České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Eva Mayerová**

Studijní program: Otevřená informatika
Obor: Počítačová grafika a interakce

Název tématu: **Interaktivní vizualizace hierarchií obalových těles**

Pokyny pro vypracování:

Prostudujte metody vizualizace datových struktur a grafů. Navrhněte metodu interaktivní vizualizace hierarchií obalových těles (BVH). Soustřeďte se na vizualizaci jak statických vlastností BVH (hloubka, plochy obálek), tak i dat získaných měřením (např. počet navštívení daného uzlu při výpočtu průsečíků). Implementujte aplikaci využívající jazyk C++ a OpenGL pro interaktivní náhled na vizualizovaná data. Implementace bude vizualizovat topologii stromu a související data a zároveň umožní zobrazit vybrané uzly BVH v prostoru zpracovávané 3D scény. Implementaci realizujte tak, aby bylo možné interaktivně studovat i rozsáhlé hierarchie obsahující několik miliónů uzlů. Implementaci otestujte na nejméně pěti scénách a souvisejících BVH získaných různými algoritmy stavby BVH (vlastní BVH i naměřená data poskytne vedoucí práce). Vybrané výstupy z vizualizace zdokumentujte v diplomové práci.

Seznam odborné literatury:

[1] BARLOW, T. AND NEVILLE, P., 2001, A comparison of 2-D visualizations of hierarchies. IEEE Symposium on Information Visualization, pp. 131-138, 2001

[2] GRIBBLE, C., FISHER, J., EBY, D., QUIGLEY, E., LUDWIG, G. Raytracing visualization toolkit. Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D, pp. 71-78, 2012.

[3] DACHS, L. Visualization of Spatial Data Structures. Diplomová práce ČVUT FEL, 1999.

[4] BITTNER, J., HAPALA, M., and HAVRAN, V. Fast insertion-based optimization of bounding volume hierarchies. Computer Graphics Forum 32, 1, pp. 85-100, 2013.

Vedoucí: doc. Jiří Bittner Ing., Ph.D.

Platnost zadání: do konce zimního semestru 2017/2018

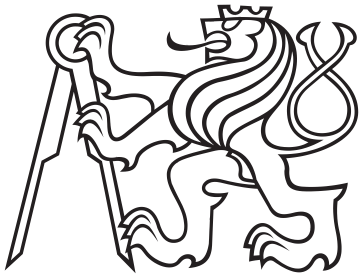
L.S.

prof. Ing. Jiří Žára, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 29. 2. 2016

Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering

Department of Computer Graphics and Interaction

Interactive visualization of bounding volume hierarchies

Bc. Eva Mayerová

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Field of study: Computer graphics and interaction

Subfield: Open informatics

May 2016

Acknowledgements

I would like to thank doc. Ing. Jiří Bittner, Ph.D. for very useful advices and professional attitude. Next, I would like to thank my dear friend Tomáš, who helped me at the very beginning and with the final corrections. The biggest thank goes to my love Vratislav, who was patient with me and was the best support while I was working on this thesis. Finally, I would like to thank my parents and family for their support throughout my student years.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 26th May 2016

Abstract

The thesis deals with current state of 3D hierarchical data structures visualisation, especially the bounding volume hierarchies (BVH). It summarizes current solutions and based on this comes up with a new one, which complements the shortcomings. The result is an application written in C++, based on libraries OpenGL 4.3 and Qt 5.5. The application allows the user to view created hierarchical data structure for given scene. The BVH is displayed either as a 2D view (tree) or directly as a part of the scene. The structure can be observed on the basis of various parameters, that are displayed using given scalar values, as for example the size of bounding volume or number of triangles per sub-tree. These values are generated by the application, but it is possible to import any other. Finally, application is able to display more bounding volume hierarchies for one scene, so user can compare their quality.

Keywords: BVH, bounding volume hierarchy, visualization, ray-tracing

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Abstrakt

Práce se zabývá analýzou současného stavu vizualizací 3D hierarchických datových struktur, obzvláště pak hierarchií obalových těles (BVH). Shrnuje současná řešení a na jejich základě navrhuje nové, které doplňuje jejich nedostatky. Výstupem práce je spustitelná aplikace v jazyce C++ využívající knihovny OpenGL 4.3 a Qt 5.5. Aplikace umožňuje interaktivní prohlížení vytvořené hierarchické struktury nad scénou a její zobrazení jak ve 2D pohledu (strom), tak přímo ve scéně. Strukturu je možné pozorovat na základě rozličných parametrů, které jsou dány skalárními hodnotami, jako například velikost obalového tělesa a počet trojúhelníků v podstromu. Tyto hodnoty jsou pro zadaný strom generovány aplikací, ale je také možné importovat jakékoliv další. Pro jednu scénu je možné prohlížet několik datových struktur zároveň a porovnávat tak jejich kvalitu.

Klíčová slova: BVH, hierarchie obalových těles, vizualizace, ray-tracing

Překlad názvu: Interaktivní vizualizace hierarchií obalových těles

Contents

Figures

Tables

Chapter 1

Introduction

Computer graphics is a discipline that uses a computer to generate graphical output. Input data used for the rendering are modeled, described and organized in some data structure. In computer graphics, the emphasis is usually placed on various parameters depending on the application. In game development we prioritize the rendering speed at the expense of quality. In contrast with this, architecture visualization or another field of industry needs photo-realistic results and emphasises the quality of rendered images. For this purpose some advanced rendering techniques are needed.

1.1 Bounding volume hierarchies in rendering

Before we describe the data structures that hold the geometry, the basic rendering techniques are mentioned. The methods are usually classified into two basic approaches. Rasterization and ray tracing.

Rasterization method renders an image using graphical primitives that are passed through the graphics pipeline. The vertices, that are at the input of this process are interpolated by groups according to the needed shape (usually the triangles) and transformed into the raster space. These raster points then become a fragments (the pixel candidates). It depends on the other factors, especially if another object covers them, whether they become the pixels and are actually rendered on the screen [Scrb]. Rasterization is very fast and it is often used for real-time graphics applications.

Figure 1.1: Ray-traced image by NVIDIA® OptiX™ Ray Tracing Engine [Nvi09]

The method important for this thesis is ray tracing. Ray tracing casts rays from camera to scene, one for each pixel. The scene is tested for an intersection with the ray. When the ray hits some object, it needs to be tested if it is lit or lies in the shadow. According to many factors (that depend on the actual chosen model) such as the reflection or the refraction coefficient, colour of the object and the light source or the ambient and emissive light, the final colour of the pixel is calculated. An example result obtained by ray tracing is shown in the figure 1.1. In contrast with rasterization, this method naturally simulates shadows and reflections.

Numerous data structures are used to hold the geometry. They can be classified into the spatial subdivisions or the object hierarchies. These structures are explained in detail in chapter 2.

The bounding volume hierarchy, hereinafter referred to as BVH, is a hierarchical data structure used to store a geometric data. Each primitive is wrapped into a bounding volume that can be defined variously. It should be chosen as the best compromise between the intersection test complexity and the approximation of real shape. In our case, the bounding volume is chosen as axis aligned bounding box. The idea can be seen at figure 2.4, taken from [Wik].

The construction of BVH has a crucial impact on its efficiency. Bad construction can cause for example an unbalanced number of intersection

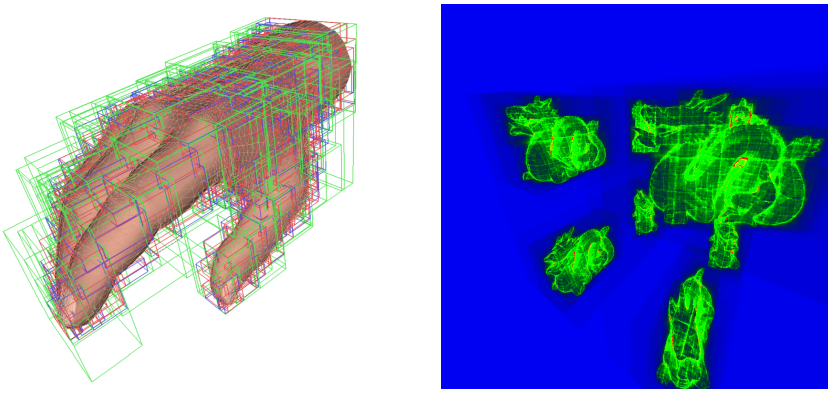


Figure 1.2: BVH structure visualization (left) [GFE+12], ray traversal visualization (right) [RMP15]

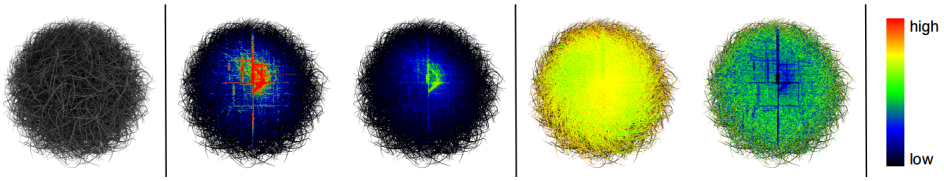


Figure 1.3: One of BVH traversal visualizations. [AGGW15]

tests in the sub-trees. To avoid these situations, it is needed to understand what defines the quality of created BVH and measure it. There are many ways to visualize the hierarchies. In the chapter 3, we will discuss these methods in detail. To see the example of possible visualization of the structure and a ray traversal, see the figure 1.2 or 1.3.

1.2 Goals of the thesis

Main goals of this thesis and following chapters will be summarized now. The chapter 2 describes the theoretical background, which is important for understanding next parts of the thesis. It describes acceleration data structures used for ray tracing and the bounding volume hierarchies more in detail.

One of the goals is to discuss current solutions of hierarchy visualization. Chapter number 3 describes some of the usually used tree visualisations and compares them with each other. Finally it summarizes the observations and

comes up with a solution suitable to our needs.

The design of application functionality, data formats and GUI are described in chapter 4. It describes the needed functionality. From a given BVH, the application creates a 2D view of the tree, so it is possible to see its topology. Each node can also hold some scalar value, either dynamic or static, eg. the number of triangles in the sub-tree. These values are displayed as a different node colours by using predefined colour mapping. The visualization has to be interactive, therefore it is allowed to scroll and move over the tree. In a closer look, user can see a different information that may not be visible in full view. The visualization of the scene is also important. The scene is imported into the application together with the BVH data. When the user selects one of the nodes in the 2D view, the corresponding area in the scene is highlighted.

Next, the chapter 5 describes the implementation details such as the data representation, the OpenGL usage, GUI and more. Some advanced techniques, like a triangle picking, which uses a ray tracer, are also described here.

The last chapter (6) depicts the created application from the user view. All possible options, that can be modified in the application, are listed together with some performance tests.

Chapter 2

Background

2.1 Ray tracing acceleration data structures

The ray tracing rendering times are getting significantly worse with increasing number of scene objects, when used naively. The naive algorithm casts rays from the camera and does an intersection tests with all of the triangles in the scene. Therefore it is necessary to use some acceleration data structure to reduce the number of these tests.

2.1.1 Object vs. spatial division

The acceleration is based on the division of the scene into smaller chunks, on which the intersection tests are faster then on the whole scene [Scra]. Instead of testing intersections with all of the triangles, we do intersection tests with their bounding volumes. The possible bounding volume types are discussed in section 2.1.2. Usually, the object division can be classified into two groups.

First of these groups is called **space partitioning**. The scene bounding volume is divided into smaller non-overlapping regions that fully cover the original spatial region. The cutting primitive is determined by chosen data structure and other properties. Let us now shortly describe some of the spatial partitioning data structures.

