if not empty.

During the construction of a SVO of depth L, I kept for the $L-2$ first levels a temporary vector of pointers to the position (of the *childmask*) of each node, which will be used for the reduction into a DAG.

3) Compression of the lower levels: By using the temporary vector pointing to the nodes in the level $L-2$, I transform each of these nodes into a `64-bits unsigned integer` leaf representing the subtree of depth 2 starting at this node. This is done simply by concatenating the 8 *childmasks* of the level $L-1$ children nodes (and adding 8 null bits when the node is empty).

4) DAG: To reduce the SVO into a SVDAG, first identical leaves are found and merged (using a `set` data structure), and the nodes in the last level are updated accordingly.

Then a similar process is done bottom-up. However, since the nodes are not of fixed size, at each level, I kept (in addition to the already computed vector of position of the beginning of each nodes $node_{begin}$) a vector containing the position of the end of each node $node_{end}$. I can then define comparison operation for nodes' indexes and use them to sort the indexes of the nodes. I had to sort a separate vector containing the indexes of the nodes instead of sorting directly $node_{begin}$ because $node_{end}$ needed to be sorted simultaneously in the exact same way. Once the sorting is done, I can remove identical nodes. Like the authors of [1], I keep a map from the old node position to the new index of the node and use it to update the pointers of the higher level.

B. Ray tracing using the data structure

I decided to use the SVDAG for shadow and AO computation. Therefore, I only needed a function to test shadow rays (i.e whether or not a ray is blocked before a certain maximal distance).

1) Shadow ray traversal: The algorithm used for the shadow ray is heavily inspired by [2]. The first step is to locate the ray.

- if the ray is outside the biggest voxel (which should not happen for shadow rays in theory but may occur due to the offset on the ray origin used to avoid self shadowing), the ray either don't intersect the voxel (then return false), or we can define an $entry\_point \in \mathbb{R}^3$ and an $entry\_direction \in 0, 1, 2$ which is the axis from which the ray entered the voxel.
- if the ray is inside, we define $entry\_direction$ and $entry\_point$ by seeing where the ray exits the voxel it is currently in (the ray is not blocked by the starting voxel to avoid self shadowing).

The current voxel is represented by a stack corresponding to the path in the SVDAG to the voxel and we move along the ray until it is blocked, has reached the maximal distance or exited the SVDAG by using the following operations:

- PUSH: If the voxel is not empty but not at maximal depth, we find the child voxel that contains the $entry\_point$.
- ADVANCE and POP: If the voxel is empty, we compute the axis on which we need to shift to advance (the direction of the shift depending of the sign of the ray's direction along the shift axis) and pop nodes from the stack until we can advance on the shift axis.
- BLOCKED: if the voxel is at maximal depth and full, the ray is blocked, we return whether or not the distance condition is respected.

In my implementation, because the lower levels are compressed into full Voxel Octrees, I implemented separate functions to manage the traversal of these lower levels which do not require a stack and that are called when ADVANCE is used on a node of the lowest level.

This function can then be used as is to replace the function (based on a BVH) used in the ray tracing software application developed during the practical sessions.

2) Ambient occlusion: Ambient occlusion is often computed using the following formula:

$$AO(\mathbf{p}, \mathbf{n}) = \frac{1}{\pi} \int_{\omega \in \Omega} \rho(D(\mathbf{p}, \mathbf{w}))\langle \mathbf{w}, \mathbf{n} \rangle dw$$

where ρ is a falloff function (a decreasing function), Ω the hemisphere around the normal and D the distance of the first hit. To approximate this integral, we use a Monte-Carlo scheme by sampling rays on the hemisphere with a cosine function. In practice, according to inverse transform sampling, I simply had to sample the sine of theta (spherical coordinates) uniformly in $[0, 1]$.

