High-Performance Computing

Final Project

High Performance Ray Tracing

Keqing Chen	Yihan Sun	Xinran Xu
CS&T05	CS&T05	CS&T05
2010011347	2010011356	2010011358

December 28, 2012

1 Introduction

Ray tracing[1] is a popular rendering method nowadays. Instead of regular way of thinking, its main idea is to generate an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. We'd like to implement a real-time ray tracer to handle deformable objects using parallel techniques as the final project. Following I will introduce some simple concept of this domain.

1.1 Polygon Mesh

A polygon mesh or unstructured grid is a collection of vertices, edges and faces that defines the shape of a polyhedral object in 3D computer graphics and solid modeling. The faces usually consist of triangles, quadrilaterals or other simple convex polygons, since this simplifies rendering, but may also be composed of more general concave polygons, or polygons with holes. See a example as *Figure 1*.

In this project, we use polygon meshes only consisting of triangles.

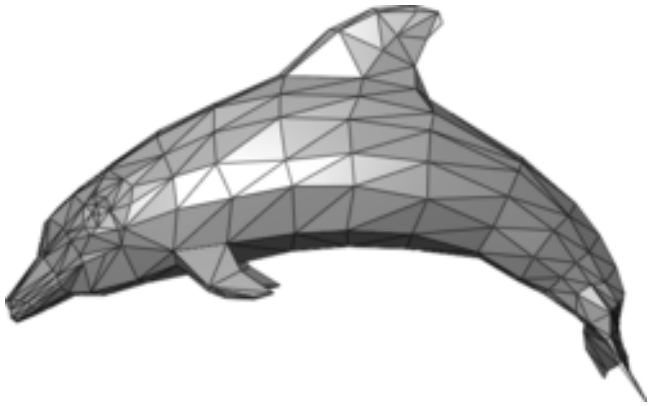

Figure 1: Example of a triangle mesh representing a dolphin

1.2 Ray Tracing

In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television visual effects, and more poorly suited for real-time applications like video games where speed is critical. Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and dispersion phenomena (such as chromatic aberration). *Figure 2* is an example of recursive ray tracing of a sphere illustrating the effects of shallow depth of field, area light sources and diffuse interreflection.

1.3 Bounding Volume Hierarchy (BVH)

A bounding volume hierarchy (BVH) is a tree structure on a set of geometric objects. All geometric objects are wrapped in bounding volumes that form the leaf nodes of the tree. These nodes are then grouped as small sets and enclosed within larger bounding volumes. These, in turn, are also grouped and enclosed within other larger bounding volumes in a recursive fashion, eventually resulting in a tree structure with a single bounding volume at the top of the tree.

It is not difficult to build a BVH. In fact, we shall just build it recursively and update the bound of each node. At the very beginning, all the triangles in our scene are in the same node, which is the root of the tree. Then we should split it into two parts and create a node to hold them. For every part, as long as there are more than one triangle in it, we will split it. Finally, each triangle in the scene

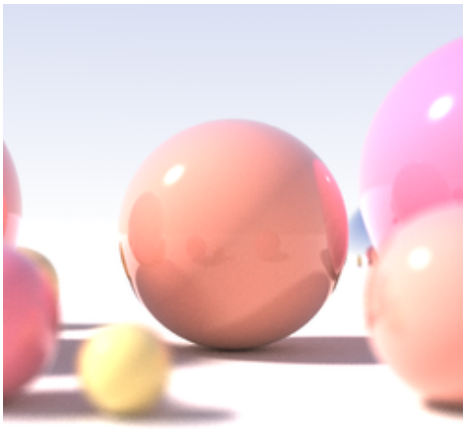

Figure 2: This recursive ray tracing of a sphere demonstrates the effects of shallow depth of field, area light sources and diffuse interreflection.

will identifies a single leaf node. Each node is the minimum or smallest bounding or enclosing box of the triangles in it. Imagine that there is a box or cuboid which includes all the triangles in the node, and the coordinate value of the box should be the bound. Obviously, we can calculate the bound of each node by combining its two sub-nodes (just find the minimum and maximum value of coordinates in each dimension). This can be easily implemented recursively.