- **KD-tree** is a binary tree, where every node is k-dimensional point. Every non-leaf node represents a hyperplane, that separates the actual

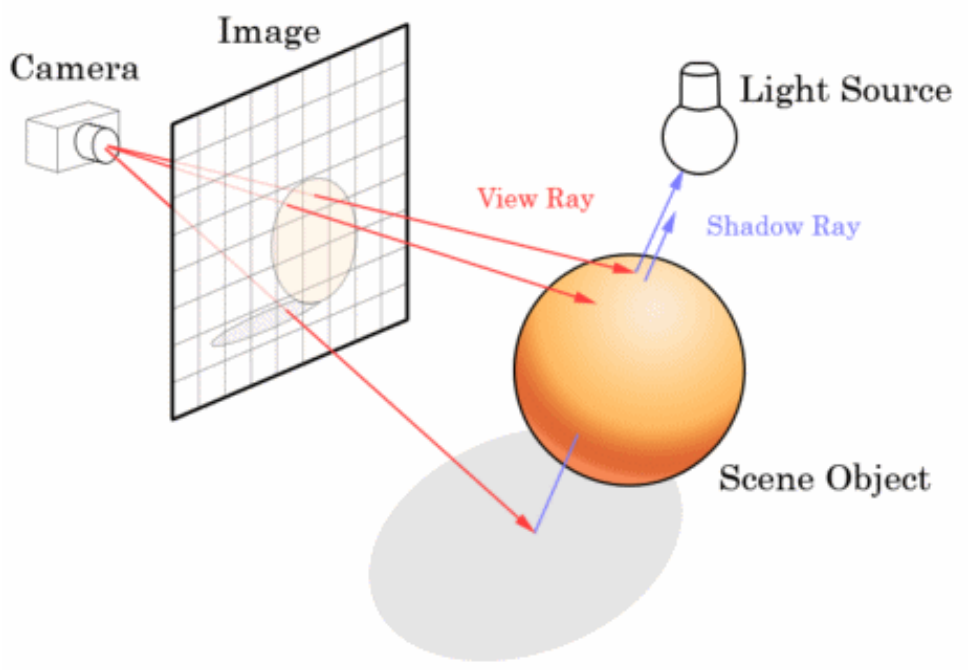


Figure 2.1: Ray-tracing scheme

space into two regions. There are multiple heuristics to determine the splitting plane and the splitting axis. It is widely used in computer graphics for its logarithmic traverse, deletion and insertion time.

- **Uniform grid** separates space into a regular 3D boxes. The intersection tests are only performed on primitives, that lies in the cubes, that are traversed by the ray. The cell size is chosen according to the needs. Large cells contain too many primitives and small cells may lead to too many primitives spread over multiple cells. Uniform grids may be extended into hierarchical regular grids similairly to other space partitioning structures.

- **Octree** is a 3D hierarchical data structure in which each interior node has exactly eight children (Quadtree in 2D has four children in each interior node). Octrees are less memory demanding than hierarchical regular grids, because subtrees with no primitives inside are collapsed into leaves.

Next division group is called **object subdivision**. In contrast with spatial

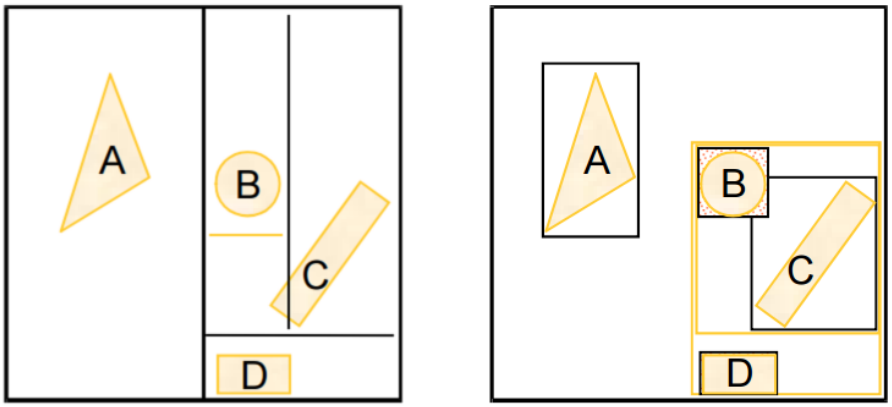


Figure 2.2: Spatial subdivision (left) and object subdivision (right)

subdivision, the region is not divided into new ones that completely cover it. The primitives in a node are split into two groups. Afterward a bounding volume is constructed for each group. These volumes, possibly overlapping, form a new child nodes of the original one. This subdivision is also called a Bounding Volume Hierarchy. For more details about this structure see the section 2.2. The image 2.2 from [Hav15] makes the difference more clear.

2.1.2 Bounding volumes

There are many types of bounding volumes that can be used to encapsulate the object. More complex shapes may give better approximation of the real object shape, but the intersection test times and memory consumption are worse.

Bounding spheres are the simpliest bounding volumes with the respect of computing an intersection with a ray. The stored information consists only of the sphere radius and the coordinates of its center. An intersection test with the ray is really quick, but the culling does not correspond with the original shape well in most cases.

A good compromise between intersection test time, memory consumption and culling precision of an object is an axis-aligned bounding box (hereinafter AABB). The data that have to be stored consist only of the minimal and

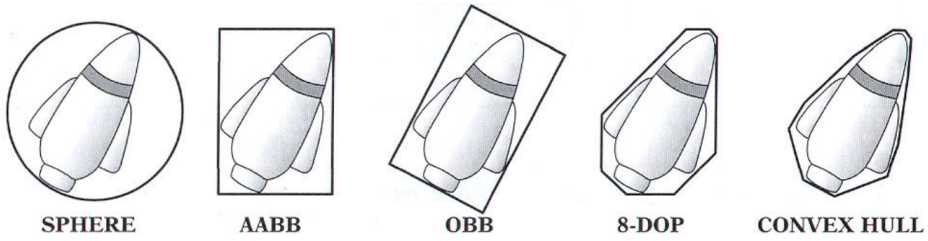


Figure 2.3: The mostly used bounding volumes. [Eri05]

the maximal coordinates in each axis. Alternatively, center coordinates and extension in each axis may be used.

Next object, called oriented bounding box, is similar to AABB, but it is not aligned to the coordinate system axes.

The k-dops are better in approximation of the original shapes, but the memory consumption is not negligible. The intersection tests are also more complicated.

Convex hulls, of all the mentioned methods, are the best at approximating the original shape. The intersecion tests with rays are much more expensive. Their construction is also quite difficult.

Overview of these structures can be seen on figure 2.3. In our application, we expect the bounding volumes to be the AABB.

2.2 BVH

Bounding volume hierarchy is a hierarchical data structure with bounding volumes in interior nodes. The whole scene is located in a root and is recursively separated into smaller parts that form the sub-trees. The division of the scene depends on the selected bounding volume.

2.2.1 Construction methods

The tree can be constructed in many ways. A splitting is determined by the SAH (surface area heuristic). This heuristic defines the cost function, that should be minimized. Cost of a node is the sum of the traversing cost, the cost of incident operations and cost of accessing the data in memory, each

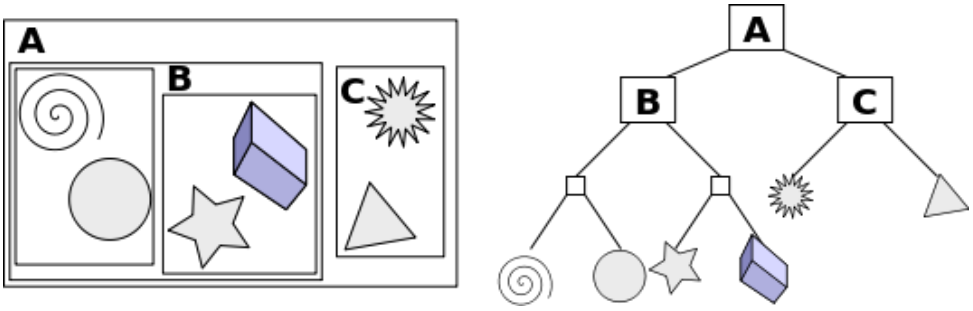


Figure 2.4: Bounding volume hierarchy with the axis aligned bounding box as a bounding volume

multiplied by the number of these operations [MB90]. Three basic methods of a BVH construction follows.

Top-down

The tree is constructed recursively from the whole set of data. It is quite similar to KD-tree with the difference that two sub-trees can overlap. The construction is done recursively by splitting the current node geometry into two smaller groups. The recursion stops when the group contains less then a specified number of primitives, the tree depth reaches a specified limit or any other criteria. The time complexity of tree construction this way is $O(n \log n)$.

Bottom-up

First, bounding volumes of all primitives separately are constructed to form leaf nodes. Then, in each iteration, multiple nodes are grouped. The algorithm stops when all nodes are connected under single root. Nodes are grouped based on their distances to each other. This method is more difficult to implement, but may produce trees with better quality in general. Time complexity is up to $O(n^3)$.

Incremental

On-line method, that builds the tree by inserting one primitive at a time. Each primitive is inserted into partially constructed tree. It becomes a leaf, which may also produce additional interior nodes. The time complexity is similar with the top-down construction, but the quality of the tree may be worse. The construction time complexity is $O(n \log n)$.

2.2.2 Traversal

The traversal means to find the closest primitive that intersects the ray. Several methods can be used. Typically, a depth-first traversal method is used when ray casting. The goal is to reach the closest primitive as quickly as possible to minimize the traversal cost [FLF12]. There are some heuristics to determine the child node which should be traversed first. For example, deciding according to the distance to the children node centres.

- **Stack-based** method works recursively with maintaining a stack. This method is often used for ray tracing. The problem can occur, when the rays are traced in parallel with limited memory (such as on GPU). The cost of maintaining a full stack for each ray can be very high [fAT14].
- **Stackless** method. Iterative method, where stack is usually replaced by a state logic [HDW+11].

2.2.3 Properties determining the optimal BVH

When creating a tree, we put emphasis on certain characteristics, that are important for the optimal use with ray tracing. These properties can be divided into two groups.

- **Static**

  Generally, the surface area of the envelope should be as small as possible relative to its volume. The tree balancing is also important to maintain the logarithmic depth. The balance can not be determined solely by the

Algorithm Stack-based BVH traversal

```
procedure TRAVERSE(node, ray)
    if ray hits the node then
        if node is leaf then
            intersection test with all primitives inside
        else
            determine near and far child
            TRAVERSE(node.child[near], ray)
            TRAVERSE(node.child[far], ray)
        end if
    end if
end procedure
```

number of bounding volumes in the tree, but also by the distribution of triangles. Otherwise, rays will not be distributed evenly. This brings us to the dynamic properties.

- **Dynamic**

  Properties defined by this group are detected directly by running the ray tracing. It is eg. the average number of nodes traversed in search of the intersection per ray. Another important feature is the average number of traversed nodes per triangle.

Properties defined in both of these groups are visualized in the application, so that the programmer can evaluate the quality of the generated tree.

Chapter 3

Visualization of hierarchies

Hierarchies are visualized many ways. It depends of course on the purpose of the created tree. Nice comparison between 2D hierarchical visualizations can be found in the article [BN01]. The author defined four basic ways to show the hierarchy. The methods can be seen on figure 3.1, which has been taken from the mentioned article.

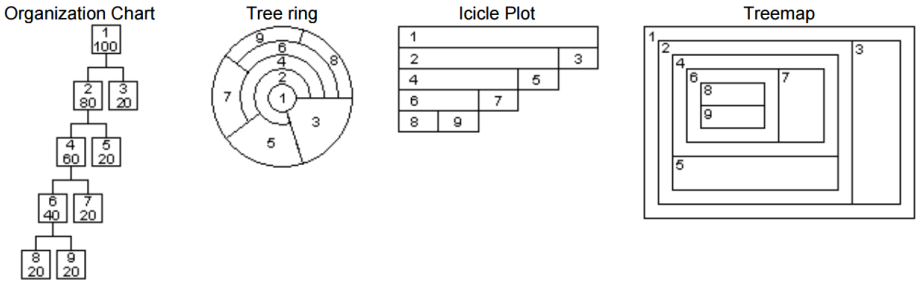


Figure 3.1: Different ways to visualize hierarchy in 2D

3.1 Organization chart

This representation is very intuitive for the visualizing of a tree. If a tree holds some complex information that should be displayed, this method can be found less useful when displaying the hundreds of leaves so the information could disappear. For our purpose we need only to display the colours, so this limitation should not interfere us.

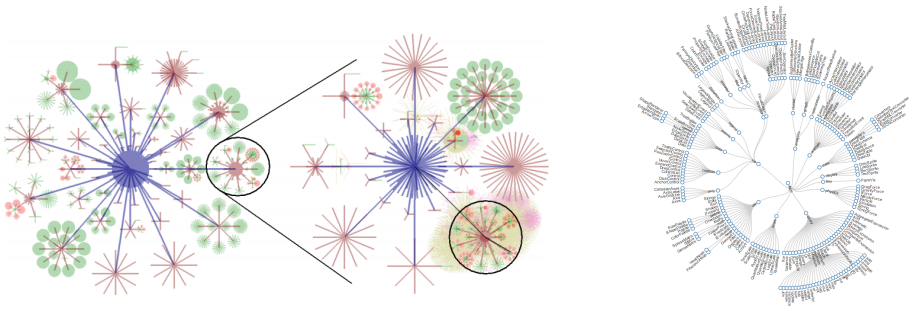


Figure 3.2: Tree ring visualizations

3.2 Tree ring

As author mentioned, this is a space-filling visualization method. It could show the real size of a node, relative to the others. Nice examples can be seen at figure 3.2, the left one found in the article [TM02], the right one at [Bos]. It is a good way to visualize the trees with larger arity (maximal number of children per node). Unfortunately, this is not a good representation for BVH, at least for our case, because the tree has mostly the arity of two. Secondly, the colour information in the leaves should be quite confusing for the sharp colour transitions between the non-neighbouring nodes.

