Wirtualna rzeczywistość i wizualizacja

Implementacja przykładu wizualizacji z użyciem oprogramowania WebXR.

Wprowadzenie

Celem projektu było zwizualizowanie modelu złotego smoka znanego z zajęć Renderingu w czasie rzeczywistym na określonym markerze. Po wygenerowaniu modelu na markerze powinien on być zakotwiczony — próby przemieszczenia lub obrócenia markera winny skutkować analogicznymi ruchami modelu. W implementacji skorzystaliśmy z biblioteki react-three/xr.

Implementacja

Jako podstawy aplikacji użyliśmy frameworka React.js w połączeniu z templatką TypeScript. Można ją szybko stworzyć za pomocą poniższego snippetu (należy mieć zainstalowane środowisko Node.js wraz z menedżerem pakietów npm):

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh
nvm install --lts
node --version
npm --version
npx create-react-app AGH-webxr-example --template typescript
```

Następnie skorzystaliśmy z gotowych przykładów użycia bibliotek react-three/xr dostępnych na publicznym repozytorium: https://github.com/pmndrs/xr/. Do budowania aplikacji użyliśmy frameworka Vite, jednego z najpopularniejszych narzędzi w aplikacjach frontendowych: https://vite.dev/.

W naszym przypadku potrzebowaliśmy przykładu wizualizującego wybrany model graficzny na ekranie urządzenia. Skorzystaliśmy ze snippetu zawartego w README głównego repozytorium react-three/xr:

```
const store = createXRStore()
export default function App() {
  const [red, setRed] = useState(false)
  return (
    <>
      <button onClick={() => store.enterAR()}>Enter AR</button>
      <Canvas>
        <XR store={store}>
          <mesh pointerEventsType={{ deny: 'grab' }} onClick={() => setRed(!red)}
position={[0, 1, -1]}>
            <boxGeometry />
            <meshBasicMaterial color={red ? 'red' : 'blue'} />
          </mesh>
        </XR>
      </Canvas>
    </>
  )}
```

Następnie dodaliśmy obsługę błędów na wypadek przeglądarek, które nie współpracują z narzędziami do AR/XR i wyświetlamy ewentualną wiadomość jako toast:

```
async function enterSession() {
  await store.enterAR().catch((reason) => {
    toast(`your browser does not support this application: ${reason}`);
  });}
```

Następnie przyszedł czas na renderowanie naszego modelu. Do repozytorium dodaliśmy pliki .obj, .mtl oraz teksturę dla smoka i odpowiednie renderowanie:

```
export interface DragonProps extends MeshProps {}
export default function Dragon(props: DragonProps) {
    const mtl = useLoader(MTLLoader, dragonMtl);
    const model = useLoader(OBJLoader, dragonModel, (loader) => {
        mtl.preload();
        loader.setMaterials(mtl);
    });
    const {nodes, materials} = useGraph(model);
    const texture = useTexture(dragonTexture);
    texture.wrapS = THREE.RepeatWrapping;
    texture.wrapT = THREE.RepeatWrapping;
    return ( <>
        <mesh geometry={(nodes.dragon as Mesh).geometry} {...props}>
            <meshPhongMaterial {...materials.Material} map={texture}
             specularMap={texture} lightMap={texture}/>
        </mesh>
    </>
   );}
```

Wstępnie wyświetlaliśmy smoka w statycznie ustawionym punkcie po starcie aplikacji:

```
return (
    <div style={{ width: "100%",height: "100%"}}>
      <button onClick={enterSession}>Enter AR</button>
      <Canvas>
        <React.Suspense fallback={<Loader/>}>
          <XR store={store}>
            <pointLight position={[5, 5, 5]} intensity={4}/>
            <ambientLight intensity={0.2}/>
            <Dragon position={[0,0, -5]} receiveShadow={true}/>
          </XR>
        </React.Suspense>
      </Canvas>
      <ToastContainer/>
    </div>
  )
```

Kolejnym krokiem było wykrycie odpowiedniego markera, aby zakotwiczyć na nim wygenerowany model, tak żeby ruch kamerą lub markerem nie spowodował pozostania modelu w miejscu. Skorzystaliśmy z oficjalnej dokumentacji bibliotek xr i zakładek poświęconych detekcji obiektów oraz zakotwiczaniu. Na początek próbowaliśmy wykryć ściany i pokolorować je na czerwono:

```
export default function RedWalls() {
    const wallPlanes = useXRPlanes('wall')
    return (
      <>
        {wallPlanes.map((plane) => (
          <XRSpace space={plane.planeSpace}>
            <XRPlaneModel plane={plane}>
              <meshBasicMaterial color="red" />
            </XRPlaneModel>
          </XRSpace> ))}
      </>
   )}
```

Próby połączenia detekcji z zakotwiczeniem obrazu spełzły na niczym, jednakże na stronie poświęconej różnym eksperymentom z AR udało nam się znaleźć podobny przykład, napisany niestety w czystym JavaScript: https://github.com/graemeniedermayer/ArExperiments/blob/main/javascript/webxrMarker.js

Za pomocą odpowiedniego nadpisywania metod i wrapperów z Reacta wgrywaliśmy nasz marker jako bitmapę oraz włączaliśmy wszystkie wymagane funkcjonalności w sesji XR:

```
const img = document.getElementById('bitmap') as HTMLImageElement;
const imgBitmap = await createImageBitmap(img, {});
const store = createXRStore({
  customSessionInit: {
    requiredFeatures: ["anchors", "image-tracking", "local-floor"],
    trackedImages: [{image: imgBitmap, widthInMeters: 0.2}]
  }
});
```

W kodzie generującym smoka dodaliśmy ustawianie przestrzeni dla modelu po wykryciu markera oraz zakotwiczanie go w tym miejscu:

```
export default function Dragon() {
  const [space, setSpace] = useState(null);
  const mtl = useLoader(MTLLoader, dragonMtl);
  const model = useLoader(OBJLoader, dragonModel, (loader) => {
    mtl.preload();
    loader.setMaterials(mtl);
  });

  const texture = useTexture(dragonTexture);
  texture.wrapS = THREE.RepeatWrapping;
  texture.wrapT = THREE.RepeatWrapping;
```

```
  useFrame((_state: RootState, _delta: number, frame?: _XRFrame) => {
    if (frame === undefined) { return; }
    const results = frame.getImageTrackingResults();
    for (const result of results) {
      setSpace(result.trackingState  === "tracked" ? result.imageSpace : null);
    }
  });

  return (
    <>
    {space && (
        <XRSpace space={space}>
        <mesh geometry={(model.children[0] as Mesh).geometry} scale={[0.1, 0.1, 0.1]}>
            <meshPhongMaterial map={texture} specularMap={texture} lightMap={texture}/>
        </mesh>
        </XRSpace>)}
    </>
  );
}
```

Wnioski

Najtrudniejszą częścią projektu okazało się nadpisanie domyślnej implementacji funkcji createXRStore z biblioteki Reacta oraz wykrywanie markera na podstawie przechwytywanych ramek. Problematyczne okazało się również to, że nie wszystkie urządzenia oraz przeglądarki mobilne obsługują narzędzia AR/XR. Koniec końców rezultat okazał się satysfakcjonujący z zastrzeżeniem, że marker musi cały czas być wykrywany przez aplikację, aby efekt zakotwiczenia modelu na nim był widoczny.

Tobiasz Szulc
Michał Wójcik