However, as is, this method requires knowing at which distance the ray was blocked. The paper [3] proposes a sampling method to allow the use of simple shadow rays that do not output D the distance of the first hit. To do so, we use the falloff function to sample a maximum distance l for the shadow ray. More precisely, quick computations show that if the falloff function is strictly decreasing and $\rho(0) = 1$, we can sample l with the probability distribution $p(l) = -\rho'(l)$, since:

$$\mathbb{E}(O(\mathbf{p}, \mathbf{w}, \mathbf{l})) = \mathbb{E}(\mathbb{E}(\mathbb{1}_{l \geq D} | D)) = \mathbb{E}(p(l \geq D)).$$

In my implementation, I used an exponential falloff function $\rho(d) = e^{-\frac{d}{\lambda}}$, where λ needs to be of a magnitude similar to the size of the objects of the scene. Indeed, this formula gives an exponential distribution of parameter $\frac{1}{\lambda}$ for the maximum distance l (which means that it will be λ in average).

IV. Results

To check the sanity of the SVDAG structure, we can use it to render with a parallel projection along one of the main axes. The result of such rendering with *example_highres* can be seen on Figure 1.

Fig. 1. Projection of the computed SVDAG along the z-axis, using grey-scale to represent the maximum coordinate along the z-axis

A. Size

TABLE I
SIZE OF THE DATA STRUCTURE

Mesh	Resolution	Structure	Size	Nodes[1]
example_highres	1024^3	SVO	11Mo	2.8M
		SVOc[2]	3.8Mo	87K +260K
		SVDAG	1.9Mo	73K + 37K
	2048^3	SVO	44Mo	11M
		SVOc	15Mo	350K + 1M
		SVDAG	6.4Mo	270K + 48K
	4096^3	SVO	180Mo	44M
		SVOc	61Mo	1.4M + 4.1M
		SVDAG	23Mo	970K + 55K
	8192^3	SVO	710Mo	180M
		SVOc	240Mo	5.5M + 17M
		SVDAG	77Mo	3.3M + 65K
example_lowres	2048^3	SVO	44Mo	11M
		SVOc	15Mo	350K + 1M
		SVDAG	5.7Mo	240K + 44K
raptor	2048^3	SVO	36Mo	9M
		SVOc	12Mo	270K + 840K
		SVDAG	6.7Mo	240K + 190K

As we can see in Table I, the SVDAG is always more memory efficient than the SVO. This result was expected, because I did not use memory efficient ways of storing the SVO and therefore, since the SVDAG and the SVOc (with compressed lower levels) are stored the exact same way, the SVDAG is necessarily smaller in terms of size.

Also, as expected, Table II shows that the lower levels of the tree are were most of the size reduction is found, since this is were most nodes are while the size of the space of different possible nodes is smaller. It is interesting to note that the number of different nodes kept after reduction in the SVDAG at level 9 (for a depth of 11, and a reduced depth of 9) is much smaller that the size of the space of different possible nodes 2^{64}. It is because most of these subtrees are the result of a plane-voxel intersection (since the resolution is quite high), and therefore, many configuration do not happen.

Another interesting point highlighted by the comparison of the *example_highres* and the *example_lowres* meshes is that the more detailed the mesh is, the less compression there is by using SVDAG. Indeed, at the same resolution, the lowres mesh will provide simpler shapes (mainly planes with varying orientations) which have a higher chance of appearing many times at different places. Therefore, a high polygon count has only an effect on the compression into a SVDAG whereas the SVO keep similar sizes. The mesh *raptor*, which has the highest polygon-count, illustrates well this concept: the SVO is smaller than the one of the *example_lowres* and *example_highres* (because there are more blank spaces due to the shape of the mesh), however, the SVDAG is bigger because of the high level of detail that reduced the amount of similar patterns.

TABLE II
NUMBER OF STORED NODES PER LEVEL

Level	SVO	SVOc	SVDAG
0	1	1	1
1	8	8	8
2	54	54	54
3	238	238	238
4	989	989	989
5	4K	4K	4K
6	16K	16K	16K
7	65K	65K	60K
8	260K	260K	190K
9	1M	1M(64bits)	48K(64bits)
10	4.2M	-	-
11 (implicit)	-	-	-

B. Construction speed

Table III displays the construction time of the SVDAG structure for different resolutions for a given mesh. These speeds are not optimal: for example, part of the process could be parallelized.