1.4 Overview

As the name of ray tracing shows, the two main processes of ray tracing are: 1. ray-mesh intersection checking; 2. estimating the color and shadow effect of each vertex (or points) on the triangular mesh.

For estimating the color and shadow effect of each vertex (or points) on the triangular mesh, we directly used the reflect ray.

Ray-mesh intersection checking is the main function call in the framework. It is used for checking which points on the triangular mesh are represented for all the pixels on the screen, as well as checking the reflection and refraction result on these points affected by the lighting environment. Some data structures(BVH, KD-Tree, etc.) are specially designed for ray tracing to accelerate the simulating process. Traditional ray tracing algorithms can only handle static scenes, but in recent years, some techniques have been presented to handle deformable scenes[5][6][7]. Using parallel methods can greatly improve its performance. We would like to study on the data structure and implement a good ray tracing engine which is suitable for deformable scenes using parallel methods.

2 Algorithms

The framework of our algorithm is to build a BVH with an input model and then do the rendering. If the scene is deformable, we will alternative execute building a BVH and rendering. If the scene is static, we just re-rendering.

3 BVH Construction Details

In this section we will introduce the implementation details of our acceleration data structure – Bounding Volume Hierarchy.

3.1 SAH-Based BVH

As mentioned above, it is not difficult to build a simple BVH. The only problem is to balance the tree in order to improve its efficiency. As we can see, if we split the nodes into parts with almost the same size, this data structure will have the best performance. In other words, we would like the *volume* of the node to be as small as possible. We can define the volume as a heuristic function. Figure 3 show how different BVH works. A simple idea proposed in [8] is to define a heuristic function called SAH(Surface Area Heuristic). It define the cost of dividing a node V into two parts L and R like the following equation.

$$Cost(V \to \{L, R\}) = K_T + K_I \left(\frac{SA(V_L)}{SA(V)} N_L + \frac{SA(V_R)}{SA(V)} N_R \right) \tag{1}$$

Where $SA(V)$ is the surface area of V, and K_T and K_I are some implementation-specific constants for the estimated cost of a traversal step and a triangle intersection, respectively. $N(V)$ represents the total triangle number is the node V.

For K_T and K_I are constants, and we shall only compare different cost between different methods, we simplify the constant K_T to be 0, and K_I to be 1. Also, for the same node V, $SA(V)$ is a constant. As a result, we can simplify the equation like follows.

$$H(V) = SA(V_L) \times N_L + SA(V_R) \times N_R \tag{2}$$

A simple idea is to sort center of gravity of each triangle in terms of the three dimensions. Then try to split the sequence from each gap of the nodes. Record and compare the heuristic function of each scheme and choose the one with the smallest *volume*.

The algorithm is first proposed in [4]. A simple version of pseudo code to build a BVH is like follows.

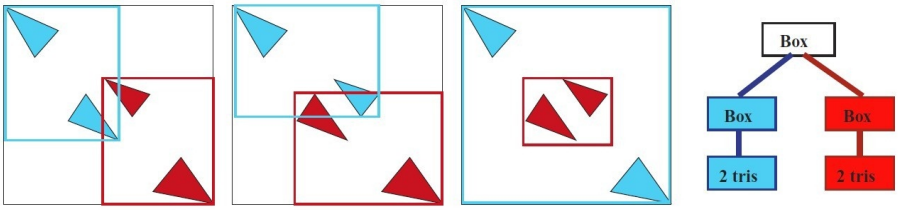

Figure 3: A example of building a BVH

Algorithm 1: BuildBVH

void buildBVH($curNode, setTri$)
// $curNode$: current Node
//$setTri$: the set of all triangles in this set, represented by barycentric center
set $N \leftarrow |setTri|$
if (N is small enough):
    Set N triangles in setTri into a leaf
    Compute the BoundingBox of curNode
    return;
end if
$bestCut \leftarrow \infty$
$orderX \leftarrow Sort(setTri.x)$
Update $bestCut$ by all possible cut in $orderX$
$orderY \leftarrow Sort(setTri.y)$
Update $bestCut$ by all possible cut in $orderY$
$orderZ \leftarrow Sort(setTri.z)$
Update $bestCut$ by all possible cut in $orderZ$
Split $\{setTri\}$ into $\{leftTri\}$ and $\{rightTri\}$
buildBVH($leftNode, leftTri$)
buildBVH($rightNode, rightTri$)

