Fruit Ninja Using PoseNet Library

Group members:

Vivek

Tusha

Shivam Saini

Saheer Azfar

STEP 1: Breaking the problem down

STEP 2: Picking the TOOLs

STEP 3: Learning the things

STEP 4: Get the pose detection working

STEP 5: Add 3D objects

STEP 6: Map the 2D hands movements to the 3D world

STEP 7: Add collision detection between the hands and 3D objects

STEP 8: Add the game logic (points, sounds, game over, etc...)

STEP 9: HOSTONSERVER //TODO

Step 1: Breaking the problem down

The first thing I we do when I come up with an idea, is spend some time figuring out how to break it into smaller pieces.

This way, it allows us to identify parts of the project We might already know how to build, where we need to do some extra research, identify the different tools we need to use based on the features, and finally, have a rough idea of the timeframe needed to build it.

For this particular project, we ended up with the following parts:

1) Get the pose detection working

2) Set up the 3D scene

3) Add 3D objects

4) Map the 2D hands movements to the 3D world

5) Creating the hand trail animation

6) Add collision detection between the hands and 3D objects

7) Add the game logic (points, sounds, game over, etc...)

8) Refactor

9) Deploy

Step 2: Picking the tools

Now that the project is broken down into independent chunks, we can start thinking about the tools we need to build it.

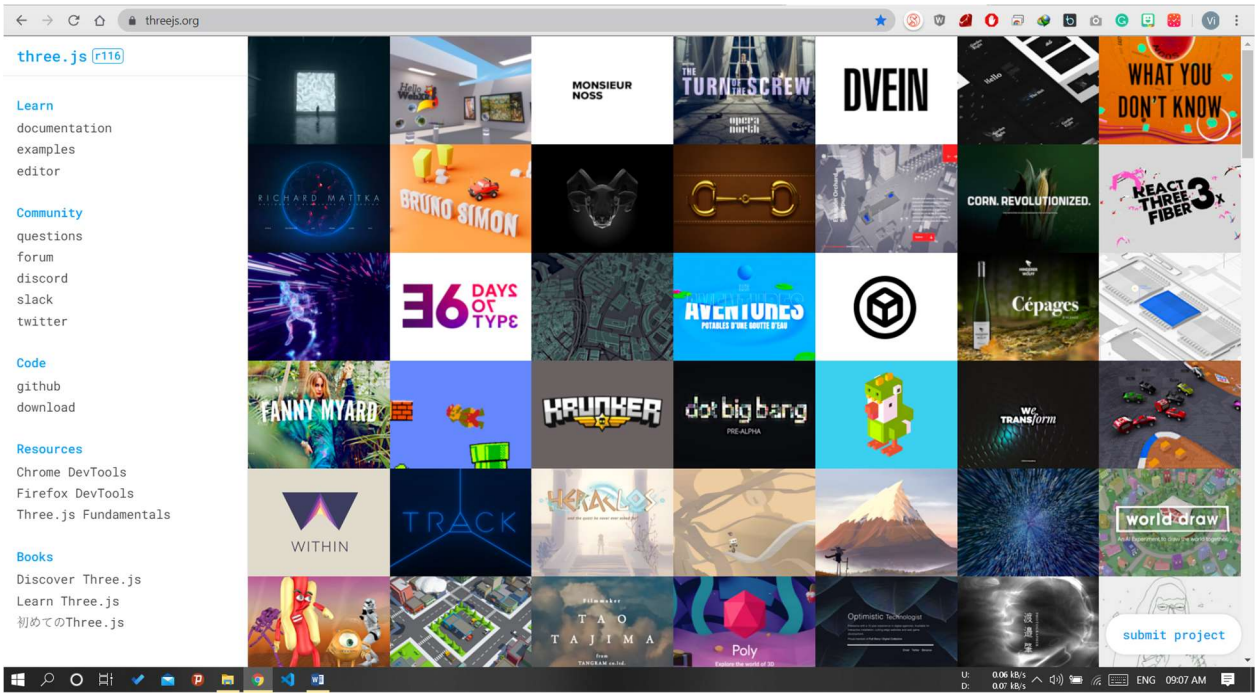
Pose detection

We knew we wanted to be able to detect hands and their position.

We learn **PoseNet library** during our ML course, not only did we know that it was a good tool to do this, but I also knew it wouldn't take me too long to implement it again.