3.3 Icicle plot

The icicle plot shows the node size as well as the tree ring, but unlike it, the icicle plot contains empty spaces. This could be really useful for the BVH visualization, since the arbitrary arity would work here.

3.4 Tree map

This is the filling-space method used for special purposes, such as a data mining. The size of nodes should be proportional to the object real size. The participants mentioned in the article [BN01] found this method the most confusing. It is not also a good representation of BVH because the overlapping of the nodes. Another visualization using tree map can be seen at 3.3. It was

Figure 3.3: Tree maps

mentioned in the article [HMM00]. Colours represent the same tree depth.

3.5 Summary

Based on an experiment with a number of respondents, the article [BN01] found the organization chart and the icicle plot as a two best rated visualization methods. According to our needs, we would follow the organization chart in a little combination with the icicle plot. There is no need to visualize the connections between nodes, it will be straightforwardly seen like in icicle plot, but the nodes at one floor will have the same width like in the organization chart, even if they are not actually the same size. For this purpose we have colours to visualize the size.

Chapter 4

Application design

4.1 Functionality

The application should allow the user to simultaneously view the scene and the associated BVH. The main part of the window are two widgets that draw on the screen using OpenGL functions. The first widget displays the 2D view of the tree and the second one the current scene. The widgets use the landscape orientation, because the BVH tree is also orientated landscape due to its logarithmic depth. The third part of the window is a control panel. The control panel GUI is described more in detail in 4.2. The layout design can be seen in figure 4.1.

The backbone of the application is its interactivity with the user. It allows to pick a node from the 2D view and show some statistics about it. It is needed to display its bounding box. After picking a node, its dimensions are displayed in the control panel and the node is visualized as a real box in the scene. The user can also click into the scene and pick one of the tree leaves. The path from the root to this leaf is displayed in the 2D view. The statistics about the leaf are displayed in the control panel as well.

The tree node colours are displayed using a specific transfer function. The input to the function is given by a scalar value, that is one of the tree properties. Some of the tree properties, that are used to measure its quality, were discussed in the previous chapter.

The application visualizes two kinds of data. The first are automatically

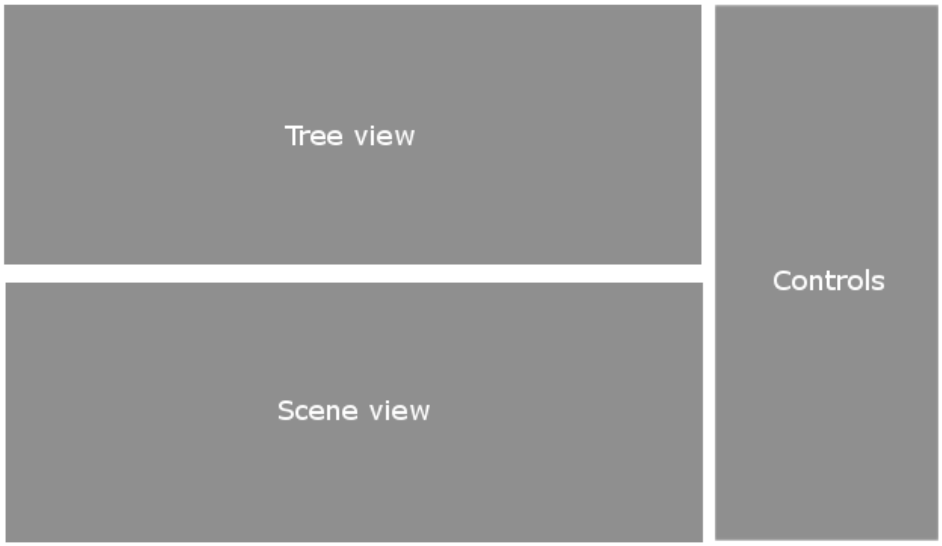


Figure 4.1: Main window layout

computed by the application when the BVH file is loaded. An additional values can be imported by the user. The computed values follow.

- Surface area of the bounding box relative to the surface area of the root
- Volume of the bounding box relative to the volume of the root
- Surface area of the bounding box multiplied by the depth of the node to the power of two, relative to the surface area of the root
- Volume of the bounding box multiplied by the depth of the node to the power of two, relative to the volume of the root
- Number of triangles in the sub-tree
- Sum of children's surface area relative to the parent's surface area
- Sum of children's volume relative to the parent's volume

The sum of children's volume relative to the parent's volume displays the overlapping of children. If this value is lower or equal to one, children probably do not overlap so much.

The user can manipulate with both OpenGL widgets as well. The 2D view of the tree can be moved by a right mouse button and scaled using a mouse wheel. The scene can be manipulated the same as the 2D view. Additionally, it can be rotated by holding a middle mouse button while moving it.

4.2 GUI design

The graphical user interface has to be clear and easy to understand. The most used features have to be situated at the top of the control panel. Features, that are important for the application are listed below.

- **Tabs to switch the BVH** are located at the top of the control panel and are used for the switching between more imported trees that corresponds to the scene.
- **Tree statistics** is the basic information about the tree, that the application immediately presents to the user. It contains information such as the number of triangles, the number of BVH nodes, the depth of the tree or the average number of triangles per leaf.
- **Displayed property** is another key factor of the application. There are multiple properties of the tree, so the user must be able to switch between them. The user can do so in a combo-box widget. He can also set the minimum and the maximum values using two sliders. These properties are in varying range and their values might not be distributed uniformly. As a consequence, the transfer function should not be linear. It is up to the user to its characteristics. This possibility should be provided by a spin box. The transfer function is described more in detail in chapter 5.5. This area also contains a button to load additional properties for the tree.
- **Blending function** controls the appearance of pixels, which are covered by multiple nodes. In various cases the user wants to see different data. For this purpose, he can switch between predefined blending functions,

such as a minimum, a maximum or an average value. These options are accessible by a radio buttons.

- **Display mode** changes the way the nodes are displayed. Available are four options.
  - Nodes displayed as a filled boxes. This is a default option.
  - Line representation, where only contours of a nodes are displayed.
  - Ellipses. In this mode each node is displayed as a filled ellipse constructed of 16 line segments at its circuit.
  - The line representation of ellipses.

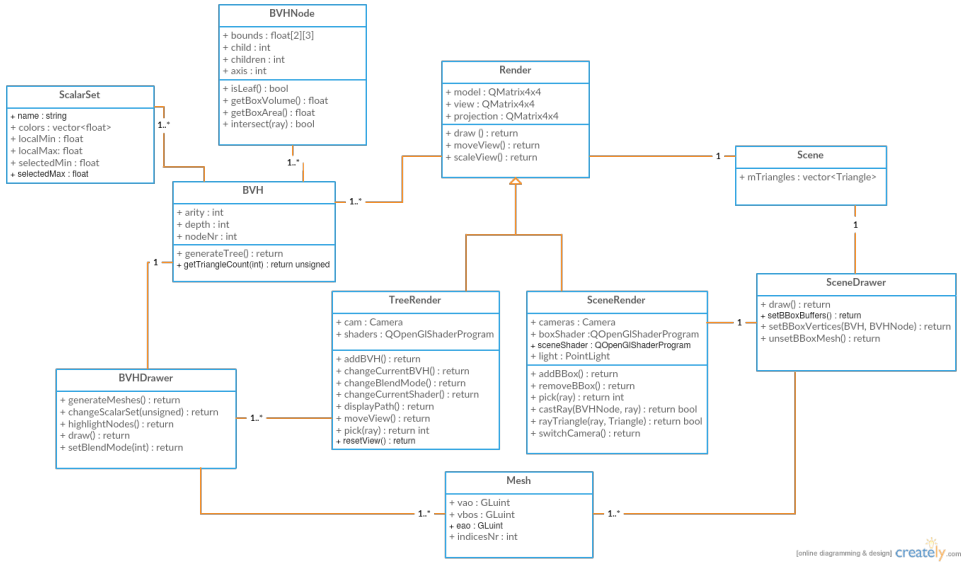
4.3 Input data format

The application loads all relevant data from a user specified file. The file contains both the information about the geometry and the already constructed BVH. The BVH is represented by an array of nodes, each carrying its bounding box and children indices. The BVH node structure can be seen in the listing 4.1.

Listing 4.1: BVH node struct

```
typedef struct
{
        float bounds[2][3];
        int32_t child;
        uint8_t axis;
        uint8_t isLeftCheaper;
        int16_t children;
} BVHNode;
```

The `bounds` define the maximum and the minimum coordinates of the bounding box in all three axes. The `axis` variable defines the current splitting axis. The value three means the node is a leaf. The `child` is an array index of



the first child. The other children are organized sequentially following the first one. If the node is a leaf, it is an index of the first primitive associated with this node. The variable `children` defines the number of children nodes (or the primitives). The attribute `isLeftCheaper` is not used in the application.

4.4 Object design

The application window consists of two panels that use OpenGL functions and one panel with graphical user interface. The OpenGL panels are represented by two different classes that inherit from `QOpenGLWidget` class. Each of these widgets has its own OpenGL context. Furthermore, each has an instance of class `Render`, specifically their descendants — TreeRender and SceneRender.

Class `TreeRender` has an array of pointers to instances of class `BVHDrawer` and retains the index of the currently displayed `BVH`. Class `BVHDrawer` provides drawing functions. It owns an instance of class `Mesh`. Its OpenGL buffers are generated using class `BVH`, that contains the information about the tree. Class `BVH` contains an array of `BVHNode` instances and an array of pointers to instances of class `ScalarSet`. Class `ScalarSet` contains a scalar value for each BVH node related to the current tree, which serve as an input to the transfer function. These are the values visualized as BVH node colours.

The class `SceneRender` is defined analogically to the `TreeRender`. It contains a pointer to the instance of class `SceneDrawer`. The class `SceneDrawer` owns an instance of the class `Scene`, which holds the information about the geometry. It contains an array of triangles and their indices to connect them with respective BVH nodes.

Chapter 5

Implementation

5.1 BVH representation

The tree is stored in two structures. The first one is used to store the topology. It holds the information about children and bounding boxes. The second one is used for rendering and contains an array of the coordinates of the node centers. These are then processed in the geometry shader, based on the chosen display mode, to be rendered.

Positions of the nodes are calculated in the normalized device coordinates (NDC). The view matrix is always set to identity, because the camera does not move. The projection matrix is set to orthographic projection. The model matrix varies because of the user moving and zooming the tree.

Normalized device coordinates are in the range $(-1, 1)^3$. These coordinates are further transformed into the window space coordinates, which depends on the particular dimensions of the viewport. The calculation of node positions is made by the breadth-first search. The height of the node is calculated first and is constant for all of the nodes and determined by the maximum depth of the tree. The width of the nodes are computed individually. The width is given by the depth of the node and decreases exponentially as the depth increases.

The tree is not complete, therefore some levels may not be completely filled with nodes. Subsequently, it is not possible to calculate the horizontal positions of the nodes sequentially with constant increment for each level.

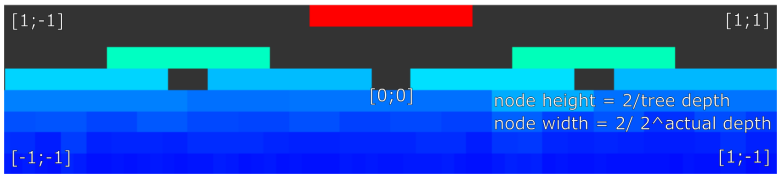


Figure 5.1: Tree with normalized device coordinates

Finally, the horizontal position of the node is computed as its parent's horizontal position with offset based on the number of the parents' children.

Example of the tree with the arity of two can be seen at figure 5.1.

5.2 Visualizing nodes at lower levels

The nodes at lower levels of the tree are computed to be thinner then one pixel with this method. The standard rasterization, which is a default option, leave these pixels empty. Empty spaces then appears at these places. They are only filled with growing zoom factor. This problem can be solved multiple ways. For example by rendering of the centroids of the nodes first, that are exactly one pixel wide. Unofrtunatelly, the centroids would not fill the whole height of a node and there still would be a lot of empty space in vertical direction.