3.2 Straight-forward BVH construction

For there are three dimensions, we should do the same thing as mentioned above(sort, enumerate and choose). The time complexity of sorting is $O(n \log n)$ and the number of triangles are always 10^8 or more. This is very expensive even though we can parallel it to some extent. In deformable scene, we should give the result as soon as the new scene is constructed and make the delay as short as possible. As a result, we want to save the time of sorting and utilize the node order which is generated when building the node of the current node's parent.

We can just mark each node whether it will be allocated to the left node or the right node(e.g., 0 for left node, 1 for right node), then the order will not be disturbed. As a result, we do not need to sort all the triangles each time when we are building a node. A version of pseudo code is like follows.

Algorithm 2: advanced BuildBVH

void buildBVH($curNode, setTri$)
// $curNode$: current Node
//$setTri$: the set of all triangles in this set, represented by barycentric center
set $N \leftarrow |setTri|$
if (N is small enough):
    Set N triangles in setTri into a leaf
    Compute the BoundingBox of curNode
    return
end if
$bestCut \leftarrow \infty$
Update $bestCut$ by all possible cut in $orderX$
Update $bestCut$ by all possible cut in $orderY$
Update $bestCut$ by all possible cut in $orderZ$
Split $\{setTri\}$ into $\{leftTri\}$ and $\{rightTri\}$ in order
buildBVH($leftNode, leftTri$)
buildBVH($rightNode, rightTri$)

3.3 Bin-tech BVH Construction

Now we make an investigation of the previous algorithm in 3.1, most calculations appear in numerating the cut plains and making partition in all three dimensions. These operations are redundant. What's more, these operation are hard to parallel in the top levels of the tree. To improve parallelism, we implement the algorithm in [3]. The basic idea of this algorithm is to cut the triangles into 32 groups evenly according to their coordinate value. Then we only compare the effect of cutting them at the gaps of the groups. [3] proved that the tree quality will not decrease obviously, yet the number of operations will have a significant decrease and the parallelism will be improved greatly.

The pseudo code of the approximation algorithm presented above is as follows.

Algorithm 3: Bin-tech BVH Construction

void buildBVH($curNode, setTri$)
// $curNode$: current Node;
//$setTri$: the set of all triangles in this set, represented by barycentric center
set $N \leftarrow |setTri|$
if (N is small enough):
    Set N triangles in setTri into a leaf
    Compute the BoundingBox of curNode
    return
end if
$bestCut \leftarrow \infty$
Cut the $setTri$ into 32 groups evenly according to the x-axis value
Update $bestCut$ by all possible cut in the 32 groups
Cut the $setTri$ into 32 groups evenly according to the y-axis value
Update $bestCut$ by all possible cut in the 32 groups
Cut the $setTri$ into 32 groups evenly according to the z-axis value
Update $bestCut$ by all possible cut in the 32 groups
Split $\{setTri\}$ into $\{leftTri\}$ and $\{rightTri\}$
buildBVH($leftNode, leftTri$)
buildBVH($rightNode, rightTri$)

3.4 Parallel the Process of Building a BVH

Two main aspects can be considered to making the parallelism. If we carefully analyze the pseudo code in **Algorithm 3**, we can find that this is a divide-and-conquer technique based algorithm, while in each divide process, the cut checking and partitioning the $setTri$ in linear time. We can process it in the following to sections.

3.4.1 Parallel invoke

For the two subtree building, we can get two different threads to do it in parallel. In particular, we use *Cilk++* to implement it, which the code is:

```
cilk_spawn
buildBVH(leftNode, leftTri)
buildBVH(rightNode, rightTri)
cilk_sync;
```

However, we do not want to this parallel invoke to be proceeded too many times. The reason is that we only have several cores (2-8) on a commonly used CPU, and it costs considerable to allocating space and parameter transferring. Hence, we can only to this for the top levels, and as long as we got enough threads, we change it into sequential version.