[1]When the lower levels are compressed, the corresponding nodes are not counted and the number of such subtrees is given next to the number of nodes
[2]SVO data structure with the lower levels compressed in 64bits

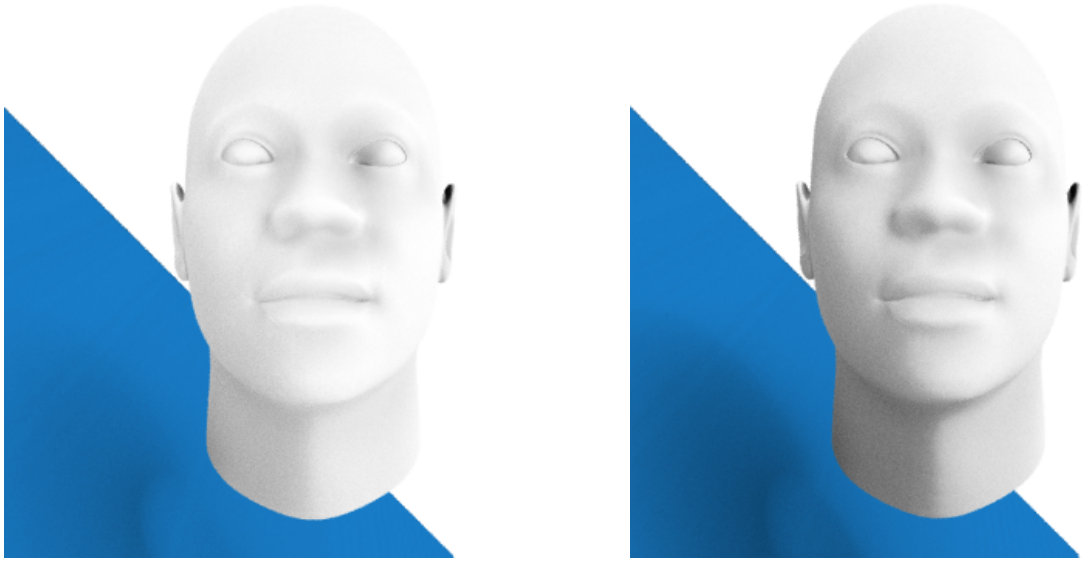

Fig. 2. Rendering with only AO (16 samples per pixel, 64 AO samples per ray, no light source, no secondary rays), with different falloff characteristic distances (*left*: 0.4, *right*: 1).

TABLE III
CONSTRUCTION TIME FOR *example_highres*

Depth	Resolution	Construction time
10	1024^3	16s
11	2048^3	37s
12	4096^3	92s
13	8192^3	290s

C. Shadow rays

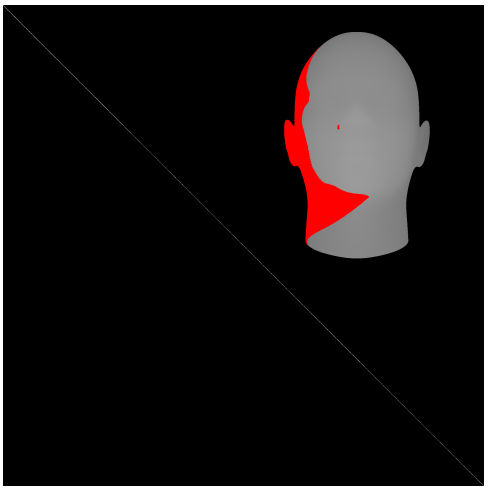

Fig. 3. Parallel rendering (using the SVDAG) of the scene used for rendering. The diagonal that can be seen is the plane placed below the mesh. In red are voxels that are blocked from the light source by another voxel (computed using the shadow rays of the SVDAG).