Next option, that could solve this problem, is to use the conservative rasterization. This method draws even into the pixels, that are filled at least with small part by the primitive, not the majority [Sto14].

Finally, it is up to the user, which information he would like to display. He can choose between the average value of all pixel candidates, the minimum or the maximum. The corresponding blending function is set accoding to the needs. For more information about blending functions see the section 5.3.

5.2.1 Conservative rasterization

The width of individual nodes decreases as the depth of the tree increases. Let us assume, that screen is separated into pixels and every pixel is defined by its center. The pixels are usually treated as to be covered by the primitive,

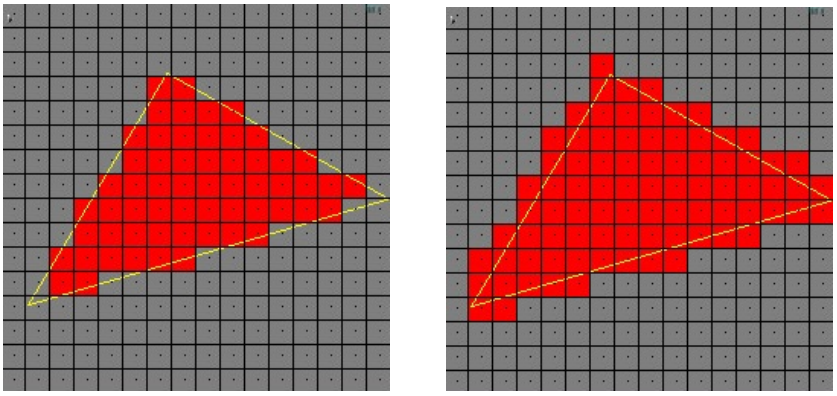


Figure 5.2: Standard rasterization (left) vs. the conservative rasterization (right)

when that primitive covers its center. To avoid this behaviour, we use a technique called conservative rasterization.

This technique renders also the pixels, that are at least partially covered by the primitive. The difference between the standard rasterization and the conservative rasterization can be seen at the figure 5.2.

This rendering method is available in the OpenGL as a vendor specific extension by NVIDIA. The use is simple, but the extension is dependent on environment. There is also a possibility to make it work without a hardware support by using a Geometry shader. In the article [JH] are described two algorithms, that can be used. These algorithms have different performance characteristics. Their common goal is to extend the given primitive by the semi-diagonal of a pixel.

- The **first algorithm** computes an optimal bounding polygon of given primitive. It is correct in filling the fragments, but it is also expensive. It completely works in the geometry shader program and the vertices must be replicated. The bounding polygon is computed by a splitting the problem into three cases, according to the direction of normals of edges adjacent to the given vertex. In the implementation, each edge normal is represented by the semi-diagonal of the pixel of the same quadrant, where the real normal is oriented. Next, the dot product of the semi-diagonals of the adjacent edges gives the information about

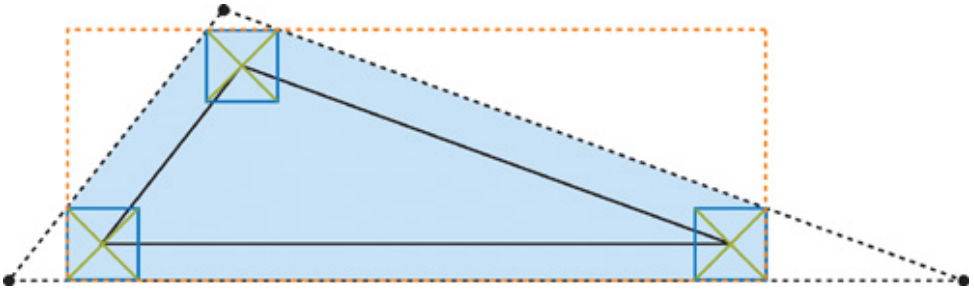


Figure 5.3: Intersection of bounding triangle and axis-aligned bounding box

interrelation between them. If the dot product is larger than zero, the normals are in the same quadrant. If it is less than zero, the normals are in the neighbouring quadrant and if it is equal to zero, the normals are in the opposite quadrants. Thanks to this information, we can now pass the corresponding number of vertices to fragment program.

- The **second algorithm** is less expensive, because it passes only three vertices out of the geometry shader program and discards the unnecessary fragments in the fragment shader program. This makes it less expensive, but it works bad with triangles with acute angles. The bounding polygon of the primitive is computed as an intersection of the bounding triangle and the axis-aligned bounding box of the primitive. The intersection is visualized in the figure 5.3. The bounding triangle is computed in the geometry shader program and then the fragment shader program discards the fragments, that do not overlap the axis-aligned bounding box. Computing the bounding triangle is done by moving each original edge by the worst-case semi-diagonal. That is the one in the same quadrant as the edge normal.

5.2.2 Implemented solution

The second algorithm is definitely better to our needs. It works faster, which is a key factor for application, that needs to work in real time. In our case, this algorithm can still be simplified.

Let us imagine the probably worst case input BVH. For millions of triangles,

it would have also millions of nodes. The depth of this tree would be maximally around many dozens. Of course, if the tree would be totally unbalanced, the depth can be up to half of the triangles. But do not consider these extreme cases. The depth of the tree will be much lower then the number of pixels of the window in vertical direction in almost every case. According to this fact, we can neglect the conservative rasterization of triangles in vertical direction.

Now, the solution is truly simple. Instead of three vertices generated in geometry shader, that are then passed to the fragment shader, we modify these vertices by moving the x-coordinate of each by the half x-size of the pixel.

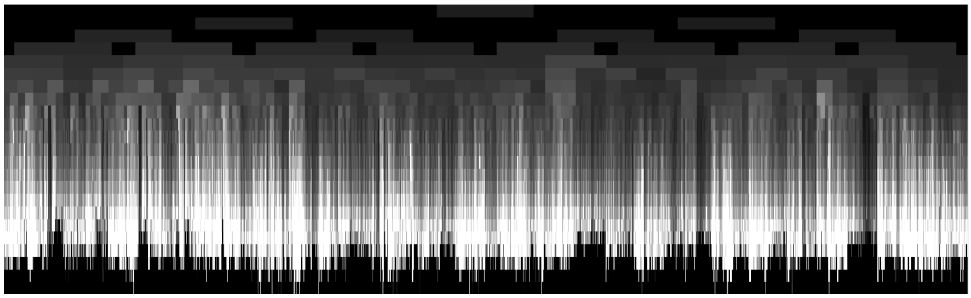
When using lines as the rendering primitives, the conservative rasterization is used by default.

5.3 Blending types

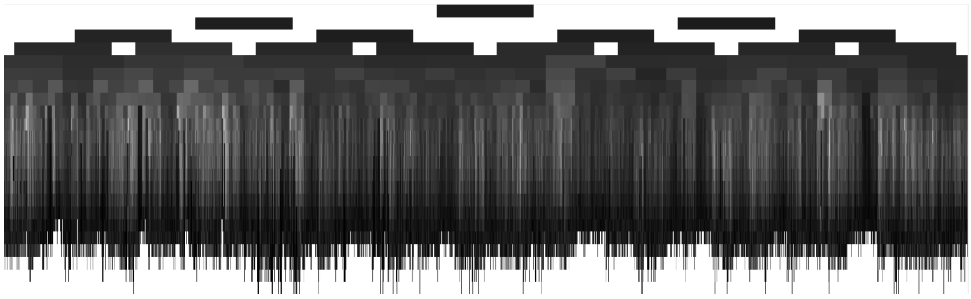
This section describes situations, where positions of more nodes than one are computed to cover the same pixel. The displayed value is up to the user. Application allows three types. The maximum, the minimum and the average value of the pixel candidates. Common solution of these methods lies in the rendering by multiple passes. First, the desired values are rendered to a texture, which is used in subsequent rendering on the screen. The textures with the maximum and the minimum values can be compared at figure 5.4. Notice the bottom part of the tree, where the difference can be seen the most. The values are in the range from zero (black colour) to one (white colour).

- **Maximum value**

  Values are rendered to a texture with enabled blending, which function is set to `GL_MAX`. The clear colour is set to zero value. The rendered texture contains float values in range $(0, 1)$. Next, the current frame buffer is set to the default screen buffer. In the fragment shader, we use the generated texture and put its values as an input values to the transfer function.



(a) : `GL_MAX`



(b) : `GL_MIN`

Figure 5.4: Textures with different blending functions

- **Minimum value**

  The process is the same as with the maximal value. The only difference is in the blending function, which is set to `GL_MIN` and the clear colour, which is, in this case, set to one.

- **Average value**

  To compute the average value, we need to store the information about sum of all contributors and their number. These two values are stored in a two textures. Therefore, the rendering is made by three passes. The first pass renders the sum of all contributors using the blend function `GL_ADD`. Next, the number of them is rendered using the same blend function. The difference is in the fragment shader, where the number one is added every time. In the third pass we render to the screen. The fragment shader program uses the two created textures and compute the arithmetic mean.

5.4 Picking

The application allows the interconnection between the scene and the respective BVH. When the user clicks into the scene, the appropriate BVH leaf and its triangles are highlighted. In the tree view the whole path from the root to the leaf is highlighted. Let us now describe the process of selecting the leaf in the scene.

- The clicked coordinates are recorded in the window space, i.e. in the range $(0, width)$ and $(0, height)$ of the window. These coordinates are transferred to the NDC, that are in the range $(-1; 1)^3$.
- The ray position is set to the point $[0; 0; 0]$ in camera coordinates. The target ray position is transformed to the camera coordinates by multiplying the window space coordinates the inverse of the projection matrix.
- Both of these parameters have to be transferred into a model coordinates. This is achieved by multiplying the coordinates first by the inverse of the view matrix and then the inverse of the model matrix.
- The ray direction is calculated by deducting the ray position from the ray target. These values are used as an inputs to the function `bool pick(ray)`. Now the ray tracing starts.
- A stack-based algorithm is used. Its pseudocode can be seen in the section 2.2.2. If the algorithm recursively reaches the tree leaf and the ray intersects some of the triangles inside, the bounding box of the leaf is displayed semi-transparently in the scene.

In the upper window, where the tree is displayed, the intersection calculation is simplified. Node positions are specified in the NDC coordinates. Therefore, it is just needed to convert the cursor position from the window space to the NDC and multiply it by the inverse of the model matrix. The nodes are traversed by the breadth-first search and tested for an intersection with the cursor position.

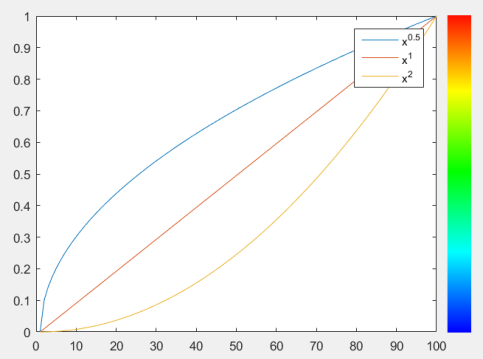


Figure 5.5: Transfer functions with corresponding colours in y axis

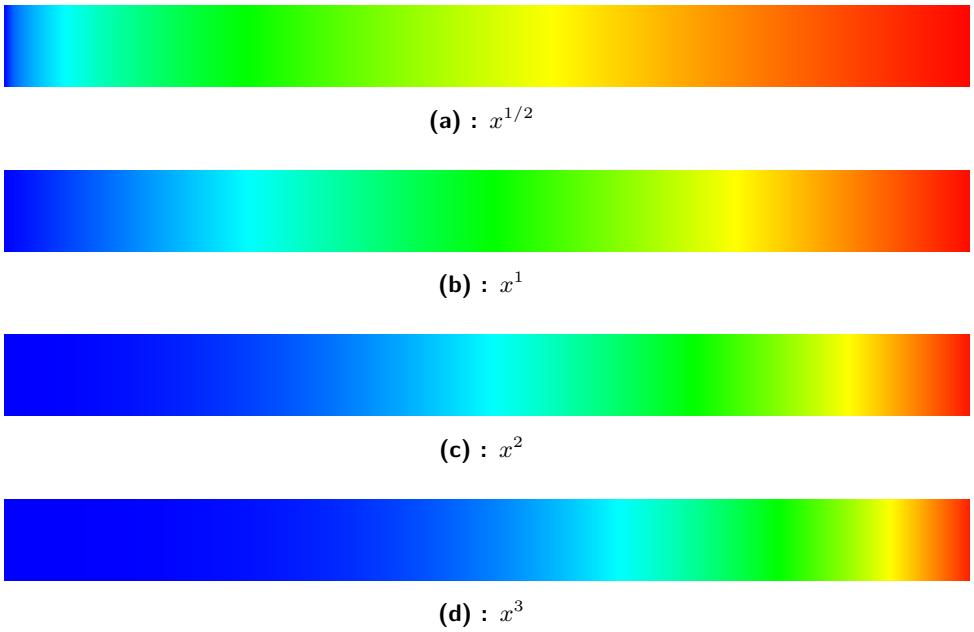
5.5 Transfer function

The scalar data defining the tree are generally not uniformly distributed. Subsequently, the median value does not have to be in the middle of data range. It is needed to have some more sophisticated transfer function than linear. The sensitivity, i.e. the difference between the function values of the nearby function input values, should differ in a different parts of the range. According to the fact, that we would like to define a function, that is increasing variously, i.e. the function value of a middle point has to be sometimes larger and sometimes smaller than the actual middle value, we use a **power function**. The power functions with various exponents are displayed at figure 5.5. The colours corresponding to the used colour palette see in the figure 5.6.