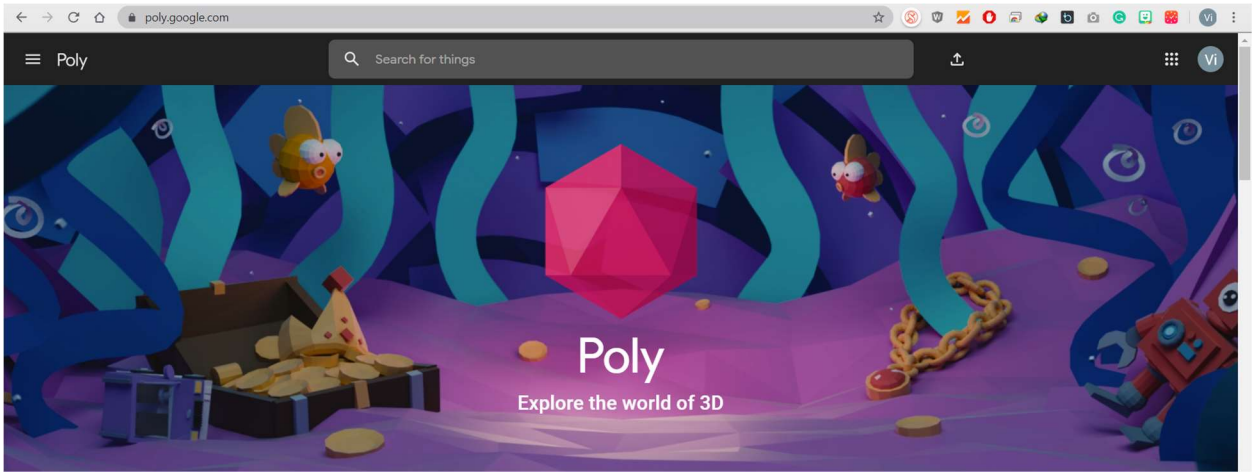
3D scene

We also knew we wanted to use 3D in the browser. One of the best libraries for this is the amazing ***Three.js***. We also research about other libraries. we decided to go with it instead of trying something like **Babylon.js** or **p5.js.**



3D objects

The goal of the game is to slice some fruits and avoid bombs, so we needed to load these 3D objects in the game. Even though we could have gone ahead and designed them myself in softwares like Blender, this would have taken a lot longer. Instead, we used Poly to search through assets created by other people and available to download.



Hand trails

I wanted to visualise where my hand was in the 3D scene. I could have done it by showing a simple cube but I wanted to try something a little different. I had never tried to create some kind of "trail" effect so I did some research and found a really cool little library called **TrailRendererJS** that lets you create a nice looking trail effect.

STEP 3: Learning the things

As we are new to most of the libraries we have to learn all of these:

1. Tenserflow
2. PoseNet
3. Three.js
4. Node.js(server)
5. 3D modeling
6. Converting 2D hands movements to the 3D world
7. TrailRendererJS
8. Web app development

STEP 4: Get the pose detection working

To use PoseNet, you need to start by adding the following scripts to your HTML, if you're not using it as an npm package:

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs"></script>
```

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow-models/posenet"></script>
```

Once you have access to the library, you need to load the model:

```
const loadPoseNet = async () => {
  net = await posenet.load({
    architecture: "MobileNetV1",
    outputStride: 16,
    inputResolution: 513,
    multiplier: 0.75,
  });
  video = await loadVideo();
  detectPoseInRealTime(video);
};
```

Here, we start by loading the **machine learning** model, then we initialise the video feed and once both of these steps have completed, we call the function responsible for detecting the body position in the webcam feed.

STEP 5: Add 3D objects

We added 3d object in our project. In 3d objects .obj consist structural geometrical data and .mtl files consist color codes.

We downloaded 3d models from Poly (google 3d obj) and make some modifications using blender.

STEP 6: Map the 2D hands movements to the 3D world

This part is most bug and most difficult part in all coding. Thanks to internet we are managed to build that. The complexity lies in the fact that the **2D coordinates** from **PoseNet** don't map directly to coordinates in the **Three.js** scene.

The coordinates **PoseNet** gives us are the same you would get if you were logging the position of the mouse in the browser window, so the value on the **x axis** would go from **0 to over 1280** for the width in pixels. However, coordinates in a 3D scene don't work the same way so you have to convert them.

To do this, we start by creating a vector from our hand coordinates.

```
const handVector = new THREE.Vector3();
// the x coordinates seem to be flipped so i'm subtracting them from window innerWidth
handVector.x =
    ((window.innerWidth - hand.coordinates.x) / window.innerWidth) * 2 - 1;
handVector.y = -(hand.coordinates.y / window.innerHeight) * 2 + 1;
handVector.z = 0;
```

Then, we use the following bit of magic to map the coordinates to a 3D world and apply them to our hand mesh.

```
handVector.unproject(camera);
const cameraPosition = camera.position;
const dir = handVector.sub(cameraPosition).normalize();
const distance = -cameraPosition.z / dir.z;
const newPos = cameraPosition.clone().add(dir.multiplyScalar(distance));
```

```
hand.mesh.position.copy(newPos);
```

STEP 7: Add collision detection between the hands and 3D objects

This part is the other tricky one.

Only after the 2D coordinates have been mapped to 3D ones can we work on collision detection. From what we know, you cannot work on this collision detection directly from 2D coordinates to 3D objects.

The way we're doing this is by implementing what is called **Raycasting (**https://threejs.org/docs/#api/en/core/Raycaster**)**.

Raycasting is the creation of a ray casted from an origin vector (our hand mesh) in a certain direction. Using this ray, we can check if any object in our scene intersects it (collision).

The code to do this looks like this:

```
const handGeometry = hand.mesh.geometry;
const originPoint = hand.mesh.position.clone();
for (
  var vertexIndex = 0; vertexIndex < handGeometry.vertices.length;
  vertexIndex++
) {
  const localVertex = handGeometry.vertices[vertexIndex].clone();
  const globalVertex = localVertex.applyMatrix4(hand.mesh.matrix);
  const directionVector = globalVertex.sub(hand.mesh.position);

  const ray = new THREE.Raycaster(originPoint, directionVector.clone().normalize()
  );
  const collisionResults = ray.intersectObjects(fruitsObjects);
  if (collisionResults.length > 0) {
    if (collisionResults[0].distance < 200) { // This distance value is a little bit arbitrary.
      console.log("Collision with a fruit!! 👐");
```

```
    }
  }
}
```

STEP 8: Add the game logic (points, sounds, game over, etc...)

As Fruit Ninja is old game most of the logics we easily find on internet but there some challenges we face while making this are:

Positing 3d objects , collision of the axis of object and hand , adding sounds, optimizing ..