Paper Title*

*Note: Sub-titles are not captured in Xplore and should not be used

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

Abstract—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.

Index Terms—component, formatting, style, styling, insert

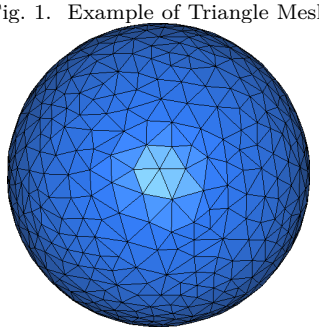
For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [?].

I. Background

A. Triangle mesh

Triange mesh is a kind of polygon mesh. Polygon mesh is also called "Mesh". It is a data structure used in computer graphics to model various irregular objects. The surface of objects in the real world is intuitively composed of curved surfaces. In the computer world, however, only discrete structures can be used to simulate continuous things in reality. So the curved surface in the real world is actually composed of countless small polygon patches in the computer. For example, the model in the Fig.1.

Fig. 1. Example of Triangle Mesh



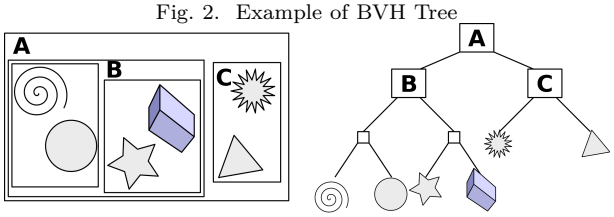
In fact, a large number of small triangles are used inside the computer to form such a shape. Such a collection of facets is called Mesh. Mesh can be composed of triangles or other plane shapes such as quadrilaterals, pentagons, etc.; Because of all the plane polygons can also be subdivided into triangles, it's also general to use a triangle mesh composed of triangles to represent the surface of an object.

For the Triangle Mesh, most of kenrels use indices and vertex to set the data of Triangle, because for a Triange Mesh, not all the Triangle needs 3 Vertexs, because it can be connected with each ohter, so we need to set the vertex data. And a triangle whose vertices are laid out counter-clockwise has its geometry normal pointing upwards outside the front face, we use indice to contral counter-clockwise or clockwise.

B. 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. Bounding volume hierarchies are used to support several operations on sets of geometric objects efficiently, such as in collision detection and ray tracing [1].

Fig. 2. Example of BVH Tree



Identify applicable funding agency here. If none, delete this.

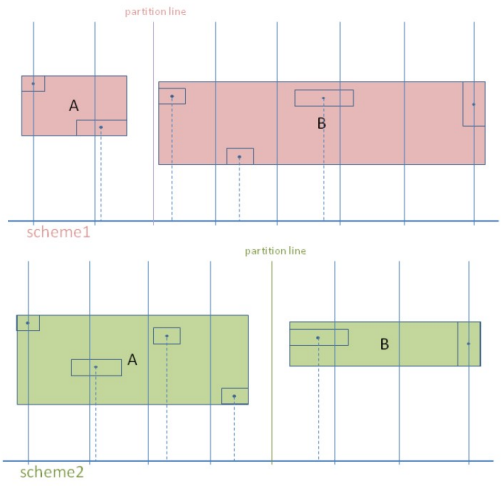
For example, when we implement Snell's law in Python, when calculating multiple geometry or planes, we tested for 2D scene, for complexing 3D scene, we need to use a algorithm called AABB. the method we used is to let the ray operate with each graph to calculate the intersection point, and then determine its true intersection point, so algorithmic complexity is n. And we use a tree structure to manage graphics, which can reduce the complexity to $log(n)$.

Using BVH requires two stages of work to be considered: Build and Traversal. The construction work considers how to construct a binary tree that can effectively describe the current scene information. The key to this is how to divide assumed all objects scattered randomly in the scene Partition, that is, decide which objects should be divided into the left subtree and which objects should be divided into the right subtree.

The best way is called Surface Area Heuristic, also SAH, the SAH algorithm calculates the mean value of the average complexity based on the size of the surface area and the direction of the rays, thereby dividing the BVH tree into the structure with the lowest mean value.

First select an axis with the largest divergence in the graphic distribution, and then spilt it equally along the axis in space, Fig.3. shows two different scheme spilting, and traversal all of scheme to find the best one.

Fig. 3. Example of SAH Spilt

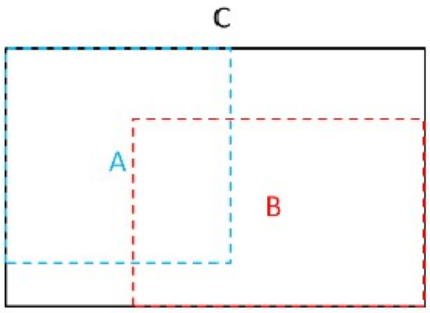


SAH does not completely "non-overlap" or make the distribution after partitioning "uniform", like Fig.4. but each time it makes a partition, it chooses the best plan under the current situation. So it is called a Heuristic algorithm.