A slider that defines the bounding values of the transfer function is discretized into constant number of steps defined from zero to the maximum value. Following function is used to compute the real value from a slider. The `multiplicationConstant` defines the actual range of the scalar values divided by the maximal value of a slider. The `additiveConstant` defines the difference between zero and the bottom value of the range.

Algorithm Computation of power function value

```
function PowerFunction(value, exponent, maxValue)
    resizedVal = pow(value, exponent) / pow(maxValue, exponent - 1)
    return multiplicationConstant * resizedVal + additiveConstant
end function
```



(a) : $x^{1/2}$

(b) : x^{1}

(c) : x^{2}

(d) : x^{3}

Figure 5.6: Various exponents of transfer function

5.6 OpenGL usage

The OpenGL functions are used to rende into two main widgets of the application. The functions are already included in these widgets, because both of them inherit from the class `QOpenGLWidget`, which is using OpenGL functions by default. The current context is accessed by the call `this->makeCurrent()`, which has to be used at each `paintGL()` and `resizeGL()` event. This is caused by the switching between two widgets, which both have their own OpenGL context. The OpenGL functions can be accessed by Qt, which has its custom wrap to these functions. The use of pure OpenGL functions can be done by including them from a specific OpenGL version. In this case, we are using version 4.3, which is accessed by inheriting from the class

`QOpenGLFunctions_4_3_Core`.

5.6.1 Tree view widget

Geometry in this widget is generated by the BFS traversal. The computation of single nodes is described in section **??**. The data, that are used for the next processing on GPU, are located in a vertex buffer objects (VBO) and wrapped into a single vertex array object (VAO). These data serve as an inputs to the GLSL vertex shader program. The vertex shader input data consist of the node position, which is given only by the centroid of the node, scalar value for the current node, half width and half height of the node. The last two values are used in the geometry shader. Application works with several geometry shaders that are used to change the tree appearance.

The tree is rendered using points as the primitives. It is established by the call `glDrawArrays(GL_POINTS, 0, primitivesNr)`, where `primitivesNr` is the number of tree nodes.

Next, the MVP (model-view-projection) matrix is passed into the vertex shader as a uniform variable. The view and the projection matrices are both identities, so the MVP matrix is represented only by the model matrix.

The outputs of the vertex shader serve as the inputs of the geometry shader. This part is used to generate another primitives based on the input points. These primitives are then passed into the fragment shader. The type of output primitives depends on the selected tree appearance, but let us now discuss the version with the filled rectangles. The shader output primitive is set to the `triangle_strip` and the number of output vertices is set to four. The input values such as the half width of the node and the half height of the node are used to compute the positions of rectangle vertices. The corresponding scalar value is passed to the fragment shader and is constant for these four vertices.

Fragment shader uses textures defined in section 5.3. If the value passed from the geometry shader is equal to -1, it means that the working node is actually selected, so it is rendered white. In other cases, the texture is

used. One of the values passed from the geometry shader defines the fragment position in the NDC coordinates. This value is used as a texture look up coordinate. The corresponding value in the texture is then passed to the function that defines the colour palette.

After a tree rendering, the visualization of the current transfer function has to be rendered as well. It is done similarly. The vertex positions are generated at first. These are computed by dividing the window height by the number of steps in the sliders defined in the section 5.5. Each vertex passed into the vertex shader consists only of one float value, that defines the y-coordinate in the range $(-1, 1)$. The next input value is a pair of floats that define the scalar values in i^{th} and the subsequent vertex that serves as the inputs to the transfer function. In vertex shader, these values are without modification passed into the geometry shader. The vertices are modified there into a triangle strip. The width of the bar is defined here. Four vertices are passed into the fragment shader. The two bottom vertices are associated with i^{th} scalar value and the top two with the subsequent scalar value. The fragment shader maps these scalar values to colours defined by the colour palette.

5.6.2 Scene view widget

The scene is rendered more straightforward then the tree view. The call `glDrawArrays(GL_TRIANGLES, 0, primitivesNr * 3);` is used (`primitivesNr` stands for a number of triangles in the scene). There are multiple values that are used as the input variables of the vertex shader program. These values are a vertex position, direction of a normal and a diffuse material colour of current vertex. The uniform values are the model matrix and the MVP matrix. The model matrix is used to transform the vertex position from the model space to the world space. The world space coordinates of the vertex are used to compute lighting. The values passed to the fragment shader consist of the vertex position, the normal direction and the material colour. To compute the lighting, the Phong illumination model [Pho75] is

used.

If the user selects one of the nodes in the tree view, the next element rendered into the scene view is the bounding box of the selected node. The rendering starts with a call `glDrawElements(GL_TRIANGLES, indicesNr, GL_UNSIGNED_INT, NULL)` (`indicesNr` is the number of triangle vertices passed to the vertex shader program). The cube has 6 faces, therefore the number of triangles is 12. The number of triangle vertices is then 12*3. The lighting model is not computed. The vertices are only transformed by the MVP matrix. The bounding box is rendered semi-transparently, therefore the blending function is enabled.

5.7 GUI

The GUI elements used in the application are provided by the Qt library. The control panel design was described in the chapter 4.2. All of the graphical elements must be a children of any widget. Because of this, the class `ControlPanel` wraps all of the control panel elements. It has its own container, which is an instance of class `QWidget`. All of the children elements are attached to this container. Each widget has its own layout. The control panel elements are placed in the vertical box layout defined by a class `QVBoxLayout`. A layout class wraps the GUI elements, like the widgets or the items (i.e. `QSpacerItem` is used on the bottom of the control panel).

The individual parts of the control panel are defined by a structures. The elements of the control panel can be seen below.

```
class ControlPanel {
public:
        ControlPanel();

        QScrollArea *scrollArea;
        QWidget *container;
        CurrentTreeStats *treeStats;
```

```
        CurrentNodeStats *currNodeStats;
        ScalarValuesGUI *scalars;
        BlendingType *blendingType;
        DisplayMode *displayMode;
};
```

5.7.1 Signals and slots

To handle the events such as button press or change of a slider value, Qt uses signals and slots. It is an alternative to the callback technique. We can define, that some signal coming out of some object, is handled by a slot of another object. The signals have to be placed in the header files under an accessor `signals`, slots under an accessor `slots`. The signals and slots have to be connected in the application. The example follows.

```
connect(button, SIGNAL(released()), this, SLOT(handleButton()))
```

5.8 Dependencies

The main part of the application is made using the OpenGL 4.3.

The application uses the Qt library for managing the window system and the graphical user interface. Used version is the Qt 5.5 under a LGPL (GNU Lesser General Public License v. 2.1).

Chapter 6

Results

This chapter discusses the functionality of the application. The implementation met all of the declared features mentioned in the introduction. At the figure 6.1 see the final layout of the application. The individual parts of the window are described below. The numbers in the figure correspond to the sections in this chapter. The sections give more explanation.

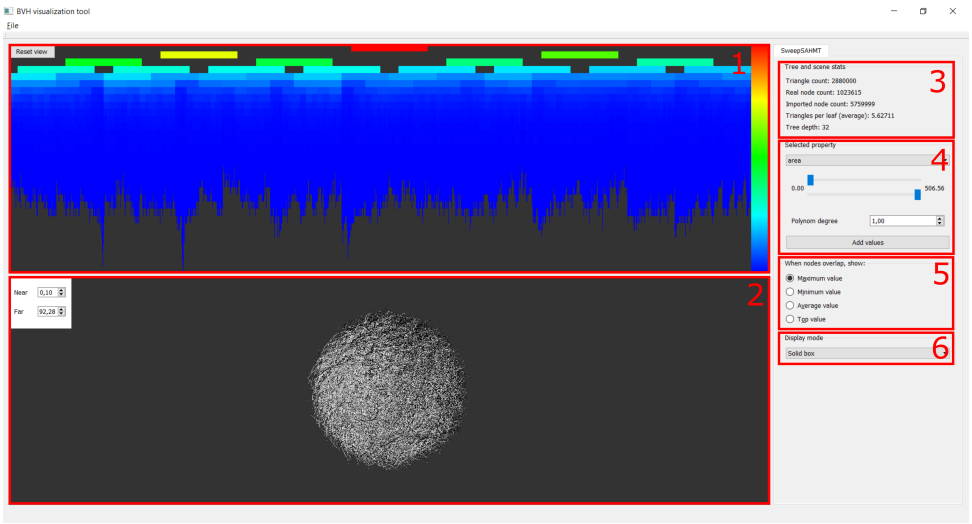


Figure 6.1: Screenshot of final application with the scene named Hairball

6.1 Tree view

This widget is the most important part of the window. It displays the imported tree and its topology. Nodes are displayed in different colours,

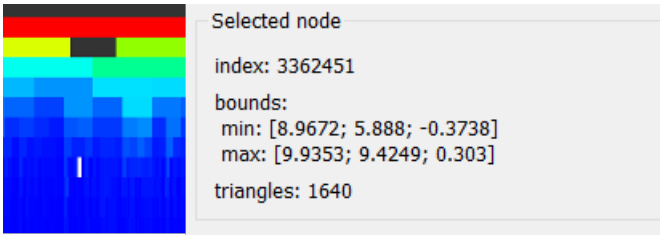


Figure 6.2: Displayed information about the selected node

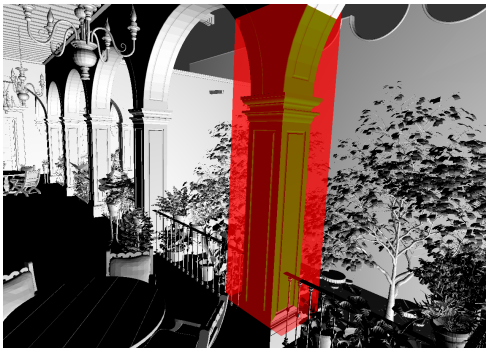


Figure 6.3: Detailed scene view with the selected bounding box

which are given by the transfer function. This function could be one of the precomputed by the application, such as the bounding volume, the surface area, the number of triangles, etc. or imported by the user. The user can manipulate with the tree. The middle mouse button can be used to zoom and see the details. The view can also be moved by holding the right mouse button.

Left mouse button is used to select the node in the tree. When the node is selected, it is highlighted by a white colour and the information is displayed in the control panel. What happens, when the user selects the node is shown in the figure 6.2. The bounding box is also displayed in the scene view. See the detail in the figure 6.3.

The right side of the tree view is filled by the visualization of the user-defined transfer function. The function is influenced by the selected bounds and the chosen function exponent.

The button called `Reset view` is located in the top-left corner of the tree view. This button sets the view to the initial position.

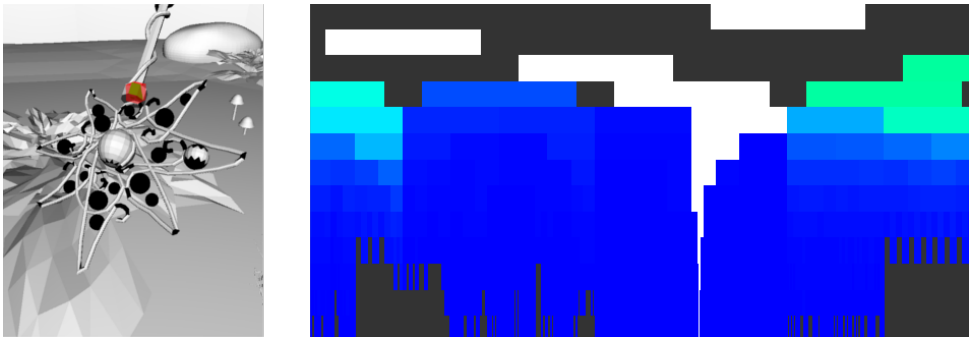


Figure 6.4: Selected BVH leaf in the scene with displayed path to the root in tree view

6.2 Scene

The scene view helps the user to better understand the partitioning of the scene. When the node is selected in the tree view, its bounding box is rendered into the scene view semi-transparently. If one of the triangles is selected by the left mouse button, the appropriate leaf bounding box is rendered. The path from the root to this leaf is displayed in the tree view together with the information about the leaf in the control panel. The selected node and the visualized path from the root can be seen at 6.4. The user can also manipulate with the scene. The middle mouse button can be used for zooming. The view can also be rotated by holding the middle mouse button. The example of the visualization of the bounding box of the scene see in the figure **??**.