Figure 3 illustrates the proper behavior of the implemented shadow ray routine of the SVDAG.

D. Ambiant occlusion

Figure 2 illustrate ambient occlusion with different falloff characteristic distances (used in the exponential falloff function). We can see in the shadow how a high distance creates higher scale phenomenon (such as the mutual shadowing of the face and the plane), without impacting smaller scale effects. Figure 4 shows how adding ambiant occlusion impacted the final result. The eyes and the ears are where the difference is most noticeable.

E. Ray tracing speed

TABLE IV
EXECUTION SPEED OF THE RAYTRACING

Scene	Method	Execution speed (+ SVDAG construction)
example_highres	BVH	74s
	SVDAG	46s (+16s)
example_lowres	BVH	41s
	SVDAG	38s (+6s)
raptor	BVH	59s
	SVDAG	46s (+32s)

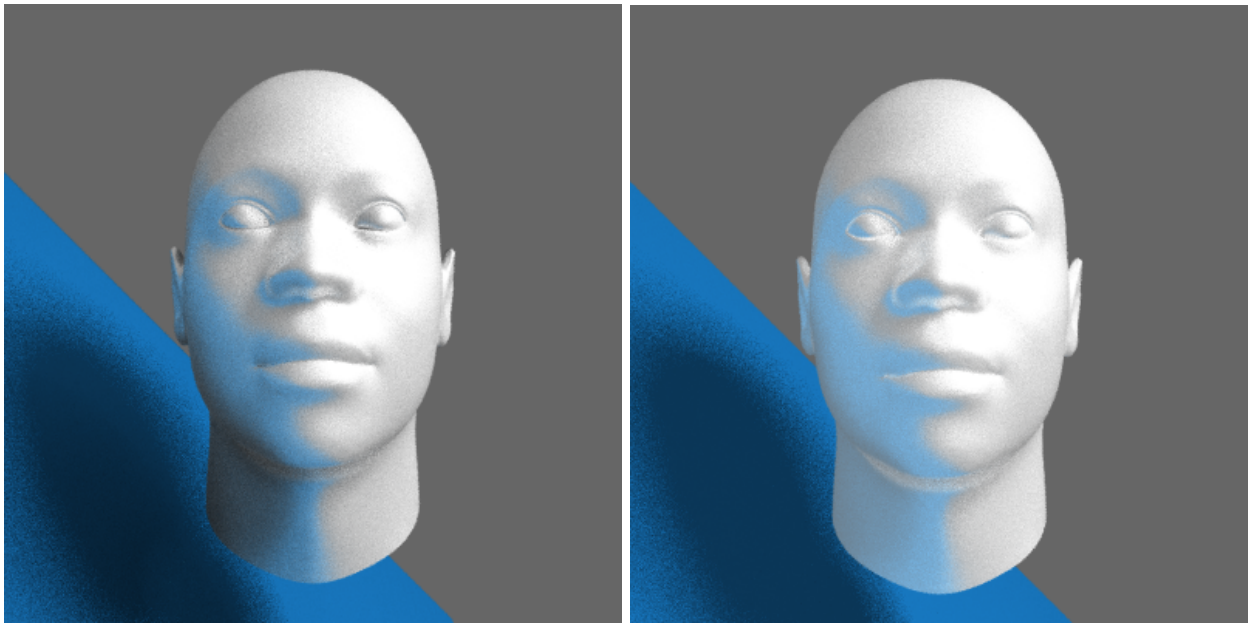

Fig. 4. Rendering using the SVDAG for shadow rays, using a square light source, 16 ray per pixel, 3 bounces and 16 secondary rays (importance sampling). For secondary rays, only one sample with the direction with maximum probability (reflection) is used for subsequent bounces. *left*: with AO (1.0m falloff distance, 16 rays for AO estimation on the primary ray, 1 ray for secondary rays), *right*: without AO

Table IV displays the execution speed of the ray tracing application when using BVH (as in the practical sessions) or the SVDAG to compute the shadow rays for both shadows and AO, on different scenes. All DAGs have a depth of 11 which was mandatory to obtain proper AO on the *example_highres* mesh (eyelid/eyeball contact for example). The scene consists of the mesh and a plane (see Figure 3 for a zoomed out representation of the global scene). The construction times are higher in Table III than in Table IV because the addition of the plane in the scene increased the size of the root voxel and created lots of empty space (see Figure 3).