3.4.2 Parallel bucket sort

For partitioning and scanning in **Algorithm 3**, we can only get 3 independent threads (on x, y, z axis). So we can also do a similar parallel spawn into 3 threads. This is not a good algorithm with high algorithm.

Nevertheless, all these steps (partitioning and scanning), becomes bucket sorts. For checking all the cuts, we can put all the triangles into 32 buckets, and then check these 32 buckets as super-triangle. The main time-consuming process is obviously bucket sort. For partitioning, it is equivalent to put all the triangles into two sets (subtrees), which is also a bucket sort.

We have did a project related with parallel bucket sort. Here the number of buckets is quite small (32 or 2), we can complete it in the following algorithm 4.

Algorithm 4: Parallel Bucket Sort

void parallelBucketSort(A)
For each Processor Do :
    Compute the information (number, bounding box)
        of each bucket separately in disjoint sets
Combine the information of different processors together
For each Processor Do : (Optional)
    Put the triangles into their positions separately

3.4.3 Parallel pseudocode

Here comes the our final parallel version:

Algorithm 5: Parallel BuildBVH

void buildBVHParallel($curNode, setTri$)
// $curNode$: current Node
// $setTri$: the set of all triangles in this set, represented by barycentric center
set $N \leftarrow |setTri|$
if ($N < threshold$) buildBVH($curNode, setTri$)
$bestCut \leftarrow \infty$
Parallel Bucket sort in x-axis, check 31 gaps and update $bestCut$
Parallel Bucket sort in y-axis, check 31 gaps and update $bestCut$
Parallel Bucket sort in z-axis, check 31 gaps and update $bestCut$
Parallel Split $\{setTri\}$ into $\{leftTri\}$ and $\{rightTri\}$
cilk_spawn buildBVH($leftNode, leftTri$)
buildBVH($rightNode, rightTri$)
cilk_sync

4 Ray Tracing Details

Optical ray tracing describes a method for producing visual images constructed in 3D computer graphics environments, with more photorealism than

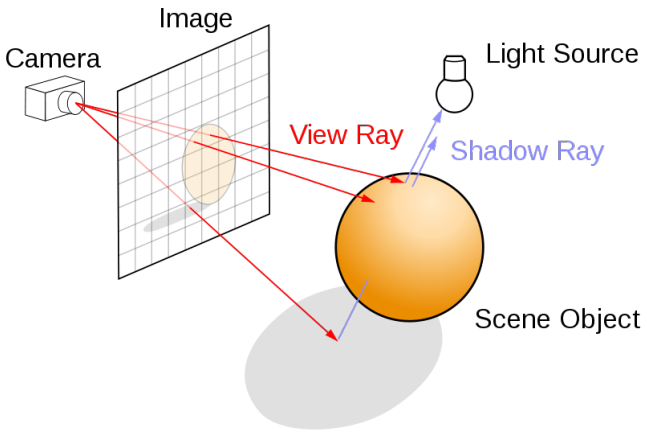

Figure 4: The ray tracing algorithm builds an image by extending rays into a scene.

either ray casting or scanline rendering techniques. It works by tracing a path from an imaginary eye through each pixel in a virtual screen, and calculating the color of the object visible through it.

Scenes in raytracing are described mathematically by a programmer or by a visual artist (typically using intermediary tools). Scenes may also incorporate data from images and models captured by means such as digital photography.

Typically, each ray must be tested for intersection with some subset of all the objects in the scene. Once the nearest object has been identified, the algorithm will estimate the incoming light at the point of intersection, examine the material properties of the object, and combine this information to calculate the final color of the pixel. Certain illumination algorithms and reflective or translucent materials may require more rays to be re-cast into the scene.

It may at first seem counterintuitive or "backwards" to send rays away from the camera, rather than into it (as actual light does in reality), but doing so is many orders of magnitude more efficient. Since the overwhelming majority of light rays from a given light source do not make it directly into the viewer's eye, a *forward* simulation could potentially waste a tremendous amount of computation on light paths that are never recorded.