The group of GUI elements, that can modify the scene view, is located in the top-left corner of the widget. The button called `Next view` is displayed if there is more then one camera defined for the current scene. The button switches between the defined cameras. Two spin boxes are located bellow. The one is called `Near` and the second `Far`. The values set in these boxes define the near and the far plane of the view frustum. The far plane is initially set to a multiple of the size of the root bounding box. It is up to the user to change these values.

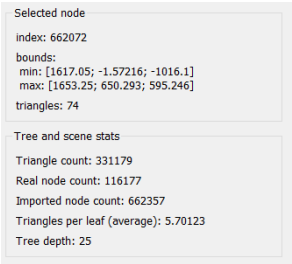


Figure 6.5: Tree and node statistics

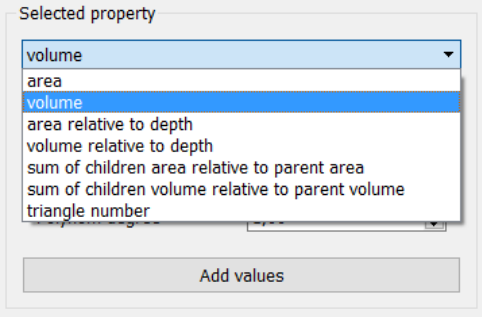


Figure 6.6: The combobox with tree properties

6.3 Scene and tree statistics

This widget shows another important information about the displayed tree. When none of the nodes is selected, the general information is displayed, such as the tree depth, the number of the nodes or the triangles in the scene. See this widget in the figure 6.5. One of the rows is called `Imported node count`. It displays the size of the imported node buffer. The `Real node count` is the real number of nodes, which are traversed by the breadth-first search.

6.4 Selected tree property

The user can switch between the different scalar sets, that define some of the tree properties. Initially, the precomputed properties can be selected in the combobox displayed in the figure 6.6. The value range can be modified using two sliders located below. The visual change caused by moving the top slider can be seen in the figure 6.9.

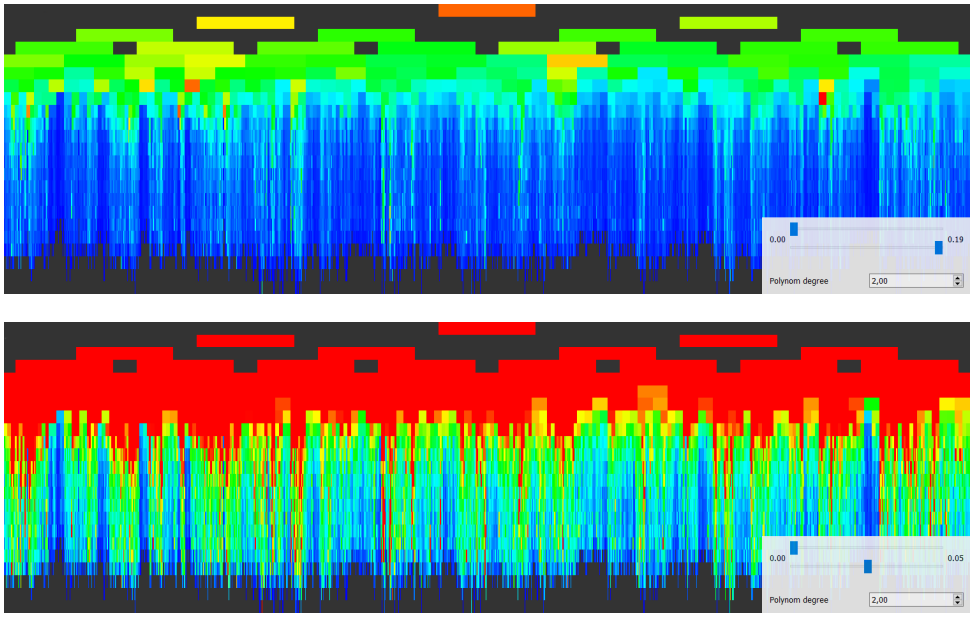


Figure 6.7: Tree view with different max threshold value

The transfer function can be modified by spin box that defines the exponent of the transfer function. Initially, the value is set to one. The lower limit of this value is zero. The example of differently defined transfer functions can be seen in the figure 5.5. The currently defined transfer function is visualized on the right side of the tree view.

6.5 Blending mode

The widget marked with the number five is used to define the blending function. The difference between selected functions appears at lower depth, where multiple nodes cover the same pixel. The individual options can be switched by a radio buttons. Three functions are available. The maximum, the minimum and the average value. The differences see at figure 6.8.

6.6 Tree appearance

To revitalize the appearance of the tree, the user can choose one of the four display options. The options are defined by the selected geometry shader.

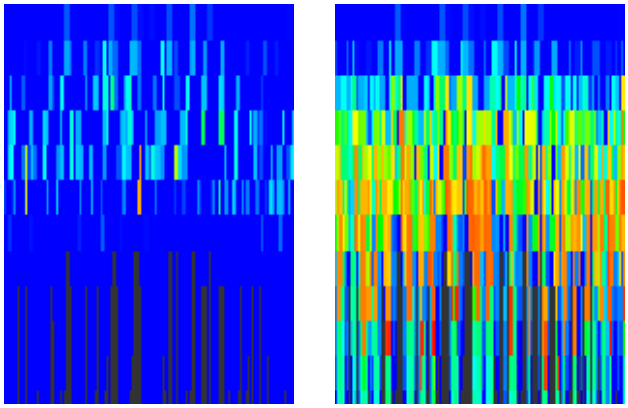
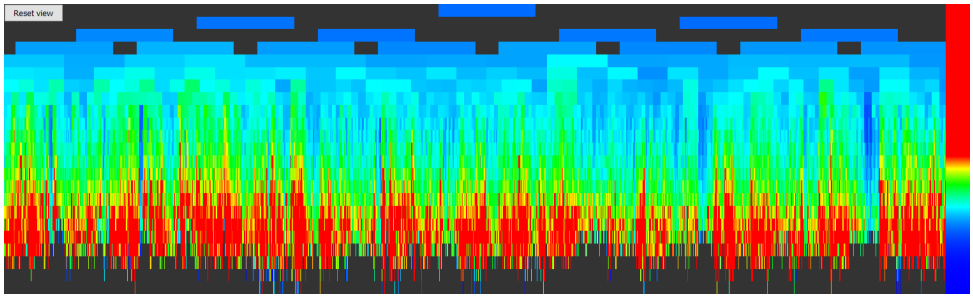
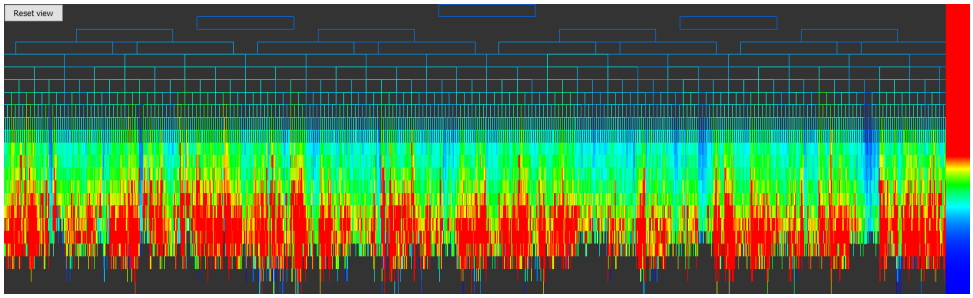


Figure 6.8: Tree detail with blending function set to minimum (left) and maximum (middle)

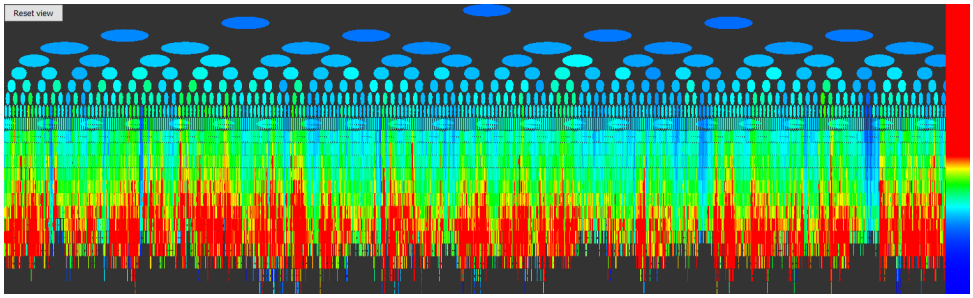
The possible tree designs are following. The individual nodes can be displayed as filled rectangles, wired rectangles, filled ellipses or wired ellipses. The chosen option determines the clarity of the visualization vs. the response time of interaction. The results of measurement can be find in section 7.



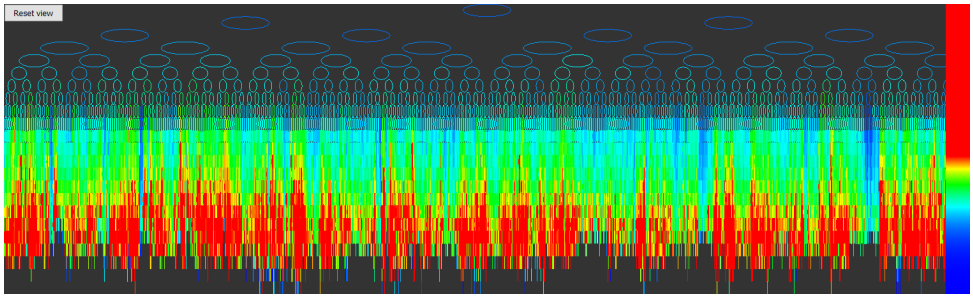
(a) : Filled rectangles



(b) : Wired rectangles



(c) : Filled ellipses



(d) : Wired ellipses

Figure 6.9: Different tree appearance styles

Chapter 7

Testing

This chapter presents the application performance tests. The resolution of the application was in all cases 1920x1030 pixels. The timers from the Qt library with the milliseconds precision were used. The testing PC parameters follow.

- CPU: Intel Core i7-4710HQ, 2.50 GHz
- GPU: GeForce GTX 860M
- Memory: 16,0 GB
- OS: Microsoft Windows 10 Home, 64-bit
- Compiler: Microsoft (R) C/C++ Compiler Version 19.00

The application was tested with six scenes. Its overview, i.e. its names, number of triangles and number of tree nodes can be seen at table 7.1.

The table 7.2 shows the time between opening the scene and loading of all resources. It is also displayed in graph in the figure 7.1.

We also compared the average response times in the table 7.3, which shows the average time between the user interaction with the application and its response. The measuring starts when the user selects a node from the view and ends when the application shows the information about it.

Finally, the rendering times of both of the widgets are compared in the table 7.4. The values represent the average rendering time per frame. As we can see, the tree view of the scene PowerPlant ceases to be smooth (if

scene name	triangles	BVH nodes
FairyForest	174,117	60,277
Conference	331,179	116,177
Buddha	1,087,474	383,159
HairBall	2,880,000	1,023,615
SanMiguel	7,880,512	2,769,557
PowerPlant	12,759,246	4,089,385

Table 7.1: Testing scenes

scene name	file loading	generating nodes	creating scalar sets	total time
FairyForest	18	16	30	135
Conference	36	35	56	237
Buddha	116	105	238	705
HairBall	302	302	638	1,842
SanMiguel	801	767	1,979	5,286
PowerPlant	1,322	1,191	3,140	8,305

Table 7.2: Initial loading times (ms)

we assume the application to be smooth with 25 frames per second). The comparison between rendering times of different display modes is displayed in the table 7.5.

