**📚 Resources**

👉 See the MeasurementUtils [docs](https://docs.thatopen.com/Tutorials/Components/Core/MeasurementUtils)

👉 Check the [Line3 docs](https://threejs.org/docs/?q=Line#api/en/math/Line3) from ThreeJS

**🗝 Key Concepts**

**📏 Edge-Based Measurement**

Measuring distances based on selected edges is a technique that allows users to take precise measurements between non-parallel points on a BIM model. This is particularly useful for architectural applications.

**📐 Directional Analysis of Vectors**

Directional vectors and dot products are used to determine which vertices are relevant for measurements based on the orientation relative to a reference edge. This ensures accurate measurement lines are created.

**🔄 Dynamic Edge Highlighting**

Visual feedback, such as highlighting, indicates the active edge being selected for measurement, making the user interaction more intuitive and user-friendly.

**🎯 Raycasting for Point Selection**

Raycasting techniques are employed to detect and select the edge nearest to the user's mouse cursor. The hit point is used to calculate the shortest distance to each edge.

**🧮 Vertex Projection and Calculation**

Projecting the furthest vertices onto the edge plane facilitates the calculation of precise measurement endpoints and provides the correct values for dimensions.

**📐 Geometric Data Manipulation**

Using libraries like Three.js allows for the precise manipulation of geometric data, such as edges and vertices, to create accurate measurements and visual representations.

**🛠 Modular Component Design**

Organizing functionality into a modular component promotes scalability and makes it easier to reset state or add new functions without major modifications.

**🤓 Exercise Development**

The following is the step-by-step guide on how to accomplish the exercise. Use this as complementary information from the video. 📽

**🚧 Scaffolding the BIM App**

If you do not have any app yet, create one using the official engine template. Create an empty folder anywhere on your PC and open it with your IDE of choice. Then, in the terminal, execute the following command:

npm create bim-app@latest

Right after, in the bim-components folder of the new project, create another folder named OpeningMeasurement. Inside, create an index.ts file to start coding the component.

**🧙‍♀️ Crafting the Component**

As everything is already setup, let’s go to bim-components/OpeningMeasurement/index.ts and write the following:

import \* as OBC from "@thatopen/components";

import \* as OBF from "@thatopen/components-front";

import \* as THREE from "three";

export class OpeningMeasurement extends OBC.Component {

static uuid = /\* palce here your own UUID v4 compliant ID \*/ as const;

enabled = true;

// We need to keep track of the world because the preview geometry

// needs to be added to the scene to be displayed, and when the world is

// removed, the preview must be removed too, so it doesnt leak memory or

// causes visual bugs.

private \_world: OBC.World | null = null;

set world(value: OBC.World | null) {

this.\_world = value;

// When the world is set, we add the edge preview to the scene.

// This is the graphical representation of the edge took for the measurement.

if (value) {

value.scene.three.add(this.\_edgePreview);

} else {

// If no world is provided, it means the component is not longer used

// so we remove the preview from the parent so it is properly cleaned up.

this.\_edgePreview.removeFromParent();

}

}

get world() {

return this.\_world;

}

// We need to store the edge because this is the base line for the measurement.

// It will define the two of the four endpoints for the measurement.

private \_edge: THREE.Line3 | null = null;

set edge(value: THREE.Line3 | null) {

this.\_edge = value;

// We make the preview visible if an edge was set.

this.\_edgePreview.visible = !!value;

if (!value) return;

// If we have a new edge, we update the preview geometry with the

// start and end of the line.

const previewGeometry = this.\_edgePreview.geometry;

previewGeometry.setFromPoints([value.start, value.end]);

previewGeometry.attributes.position.needsUpdate = true;

previewGeometry.computeBoundingBox();

previewGeometry.computeBoundingSphere();

// Those two last operations are performed because this is a "dynamic" object

// that updates its vertices and thus we need to tell threejs to recalculate

// the geometry bounds to not have culling problems.