C. About the tools

a) Python: Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. [2]

Fig. 4. Example of Spilt



Compared to C++, Python has a simpler syntax,and Python also has many extension packages. such as Numpy,it can be used to calculate vectors directly, Sympy can also be used to solve equations with symbol. By importing those packages, we don't need to overload the operator for vectors calculating, and we don't need to solve some complex equations by ourselves.

In addition, Python also supports interactive shells, such as ipython, which is very friendly to beginners in programming, can speed up our language learning speed, and greatly improve our debugging capabilities.

b) C++: Although Python is very convenient for mathematical operations and friendly enough for beginners, but Python has cross-platform packaging problems and the performance is not good enough. Therefore, projects with huge calculations such as ray tracing generally use better performance and Better use of programming languages with different data structures, such as C++.

c) Embree: Intel® Embree is a collection of high-performance ray tracing kernels, developed at Intel. The target users of Intel® Embree are graphics application engineers who want to improve the performance of their photo-realistic rendering application by leveraging Embree's performance-optimized ray tracing kernels. The single-ray traversal kernels of Intel® Embree provide high performance for incoherent workloads and are very easy to integrate into existing rendering applications. Using the stream kernels, even higher performance for incoherent rays is possible, but integration might require significant code changes to the application to use the stream paradigm. In general for coherent workloads, the stream mode with coherent flag set gives the best performance. Intel® Embree also supports dynamic scenes by implementing high-performance two-level spatial index structure construction algorithms. [3]

Embree is compiling with C++, and we can use Embree API to realise Raytracing.

II. Simulation Design

A. The use of Embree

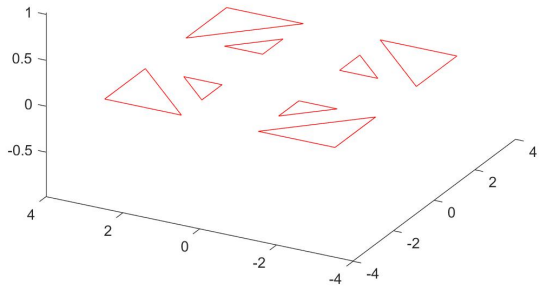
a) Compiling Embree: We compiled Embree on Ubuntu, and installed the release version of Embree on the Windows platform. Our embree-based program was developed on Windows platform. The official recommendation is that using Cmake to compile to generate project files on different platforms. For our Windows Platform, we use Visual Studio as IDE to compile our project.

b) Embree API: For the simplest case, we should at lest have a Device object, Scene object and Geometry object.

- Device object: it allows different components of the application to use the Embree API without interfering with each other.
- Scene object: it's a container for a set of geometries, and contains a spatial acceleration structure which can be used to perform different types of rayqueries.
- Geometry object: it depends on geometry type, like triangle mesh.

c) Our Demo: We refer to the official tutorial and the official API to achieve a hit of ray and scene. The scene contains 8 triangule mesh, and we have 2 rays.

Fig. 5. Triangle location



We are not using BVH Tree , so the complexity of Intersect's algorithm is $o(n)$. At the same time, we calculated the running time of the program, although there are millisecond tolerance, the accuracy is sufficient.

III. Result

A. Embree Demo

a) Our Demo: We just print the result in Command, there is no image output.

Single ray structure are define in this structure, it's also in Embree API.

```
#include <embree3/rtcore_ray.h>
struct RTC_ALIGN(16) RTCRay
{
    float org_x;
    float org_y;
    float org_z;
    float tnear;
    float dir_x;
    float dir_y;
    float dir_z;
    float time;
    float tfar;
    unsigned int mask;
    unsigned int id;
    unsigned int flags;
};
```

the ray's org is the coordinate of ray origin. and geometry 0 is the gemoetry ID of our gemoetry object, primitive is ID of our triangle, base on our indices buffer, tfar is end of ray segment.

From Fig.6 we can see two rays hit 2 Trianges, and the program run time is 17ms.

Fig. 6. Demo Result

```
the ray's org: 2.500000, 2.500000, -1.000000: Found intersection on geometry 0, primitive 1 at tfar=1.000000
the ray's org: 1.000000, 1.000000, -1.000000: Found intersection on geometry 0, primitive 0 at tfar=1.000000
the runtime is: 17 ms
```

b) BVH Result: Embree's Tutorial give us a Demo to show the performance of BVH with SAH.