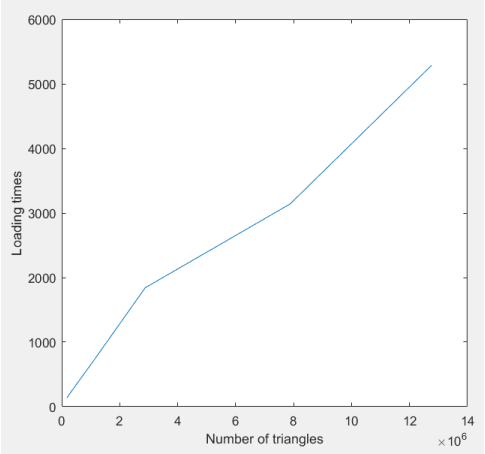


Figure 7.1: Initial loading times (ms)

scene name	pick triangle	pick node
FairyForest	1	8
Conference	9	12
Buddha	41	30
HairBall	1,178	133
SanMiguel	551	498
PowerPlant	2,723	518

Table 7.3: Average interaction response times (ms)

7.1 Discussion

Based on the performance testing, the application renders smoothly the scenes with a millions of triangles. It is also able to interact with the user in a real-time. The tree rendering times differs with the chosen display mode. The fastest rendering mode is the one with the filled rectangles. The worst rendering times obtained the filled ellipses. The ellipses can clarify the visualized information, but the rendering times are quite bad with larger scenes. It is up to the user to select the compromise.

scene name	tree view (filled rectangles)	scene view
FairyForest	2	0
Conference	2	1
Buddha	9	2
HairBall	17	8
SanMiguel	41	19
PowerPlant	62	31

Table 7.4: Average rendering times (ms)

scene name	filled rectangle	wired rectangle	filled ellipse	wired ellipse
FairyForest	1	1	7	1
Conference	4	4	14	3
Buddha	8	10	45	12
HairBall	17	21	117	28
SanMiguel	40	51	308	68
PowerPlant	61	76	455	102

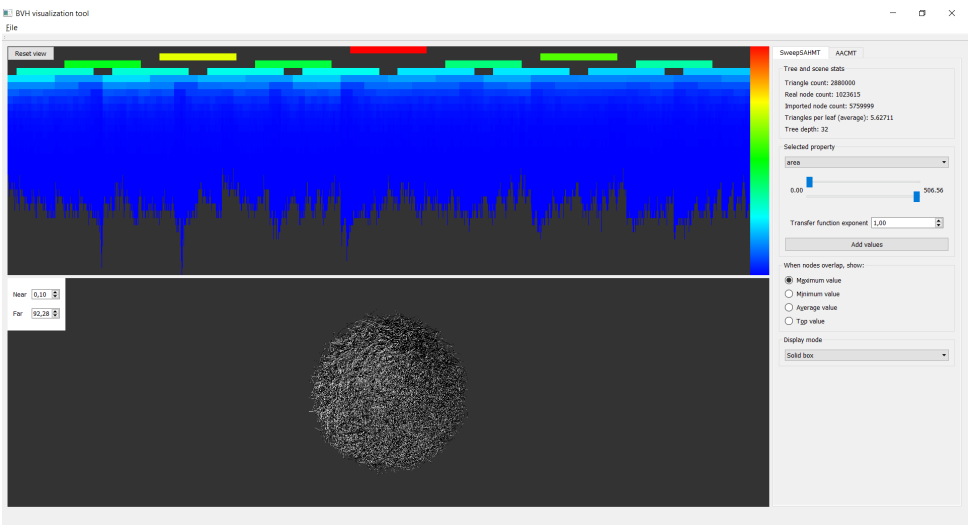
Table 7.5: Average tree rendering times with different display modes (ms)

7.2 The comparison between different BVHs

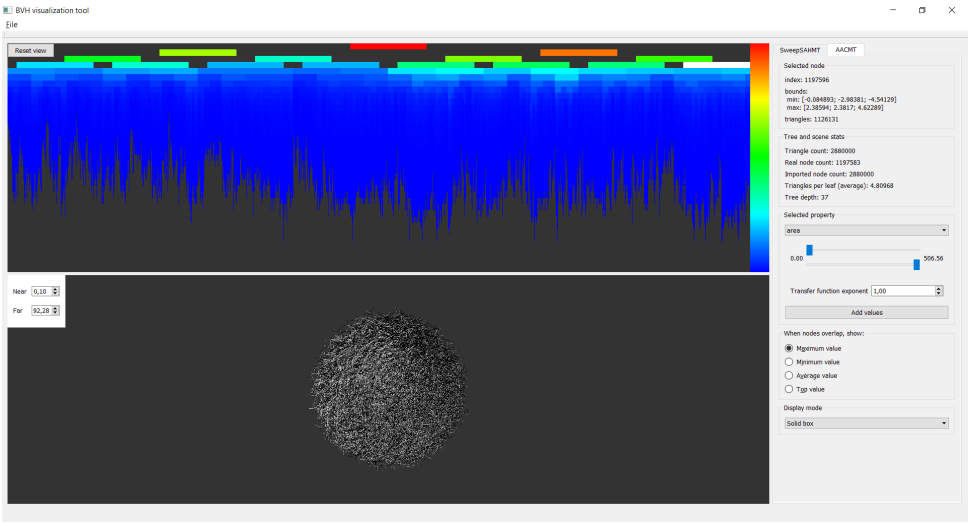
This section describes the visualization of two BVHs generated by a different builders. Let us discuss the brief comparison of the visualizations that can be seen in the figure 7.2.

The first thing we can see is the varying depth of the trees. The SweepSAH tree has lower depth. It has also the lower number of the BVH nodes. Therefore, it has more triangles per leaf on average then the tree generated by AAC. When we display the bounding volumes of the interior nodes, we can see, that this value decreases with increasing depth faster in the SweepSAH generated tree then in the AAC tree. This can be observed also from the value called `sum of children volumes relative to the parent volume` that indicates the children's overlapping. This value can be seen in the figure

??. The tree built by AAC is not coloured as evenly as the SweepSAH tree. That means, that the children overlap more in this case. As a consequence, the SweepSAH tree is probably faster to traverse.

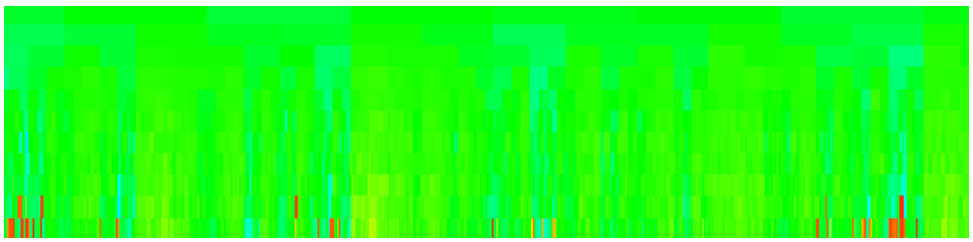


(a) : SweepSAH builder

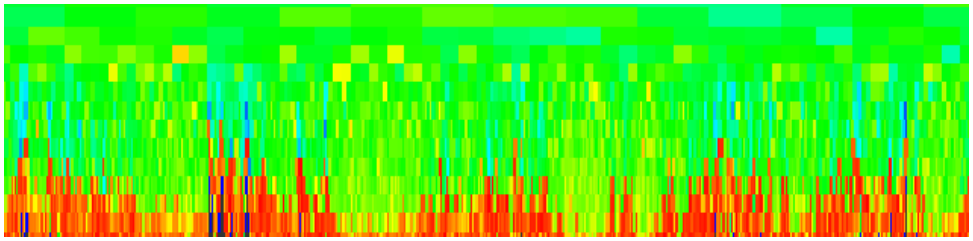


(b) : AAC builder

Figure 7.2: Trees built by a different builders



(a) : SweepSAH builder



(b) : AAC builder

Figure 7.3: The sum of the children volume relative to parent volume

Conclusion

I compared most commonly used 2D hierarchy visualisations with the focus on visualizing the bounding volume hierarchies. Few of the methods were found suitable. The first of them is the organization chart, which was found pretty useful for our application, thanks to the clarity and usability for the BVH structure. The second one, called the icicle plot, is also suitable, because it is displaying only the nodes, not the edges. The combination of these two methods is used in our application.

Next, I made the application design with the respect to the goals of the thesis. The application window is separated into three individual parts. The tree view, the scene view and the control panel containing the GUI. The main features of the application are the visualization of multiple BVH for one scene, the import of the user defined scalar values and the possibility to modify the transfer function.

I have implemented the designed application and dealt with some implementation issues. One of them was the representation of the rendered data. With the decreasing width of a node with growing depth, the nodes at lower levels were so thin thus are not displayed. This is solved using the conservative rasterization. Next issue was for example the displaying of the different information at these places depending on the selected blending mode. The values are rendered to a texture at first, therefore the rendering is made by multiple phases.

I have tested the application with six scenes and compared the application

performance. It works smoothly with scenes about millions of triangles. It can be used to optimize the constructed bounding volume hierarchies. The user immediately sees the balancing of the tree. It is possible to determine the tree quality thanks to the visualized colours, that represents the tree features.

Bibliography

[AGGW15] Jefferson Amstutz, Christiaan Gribble, Johannes Günther, and Ingo Wald, *An evaluation of multi-hit ray traversal in a BVH using existing first-hit/any-hit kernels*, Journal of Computer Graphics Techniques (JCGT) **4** (2015), no. 4, 72–88.

[BN01] T. Barlow and P. Neville, *A comparison of 2-d visualizations of hierarchies*, Information Visualization, 2001. INFOVIS 2001. IEEE Symposium on, Oct 2001, pp. 131–138.

[Bos] Mike Bostock, *Radial reingold–tilford tree*, `http://bl.ocks.org/mbostock/4063550`, Accessed: 11 May 2016.

[Eri05] Christer Ericson, *Real-time collision detection*, Morgan Kauffman Publishers, 2005.

[fAT14] Szirmay-Kalos L. Áfra A. T., *Stackless multi-bvh traversal for cpu, mic and gpu ray tracing*, 129–140.

[FLF12] Nicolas Feltman, Minjae Lee, and Kayvon Fatahalian, *SRDH: Specializing BVH Construction and Traversal Order Using Representative Shadow Ray Sets*, Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics (Carsten Dachsbacher, Jacob Munkberg, and Jacopo Pantaleoni, eds.), The Eurographics Association, 2012.

[GFE+12] Christiaan Gribble, Jeremy Fisher, Daniel Eby, Ed Quigley, and Gideon Ludwig, *Ray tracing visualization toolkit*, Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (New York, NY, USA), I3D '12, ACM, 2012, pp. 71–78.

[Hav15] Vlastimil Havran, *Ray shooting and its applications*, University Lecture, 2015.

[HDW+11] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek, *Efficient stack-less bvh traversal for ray tracing*, Proceedings of the 27th Spring Conference on Computer Graphics, SCCG '11, 2011, pp. 7–12.

[HMM00] I. Herman, G. Melancon, and M. S. Marshall, *Graph visualization and navigation in information visualization: A survey*, IEEE Transactions on Visualization and Computer Graphics **6** (2000), no. 1, 24–43.

[JH] Lennart Ohlsson Jon Hasselgren, Tomas Akenine-Möller, *Gpu gems 2, chapter 42. conservative rasterization*, `https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter42.html`, Accessed: 21 March 2016.

[MB90] J. David MacDonald and Kellogg S. Booth, *Heuristics for ray tracing using space subdivision*, The Visual Computer **6** (1990), no. 3, 153–166.

[Nvi09] *Nvidia® optix™ ray tracing engine unveiled at siggraph 2009 – and a different future for games ray tracing*, `https://thepriorart.wordpress.com/2009/08/05/nvidia-optix-ray-tracing-engine-unveiled-at-siggraph-200/`, August 2009, Accessed: 22 January 2016.

[Pho75] Bui Tuong Phong, *Illumination for computer generated pictures*, Commun. ACM **18** (1975), no. 6, 311–317.

[RMP15] R. MARCUS R. MARCUS and V. PROFILE, *Real-time ray-tracing part 2*, http://robbinmarcus.blogspot.cz/2015/10/real-time-raytracing-part-2.html, 2015, Accessed: 22 January 2016.

[Scra] Scratchapixel, *Introduction to acceleration structures*, http://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure, Accessed: 8 May 2016.

[Scrb] ———, *Rasterization: a practical implementation*, http://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation, Accessed: 19 January 2016.