}

get edge() {

return this.\_edge;

}

// We create a preview of the edge to be measured, so the user can see

// what's being measured.

private \_edgePreview = new THREE.Line(

new THREE.BufferGeometry(),

new THREE.LineBasicMaterial({

color: "red",

depthTest: false,

}),

);

constructor(components: OBC.Components) {

super(components);

components.add(OpeningMeasurement.uuid, this);

// We add this component to the components manager, so we can retrieve it later.

}

// This will be a 2D array of points because we are processing edges

// and each edge has two points. Those points will be used to calculate

// the size of the opening.

edges: THREE.Vector3[][] = [];

measure() {

// We cannot perform a measurement without an edge so we return the function.

if (!this.edge) return;

const edge = this.edge;

// We need to know the direction of the edge because the algorithm

// calculates the projection points that are perpendicular to the edge

const edgeDirection = new THREE.Vector3();

edge.delta(edgeDirection);

edgeDirection.normalize();

// We use flat here because we have a 2D array and we need to iterate

// over a 1D array of points, so we get a collection with all

// the vertices from the element face.

const vertices = this.edges.flat();

// We need the farthest point from the start of the edge.

// This point indicates the "depth" of the measurement.

// For example, if it's a door, this will be the door frame depth.

const farthestFromStart = vertices.reduce((prev, curr) => {

const currLine = new THREE.Line3(edge.start, curr);

const currLineDir = new THREE.Vector3();

currLine.delta(currLineDir);

// We only consider points that are in the oposite direction

// of the edge start. See the video for a deeper explanation

// on this.

if (currLineDir.dot(edgeDirection) >= 0) return prev;

const prevLine = new THREE.Line3(edge.start, prev);

// Here we compare the distance and return the farthest one.

if (currLine.distance() > prevLine.distance()) return curr;

return prev;

}, edge.start);

// Same logic as above but getting the farthest point from the end of the edge.

const farthestFromEnd = vertices.reduce((prev, curr) => {

const currLine = new THREE.Line3(edge.end, curr);

const currLineDir = new THREE.Vector3();

currLine.delta(currLineDir);

// We only consider points that are in the oposite direction

// of the edge end.

if (currLineDir.dot(edgeDirection) <= 0) return prev;

const prevLine = new THREE.Line3(edge.end, prev);

if (currLine.distance() > prevLine.distance()) return curr;

return prev;

}, edge.end);

// We get the length measurement component to create the visual representation of the

// measurement.

const lengthMeasurement = this.components.get(OBF.LengthMeasurement);

// If the farthest point from the start is not the same as the start point,

// it means we have a point to create a measurement, so we project the

// farthest point to the edge.

if (!edge.start.equals(farthestFromStart)) {

const projection = new THREE.Vector3();

edge.closestPointToPoint(farthestFromStart, false, projection);

const line = new THREE.Line3(edge.start, projection);

// We check the distance because sometimes the point could be the same

// and then we won't need to create a visual representation of the line.

if (line.distance() > 0) {

lengthMeasurement.createOnPoints(edge.start, projection);

}

}

// Same logic as above but for the end point of the edge.

if (!edge.end.equals(farthestFromEnd)) {

const projection = new THREE.Vector3();

edge.closestPointToPoint(farthestFromEnd, false, projection);

const line = new THREE.Line3(edge.end, projection);

if (line.distance() > 0) {

lengthMeasurement.createOnPoints(edge.end, projection);

}

}

}

reset() {

// We reset the edges and the edge to start a new measurement.

this.edges = [];

this.edge = null;

}

}

**📏 Using the Measurer**

As the component is now ready, go to the main.ts file of the app and import the component:

import { AppManager, OpeningMeasurement } from "./bim-components";

Then, down in the code right before the toolbar declaration, write the following:

// We retrieve the opening measurement component.

// This component is responsible for creating the opening measurements.

const openingMeasurement = components.get(OpeningMeasurement);

// We set the world to the opening measurement component,

// so it can properly display the preview geometry within the scene.

openingMeasurement.world = world;

// We obtain the raycaster associated with the current world.