V. Conclusion

In conclusion, the SVDAG structure effectively reduces the size of a SVO structure (even though we did not use the most size efficient SVO data structure, the paper shows that SVDAG remains often better, without further optimization of the format). Moreover, I could show that using a secondary structure to represent a scene improved the rendering speed (if we do not count the construction time). However, considering the time spent constructing the structure could be optimized (parallelisation) and a structure can be reused as long as the meshes do not move in the global frame, the results are still pertinent. Finally, it was an occasion to implement ambiant occlusion in the rendering software of the practical sessions and study its impact on the realism of the rendered image.

The main improvement that could be made to the software and that was not implemented in my work is the use of different levels of detail in the SVDAG for secondary rays to further speed up rendering without changing the result image much.

References

[1] V Kämpe and al. High Resolution Sparse Voxel DAGs http://www.cse.chalmers.se/~uffe/HighResolutionSparseVoxelDAGs.pdf, 2013.

[2] S. Laine and al. Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation https://research.nvidia.com/sites/default/files/pubs/2010-02_Efficient-Sparse-Voxel/laine2010tr1_paper.pdf, 2010.

[3] S. Laine and al. Two Methods for Fast Ray-Cast Ambient Occlusion https://users.aalto.fi/~laines9/publications/laine2010egsr_paper.pdf, 2010.

[4] AABB-Triangle Intersection https://gdbooks.gitbooks.io/3dcollisions/content/Chapter4/aabb-triangle.html

VI. Appendix

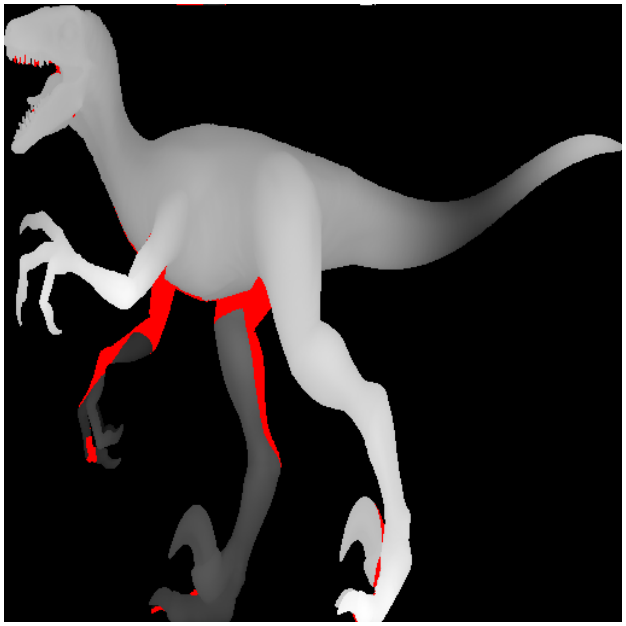

Fig. 5. Parallel projection of the SVDAG along the z-axis. In red are the voxels that are blocked from a given point light source by another voxel. The light is in (2,2,4) while the root voxel has a size of (1.6, 0.8, 0.3) approximately centered on the origin and the viewer's plane (i.e white pixels) is z=0.15. The mesh is deformed because voxels don't have to be isotropic, therefore we chose the root voxel as the bounding box of the mesh.