Therefore, the shortcut taken in raytracing is to presuppose that a given ray intersects the view frame. After either a maximum number of reflections or a ray traveling a certain distance without intersection, the ray ceases to travel and the pixel's value is updated.

The process can be illustrated by Figure 4.

The pseudo code is shown in **Algorithm 6**.

Algoritm 6: Ray Tracing

```
TracingIntersectColor(vBeginPoint, vDirection)
{
    Determine IntersectPoint
    Color = ambient color
    for each light
        color += local shading term
    if (surface is reflective)
        color += reflect Coefficient × IntersectColor(IntersecPoint, Reflect_Ray)
    return color;
}
```

However, there still exists a problem of determining intersect points. We now give the pseudo-code to check the first intersection point with all the triangles in **Algorithm 7**.

Algorithm 7: Traversal in BVH

```
RayTreeIntersect(Ray, Node)
    if (Node is NIL || Dist(Ray, Node) > firstDist) return
    if (Node is a leaf)
    then
        check intersection in each primitive in the list
    else
        if (Ray hits LeftNode && doesn't hit RightNode)
            RayTreeIntersect(Ray, leftNode)
        else
        if (Ray hits Right Node && doesn't hit Left Node)
            RayTreeIntersect(Ray, rightNode)
        else
            Near ← the nearer subtree in { LeftNode, RightNode }
            Far ← the farther subtree in { LeftNode, RightNode }
            RayTreeIntersect(Ray, Near)
            RayTreeIntersect(Ray, Far)
    end
```

5 Result

For the two main parts in our algorithm, namely, building a BVH and ray tracing, we paralleled both algorithm and tested the performance improvement. We also give our result of the ray tracer.

5.1 Parallel Building a BVH

We parallel the algorithm of building a BVH with 32 cores on different models. We get the models from `http://graphics.stanford.edu/data/3Dscanrep/`.

The result is shown in Table 1. In the table we give the name of the model, and the models can be found from the above url.

model name	number of triangles	sequencial	paralleled	speed-up
Crytek Sponza	132267	1.574948	0.355206	4.4339
Chinese Dragon	871306	11.75276	2.399263	4.898489
Happy Buddha	1087474	15.11632	3.479823	4.34399

Table 1: Time in Building a BVH

We can use a column diagram to illustrate the result in Figure 5.

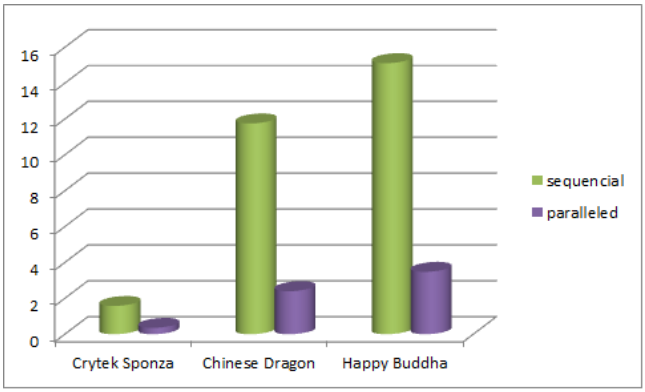

Figure 5: Time in Building a BVH

We can see from the table that with a model of 1×10^6 triangles, the algorithm will give a result in about $3.5s$. Actually, for commonly used models, they usually contain 50,000 - 80,000 triangles. On that order of magnitude, the time will be within $0.2s$, which is almost real-time.

Also, paralleled version is about 5 times faster than the sequential version, which demonstrate the advantage of parallel computing.

5.2 Parallel Ray Tracer

We also paralleled the process of ray tracing. We use 3 different models with different focal length(6mm and 13mm). The first two scene are 600×800 and the third one is 1920×1080. We also tried openmp and cilk++. We use openmp in *4 threads, dynamic, 10*

The time is recorded as the following table.