// The raycaster will be used to detect the objects we are hovering and clicking on.

const raycaster = components.get(OBC.Raycasters).get(world);

// We listen to the mousemove event to perform real-time calculations

// based on the mouse position. This is where the preview is updated.

window.addEventListener("mousemove", () => {

// We cast a ray from the mouse position into the scene, to see if we're

// hovering over any object.

const result = raycaster.castRay();

if (!result) {

// If we don't hit anything, we reset the opening measurement

// because we're not focusing on an element, which avoids leaving

// the preview geometry in an invalid state.

openingMeasurement.reset();

return;

}

// If we hit something, we extract the data of the raycast result.

const { object, instanceId, faceIndex, point: hitPoint } = result;

// We validate if we are actually hitting a FragmentMesh

// to avoid errors when the raycast is hitting other kinds of objects

// as the logic to get the face edges depends on the object to be

// a FragmentMesh

if (

!(

instanceId !== undefined &&

faceIndex !== undefined &&

object instanceof FRAGS.FragmentMesh

)

) {

// If not, we reset the component because is not the correct object.

openingMeasurement.reset();

return;

}

// If all is correct, we get the measurement utility component because

// we need to obtain the face data from the raycast result.

const measurement = components.get(OBC.MeasurementUtils);

const face = measurement.getFace(object, faceIndex, instanceId);

// If we can't obtain the face data, we exit because we can't perform the measurement

if (!face) return;

// If we have face data, we map to get the points of all edges.

// The reason for this is because the `OpeningMeasurement` needs

// all the possible edges to extract the vertices and take the farthest

// from each endpoint.

const edges = face.edges.map((edge) => edge.points);

openingMeasurement.edges = edges;

// We find the closest edge to the mouse position.

// This is the line where we will be measuring the opening.

const closestEdge = edges.reduce((prev, curr) => {

// We unpack the points of the previous edge

const [pv1, pv2] = prev;

// We calculate the distance from the hit point (mouse position) to

// the previous edge using a utility function.

const previousDistance = OBC.MeasurementUtils.distanceFromPointToLine(

hitPoint,

pv1,

pv2,

true,

);

// We unpack the points of the current edge.

const [cv1, cv2] = curr;

// We calculate the distance from the hit point (mouse position) to

// the current edge using a utility function.

const currentDistance = OBC.MeasurementUtils.distanceFromPointToLine(

hitPoint,

cv1,

cv2,

true,

);

// We compare the current distance with the previous distance,

// and we also check if the distance is less than 0.3, so

// the user needs to be close to the edge to actually select it.

if (currentDistance < 0.3 && currentDistance < previousDistance) {

// If the current edge is closer we return it.

return curr;

}

// Otherwise, we keep the previous edge.

return prev;

});

// Finally, we create a new line3 with the selected edge.

openingMeasurement.edge = new THREE.Line3(closestEdge[0], closestEdge[1]);

});

window.addEventListener("click", () => {

// When the user clicks, we perform the final opening measurement.

openingMeasurement.measure();

});

You’re ready to test your new measuring tool!

**⚔ Quest (code: 8Jt7N)**

The quest is your chance to grow. It will let you increase your level to earn different badges and get access to benefits that only people with certain levels can achieve, like getting Bounties 💰 (cash rewards) for contributing to That Open Company's open-source libraries, or special gifts. To complete this lesson’s quest, do the following: 👇

Update the component to disallow the selection of elements while measuring. That way, you improve the user experience.

**🦶 Development Steps**

1. In the OpeningMeasurement class definition, replace the enabled property with accessors (that’s a setter and getter). The reason is because we want to perform a side operation anytime the component gets active. TIP: Remember to create the private enabled property to store the actual value; make it disabled by default.
2. In the enabled setter, get the highlighter component. Then, set the highlighter enabled state to be the opposite of the value being set for the enabled property in the OpeningMeasurement component.
3. In the measure method, make sure to not make the processing if the component is not enabled.
4. In the main file, right after the world is assigned to the openingMeasurement component, set the enabled state as true. That means the component will be available by default and the highlighter will become disabled.
5. [OPTIONAL] Is up to you if you want to create a button in the user interface to enable/disable the component.