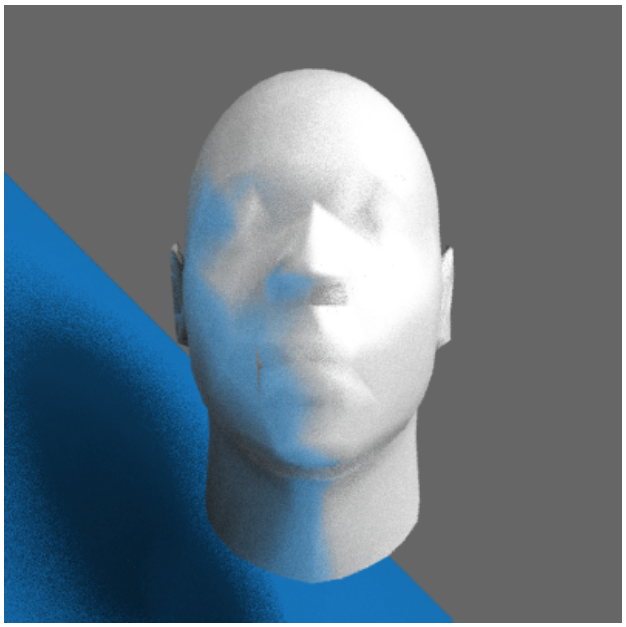

Fig. 6. Rendering of the *example_lowres* mesh using the same parameters as in Figure 4

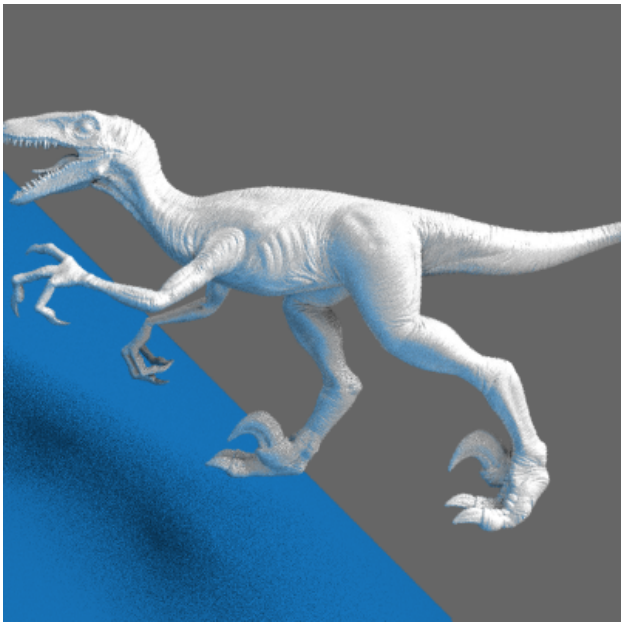

Fig. 7. Rendering of the *raptor* mesh using the same parameters as in Figure 4

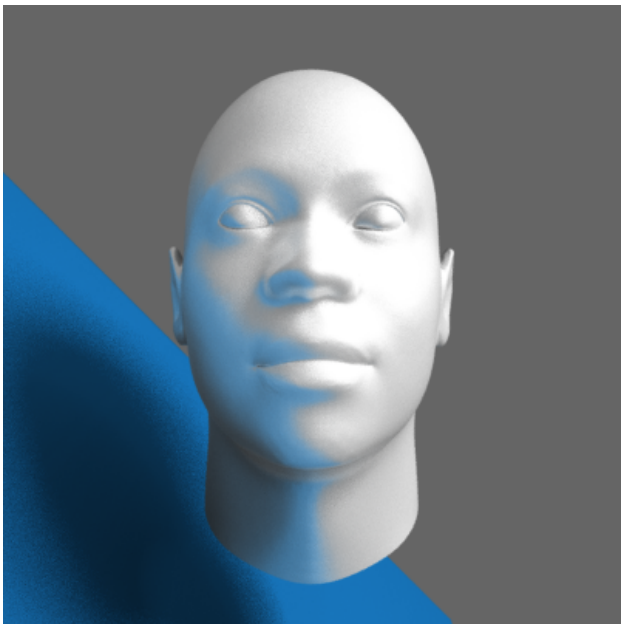

Fig. 8. Rendering of the *example_highres* mesh using a high amout of sample rays (64 samples per pixel, secondary rays and rays for ambient occlusion)