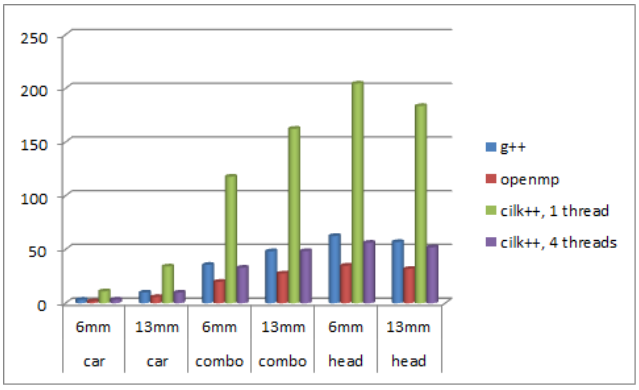

Figure 6: Time in ray tracing

model name	focal length	g++	openmp	cilk++, 1 thread	cilk++, 4 threads
car	6mm	3.212083	1.877977	11.0706	3.29703
car	13mm	9.91627	5.761973	34.19067	9.860459
combo	6mm	35.72907	19.93217	117.8203	33.05656
combo	13mm	48.44558	27.7024	162.4477	48.5733
head	6mm	62.55285	34.93843	204.5045	56.2839
head	13mm	56.98803	31.9601	183.5668	52.18083

We can see cilk++ may have a improvement of over 4, and *openmp* will have a better performance than the others. A column diagram to illustrate the result is as Figure 6.

5.3 Ray Tracer Result

For our final version, we can do rendering on the input model. Here are some output images of our algorithm.

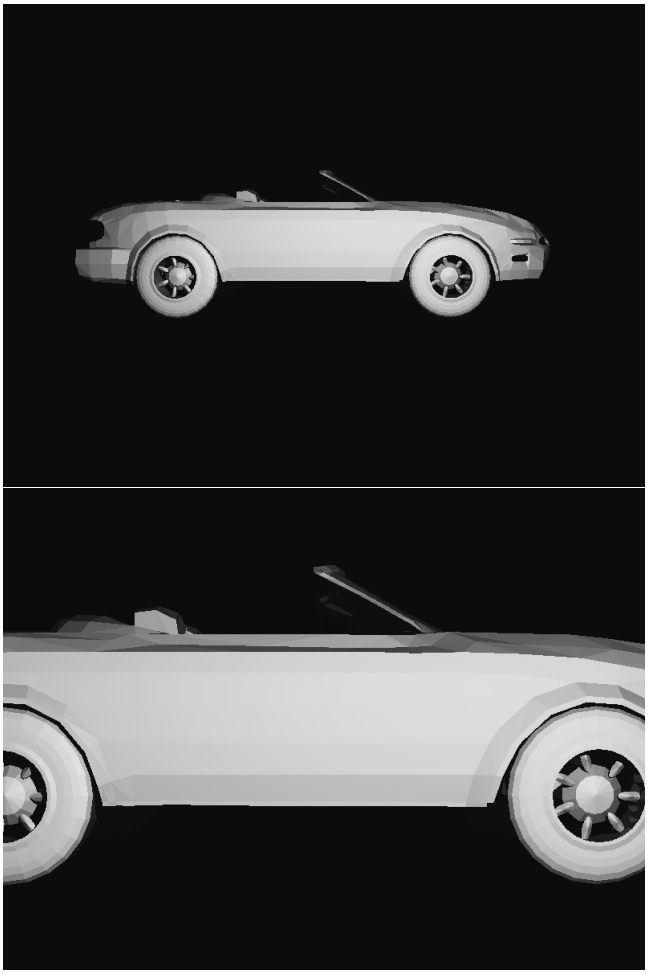

Figure 7: The result on model *car* with different focal length

References

[1] A.S. Glassner. An introduction to ray tracing. Morgan Kaufmann, 1989.

[2] H.W. Jensen. Global Illumination using Photon Maps. Rendering Techniques '96 , pages 21-30, 1996.

[3] Ingo Wald. On fast Construction of SAH-based Bounding Volume Hierarchies. SIGGRAPH 2007.

[4] Ingo Wald, and Vlastimil Havran. On building fast kd-Trees for Ray Tracing, and on doing that in $O(NlogN)$. Interactive Ray Tracing 2006.

