VR MESH VIEW

A PROJECT REPORT

SOSE 2020

By

Erdem Ünal, Boyang Liu, Christian Gärber, Vincent Adam, Ye Tao, Stefanie Krell

Under the guidance of

David Groß

I. Introduction

Our project is to create a mesh view and manipulate mesh in VR. Our tasks include 8 work packages and our group consists of 5 members. In the following, we will introduce the specific work packages and the corresponding members who are responsible for it. Then each member of our group will introduce what they have implemented in detail. Finally, we will give an overview on how to use the programm.

II. Division of Work

	Work Packages	Tasks	Assignee
1	Mesh data structure	a. Extract triangle data from simple_mesh.h b. Build your own mesh data structure	Christian Vincent
2	Acceleration data structure	a. Building of an acceleration structure b. Ray triangle intersection	Ye,Boyang Erdem
3	Measuring	a. Surface area and volume b. Shortest distance to mesh	Stefanie
4	Animation	a. Animate meshes along predefined paths b. Draw 3D trajectory in VR as animation path	Boyang
5	VR view integration	Switch your project to use VR view	Christian Vincent
6	Mesh editing	a. Tessellation b. Manipulation c. Smoothing d. Vertex deletion e. Face deletion f. Build simple mesh from HE	Ye Erdem Stefanie Erdem Ye, Stefanie Ye, Stefanie
7	VR environment	Add a simple environment in the VR	Christian Vincent
8	CSG	Constructive Solid Geometry	Christian Vincent

III. Report of each member / each work package

W1 Mesh Data Structure

To perform the operations on the mesh and to create the necessary other structures (like an AABB Tree, see W2) we were tasked to implement a more easily accessible triangulated data structure for the mesh. The chosen data structure should make it possible to reference faces and to retrieve information on surrounding faces for a given one.

With these conditions in mind the “Half Edge” data structure was chosen. Each edge of the original (triangulated) simple_mesh is split into two “half-edges” which internally store their “twin” (the other half of the original edge), the face that they belong to and the next vertex. A face in this data structure stores one adjacent half-edge.
Using this information it is possible to traverse all half-edges that surround a face. With these half-edges all adjacent faces can be found using the twin edges. Since each face, vertex and edge is represented as a struct and stored in an internal list the referencing of faces is made possible.

The class “HE_Mesh”, which represents a Half Edge Mesh was later extended by others. Six methods were added when they were needed for new work packages:

- changeVertexPos
- isClosed
- AddBoundary
- deleteFace
- deleteVector
- showAllInfo

To construct this Half Edge mesh a method was written in vr_mesh_view.cxx. This method takes in a simple_mesh and constructs a new HE_Mesh on the heap and returns a pointer to the created object. Since a triangulated mesh was needed for the other tasks the triangulation method of simple_mesh is used.

(Code in: halfedgemesh.h / halfedgemesh.cpp & vr_mesh_view.cxx method “generate_from_simple_mesh”)

W2 Acceleration Data Structure

Acceleration data structure is very helpful for ray intersection and measurement. It can as soon as possible decide which triangles are likely to be intersected with the ray and reject a large mount of triangles. In this project, the acceleration data structure is a bounding volume hierarchy. Specifically, it is a binary tree structure. And each node in the tree stores an axis aligned bounding box of some triangles. The root node stores the biggest bounding box which holds all the triangles of the mesh. And the leaf node stores the smallest bounding box which only contains one triangle.

The process of building it is that firstly, we get all the faces from the half edge data structure and convert it into a triangle type and then store them in the primitive list. Secondly, we build this acceleration data structure from the root recursively. When we split the primitive list into two distinct lists and store them into two child nodes, we also sort the primitive list along the largest bounding box extent. Finally, we stop building the tree until each node only contains a single triangle. This acceleration data structure can not only be used for triangles, but also for any primitive. WeI tested for small meshes and it works well. But in order to verify the correctness of this function, we also implemented a function which can render all the bounding boxes in the leaf node.

Code:

1. aabb_tree.h and aabb_tree.cpp build axis aligned bounding box from half edge data structure;
2. code of rendering bounding box in draw() with the show_bounding_box variable.

Ray-Triangle Intersection

The file *ray_intersection.h* is composed of a bunch of ray intersection methods that we used in this project. The ray of the VR controller is structured as the origin (the eye of the ray) and the direction (the direction vector of the ray). All the ray intersection methods are based on ray-triangle intersections. We are able to get the intersected point, face, vertex, mesh, bounding box, AaBb Tree and the distance to the intersection point via the intersection methods in ray_intersection.h

With the addition of acceleration data structure, instead of using rayMeshIntersect() function we started to use rayTreeIntersect() as the search operation and calculation time is much more efficient. In the conventional ray-mesh intersection search, it searches the intersection on all the faces of the mesh with a time complexity of *O(N)*. On the other hand, bounding box mapping in the AaBb tree allows us to search the intersection recursively by checking ray-node intersections. Then the node intersection is achieved, the algorithm checks the left and right child nodes until the leaf node is reached. It checks the box of the leaf node and then the ray-triangle intersection from the triangle inside of the box. This process takes *O(logN)* time.