The result is:

Fig. 7. BVH Result

```
Low quality BVH build:
iteration 0: building BVH over 2300000 primitives, 285.815ms, 8.04716 Mprims/s, sah = 363.26 [DONE]
iteration 1: building BVH over 2300000 primitives, 190.058ms, 12.1016 Mprims/s, sah = 363.26 [DONE]
iteration 2: building BVH over 2300000 primitives, 193.105ms, 11.9106 Mprims/s, sah = 363.26 [DONE]
iteration 3: building BVH over 2300000 primitives, 201.019ms, 11.4417 Mprims/s, sah = 363.26 [DONE]
iteration 4: building BVH over 2300000 primitives, 193.093ms, 11.9113 Mprims/s, sah = 363.26 [DONE]
iteration 5: building BVH over 2300000 primitives, 184.894ms, 12.4395 Mprims/s, sah = 363.26 [DONE]
iteration 6: building BVH over 2300000 primitives, 193.961ms, 11.858 Mprims/s, sah = 363.26 [DONE]
iteration 7: building BVH over 2300000 primitives, 187.12ms, 12.2916 Mprims/s, sah = 363.26 [DONE]
iteration 8: building BVH over 2300000 primitives, 195.339ms, 11.7744 Mprims/s, sah = 363.26 [DONE]
iteration 9: building BVH over 2300000 primitives, 186.65ms, 12.3225 Mprims/s, sah = 363.26 [DONE]
Normal quality BVH build:
iteration 0: building BVH over 2300000 primitives, 782.81ms, 2.93813 Mprims/s, sah = 340.849 [DONE]
iteration 1: building BVH over 2300000 primitives, 1046.04ms, 2.19878 Mprims/s, sah = 340.849 [DONE]
iteration 2: building BVH over 2300000 primitives, 917.511ms, 2.50678 Mprims/s, sah = 340.849 [DONE]
iteration 3: building BVH over 2300000 primitives, 1310ms, 1.75572 Mprims/s, sah = 340.849 [DONE]
iteration 4: building BVH over 2300000 primitives, 1441.77ms, 1.59527 Mprims/s, sah = 340.849 [DONE]
iteration 5: building BVH over 2300000 primitives, 1270.22ms, 1.81071 Mprims/s, sah = 340.849 [DONE]
iteration 6: building BVH over 2300000 primitives, 1225.52ms, 1.87675 Mprims/s, sah = 340.849 [DONE]
iteration 7: building BVH over 2300000 primitives, 1187.42ms, 1.93697 Mprims/s, sah = 340.849 [DONE]
iteration 8: building BVH over 2300000 primitives, 1157.94ms, 1.98628 Mprims/s, sah = 340.849 [DONE]
iteration 9: building BVH over 2300000 primitives, 1140.54ms, 2.01658 Mprims/s, sah = 340.849 [DONE]
High quality BVH build:
iteration 0: building BVH over 2300000 primitives, 1934.26ms, 1.18908 Mprims/s, sah = 339.751 [DONE]
iteration 1: building BVH over 2300000 primitives, 1932.03ms, 1.19045 Mprims/s, sah = 339.751 [DONE]
iteration 2: building BVH over 2300000 primitives, 1955.2ms, 1.17635 Mprims/s, sah = 339.751 [DONE]
iteration 3: building BVH over 2300000 primitives, 1924.15ms, 1.19533 Mprims/s, sah = 339.751 [DONE]
iteration 4: building BVH over 2300000 primitives, 1914.13ms, 1.20159 Mprims/s, sah = 339.751 [DONE]
iteration 5: building BVH over 2300000 primitives, 1909.3ms, 1.20463 Mprims/s, sah = 339.751 [DONE]
iteration 6: building BVH over 2300000 primitives, 1884.56ms, 1.22045 Mprims/s, sah = 339.751 [DONE]
iteration 7: building BVH over 2300000 primitives, 1940.36ms, 1.18048 Mprims/s, sah = 339.751 [DONE]
iteration 8: building BVH over 2300000 primitives, 1902.84ms, 1.20872 Mprims/s, sah = 339.751 [DONE]
iteration 9: building BVH over 2300000 primitives, 1893.38ms, 1.21476 Mprims/s, sah = 339.751 [DONE]
```

from Fig.7 we can see, it has over 2300000 primiteves, and the run time is base on the quality is between 200ms and 2000ms, in our demo, we only have 8 simple triangle primitives, and the run time is 17ms. so we can find that BVH is very fast.

References

[1] Wiki. Bounding volume hierarchy. [Online]. Available: https://en.wikipedia.org/wiki/

[2] D. Kuhlman, A Python Book: Beginning Python, Advanced Python, and Python Exercises, 2012.

[3] Intel. Embree overview. [Online]. Available: https://github.com/embree/embree