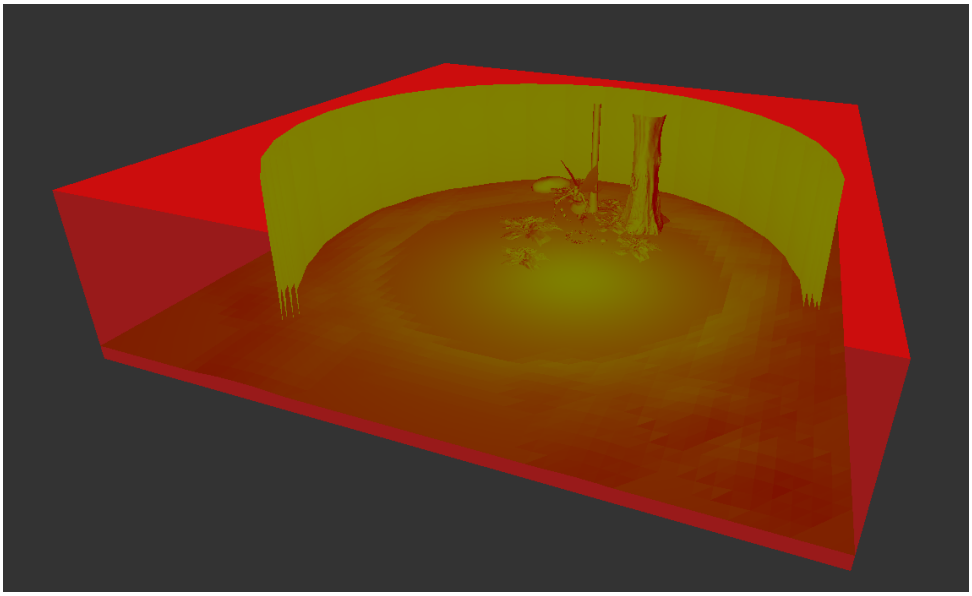
[Sto14] Jon Story, *Don't be conservative with conservative rasterization*, https://developer.nvidia.com/content/dont-be-conservative-conservative-rasterization, November 2014, Accessed: 19 January 2016.

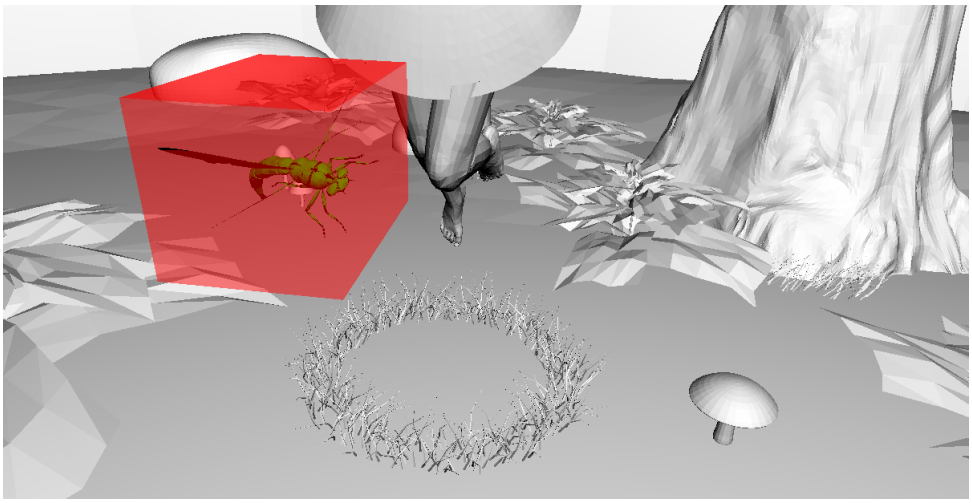
[TM02] Soon Tee Teoh and Kwan-Liu Ma, *Rings: A technique for visualizing large hierarchies*, Revised Papers from the 10th International Symposium on Graph Drawing (London, UK, UK), GD '02, Springer-Verlag, 2002, pp. 268–275.

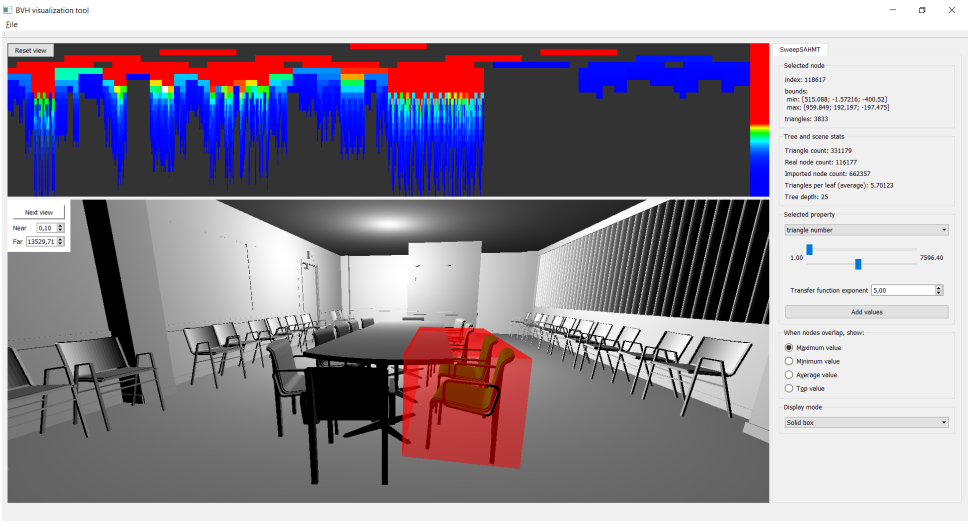
[Wik] *Bounding volume hierarchy*, https://en.wikipedia.org/wiki/Bounding_volume_hierarchy, Accessed: 20 January 2016.

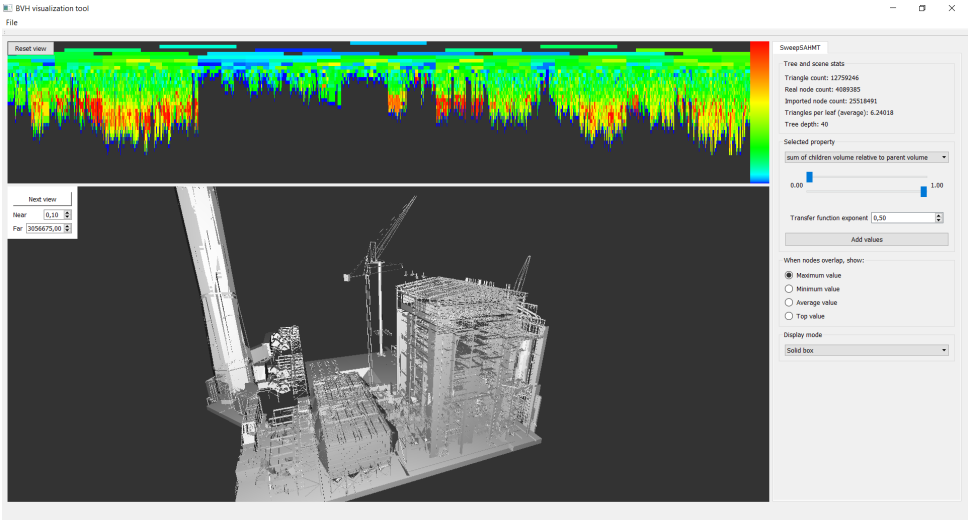
Appendix A

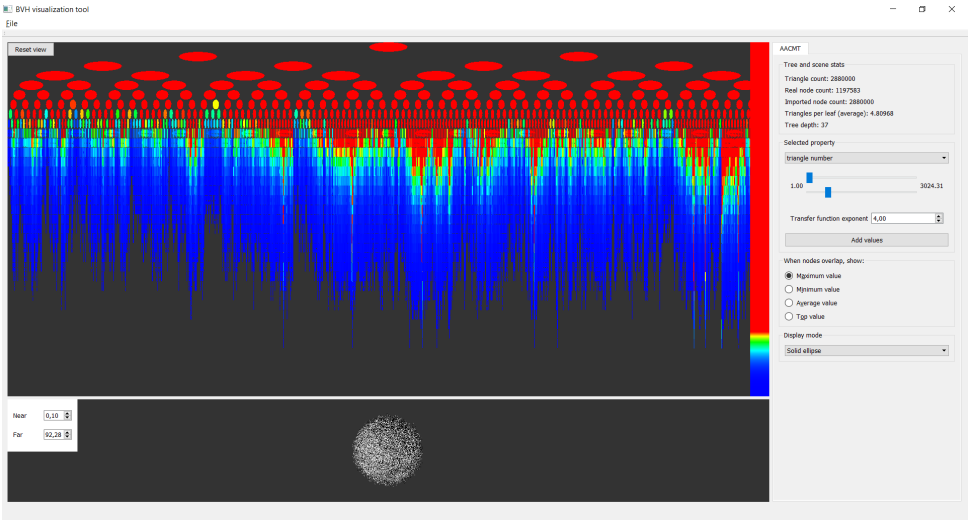
Application screenshots






BVH visualization tool
File
Reset view
Next view
Near 0,10
Far 13529,71
SweepSAHMT
Selected node
index: 118617
bounds:
min: [515,088; -1,57216; -400,52]
max: [959,849; 192,197; -197,475]
triangles: 3833
Tree and scene stats
Triangle count: 331179
Real node count: 116177
Imported node count: 662357
Triangles per leaf (average): 5.70123
Tree depth: 25
Selected property
triangle number
1.00
7596.40
Transfer function exponent 5,00
Add values
When nodes overlap, show:
Maximum value
Minimum value
Average value
Top value
Display mode
Solid box


BVH visualization tool
File
Reset view
Next view
Near 0,10
Far 3656675,00
SweepSAHMT
Tree and scene stats
Triangle count: 12759246
Real node count: 4089385
Imported node count: 25518491
Triangles per leaf (average): 6.24018
Tree depth: 40
Selected property
sum of children volume relative to parent volume
0.00
1.00
Transfer function exponent 0,50
Add values
When nodes overlap, show:
Maximum value
Minimum value
Average value
Top value
Display mode
Solid box


BVH visualization tool
File
Reset view
Near 0,10
Far 92,28
AACMT
Tree and scene stats
Triangle count: 2880000
Real node count: 1197583
Imported node count: 2880000
Triangles per leaf (average): 4.80968
Tree depth: 37
Selected property
triangle number
1.00
3024.31
Transfer function exponent 4,00
Add values
When nodes overlap, show:
Maximum value
Minimum value
Average value
Top value
Display mode
Solid ellipse

Appendix B

Format of input scalar values

The following function is used to load the input scalar values. The file starts with the number of the BVH nodes. The number of scalar sets present in this file follows. Next, each scalar set starts with the length of its name, than comes the name itself and finally the float array of values. The `area` value should be present in every file. It is used as a control, if the values correspond with the current BVH.

```
bool SceneImporter::loadScalars(const string &fileName)
{
  std::ifstream inFile(fileName, ios::binary);

  if (!inFile)
    return false;

  uint32_t actualNodeSize;
  inFile.read(reinterpret_cast<char*>(&actualNodeSize),
      sizeof(uint32_t));
  if (actualNodeSize != bvh->mNodes.size())
    return false;

  uint8_t colorSetSize;
  inFile.read(reinterpret_cast<char*>(&colorSetSize),
     sizeof(uint8_t));
```

```
size_t nameLen;
char *setName;
for (uint8_t i = 0; i < colorSetSize; i++)
{
  setName = NULL;
  inFile.read(reinterpret_cast<char*>(&nameLen),
      sizeof(size_t));
  setName = new char[nameLen];
  inFile.read(setName, nameLen*sizeof(char));

  if (strcmp(setName, "area") == 0)
  {
    vector<float> areas;
    areas.resize(actualNodeSize);
    inFile.read(reinterpret_cast<char*>(areas.data())
        , actualNodeSize * sizeof(float));
    for (uint32_t j = 0; j < actualNodeSize; j++)
    {
      if (areas[j] != bvh->mBoxSizes[j])
        return false;
    }
    areas.clear();
  }
  else
  {
    ScalarSet *s = new ScalarSet();
    s->name.assign(setName, nameLen);
    s->colors.resize(actualNodeSize);
    inFile.read(reinterpret_cast<char*>(s->colors.
        data()), actualNodeSize * sizeof(float));
```

```
            bvh->mScalarSets.push_back(s);
            bvh->normalizeScalarSet(bvh->mScalarSets.size() -
                1);
        }
        delete[] setName;
    }

    inFile.close();
    return true;
}
```

Appendix C

DVD content

- `src` ..... source codes
- `doc`
  - `thesis.pdf` ..... the thesis
  - `latex` ..... LaTeX version of the thesis
- `bin` ..... runnable version of the application
- `data` ..... tested scenes
- `README.txt` ..... description of the DVD content