W3 Measuring

The code for this work package can be found in the file mesh_utils.h

Surface Area

Since a triangle mesh is used, the surface area is calculated by summing up the area of each triangle. The Area of one triangle can be calculated by taking the half of the cross product of two vectors of the triangle edges.

Volume

To calculate the volume, the signed volume of each tetrahedron (consists of a triangle and zero point) is summed up. The signed volume can be calculated by using the triple product and dividing by 6.

Calculation of the shortest Distance

Two calculate the shortest distance to the mesh from any given point p two implementations are available, one which is an exact calculation and the second approximates by using the Acceleration data structure.
For the exact calculation, for every triangle in the mesh the distance is calculated. To calculate the shortest distance to a triangle, first the orthogonal projection p' of the given point p onto the surface of the triangle is computed. If p' is inside the triangle, p' is the closest point to p and the shortest distance to this triangle can be calculated.If p' does not lay inside the triangle, p' needs to be projected onto a triangle edge to create p'''. Then p''' is the closest point on the triangle and the shortest distance can be calculated.
The faster implementation makes use of the acceleration data structure(AD). Therefore the AD is iterated through from top to bottom. For each node the distance from p to the bounding box center is calculated. If the distance to the left node is shorter, the left node is used otherwise the right. If the algorithm reaches the bottom layer, the distance to the triangle is calculated as before.

The closest point to the right controller can be visualized by clicking the left stick button of the right controller. In the gui the calculation of the closest point can be changed. As a visualisation, a black sphere is rendered. The closest point with the AD is not as accurate as if all triangles are examined, but it is faster especially for larger meshes.

W4 Animation

The code for this work package can be found in the file vr_mesh_view.cxx.

Draw 3D trajectory in VR as animation path

The general idea is that the user uses the right controller to move the mesh in order to self define an animation path. The path is set according to the movement distance and rotation angle of the controller. Every moment the mesh positions are stored. The 3D trajectory is drawed by calling the drawpath() function. The 3D trajectory starts from the mesh centroid position. If the user changes the mode to editing mode, the animation path will be cleared and the user can redefine the animation.

Animate meshes along defined paths

When the user presses the corresponding button, the mesh returns to the origin position and replays the animation along the path. When the user releases the button, the user can continue to define the path. The speed of the translation of the mesh is determined when the user defines the path.

W5 VR Integration

After implementing the W1 - W4 in a non-VR application the code was transferred to work in VR. This was based on the “vr_test” example provided by the framework. Since all renderers work the same way in VR little to no further code changes were necessary. Due to restrictions in VR some features could not be ported directly into the VR view. These include the selection of the *.obj file, the scaling factor of the mesh, options to de-/activate the rendering of vertices and edges and the settings for the CSG (see W8).

W6 Mesh Editing

Smoothing

Here the Laplacian Smoothing was implemented. The new position of every vertex is calculated by calculating the average of the neighbor nodes. Here it is possible to apply the smoothing to the whole mesh(every vertex) or just some vertices (User can select faces and the vertices of the faces are smoothed). The User selects the faces by holding the controller beam onto the face and clicking the button. the the face is shown in black. Here a new simple_mesh called smoothingMesh is created and new faces are added when the user selects more faces. To apply the smoothing function the User clicks the button.
The code can be found in the vr_mesh_view.cxx especially in the functions: add_face_to_smoothingMesh(HE_Face* f), applySmoothing(), applySmoothingPoints() and under the specific button operations.

Tessellation

Tessellation will be triggered when the user in the VR sends a ray to the mesh and clicks a button of the controller. If the ray intersects with the mesh, the intersected point will split the intersected face into 3 faces. Firstly, we calculate the intersection point position and transform it to the local system. Then we delete this intersected face in the half edge data structure. And we add three new faces to the half edge data structure. Finally, we rebuild a new simple mesh and render it. We also write the new mesh into an object file and the user can reload it.

Code:

1. tessellation() in vr_mesh_view.cxx is the main part of this function;
2. deleteFace() in halfedgemesh.h / halfedgemesh.cpp is used to delete a face that is intersected with the ray;
3. build_simple_mesh_from_HE() in vr_mesh_view.cxx is used to rebuild an edited mesh.

Vertex Manipulation and Deletion

Our project allows us to create a new vertex (tessellation), delete an existing vertex and manipulate the position of a vertex by simply dragging it on the mesh. The deletion of the vertex is processed when the user picks the vertex with the VR controller by pressing the corresponding vertex deletion key. This triggers the *vertex_deletion()* method in *vr_mesh_view.cxx*. If the vertex intersection occurs during that process, the neighboring faces of the vertex is deleted first and then the vertex is deleted using the *deleteFace()* and *deleteVertex()* methods in halfedgemesh.cpp. Then new triangle faces are created by creating new half edges using the neighboring vertices.