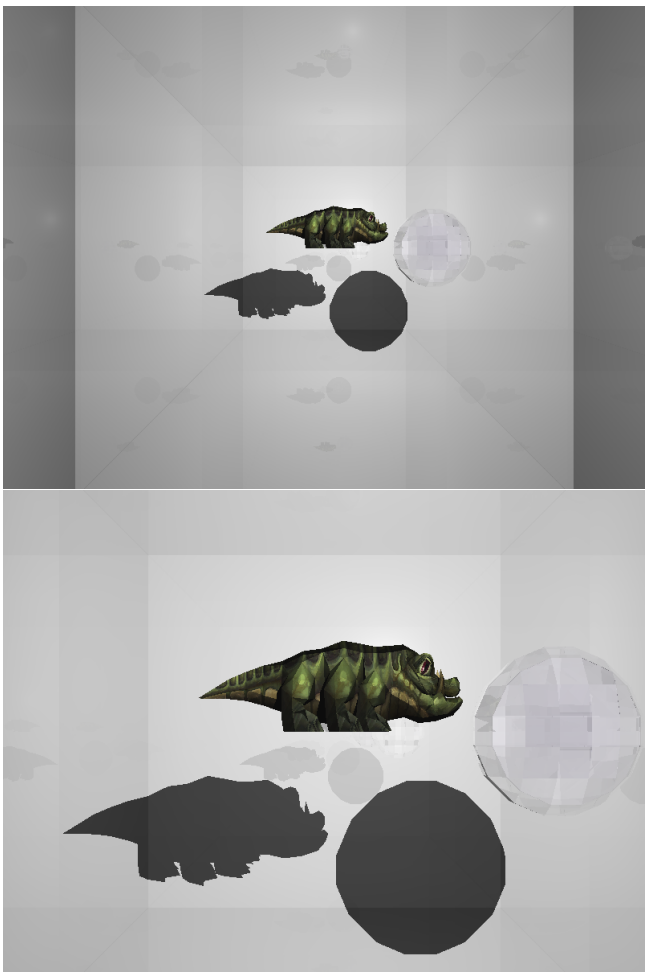

Figure 8: The result on model *combo* with different focal length

[5] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. SIGGRAPH 2006, ACM Trans. Graph. 26, 1, Article 6.

[6] Ingo Wald. Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. Interactive Ray Tracing 2007.

[7] Tero Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. High Performance Graphics 2012.

[8] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. Visual Computer, 6(6):153C65, 1990.

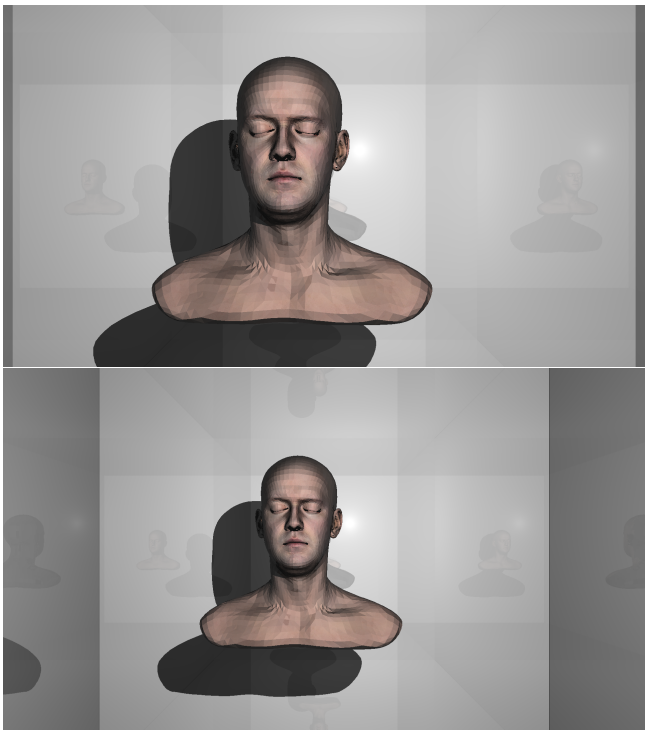

Figure 9: The result on model *head* with different focal length