In order to move the chosen vertex, the user has to press the corresponding vector manipulation button and simply pick the vertex by pointing the ray of the VR controller. This triggers the *vertex_manipulate()* function in *vr_mesh_view.cxx*. If the vertex intersection is achieved, by simply toggling the button without releasing it, the user can move the chosen vertex and can observe the dragging instantaneously using the POSE handle event of the VR controllers. *vertex_manipulate()* function simply changes the position of the *intersectedVertex* both in HE_mesh and simple mesh data structure. In order to finish the dragging and assigning the new position of the vertex, simply releasing the button will trigger rebuilding of the simple mesh and the edges will be properly drawn.

W7 VR Environment

For the environment there are four options:

- Nothing
- Room with table
- Empty Room
- Floor and colored boxes in the surrounding

These environments are based on the environment in vr_test. In case of the room with a table the loaded mesh will be offset so it is above the table.

W8 Constructive Solid Geometry (CSG)

The Constructive Solid Geometry part of the work has been simplified in order to reduce implementation time. It currently only works on the mesh that is currently loaded and a sphere that the user can place. Further restrictions are that the center of the sphere needs to be outside of the loaded mesh and that the performed operation must not split the mesh into >= two sub-meshes. If the mesh is split the currently implemented code might crash the program or run into an endless loop (which is due to the way that the two meshes are combined).

IcoSphere

An IcoSphere is used as the second mesh for our implementation of CSG. An IcoSphere was used because it creates an even distribution of vertices on the sphere.
Before the IcoSphere can be created, the user has to select the number of subdivisions for the IcoSphere in the GUI window within a range from 1 to 5. The default value is 3. This number determines the number of times, each of the IcoSphere's faces are split into four new faces to achieve a more detailed approximation of a sphere.
After the selection is made, the user can start the construction of the IcoSphere by pressing and holding the assigned button on the controller. The IcoSphere's center point is then set at the current location of the controller. By dragging the controller through 3D space, the radius is determined by the distance of the controller from its initial position. A Sphere is rendered to visualise the current size of the IcoSphere for the user. When the button is released and the IcoSphere is generated from the center point, the radius and the selected number of subdivisions. Immediately after, the CSG calculations are started.
(Code in: icoshpere.h / icosphere.cpp)

CSG Operations

The user has the choice of three operations with our implementation of Constructive Solid Geometry (CSG). These are Union (two meshes are combined), Intersection (the section which is part of both meshes) and Difference (The part of mesh 1 that is not included in mesh 2). The operation has to be selected from a drop-down menu in the GUI window.
After the Sphere has been drawn by the user and converted to an IcoSphere, the calculations for the previously selected CSG Operation is started.
First, for both the Half-Edge Mesh and the Icosphere, the algorithm determines whether Vertices are located within or without the respective other mesh. This is achieved by ray-intersection for the IcoSphere's vertices and simple distance calculations for the mesh's vertices. From this all faces which will remain in the new Mesh are calculated and added to a new simple_mesh. This results in a single mesh that contains the two parts of the input meshes. Those two parts are not connected so a simple routine was implemented to find the boundary edges of each of the mesh parts around the gap. This gap is then closed in two stages.
The first step is joining each of the edges of the Half-Edge Mesh to the closest vertex on the boundary of the IcoSphere's remaining faces. After this is completed, the gaps on the IcoSphere's side are detected and filled by a simple triangulation routine.

Finally, the resulting mesh is returned to the VR view, converted into a new Half-Edge Mesh and displayed. The AABB-Tree and transformation matrices are reset and recalculated. Due to the fact that the CSG works with a Half-Edge mesh the resulting mesh will not retain non-triangular faces, these will consist of triangles after the first CSG operation is performed.
(Code in: simple_csg.h / simple_csg.cpp)

IV. Overview on How to use the program

First a mesh needs to be loaded manually via mouse into the application. Beforehand it is possible to select a scaling for the mesh which is applied when the mesh is loaded.

In the VR environment, you have the right and the left controller which are used for different operations. In order to separate some tasks, two modes between the user can change are available. The animation mode and the mesh editing mode. To switch between the modes, the menu button on the right or on the left controller needs to be clicked. In the VR environment in addition to the loaded mesh you also see a quad with some text on it. There the mode, the buttons to use, and volume/ surface area of the mesh are displayed.

Mesh editing mode:

- Left Controller:
  - Stick up: Vertex Manipulation
  - Stick down: Select Face for Smoothing
  - Stick right: Apply Smoothing
  - Stick left: Tessellation
- Right Controller:
  - Stick up: Vertex Deletion
  - Stick down: CSG Operation
  - Stick right: Recalculate Volume, Surface
  - Stick left: Closest Point / shortest distance to mesh from right controller

Animation mode:

- Left Controller: Translation and Rotation of meshes
  - Stick up: //
  - Stick down: //
  - Stick right: //
  - Stick left: //
  - Touching: Replay animation ,while pressing Right Stick Right. Draw animation(do rotation),while without pressing Right Stick Right.
- Right Controller:
  - Stick up: //
  - Stick down: //
  - Stick right: Press, mesh goes to original position/release, mesh goes to final position
  - Stick left: //
  - Touching:Draw animation path(do translation)